Cactus Representation of Minimum Cuts: Derandomize and Speed up

Zhongtian He
Princeton University

Shang-En Huang[*]
Boston College

Thatchaphol Saranurak[†]
University of Michigan

Abstract

Given an undirected weighted graph with n vertices and m edges, we give the first deterministic $m^{1+o(1)}$-time algorithm for constructing the cactus representation of *all* global minimum cuts. This improves the current $n^{2+o(1)}$-time state-of-the-art deterministic algorithm, which can be obtained by combining ideas implicitly from three papers [Kar00, Li21, Gab16]. The known explicitly stated deterministic algorithm has a runtime of $\tilde{O}(mn)$ [Fle99, NNI00]. Using our technique, we can even speed up the fastest randomized algorithm of [KP09] whose running time is at least $\Omega(m \log^4 n)$ to $O(m \log^3 n)$.

Contents

1 Introduction

The global minimum cut problem has been studied for decades. For an undirected, weighted graph $G = (V, E, w)$, the global mincut of G is a minimum weight subset of edges that disconnecting the graph by removing them. Lots of beautiful works on this problem appeared in the last century [GH61, HO92, NI92a, NI92b, SW97], and then a series of work utilizing randomization [Kar93, KS96] finally led to an near-linear time *Monte Carlo* algorithm by Karger [Kar00] in 1996.

It turns out that there is a cactus representation of *all* (possibly $\Theta(n^2)$) minimum cuts using an $O(n)$-edge cactus graph introduced by [DKL76] (see also [FF09]). The cactus representation of global mincuts has found several algorithmic applications: it is a key subroutine for several edge connectivity augmentation algorithms [Gab91, NGM97, CLP22a, RZZ23] and also in several dynamic mincut algorithms [Hen95, GHT18]. Many algorithms were designed to find the cactus representation of minimum cuts. [KT86] outlined the first algorithm for constructing the cactus that takes $\Theta(n^3)$ time. Their algorithm was parallelized by [NV91] and refined by [NK94]. Later on, faster algorithms were developed by [Gab91, KS96, NNI00] and [Fle99] where the latter two algorithms running in $\tilde{O}(mn)$ are the fastest explicitly stated deterministic algorithms for computing cactus representation. Finally, [KP09] showed that the cactus representation problem is near-linear time computable by randomized algorithms as well.

All these near-linear time algorithms mentioned so far have one drawback: they are *Monte Carlo* meaning that they can err. It was a big open problem if there are near-linear (or almost-linear[1]) time deterministic algorithms for computing global mincuts and even computing cactus. After a series of works [KT18, HRW20, Sar21] and [LP20], Li recently showed an $m^{1+o(1)}$-time deterministic algorithm for computing a global minimum cut [Li21], by derandomizing the construction of the skeleton graph which is the single randomized procedure in Karger's near-linear time mincut algorithm [Kar00].

A natural question is that, given that we can deterministically compute a global mincut, whether we can also compute a cactus representation for all global mincuts as well. It turns out that Li's deterministic skeleton graph construction [Li21] applied to Karger and Panigrahi's algorithm [KP09] is not sufficient. Instead, one may replace the procedure of [KP09] by Gabow's algorithm [Gab16] together with Karger's dynamic programming technique [Kar00], achieving a deterministic $n^{2+o(1)}$-time. Recently, Kawarabayashi and Thorup [KT18] and Lo, Schmidt, and Thorup [LST20] successfully showed how to construct cactus deterministically in near-linear time, *assuming* simple graphs. Nonetheless, the general problem of whether such an algorithm exists for general weighted graphs is still open.

In this work, we improve the quadratic $n^{2+o(1)}$-time barrier by showing the first almost-linear deterministic algorithm for computing the cactus representation of minimum cuts for undirected weighted graph, which positively answers the open question raised by [KP09][2]. Furthermore, using our technique, we also speed up the best previous randomized algorithm [KP09] by a logrithmic factor.

---

[1]As a convention from the literature [Kar00, Li21], we say a function that is $\tilde{O}(m)$ to be *near-linear* and $m^{1+o(1)}$ to be *almost-linear*.

[2]Actually, they asked if there exists an efficient algorithm to (deterministically) compute a *certificate* of a cactus representation, which turns their Monte Carlo algorithm into a Las Vegas algorithm.

Theorem 1.1. *There are algorithms for computing cactus representation of all (global) minimum cuts in an undirected weighted graph with the following guarantees*

- *Deterministic algorithm in $m^{1+o(1)}$ time.*
- *Randomized Monte-Carlo algorithm in $O(m \log^3 n)$ time.*

This is the first almost-linear time deterministic algorithm for computing cactus. Previously, the fastest deterministic algorithm takes $\Omega(mn)$ by [Gab16, Fle99, NNI00]. The $m^{o(1)}$ factor in our running time is solely from the overhead of $n^{o(1)}$ in the deterministic tree packing algorithm by Li [Li21]. If this factor was improved to $O(\text{polylog}(n))$, our running time would be $\tilde{O}(m)$ too.

By plugging a faster randomized tree packing [Kar00] into our new approach, we obtain a randomized algorithm that is even faster than the fastest known algorithm by [KP09] which takes time at least $\Omega(m \log^4 n)$. We discuss this in more details in Section 3.4.

New Development. After we have published the manuscript, a new result by Henzinger, Li, Rao, and Wang [HLRW24] shows how to deterministically construct in $O(m \, \text{polylog}(n))$ time a collection of $\text{polylog}(n)$ trees that every minimum cut 2-respects one of the trees. By our reduction, this immediately that we can deterministically compute a cactus in $O(m \, \text{polylog}(n))$ time.

Application. Cactus construction has an immediate application to the $+1$-*edge-connectivity augmentation problem* defined as follows: given an undirected integer-weighted graph $G = (V, E, w)$ where $w : E \to \mathbb{N}$, compute an edge set E' of the minimum size such that the minimum cut on $G' = (V, E \cup E')$ has value $\lambda(G) + 1$, where $\lambda(G)$ is the value of the minimum cut of G. To solve this problem, one can simply compute a cactus representation of G and then apply in $O(n)$ time a DFS traversal algorithm from Naor, Gusfield, and Martel on top of the cactus [NGM97, Section 3]. Therefore, Theorem 1.1 gives immediately the first $m^{1+o(1)}$-time deterministic algorithm and a $O(m \log^3 n)$-time randomized algorithms for this problem, improving the previous bound of $\Omega(m \log^4 n)$ implied by [KP09].[3]

1.1 Related Works

Deterministic Algorithms. For decades, significant effort has been devoted to devising deterministic algorithms that is as fast as their randomized counterparts. Examples include the line of work on deterministic minimum edge cut algorithms [KT18, HRW20, Sar21, LP20, Li21], deterministic minimum vertex cut algorithms [Gab06, SY23], and deterministic Laplacian solvers and approximate max flow [CGL+20]. Each of these deterministic algorithms usually deepens insight on the problems. Our result extends this line of research and reveals deeper structural understanding on 2-respecting mincuts (defined later in Section 3.1).

---

[3]We remark that if G is an unweighted graph, the $+1$-edge-connectivity augmentation problem can be solved in $O(m \log^2 n (\log \log n)^2)$ time by [HRW20, LST20]. In addition, if the goal is to increase the edge connectivity to a particular quantity τ in a weighted graph, this variant can also be solved in $\tilde{O}(m)$ time by [CLP22b].

Faster Randomized Global Mincut Algorithms. Recently, [GMW20] and [MN20] gave faster randomized algorithms for computing a *single* global minimum cut in an undirected weighted graph. Their algorithms run in $O(m \log^2 n)$ and $O(m \log^2 n + n \log^6 n)$ respectively, which improved Karger's long-standing $O(m \log^3 n)$ time algorithm. But their approaches are difficult to generalize to find all global mincuts. Our algorithm leaves an $O(\log n)$ gap between finding one mincut with all mincuts in randomized setting, which is left for future work.

2 Preliminaries

Let $G = (V, E, w)$ be an undirected weighted graph, with n vertices and m nonnegatively weighted edges. A cut $(X, V \setminus X)$ (or the cut *induced by* X, or simply denoted by X if the context is clear) is a proper partition of V, and the edges across the cut are called *cut edges*. The *weight* of the cut is defined to be the sum of all edge weights across the cut, denoted by $\mathcal{C}(X) := \mathcal{C}(X, V \setminus X)$. By extending the above notation we define for any two (not necessarily disjoint) subsets X and Y, let $\mathcal{C}(X, Y)$ be the sum of weights of edges with one endpoint being in X and another endpoint being in Y. Notice that an edge with both endpoints in $X \cap Y$ will be counted twice.

Global Mincuts. The *global minimum cut* or simply *mincut* is a cut whose weight is the smallest among all cuts. Throughout the paper we use λ to denote the weight of any global minimum cut. We also assume that the value λ is already precomputed [Li21, Kar00, GMW20, MN20].

Minimal Mincuts. Since a cut separates some vertices from any given vertex, throughout the paper, we will designate an arbitrary but fixed *root vertex* $r \in V$. After fixing the root, we are able to characterize the mincuts that separate a vertex or an edge from the root. The *size* of a cut $(X, V \setminus X)$ where $r \notin X$ is then defined to be the number of vertices in X. Intuitively, for each vertex v (or an edge $e = (u, v)$), the most relevant global mincut would be the one that minimizes the number of the vertices "on the v (or e) side" of the cut. Thus, we have the following definition:

Definition 2.1 (Minimal mincuts). The *minimal mincut* of a vertex v is the mincut of the least size separating v from r. If v is not separated from r by any mincut, then its minimal mincut is null. The minimal mincut of an edge (u, v) is defined similarly except that the mincut must separate both u and v from r.

2.1 Crossing Mincuts, Uniqueness of Minimal Mincuts

The most important property of minimum cuts along the history should be the *submodularity* of crossing cuts. Two cuts $(X, V \setminus X)$ and $(Y, V \setminus Y)$ are said to be *crossing* if each of $X \cap Y, X \setminus Y, Y \setminus X, (V \setminus X) \cap (V \setminus Y)$ is non-empty. With submodularity one can show that:

Lemma 2.2 ([DKL76]). *If $(X, V \setminus X)$ and $(Y, V \setminus Y)$ are crossing mincuts, then each of the cuts induced by $X \cap Y$, $X \setminus Y$, $Y \setminus X$, and $X \cup Y$ are also mincuts. Furthermore, we have $\mathcal{C}(X \cap Y, (V \setminus X) \cap (V \setminus Y)) = \mathcal{C}(X \setminus Y, Y \setminus X) = 0$.*

With the crossing property above, we are able to deduce the uniqueness of the minimal mincut of a vertex (or an edge). This uniqueness property plays an important role in many applications (closest mincut, left-most mincuts, ...etc) as well as constructing a cactus representation of mincuts.

Lemma 2.3 ([KP09]). *If a minimal mincut of a vertex or edge exists, then it is unique.*

2.2   Cactus: Representation of all Global Mincuts

Dinitz et al [DKL76] showed that there exists a cactus graph H (every edge belongs to at most one cycle) with $O(n)$ edges that represents all global mincuts on G. The representation has the following properties. Each vertex of G is mapped to a node on H. (This mapping could be neither surjective or injective.) For every edge in H or two edges in the same cycle of H that split all nodes in H into two parts, the corresponding partition of vertices in G forms a global mincut. Conversely, for any global mincut in G one can also find a corresponding edge or a pair of edges in H that represents this mincut.

Although the size of the cactus representation is $O(n)$, it is highly nontrivial to compute such a representation from G. Karger and Panigrahi [KP09] gave the first randomized Monte Carlo algorithm that computes a cactus representation in $\tilde{O}(m)$ time. Since there can be $\Omega(n^2)$ many mincuts (in a cycle, for example), we note that this cactus has to be computed without explicitly listing all mincuts.

3   The Framework of [KP09] and Our Improvement

Since our improvement is mainly based on the framework of Karger and Panigrahi [KP09], in this section we introduce the framework and describe our contribution in details.

3.1   2-Respecting Mincuts and Tree Packing

Similar to most of the fastest exact minimum cut algorithms [Kar00, KP09, GMW20, MN20], our algorithm is based on computing global minimum cuts that 2-respect a spanning tree. Let T be any spanning tree on G. A cut is said to be *k-respecting* T if the spanning tree T contains at most k cut edges. A cut is said to *strictly k-respect* a spanning tree of a graph if the spanning tree contains exactly k cut edges. Karger [Kar00] first showed that there exists a collection $\mathcal{T}$ of $O(\log n)$ spanning trees in G such that every global mincut 2-respects some spanning tree in $\mathcal{T}$. Such collection $\mathcal{T}$ is also called a *tree packing*. In the same paper Karger also gave a randomized Monte Carlo algorithm that in $O(m + n \log^3 n)$ time computes a tree packing with high probability.[4] Recently, Li [Li21] gave the first *deterministic* algorithm that computes a tree packing of size $n^{o(1)}$ in $m^{1+o(1)}$ time.

Once a tree packing $\mathcal{T}$ is found, the task of searching for a global minimum cut can be reduced to checking all global minimum cuts that 2-respect a spanning tree $T \in \mathcal{T}$. We summarize these useful algorithms computing tree packings in Theorem 3.1.

---

[4]Later on, Gawrychowski, Mozes and Weimann [GMW20] give another time bound $O(m \log^2 n)$, which is faster on sparse graphs, but we do not exploit this new bound in our paper.

Theorem 3.1. *Given an undirected weighted graph G, there are algorithms that compute a tree packing $\mathcal{T}$ consisting of*

- $n^{o(1)}$ *spanning trees by a deterministic algorithm in $m^{1+o(1)}$ time, or*
- $O(\log n)$ *spanning trees by a randomized Monte Carlo algorithm[5] in $O(m + n \log^3 n)$ time,*

such that each global mincut 2-respects some tree in $\mathcal{T}$.

Karger and Panigrahi's algorithm [KP09] reduces the cactus construction problem to computing the minimal mincuts of all the vertices and edges. Besides designing an efficient algorithm which computes these minimal mincuts, it is also important to store these minimal mincuts in a succinct way. Using Theorem 3.1, each minimal mincut 2-respects some tree in $\mathcal{T}$. As long as there is a way to represent a 2-respecting mincut on a tree using $O(\log n)$ bits, each minimal mincut can be represented efficiently in a total of $O(m \log n)$ bits. We refer to these representations as *cut labels*.

3.2 Cut Labels and Three Types of 2-Repecting Mincuts

Fix a spanning tree $T \in \mathcal{T}$ with root r. The set of descendants of a vertex v is in a spanning tree T is denoted by $v_T^\downarrow$, and the set of strict descendants of v is denoted by $v_T^\Downarrow$, i.e. $v_T^\Downarrow = v_T^\downarrow \setminus \{v\}$. Similarly, the set of ancestors of v in T is denoted by $v_T^\uparrow$, and we define $v_T^\Uparrow = v_T^\uparrow \setminus \{v\}$. We may drop the subscript and simply denote the sets by $v^\downarrow$, $v^\Downarrow$, $v^\uparrow$, and $v^\Uparrow$ if there is no confusion.

Consider any 2-respecting cut on T. This cut must intersect with at most two edges on T. In particular, this cut can be either 1-respecting T or strictly 2-respecting T. In the case where there are two edges across the cut on T, these two edges may or may not be on the same path from the root r to some vertex on T. Hence, we can classify any 2-respecting cut as one of the following three types:

Type 1. The cut 1-respects T. In this case there exists a vertex v so that $v_T^\downarrow$ induces the cut.

Type 2-Comparable. The cut strictly 2-respects T, and the two tree edges across the cut belong to the same path from the root r. Let v and w be the lower endpoints to the tree edges across the cut. Then, we must have $v \in w^\downarrow$ or $w \in v^\downarrow$ and we say that v and w are *comparable* (denoted by $v \parallel w$). Moreover, without loss of generality let $w \in v^\downarrow$, then this cut must be induced by $v_T^\downarrow \setminus w_T^\downarrow$. In this case we say that w is the *lower vertex* and v is the *upper vertex*.

Type 2-Incomparable. The cut strictly 2-respects T, and the two tree edges across the cut belong to different paths from r. Let v and w be the lower endpoints to the tree edges across the cut. Then we must have $v \notin w^\downarrow$ and $w \notin v^\downarrow$. In this case we say that v and w are *incomparable* (denoted by $v \perp w$). Again, this cut must be induced by $v_T^\downarrow \cup w_T^\downarrow$.

With the above classification of 2-respecting mincuts, one immediately sees that any minimal mincut of a vertex (or an edge) can be represented by an $O(\log n)$-bit cut label of the form $(type, v, w, T)$. Once we obtain cut labels of minimal mincuts for all vertices and all edges, a cactus representation can be constructed efficiently:

---

[5]success with high probability $1 - n^{-\Theta(1)}$.

Lemma 3.2. *Given a graph $G = (V, E)$, a tree packing $\mathcal{T}$ and the set of cut labels representing minimal mincuts of each vertex $v \in V$ and each edge $e \in E$, there exists a deterministic algorithm that computes a cactus representation in $O(m\alpha(m, n) + n|\mathcal{T}|)$ time, where $\alpha(m, n)$ is the inverse Ackermann function.*

In [KP09], a similar statement was given. However, only an imprecise $\tilde{O}(m)$ time bound was given and some description of the algorithms and argument of the proofs were omitted (e.g., the proof to Lemma 3.12 in [KP09] and the cases analysis on their last page). In this paper, we give a formal detailed proof of Lemma 3.2. Since the technical contribution is mainly to complete (and simplify) the argument of [KP09] by incorporating formal concepts from [Gab16], we defer the proof to Appendix A.

3.3 Computing Cut Labels Efficiently

With the help of Theorem 3.1 and Lemma 3.2, the task of computing a cactus representation reduces to computing cut labels of the minimal mincuts for each vertex and each edge. Since the number of trees in the tree packing $\mathcal{T}$ is small, it suffices to compute a *minimal 2-respecting mincut candidate* on each tree T for each vertex and each edge. That is, whenever the minimal mincut of an edge or a vertex 2-respects T, the returned candidate must be this mincut.

Notice that for a particular vertex v (or an edge e), the candidate may not exist in every tree. Now, if we are able to compute minimal 2-respecting mincut candidates in almost-linear time, we are able to compute the cactus representation in almost-linear time as well:

Lemma 3.3. *Suppose there is a deterministic algorithm that, given a spanning tree T, computes a label for every vertex v and edge e representing its minimal 2-respects mincut candidate in t_{tree} total time. Then, to compute the cactus representation of a graph, there is a deterministic algorithm that runs in $t_{tree} \cdot n^{o(1)} + m^{1+o(1)}$ time, and there is also a randomized Monte Carlo algorithm that runs in $O(t_{tree} \log n + m \log^2 n)$ time.*

Proof. We first invoke Theorem 3.1 and obtain a tree packing $\mathcal{T}$. Then, for each spanning tree $T \in \mathcal{T}$, we compute a cut label for every vertex v and edge e representing its minimal 2-respects mincut. Since every mincut 2-respects one of the tree from $\mathcal{T}$, one of the cut label of each vertex v and edge e corresponds to its minimal mincut (and we can obtain it by comparing the weight and size in $O(m)$ total time). In total, this takes either $t_{tree} \cdot n^{o(1)} + m^{1+o(1)}$ time deterministically or $O(t_{tree} \log n + m \log^2 n)$ time Monte-Carlo randomized.

Given labels representing minimal mincuts of all vertices and edges, we obtain a cactus representation in $O(m \log^2 n)$ additional time by Lemma 3.2. □

Computing Minimal 2-Respecting Mincut Candidates. The remaining task is to compute cut labels of the minimal 2-respecting mincut candidates on every tree $T \in \mathcal{T}$, for each vertex and each edge. Karger and Panigrahi [KP09] provided a deterministic algorithm (see Lemma 3.4) for computing these cut labels for vertices — they partially bypassed the challenge for computing cut labels of the edges via randomization. However, there is a missing case in [KP09] when computing cut labels for vertices.

6

Regarding the gap in [KP09], we believe that the approach is not wrong, but the fix seems to require more than changing some typos. The following Lemmas 3.4 and 3.5 summarizes the tasks for obtaining cut labels to minimal 2-respecting mincut candidates for vertices and edges on a given tree T.

Lemma 3.4 ([KP09])**.** *Given a spanning tree T, we can deterministically compute, for each vertex v, a cut label representing a minimal 2-respecting mincut candidate of v in $O(m \log^2 n)$ total time.*

In this paper, we give a simpler and complete algorithm of Lemma 3.4 in Appendix B.

Lemma 3.5 (Key Lemma)**.** *Given a spanning tree T, we can deterministically compute, for each edge e, a cut label representing a minimal 2-respecting mincut candidate of e in $O(m \log^2 n)$ total time.*

The algorithm and the proof to Lemma 3.4 will be in Appendix B. The rest of our paper is devoted to proving Lemma 3.5. By plugging Lemmas 3.4 and 3.5 into Lemma 3.3, we can conclude Theorem 1.1.

Proof of Theorem 1.1. By Lemmas 3.4 and 3.5, we have that $t_{tree} = O(m \log^2 n)$. Thus, by Lemma 3.3 we obtain a deterministic algorithm computing a cactus respresentation in $m^{1+o(1)}$ time, and also a randomized Monte Carlo algorithm that runs in $O(m \log^3 n)$ time. □

Toward the Proof of Lemma 3.5. We show a path to prove our key lemma. The goal is to find a minimal 2-respecting mincut candidate for each edge e on a given tree T. Since there are three types of cuts that 2-respects a tree T, it is natural to split the task into three subproblems, with each of them focusing on Type 1 cuts, Type 2-Comparable cuts, and Type 2-Incomparable cuts respectively.

Minimal 1-respecting mincut candidate for vertices and edges Computing the minimal 1-respecting mincut candidate for all vertices and edges is relatively simple, and can be derived from [Kar00]. For the sake of completeness we include the proof below.

The following basic lemma from [Kar00] can be implemented by dynamic program.

Lemma 3.6 (Lemma 5.1 in [Kar00])**.** *The values of all cuts that 1-respect a given spanning tree T can be determined in $O(m + n)$ time.*

In the second step, we show how to compute the minimal 1-respect mincut candidate for vertices.

Lemma 3.7. *Given a graph G and a spanning tree T, there is an algorithm such that, in $O(m+n)$ time computes the minimal 1-respecting mincut candidates for all vertices.*

Proof of Lemma 3.7. First, we invoke Lemma 3.6 to compute the value of all the 1-respecting cut. By comparing the value of each 1-respecting cut with the value of mincut λ, we can identify all the 1-respecting mincuts. If the 1-respecting cut $v^{\downarrow}$ seperating u from r, then u must in the subtree of v. Therefore, we can run a DFS and maintain the minimal 1-respecting mincut containing the current vertex u, which can be done in linear time. □

Finally, we get the minimal 1-respecting mincut candidate for all edges.

Lemma 3.8. *Given a graph G and a spanning tree T, there is an algorithm such that, in $O(m+n)$ time computes the minimal 1-respecting mincut candidates for all edges.*

Proof. Observe that a 1-respecting cut contains $e = (u_1, u_2)$ if and only if it contains $\text{LCA}_e = \text{LCA}(u_1, u_2)$. By Lemma 3.7, we compute the minimal 1-respecting mincut for all the vertices in $O(m+n)$ time. In addition, computing LCA_e for all the edges can be done in linear time. Therefore, we get the minimal 1-respecting mincut candidate for all the edges in $O(m+n)$ time. $\qquad\square$

Therefore, the remaining challenges are finding a strictly comparable 2-respecting mincut candidate (see Lemma 5.1) and a strictly incomparable 2-respecting mincut candidate (see Lemma 6.1) for each edge.

3.4 Technical Contribution

New Algorithm for Minimal 2-Respecting Mincuts. Lemma 3.5 is the key technical contribution. We give the first deterministic algorithm for computing minimal mincuts for edges in almost-linear time, which leads to the deterministic algorithm for computing cactus representation. With Lemma 3.5, we also obtain an $O(m \log^3 n)$ randomized algorithm for computing a cactus representation, while the previous fastest (randomized) algorithm by [KP09] requires $\Omega(m \log^4 n)$ runtime (see Appendix C.)

In fact, our algorithm is more modular than the algorithm by Karger and Panigrahi [KP09] in the following sense: their algorithm only computes minimal mincuts for only *some* random edges, but they show that this set of edges is sufficient. It requires more intricate proof to show that these mincuts suffice for constructing a correct cactus representation. The approach makes the overall framework less modular.

Structural Properties for 2-Respecting Mincuts. What enables us to achieve Lemma 3.5 are new structural lemmas about 2-respecting mincuts. 2-respecting mincuts are not esoteric objects. In fact, fast algorithms related to 2-respecting mincuts have been the only known pathway for obtaining global mincuts in general weighted graphs in near-optimal complexity in many models of computation (sequential [Kar00], parallel [GG21], distributed [DEMN21], streaming and cut queries [MN20]. We are hopeful that these structural lemmas are promising for these models too.

For example, when the minimal mincut of an edge e is a comparable 2-respecting cut, a non-trivial observation is that the task is reduced to computing only the *lower vertex* l_e of the two vertices that define the 2-respecting cut. This allows us to focus on just "half of the problem" and hence the algorithm can be greatly simplified. Structural insights of Lemma 5.9, Corollary 5.10, and Lemma 5.11 allow us to design a DFS procedure that obtains these lower vertices in *just one pass*. Lemma 6.8 plays a similarly important role when the minimal mincut of an edge e is an incomparable 2-respecting cut.

A Full Detailed Proof to Cactus Construction. We also provide a not only comprehensive but also simplified algorithm in Appendix A for constructing a cactus from the labels of minimal mincuts of vertices and edges. Some correctness proofs in the last section of Karger and Panigrahi's paper [KP09] were missing, and unfortunately there was no full version. By revisiting the work of Gabow's [Gab16] and Karger and Panigrahi's [KP09], we believe Appendix A helps the readers and the community understand Karger and Panigrahi's algorithm with a much higher confidence.

Other Technical Contributions. Besides the key technical contribution and the simplified cactus construction algorithm, we also introduce a new algorithm for computing minimal mincut of vertices, which is simpler and fixes a gap in [KP09]. This algorithm serves as an alternative proof to Lemma 3.4 and is described in Appendix B. Last but not least, we formalize the reduction to path using path decomposition in Lemma 4.1, which allows us to focus on paths. This reduction simplifies the description of the algorithm, makes the analysis more modular, and can become handy in other applications.

4 Useful Tools

Reduction to Paths via Path Decomposition. To facilitate our algorithm throughout the paper, we utilize the reduction that reduces a problem on a tree to several problems on a collection of paths. Similar techniques have been developed in order to solve problems related to minimum cuts that 2-respecting trees. Two specific ways of decomposing a tree into paths were used: *bough decomposition* [Kar00] and *heavy path decomposition* [GMW20, BLS20] (see also [MN20, GG21] for more discussions).

It turns out that all we need is a balanced property for any decomposition of a tree into a collection of paths. Let T be a tree rooted at r. An *oriented path* on T is a path P with the vertex closest to r being an endpoint. A *path decomposition* $\mathcal{P}$ of T is a collection of oriented paths on T so that each vertex of T belongs to exactly one path. We say that a path decomposition $\mathcal{P}$ is *balanced* if for any vertex v, the path from the root of the tree to v intersects with $O(\log n)$ paths in $\mathcal{P}$. Both bough decomposition and heavy path decomposition of a tree are balanced, and can be computed in linear time.

With the balanced property, it is straightforward to see that there will be only an $O(\log n)$ overhead if we are allowed to process each path $P \in \mathcal{P}$ with a runtime related to the size of the subtree rooted at the highest vertex of P (e.g., perform a DFS). Specifically, given a path P, we define $P^{\downarrow}$ to be the set of all vertices with at least one ancestor in P. Let $E(P^{\downarrow})$ be the set of edges incident to at least one vertex in $P^{\downarrow}$ and let $d(P^{\downarrow}) = \sum_{v \in P^{\downarrow}} \deg(v)$ be the *unweighted volume* of the subtree. Lemma 4.1 below describes how we will bound the total running time using a balanced path decomposition in this paper.

Lemma 4.1. *If there exists an algorithm that preprocess G and a spanning tree T in t_p time such that, for any path P in a balanced path decomposition $\mathcal{P}$ of T, and a specific function $g(e, P)$, computes $g(e, P)$ for all $e \in E(P^{\downarrow})$ in total time $O(d(P^{\downarrow}) \log n)$. Then, we can compute in $t_p + O(m \log^2 n)$ time $g(e, P)$ for all $e \in E$ and for all P where $e \in E(P^{\downarrow})$.*

Proof. By the property of balanced path decomposition, for each edge e, there are at most $O(\log n)$ paths P such that $e \in E(P^{\downarrow})$. Therefore, we have

$$\sum_{P \in \mathcal{P}} d(P^{\downarrow}) = O(m \log n) . \tag{1}$$

For each $P \in \mathcal{P}$, since the algorithm computes $g(e, P)$ values every $e \in E(P^{\downarrow})$ in $O(d(P^{\downarrow}))$ time. We can compute $g(e, P)$ for all $e \in E$ and all P where $e \in E(P^{\downarrow})$ in total time $O(\sum_{P \in \mathcal{P}} d(P^{\downarrow}) \log n) = O(m \log^2 n)$. Adding the preprocessing time t_p, the algorithm runs in $t_p + O(m \log^2 n)$ time. □

The usages of Lemma 4.1 in this paper are quite similar in the taste: suppose we would like to compute some information (e.g., a minimal incomparable 2-respecting mincut candidate) of an edge $g(e)$, and realizes that $g(e)$ can be computed efficiently from the set $\{g(e, P)\}$ where $e \in P^{\downarrow}$ (e.g., a minimal 2-respecting mincut candidate with one crossing edge on P). Then by applying Lemma 4.1 we can focus on computing $g(e, P)$ values for each specific path $P \in \mathcal{P}$. We apply Lemma 4.1 in many cases in Section 5, Section 6, and Appendix B.

The path decomposition $\mathcal{P}$ we use throughout in this paper will be assumed to be balanced.

Data Structures on Trees. The second tool that are extensively used are dynamic tree data structures (and top-tree data structures). These data structures maintain values associated with vertices, with the list of operations supported in Lemma 4.2.

Lemma 4.2. *There exists a data structure over a dynamic forest of n vertices, supporting the following operations in the worst case $O(\log n)$ time:*

- $\texttt{Link}(v, w)$: where v, w are in different trees, links these trees by adding the edge (v, w) to our dynamic forest.

- $\texttt{Cut}(e)$: remove edge e from our dynamic forest.

- $\texttt{AddPath}(u, x)$: add x to the value of every vertices on the path from u to the root.

- $\texttt{MinPath}^{\downarrow}(u)$: return argmin of the value of vertex on the path from u to the root, and break tie by finding the deepest one.

- $\texttt{MinPath}^{\uparrow}(u)$: the same as $\texttt{MinPath}^{\downarrow}(u)$, but break tie by finding the highest one.

All these operations can be supported with a dynamic tree [ST83]. Besides, we need the following operations, which can be implemented using top-tree [AHLT05].

- $\texttt{MinTree}^{\downarrow}(u)$: returns a vertex v with minimum value in the tree T that contains u, breaking tie by finding the one with the smallest subtree size $|v^{\downarrow}|$.

- $\texttt{MinTree}^{\uparrow}(u)$: the same as $\texttt{MinTree}^{\downarrow}(u)$, but break tie by finding the one with the largest subtree size $|v^{\downarrow}|$.

- $\texttt{MinNonPath}(v, w)$: where v, w are in the same tree T, return the vertex u with the minimum value such that $u \in T$ but u is not in the path between v and w.

Proof. The operations $\texttt{Link}(v, w)$, $\texttt{Cut}(e)$ and $\texttt{AddPath}(u, x)$ are basic primitives of dynamic tree [ST83]. To implement $\texttt{MinPath}^{\uparrow}(u)$ and $\texttt{MinPath}^{\downarrow}(u)$, we just need to show how to break tie, since finding the argmin along the path is also a primitive. WLOG we consider implementing $\texttt{MinPath}^{\uparrow}(u)$. In the preprocessing step, we add $-\epsilon|u^{\downarrow}|$ to the value of vertex u where $\epsilon \ll 1/n$. Then for the values of two vertices equals before, they will become difference since the two vertices are comparable for they are on the path from u to the root, and the value of the higher one will become smaller. By reversing the sign the same approach works for $\texttt{MinPath}^{\downarrow}(u)$.

Since $\texttt{MinTree}(u)$ is a primitive of top-tree[6], we can use the same approach as above to break tie. Finally $\texttt{MinNonPath}(v, w)$ can be implemented by the $\texttt{MaxNonPath}$ primitive of a top-tree, which appears in the proof of Theorem 4 in [AHLT05]. $\square$

5 Comparable 2-respecting Minimal Mincuts of Edges

In this section, we present the algorithm computing the minimal mincut of edge when it is a comparable 2-respecting mincut of T. Henceforth, for every edge $e \in E$ we call T the *right* tree for e if the minimal mincut of e is a comparable 2-respecting mincut of T. (In this case we also call e a *right edge* in T.) In particular, we can represent this minimal mincut using a vertex pair (u_e, l_e) on the tree, indicating that $u_e^{\downarrow} \setminus l_e^{\downarrow}$ is the comparable 2-respecting mincut we found for the edge e. For convenience, for any comparable 2-respecting mincut $w^{\downarrow} \setminus v^{\downarrow}$, we call v the *lower vertex* and w the *upper vertex* in the cut $w^{\downarrow} \setminus v^{\downarrow}$.

Lemma 5.1 summarizes the algorithm that computes such a vertex pair (u_e, l_e) for every edge e on any given spanning tree $T \in \mathcal{T}$ in $O(m \log^2 n)$ time. Notice that when we analyze the correctness of the algorithm on a spanning tree T, it suffices to focus on the edges where T is the right tree. In the case that T is *not* the right tree for an edge e, it could be that the returned vertex pair (u_e, l_e) be either some arbitrary mincut or it could be $(\texttt{null}, \texttt{null})$.

Lemma 5.1. *There is an algorithm that, given a graph $G = (V, E)$ and a spanning tree $T \in \mathcal{T}$, in total time $O(m \log^2 n)$ computes, for every edge $e \in E$, a vertex pair (u_e, l_e) where $u_e, l_e \in V \cup \{\texttt{null}\}$ with the following guarantee: if T is the right tree for e, then $u_e^{\downarrow} \setminus l_e^{\downarrow}$ is the minimal mincut of e.*

Our algorithm is divided into two main steps. In the first step the lower veritces l_e are computed. Then based on l_e, in the second step the algorithm finds their correponding upper vertices u_e.

It turns out that the second step becomes simpler once we have computed the lower vertices l_e for all edge e. This reduction is presented in Section 5.1. Surprisingly, there is a deterministic algorithm that guarantees to find lower vertices l_e for each right edge e efficiently. We present the most important structural property supplemented with the algorithm in Section 5.2.

5.1 Reduction to Computing Lower Vertices

Fix a spanning tree $T \in \mathcal{T}$ and suppose that we have already obtained all lower vertices $l_e \in V \cup \{\texttt{null}\}$ for each edge. Lemma 5.2 states that there exists an efficient algorithm that guarantees to find corresponding upper vertices u_e for all right edges on T.

---

[6]Theorem 4 in [AHLT05].

Lemma 5.2 (Reduction to lower vertices). *There is an algorithm that, given a graph $G = (V, E)$, a spanning tree T, and a lower vertex $l_e \in V \cup \{\mathsf{null}\}$ for every edge $e \in E$, in time $O(m \log^2 n)$ computes an upper vertex $u_e \in V \cup \{\mathsf{null}\}$ for every edge $e \in E$ with the following guarantee: If T is the right tree for e and l_e is the lower vertex of the minimal mincut of edge e, then u_e is the upper vertex of the minimal mincut.*

Intuitively, for any lower vertex l_e of $e = (u_1, u_2)$, if we obtain the list of *comparable partners* who form comparable 2-respecting mincuts with l_e, then the upper vertex we are looking for must be the lowest ancestor of $\mathrm{LCA}(l_e, \mathrm{LCA}(u_1, u_2))$ which appears in the list. However, obtaining the list is inefficient. Fortunately, based on the path decomposition, we can maintain all such candidates of upper vertices on-the-fly and answer all the queries using a dynamic data structure. Specifically, whenever the algorithm processes a vertex v on a path $P \in \mathcal{P}$, this data structure finds all upper vertices for all edges in $Q_v = \{e \in E \mid l_e = v\}$, which is summarized in Lemma 5.3. Hence, by applying a very similar path decomposition framework as in Lemma 4.1, we can compute all upper vertices efficiently, which is summarized in Lemma 5.4.

Lemma 5.3. *Let $\mathcal{P}$ be a path decomposition of T. We can preprocess the graph G and the spanning tree T in $O(m \log n)$ time so that, given any path $P \in \mathcal{P}$, we can compute the upper vertex u_e for Lemma 5.2 for every edge $e \in \bigcup_{v \in P} Q_v$ in $O(d(P^{\downarrow}) \log n + \sum_{v \in P} |Q_v| \log n)$ time.*

Lemma 5.4 (A variant of path decomposition). *Let g be a function of $e \in E$. If there exists an algorithm that preprocess G and a spanning tree T in t_p time such that, for any path P in a balanced path decomposition $\mathcal{P}$ of T, any partition of subset of $E' \subseteq E$ into $\{Q_v\}_{v \in V}$, computes $g(e)$ for all $e \in \bigcup_{v \in P} Q_v$ in total time $O((d(P^{\downarrow}) + \sum_{v \in P} |Q_v|) \log n)$, then we can compute in $t_p + O(m \log^2 n)$ time $g(e)$ for all $e \in E'$.*

Proof. For each $P \in \mathcal{P}$, the algorithm computes $g(e)$ values for all $e \in \bigcup_{v \in P} Q_v$ in total time $O((d(P^{\downarrow}) + \sum_{v \in P} |Q_v|) \log n)$. Since $\bigcup_{v \in V} Q_v = E'$ and $\bigcup_{P \in \mathcal{P}} P = V$, we can compute $g(e)$ for all $e \in E'$ by running the algorithm on every path $P \in \mathcal{P}$. Summing up over $P \in \mathcal{P}$, we can compute $g(e)$ for all $e \in E'$ in total time $O(\sum_{P \in \mathcal{P}} (d(P^{\downarrow}) + \sum_{v \in P} |Q_v|) \log n) = O(m \log^2 n + m \log n) = O(m \log^2 n)$ by Equation (1). Adding the preprocessing time t_p, the algorithm runs in $t_p + O(m \log^2 n)$ time. $\qquad \square$

By plugging in Lemma 5.3 to Lemma 5.4, we obtain Lemma 5.2.

Proof of Lemma 5.2. First, for each edge e with $l_e = \mathsf{null}$, we set $u_e = \mathsf{null}$. Then, we partition the set of the remaining edges into $\{Q_v\}_{v \in V}$ such that Q_v contains all the edges e with the lower vertex $l_e = v$. By Lemma 5.3, there exists an algorithm that preprocess G and tree T in $O(m \log n)$ time such that, given a path P in a path decomposition $\mathcal{P}$, compute u_e for every edge $e \in \bigcup_{v \in P} Q_v$ in total time $O((d(P^{\downarrow}) + \sum_{v \in P} |Q_v|) \log n)$. Since it satisfies the condition of Lemma 5.4, we can compute the upper vertex u_e for every edge in $O(m \log^2 n)$ time. $\qquad \square$

Now we describe an algorithm that achieves Lemma 5.3.

Proof of Lemma 5.3. Fix a spanning tree $T \in \mathcal{T}$. For any vertex $v \in V$ and its ancestor $w \in v^{\Uparrow}$, the weight of the comparable cut $w^{\downarrow} \setminus v^{\downarrow}$ is given by:

$$\mathcal{C}(w^{\downarrow} \setminus v^{\downarrow}) = \mathcal{C}(w^{\downarrow}) - \mathcal{C}(v^{\downarrow}) + 2(\mathcal{C}(v^{\downarrow}, w^{\downarrow}) - \mathcal{C}(v^{\downarrow}, v^{\downarrow})) . \tag{2}$$

Suppose v is the lower vertex of the minimal mincut of e. Then by factoring out the terms only related to v, it suffices to compute the following *comparable precut values* for all $w \in v^{\Uparrow}$.

Definition 5.5 (Comparable precut value). The *comparable precut value* of v at w, is defined by

$$\mathcal{C}_v^{\parallel}(w) := \mathcal{C}(w^{\downarrow}) + 2\mathcal{C}(v^{\downarrow}, w^{\downarrow}) .$$

Note that the value is only defined for $w \in v^{\Uparrow}$. We say that w is a *comparable partner* or just a partner of v if w is a minimizer of comparable precut value at v.

To see the high level idea, we first show how to compute the upper vertex u_e for every edge $e \in Q_v$ assuming that the comparable precut value of v at w has already been computed for every $w \in v^{\Uparrow}$. Suppose $e = (u_1, u_2) \in Q_v$ is a right edge on T. Let LCA_e denote $\mathrm{LCA}(u_1, u_2)$. The minimal mincut of e must contain the vertex $x_e = \mathrm{LCA}(v, \mathrm{LCA}_e)$ since the minimal mincut is a comparable 2-respecting mincut in T with the lower vertex $l_e = v$. As the minimal mincut $u_e^{\downarrow} \setminus l_e^{\downarrow}$ of edge e is the mincut satisfying the condition above with the minimal size, we have that u_e must be the lowest partner of v such that $u_e \in x_e^{\uparrow}$, which can be found using $\texttt{MinPath}^{\downarrow}(x_e)$ (see Figure 1). Below we show how to remove the assumption that comparable precut values of v at all $w \in v^{\Uparrow}$ have been precomputed.

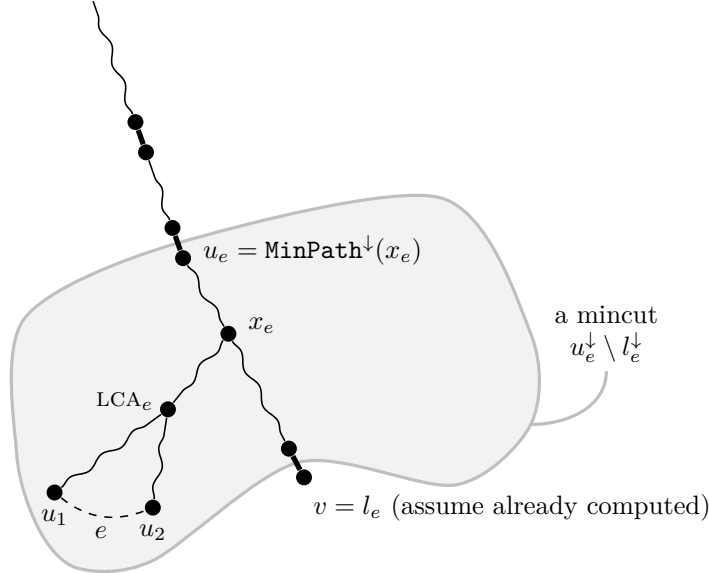Figure 1: High level idea: once the lower vertex of an edge l_e has found, we may locate the upper vertex u_e along the path via a $\texttt{MinPath}^{\downarrow}$ query.

Given a path $P = (v_1, v_2, \ldots, v_k)$ from the path decomposition with v_1 being the deepest vertex, our algorithm will process v_i starting from $i = 1, 2, \ldots, k$. We will maintain the invariant that once

13

we process the vertex v_i the values $\mathcal{C}^{\parallel}_{v_i}(w)$ for all $w \in v_i^{\Uparrow}$ can be accessed via $val[w]$.

Next we show how to maintain the invariant. In the preprocessing step before the path P was given, we set $val[w] = \mathcal{C}(w^{\downarrow})$ for each vertex w and create a dynamic tree on T (Lemma 4.2), which can be done in $O(m \log n)$ time.[7] Now we start from the deepest vertex v_1, the algorithm needs to add $2\mathcal{C}(v_1^{\downarrow}, w^{\downarrow})$ to each $val[w]$ so that $val[w] = \mathcal{C}^{\parallel}_{v_1}(w)$. For each edge (u, u') where $u \in v_1^{\downarrow}$, we invoke $\texttt{AddPath}(u', 2\mathcal{C}(u, u'))$ so that two times the weight of the edge (u, u') is added to $val[w]$ for each $w \in u'^{\uparrow}$. The total time to recover the invariant is $O(d(v_1^{\downarrow}) \log n)$.

After obtaining $\mathcal{C}^{\parallel}_{v_1}(w)$ values, for every $e \in Q_{v_1}$, we first compute $x_e = \text{LCA}(v_1, \text{LCA}_e)$ and then compute $\tilde{u}_e = \texttt{MinPath}^{\downarrow}(x_e)$. Finally, we check if $\tilde{u}_e$ is a partner of v_1. If so, then we know that the upper vertex $u_e = \tilde{u}_e$ because $\texttt{MinPath}^{\downarrow}(x_e)$ the lowest ancestor of x_e that is a partner of v_1 if there exists some partners. For running time, by Lemma 4.2, this requires $O(\log n)$ time for each $e \in Q_{v_1}$. So the total time for finding the upper vertices u_e such that their corresponding lower vertex is $l_e = v_1$ is $O(|Q_{v_1}| \log n)$ time. Therefore, the whole process on vertex v_1 can be done in $O(d(v_1^{\downarrow}) \log n + |Q_{v_1}| \log n)$ time.

Then, the algorithm scans through the rest of vertices $v_2, v_3, \cdots v_k$ on the path one by one. Suppose the algorithm reaches v_i now. The algorithm is similar to what we did at v_1. With the invariant after processing v_{i-1}, it suffices to add $2\mathcal{C}(v_i^{\downarrow} \setminus v_{i-1}^{\downarrow}, w)$ to $val[w]$ for each $w \in v_i^{\uparrow}$ by invoking $\texttt{AddPath}(u', 2\mathcal{C}(u, u'))$ for each edge (u, u') where $u \in v_i^{\downarrow} \setminus v_{i-1}^{\downarrow}$. (These edges can be found in $O(d(v_i^{\downarrow} \setminus v_{i-1}^{\downarrow}))$ time using a DFS from v_i without searching the subtree rooted at v_{i-1}.) Therefore, in $O(d(v_i^{\downarrow} \setminus v_{i-1}^{\downarrow}) \log n)$ time, $val[w]$ are updated to $\mathcal{C}^{\parallel}_{v_i}(w)$ for all $w \in v_i^{\Uparrow}$, and then the algorithm uses $\texttt{MinPath}^{\downarrow}(x_e)$ to find u_e for all $e \in Q_{v_i}$ in $O(|Q_{v_i}| \log n)$ time.

After finishing all the process on the path P, we need to roll back to the initial state $val[w] = \mathcal{C}(w^{\downarrow})$ in order to process other paths. Therefore, the algorithm computing u_e for every $e \in \bigcup_{v \in P} Q_v$ takes $O(d(P^{\downarrow}) \log n + \sum_{v \in P} |Q_v| \log n)$ time in total, which proves Lemma 5.3. $\qquad\square$

5.2 Computing the Lower Vertex for each Edge

In the rest of this section, we show how to compute the lower vertex for each edge. Fix a tree $T \in \mathcal{T}$. If the minimal mincut of edge e is a comparable 2-respecting mincut of T, then the algorithm will find the lower vertex l_e for e. Lemma 5.6 summarizes the algorithm:

Lemma 5.6. *There is an algorithm that, given a graph $G = (V, E)$ and a spanning tree T, in $O(m \log^2 n)$ time computes a vertex $\hat{l}_e$ for every edge e, such that if e is a right edge in T where the minimal mincut of e is $u_e^{\downarrow} \setminus l_e^{\downarrow}$, then $\hat{l}_e = l_e$.*

The algorithm described in Lemma 5.6 consists of two parts. In the first part the algorithm computes *the highest partner* $H(v)$ (defined below) for each vertex $v \in V$ as a preprocessing step. Then in the second part we apply a specialized depth first search that obtains l_e values for all $e \in E$.

---

[7]We first compute $\text{LCA}_e = \text{LCA}(u_1, u_2)$ for each edge $e = (u_1, u_2)$ in linear time [GT85]. Then it suffices for obtaining $val[w] = \mathcal{C}(w^{\downarrow})$ by invoking $\texttt{AddPath}(u_1, w(e))$, $\texttt{AddPath}(u_2, w(e))$, and $\texttt{AddPath}(\text{LCA}_e, -2w(e))$ for every edge e in $O(m \log n)$ total time.

5.2.1 Computing Highest Partner of each Vertex

Fix a spanning tree $T \in \mathcal{T}$. For each vertex v, we denote $H_T(v)$ the highest (comparable) partner of v. If there is no comparable 2-respecting mincut with lower vertex v, then $H_T(v) :=$ null. When there is no confusion, we shall drop the subscript T and simply denote it by $H(v)$. The goal for the algorithm is to compute $H(v)$ for all $v \in V$.

We will use the reduction to path from Lemma 4.1. For any $P \in \mathcal{P}$, define $g(e, P) = H(v)$ if $v \in P$ and e is the tree edge with v being the lower vertex, otherwise $g(e, P) =$ null.

Given a path $P = (v_1, v_2, \ldots, v_k) \in \mathcal{P}$ with v_1 being the deepest vertex, the algorithm computes the highest partner for each $v_i \in P$ as follows. Similar to the proof of Lemma 5.3, the algorithm processes the vertices in the order $v_1, v_2, \ldots, v_k$. A dynamic tree on T is used and $val[w]$ is maintained such that after processing v_i we obtain precut values $val[w] = \mathcal{C}_{v_i}^{\|}(w)$ for all $w \in v_i^{\uparrow}$. Then, using a dynamic tree query $\mathtt{MinPath}^{\uparrow}(v_i)$ the algorithm obtains a highest vertex w with the minimum precut value $\mathcal{C}_{v_i}^{\|}(w)$. Finally, we are able to assign $g(e, P) = w$ (where e is the tree edge with v_i being the lower vertex, i.e., $H(v_i) = w$) if the cut $w^{\downarrow} \setminus v^{\downarrow}$ is indeed a mincut. According to Equation (2), checking whether $\lambda = \mathcal{C}(w^{\downarrow} \setminus v^{\downarrow})$ can be done in constant time as long as the value $\mathcal{C}(v^{\downarrow}) + 2\mathcal{C}(v^{\downarrow}, v^{\downarrow})$ is precomputed.

From the discussion above, we have an algorithm that, given a path P, computes $g(e, P)$ for all $e \in E(P^{\downarrow})$ in $O(d(P^{\downarrow}) \log n)$ time. The preprocessing step is the same with the one in Lemma 5.3, which can be done in $O(m \log n)$ time. By plugging in the path decomposition Lemma 4.1, we obtain an algorithm that computes $H(v)$ of all vertices $v \in V$ in $O(m \log^2 n)$ total time.

5.2.2 Main Algorithm for Computing Lower Vertices

In this subsection we state the main algorithm for Lemma 5.6. Recall that the goal is, for every right edge e in T, we want to compute its lower vertex l_e. Recall that when we say that l_e is a lower vertex, we means that there exists some u_e where $u_e^{\downarrow} \setminus l_e^{\downarrow}$ forms a minimal mincut of some right edge e. For convenience, we denote $\overline{vw}$ as the set of vertices on the path between any two vertices v and w on T. For any right edge e, let $\hat{P}_e = \overline{u_e l_e}$ be its *canonical path*. For any $e = (u_1, u_2)$, let $\mathrm{LCA}_e = \mathrm{LCA}(u_1, u_2)$.

Motivation: high-level approach and the key structural lemma. At the highest level, the description of our algorithm is as follows. We will perform a post-order traversal on the tree T (i.e. if a is an ancestor of b then a is visited after b). When we visit u, we will be able to compute *some* lower vertices l_e of edges e where l_e is below u (i.e. $l_e \in u^{\Downarrow}$). At the end, we make sure that we have computed all lower vertices l_e of all the right edges e in T. To specify our algorithm in more details, we start with this definition.

Definition 5.7 (Valid lower vertices below u)**.** We call a vertex v a *valid lower vertex below u* if $v \in u^{\Downarrow}$ and $H(v) \in u^{\uparrow}$. Let L_u denote the set of all valid lower vertices below u.

In other words, $v \in L_u$ if there exists a comparable 2-respecting mincut containing u with the lower vertex v. Since $H(l_e)$ must be an ancestor of u_e, we have following:

Fact 5.8. *For every right edge e, $l_e \in L_u$ for any $u \in \hat{P}_e \setminus \{l_e\}$.*

Suppose that, magically, there is a data structure that, given a vertex u, can return the set L_u of all valid lower vertices below u. One idea would be that whenever the post-order traversal visits u, we query the data structure with u. Then, whenever $u \in \hat{P}_e \setminus \{l_e\}$, then l_e would be reported. However, there is an obvious issue in this approach: the total size of L_u over all u is simply too large to be reported quickly.

Therefore, we should consider a small subset of L_u that still contains l_e. Which subset of L_u satisfies this? Intuitively, since $u_e^{\downarrow} \setminus l_e^{\downarrow}$ is a minimal mincut, l_e should be "as high as possible" (and u_e should be "as low as possible"). This motivates the following definition: for any vertex set S, the set of *top vertices* of S, denoted by $\text{top}(S)$, contains all vertices $v \in S$ where there is no other $v' \in S \cap v^{\Uparrow}$ strictly above v. It makes sense to hope that $\text{top}(L_u) = \{l_e\}$ for any $u \in \hat{P}_e \setminus \{l_e\}$. This would be perfect because, not only that the output size is small, the data structure even identifies l_e for us. Unfortunately, this cannot be true. For example, let $x_e = \text{LCA}(l_e, \text{LCA}_e)$, for any $u \in \overline{l_e x_e} \setminus \{l_e, x_e\}$ strictly between l_e and x_e, $\text{top}(L_u)$ might not contain l_e because there might exist another mincut $u'^{\downarrow} \setminus l'^{\downarrow}$ such that l_e, l', u, u', x_e are ancestors of each other in this order and so $l_e \notin \text{top}(L_u)$ because of l'. Similarly, for any $u \in \overline{u_e x_e} \setminus \{u_e, x_e\}$ strictly between u_e and x_e, $\text{top}(L_u)$ might not contain l_e as well (see Figure 2). So we could only hope to guarantee that $\text{top}(L_u) = \{l_e\}$ when $u = x_e$.
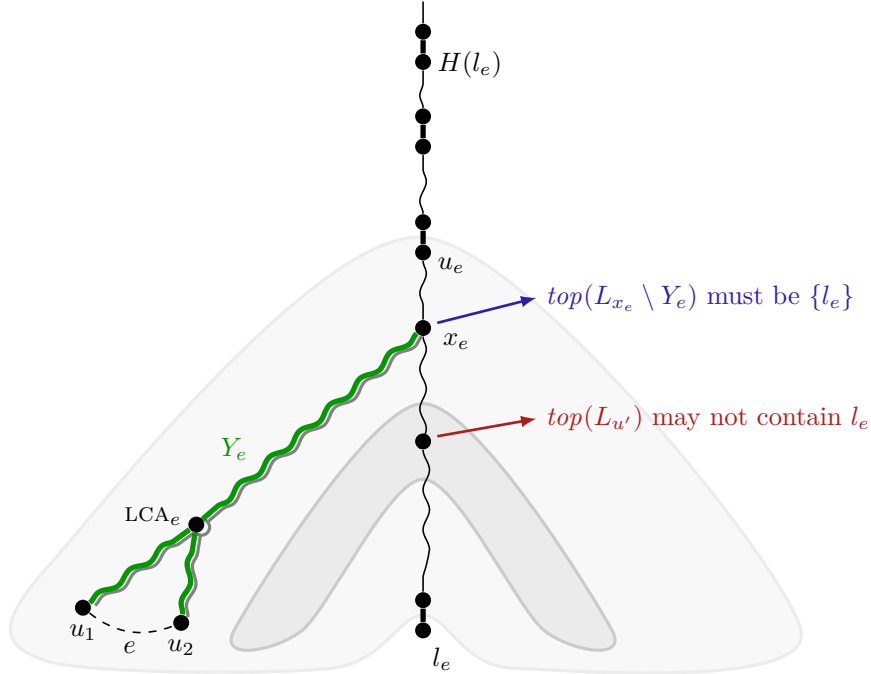
Figure 2: Querying x_e must find l_e. Querying at other vertices might not find l_e.

Surprisingly, something very close to the above wishful claim is actually true:

Lemma 5.9 (Pinpoint the lower vertex). *Suppose $e = (u_1, u_2)$ is a right edge whose minimal mincut is $u_e^{\downarrow} \setminus l_e^{\downarrow}$. For vertex $x_e = \text{LCA}(l_e, \text{LCA}_e)$, we have $\text{top}(L_{x_e} \setminus Y_e) = \{l_e\}$, where $Y_e := \overline{u_1 u_2} \cup \overline{x_e \text{LCA}_e}$ is the avoiding set of e.*

Before showing the proof, let us discuss the purpose of Lemma 5.9. It helps us pinpoint the lower vertex l_e because it says that, the lower vertex l_e is exactly the unique top vertex of $L_{x_e} \setminus Y_e$, the set of valid lower vertices below x_e excluding Y_e. Note that it is very natural to exclude Y_e because, for any right edge $e = (u_1, u_2)$, l_e cannot be on Y_e, otherwise $u_e^\downarrow \setminus l_e^\downarrow$ would not contain e.

Proof of Lemma 5.9. Suppose for contradiction that $\text{top}(L_{x_e} \setminus Y_e) \neq \{l_e\}$. First, observe that $\text{top}(L_{x_e} \setminus Y_e)$ is not empty since the set $L_{x_e} \setminus Y_e$ contains the vertex l_e. Now we arbitrarily select a vertex $l' \in \text{top}(L_{x_e} \setminus Y_e)$ different from l_e. By the definition of valid lower vertex below x_e, there exists $u' \in x_e^\uparrow$ that $u'^\downarrow \setminus l'^\downarrow$ forms a mincut. Since $l' \in L_{x_e} \setminus Y_e$ and $u' \in x_e^\uparrow$, the mincut $u'^\downarrow \setminus l'^\downarrow$ also contains edge e.

Next we consider the relative position of these two mincuts $u_e^\downarrow \setminus l_e^\downarrow$ and $u'^\downarrow \setminus l'^\downarrow$. Since l' is a top vertex of $L_{x_e} \setminus Y_e$, we have $l' \notin l_e^\downarrow$. So there are only two cases: $u'^\downarrow \setminus l'^\downarrow \subset u_e^\downarrow \setminus l_e^\downarrow$ or these two mincuts cross with each other. For the first case, $u'^\downarrow \setminus l'^\downarrow$ is a smaller mincut containing e, contradicts to minimality. For the second case, by the crossing property of mincuts (Lemma 2.2), the intersection of these two mincuts is a smaller mincut containing e, again contradicts to the minimality of $u_e^\downarrow \setminus l_e^\downarrow$. $\qquad\square$

Let $Y_{e,u} := \overline{u_1 u_2} \cup \overline{u\text{LCA}_e}$. Note that $Y_e = Y_{e,x_e}$. Recall that our plan is to perform a post-order traversal. By Lemma 5.9, whenever we arrives at $u = x_e$ and query $\text{top}(L_u \setminus Y_{e,u})$, the lower vertex l_e would be returned for us and we are done for e. But we do not know x_e. Then, from which vertex u should we query $\text{top}(L_u \setminus Y_{e,u})$ for finding l_e? Since we know that at least $x_e \in \text{LCA}_e^\uparrow$, we can query for finding l_e when $u \in \text{LCA}_e^\uparrow$. Now, the corollary below will be helpful because it says that for all $u \in \overline{x_e\text{LCA}_e}$, whenever we query for finding l_e, we will actually see nothing before the traversal actually reaches x_e.

Corollary 5.10. *For any vertex $u \in \overline{x_e\text{LCA}_e} \setminus \{x_e\}$, we have $\text{top}(L_u \setminus Y_{e,u}) = \emptyset$.*

Proof. Suppose for contradiction that there exists vertex $u \in \overline{x_e\text{LCA}_e} \setminus \{x_e\}$ such that $\text{top}(L_u \setminus Y_{e,u}) \neq \emptyset$. Since $l_e \notin u^\downarrow$, the set $\text{top}(L_u \setminus Y_{e,u})$ contains other vertex l' different from l_e. Since $\text{top}(L_{x_e} \setminus Y_e) = \{l_e\}$ by Lemma 5.9, $H(l') \in x_e^\downarrow$. But this implies that $H(l')^\downarrow \setminus l'^\downarrow$ is a mincut containing e with smaller size than $u_e^\downarrow \setminus l_e^\downarrow$, contradicts to minimality. $\qquad\square$

Equipped with this insight, now we are ready to move our attention on how to implement our high-level approach efficiently.

Implementation.   There are two main challenges in implementing the above high-level approach.

1. **(Efficiency of queries):** For any fixed vertex u, how can we return $\text{top}(L_u \setminus Y_{e,u})$ quickly given an edge e as a query? Furthermore, as we perform a post-order traversal, the vertex u is not fixed. We need a dynamic data structure where u can be updated too.

2. **(The number of queries):** For any fixed edge e, if we query for $\text{top}(L_u \setminus Y_{e,u})$ on all u or even just on all ancestors of LCA_e, then the total number of queries would be already super linear in m. We will exploit a structural lemma (Lemma 5.11) to reduce the number of queries.

In order to describe our algorithm and address how do we cope with both challenges, we first describe an algorithm that solves a simpler case when $x_e = \text{LCA}_e$. That is, if we have a data structure that supports the queries to $\text{top}(L_{x_e} \setminus Y_e)$ then all lower vertices will be found by Lemma 5.9. After we describe the algorithm that solves the simpler case, we generalize the algorithm and solve both challenges in the general case.

Simple Case: LCA_e is always on the path $\hat{P}_e$.

Let us assume here that, for every right edge e, LCA_e is on the path $\hat{P}_e$. Equivalently, $x_e = \text{LCA}_e$. Even with this assumption, we will need to deal with the first challenge above. We will remove this assumption in the next part.

We perform a post-order traversal on T. Suppose that u is the current vertex. There are two main tasks: (1) we will show how to maintain all the valid lower vertices below u, (2) we will show how to find the lower vertex l_e for every right edge e with $\text{LCA}_e = u$ via queries to the top-tree data structure.

To help solving the first task, we will exploit top-tree as follows. The top-tree we are maintaining is always a subgraph (forest) of T, and each vertex is associated with a value that satisfying the following invariant. Suppose the current vertex is u. For each vertex $v \in u^\downarrow$, if v is a valid lower vertex below u, the value of v in the top-tree should be $depth_v$ (the depth of v in T). Otherwise, if v is not a valid lower vertex below u, the value of v is ∞.

Now we solve the first task. The algorithm will maintain the invariant while running the post-order traversal. Initially, the top-tree is the same as the spanning tree with a super large value $\infty \gg n$ assigned to each vertex. When the traversal reaches u, the algorithm will do the following updates on the top-tree. First, the algorithm checks for each child v of u: if $H(v) \neq \text{null}$, then assign value $depth_v$ to vertex v, otherwise leave the value of v unchanged (which is ∞). Second, for any vertex $v \in u^\downarrow$ such that $H(v)$ is a child of u, the algorithm assigns ∞ to v. These vertices can be preprocessed once $H(v)$ is found. Furthermore, we can safely assign ∞ to v because that v will no longer be a valid lower vertex below u or below any vertex reached later in the post-order traversal. Since for each vertex v the value of v is changed at most twice in the algorithm, by Lemma 4.2 the top-tree can be maintained in $O(n \log n)$ total time.

Now we solve the second task. We show how to use the top-tree to find l_e for each edge $e = (u_1, u_2)$ where $\text{LCA}_e = u$ upon the reaching u in the post-order traversal. Recall that by Lemma 5.9, the lower vertex l_e is the unique vertex in $\text{top}(L_{x_e} \setminus Y_e) = \text{top}(L_u \setminus \overline{u_1 u_2})$. It implies that l_e is the unique vertex with smallest value among all vertices in $u^\downarrow \setminus \overline{u_1 u_2}$ stored in the top-tree. Therefore, we first apply $\text{Cut}(u, \text{parent}(u))$ to separate the subtree rooted at u. Then the lower vertex can be found by $l_e = \text{MinNonPath}(u_1, u_2)$. Finally, we apply $\text{Link}(u, \text{parent}(u))$ to restore the tree. See Figure 3 for an illustration.

General Case: LCA_e may not be on the path $\hat{P}_e$.

In the general case, we no longer have the assumption $\text{LCA}_e \in \hat{P}_e$. By Lemma 5.9 and Corollary 5.10, for each right edge e, it is natural to consider climbing up the tree from LCA_e toward x_e. The first time where the algorithm climbs up to a vertex u such that $\text{top}(L_u \setminus Y_{e,u}) \neq \emptyset$ implies that
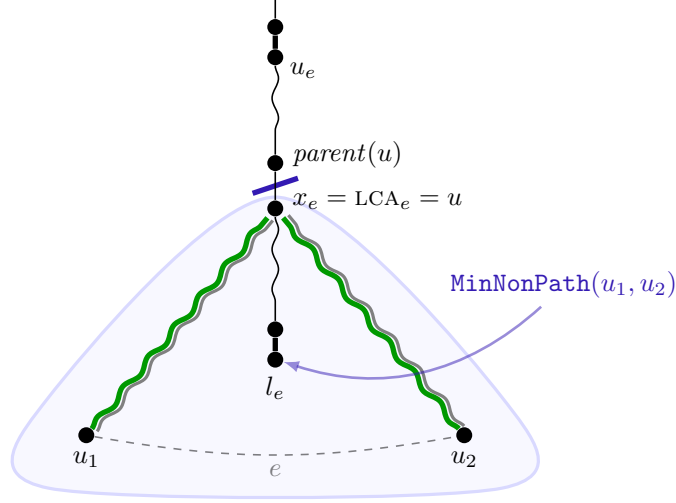
18

Figure 3: Simple Case: l_e can be found by `MinNonPath`(u_1, u_2) after separating the subtree rooted at $u = \text{LCA}_e$.

$u = x_e$. However, the time cost of performing such multiple queries per edge is unacceptable.

To deal with the challenge, we establish a key observation (Lemma 5.11) that leads to the following "packaging" idea. Initially every edge e is individually packed and is assigned to the vertex LCA_e. Upon reaching a vertex v in the post-order traversal, the data structure checks for each package whether or not a lower vertex can be assigned. If a lower vertex l is found, then all edges e in the same package get the same lower vertex $l_e = l$. Otherwise, all packages will be combined into one large package and sent to the parent of v. Our key observation states that, for all right edges e so that $v \in \hat{P}_e^{\Leftrightarrow} := \hat{P}_e^{\downarrow} \setminus (\hat{P}_e \cup l_e^{\downarrow})$, these right edges will be in the combined package and they all have the same minimal mincut. In particular, their lower vertices will be the same (see Figure 4).

Lemma 5.11. *Fix vertex v. For all right edges e_1, e_2 where their endpoints are in $v^{\downarrow}$ and $v \in \hat{P}_{e_1}^{\Leftrightarrow} \cap \hat{P}_{e_2}^{\Leftrightarrow}$ (i.e. $l_{e_1}, l_{e_2} \notin v^{\downarrow}$ and $u_{e_1}, u_{e_2} \in v^{\Uparrow}$), we have that $\hat{P}_{e_1} = \hat{P}_{e_2}$.*

Proof. First, we show that the minimal mincut of e_1 also contains e_2, and vice versa. Since $v \in \hat{P}_{e_1}^{\Leftrightarrow}$, the minimal mincut of e_1 contains the whole subtree $v^{\downarrow}$. Since both the endpoints of e_2 are in $v^{\downarrow}$, the minimal mincut of e_1 contains e_2. Symmetrically, the arguments also hold if we swap e_1 and e_2. By Lemma 2.3, the minimal mincut of e_1 and e_2 are the same and hence $\hat{P}_{e_1} = \hat{P}_{e_2}$. $\square$

Now we implement the algorithm in the general case via top-tree. When the traversal reaches vertex u, there are several packages at u. For the single edge packages that are directly created in preprocessing, the same procedure from simple case works: we cut off the subtree rooted at u and query `MinNonPath` for each of these edges. For any combined package that was delivered from a child v, since there is no valid lower vertex found in $v^{\downarrow}$, we claim that it suffices to check $\text{top}(L_u \setminus v^{\downarrow})$.

To see why, first observe that $l_e \notin L_v$ because l_e is never in $Y_{e,v}$ and we also know that $l_e \notin \text{top}(L_v \setminus Y_{e,v})$ (since e is forwarded from v). Now, suppose that $l_e \in \text{top}(L_u)$. We claim that l_e actually is in $\text{top}(L_u \setminus v^{\downarrow})$. This is because if $l_e \in v^{\downarrow}$, then $l_e \in L_v$. But we already concluded
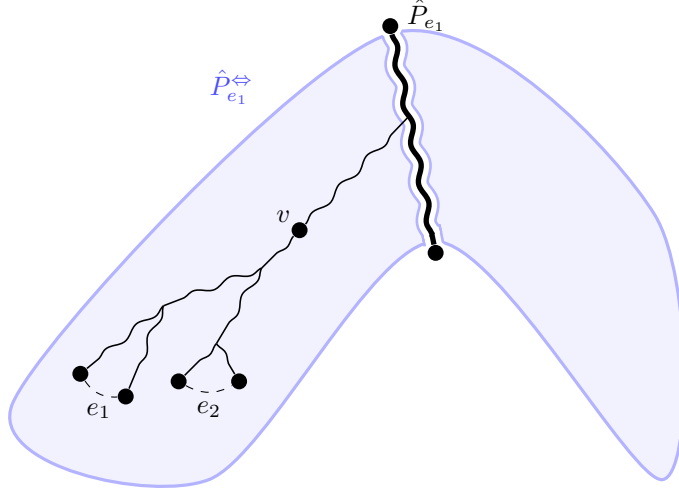
Figure 4: An illustration to Lemma 5.11: if the algorithm arrives at vertex v but has not found any lower vertex for e_1 and e_2 yet, then $\hat{P}_{e_1} = \hat{P}_{e_2}$.

above that $l_e \notin L_v$. This completes the claim that if $l_e \in \text{top}(L_u)$, then $l_e \in \text{top}(L_u \setminus v^\downarrow)$. Now, by Lemma 5.9, if $l_e \in \text{top}(L_u)$, then $\{l_e\} \subseteq \text{top}(L_u \setminus v^\downarrow) \subseteq \text{top}(L_u \setminus Y_{e,u}) = \{l_e\}$. Therefore, $\text{top}(L_u \setminus v^\downarrow) = \{l_e\}$.

Therefore, the algorithm separates the subtree induced from $u^\downarrow \setminus v^\downarrow$ by cutting off the edges $(u, \text{parent}(u))$ and (v, u). Then $\text{top}(L_u \setminus v^\downarrow)$ is obtained by querying $\texttt{MinTree}^\uparrow(u)$. If $\texttt{MinTree}^\uparrow(u)$ returns a vertex l with a value not equal to ∞, then all edges within the package from v has their lower vertex assigned to l by Lemma 5.11. After processing all packages at u, the algorithm combines all edges where their lower vertices are not found yet into one package, which can be efficiently implemented with a linked list. Back to a high-level explanation, observe that although there can be many edges in the package from v forwarded to its parent u that we want to query. We only need to query *once* for each package. This is how we resolve the second implementation challenge about the number of queries.

To analyze the runtime, we notice that the number of top-tree queries made is linear to the total number of packages that has ever created, which is $O(n + m)$ top-tree queries. Therefore, the algorithm for computing l_e for every edge e can be done in $O(m \log n)$ total time. However, from Section 5.2.1 we know that preprocessing $H(v)$ values takes $O(m \log^2 n)$ time. Hence, the total time computing lower vertices for each edge takes $O(m \log^2 n)$ time, which proves Lemma 5.6.

6 Incomparable 2-respecting Minimal Mincuts of Edges

In this section, we present the algorithm computing the minimal mincut of edge when it is a incomparable 2-respecting mincut of T. Similar to the comparable case, for every edge $e \in E$ we call T the *right* tree for e if the minimal mincut of e is a incomparable 2-respecting mincut of T. (In this case we also call e a *right edge* in T.)

Lemma 6.1. *There is an algorithm that, given a graph $G = (V, E)$ and a spanning tree $T \in \mathcal{T}$, in total time $O(m \log^2 n)$ computes, for every edge $e \in E$, an unordered vertex pair $g(e) = (v_e, w_e)$ or null where $v_e, w_e \in V$ with the following guarantee: if T is the right tree for e, then $v_e^{\downarrow} \cup w_e^{\downarrow}$ is the minimal mincut of e.*

Let $e = (u_1, u_2)$ be a right edge in T whose minimal mincut is $v^{\downarrow} \cup w^{\downarrow}$ where $v \neq w$. To prove Lemma 6.1, there are two main cases to consider: (1) the endpoints are in different subtrees: $u_1 \in v^{\downarrow}$ and $u_2 \in w^{\downarrow}$, or (2) both endpoints are in the same subtree, e.g. both $u_1, u_2 \in v^{\downarrow}$.

For the first case, we solve it based on the main lemma below. We will devote to most of this section for proving it.

Lemma 6.2. *Let $\mathcal{P}$ be a path decomposition of T. There is an algorithm that preprocesses $G = (V, E)$ and spanning tree T in $O(m \log n)$ time so that, given any path $P \in \mathcal{P}$, in time $O(d(P^{\downarrow}) \log n)$ the algorithm computes, for every edge $e = (u_1, u_2)$ such that $e \in E(P^{\downarrow})$, an unordered vertex pair $g(e, P) = (v, w)$ or null with the following guarantee:*

Suppose $v^{\downarrow} \cup w^{\downarrow}$ is the incomparable minimal mincut for e such that $u_1 \in v^{\downarrow}$ and $u_2 \in w^{\downarrow}$. Let $P_v, P_w \in \mathcal{P}$ be the paths that contains v and w respectively. Then, either $g(e, P_v) = (v, w)$ or $g(e, P_w) = (v, w)$.

For the second case, observe that $\text{LCA}(u_1, u_2) \in v^{\downarrow}$ since $u_1, u_2 \in v^{\downarrow}$. Therefore, the minimal mincut of the right edge e is also the *minimal incomparable 2-respecting mincut* of vertex $\text{LCA}(u_1, u_2)$, which can be computed efficiently using Lemma 6.3. The proof of Lemma 6.3 is deferred to Appendix B, since it also serves as a building block of Lemma 3.4.

Lemma 6.3. *There is an algorithm that, given a spanning tree T of $G = (V, E)$, in total time $O(m \log^2 n)$ computes, for every vertex $u \in V$ the minimal incomparable 2-respecting mincut candidate $f(u) = (v_u, w_u)$ or null with the following guarantee:*

If there exists an incomparable 2-respecting cut that separating u from root r, then $f(u) \neq$ null and $v_u^{\downarrow} \cup w_u^{\downarrow}$ is such a mincut with smallest size.

Now we prove Lemma 6.1 using Lemmas 6.2 and 6.3:

Proof of Lemma 6.1. Suppose $v^{\downarrow} \cup w^{\downarrow}$ is the minimal mincut for a right edge $e = (u_1, u_2)$. Depending on whether the endpoints of e are in the same subtree (rooted in either v or w), or in the different subtrees, we consider the following two cases:

Case 1: (different subtrees) WLOG, assume $u_1 \in v^{\downarrow}$ and $u_2 \in w^{\downarrow}$. Given a balanced path decomposition $\mathcal{P}$, for each path $P \in \mathcal{P}$, the algorithm in Lemma 6.2 computes a candidate vertex pair $g(e, P) = (v', w')$ or null for every edge $e = (u_1, u_2)$ such that $e \in E(P^{\downarrow})$. Plugging in the path decomposition framework Lemma 4.1, we can compute $g(e, P)$ for all edge $e = (u_1, u_2) \in E$ and all $P \in \mathcal{P}$ such that $e \in E(P^{\downarrow})$ in $O(m \log^2 n)$ time.

Next, for each edge e, the algorithm compares the size of all candidate mincuts $g(e, P)$ for each path P such that $e \in E(P^{\downarrow})$ and select the smallest one to be (v_e, w_e). This step takes $O(\log n)$ time for each edge e since there are at most $O(\log n)$ paths P such that $e \in E(P^{\downarrow})$

by the balanced property of $\mathcal{P}$. Since Lemma 6.2 guarantees that either $g(e, P_v)$ or $g(e, P_w)$ equals to (v, w), our algorithm find the minimal mincut of e.

The total time for this case is $O(m \log^2 n) + O(m \log n) = O(m \log^2 n)$.

Case 2: (same subtree) WLOG, assume $u_1, u_2 \in v^{\downarrow}$. This implies $\text{LCA}(u_1, u_2) \in v^{\downarrow}$. Therefore, the minimal mincut of e will also be the minimal incomparable 2-respecting mincut of $\text{LCA}(u_1, u_2)$, which has been computed via the algorithm from Lemma 6.3 in $O(m \log^2 n)$ time.

Since it takes $O(m \log^2 n)$ time for both cases, the whole algorithm runs in $O(m \log^2 n)$ total time.
$\square$

The rest of the section is devoted for proving Lemma 6.2. In Section 6.1 we introduce several essential concepts that allow us to describe and prove the algorithm in a precise way. In Section 6.2 we describe the high-level idea to the algorithm that finds all incomparable minimal 2-respecting mincut candidate for each edge, and in Section 6.3 we complete the implementation details. Finally in Section 6.4 and Section 6.5 we prove the correctness of the algorithm and analyze the runtime, concluding the proof of Lemma 6.2.

6.1 Minimum v-Precuts and P-Outer Minimum v-Precuts

Fix a spanning tree $T \in \mathcal{T}$. Our algorithm for Lemma 6.2 requires implementation on efficiently identifying incomparable 2-respecting mincuts $v^{\downarrow} \cup w^{\downarrow}$ with one vertex v on the path $P \in \mathcal{P}$. In this section we first recall the concepts of *minprecut values* from [Kar00] and then we introduce *outer minprecut values* that help us to describe the algorithm with clarity.

Observe that for any pair of incomparable vertices $v \perp w$, the weight of the cut induced by $v^{\downarrow} \cup w^{\downarrow}$ can be expressed as

$$\mathcal{C}(v^{\downarrow} \cup w^{\downarrow}) = \mathcal{C}(v^{\downarrow}) + \mathcal{C}(w^{\downarrow}) - 2\mathcal{C}(v^{\downarrow}, w^{\downarrow}) .$$

Suppose $v^{\downarrow}$ is one *part* (the other part is $w^{\downarrow}$) of the minimal incomparable 2-respecting mincut of e. Then by factoring out the terms only related to v, it suffices to compute the following *incomparable precut values* for all $w \perp v$.

Definition 6.4 (Incomparable precut value)**.** The *incomparable precut value* of v at w, is defined by

$$\mathcal{C}_v^{\perp}(w) := \mathcal{C}(w^{\downarrow}) - 2\mathcal{C}(v^{\downarrow}, w^{\downarrow}) .$$

Note that the value is only defined in the incomparable scenario $w \perp v$.

The incomparable precut values are defined analogously to the comparable precut values. Actually, they share the same functionality in the sense that given a vertex v, an algorithm can be designed to find a *partner* w such that v and w together idetifies a incomparable (resp. comparable) 2-respecting mincut $v^{\downarrow} \cup w^{\downarrow}$ (resp. $w^{\downarrow} \setminus v^{\downarrow}$). Such partner should be a minimizer of the *minpercut value* defined below.

Definition 6.5. The *incomparable minprecut value* of v, is defined by

$$\mathcal{C}_v^\perp := \min\{\mathcal{C}_v^\perp(w) \mid w \perp v\} \ .$$

A vertex w is called an *incomparable mincut partner* of v if $v^\downarrow \cup w^\downarrow$ is a mincut. For brevity, we will omit the word "comparable/incomparable" and simply call $\mathcal{C}_v^\perp(w)$ as precut value of v (at w) and call w a *mincut partner* of v if the context is clear. Besides, we call w a *minprecut partner* of v if $\mathcal{C}_v^\perp(w) = \mathcal{C}_v^\perp$. Note that a mincut partner must be a minprecut partner, but it may not be correct conversely.

Given a path $P \in \mathcal{P}$, our algorithm will maintain the minprecut value of the current vertex v when climbing up the path P. It turns out that if we further exclude all the candidates of partner within $P^\downarrow$, the algorithm can maintain minprecut values and partners of v in a more efficient way, which leads to the following definitions.

Definition 6.6. We call w an outer vertex of path P if $w \notin P^\downarrow$.

The *outer minprecut value* is defined similar to the minprecut value except that only the outer vertices are taken in consideration.

Definition 6.7. The *P-outer minprecut value* of v is defined as the minprecut value of v such that the minprecut partner w is an outer vertex of v. Specifically,

$$\mathcal{C}_{v,P}^\perp = \min\{\mathcal{C}_v^\perp(w) \mid w \notin P^\downarrow, w \perp v\}$$

We call w a *P-outer minprecut partner* of v if w is an outer vertex of P and $\mathcal{C}_{v,P}^\perp(w) = \mathcal{C}_{v,P}^\perp$. Now we are ready to describe the algorithm.

6.2 Algorithm Description

Now, we are ready to describe the algorithm for Lemma 6.2.

The high-level description of the algorithm is as follow: Given a path $P = (v_1, v_2 \cdots, v_k) \in \mathcal{P}$ with v_1 being the deepest vertex, our algorithm visits v_i in the order of $i = 1, 2, \ldots, k$. At iteration i, the algorithm visits v_i and maintains the invariant such that the values $\mathcal{C}_{v_i}^\perp(w)$ for all P-outer vertex w can be accessed via $val[w]$, and the P-outer minprecut value $\mathcal{C}_{v_i,P}^\perp$ is stored in val^*.

Whenever the algorithm reaches the vertex v_i, it first calls a subroutine called $\texttt{LocalUpdate}(v_i)$ that will recover the invariant on $val[w]$. Once the invariant holds, if v_i has a P-outer mincut partner w, then any edge $e = (u_1, u_2)$ where $u_1 \in v_i^\downarrow$ and $u_2 \in w^\downarrow$ which has not been assigned a minimal mincut candidate yet should obtain an incomparable 2-respecting mincut (v_i, w') for some specific choice of $w' \in w^\downarrow$ as the minimal mincut candidate $g(e, P)$.

To implement this high-level plan, our algorithm will maintain a set $\hat{E} \subseteq E(v_i^\downarrow)$ that contains all edge e whose $g(e, P)$ values is not assigned yet, and a *witness* set W of vertices that will be helpful for extracting the correct minimal mincut candidates. We summarize the invariant for W as follows, and we defer the proof to the end of Section 6.4.

Lemma 6.8 (Invariant for W and $\hat{E}$). *The witness set W and the edge set $\hat{E}$ satisfies the following invariant whenever the algorithm returns from* $\texttt{LocalUpdate}(v_i)$ *when visiting* $v_i \in P$:

(1) W *is always a subset of P-outer minprecut partners of the current visiting vertex v_i.*

(2) *Any edge $e = (u_1, u_2) \in \hat{E}$ satisfies that $u_1 \in v_i^{\downarrow}$ and $u_2 \notin P^{\downarrow}$.*

(3) *(**Correctness Guarantee**): for any edge $e = (u_1, u_2) \in \hat{E}$ such that $u_1 \in v_i^{\downarrow}$, if there exists a P-outer mincut partner of v_i that is an ancestor of u_2, then there exists a witness $w \in W$ that is an ancestor of u_2.*

Use W to find incomparable minimal 2-respecting mincuts for edges. Following this property, we are able to present the algorithm to find incomparable minimal 2-respecting mincuts for edges using W and $\hat{E}$. The algorithm visits $v_1, v_2, \ldots, v_k$ along the path. Whenever the algorithm visits v_i, all edges $e = (u_1, u_2)$ such that $u_1 \in v_i^{\downarrow} \setminus v_{i-1}^{\downarrow}$ and $u_2 \notin P^{\downarrow}$ shall be added to $\hat{E}$ (let $v_0^{\downarrow} = \emptyset$ for convenience).

In order to obtain correct results, after v_i is visited, the algorithm should identify a mincut candidate $g(e, P) := (v_{e,P}, w_{e,P})$ for all right edges $e \in \hat{E}$ with $v_{e,P} = v_i$. One way to achieve this is to scan through all edges in $\hat{E}$ and use $\texttt{MinPath}^{\downarrow}$ queries on a dynamic tree to find $w_{e,P}$ (whenever $v_{e,P} = v_i$ we will find $w_{e,P}$ correctly). However, the amount of edges whose $v_{e,P} = v_i$ could be a tiny fraction in $\hat{E}$.

Using the correctness guarantee from Lemma 6.8, we will be able to identify all edges whose $v_{e,P} = v_i$ without spending any time on any other edges: all we need to do is to extract the subset of $\hat{E}$ whose outer endpoints are descendants of any $w \in W$. Interestingly, the task of finding all descendant outer endpoints can be reduced to a dynamic range query problem, which can be solved efficiently using a standard binary search tree. We discuss the implementation details in Section 6.3 and also in $\texttt{AssignMinCut}(v_i)$ (see Algorithm 3).

Now it comes down to efficiently maintain the witness set W.

Maintaining the set W. The algorithm maintains the set W as following:

- In the beginning of iteration i, if the minprecut value of v_i is the same of the minprecut value of v_{i-1}, we keep the set W. Otherwise reset W to be empty.

- For each newly added edge $e = (u_1, u_2)$ where $u_1 \in v_i^{\downarrow} \setminus v_{i-1}^{\downarrow}, u_2 \notin P^{\downarrow}$, we find the highest ancestor of u_2 which is a P-outer minprecut partner of v_i, and add it to W.

- For the current vertex v_i, if it has some P-outer mincut partners, reset W to be empty in the end of this iteration.

We summarize and implement the procedure that maintains W, together with maintaining the precut values as we move from the vertex v_{i-1} to v_i into the algorithm $\texttt{LocalUpdate}(v_i)$. The high level implementation of the entire algorithm is summarized in Algorithm 1.

24

Algorithm 1 Computing incomparable 2-respecting minimal mincuts of edges

---

1: Initialize the witness set $W \leftarrow \emptyset$, $\hat{E} \leftarrow \emptyset$, and $val^* = \infty$.
2: **for each** $i = 1, 2, \ldots, k$ **do**   ▷ The i-th iteration handles v_i.
3:      Call `LocalUpdate`(v_i).   ▷ Algorithm 2: update W, $\hat{E}$, and val^*.
4:      Call `AssignMinCut`(v_i).   ▷ Algorithm 3.
5:      **if** $\mathcal{C}^{\perp}(v_i^{\downarrow}) + val^* = \lambda$ **then**
6:          Reset $W \leftarrow \emptyset$.
7:      **end if**
8: **end for**

6.3   Implementation

There are two steps that we need to provide a detailed implementation: maintaining the set W (implementing `LocalUpdate`(v_i)), and using W to compute the minimal 2-repsecting mincut candidates of all edges via range queries (implementing `AssignMinCut`(v_i)).

Implementing `LocalUpdate` and Preprocessing. We use a dynamic tree (Lemma 4.2) to maintain the precut values of the current vertex v_i while climbing up the path P. We maintain the invariant that once we process the vertex v_i the values $\mathcal{C}_{v_i}^{\perp}(w)$ for all $w \perp v_i, w \notin P^{\downarrow}$ can be accessed via $val[w]$. Recall that $\mathcal{C}_{v_i}^{\perp}(w) = \mathcal{C}(w^{\downarrow}) - 2\mathcal{C}(v_i^{\downarrow}, w^{\downarrow})$. In the preprocessing step before the path P was given, we set $val[w] = \mathcal{C}(w^{\downarrow})$ for each vertex w and create the dynamic tree on T, which can be done using the same preprocessing step of comparable case in footnote 7. After that, we initialize the minprecut value val^* to be the minimum of $val[w]$ over all the P-outer vertices, which can be done using the following dynamic tree operations. We first apply `Cut`$(v_n, \text{parent}(v_n))$ to separate the subtree $P^{\downarrow}$. Then we apply `AddPath`$(\text{parent}(v_n), \infty)$ to set the value of all the ancestor of v_n to be ∞, since they are not incomparable vertices of any vertex $v_i \in P$. Now we use `MinTree`$^{\uparrow}(\text{parent}(v_n))$ to find w which is argmin of $val[w]$ over all the P-outer vertices, and set $val^* = val[w]$. Finally, we apply `Link`$(v_n, \text{parent}(v_n))$ to restore the tree.

Given the path P, the algorithm scans through the vertices $v_1, v_2, \cdots v_k$ on the path one by one. Suppose the algorithm reaches v_i now. With the invariant after processing v_{i-1} (the invariant for v_0 is $val[w] = \mathcal{C}(w^{\downarrow})$), it suffices to substract $2\mathcal{C}(v_i^{\downarrow} \setminus v_{i-1}^{\downarrow}, w)$ to $val[w]$ for each $w \perp v_i, w \notin P^{\downarrow}$ by invoking `AddPath`$(u, -2\mathcal{C}(v, u))$ for each edge (v, u) where $v \in v_i^{\downarrow} \setminus v_{i-1}^{\downarrow}, u \notin P$. (These edges can be found in $O(d(v_i^{\downarrow} \setminus v_{i-1}^{\downarrow}))$ time using a DFS from v_i without searching the subtree rooted at v_{i-1}.) Therefore, in $O(d(v_i^{\downarrow} \setminus v_{i-1}^{\downarrow}) \log n)$ time, $val[w]$ are updated to $\mathcal{C}_{v_i}^{\perp}(w)$ for all P-outer vertex $w \perp v_i$.

While the algorithm updates the precut value as stated above, it also updates the minprecut value val^*, set W and set $\hat{E}$. The algorithm scans through the edges (v, u) where $v \in v_i^{\downarrow} \setminus v_{i-1}^{\downarrow}, u \notin P$ and invokes `MinPath`$^{\uparrow}(u)$ to find the vertex w with the minimum precut value, and break tie by finding the highest one. Then we update the minprecut value val^*. If val^* changes, we reset W to be empty. Finally, if $val[w] = val^*$, we insert w to the set W. These steps for maintaining the set W are equivalent to the high-level description in Section 6.2.

Notice that in Algorithm 2 we explicitly use a global binary search tree data structure (BST) representing the set $\hat{E}$. Thus, there will be some steps (line 3) involving BST that will be used for

finding the mincut candidates, which is described in the next paragraph.

Algorithm 2 LocalUpdate(v_i)

1: Call AddPath(v_i, ∞).
2: **for each** edge $e = (v, u)$ such that $v \in v_i^{\downarrow} \setminus v_{i-1}^{\downarrow}, u \notin P^{\downarrow}$ **do**
3:      Call InsertBST(pre_order$[u], e$). $\triangleright$ Insert e to $\hat{E}$.
4:      Call AddPath($u, -2\mathcal{C}(v, u)$).
5:      Call $w \leftarrow$ MinPath$^{\uparrow}(u)$.
6:      **if** $val[w] < val^*$ **then**
7:          $val^* \leftarrow val[w]$.
8:          Reset $W \leftarrow \emptyset$.
9:      **end if**
10:     **if** $val[w] = val^*$ **then**
11:         Insert w into W.
12:     **end if**
13: **end for**

Computing minimal mincut candidates of edges from W. First, we clarify the global data structures in the algorithm. While climbing up the path P, we use a binary search tree (BST) to store all the edges in $\hat{E}$, with the key equals to the pre-order indices of the outer endpoints. Note that there will be only one global BST in our algorithm. The BST supports the following primitives.

- InsertBST(x, e), store edge $e = (u_1, u_2)$ ($u_1 \in P^{\downarrow}, u_2 \notin P^{\downarrow}$) into the BST with key x. In our algorithm, we will always set x to be pre_order$[u_2]$, i.e., the pre-order index of vertex u_2.

- ExtractBST($w^{\downarrow}$), extract all the edges with endpoints in $w^{\downarrow}$, i.e. the edges with index in the interval corresponds to $w^{\downarrow}$.

We use the BST to store the edges that have not found their minimal mincut candidates yet. When the algorithm reaches v_i on the path, it inserts all edges $e = (v, u)$ with v in $v_i^{\downarrow} \setminus v_{i-1}^{\downarrow}$ and $u \notin P^{\downarrow}$ to the BST, with the key pre_order$[u]$. This step is implemented in line 3 of Algorithm 2. To compute the minimal mincut candidates of edges using W, we first check whether there exists a mincut partner of v_i (line 1 in Algorithm 3). If there exists, we perform the range query using BST to extract all the edges with endpoint $u \in w^{\downarrow}$ for each $w \in W$, which is implemented in Algorithm 3. Finally, for each extracted edge with endpoint $u \in w^{\downarrow}$, we find w' to be the lowest ancestor such that w' is the mincut partner of v_i, and set $g(e, P) = (v_i, w')$.

Algorithm 3 AssignMinCut(v_i)

1: **if** $\mathcal{C}(v_i^\downarrow) + val^* = \lambda$ **then**
2:     **for each** $w \in W$ **do**
3:         **for each** $e = (v, u) \leftarrow \texttt{ExtractBST}(w'^\downarrow)$ **do**
4:             $w' \leftarrow \texttt{MinPath}^\downarrow(u)$.
5:             Set $g(e, P) = (v_i, w')$.
6:         **end for**
7:     **end for**
8: **end if**

6.4 Correctness

First, we show a basic property of path decomposition and outer vertex that will be useful for analyzing the algorithm.

Lemma 6.9. *For any spanning tree T, any path decompostion $\mathcal{P}$, and any vertices v and w so that $v \perp w$, let $P_v \neq P_w \in \mathcal{P}$ be the paths that contains v and w respectively. Then either v is a P_w-outer vertex or w is a P_v-outer vertex.*

Proof. Observe that either $P_v \cap P_w^\downarrow = \emptyset$ or $P_w \cap P_v^\downarrow = \emptyset$, which implies the lemma. $\qquad\square$

For any right edge $e = (u_1, u_2)$, suppose the minimal mincut of e is $v_e^\downarrow \cup w_e^\downarrow$ where $u_1 \in v_e^\downarrow$ and $u_2 \in w_e^\downarrow$. By Lemma 6.9, without loss of generality, suppose w is a P_v-outer vertex where P_v is the path containing v. The main property for the correctness of the algorithm can be summarized in Corollary 6.10, which can be derived from maintaining the invariant of W (Lemma 6.8).

Corollary 6.10. *For a path $P \in \mathcal{P}$ and an edge $e = (u_1, u_2)$ with $u_1 \in P^\downarrow$ and $u_2 \notin P^\downarrow$, let v_i be the deepest vertex in P that has an outer mincut partner w where $u_1 \in v_i^\downarrow$ and $u_2 \in w^\downarrow$. Then, there exists an ancestor of w in W when returned from $\texttt{LocalUpdate}(v_i)$ at iteration i in Algorithm 1.*

Proof. It suffices to show that $e \in \hat{E}$ at iteration i after returning from $\texttt{LocalUpdate}(v_i)$. Then the lemma is implied by the correctness guarantee of Lemma 6.8.

Since v_i is the deepest vertex in P that has an outer mincut partner w where $u_2 \in w^\downarrow$, the edge e has not obtained its mincut candidate $g(e, P)$ yet, hence $e \in \hat{E}$. $\qquad\square$

Using Corollary 6.10, we are able to prove that all minimal incomparable 2-respecting mincut candidate can be correctly found:

Proof of the correctness of Lemma 6.2. For any right edge $e = (u_1, u_2)$, suppose the minimal mincut of e is $v^\downarrow \cup w^\downarrow$ where $u_1 \in v^\downarrow$ and $u_2 \in w^\downarrow$. By Lemma 6.9, WLOG, suppose w is a P_v-outer vertex where P_v is the path containing v. First, we show that there doesn't exist any mincut $v'^\downarrow \cup w'^\downarrow$ containing e such that $v' \in v^\Downarrow$. Otherwise, the intersection of $v'^\downarrow \cup w'^\downarrow$ and $v^\downarrow \cup w^\downarrow$ will be a smaller mincut containing e, contradicts to minimality.

Let $P = P_v$ for brevity. By the argument above, v is the deepest vertex in P that has an outer mincut partner w where $u_1 \in v_i^{\downarrow}$ and $u_2 \in w^{\downarrow}$. Then by Corollary 6.10, there exists w' to be an ancestor of w such that $w' \in W$ at the iteration of v in Algorithm 1. Therefore, the algorithm will extract the edge e since $u_1 \in v^{\downarrow}$ and $u_2 \in w'^{\downarrow}$. Then the algorithm finds w as it is the lowest ancestor of u_2 and also a P-outer mincut partner of v. Finally, the algorithm sets $g(e, P) = (v, w)$ as desired. □

Finally, we prove that the invariant for W and $\hat{E}$ holds.

Proof of Lemma 6.8. The first and second invariants are directly from the algorithm.

Now we prove the third invariant correctness guarantee. Consider when the algorithm returns from `LocalUpdate`(v_i). For any edge $e = (u_1, u_2) \in \hat{E}$ where $u_2 \notin P^{\downarrow}$ such that there exists a P-outer mincut partner of v_i that is an ancestor of u_2, let w be the lowest P-outer mincut partner of v_i that is an ancestor of u_2, then we prove that $w \in W$.

Observe that val^* is non-increasing over the time. Let j be the largest index such that $\mathcal{C}_{v_j, P}^{\perp}(w) \neq \mathcal{C}_{v_{j-1}, P}^{\perp}(w)$. Since w is a mincut partner of v_i, we deduce that w is inserted to W in `LocalUpdate`(v_j) and val^* doesn't change during the iterations $(j, i]$, which implies that W is incremental in the following iterations. Therefore, w remains in W when the algorithm returns from `LocalUpdate`(v_i). □

6.5 Runtime Analysis

Proof of the running time of Lemma 6.2. First, for the preprocessing step stated in the beginning of Section 6.3, it takes $O(m \log n)$ time to initialize $val[w] = \mathcal{C}(w^{\downarrow})$ for every vertex w and create the dynamic tree on T. After given a path P, it takes constant dynamic tree operations to initialize val^*.

In iteration i of Algorithm 1, we show that the cost of the subroutines `LocalUpdate`(v_i) is in $O(d(v_i^{\downarrow} \setminus v_{i-1}^{\downarrow}) \log n)$ time. And we show that the total time processing `AssignMinCut`(v_i) is $O(d(P^{\downarrow}) \log n)$ for the whole path P. Therefore, we deduce that the total time processing a path P is within $O(d(P^{\downarrow}) \log n)$ time.

In Algorithm 2 `LocalUpdate`(v_i), for each edge e with one endpoint in $v_i^{\downarrow} \setminus v_{i-1}^{\downarrow}$, the algorithm invoke constant time dynamic tree operations and BST operations. Therefore, the total time cost is $O(d(v_i^{\downarrow} \setminus v_{i-1}^{\downarrow}) \log n)$.

In Algorithm 3 `AssignMinCut`(v_i), it only goes into the if clause when there exists some mincut partner of v_i. Then the time cost will be $O(p_i + q_i \log n)$ where p_i is the size of set W and q_i is the number of edges extracted from the BST in iteration i. Since W is reset to be empty each time when the if-clause is executed, and by line 11 in Algorithm 2 each edge in $E(P^{\downarrow})$ causes at most one insertion to W, hence $\sum_i p_i$ is bounded by $O(d(P^{\downarrow}))$. Furthermore, each edge got inserted and deleted at most once in $\hat{E}$ so $\sum_i q_i$ is bounded by $O(d(P^{\downarrow}))$.

Therefore, the algorithm preprocesses $G = (V, E)$ and spanning tree T in $O(m \log n)$ time, and processes each path P in $O(d(P^{\downarrow}) \log n)$ time. □

7 Putting Everything Together: Proof of Lemma 3.5

Finally, we show that given a spanning tree T, we can combine the results in Sections 5 and 6 to get the minimal 2-respecting mincut of every edge e in $O(m \log^2 n)$ total time. As the discussion below Lemma 3.5, the mincut can be classified into three types.

Type 1. By Lemma 3.8, the algorithm computes the minimal 1-respecting mincut candidate for every edge e.

Type 2-Comparable. By Lemma 5.1, the algorithm computes the minimal comparable 2-respecting mincut candidates for every edge e.

Type 2-Incomparable. By Lemma 6.1, the algorithm computes the minimal incomparable 2-respecting mincut candidates for every edge e.

For each edge e, we get three minimal mincut candidates as above. If all of the three candidates are null, then the minimal 2-respecting mincut of e respects to T is null. Otherwise, the minimal 2-respecting mincut of e respects to T is the mincut with the smallest size among the minimal mincut candidates.

Since the algorithm for each case runs in $O(m \log^2 n)$ time and the comparing time is constant for each edge, the whole algorithm runs in $O(m \log^2 n)$ total time, concluding Lemma 3.5.

Acknowledgment

References

[AHLT05] Stephen Alstrup, Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. Maintaining information in fully dynamic trees with top trees. *Acm Transactions on Algorithms (talg)*, 1(2):243–264, 2005. 10, 11

[BLS20]   Nalin Bhardwaj, Antonio Molina Lovett, and Bryce Sandlund. A simple algorithm for minimum cuts in near-linear time. In Susanne Albers, editor, *17th Scandinavian Symposium and Workshops on Algorithm Theory, SWAT 2020, June 22-24, 2020, Tórshavn, Faroe Islands*, volume 162 of *LIPIcs*, pages 12:1–12:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. 9

[CGL+20] Julia Chuzhoy, Yu Gao, Jason Li, Danupon Nanongkai, Richard Peng, and Thatchaphol Saranurak. A deterministic algorithm for balanced cut with applications to dynamic connectivity, flows, and beyond. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1158–1167. IEEE, 2020. 2

[CLP22a]   Ruoxu Cen, Jason Li, and Debmalya Panigrahi. Augmenting edge connectivity via isolating cuts. In *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 3237–3252. SIAM, 2022. 1

[CLP22b]   Ruoxu Cen, Jason Li, and Debmalya Panigrahi. Edge connectivity augmentation in near-linear time. In Stefano Leonardi and Anupam Gupta, editors, *STOC '22: 54th Annual ACM SIGACT Symposium on Theory of Computing, Rome, Italy, June 20 - 24, 2022*, pages 137–150. ACM, 2022. 2

[DEMN21]   Michal Dory, Yuval Efron, Sagnik Mukhopadhyay, and Danupon Nanongkai. Distributed weighted min-cut in nearly-optimal time. In Samir Khuller and Virginia Vassilevska Williams, editors, *53rd Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 1144–1153, 2021. 8

[DKL76]   Efim A. Dinits, Alexander V. Karzanov, and Micael V. Lomonosov. On the structure of a family of minimum weighted cuts in a graph. *Studies in Discrete Optimization*, pages 209–306, 1976. 1, 3, 4

[FF09]   Tamás Fleiner and András Frank. A quick proof for the cactus representation of mincuts. *EGRES Quick-Proofs Series*, 3, 2009. 1

[Fle99]   Lisa Fleischer. Building chain and cactus representations of all minimum cuts from hao–orlin in the same asymptotic run time. *Journal of Algorithms*, 33(1):51–72, 1999. 1, 2

[Gab91]   Harold N Gabow. Applications of a poset representation to edge connectivity and graph rigidity. In *[1991] Proceedings 32nd Annual Symposium of Foundations of Computer Science*, pages 812–821. IEEE Computer Society, 1991. 1

[Gab06]   Harold N Gabow. Using expander graphs to find vertex connectivity. *Journal of the ACM (JACM)*, 53(5):800–844, 2006. 2

[Gab16]   Harold N Gabow. The minset-poset approach to representations of graph connectivity. *ACM Transactions on Algorithms (TALG)*, 12(2):1–73, 2016. 1, 2, 6, 9, 33, 35, 36, 37, 38, 39

[GG21]   Barbara Geissmann and Lukas Gianinazzi. Parallel minimum cuts in near-linear work and low depth. *ACM Trans. Parallel Comput.*, 8(2):8:1–8:20, 2021. 8, 9

[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

[GHT18]   Gramoz Goranci, Monika Henzinger, and Mikkel Thorup. Incremental exact min-cut in polylogarithmic amortized update time. *ACM Transactions on Algorithms (TALG)*, 14(2):1–21, 2018. 1

[GMW20]   Paweł Gawrychowski, Shay Mozes, and Oren Weimann. Minimum cut in $O(m \log^2 n)$ time. In *47th International Colloquium on Automata, Languages, and Programming, ICALP*, 2020. 3, 4, 9

[GT85]   Harold N Gabow and Robert Endre Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of computer and system sciences*, 30(2):209–221, 1985. 14

[Hen95]   Monika Rauch Henzinger. Approximating minimum cuts under insertions. In *International Colloquium on Automata, Languages, and Programming*, pages 280–291. Springer, 1995. 1

[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)*, 2024. 2

[HO92]   Jianxiu Hao and James B. Orlin. A faster algorithm for finding the minimum cut in a graph. In *Proceedings of the Third Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms (SODA)*, 1992. 1

[HRW20]   Monika Henzinger, Satish Rao, and Di Wang. Local flow partitioning for faster edge connectivity. *SIAM Journal on Computing*, 49(1):1–36, 2020. 1, 2

[Kar93]   David R Karger. Global min-cuts in rnc, and other ramifications of a simple min-cut algorithm. In *SODA*, volume 93, pages 21–30. Citeseer, 1993. 1

[Kar00]   David R Karger. Minimum cuts in near-linear time. *Journal of the ACM (JACM)*, 47(1):46–76, 2000. 1, 2, 3, 4, 7, 8, 9, 22

[KP09]   David R Karger and Debmalya Panigrahi. A near-linear time algorithm for constructing a cactus representation of minimum cuts. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 246–255. SIAM, 2009. 1, 2, 4, 5, 6, 7, 8, 9, 29, 32, 33, 34, 35, 43, 45, 48

[KS96]   David R Karger and Clifford Stein. A new approach to the minimum cut problem. *Journal of the ACM (JACM)*, 43(4):601–640, 1996. 1

[KT86]   Alexander V Karzanov and Eugeniy A Timofeev. Efficient algorithm for finding all minimal edge cuts of a nonoriented graph. *Cybernetics*, 22(2):156–162, 1986. 1

[KT18]   Ken-ichi Kawarabayashi and Mikkel Thorup. Deterministic edge connectivity in near-linear time. *J. ACM*, 66(1):4:1–4:50, 2018. 1, 2

[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. 1, 2, 3, 4

[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. 1, 2

[LST20]   On-Hei Lo, Jens Schmidt, and Mikkel Thorup. Compact cactus representations of all non-trivial min-cuts. *Discrete Applied Mathematics*, 303, 04 2020. 1, 2

[MN20]   Sagnik Mukhopadhyay and Danupon Nanongkai. Weighted min-cut: sequential, cut-query, and streaming algorithms. *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, 2020. 3, 4, 8, 9

[NGM97]   Dalit Naor, Dan Gusfield, and Charles Martel. A fast algorithm for optimally increasing the edge connectivity. *SIAM Journal on Computing*, 26(4):1139–1165, 1997. 1, 2

[NI92a]   Hiroshi Nagamochi and Toshihide Ibaraki. Computing edge-connectivity in multigraphs and capacitated graphs. *SIAM Journal on Discrete Mathematics*, 5(1):54–66, 1992. 1

[NI92b]   Hiroshi Nagamochi and Toshihide Ibaraki. A linear-time algorithm for finding a sparse k-connected spanning subgraph of ak-connected graph. *Algorithmica*, 7(1):583–596, 1992. 1

[NK94]   Hiroshi Nagamochi and Tiko Kameda. Canonical cactus representation for minimum cuts. *Japan Journal of Industrial and Applied Mathematics*, 11:343–361, 1994. 1

[NNI00]   Hiroshi Nagamochi, Yoshitaka Nakao, and Toshihide Ibaraki. A fast algorithm for cactus representations of minimum cuts. *Japan journal of industrial and applied mathematics*, 17(2):245–264, 2000. 1, 2

[NV91]   Dalit Naor and Vijay V Vazirani. Representing and enumerating edge connectivity cuts in rnc. In *Workshop on Algorithms and Data Structures*, pages 273–285. Springer, 1991. 1

[RZZ23]   R Ravi, Weizhong Zhang, and Michael Zlatin. Approximation algorithms for steiner tree augmentation problems. In *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2429–2448. SIAM, 2023. 1

[Sar21]   Thatchaphol Saranurak. A simple deterministic algorithm for edge connectivity. In *Symposium on Simplicity in Algorithms (SOSA)*, pages 80–85. SIAM, 2021. 1, 2

[ST83]   Daniel D Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of computer and system sciences*, 26(3):362–391, 1983. 10, 11

[SW97]   Mechthild Stoer and Frank Wagner. A simple min-cut algorithm. *Journal of the ACM (JACM)*, 44(4):585–591, 1997. 1

[SY23]   Thatchaphol Saranurak and Sorrachai Yingchareonthawornchai. Deterministic small vertex connectivity in almost linear time, 2023. To appear at SODA'23. 2

A   Constructing Cactus from Minimal Mincuts

The goal of this section is to prove Lemma 3.2:

Lemma 3.2. *Given a graph $G = (V, E)$, a tree packing $\mathcal{T}$ and the set of cut labels representing minimal mincuts of each vertex $v \in V$ and each edge $e \in E$, there exists a deterministic algorithm that computes a cactus representation in $O(m\alpha(m, n) + n|\mathcal{T}|)$ time, where $\alpha(m, n)$ is the inverse Ackermann function.*

The difference between Lemma 3.2 and the statement given in [KP09] is that we are given all cut labels representing minimal mincuts of *all* edges, while in [KP09] they are given only the labels

of minimal mincuts of some edges.[8] Because of this, we are able to give a simpler algorithm than the one provided in [KP09].

This section is organized as follows. In Appendix A.3 and Appendix A.4 we define the *hierarchy representation H* for all mincuts. With the hierarchy representation H, a cactus can then be constructed in linear time (Lemma A.10). In Appendix A.5 we describe the algorithm that computes the hierarchy representation H. Before we introduce the notations and dive into the construction — for both the warmup as well as historical reasons — we would like to introduce a cruder hierarhcical representation which we call it *nesting relation tree $\hat{T}$* in Appendix A.1 and Appendix A.2. Both warmup subsections are not strictly required in Appendix A.3 and Appendix A.4 but they become useful for proving correctness in Appendix A.5.

A.1 Warmup I: Assumptions and Preprocessing

Let r be the root we chose to fix throughout the algorithm. As a technical reminder: from now on, we will abuse the notation and identify every cut $(X, V \setminus X)$ by the subset X where $r \notin X$. Any two cuts X and Y are nesting if either $X \subseteq Y$ or $Y \subseteq X$. For any vertex $v \neq r$ let X_v to be the minimal mincut of v on G. For convenience we also define $X_r = V$. Similarly, for each edge e, if there exists a minimal mincut for e it is denoted by X_e; otherwise we set $X_e = V$.

Preprocessing: Merging Vertices with the Same Minimal Mincuts. First, without loss of generality we may assume that all vertices (except r) have distinct minimal mincuts. This assumption can be achieved easily by merging vertices with the same minimal mincut: if two vertices u and v have the same minimal mincut, then any mincut does not separate u and v. If a vertex u does not have a minimal mincut, then any mincut does not separate u and r. Therefore, all mincuts are preserved in the merged graph.

The preprocessing can be done in linear time deterministically: the algorithm first performs a bucket sort to all vertices' minimal mincut labels according to their sizes and breaking ties lexicographically. Then, two vertices can be merged if and only if they are neighbors in the sorted order and they have identical cut labels. Notice that we may assume that two cuts are the same if and only if their labels are the same.

A.2 Warmup II: Nesting Relation Tree

Readers may skip this subsection if the goal is to obtain only a high-level idea of the algorithm.

---

[8]In particular, [KP09] reduces the problem into several *one-layer cactus construction problems*, and piecing back all constructed partial (and contiguous) cactus after solving these separated subproblems. The main reason for introducing this seemly-extra reduction step is because in their algorithm some edges do not have correct minimal mincut labels — these erroneous labelings will not cause a problem (with high probability) in the reduced problems. This reduction makes the problem simpler to solve. However, the reduction complicates the correctness proof and some details were omitted.

Our simplified algorithm described in this section does not depend on the reduction mentioned above. We emphasize that this algorithm is in fact equivalent to performing [KP09] in a bottom-up fashion. We give the algorithm and the correctness proof via a *hierarchy representation* introduced implicitly by Gabow [Gab16] in Appendix A.5.

Let $v \neq r$ be a non-root vertex. By the non-crossing property (Lemma 2.2) of the mincuts, any two minimal mincuts of vertices are either nesting or disjoint. With the preprocessing from Appendix A.1, we are able to see that for all $v \neq r$, there exists a unique vertex $p_v \in V$ such that $X_v \subsetneq X_{p_v}$ and there exists no vertex $u \notin \{v, p\}$ with $X_v \subsetneq X_u \subsetneq X_{p_v}$.[9]

Now, the nesting relation tree $\hat{T}$ can be defined by designating p_v as the parent of v for each non-root v. It is straightforward to check that $\hat{T}$ is a well-defined tree, by observing that the nesting relations among all minimal mincuts of vertices are acyclic. The following lemma states that $\hat{T}$ can be constructed efficiently.

Lemma A.1. *Given the labels to minimal mincuts of vertices and the tree packing $\mathcal{T}$, there is a deterministic algorithm that constructs $\hat{T}$ in $O(n|\mathcal{T}|)$ time.*

Proof. Consider the collection of all minimal mincuts of vertices $\mathscr{X} := \{X_v\}_{v \in V}$. For each vertex $v \neq r$, to find its parent p_v on $\hat{T}$, it suffices to search for the second smallest sized set $\mathscr{X}$ that contains v (the smallest one is X_v by definition.)

This can be done by considering each tree separatedly. Recall that each mincut has a label $(type, v, w, T)$. For each tree $T \in \mathcal{T}$, we define $\mathscr{X}_T$ to be all minimal mincuts of vertices with its label referring to T. For any vertex v, if we are able to obtain the smallest and the second smallest mincuts of $\mathscr{X}_T$ for every tree $T \in \mathcal{T}$, then we are able to obtain the parent p_v by selecting the second smallest sized minimal mincuts among the $\leq 2|\mathcal{T}|$ mincuts that were returned.

Now, observe that $\mathscr{X}_T$ is also a laminar set. Thus, if we mark the $O(1)$ entry points for each mincut in $\mathscr{X}_T$ on the tree T, a simple DFS can be applied to find the first and the second smallest mincuts in $\mathscr{X}_T$ that contains any given vertex in $O(n)$ total time for each tree T. Hence, the total time required is $O(n|\mathcal{T}|)$. $\square$

For any vertex $v \in V$, we define $\hat{T}_v$ to be the set of vertices within the subtree rooted at v. The following lemma gives a neat observation to this nesting relation tree $\hat{T}$:

Lemma A.2. *For all $v \in V$, $X_v = \hat{T}_v$.*

Proof. Observe that $X_u \subseteq X_v$ if and only if u is a descendant of v. Then the statement follows. $\square$

With the help of the nesting relation tree $\hat{T}$, the following lemma enables us to categorize any global mincut:

Lemma A.3. *Let X be any (proper) mincut on G. Then, there is a set of vertices $S \subseteq V$ such that all vertices in S have the same parent on $\hat{T}$ and $X = \bigcup_{u \in S} \hat{T}_u$.*

Proof. First, observe that any mincut X satisfies $X = \cup_{u \in X} X_u$. To see this, if X crosses with some minimal mincut X_v of vertex v, then by Lemma 2.2, both $X_v \setminus X$ and $X_v \cap X$ are mincuts. Since v lies in one of $X_v \setminus X$ and $X_v \cap X$, we have found a smaller mincut that contains v, contradicts to the definition of X_v. By Lemma A.2, we further have that $X = \cup_{u \in X} \hat{T}_u$ so X is union of some subtrees in $\hat{T}$.

---

[9]In [KP09] the authors call X_{p_v} the second smallest minimal mincut of v.

Now we would like to show that the roots of these subtrees have the same parent in $\hat{T}$. Let $S \subseteq X$ be comprised of all vertices $x \in X$ such that the parent of x is not in X. Suppose there are two vertices $x, y \in S$ so that their parent vertices p and q are different. Since $p \neq q$, without loss of generality we may assume that $p \notin \hat{T}_q$ Now, X must cross with $X_q(= \hat{T}_q)$ since $x \notin \hat{T}_q$ but $y \in \hat{T}_q$. In this case, one of $X_q \cap X$ or $X_q \setminus X$ is a smaller mincut that contains q, contradicting the definition of X_q. Hence, all vertices in S has the same parent and the statement is true. $\qquad\square$

The next Appendix A.3 gives important characterization to global mincuts, which leads to the actual cactus construction algorithm. [10]

A.3   Chains and Chain Certificates

The goal of this section is to show that every global mincut has a "simple certificate" corresponding to it. This will be crucial for defining the hierarchical representation of all mincuts, which will be given in Appendix A.4. In fact, a generalization of the fact was already shown by Gabow [Gab16] but we give an alternative (and arguably simpler) presentation of this characterization using terminology specific to mincuts.

Definition A.4 (Mincut Certificates)**.** Let $X \subseteq V$ be any set of vertices. We say X has a *vertex certificate* if there is a vertex v so that $X = X_v$. Similarly, we say X has an *edge certificate* if X is the minimal mincut for some edge e.

By the end of this section, we will prove in Lemma A.7 that every mincut must either has a vertex certificate, an edge certificate, or a *chain certificate*. In order to understand the last object, we first give the definitions of chains and chain certificates below.

Definition A.5 (inspired by [Gab16], page 33–35)**.** A *chain* is a sequence of disjoint non-empty vertex subsets $(C_0, C_1, \ldots, C_\ell)$ where $\ell \geq 1$ and is defined recursively:

1. For each i, $C_i \subseteq V$ has either a vertex certificate, an edge certificate, or a *chain certificate*.
2. For each $0 \leq i < \ell$, $C_i \cup C_{i+1}$ has an edge certificate.

We say that a subset of vertices $X \subseteq V$ has a *chain certificate* if there exists a chain $(C_0, C_1, \ldots, C_\ell)$ such that $X = \cup_{i=0}^\ell C_i$. A *subchain* is a consecutive subsequence of a chain. A *maximal chain* is a chain that is not a subchain of any longer chain.

It is very important to note that Definition A.5 is well-defined, in the sense that once all subsets with vertex or edge certificates are fixed, chains and maximal chains will be unambiguously defined. Now, we explore some useful properties of chains. These properties can all be derived from the crossing lemma (Lemma 2.2).

---

[10][KP09] stopped at the above Lemma A.3 and showed that by partitioning all mincuts according to the common parent vertex, the cactus can be constructed layer-by-layer. However, the correctness proof to the remaining part of [KP09] dealing with one-layer cactus construction is not explicitly stated. In the next subsection, we fill in the missing proof (and slightly simplify their algorithm) for Karger and Panigrahi's algorithm.

Lemma A.6 (basic properties of a chain)**.** *Let* $(C_0, C_1, \ldots, C_\ell)$ *be a chain and let* $X = C_0 \cup C_1 \cup \cdots \cup C_\ell$. *Then the following properties are true:*

(1) X *is either a mincut or the vertex set* V.

(2) *For any edge* e *leaving* X *(i.e.,* e *connects* X *and* $V \setminus X$*), then* e *has an endpoint in either* C_0 *or* C_ℓ.

(3) *For any* $0 \leq i < \ell$, $\mathcal{C}(C_i, C_{i+1}) = \lambda/2$; *also,* $\mathcal{C}(C_0, V \setminus X) = \mathcal{C}(C_\ell, V \setminus X) = \lambda/2$. *In particular, for any non-neighbor indices* i *and* j *such that* $|i - j| > 1$ *we have* $\mathcal{C}(C_i, C_j) = 0$.

(4) *(Lemma 4.6 in [Gab16]) For any* i, *if* C_i *has a chain certificate* $(C_0', C_1', \ldots, C_{\ell'}')$, *then this certificate must be a maximal chain.*

(5) *For any edge* e *leaving* C_0 *(resp.* C_ℓ*) to* $V \setminus X$, *the minimal mincut of* e *either contains* C_0 *(resp.* C_ℓ*) but nothing from* $X \setminus C_0$ *(resp.* $X \setminus C_\ell$*), or contains the entire* X.

Proof. We prove each property one by one below.

Property (1). This can be done by induction on $|X|$ when $|X| < n$: we know that C_ℓ has a certificate so it is a mincut. Now, by the fact that $X \setminus C_\ell$ cross with $C_{\ell-1} \cup C_\ell$, by Lemma 2.2 we know that X is a mincut.

Property (2). Fix i with $1 \leq i \leq \ell - 1$. Since any subchain is a chain, by property (1) we have $A := C_0 \cup \cdots \cup C_i$ and $B := C_i \cup C_{i+1} \cup \cdots \cup C_\ell$ are two mincuts. By Lemma 2.2, there is no edge going from $A \cap B = C_i$ to the outside $V \setminus X$.

Property (3). For any $0 \leq i < \ell$, we focus on the term $\mathcal{C}(C_i, C_{i+1})$ by the following: let $A = V \setminus (C_i \cup C_{i+1})$, since all of C_i, C_{i+1}, and $C_i \cup C_{i+1}$ are mincuts, we know that any two terms of $\mathcal{C}(A, C_i)$, $\mathcal{C}(C_i, C_{i+1})$, and $\mathcal{C}(C_{i+1}, A)$ add up to the value λ. Thus, each of them must be exactly $\lambda/2$. To show that $\mathcal{C}(C_0, V \setminus X) = \lambda/2$ we apply the same argument to the three mincuts C_0, $C_1 \cup \cdots \cup C_\ell$ and X. Similar argument for $\mathcal{C}(C_\ell, V \setminus X) = \lambda/2$.

To prove the second statement, we assume $i + 1 < j$ and use Lemma 2.2 on mincuts $A := C_i \cup \cdots \cup C_{j-1}$ and $B := C_{i+1} \cup \cdots \cup C_j$. Since $|i - j| > 1$, we know that A crosses with B and hence $\mathcal{C}(C_i, C_j) = \mathcal{C}(A \setminus B, B \setminus A) = 0$.

Property (4). Without loss of generality assume $i < \ell$ (otherwise we simply reverse the chain.) Assume for the contradiction that the chain certificate of C_i is not maximal. Without loss of generality there exists an edge e such that $X_e = C_{\ell'}' \cup C_{\ell'+1}'$. Moreover, since C_i is part of a chain, there exists an edge f so that $X_f = C_i \cup C_{i+1}$. By property (2), we know that one of f's endpoints must be in C_0' (i.e., cannot be leaving from $C_{\ell'}'$). By property (3), the other endpoint of f should not occur in $C_{\ell'+1}'$. So both endpoints of f do not occur in X_e.

If X_e and X_f cross, by the crossing lemma (Lemma 2.2), $X_f \setminus X_e$ is a smaller mincut that contains f, a contradiction.

If X_e and X_f do not cross, then $X_e \subseteq X_f$. Notice that now $Y := C_i \cup X_e$ is a mincut that crosses with C_{i+1}. By the crossing lemma again (Lemma 2.2) we know that $\mathcal{C}(Y \setminus C_{i+1}, C_{i+1} \setminus Y) = 0$. However, f is an edge that goes from $C_i \setminus X_e$ ($\subseteq Y \setminus C_{i+1}$) to $C_{i+1} \setminus X_e$ ($\subseteq C_{i+1} \setminus Y$), a contradiction too.

Property (5). We first show that $C_0 \subseteq X_e$: suppose not, C_0 cross with X_e (since e has an endpoint in C_0). Let f be an edge so that $X_f = C_0 \cup C_1$. Notice that X_e and X_f cross and so $X_f \setminus X_e$ is a smaller mincut. Observe that $X_f \setminus X_e$ contains both endpoints of f because, by Lemma 2.2, there is no edge from $X_e \cap C_0$ to $C_1 \subseteq (V \setminus X_e \cup C_0)$. This contradicts to the fact that X_f is the minimal mincut of f.

Now that $C_0 \subseteq X_e$, we assume for the contradiction that X_e contains strictly more than C_0 but not entire X. Then, X_e cross with $A := C_1 \cup C_2 \cup \cdots \cup C_\ell$. By Lemma 2.2 $X_e \setminus A$ is also a mincut. Now $X_e \setminus A$ is a smaller mincut that contains e, a contradiction too.

Therefore, X_e contains either entire X or just C_0 but none of vertices from $X \setminus C_0$. □

As we will see, each chain actually corresponds to a further-from-root part of a cycle on a cactus.

Now, Gabow's characterization of mincuts shows that every mincut has a certificate:

Lemma A.7 ([Gab16], Lemma 4.4 and Lemma 4.5)**.** *Every mincut on G has either a vertex certificate, an edge certificate, or a chain certificate.*

Proof. Assume G is connected (so $\lambda > 0$). Let X be a mincut (so the induced subgraph $G[X]$ is connected). We prove by induction on the size of X.

Base Case. When $|X| = 1$, X trivially has a vertex certificate.

Inductive Case. Now suppose $|X| > 1$. If X has a vertex/edge certificate then we are done.

Suppose X has no vertex/edge certificate. Consider the collection $\mathcal{X}$ that consists of all maximal proper subsets of X that are mincuts. Then, since $X_v \subsetneq X$ for all $v \in X$, we know that $\mathcal{X} \neq \emptyset$. Take any $A \in \mathcal{X}$. By induction hypothesis, A has a certificate denoted as $(C_0, \ldots, C_\ell)$. For convenience, if A has a vertex certificate or an edge certificate, we use the same notation where $C_0 := A$ and $\ell = 0$.

Finally, we claim that we can actually attach the entire $X \setminus A$ to either end of the certificate, forming an authentic chain $(C_0, \ldots, C_\ell, X \setminus A)$ or $(X \setminus A, C_0, \ldots, C_\ell)$ and conclude the proof. Since $G[X]$ is connected, there is at least one edge $e \in E(A, X \setminus A)$. Let X_e to be the minimal mincut of e. We know that $X_e \subsetneq X$ because X has no edge certificate. Since $X_e \cap A \neq \emptyset$ but X_e is not a superset of A (since A is maximal), we know that X_e cross A. By Lemma 2.2, both $X_e \setminus A$ and $X_e \cup A$ are mincuts. Now, we notice that $X_e \cup A$ is a superset of A. This implies that $X_e \cup A = X$ and $X_e \setminus A = X \setminus A$ is a mincut.

By property (2) of Lemma A.6, e must leave from C_0 or C_ℓ. By property (5) of Lemma A.6, with the clue $X_e \neq X$, we know that either $X_e = (X \setminus A) \cup C_0$ or $X_e = (X \setminus A) \cup C_\ell$. Hence, by definition, one of $(X \setminus A, C_0, \ldots, C_\ell)$ or $(C_0, \ldots, C_\ell, X \setminus A)$ is a chain certificate of X. □

The following Lemma A.8 shows that chain certificates of a mincut are basically unique. This is helpful in the following sense: suppose we have extended the chain certificate of a mincut A to some longer chain (let $B \supset A$ be the associated mincut). Then, we do not need to store A in the memory because the mincut A is "preserved" (as a subchain) in any chain certificate of B.

Lemma A.8 (Briefly mentioned in [Gab16], page 36)**.** *Let X be a mincut that has a chain certificate $(C_0, C_1, \ldots, C_\ell)$. Then this certificate is unique up to reversing the chain.*

Proof. Let $e = (u, v)$ be any edge from the boundary $E(V \setminus X, C_0)$ with $v \in C_0$. (The existence of e is guaranteed by property (2) of Lemma A.6.) Let $(C'_0, C'_1, \ldots, C'_{\ell'})$ be any maximal chain certificate of X. Without loss of generality we have $v \in C'_0$ (reverse the chain if necessary).

Now, let i be the smallest index such that $C_i \neq C'_i$. It is easy to see that $i = 0$: otherwise there are two edges e and f such that $X_e = C_{i-1} \cup C_i$ and $X_f = C_{i-1} \cup C'_i$. By property (3), the endpoints of both e and f belongs to $C_i \cap C'_i$. We obtain a contradiction as $C_{i-1} \cup (C_i \cap C'_i)$ is a smaller mincut for either e or f (or both). Thus, $C_0 \neq C'_0$ but $C_0 \cap C'_0 \neq \emptyset$. Depending on whether or not C_0 crosses with C'_0, there are two cases now:

Case 1: If C_0 crosses C'_0, this implies a contradiction to Lemma 2.2 because $\mathcal{C}(C_0 \cap C'_0, V \setminus (C_0 \cap C'_0)) = 0$ but the edge e connects $C_0 \cap C'_0$ and $V \setminus X \subseteq V \setminus (C_0 \cap C'_0)$. Therefore $C_i = C'_i$ for all i and the chain is unique.

Case 2: If C_0 and C'_0 do not cross, then without loss of generality we may assume $C_0 \subset C'_0$. Suppose there exists an index $i > 0$ such that C'_0 crosses with C_i and let i to be the smallest one among all such indices. Then, for each index j such that $0 \leq j < i$, we know that either $C_j \subsetneq C'_0$ or $C_j \cap C'_0 = \emptyset$. However the latter case is impossible: notice that $C_0 \subseteq C'_0$ and $C_i \cap C'_0$, by property (3), C_0 and $C_i \cap C'_0$ are not connected in C'_0. By property (1) C'_0 should be a mincut and hence must be connected, a contradiction.

Thus, C'_0 should contain all the C_j for $j < i$, and there exists an edge e such that $X_e = C_{i-1} \cup C_i$. First observe that C'_0 doesn't contain both endpoints of e, otherwise $C'_0 \cap (C_{i-1} \cup C_i)$ will be a smaller mincut containing e for both cases that C'_0 crossing with $C_{i-1} \cup C_i$ and $C'_0 \subset C_{i-1} \cup C_i$. Then the edge e connects $C'_0 \setminus C_i$ and $C_i \setminus C'_0$ but this contradicts to Lemma 2.2 for $\mathcal{C}(C'_0 \setminus C_i, C_i \setminus C'_0) = 0$ since C'_0 crosses C_i. Therefore C'_0 must be a subchain $(C_0, C_1, \ldots, C_k)$ of the chain $(C_0, C_1, \ldots, C_\ell)$, but this contradicts to property (4) of Lemma A.6 since it is not a maximal chain. $\square$

A.4   Reducing Cactus to Hierarchical Representation of Global Mincuts

The goal of this section is to reduce the problem of cactus construction to constructing a *hierarchical representation* H of mincuts based on chain certificates defined in Appendix A.3.[11]

Lemma A.9 ([Gab16], Lemma 4.6 (i)). *Consider the collection $\mathscr{C}$ of all mincuts X such that either all chain certificates of X are maximal, or X has no chain certificate. Then $\mathscr{C}$ is laminar.*[12]

Proof. We first show that for any two mincuts A and B that cross each other, then both A and B have a chain certificate. Suppose A does not have a chain certificate, by Lemma A.7 we know that A has a vertex certificate ($A = X_v$) or an edge certificate ($A = X_e$). If $A = X_v$ for some $v \in V$ then

---

[11]Gabow [Gab16] defined special directed graphs based on partial order sets called *chain-trees*. However, the chain-trees are not uniquely defined. Gabow showed that a specific chain-tree can be algorithmically created based on a particular laminar collection of mincuts. Here we define the hierarchical representation H based on exactly the same collection as Gabow's. There are two benefits introducing H: (1) the representation itself is based purely on the structural property of G, not algorithmically, and (2) with the preprocessing mentioned in Appendix A.1, the representation is unique up to reversing the chains in the chain certificates.

[12]The collection defined corresponds to $\mathcal{F}^-$ in [Gab16].

by the crossing lemma either $A \setminus B$ or $A \cap B$ is a smaller mincut that contains v, a contradiction. If $A = X_e$ then e must have one endpoint in $A \setminus B$ and another in $A \cap B$, otherwise one of the two sets contains e, which contradicts minimality of X_e. But this implies that $(A \setminus B, A \cap B)$ is a chain certificate for A, which is also a contradiction. The same proof applies to B.

Now, we show that if two mincuts A and B that cross each other, then none of A and B has a maximal chain certificate. Consider any edge $e \in E(A \cap B, B \setminus A)$. Since both endpoint of e are in B, we know that $X_e \subseteq B$. Now, since A has a chain certificate, and X_e does not contain entire A, by property (5) from Lemma A.6 we know that e is an edge that extends the chain certificate of A. Hence, there is a chain certificate of A that is not maximal. Same proof for B.

To conclude the statement, we notice that if there are two mincuts A and B crossing each other, then both A and B does not appear in $\mathscr{C}$. So $\mathscr{C}$ must be laminar. $\qquad\square$

As a sanity check we note that a mincut with a vertex certificate has no chain certificate, so $X_u \in \mathscr{C}$ for all $u \in V$. Since $\mathscr{C}$ is laminar on V, we know that $|\mathscr{C}| = O(n)$ and the total length of chain certificates from mincuts in $\mathscr{C}$ is also $O(n)$.

The Hierarchical Representation. Let $\mathscr{C}$ be the collection described in Lemma A.9. The collection $\mathscr{C}$ naturally defines a *hierarchy* tree H: for each mincut $A \in \mathscr{C}$ there is a node v_A in H. Moreover, the children of v_A in H are all nodes v_B such that $B \in \mathscr{C}$ is the maximal proper subset of A.

According to the definition of H and Lemma A.8, the hierarchy H is unique, but the chain certificates are unique up to reversion. In particular, Lemma A.8 implies that every mincut X on the graph G can be "captured" by H: if X does not have a chain certificate, then $X \in \mathscr{C}$. Otherwise, X has a chain certificate. By extending this chain certificate to a maximal chain, we know that there exists a mincut $Y \in \mathscr{C}$ such that the chain certificate of X occurs as a subchain to Y's maximal chain certificate.

We finish this subsection by showing that a hierarchy H can be easily turned into a cactus. Hence, once a hierarchy H is formed with all certificates given, a cactus representation of graph G can be constructed in linear time.

Lemma A.10 ([Gab16], Section 4.5)**.** *Given a graph $G = (V, E)$ and its corresponding hierarchy H with certificates in $O(n)$ total size, a cactus representation of G can be constructed in $O(n)$ time.*

Proof. A cactus graph P can be constructed from the hierarchy H as follows. For each node v_A on H where the corresponding mincut $A \in \mathscr{C}$ has a (maximal) chain certificate $(C_0, C_1, \ldots, C_\ell)$. By property (4) of Lemma A.6, all parts C_i either have a maximal chain certificate or a vertex/edge certificate. Any superset of C_i that is proper in A has a chain representation but they are always extendable. Thus, for each C_i there is a corresponding node v_{C_i} in H and they are children of v_A. We replace this star $\{(v_A, v_{C_i})\}_{i=0}^{\ell}$ by a cycle of length $\ell + 1$: $(v_A, v_{C_0}, v_{C_1}, \ldots, v_{C_\ell})$.

After all the replacement are done, it is straightforward to verify that P is a cactus. Now we assign vertices V to nodes in P. Since $X_u \in \mathscr{C}$ for all $u \in V$, by simply assign each vertex $u \in V$ to the node v_{X_u} then we are done.

All mincuts are preserved: a mincut with a chain certificate can be found by cutting two edges from the replaced cycle. A mincut A without a chain certificate can be found by cutting the edge from v_A with its parent on P. On the other hand, cutting a bridge or a pair of edges in the same cycle corresponds to a mincut too. $\square$

Now the task of constructing a cactus representation reduces to computing a hierarchy representation H.

A.5 Constructing a Hierarchical Representation

In this section we describe an algorithm that constructs the hierarchy H defined in Appendix A.4.[13]

Assume that the graph has been preprocessed as described in Appendix A.1 such that every vertex has a distinct minimal mincut. The algorithm constructs the hierarchy by processing all vertex/edge-certificated mincuts in the non-decreasing order of their sizes.

Two variables are introduced explicitly: a (partial) collection $\mathscr{C}$ and a (partial) hierarchy forest H. $\mathscr{C}$ is initialized as an empty set and H is initialized as an empty forest. At any moment, the algorithm maintains a collection $\mathscr{C}$ of disjoint mincuts with corresponding certificate. In the meantime, the algorithm maintains a hierarchy forest H with the invariant such that there is always a bijection between $\mathscr{C}$ and the roots of all trees in H. Throughout processing the mincuts, two subsets in $\mathscr{C}$ may be merged (so the corresponding trees in H may be merged at the root.) Once a mincut $A \in \mathscr{C}$ is removed, we guarantee that some superset B of A is added to $\mathscr{C}$ and v_A becomes a child of v_B in H.

Before we describe the steps of the algorithm, we state the most important property that leads to the correctness of the algorithm.

Lemma A.11 (Inductive Correctness Guarantee). *Upon processing a mincut of size t, any mincut of size strictly less than t can be uniquely "represented" in H. That is, let X be a mincut with $|X| < t$. If X has a chain certificate C, then there is a unique node v_A in H with a chain certificate containing C as a subchain. If X does not have a chain certificate, then X has a corresponding node $v_X \in H$.*

This property explains the validity of certain steps in our algorithm. As you can see, by Lemma A.11, at the end the algorithm returns a correct hierarchy representation H defined in Appendix A.4. Now we describe this high-level algorithm.

Let $\mathcal{L}$ be the list of minimal mincuts of vertices and edges. The mincuts in $\mathcal{L}$ is sorted by their size in increasing order. The algorithm processes mincuts in $\mathcal{L}$ one by one. Let $X \in \mathcal{L}$ be the current processing minimal mincut. If there is already a mincut $A \in \mathscr{C}$ such that $X \subseteq A$, then the algorithm does nothing. Now we assume the otherwise: X is not contained in any mincut in $\mathscr{C}$. X could be a minimal mincut of a vertex $u \in V$, or a minimal mincut of an edge $e \in E$.

Suppose that $X = X_u$ has a vertex certificate. Using the crossing lemma (Lemma 2.2), we deduce that the minimal mincut X_u does not cross with any mincut. In particular, X_u does not

---

[13]Our algorithm is simpler than Karger and Panigrahi's algorithm because we do not reduce the problem into one-layer cactus construction problems.

cross any mincut in the current $\mathscr{C}$. In this case, we remove any mincut that is a subset of X_u from $\mathscr{C}$ and add X_u to $\mathscr{C}$. The hierarchy forest H is updated accordingly, by creating a node v_{X_u} and assign the nodes v_A to be its children for all $A \subseteq X_u$ that was removed from $\mathscr{C}$.

Suppose that $X = X_e$ has an edge certificate $e = (u, v) \in E$. The algorithm checks if this edge helps extending or creating a chain. We observe that right now u and v must belong to different mincuts in $\mathscr{C}$, otherwise there is already a mincut $A \in \mathscr{C}$ containing X. Let the mincuts $A, B \in \mathscr{C}$ such that $u \in A$ and $v \in B$. If X_e crosses with one of the mincut, say A, then we must have $X \setminus A$ be a mincut. By Lemma A.11, since $|X \setminus A| < |X|$, we know that the mincut $X \setminus A$ must have already been represented in the current hierarchy H, which implies that there exists a mincut in the current collection $\mathscr{C}$ that contains $X \setminus A$. Since $v \in X \setminus A$ and by the assumption $v \in B \in \mathscr{C}$, we know that $X \setminus A \subseteq B$.

Therefore, only three cases can occur between X and $A \cup B$: either $X = A \cup B$, $X \subsetneq A \cup B$, or $X \supsetneq A \cup B$. In the first case where $X = A \cup B$, we know that a new chain is formed. We create a new node v_X and set v_A and v_B be its children in H. The certificate of v_X will be a chain certificate (A, B). In the second case where $X \subsetneq A \cup B$, we know that some chain can be extended (or possibly, two chains are concatenated.) Without loss of generality we assume that X cross with A. Now there will be two sub-cases, either $X \setminus A = B$ or $X \setminus A \subsetneq B$. If $X \setminus A = B$ then on the hierarchy H we rename v_A to be $v_{A \cup B}$ and then put v_B as a new child of $v_{A \cup B}$. If $X \setminus A \subsetneq B$, we know that e concatenates the two chains of A and B. On the hierarchy H it suffices to merge the two trees rooted at v_A and v_B, and update the certificate of $v_{A \cup B}$. In the third case where $X \supsetneq A \cup B$, similar to the vertex case, the algorithm removes all mincuts that are contained in X. The hierarchy H is updated accordingly.

We summarize this high-level algorithm in Algorithm 4.

Algorithm 4 High-Level Approach for Constructing a Hierarchy Representation

---

Require: A graph $G = (V, E)$, tree packing $\mathcal{T}$, labels of minimal mincuts of all vertices and edges.
Ensure: A hierarchy tree $H = (V_H, E_H)$. Every node on H has a certificate.

1: $\mathcal{L} \leftarrow$ the list of minimal mincuts of all vertices and edges, sorted by their sizes.
2: $\mathscr{C} \leftarrow \emptyset$.   ▷ A disjoint collection of mincuts.
3: $H \leftarrow \emptyset$.   ▷ A hierarchy forest with certificates.
4: **for** each vertex/edge minimal mincut $X \in \mathcal{L}$ **do**
5:      **if** there does not exist $A \in \mathscr{C}$ such that $X \subseteq A$ **then**
6:          **if** X has a vertex certificate $u \in V$ **then**
7:              `AddNestingSuperset`(X_u).
8:          **else** (now X has an edge certificate $e = (u, v) \in E$)
9:              Let $A, B \in \mathscr{C}$ such that $u \in A$ and $v \in B$.
10:              **if** $X = A \cup B$ **then**
11:                  `AddNewChain`(X_e, A, B).   ▷ Create a new chain (A, B).
12:              **else if** $X \subsetneq A \cup B$ **then**
13:                  Swap the role of A and B so that A cross with X.
14:                  **if** $X \setminus A = B$ **then**
15:                      `ExtendChain`(X_e, A, B).   ▷ Extend A's chain by adding B at the end.
16:                  **else** (now $X \setminus A \subsetneq B$)
17:                      `ConcatChains`(X_e, A, B).   ▷ Concat A and B's chain certificates via X_e.
18:              **else** (now $X \supsetneq A \cup B$)
19:                  `AddNestingSuperset`(X_e).

Correctness Proof. We end this subsection by proving Lemma A.11, which implies that Algorithm 4 does produce a correct hierarchy H after processing all minimal mincuts of vertices and edges.

Proof of Lemma A.11. Let's apply mathematical induction on t and let A be a mincut on G of size $|A| < t$. First, since a chain certificate can only be extended or concatenated, it is straightforward to see that if A is added to $\mathscr{C}$, A will be "represented" in any future moment. Notice that whenever A has a vertex certificate or an edge certificate, either A will be added to $\mathscr{C}$ or is used for extending/concatenating a chain.

Now, assume that A has a chain certificate $(C_0, C_1, \ldots, C_\ell)$ with $\ell \geq 1$. Since $|A| < t$, for any $0 \leq i < \ell$ we know that $|C_i \cup C_{i+1}| < t$ as well. Since $C_i \cup C_{i+1}$ has an edge certificate, we know that $C_i \cup C_{i+1}$ must have been processed already for all i. If $\ell = 1$, then Line 11 correctly construct a chain. If $\ell > 1$, then the mincuts $A_l := C_0 \cup \cdots \cup C_{\ell-1}$ and $A_r := C_1 \cup \cdots \cup C_\ell$ can be found uniquely in some node (say v_l and v_r respectively) on H, by the induction hypothesis of Lemma A.11. Since the node that represents C_1 can also be uniquely found as a child of both v_l and v_r, we conclude that $v_l = v_r$. Therefore, A can be found uniquely in $v = v_l = v_r$ too. $\qquad \square$

A.6   Efficient Implementation of Algorithm 4

Once the high-level idea is confirmed, the remaining parts of the implementation become relatively easier tasks. There may be different ways to implement Algorithm 4, and we give one of them in this subsection.

For $\mathscr{C}$ the algorithm maintains an additional disjoint set data structure (with UNION and FIND operations). For any mincut $A \in \mathscr{C}$, we store (1) its size $|A|$, (2) one vertex $v \in A$, and (3) certificates of A. If there are multiple certificates available for the same mincut, we store one certificate of each kind: vertex, edge, and chain. For a chain certificate $(C_0, C_1, \ldots, C_\ell)$, we assume that a doubly linked list of edges $(e_1, e_2, \ldots, e_\ell)$ is stored in the memory where $X_{e_i} = C_{i-1} \cup C_i$ for all $1 \leq i \leq \ell$. That said, the operations to chains (e.g., `AddNewChain` in Line 11, `ExtendChain` in Line 15, and `ConcatChains` in Line 17) can be implemented in a straightforward way in $O(1)$ time).

Containment Queries.   In Algorithm 4, the algorithm is often required to test whether two given mincuts A and B satisfies $A \subseteq B$. (Specifically, this operation is used to implement Line 5, Line 10, and Line 14.) This test is denoted by Karger and Panigrahi [KP09] as the *containment query*: CONTAINMENT(A, B) returns true if and only if $A \subseteq B$. In [KP09] the authors merely mentioned that the containment queries can be answered in $O(1)$ by answering LCA queries in the corresponding tree. The authors did not describe an algorithm that answers containment queries — it becomes highly non-trivial when A or B has a chain certificate. Fortunately, thanks to the crossing lemma, most of the containment queries in the high-level Algorithm 4 can be implemented by simply checking and comparing the sizes of the mincuts. In below, we describe the detailed implementations line by line.

Line 1.   First of all, Line 1 in Algorithm 4 can be done efficiently in $O(m+n|\mathcal{T}|)$ time, by computing the sizes of the mincuts in $O(n|\mathcal{T}|)$ time and performing a bucket sort in $O(m+n)$ time. Notice that in order to correctly implement Line 19, we require that the same minimal mincuts are listed together. This can be achieved by, e.g., breaking ties using the lexicographical order of the cut labels. Moreover, for the same mincut we process vertices first then the edges.

Line 5.   Since all mincuts in $\mathscr{C}$ are disjoint and they are only replaced by supersets, it suffices to use a standard disjoint set data structure supporting membership queries. In particular, FIND(x) returns the mincut in $\mathscr{C}$ that contains x, or $\perp$ if such mincut does not exist.

To implement Line 5, if $X = X_u$ is a minimal mincut of vertex u, then we know that FIND$(u) = \perp$, and that the **if** statement is always evaluated to true.

If $X = X_e$ is a minimal mincut of an edge $e = (u, v)$. We observe that there exists $A \in \mathscr{C}$ that contains X if and only if FIND$(u) =$ FIND(v).

Line 10.   Let $X = X_e$ with $e = (u, v)$. Suppose now that FIND$(u) \neq$ FIND(v) and there are two mincuts $A, B \in \mathscr{C}$ with $u \in A$ and $v \in B$. There are only three cases to distinguish: $X = A \cup B$, $X \subsetneq A \cup B$, and $X \supsetneq A \cup B$. Since $A \cap B = \emptyset$, it suffices to compare the size $|X|$ with $|A| + |B|$.

Line 13.   To test whether $A \subseteq X$, we utilize the cut label of X and the certificate of A. Let $(type, v, w, T)$ be the cut label of X. Using a prebuilt data structure on T it is easy to check

43

whether a vertex belongs to X in $O(1)$ time.

If $A = X_u$ has a vertex certificate, we know that $A \subseteq X$ for sure by definition of minimal mincut of u. If $A = X_{e'}$ has an edge certificate $e' = (u', v')$, then definition of $X_{e'}$ we know that $A \subseteq X$ if and only if $u' \in X$ and $v' \in X$, and this can be tested in $O(1)$ time. If A has a chain certificate $(C_0, C_1, \ldots, C_\ell)$, we know that by property (5) from Lemma A.6, A cross with X if and only if exactly one of $\{C_0, C_\ell\}$ is contained in X but the other one does not. To test so, it suffices to choose an arbitrary vertex from each of C_0 and C_ℓ and test whether or not it belongs to X. This can be done in $O(1)$ time too.

Line 14. To test whether $X \setminus A = B$ or not, it suffices to check again if $B \subseteq X$ or B cross with X. This can be achieved as described above (implementation of Line 13).

`AddNestingSuperset`. It is a bit challenging if we want to search for all mincuts that are currently in $\mathscr{C}$ that is contained in the given mincut X — enumerating all vertices in X and then using the disjoint set data structure takes too much time!

To cope with this, we handle minimal mincuts for vertices and edges differently, and describe the implementation details below.

Vertex Case (Line 7). Let $X = X_v$ be the minimal mincut of a vertex $v \in V$. A cool trick is, we can implement this step utilizing the nesting relation tree $\hat{T}$ defined in Appendix A.2, which requires $O(n|\mathcal{T}|)$ preprocessing time by Lemma A.1.

Let $C_v = \{u_1, u_2, \ldots\}$ be the children of v on $\hat{T}$. We notice that upon processing X_v, all minimum mincuts of u_i must have been processed already. Hence, by Lemma A.11, for each u_i there exists some mincut $A_i \in \mathscr{C}$ that contains X_{u_i}. Therefore, it suffices to query the disjoint set data structure $|C_v|$ times to identify all mincuts that are covered by X.

Since $\hat{T}$ is a tree, there will be exactly $n - 1$ FIND calls and at most $n - 1$ UNION calls to the disjoint data structure in total.

Edge Case (Line 19). Unfortunately all minimal mincuts of edges does not have a hierarchy representation as $\hat{T}$ in the vertex case, so the method we use for the vertex case does not apply to this edge case[14]. However, this case can be solved easily by making sure we process a bunch of identical minimal mincuts at a time. Consider the set F of *all* edges f such that $X_f = X_e$. We claim that the sub-collection of mincuts that contain endpoints to any $f \in F$ is exactly the set of all mincuts to be subsumed. The "$\Leftarrow$" direction is trivial, and the "$\Rightarrow$" direction is true because $G[X]$ is connected, and any edge $f \in F$ connecting these mincuts whose minimal mincut has not been processed yet must have $X_f = X_e$.

In conclusion, we successfully proved Lemma 3.2 by providing an algorithm that constructs a cactus in $O(m\alpha(m, n) + n|\mathcal{T}|)$ time. Notice that it is linear time on a not-so-sparse graph whenever $m = \Omega(n|\mathcal{T}|)$ and $|\mathcal{T}| = \Omega(\log n)$.

---

[14]No pun intended.

B Minimal Mincuts of Vertices: Proof of Lemma 3.4

B.1 On the Missing Case in [KP09]

In [KP09], the key subroutine is computing the minimal mincut for vertices. Under their framework, they compute Type 1, Type 2-Comparable and Type 2-Incomparable minimal 2-respecting mincut for vertices. We believe that their approach is correct, but there seems to be a missing case for computing the minimal incomparable 2-respecting mincut of vertices.

In one of the cases where they compute the minimal incomparable mincut partner for each vertex v (corresponds to Lemma B.2), they define the "outermost" minimal minprecut parter of v. In Section 3.1 of [KP09], two copies of the tree S and T are maintained. One of them is the shrunk tree S where the algorithm contracts processed boughs and produces the bough decomposition. On the other hand, the algorithm does not shrink T (see [KP09, Figure 2]). The outermost minprecut is then defined and computed on the uncontracted tree T. Upon processing the bough $(v_1, v_2, \ldots, v_k)$ where v_1 is the lowest vertex, the algorithm examines all edges incident to any vertex in $v_k^\downarrow$ with a postorder traversal, and then dynamically maintains the outermost minimal minprecut partner. However, in the case where the outermost partner lies in $v_i^\downarrow \setminus v_{i-1}^\downarrow$, this partner may become invalid once the algorithm visits v_i. Their algorithm did not describe how to update the "outermost" partner correctly in this case.

An Example. Recall the definition of incomparable precut value $\mathcal{C}_v^\perp(w) = \mathcal{C}(w^\downarrow) - 2\mathcal{C}(v^\downarrow, w^\downarrow)$. Consider the graph in Figure 5 and the green spanning tree T, we have

$$\mathcal{C}_u^\perp(w') = \mathcal{C}(w'^\downarrow) - 2\mathcal{C}(u^\downarrow, w'^\downarrow) = 4 - 2 \times 3 = -2 \ .$$
$$\mathcal{C}_u^\perp(w) = \mathcal{C}(w^\downarrow) - 2\mathcal{C}(u^\downarrow, w^\downarrow) = 3 - 2 \times 2 = -1 \ .$$
$$\mathcal{C}_v^\perp(w') = \infty \ . \qquad \text{(since } w' \parallel v)$$
$$\mathcal{C}_v^\perp(w) = \mathcal{C}(w^\downarrow) - 2\mathcal{C}(v^\downarrow, w^\downarrow) = 3 - 2 \times 2 = -1 \ .$$

Figure 5: A Missing Case.

Therefore, w is the unique outermost minimal minprecut of v, and w' is the unique outermost minimal minprecut of u. But there doesn't exist any edge between $X = v^\downarrow \setminus u^\downarrow = \{v, w'\}$ and $w^\downarrow$, which turns out to be a missing case for Lemma 3.4 of [KP09].
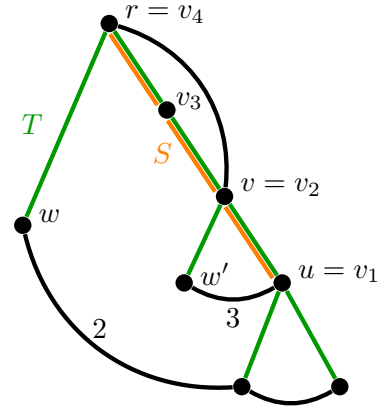
Note that maintaining the minprecuts on the uncontracted tree T instead of the shrunk tree S is necessary. The reason is, it is possible to have a vertex v whose all minprecut partners are processed in boughs of earlier phases. However, if the algorithm shrunk the bough after processing it, the minprecut values at vertices on that bough are no longer available.

In the next subsection, we provide a simpler and complete algorithm that uses a completely different approach compared to [KP09].

B.2 Our Algorithm

A natural question is this: can we add a self-loop on each vertex and reduce the problem of computing minimal mincut for vertices to computing minimal mincut for edges? The reason we cannot prove this way is that, for the incomparable case, the direction of the reduction is actually opposite. Recall that, Lemma 6.3 computes the minimal incomparable 2-respecting mincut candidates for vertices, and we use it to prove Lemma 6.1 which computes minimal incomparable 2-respecting candidates for edges. Therefore, proving Lemma 6.3 will be one of the main tasks in this section. The key insight is again exploiting the structural property of 2-respecting mincut and using top-tree to find the minimal one. (See the proof of Lemma B.2 at the end of this subsection.)

On the other hand, we can use the self-loop idea to compute the minimal comparable 2-respecting mincut candidates for vertices using Lemma 5.1, which does not involve in a circular proof.

Corollary B.1. *There is an algorithm that, given a spanning tree T of $G = (V, E)$, in total time $O(m \log^2 n)$ computes, for every vertex $u \in V$ an comparable 2-respecting minimal mincut candidate $f(u) = (v_u, w_u)$ or null with the following guarantee:*

If there exists comparable 2-respecting cut that separating u from root r, then $v_u^{\downarrow} \setminus w_u^{\downarrow}$ is such a mincut with smallest size.

Proof. Given a graph $G = (V, E)$, we insert a self-loop edge e_u on each vertex $u \in V$. Denote the new graph as $G' = (V, E')$. By Lemma 5.1, given the graph G' and the spanning tree T, there is an algorithm computing a comparable 2-respecting minimal mincut candidate $f(e) = (v_e, w_e)$ for each edge $e \in E'$ with the following guarantee: if the minimal mincut of e is a comparable 2-respecting cut of T, then $v_e^{\downarrow} \setminus w_e^{\downarrow}$ is the minimal mincut of e. Therefore, for each vertex $u \in V$, the comparable 2-respecting minimal mincut candidate of u can be set as $f(u) = f(e_u)$, since the minimal mincut of u is the same as the minimal mincut of e_u. $\square$

We shall prove Lemma 3.4 which computes minimal 2-respecting mincut candidates for vertices. The proof is essentially the same as Section 7, except that we use the corresponding subroutines computing each type of 2-respecting minimal mincut for vertices.

Proof of Lemma 3.4. Given a spanning tree T, there are three types of 2-respecting cut of T. For each type, we compute the minimal mincut candidates for vertices.

Type 1. By Lemma 3.7, the algorithm computes the minimal 1-respecting mincut candidate for every vertex v.

Type 2-Comparable. By Corollary B.1, the algorithm computes the minimal comparable 2-respecting mincut candidates for every vertex v.

Type 2-Incomparable. By Lemma 6.3, the algorithm computes the minimal incomparable 2-respecting mincut candidates for every vertex v.

For each vertex v, we get three minimal mincut candidates as above. If all of the three candidates are null, then the minimal 2-respecting mincut of v respects to T is null. Otherwise, the minimal

46

2-respecting mincut of v respects to T is the mincut with the smallest size among these minimal mincut candidates.

Since the algorithm for each case runs in $O(m \log^2 n)$ time and the comparing time is constant for each vertex, the whole algorithm runs in $O(m \log^2 n)$ total time. $\qquad\square$

In the rest of this section, we focus on proving Lemma 6.3, which exploits the following lemma as a key subroutine.

Lemma B.2. *There is an algorithm that, given a graph $G = (V, E)$ and a spanning tree $T \in \mathcal{T}$, in total time $O(m \log^2 n)$ computes, for every vertex $v \in V$, a vertex called* minimal incomparable mincut partner *$r_v \in V \cup \{\mathsf{null}\}$ with the following guarantee: if there exists incomparable mincut partner of v, then r_v is the incomparable mincut partner of v with the smallest subtree size $r_v^{\downarrow}$.*

With Lemma B.2, we can efficiently compute minimal incomparable 2-respecting mincut candidates for vertices, since all the minimal incomparable 2-respecting mincut candidates for vertices are of the form $v^{\downarrow} \cup r_v^{\downarrow}$ for some v. We shall prove Lemma 6.3, and defer the proof of Lemma B.2 to the end of this subsection.

Proof of Lemma 6.3. Observe that if an incomparable 2-respecting mincut contains u, it will also contain all the descendants of u. Hence, the minimal incomparable 2-respecting mincut of u is either the minimal incomparable 2-respecting mincut of the parent of u or the minimal incomparable 2-respecting mincut $u^{\downarrow} \cup r_u^{\downarrow}$. Therefore, we can find the minimal incomparable 2-respecting mincut of all the vertices using a one-time depth-first-search after computing all the r_v in Lemma B.2. $\qquad\square$

The algorithm for Lemma B.2 is the main technical contribution of this subsection. We highlight that top-tree is again the right data structure for exploiting the structural property: using $\mathtt{MinTree}^{\downarrow}$ we can find the partner with minprecut value and break tie by finding the one with smallest subtree size, which meets exactly the criteria of minimal incomparable mincut partner.

Proof of Lemma B.2. We will use the reduction to path from Lemma 4.1. For any $P \in \mathcal{P}$, define $g(e, P) = r_v$ if $v \in P$ and e is the tree edge with v being the lower vertex, otherwise $g(e, P) = \mathsf{null}$.

Given a path $P = (v_1, v_2, \ldots, v_k)$ from the path decomposition with v_1 being the deepest vertex, our algorithm will process v_i starting from $i = 1, 2, \ldots, k$. We will maintain the invariant that once we process the vertex v_i the incomparable precut values $\mathcal{C}_{v_i}^{\perp}(w)$ for all $w \perp v_i$ can be accessed via $val[w]$.

Next we show how to maintain the invariant. In the preprocessing step before the path P was given, we set $val[w] = \mathcal{C}(w^{\downarrow})$ for each vertex $w \perp v_1$ and ∞ for $w \parallel v_1$, which can be computed in $O(m \log n)$ time. Now we start from the deepest vertex v_1, the algorithm needs to add $-2\mathcal{C}(v_1^{\downarrow}, w^{\downarrow})$ to each $val[w]$ so that $val[w] = \mathcal{C}_{v_1}^{\perp}(w)$. To achieve this efficiently, we create a dynamic tree on T (Lemma 4.2). For each edge (u, u') where $u \in v_1^{\downarrow}$, we invoke $\mathtt{AddPath}(u', 2\mathcal{C}(u, u'))$ so that two times the weight of the edge (u, u') is added to $val[w]$ for each $w \in u'^{\uparrow}$.

Then, the algorithm scans through the rest of vertices $v_2, v_3, \cdots v_k$ on the path one by one. Suppose the algorithm reaches v_i now. With the invariant after processing v_{i-1}, it suffices to add

$2\mathcal{C}(v_i^{\downarrow} \setminus v_{i-1}^{\downarrow}, w)$ to $val[w]$ for each $w \perp v_i$ by invoking $\texttt{AddPath}(u', 2\mathcal{C}(u, u'))$ for each edge (u, u') where $u \in v_i^{\downarrow} \setminus v_{i-1}^{\downarrow}$. (These edges can be found in $O(d(v_i^{\downarrow} \setminus v_{i-1}^{\downarrow}))$ time using a DFS from v_i without searching the subtree rooted at v_{i-1}.) Therefore, in $O(d(v_i^{\downarrow} \setminus v_{i-1}^{\downarrow}) \log n)$ time, $val[w]$ are updated to $\mathcal{C}_{v_i}^{\perp}(w)$ for all $w \perp v_i$.

After obtaining $\mathcal{C}_{v_i}^{\perp}(w)$ values, we compute the minimal incomparable mincut partner r_{v_i} by the following dynamic tree operations. We first invoke $\texttt{Cut}(v_i, \text{parent}(v_i))$; then r_{v_i} can be found by $\texttt{MinTree}^{\downarrow}(\text{parent}(v_i))$; finally invoke $\texttt{Link}(v_i, \text{parent}(v_i))$ to restore the tree.

From the discussion above, we have an algorithm that, given a path P, computes $g(e, P)$ for all $e \in E(P^{\downarrow})$ in $O(d(P^{\downarrow}) \log n)$ time. By plugging in the path decomposition Lemma 4.1, we obtain an algorithm that computes r_v of all vertices $v \in V$ in $O(m \log^2 n)$ total time, because the preprocessing time is $t_p = O(m + n \log n)$ for computing $\mathcal{C}(w^{\downarrow})$ and building the dynamic tree. $\square$

C The Algorithm from [KP09] Runs in $\Omega(m \log^4 n)$ Time

Here, we give an explanation of why the algorithm by [KP09] takes $\Omega(m \log^4 n)$ time. For the first log factor, their algorithm randomly generates $\Theta(\log n)$ graphs from the original input graph as follows: in each copy, an edge with weight w is contracted with probability $\min\{\frac{w}{2\lambda}, 1\}$. Note that each contracted graph could still contain $\Theta(m)$ edges even if we increase the contraction probability by any constant (e.g. a complete graph with unit-weight edges).

Their algorithm then spends at least $\Theta(m \log^3 n)$ time on each contracted graph, which is our time-bound. These three log factors come from (1) there are $\Theta(\log n)$ trees in the tree packing, (2) for each tree, there are $\Theta(\log n)$ phases in the bough decomposition, (3) for each phase, we need to process $\Theta(m)$ edges using dynamic tree data structure, each of which takes $\Theta(\log n)$ time. Therefore, in total, the algorithm by [KP09] takes $\Omega(m \log^4 n)$ time, while ours avoids generating the randomized $\Theta(\log n)$ copies of the graph and takes only $O(m \log^3 n)$ time.