

Fully Dynamic Min-Cut of Superconstant Size in Subpolynomial Time

Wenyu Jin*

Xiaorui Sun†

Mikkel Thorup‡

Abstract

We present a deterministic fully dynamic algorithm with subpolynomial worst-case time per graph update such that after processing each update of the graph, the algorithm outputs a minimum cut of the graph if the graph has a cut of size at most c for some $c = (\log n)^{o(1)}$. Previously, the best update time was $\tilde{O}(\sqrt{n})$ for any $c > 2$ and $c = O(\log n)$ [28].

1 Introduction

In the study of dynamic graphs, we consider the scenario that a graph over a fixed vertex set undergoes edge insertions and deletions. For a property of the graph, the goal of a dynamic graph algorithm is to build a data structure that can process graph updates and queries of the graph property efficiently, assuming the updates and queries are presented online, meaning that the algorithm needs to process each update or query without knowing anything about the future.

Update and query time can be categorized into two types: *worst-case*, i.e., the upper bound on the running time of any update or query operation, and *amortized*, i.e., the running time amortized over a sequence of operations. In this paper we focus on worst-case time, so all operation times are assumed to be worst-case unless explicitly specified.

In this paper, we study the fully dynamic minimum c -cut problem for a given integer c , in which the dynamic algorithm needs to output a minimum edge cut with size at most c of the graph if such a cut exists after receiving each update of the graph. The dynamic minimum c -cut problem is a generalization of the dynamic c -edge connectivity problem. The dynamic c -edge connectivity problem is to determine if the edge connectivity of the graph is at least c , where the edge connectivity of a graph is the size of the minimum cut of the graph.

The dynamic minimum c -cut problem has been extensively studied since 1980s, mostly in the context of dynamic edge connectivity problem [5, 9, 8, 30, 13, 4, 6, 14, 16, 15, 27, 17, 20, 31, 21, 23, 24, 32, 18, 3, 1]. The fully dynamic 1-edge connectivity problem is the classic fully dynamic connectivity problem. For a graph with n vertices and m edges, the best known algorithms for the fully dynamic 1-edge connectivity have deterministic polylogarithmic amortized update time by Holm et al. [17], Monte Carlo randomized polylogarithmic worst-case update time by Kapron et al. [20], and $n^{o(1)}$ deterministic worst-case update time by Chuzhoy et al. [3].

The study of fully dynamic 2-edge connectivity dates back to the work by Westbrook and Tarjan [30] in a context of maintaining 2-edge connected components, where the c -edge-connected components of a graph are the maximal induced subgraphs with edge connectivity at least c . Galil and Italiano [9] obtained an algorithm with $O(m^{2/3})$ update time. The update time was improved to $O(\sqrt{m})$ by Frederickson [6], and $O(\sqrt{n})$ by Eppstein et al. [4]. All these running times are worst-case. To date, the best known worst-case update time is still $O(\sqrt{n})$. For the amortized case, polylogarithmic amortized update time algorithm were proposed in [14, 27, 17, 18].

The dynamic c -edge connectivity problem gets much harder for $c > 2$. For $c \leq 2$, two vertices are in the same c -edge connected component iff they are c -edge connected (i.e., the two vertices cannot be disconnected by removing fewer than c edges), but this is not the case for higher c . For example, if two vertices are connected by three edge disjoint paths of length two, then the two vertices are 3-edge connected but the 3-edge connected components are all singleton vertices. For the fully dynamic 3-edge and 4-edge connectivity, $O(n^{2/3})$ and $O(n\alpha(n))$ update time algorithms were given by Eppstein et al. [4]. Thorup [28] presented a deterministic fully dynamic minimum c -cut algorithm for any polylogarithmic c with $\tilde{O}(\sqrt{n})$ time per update. Recently, Goranci et al. gave

*University of Illinois at Chicago. Email: wjin9@uic.edu.

†University of Illinois at Chicago. This project is supported by the National Science Foundation (NSF) grant 2240024. Email: xiaorui@uic.edu.

‡University of Copenhagen. This project is supported by the VILLUM Foundation grant 16582. Email: mikkel2thorup@gmail.com.

two fully dynamic algorithms for arbitrary edge connectivity with $\tilde{O}(n)$ worst-case update time and $\tilde{O}(m^{1-1/16})$ amortized update time respectively [10].

All of the above results are for exact c -edge connectivity. For the fully dynamic approximate edge connectivity problem, Thorup and Karger [29] presented a deterministic $\sqrt{2} + o(1)$ -approximate algorithm for arbitrary edge connectivity in amortized polylogarithmic time per update. Thorup [28] presented a Monte Carlo randomized $(1 + o(1))$ -approximate algorithm in $\tilde{O}(\sqrt{n})$ update time, which only works for oblivious adversary.

On the other hand, a closely related problem to the problem studied in this paper is the dynamic (s, t) c -edge connectivity problem [9, 8, 4, 6, 14, 19]. In the dynamic (s, t) c -edge connectivity problem, graph updates and queries each containing two vertices are presented online so that for each query, the dynamic algorithm needs to determine if the edge connectivity between the two queried vertices is at least c , where the edge connectivity between two vertices in a graph is the size of the minimum cut separating the two vertices. We remark that the dynamic minimum c -cut problem (as well as dynamic c -edge connectivity problem) and the dynamic (s, t) c -edge connectivity problem are related, but do not reduce to each other efficiently. To solve the dynamic minimum c -cut problem, for any vertex s , there exists a vertex t such that the minimum cut separating s and t is a global minimum cut. However, since t could be any vertex in the graph, it is not efficient to use fully dynamic (s, t) c -edge connectivity algorithm to solve the dynamic minimum c -cut problem.

For the fully dynamic (s, t) c -edge connectivity problem, polylogarithmic, $O(\sqrt{n})$, and $O(n^{2/3})$ worst-case update and query time algorithms for $c = 1, 2$, and 3 were presented in [15, 6], and [8] respectively. But it was unknown if there is a sublinear time fully dynamic (s, t) c -edge connectivity algorithm for $c > 3$, until very recently a fully dynamic (s, t) c -edge connectivity algorithm with $n^{o(1)}$ update and query time for any $c = (\log n)^{o(1)}$ was proposed in [19] by Jin and Sun.

An obvious open question is if the fully dynamic minimum c -cut problem can also be solved in subpolynomial update time. In this paper, we give an affirmative answer to this question by proving the following result.

THEOREM 1.1. *For any $c = (\log n)^{o(1)}$, there is a deterministic fully dynamic algorithm for an undirected graph with $n^{o(1)}$ running time per update such that after processing each update of the graph, the algorithm outputs a (global) minimum cut of the graph if the graph has a cut of size at most c , or outputs an empty set if the (global) minimum cut of the graph is of size greater than c .*

1.1 Techniques While the minimum cut of a graph is a global property, we present a localization approach using a vertex partition of the graph so that the process of finding a minimum cut of the graph with cut size at most c is reduced to finding a minimum cut with certain properties among the induced subgraphs on all the vertex sets in the vertex partition.

We first give a static algorithm to illustrate the localization idea, and then show how to make the static algorithm fully dynamic.

1.1.1 A Static c -Edge Connectivity Algorithm via Localization We start with an iterative algorithm for a *static* graph based on the terminal edge connectivity sparsifier, which was proposed in [2, 19]. For simplicity, the algorithm is for the c -edge connectivity problem, ignoring the issue of retrieving a corresponding minimum cut in the dynamic minimum c -cut problem.

The static algorithm illustrates the iterative computation structure, and we will show that the iterative computation structure can be maintained efficiently also for a dynamically changing graph.

Terminal Edge Connectivity Sparsifier We first define terminal edge connectivity sparsifier [2, 19]. Roughly speaking, for a given set of vertices called terminals, the terminal edge connectivity sparsifier is a graph that preserves the minimum cut size for any partition of the terminals. Formally, for a graph G , a terminal vertex set T , and an integer c , the *terminal c -edge connectivity sparsifier* with respect to G and T is a graph containing all the terminals such that for any $\emptyset \subsetneq T' \subsetneq T$, the size of the minimum cut separating T' and $T \setminus T'$ in G is the same as the size of the minimum cut separating T' and $T \setminus T'$ in the sparsifier, if the minimum cut size is smaller than or equal to c . Since a terminal sparsifier only preserves the minimum cut size for terminal partitions, a sparsifier can be potentially much smaller than the original graph in terms of the number of vertices and edges so that the c -edge connectivity between the terminals can be efficiently determined in the sparsifier.

As we are going to apply a vertex partition to the graph, we define the terminal edge connectivity sparsifier with respect to a vertex partition. For a graph G and a vertex partition $\mathcal{P}$ of G , the terminal c -edge connectivity

sparsifier for G and $\mathcal{P}$ takes the boundary vertices of $\mathcal{P}$ as terminals, where the boundary vertices of each cluster (i.e., vertex set in $\mathcal{P}$) are those who have a neighbor in a different cluster. We denote the terminals for G and $\mathcal{P}$ as $T_{G,\mathcal{P}}$. A terminal c -edge connectivity sparsifier for G and $\mathcal{P}$ is constructed as follows: start with G , for each cluster $P \in \mathcal{P}$, replace the induced subgraph on P by a terminal c -edge connectivity sparsifier for the induced subgraph with the boundary vertices in the cluster (i.e., $T_{G,\mathcal{P}} \cap P$) as terminals. One can verify that the sparsifier constructed in this way is a terminal c -edge connectivity sparsifier for G with $T_{G,\mathcal{P}}$ as terminals.

Now, suppose we can construct the terminal c -edge connectivity sparsifier for a given graph and a vertex partition. We are going to discuss how to use it to compute the c -edge connectivity for a graph.

Global c -Edge Connectivity via Localization Suppose we have a vertex partition $\mathcal{P}$ for graph G , and a terminal sparsifier for G and $\mathcal{P}$ that preserves the minimum cut size for any partition of the boundary vertices with respect to $\mathcal{P}$ for some integer c . We show how to compute the c -edge connectivity of G .

The high level idea of our algorithm is to “localize” the computing of c -edge connectivity based on a given vertex partition $\mathcal{P}$ of the input graph G . By localization, we mean that the process of computing the c -edge connectivity of G is reduced to finding a minimum c -cut with certain properties among the induced subgraphs on all the clusters of $\mathcal{P}$. Thus, even though the c -edge connectivity is a global graph property, with the localization, it is sufficient to examine the cuts within the induced subgraph on each cluster, without knowing other clusters.

To achieve the localization, we make use of the terminal c -edge connectivity sparsifier for G and $\mathcal{P}$. We show that if there is a minimum c -cut of G that is a terminal cut (i.e., a cut that partitions terminals into two non-empty sets), then the c -edge connectivity of G is the same as the c -edge connectivity of the terminal sparsifier. Otherwise, all the minimum c -cuts of G are *local cuts*, where a cut C of G is a local cut if it is a non-terminal cut (i.e., a cut such that all the terminals $T_{G,\mathcal{P}}$ are in the same side of the cut), and the cut-set of the cut is contained in the induced subgraph of a cluster in $\mathcal{P}$.

Thus, for a vertex subset $P \in \mathcal{P}$, let the *local c -edge connectivity* for $G[P]$ be the size of the minimum c -local cut in $G[P]$ if any. The c -edge connectivity of G is the minimum of the terminal sparsifier’s c -edge connectivity and the local c -edge connectivity for all the clusters in $\mathcal{P}$.

Furthermore, if the induced subgraph on each cluster has high conductance, then each local cut has a side with subpolynomial volume. Thus the local cuts can be efficiently enumerated, and the local c -edge connectivity in the induced subgraph can be efficiently computed. This occurs when $\mathcal{P}$ is a $1/n^{o(1)}$ -expander decomposition, i.e., every cluster in $\mathcal{P}$ is a $1/n^{o(1)}$ -expander.

Iterative Construction Notice that in order to compute the c -edge connectivity for the graph by the localization, we still need to compute the c -edge connectivity of the sparsifier. Hence, we have an iterative construction starting from the original graph $G^{(0)} = G$. Starting with $i = 0$, we compute an $1/n^{o(1)}$ -expander decomposition $\mathcal{P}^{(i)}$ of $G^{(i)}$ with terminals $T_{G^{(i)},\mathcal{P}^{(i)}}$. Next, we compute the local c -edge connectivity for each cluster of $\mathcal{P}^{(i)}$. We are now done if $\mathcal{P}^{(i)}$ has only one cluster and no terminals. Otherwise, we construct a terminal c -edge connectivity sparsifier $G^{(i+1)}$ of $G^{(i)}$ and $\mathcal{P}^{(i)}$, and then repeat the above process on $G^{(i+1)}$. The c -edge connectivity of G is then the smallest local c -edge connectivity over all the clusters in the decomposition $\mathcal{P}^{(i)}$ for all i .

In this paper, we aim to make use of the above iterative construction of expander decomposition and terminal c -edge connectivity sparsifier to study the fully dynamic minimum c -cut problem. To construct a minimum c -cut corresponding to the smallest local c -edge connectivity found at some cluster P in $\mathcal{P}^{(i)}$, we have to trace how the induced subgraph of $G^{(i)}$ on P was created from the lower $G^{(j)}$ and $\mathcal{P}^{(j)}$ with $j < i$. If the graph is dynamic, the trace of local cuts becomes more complex, as it is affected by the updates for graph, expander decomposition, and terminals throughout the iterative construction.

We remark that Goranci et al. proposed and dynamically maintained the expander hierarchy, which iteratively computes the expander decomposition and contracts each cluster into a single vertex [11]. However, the work by Goranci et al. can only approximate edge connectivity, and cannot compute the exact edge connectivity, which is the issue we aim to address in this paper.

1.1.2 Retrieving A Minimum c -Cut From Terminal Sparsifier Our fully dynamic minimum c -cut algorithm makes use of the terminal edge connectivity data structure presented in [19] by Jin and Sun, where the terminal edge connectivity data structure was used to solve the fully dynamic (s, t) edge connectivity problem.

The data structure is based on a construction of terminal c -edge connectivity sparsifier for any $c = (\log n)^{o(1)}$

such that the number of vertices and edges in the sparsifier is linear to the number of terminals, and such a sparsifier can be updated in $n^{o(1)}$ time if the input graph has the conductance at least $1/n^{o(1)}$. In order to construct and update the terminal c -edge connectivity sparsifier efficiently for an arbitrary graph, the efficient update algorithm for the terminal edge connectivity sparsifier of the graphs that are $1/n^{o(1)}$ -expanders is combined with the update algorithms of expander decomposition [26, 3].

Since the computation of the c -edge connectivity between two vertices in the sparsifier may still be inefficient if a $1/n^{o(1)}$ -expander decomposition is used for the input graph, the sparsification is applied iteratively until the vertex partition contains only a single cluster.

In this paper, we show that if there is a minimum c -cut of G that is a terminal cut, then any minimum c -cut of the terminal c -edge connectivity sparsifier corresponds to a minimum c -cut of G , and thus finding a minimum terminal c -cut in the graph is reduced to finding a minimum cut in the terminal c -edge connectivity sparsifier (Lemma 4.2). This observation requires careful analysis of the aforementioned terminal c -edge connectivity sparsifier construction because the definition of the terminal c -edge connectivity sparsifier only promises to preserve the minimum cut size for the terminal cuts. Moreover, since the vertex set of the sparsifier does not contain all vertices of G , with a minimum c -cut of the sparsifier, it is not straightforward to define the corresponding cut of G . By investigating the construction of the terminal c -edge connectivity sparsifier, we build the correspondence between minimum c -cuts of the sparsifier and cuts of the same cut size in G , and show that given a minimum c -cut of the sparsifier, the corresponding terminal cut of the same cut size in G can be computed efficiently.

We remark that although Saranurak also presented an edge connectivity sparsifier based on expander decomposition [25], it is unknown how to efficiently maintain the sparsifier proposed in [25] under graph updates. Thus, for the purpose of a dynamic algorithm, we utilize the data structure proposed in [19].

1.1.3 Minimum c -Cut Data Structure For a graph G and a vertex partition $\mathcal{P}$ which is a $1/n^{o(1)}$ -expander decomposition of G , in the previous section, we discussed the way to find a global minimum cut if it is a terminal cut with respect to partition $\mathcal{P}$. In the rest of this section, we address the remaining problem of finding a global minimum cut which is a local cut.

We discuss the data structure used to maintain a minimum c -local cut for a graph G and a vertex partition $\mathcal{P}$. The main difficulty of keeping track of the minimum local cuts is that within a single cluster of a vertex partition, the number of local cuts can be $n^{O(c)}$, and these local cuts change significantly even with a single update of the graph. In order to update the data structure efficiently, we need to carefully organize the local cuts.

In our solution, we characterize some necessary conditions for a local cut that can potentially be a global minimum cut of the graph. This characterization reduces the number of local cuts that need to consider from $n^{O(c)}$ to $n \cdot c^{O(c)}$. Furthermore, we show that these local cuts can be enumerated efficiently. We use a priority queue to keep all these local cuts in the ascending order of the cut sizes. Thus, if the global minimum cut of the graph is a local cut, it is the first cut in the priority queue.

The minimum c -cut data structure contains a multi-level terminal c -edge connectivity data structure, and for each level, the minimum c -cut data structure stores an additional priority queue that maintains all local cuts that can potentially be a global minimum cut according to our characterization.

1.1.4 Fully Dynamic Minimum c -Cut Algorithm We give a brief overview of our fully dynamic algorithm. We focus on maintaining the minimum c -local cut data structure with respect to the updates of the graph and the vertex partition.

The main difficulty is that a lot of cuts may switch between local and non-local cuts when the graph gets updated. When the graph gets updated, the expander decomposition also gets changed. Consequently, since the terminals are defined as the boundary vertices of the vertex partition, the terminal set is also changed. Some previous terminals can become non-terminals (e.g., the incident edges to other clusters are removed), and some previous non-terminals can become terminals (e.g., an incident edge for a non-terminal to another cluster is inserted). Since local cuts require that the terminals within the induced subgraph are on the same side of the cuts, the change of terminals may turn some non-local cuts to local cuts, and some local cuts to non-local cuts. Thus, there can be a large number of c -cuts switching between local and non-local cuts after only one update of the graph.

As our main observation, we show that even though the total number of cuts switching between local and non-local can be large after graph updates, the number of local cuts in the priority queue that need to be updated

is always small based on the characterization in Section 1.1.3. Furthermore, we can efficiently find all local cuts in the priority queue that need to be updated.

Our solution considers the following offline update scenario: Let G be a graph, and G' be a graph after applying graph updates to G . Let $\mathcal{P}$ and $\mathcal{P}'$ be $1/n^{o(1)}$ -expander decomposition of G and G' respectively, such that each cluster of $\mathcal{P}'$ is a subset of a cluster of $\mathcal{P}$, and for each cluster $P' \in \mathcal{P}'$, the induced subgraph of G' on P' is the same as the induced subgraph of G on P' .

Note that before and after the updates, we only study local cuts of size at most c in $1/n^{o(1)}$ expanders, so local cuts always have a side of volume at most $c \cdot n^{o(1)}$. As our main observation, we show that if a cut switches from a local cut to a non-local cut, then for the local cut before the update, the side with volume at most $c \cdot n^{o(1)}$ has at least one of the following two types of vertices:

- a vertex that switches between a terminal and non-terminal in the update;
- a vertex that is an endpoint of an edge inserted or deleted in the update.

Similarly, if a cut switches from a non-local cut to a local cut, then for the local cut after the update, at least one of the two aforementioned types of vertices exists on the side of volume at most $c \cdot n^{o(1)}$.

Therefore, we update the priority queue containing local cuts of G to a priority queue containing local cuts of G' in the following way:

1. List all of the aforementioned two types of vertices in G . These are the vertices that are “affected” by the graph updates.
2. Remove all local cuts of G in the priority queue that have an affected vertex on the side of volume at most $c \cdot n^{o(1)}$.
3. Enumerate local cuts of G' that contain an affected vertex on the side of volume at most $c \cdot n^{o(1)}$ and add the enumerated cuts back to the priority queue.

We show the number of “affected” vertices is linear in the number of graph updates up to a subpolynomial factor. Therefore, the total number of cuts that need to be removed from and added to the local cut data structures in each level is at most a subpolynomial factor to the number of updates. This allows us to update the data structure efficiently (Lemma 5.4).

2 Preliminaries

Multigraph For a multigraph $G = (V, E)$, the edge set E is a set of triples (u, v, α) such that u and v are two vertices in V , and α is a positive integer which represents the edge multiplicity of the edge with u and v as two endpoints. We assume (u, v, α) is equivalent to (v, u, α) so that for every two vertices u and v , there is at most one triple with u and v as first two elements. For a graph G , we use $V(G)$ to denote the vertex set of G , and $E(G)$ to denote the edge set of G .

We use the degree reduction technique [12] to transform the input simple graph with an arbitrary maximum degree to a multigraph such that every vertex has at most a constant number of distinct neighbors. Furthermore, each edge insertion or deletion for the simple graph corresponds to a constant number of multigraph updates with the following update operations:

- $\text{insert}(u, v, \alpha)$: insert edge (u, v, α) to the graph
- $\text{delete}(u, v)$: delete the edge with u and v as two endpoints from the graph, no matter what the edge multiplicity is
- $\text{insert}(v)$: insert a new vertex v to the graph
- $\text{delete}(v)$: delete isolated vertex v from the graph.

See Appendix A for the detailed discussion on the multigraph construction and update. In this paper, unless specified, we assume graphs are undirected multigraphs with a constant number of neighbors.

Notations We say E' is an edge subset of $G = (V, E)$ if E' is a subset of E . For a subset of edges $E' \subseteq E$, we use $|E'|$ to denote $\sum_{(u,v,\alpha) \in E'} \alpha$, use $\text{End}(E')$ to denote the set of endpoints of all the edges in E' , and use $G \setminus E'$ to denote the graph $(V, E \setminus E')$.

For a graph $G = (V, E)$ and a subset of vertices $V' \subseteq V$, we use $\text{vol}_G(V')$ to denote the volume of V' in G , which is defined as

$$\text{vol}_G(V') \stackrel{\text{def}}{=} \sum_{u \in V'} \sum_{(u,v,\alpha) \in E} \alpha.$$

We use $G[V']$ to denote the induced subgraph of G on V' , and use $\partial_G(V')$ to denote the set of edges with one endpoint in V' and another endpoint in $V \setminus V'$.

Let $\mathcal{P}$ be a vertex partition of V for graph G . We call each set in $\mathcal{P}$ a cluster. We say an edge of G is an intercluster edge with respect to $\mathcal{P}$ if the two endpoints of the edge belong to different clusters, otherwise, the edge is an intracenter edge. We use $G[\mathcal{P}]$ to denote the union of the induced subgraphs on all the clusters in $\mathcal{P}$ on G , i.e., $G[\mathcal{P}] = (V, E')$ where $E' = \{(u, v, \alpha) \in E : \exists P \in \mathcal{P} \text{ s.t. } u, v \in P\}$. We use $\partial_G(\mathcal{P})$ to denote the union of $\partial_G(P)$ for all the $P \in \mathcal{P}$, i.e., the intercluster edges with respect to $\mathcal{P}$. A vertex is a boundary vertex with respect to $\mathcal{P}$ if there exists an intercluster edge incident to the vertex.

A *cut* is a bipartition of the graph vertices. The *cut-set* of a cut is the set of edges with two endpoints belonging to different sides of the cut. The *size of a cut* is the sum of edge multiplicities for all the edges in the cut's cut-set. For a given positive integer c , we say a cut is a *c-cut* if the size of the cut is at most c .

Expander Decomposition We also use expander decomposition in this paper.

DEFINITION 2.1. (EXPANDER AND EXPANDER DECOMPOSITION) For a graph $G = (V, E)$ of n vertices and m edges, the conductance of G is

$$\min_{\emptyset \subsetneq S \subsetneq V} \frac{|\partial_G(S)|}{\min\{\text{vol}_G(S), \text{vol}_G(V \setminus S)\}}$$

A graph G is a ϕ -expander if the conductance of G is at least ϕ .

A (ϕ, ϵ) -expander decomposition of G is a vertex partition $\mathcal{P}$ of V such that for each $P \in \mathcal{P}$, the induced subgraph $G[P]$ is a ϕ -expander, and the number of intercluster edges is at most $\epsilon \cdot m$. For convenience, a (ϕ, ϵ) -expander decomposition is also called a ϕ -expander decomposition.

We utilize the deterministic expander decomposition algorithm by Chuzhoy et al. [3], and the expander pruning algorithm by Saranurak and Wang in [26] used in this paper.

THEOREM 2.1. (COROLLARY 7.1 OF [3]) Given a graph $G = (V, E)$ of m edges and a parameter $0 < \phi < 1$, there is a deterministic algorithm to compute a $(\phi, \phi \cdot 2^{\delta \log^{1/3} n \log^{2/3} \log n})$ -expander decomposition of G in time $(m/\phi^2)^{1+o(1)}$ for some constant $\delta > 0$.

THEOREM 2.2. (THEOREM 1.3 OF [26], REPHRASED) Let $G = (V, E)$ be a simple ϕ -expander with m edges. Given access to adjacency lists of G and a set D of $k \leq \phi m/10$ edges, there is a deterministic algorithm to find a pruned set $P \subseteq V$ in time $O(k \log m/\phi^2)$ such that all of the following conditions hold:

1. $\text{vol}_G(P) = 8k/\phi$.
2. $|E_G(P, V \setminus P)| \leq 4k$.
3. $G'[V \setminus P]$ is a $\phi/6$ expander, where $G' = (V, E \setminus D)$.

3 Review: Multi-level Terminal Edge Connectivity Data Structure

In this section, we review the definition and main properties of the multi-level terminal edge connectivity data structure defined in [19]. The c -edge connectivity between a pair of vertices is defined as follows.

DEFINITION 3.1. For a multigraph $G = (V, E)$, two vertices $s, t \in V$, and a positive integer c , two vertices s and t are c -edge connected if s and t are connected in $G \setminus E'$ for all $E' \subseteq E$ such that $|E'| < c$.

The edge connectivity of s and t for graph G , denoted as $\lambda_G(s, t)$, is the non-negative integer such that s and t are $\lambda_G(s, t)$ edge connected, but not $(\lambda_G(s, t) + 1)$ -edge connected.

For a positive integer c , the c -edge connectivity of s and t for graph G is $\min\{c, \lambda_G(s, t)\}$, and the c -edge connectivity for graph G is

$$\min\left\{c, \min_{u, v \in V: u \neq v} \lambda_G(s, t)\right\}.$$

The dynamic (s, t) edge connectivity problem was studied in [19]. In this problem, for a fixed positive integer c , the algorithm receives updates of the graph, as well as c -edge connectivity queries each containing two vertices. The goal of the algorithm is to process each update and to answer the c -edge connectivity for each pair of queried vertices as efficiently as possible. As the main result, a fully dynamic algorithm of (s, t) c -edge connectivity for any $c = (\log n)^{o(1)}$ with $n^{o(1)}$ worst-case update and query time was presented in [19].

We remark that the fully dynamic global minimum cut problem, the problem we want to solve in this paper, cannot be solved in subpolynomial time using the fully dynamic (s, t) edge connectivity algorithm. Even though there always exist two vertices in the graph such that the edge connectivity between the two vertices is the same as the global minimum cut size, it is non-trivial to find such a vertex pair after each update of the graph, and the only way we know how to find such a pair of vertices is to first identify the global minimum cut.

3.1 Terminal Edge Connectivity Sparsifier The high level intuition of [19] is to construct and update a terminal edge connectivity sparsifier for a subset of selected vertices, called terminals, such that the c -edge connectivity is preserved for any pair of terminals in the sparsifier compared with the input graph.

DEFINITION 3.2. Let $G = (V, E)$ be a multiple graph, $T \subseteq V$ be a terminal set, and c be a positive integer. A multigraph $H = (V_H, E_H)$ is a terminal c -edge connectivity sparsifier of $G = (V, E)$ with respect to T if T is a subset of both V and V_H such that the c -edge connectivity between any two vertices $u, v \in T$ for H is the same as the c -edge connectivity between u and v for G .

We use the cut containment set, which was proposed in [22, 2], to construct a terminal edge connectivity sparsifier for a given terminal set. For a set of terminal vertices $T \subseteq V$, a cut is called a *terminal cut* if the cut partitions T into two non-empty sets, otherwise, the cut is called a *non-terminal cut*. Terminal cuts for terminal set T that are also c -cuts are called (T, c) -cuts. A (T, c) -cut that partitions T into T' and $T \setminus T'$ is called a $(T', T \setminus T', c)$ -cut.

DEFINITION 3.3. (CUT CONTAINMENT SET) Let $G = (V, E)$ be a graph, $T \subseteq V$ be a terminal set, and c be a positive integer. A set of edges CC is a c -cut containment set with respect to T if CC corresponds to intercluster edges of a vertex partition of G , and for every bipartition $(T', T \setminus T')$ of T such that G has a $(T', T \setminus T', c)$ -cut, there exists a minimum $(T', T \setminus T', c)$ -cut C such that all the edges in the cut-set of C are in CC .

The terminal edge connectivity sparsifier also makes use of the contraction technique used in [14, 17, 24]. Given an unweighted forest $F = (V, E)$ and a set of terminals $K \subseteq V$, the contraction of F with respect to K , denoted by $\text{Contract}_K(F)$, is obtained by repeatedly deleting degree-1 vertices not in K and shortcircuiting degree-2 vertices not in K . $\text{Contract}_K(F)$ may have more vertices than just terminals, but $\text{Contract}_K(F)$ is the minimal unweighted forest (in terms of the number of vertices and edges) satisfying the following two conditions:

1. All the vertices in K are vertices of $\text{Contract}_K(F)$ such that any two vertices in K are connected in $\text{Contract}_K(F)$ iff they are connected in F .
2. Each edge of $\text{Contract}_K(F)$ corresponds to a path of F that is part of a path connecting two vertices of K , and for any two edges of $\text{Contract}_K(F)$, their corresponding paths of F are edge disjoint.

As proved in [14], $\text{Contract}_K(F)$ contains $O(|K|)$ vertices and edges. See Appendix B for more details of the contraction technique. We remark that $\text{Contract}_K(F)$ can contain more vertices than K .

In [19], given a graph G , a terminal set T , a c -cut containment set CC for G with respect to T for some positive integer c , and a positive integer γ greater than c , an unweighted spanning forest F of $G \setminus CC$, the terminal c -edge connectivity sparsifier of G with respect to T , denoted as $\text{Sparsifier}_{c, \gamma}(G, T, CC, F)$, is the multigraph with the vertex set of $\text{Contract}_{T \cup \text{End}(CC)}(F)$ as vertices, the union of CC and the edge set of $\text{Contract}_{T \cup \text{End}(CC)}(F)$ as edges, and edge multiplicities defined as follow: for the edge with x and y as two endpoints in $\text{Sparsifier}_{c, \gamma}(G, T, CC, F)$

- if there is an edge $(x, y, \alpha) \in CC$ for some positive integer α , the multiplicity of the edge with x and y as two endpoints in $\text{Sparsifier}_{c,\gamma}(G, T, CC, F)$ is α ;
- otherwise the multiplicity of the edge with x and y as two endpoints in $\text{Sparsifier}_{c,\gamma}(G, T, CC, F)$ is γ .

[19] showed that $\text{Sparsifier}_{c,\gamma}(G, T, CC, F)$ has $O(|CC| + |T|)$ vertices and $O(|CC| + |T|)$ edges with different endpoints, and the c -edge connectivity in G for any pair of vertices within T is the same as that of $\text{Sparsifier}_{c,\gamma}(G, T, CC, F)$.

For the purpose of efficient updates, the terminal c -edge connectivity sparsifier is also defined with respect to vertex partitions. Given a graph G and a vertex partition $\mathcal{P}$ of G , the terminal set for G and $\mathcal{P}$, denoted as $T_{G,\mathcal{P}}$, is defined as $\text{End}(\partial_G(\mathcal{P}))$. For a positive integer c , an edge subset of G is a c -cut containment set for G and $\mathcal{P}$ if the edge set is the union of $\partial_G(\mathcal{P})$ and the c -cut containment sets of $G[P]$ with $T_{G,\mathcal{P}} \cap P$ as terminals for all $P \in \mathcal{P}$.

Given a graph G , a vertex partition $\mathcal{P}$ of G , positive integers c, γ such that $\gamma > c$, a c -cut containment set CC for G and $\mathcal{P}$, and an unweighted spanning forest F of $G \setminus CC$, the terminal c -edge connectivity sparsifier of G with respect to $\mathcal{P}$, CC and F is constructed as follows: start from G , for each $P \in \mathcal{P}$, replace $G[P]$ in G by

$$\text{Sparsifier}_{c,\gamma}(G[P], T_{G,\mathcal{P}} \cap P, CC \cap E(G[P]), F[P]).$$

Recall here that $T_{G,\mathcal{P}} = \text{End}(\partial_G(\mathcal{P}))$. We use $\text{Sparsifier}_{c,\gamma}(G, \mathcal{P}, CC, F)$ to denote the resulting sparsifier.

LEMMA 3.1. ([19]) *Let $G = (V, E)$ be a graph, $\mathcal{P}$ be a vertex partition of G , c and γ be positive integers such that $\gamma > c$, CC be a c -cut containment set for G and $\mathcal{P}$, and F be an unweighted spanning forest of $G \setminus CC$. $\text{Sparsifier}_{c,\gamma}(G, \mathcal{P}, CC, F)$ is a terminal c -edge connectivity sparsifier of G with respect to $T_{G,\mathcal{P}}$.*

We remark that the edge connectivity parameter gets smaller after the sparsifier gets updated with respect to graph updates (see Lemma 3.3). In order to answer c -edge connectivity queries, we set the edge connectivity parameter to be large enough so that as the sparsifier updated we can still use the sparsifier to answer c -edge connectivity queries, as a terminal c' -edge connectivity sparsifier is also a terminal c -edge connectivity sparsifier for any $c' > c$.

3.2 One-Level Terminal Edge Connectivity Data Structure We summarize the one-level terminal edge connectivity data structure from [19], and the algorithms for the one-level data structure.

For the convenience of constructing and updating terminal edge connectivity sparsifiers with respect to a graph, we suppose that an input graph and its terminal edge connectivity sparsifier are maintained by a *one-level terminal edge connectivity data structure*. A one-level terminal edge connectivity data structure contains an input graph G , a vertex partition $\mathcal{P}$, a cut containment set CC , an unweighted spanning forest F of $G \setminus CC$, the corresponding terminal edge connectivity sparsifier $\text{Sparsifier}_{c,\gamma}(G, \mathcal{P}, CC, F)$ for some positive integers c and γ , and other auxiliary data structure so that when updates of G occur, $\mathcal{P}$ and CC , F , and their corresponding terminal edge connectivity sparsifier can be updated efficiently.

We suppose that $\mathcal{P}$ is a ϕ -expander decomposition of G for some $\phi = 1/n^{o(1)}$, as the terminal edge connectivity sparsifier can only be efficiently constructed and updated for such a $\mathcal{P}$ by [19] (see Lemma 3.2 and Lemma 3.3 for detailed discussion). We also assume that the data structure contains $\text{Contract}_{\text{End}(CC)}(F)$ and the terminal set $T_{G,\mathcal{P}}$ such that vertices of $T_{G,\mathcal{P}}$ are stored by the cluster they belong to so that given the index of a cluster $P \in \mathcal{P}$, it takes $O(1)$ time to return the access of $T_{G,\mathcal{P}} \cap P$.

Thus, the one-level terminal edge connectivity data structure has three parameters c , ϕ and γ , where c is the edge connectivity parameter for the cut containment set, ϕ is the conductance parameter for the expander decomposition, and γ is the parameter to construct the terminal edge connectivity sparsifier satisfying $\gamma > c$. Throughout the paper, γ is a fixed parameter, and the other two parameters, c and ϕ , get smaller after applying the updates to the input graph (see Lemma 3.3).

We use $(G, \mathcal{P}, CC, F)$ to denote the one-level terminal edge connectivity data structure with respect to given $G, \mathcal{P}, CC$, and F , which consists of $\text{Sparsifier}_{c,\gamma}(G, \mathcal{P}, CC, F)$.

Now we review the initialization and update algorithms for the terminal edge connectivity data structure.

LEMMA 3.2. ([19]) *Given a graph G , two integers c and γ such that $\gamma > c$, and a conductance parameter $0 < \phi < 1$, there is an algorithm **Edge-Connectivity-One-Level-Initialization** to initialize a one-level terminal edge connectivity data structure $(G, \mathcal{P}, CC, F)$ with parameters c, ϕ and γ in time $m^{1+o(1)} \cdot c^{O(c)} / \text{poly}(\phi)$ such that $\text{Sparsifier}_{c,\gamma}(G, \mathcal{P}, CC, F)$ contains at most*

$$m \cdot \phi \cdot 2^{O(\log^{1/3} n \log^{2/3} \log n)}$$

vertices and edges.

For the updates of the one-level data structure, [19] presented an offline update algorithm for the one-level terminal edge connectivity data structure, i.e., given an update sequence of the graph, the algorithm update the one-level terminal edge connectivity data structure according to the given update sequence of the graph. The update sequence consists of the multigraph update operations defined in Section 2. Let **UpdateSeq** be an update sequence. We use $|\text{UpdateSeq}|$ to denote the length of the update sequence.

The lemma below for the update algorithm shows that for a one-level terminal edge connectivity data structure $(G, \mathcal{P}, CC, F)$, given an update sequence **UpdateSeq** of G , the one-level terminal edge connectivity data structure can be updated in time linear with respect to the length of the update sequence.

LEMMA 3.3. ([19]) *Given a one-level terminal edge connectivity data structure $(G, \mathcal{P}, CC, F)$ with parameters $c^2 + 2c, \phi$ and γ satisfying $\gamma > c^2 + 2c$, and an update sequence **UpdateSeq** of G which contains vertex/edge insertions/deletions, Algorithm **Edge-Connectivity-One-Level-Update** with running time $O(|\text{UpdateSeq}| \cdot O(c/\phi)^{O(c^2)})$ updates the data structure to $(G', \mathcal{P}', CC', F')$ with parameters $c, \phi/2^{O(\log^{1/3} n \log \log n)}$ and γ , and outputs an update sequence **UpdateSeq'** and a vertex set S satisfying the following properties:*

1. G' is the resulted graph of applying **UpdateSeq** to G .
2. $\mathcal{P}'$ is a $\phi/2^{O(\log^{1/3} n \log \log n)}$ -expander decomposition of G' . Furthermore, each cluster $P' \in \mathcal{P}'$ either contains only a single vertex or is a subset of a cluster $P \in \mathcal{P}$ such that $G'[P'] = G[P]$.
3. CC' is a c -cut containment set for G' and $\mathcal{P}'$.
4. F' is an unweighted spanning forest of $G' \setminus CC'$.
5. The sparsifier maintained in the data structure is $\text{Sparsifier}_{c,\gamma}(G', \mathcal{P}', CC', F')$.
6. **UpdateSeq'** updates $\text{Sparsifier}_{c^2+2c,\gamma}(G, \mathcal{P}, CC, F)$ to $\text{Sparsifier}_{c,\gamma}(G', \mathcal{P}', CC', F')$. The length of **UpdateSeq'** is at most $|\text{UpdateSeq}| (10c)^{O(c)}$.
7. $S = (\text{End}(\partial_G(\mathcal{P})) \setminus \text{End}(\partial_{G'}(\mathcal{P}')) \cup (\text{End}(\partial_{G'}(\mathcal{P}')) \setminus \text{End}(\partial_G(\mathcal{P})))$ such that $|S| = O(|\text{UpdateSeq}|)$.

3.3 Multi-Level Terminal Edge Connectivity Data Structure Now we give a brief summary of the multi-level terminal edge connectivity data structure. The motivation of the multi-level data structure is to apply the terminal edge connectivity sparsifier iteratively so that the size of the sparsifiers reduce gradually, and the edge connectivity is easy to compute in the sparsifier at the end.

A multi-level terminal edge connectivity data structure has three parameters c, ϕ , and γ , where c is the edge connectivity parameter, ϕ is the conductance parameter, and γ is the parameter used to construct the sparsifiers for all the levels. A multi-level terminal edge connectivity data structure with parameters c, ϕ , and γ is a set of one-level data structures $\{(G^{(i)}, \mathcal{P}^{(i)}, CC^{(i)}, F^{(i)})\}_{i=0}^{\ell}$ satisfying the following conditions:

- (c1) $(G^{(i)}, \mathcal{P}^{(i)}, CC^{(i)}, F^{(i)})$ is a one-level terminal edge connectivity data structure with parameters c, ϕ and γ .
- (c2) $G^{(0)}$ is the input graph, and $G^{(i)} = \text{Sparsifier}_{c,\gamma}(G^{(i-1)}, \mathcal{P}^{(i-1)}, CC^{(i-1)}, F^{(i)})$ for each $0 < i \leq \ell$.
- (c3) $\mathcal{P}^{(\ell)}$ contains only one cluster.

To initialize a multi-level data structure for a given graph, we apply the one-level initialization algorithm iteratively until the vertex partition contains only one cluster. And given an update sequence `UpdateSeq` for graph G , a multi-level sparsifier $\{(G^{(i)}, \mathcal{P}^{(i)}, CC^{(i)}, F^{(i)})\}_{i=0}^{\ell}$ of G is updated as follows: Suppose `UpdateSeq`⁽⁰⁾ = `UpdateSeq`. Starting from $i = 0$ to ℓ , run the one-level update algorithm on the one-level terminal edge connectivity data structure $(G^{(i)}, \mathcal{P}^{(i)}, CC^{(i)}, F^{(i)})$ with update sequence `UpdateSeq`⁽ⁱ⁾, and let `UpdateSeq`⁽ⁱ⁺¹⁾ be the update sequence outputted by the one-level update algorithm. If at some level, the update sequence contains too many updates so that the update algorithm is slower than recompute the sparsifier, then use the initialization algorithm to reconstruct the sparsifiers iteratively for the remaining levels.

The fully dynamic algorithm is obtained by applying a framework proposed in [24] that converts a batched dynamic update algorithm to a fully dynamic algorithm. See Appendix C for more details of the framework proposed in [24].

THEOREM 3.1. (THEOREM 9.3 AND 9.4 FROM [19]) *For any $c = (\log n)^{o(1)}$, there is a fully dynamic algorithm which maintains a set of multi-level terminal edge connectivity data structure such that each maintained terminal edge connectivity data structure has $O(\log^{1/10} n)$ levels all the time, and after processing each update, the algorithm provides the access to one of the maintained multi-level terminal edge connectivity data structure $\{(G^{(i)}, \mathcal{P}^{(i)}, CC^{(i)}, F^{(i)})\}_{i=0}^{\ell}$ for the up-to-date graph with parameters c, ϕ, γ satisfying the following conditions:*

$$\phi = 1/n^{o(1)}, \text{ and } \gamma > c.$$

The initialization time of the algorithm is $m^{1+o(1)}$, and the update time of the algorithm is $n^{o(1)}$ per update.

4 Minimum c -Cut Data Structure

We aim to maintain a minimum c -cut data structure that allows to compute the minimum c -cut of the input graph efficiently. In this section, we present our minimum c -cut data structure. In Section 4.3, we present the initialization and update algorithms for the minimum c -cut data structure.

4.1 The Characterization of Minimum c -cuts The high level intuition is to make use of the terminal edge connectivity sparsifier for c -edge connectivity described in Section 3 to speed up the process of finding the minimum c -cut if at least one of the minimum c -cuts of the input graph is captured by the sparsifier, as the terminal edge connectivity sparsifier is smaller than the input graph in terms of the number of vertices and edges. If no minimum c -cut is captured by the terminal edge connectivity sparsifier, we need to characterize these minimum c -cuts, and devise a new approach to find these cuts efficiently.

The two lemmas below show that the c -edge connectivity sparsifier does not only preserve the c -edge connectivity between any pair of terminals, but also allows us to reconstruct the c -terminal cuts.

LEMMA 4.1. *Let G be a graph, T be a set of terminals, CC be a c -cut containment set for G for some positive integer c , and F be an unweighted spanning forest of $G \setminus CC$. The following properties hold for $\text{Sparsifier}_{c,\gamma}(G, T, CC, F)$ for any $\gamma > c$:*

1. *The cut-set of any c -cut in $\text{Sparsifier}_{c,\gamma}(G, T, CC, F)$ is the cut-set of a cut with the same cut size in G .*
2. *If there is a c -cut C in G such that C partitions T into two non-empty subsets T' and $T \setminus T'$, then there is a cut C' of size smaller than or equal to the size of C in $\text{Sparsifier}_{c,\gamma}(G, T, CC, F)$ that partitions T into T' and $T \setminus T'$.*

Proof. By Definition 3.3, CC is the set of intercluster edges of a vertex partition of G . Let $\mathcal{R}$ denote this vertex partition. By the construction of $\text{Sparsifier}_{c,\gamma}(G, T, CC, F)$, $\text{Sparsifier}_{c,\gamma}(G, T, CC, F)$ contains all the edges of CC . Comparing G and $\text{Sparsifier}_{c,\gamma}(G, T, CC, F)$, for each $R \in \mathcal{R}$, $G[R]$ in G is replaced by a tree in $\text{Sparsifier}_{c,\gamma}(G, T, CC, F)$ such that the edge multiplicity of each edge in the tree is greater than c . Furthermore, the vertices of $\text{Sparsifier}_{c,\gamma}(G, T, CC, F)$, which contain all the endpoints of edges in CC , are also vertices of G .

With these properties of $\text{Sparsifier}_{c,\gamma}(G, T, CC, F)$, any c -cut of $\text{Sparsifier}_{c,\gamma}(G, T, CC, F)$ only contains edges from CC . Also, for any two vertices u and v in $\text{Sparsifier}_{c,\gamma}(G, T, CC, F)$, if there is a $R \in \mathcal{R}$ containing both u and v , u and v are in the same side of any c -cut of $\text{Sparsifier}_{c,\gamma}(G, T, CC, F)$. Let $C' = (V'_1, V'_2)$ be a c -cut

of $\text{Sparsifier}_{c,\gamma}(G, T, CC, F)$. We construct cut $C = (V_1, V_2)$ of G as follows: for any vertex v of G , if there is a vertex $x \in V_1'$ such that there is a $R \in \mathcal{R}$ containing both v and x , then v is in V_1 , otherwise, v is in V_2 . By the construction of C , we have the following properties:

1. For any two vertices x and y in $\text{Sparsifier}_{c,\gamma}(G, T, CC, F)$, if x and y are in the same side of C' , x and y are in the same side of C .
2. For any two vertices x and y of G , if there is a $R \in \mathcal{R}$ such that both x and y are in R , then x and y are in the same side of C .

Since every vertex in $\text{End}(\partial_G(\mathcal{R}))$ is an endpoint of some edge in CC , the cut-set of C is the same as the cut-set of C' . Hence, the first property holds.

Now we prove the second property. By Definition 3.3, there is a set of edges $W \subseteq CC$ such that W is the cut-set of a $(T', T \setminus T', c)$ -cut $C_1 = (V_1, V_2)$ of G such that the cut size of C_1 is smaller than or equal to C . We construct a cut $C' = (V_1', V_2')$ of $\text{Sparsifier}_{c,\gamma}(G, T, CC, F)$ as follows: for any vertex v of $\text{Sparsifier}_{c,\gamma}(G, T, CC, F)$, if there is a vertex $x \in V_1$ such that there is a $R \in \mathcal{R}$ containing both v and x , then v is in V_1' , otherwise, v is in V_2' . By the construction of C' , we have the following properties:

1. For any two vertices x and y in $\text{Sparsifier}_{c,\gamma}(G, T, CC, F)$, if there is a $R \in \mathcal{R}$ such that both x and y are in R , then x and y are in the same side of C' .
2. For any two vertices x and y in $\text{Sparsifier}_{c,\gamma}(G, T, CC, F)$, if x and y are in the same side of C for G , then x and y are in the same side of C' for $\text{Sparsifier}_{c,\gamma}(G, T, CC, F)$.

Thus, the cut-set of C' is the same as the cut-set of C_1 , and then the second property holds. $\square$

LEMMA 4.2. *Let G be a graph, $\mathcal{P}$ be a vertex partition of G , CC be a c -cut containment set for G and $\mathcal{P}$ with some positive integer c , and F be an unweighted spanning forest of $G \setminus CC$. The following properties hold for $\text{Sparsifier}_{c,\gamma}(G, \mathcal{P}, CC, F)$ for any $\gamma > c$:*

1. *The cut-set of any c -cut in $\text{Sparsifier}_{c,\gamma}(G, \mathcal{P}, CC, F)$ is the cut-set of a cut with the same size in G .*
2. *If there is a c -cut C in G such that C partitions terminals $T_{G,\mathcal{P}}$ into two non-empty subsets T' and $T_{G,\mathcal{P}} \setminus T'$, then there is a cut C' of size smaller than or equal to C in the sparsifier $\text{Sparsifier}_{c,\gamma}(G, \mathcal{P}, CC, F)$ such that C' partitions $T_{G,\mathcal{P}}$ into T' and $T_{G,\mathcal{P}} \setminus T'$.*

Proof. By the definition of CC , CC is the union of $\partial_G(\mathcal{P})$ and c -cut containment sets of $G[P]$ with $T_{G,\mathcal{P}} \cap P$ as terminals for all $P \in \mathcal{P}$. By Definition 3.3, CC is the set of intercluster edges of a vertex partition of G . Let $\mathcal{R}$ denote this vertex partition. $\mathcal{R}$ is a refinement of $\mathcal{P}$, i.e., for each $R \in \mathcal{R}$, there is a $P \in \mathcal{P}$ such that R is a subset of P .

Recall that $\text{Sparsifier}_{c,\gamma}(G, \mathcal{P}, CC, F)$ is obtained by replacing each induced subgraph $G[P]$ in G by

$$\text{Sparsifier}_{c,\gamma}(G[P], T_{G,\mathcal{P}} \cap P, CC \cap E(G[P]))$$

for all the $P \in \mathcal{P}$. By the construction of $\text{Sparsifier}_{c,\gamma}(G[P], T \cap P, CC \cap E(G[P]))$, the vertices of $\text{Sparsifier}_{c,\gamma}(G, \mathcal{P}, CC, F)$, which contain all the endpoints of edges in CC , are also vertices of G .

Hence, any c -cut of $\text{Sparsifier}_{c,\gamma}(G, \mathcal{P}, CC, F)$ only contains edges from CC . And for any pair of vertices in $\text{Sparsifier}_{c,\gamma}(G, \mathcal{P}, CC, F)$ that are contained by a vertex set of $\mathcal{R}$, they are in the same side of any c -cut of $\text{Sparsifier}_{c,\gamma}(G, \mathcal{P}, CC, F)$.

To prove the first property, let $C' = (V_1', V_2')$ be an arbitrary c -cut of $\text{Sparsifier}_{c,\gamma}(G, \mathcal{P}, CC, F)$. We construct a cut $C = (V_1, V_2)$ of G as follows: for any vertex v of G , if there is a vertex $x \in V_1'$ such that there is a $R \in \mathcal{R}$ containing both v and x , then v is in V_1 , otherwise, v is in V_2 . Since vertices in G from the same cluster of $\mathcal{R}$ are in the same side of C , the cut-set of C is a subset of CC . By the construction of C , two endpoints of any edge in CC are on the same side of C only if the two endpoints are on the same side of C' for $\text{Sparsifier}_{c,\gamma}(G, \mathcal{P}, CC, F)$. Hence, The cut-set of C is the same as the cut-set of C' , and thus the first property holds.

Now we prove the second property. Let S denote the cut-set of G . Consider an arbitrary $P_1 \in \mathcal{P}$. Let $S_{P_1} = S \cap E(G[P_1])$. If S_{P_1} is the cut-set of a terminal cut C_{P_1} for $G[P_1]$ with $T_{G,\mathcal{P}} \cap P_1$ as terminals, then by Lemma 4.1, there is a set of edges $S'_{P_1} \subseteq CC$ satisfying the following conditions:

- S'_{P_1} is the cut-set of a cut C'_{P_1} for $G[P_1]$.
- C_{P_1} and C'_{P_1} have the same partition on $T_{G,\mathcal{P}} \cap P_1$.
- The cut size of C'_{P_1} is smaller than or equal to C_{P_1} .

If S_{P_1} is not the cut-set of a terminal cut for $G[P_1]$ with $T_{G,\mathcal{P}} \cap P_1$ as terminals, then we let S'_{P_1} be an empty set.

Let G_1 be the graph obtained by using $\text{Sparsifier}_{c,\gamma}(G[P_1], T_{G,\mathcal{P}} \cap P_1, CC \cap E(G[P_1]))$ to replace $G[P_1]$ in graph G . $(S \setminus S_{P_1}) \cup S'_{P_1}$ is the cut-set of a cut C_1 for graph G_1 satisfying the following conditions:

- The cut size of C_1 is smaller than or equal to that of C .
- C and C_1 have the same partition on $T_{G,\mathcal{P}}$ for G and G_1 respectively.
- The edges of C_1 's cut-set that are in $G[P_1]$ are in CC .

If the above process is repeated for all the other clusters in $\mathcal{P}$, then at the end, we obtain a cut C' for $\text{Sparsifier}_{c,\gamma}(G, \mathcal{P}, CC, F)$ satisfying the following conditions:

- The cut size of C' is smaller than or equal to that of C .
- C and C' have the same partition of $T_{G,\mathcal{P}}$ for G and $\text{Sparsifier}_{c,\gamma}(G, \mathcal{P}, CC, F)$ respectively.
- The cut-set of C' are in CC .

Thus, the second property of the lemma holds. $\square$

By Lemma 4.2, if there is a minimum cut of the input graph being a terminal cut, then the terminal cut can be obtained from the terminal edge connectivity sparsifier. Otherwise, no minimum cut is a terminal cut, the terminal edge connectivity sparsifier is not helpful. Hence, we need to devise a new approach for the case that all the minimum c -cuts of the input graph are not terminal cuts. The following lemma characterizes the minimum cuts of a graph with respect to a vertex partition.

LEMMA 4.3. *Let G be a graph, and $\mathcal{P}$ be a vertex partition of G . Let C be a minimum cut on G . C is either a terminal cut with $T_{G,\mathcal{P}}$ as terminals or a cut with cut-set in $G[P]$ for some $P \in \mathcal{P}$ such that all the terminal vertices in $T_{G,\mathcal{P}}$ are in the same side of C .*

Proof. If the cut-set of C contains an intercluster edge with respect to $\mathcal{P}$, then C is a terminal cut, because the two endpoints of the intercluster edge are terminals, and belong to different sides of the cut.

On the other hand, let P be a cluster of $\mathcal{P}$ such that the edges in the cut-set of C that are also in $G[P]$ form a terminal cut for $G[P]$ with $T_{G,\mathcal{P}} \cap P$ as terminals. Since the two endpoints of any edge in the cut-set of C are on different sides of C , vertices of $T_{G,\mathcal{P}} \cap P$ are not in the same side of C for G , and thus C is a terminal cut.

Hence, if C is not a terminal cut of G , then every edge in the cut-set of C is an intracluster edge with respect to $\mathcal{P}$, and for each $P \in \mathcal{P}$, one of the following two conditions hold:

1. The cut-set of C does not contain any edge in $G[P]$.
2. The edges in the cut-set of C that are also in $G[P]$ form a non-terminal cut of $G[P]$ with $T_{G,\mathcal{P}} \cap P$ as terminals.

For any $P \in \mathcal{P}$, let C' be an arbitrary cut of $G[P]$ such that the $T_{G,\mathcal{P}} \cap P$ are in the same side of C' . The cut-set of C' is also the cut-set of a cut in graph G . Hence, if C is a minimum cut of G that is a non-terminal cut with $T_{G,\mathcal{P}}$ as terminals, then there is a $P \in \mathcal{P}$ such that $G[P]$ contains all the edges in the cut-set of C , and the cut-set of C forms a non-terminal cut of $G[P]$ with $T_{G,\mathcal{P}} \cap P$ as terminals. Since any path from a vertex in P to a vertex not in P must contain a vertex in $T_{G,\mathcal{P}} \cap P$, C is a cut such that all the vertices of $T_{G,\mathcal{P}}$ are in the same side of C . $\square$

Based on Lemma 4.2 and Lemma 4.3, since terminal cuts of size at most c are captured by the sparsifier, we use the terminal sparsifier to find the minimum terminal c -cuts, and need to devise a new approach to find the minimum non-terminal c -cuts.

4.2 Local Cut We define local cut, the information we need to maintain in our global minimum cut data structure in order to find the minimum cut efficiently.

Recall that our goal is to find the minimum non-terminal c -cuts with $T_{G,\mathcal{P}}$ as terminals for a graph G and a vertex partition $\mathcal{P}$. By Lemma 4.3, every minimum non-terminal c -cut must have the cut-set belong to a cluster in $\mathcal{P}$. Hence, we only need to focus on non-terminal cuts with the cut-set in a cluster. On the other hand, note that each cluster is supposed to be a $1/n^{o(1)}$ -expander as Theorem 3.1. By the definition of expander, every cut of size at most c has a side of volume at most $O(c \cdot n^{o(1)})$. We use the following definition to describe non-terminal cuts in an expander.

DEFINITION 4.1. *Let $G = (V, E)$ be a graph with T as terminals, and α, c be two positive integers. We say a cut $C = (U, V \setminus U)$ is an (α, c) -local cut in G with respect to T if all of the following conditions hold*

- *The size of cut C is at most c ;*
- *Either U or $V \setminus U$ has volume at most α in G ;*
- *$T \subseteq U$ or $T \subseteq V \setminus U$.*

A cut C is a minimum (α, c) -local cut in G with respect to T if C is a (α, c) -local cut in G with respect to T and there is no other (α, c) -local cuts in G with respect to T with the size smaller than C .

The following lemma shows that if there exists a c -cut in a graph, then there is a minimum cut that is either a minimum local cut with appropriate parameters or a cut with cut-set that can be obtained from the terminal edge connectivity sparsifier.

LEMMA 4.4. *Let G be a graph, $\mathcal{P}$ be a ϕ -expander decomposition of G , CC be a c -cut containment set for G and $\mathcal{P}$ with some fixed positive integer c , F be an unweighted spanning forest of $G \setminus CC$, and $\text{Sparsifier}_{c,\gamma}(G, \mathcal{P}, CC, F)$ be a terminal edge connectivity sparsifier for G and $\mathcal{P}$ with parameter $\gamma > c$. If the minimum cut of G has size at most c , then there is a minimum cut C of G satisfies one of the following two conditions:*

1. *The cut set of C is also the cut set of a minimum cut of $\text{Sparsifier}_{c,\gamma}(G, \mathcal{P}, CC, F)$.*
2. *C is a minimum (α, c) -local cut in $G[P]$ for some $P \in \mathcal{P}$ with respect to $\text{End}(\partial_G(\mathcal{P})) \cap P$, and any $\alpha \geq c/\phi$.*

Proof. By Lemma 4.3, any minimum cut of G is either a terminal cut with $\text{End}(\partial_G(\mathcal{P}))$ as terminals, or a cut within $G[P]$ for some $P \in \mathcal{P}$ such that all the terminal vertices in $G[P]$ are in the same side of the cut.

If there is a minimum c -cut of G that is also a terminal cut, by the construction of the terminal cut sparsifier, the cut-set of a minimum cut of $\text{Sparsifier}_{c,\gamma}(G, \mathcal{P}, CC, F)$ is also the cut-set of a minimum cut of G .

If there is a minimum c -cut of G which is a cut within a $G[P]$ for some $P \in \mathcal{P}$ such that all the terminal vertices in $G[P]$ are on the same side of the cut, then one side of the cut is of volume at most c/ϕ since $G[P]$ is a ϕ -expander, by Definition 4.1, it is a (α, c) -local cut in $G[P]$ with respect to $\text{End}(\partial_G(\mathcal{P})) \cap P$. Since it is a minimum cut, it is a minimum (α, c) -local cut in $G[P]$ because every (α, c) -local cut is also a cut for the entire graph. $\square$

4.3 The Global Min-Cut Sparsifier Data Structure Lemma 4.3 states that a global minimum cut is either a terminal cut or a local cut. Given a graph G , in order to keep track of the global minimum cut, we use a terminal cut data structure to keep track of all terminal cuts, and try including an additional data structure to find a minimum local cut. In order to keep track of all local cuts in G , we need help from auxiliary graphs.

Auxiliary Graph and Partition We first construct an auxiliary graph based on G by connecting all terminal vertices. The auxiliary graph makes sure that in a local cut, the side containing all terminal vertices is always connected. This makes it easier to find minimum local cuts in the auxiliary graph.

DEFINITION 4.2. *For a graph $G = (V, E)$ and a partition $\mathcal{P}$ of V , the auxiliary graph of G with respect to $\mathcal{P}$, denoted as G_* , is obtained from G as follows: For each $P \in \mathcal{P}$, add a special terminal vertex t_P and connect t_P to every terminal vertex in P .*

The auxiliary partition $\mathcal{P}_$ is defined as $\{P_* | P \in \mathcal{P}\}$, where $P_* = P \cup \{t_P\}$. The terminals of G_* with respect to $\mathcal{P}_*$ are the terminals of G with respect to $\mathcal{P}$ union all the special terminals.*

The following lemma shows how the auxiliary graph is able to help us find minimum local cuts in the original graph.

LEMMA 4.5. *Let G be a graph and $\mathcal{P}$ be a ϕ -expander decomposition of G . Let G_* be the auxiliary graph of G with respect to $\mathcal{P}$. Let α and c be two parameters such that $\alpha \geq \lceil c/\phi \rceil$. The following two conditions hold for any $P \in \mathcal{P}$:*

1. *If $(U, P_* \setminus U)$ is a minimum $(3\alpha, c)$ -local cut in $G_*[P_*]$, then $(U \cap P, (P_* \setminus U) \cap P)$ is a minimum (α, c) -local cut in $G[P]$.*
2. *Suppose $(U', P \setminus U')$ is a minimum (α, c) -local cut in $G[P]$. Without loss of generality, assume all terminals of P are in U' . Then $(U' \cup \{t_P\}, P \setminus U')$ is a minimum $(3\alpha, c)$ -local cut in $G_*[P_*]$.*

Proof. Let $(U, P_* \setminus U)$ be a minimum $(3\alpha, c)$ -local cut in $G_*[P_*]$. Whether $t_P \in U$ or $t_P \in P_* \setminus U$, since all terminals are on the same side of the cut, removing edges between t_P and other terminals does not affect the number of edges in the cut-set. Therefore, $(U \cap P, (P_* \setminus U) \cap P)$ is still a c -cut in $G[P]$. By the assumption that $G[P]$ is a ϕ -expander, one side of $(U \cap P, (P_* \setminus U) \cap P)$ has volume at most α . Therefore, $(U \cap P, (P_* \setminus U) \cap P)$ is a (α, c) -local cut in $G[P]$ that has the same size as $(U, P_* \setminus U)$. $(U \cap P, (P_* \setminus U) \cap P)$ must be a minimum (α, c) -local cut in $G[P]$ because otherwise $(U, P_* \setminus U)$ is not a minimum $(3\alpha, c)$ -local cut in $G_*[P_*]$.

Let $(U', P \setminus U')$ be a minimum (α, c) -local cut in $G[P]$ such that the terminals are in U' . We look at cut $(U' \cup \{t_P\}, P \setminus U')$ in $G_*[P_*]$. Since the edges between t_P and other terminals are not in the cut-set, the size of this cut is at most c . Since $G[P]$ is a connected graph and the volume of U' in $G[P]$ is at most α , we add at most α new edges to turn $G[P]$ to $G_*[P_*]$. The volume of $U' \cup \{t_P\}$ in $G_*[P_*]$ is at most 3α . Therefore, $(U' \cup \{t_P\}, P \setminus U')$ is a $(3\alpha, c)$ -local cut in $G_*[P_*]$ that has the same cut size as $(U', P \setminus U')$. In addition, $(U' \cup \{t_P\}, P \setminus U')$ must be a minimum $(3\alpha, c)$ -local cut in $G_*[P_*]$, because if there exists a smaller $(3\alpha, c)$ -cut, then its corresponding cut in G would be an (α, c) -local cut which has a size smaller than $(U', P \setminus U')$, and thus $(U', P \setminus U')$ would not be a minimum (α, c) -local cut in $G[P]$. $\square$

As a result of Lemma 4.5, instead of finding minimum local cuts in G with respect to $\mathcal{P}$, we work on finding the minimum local cuts in G_* with respect to $\mathcal{P}_*$. The major advantage of using G_* and $\mathcal{P}_*$ is that we only need to consider cuts with both sides connected when considering the connected components of $G_*[\mathcal{P}_*]$. We note that our approach to finding the minimum cut is different from [25], which also finds the minimum cut of the graph by investigating a vertex partition. Our approach is based on finding minimum local cuts, whereas [25] relies on Gabow's matroid characterization for edge connectivity [7].

LEMMA 4.6. *Let G be a graph and $\mathcal{P}$ be a ϕ -expander decomposition of G . Let G_* and $\mathcal{P}_*$ be the auxiliary graph and partition of G with respect to $\mathcal{P}$. Let α and c be two parameters such that $\alpha \geq \lceil c/\phi \rceil$. Let C be the cut-set of a minimum $(3\alpha, c)$ -local cut of G_* with respect to $\mathcal{P}_*$, and $P_* \in \mathcal{P}_*$ be the vertex partition such that C contains only edges in $G_*[P_*]$. The graph obtained by removing C from $G_*[P_*]$ has only two connected components.*

Proof. Let P be the cluster in $\mathcal{P}$ that corresponds to $P_* \in \mathcal{P}_*$. Since $\mathcal{P}$ is a ϕ -expander decomposition, $G[P]$ is a ϕ -expander. By construction, $G_*[P_*]$ is obtained by adding a new terminal vertex t_P to $G[P]$ and connecting t_P to existing terminal vertices. For any bipartition (X_*, Y_*) of P_* , such that neither X_* nor Y_* is $\{t_P\}$ (we can assume $t_P \in X_*$ without loss of generality), there exists a corresponding bipartition $(X = X_* \cap P, Y = Y_* \cap P)$ of P . Since $G[P]$ is a ϕ -expander,

$$\frac{|\partial_{G[P]}(X)|}{\min(\text{vol}_{G[P]}(X), \text{vol}_{G[P]}(Y))} \geq \phi.$$

Suppose among all edges added to construct $G_*[P_*]$ from $G[P]$, F_1 is the set of edges that cross bipartition (X_*, Y_*) and F_2 is the set of edges that do not cross the bipartition, then

$$\frac{|\partial_{G_*[P_*]}(X_*)|}{\min(\text{vol}_{G_*[P_*]}(X_*), \text{vol}_{G_*[P_*]}(Y_*))} = \frac{|\partial_{G[P]}(X)| + |F_1|}{\min(2 \cdot |F_2| + \text{vol}_{G[P]}(X), \text{vol}_{G[P]}(Y))}.$$

Since F_2 consists only of edges connecting t_P and terminals in X , $|F_2|$ is at most $\mathbf{vol}_{G[P]}(X)$. Therefore,

$$\begin{aligned} \frac{|\partial_{G[P]}(X)| + |F_1|}{\min(2 \cdot |F_2| + \mathbf{vol}_{G[P]}(X), \mathbf{vol}_{G[P]}(Y))} &\geq \frac{|\partial_{G[P]}(X)| + 0}{\min(3 \cdot \mathbf{vol}_{G[P]}(X), \mathbf{vol}_{G[P]}(Y))} \\ &\geq \frac{|\partial_{G[P]}(X)|}{3 \cdot \min(\mathbf{vol}_{G[P]}(X), \mathbf{vol}_{G[P]}(Y))} \\ &\geq \phi/3. \end{aligned}$$

If X_* or Y_* is $\{t_P\}$, then

$$\frac{|\partial_{G_*[P_*]}(X_*)|}{\min(\mathbf{vol}_{G_*[P_*]}(X_*), \mathbf{vol}_{G_*[P_*]}(Y_*))} = 1 \geq \phi.$$

Hence $G_*[P_*]$ is a $\phi/3$ -expander. Since C forms a minimum cut, no proper subset of C can form a local cut in $G_*[P_*]$ (otherwise because $G_*[P_*]$ is a $\phi/3$ -expander, any cut with terminals on one side has a side of size at most 3α , resulting in a smaller $(3\alpha, c)$ -local cut).

We look at the induced subgraphs after removing C from $G_*[P_*]$. Note that since all terminals including t_P are on the same side of the cut and t_P connects to all the other terminals, there can be only one connected component containing all terminals.

Consider the two subgraphs induced by the two sides of the cut are disjoint. Suppose at least one of these two subgraphs contains more than one connected component. Let X be one of these connected components. Since $G_*[P_*]$ is connected, X is connected to the other side of the cut by edges in C . Removing edges connecting X and the other side will result in a proper subset of C that still forms a local cut (equivalent to moving X to the other side; because all terminals are in the same connected component, this resulting cut is still local). We obtained a contradiction. Thus, both subgraphs induced by the two sides of the cut are connected. $\square$

One-level Sparsifier Definition Since a global minimum cut is either a terminal cut or a local cut, in order to maintain a global minimum cut, it suffices to maintain a minimum terminal cut and a minimum local cut. Therefore, a one-level global minimum cut data structure has an internal one-level terminal edge connectivity data structure (to keep track of terminal cuts), and a priority queue Λ_{G_*} to store local cuts in G_* (to keep track of local cuts in G , as per Lemma 4.5). Here G_* is the auxiliary graph of G with respect to $\mathcal{P}$. We rigorously define the one level sparsifier data structure below.

DEFINITION 4.3. (ONE-LEVEL SPARSIFIER) *Given parameters $c, \phi, c' \geq c, \gamma \geq c'$, and $\alpha \geq \lceil c/\phi \rceil$, a one-level global minimum cut data structure w.r.t. these parameters consists of two parts:*

- (1) *A one-level terminal edge connectivity data structure $\mathbf{Sparsifier}_{c', \gamma}(G, \mathcal{P}, CC, F)$ (Section 3), which contains the input graph $G = (V, E)$, a vertex partition $\mathcal{P}$ of G given by a ϕ -expander decomposition, and a c' -cut containment set CC with respect to G and $\mathcal{P}$.*
- (2) *Assuming G_* is the auxiliary graph of G with respect to $\mathcal{P}$, a priority queue Λ_{G_*} that contains all vertex sets U such that*
 - *There exists a $P \in \mathcal{P}$ such that $U \subseteq P_*$, where P_* is the corresponding cluster of P in G_**
 - *U induces a connected subgraph in G_**
 - *$(U, P_* \setminus U)$ is a $(3\alpha, c)$ -local cut in $G_*[P_*]$. The terminal set is $(\text{End}(\partial_G(\mathcal{P})) \cap P) \cup \{t_P\}$.*

The elements in Λ_{G_} are ordered by the size of each cut $(U, P_* \setminus U)$.*

Note that, by storing a vertex subset U of G_* , we are essentially storing the cut $(U, P_* \setminus U)$. Since $(U, P_* \setminus U)$ is a cut in $G_*[P_*]$ with all boundary vertices on the same side, the cut-set of $(U, P_* \setminus U)$ induces a cut $(U, V_* \setminus U)$ in G_* , with the same size. By Lemma 4.6, all minimum $(3\alpha, c)$ -local cuts in $G_*[P_*]$ have two connected sides. Therefore, if Λ_{G_*} contains all *connected* U such that $(U, P_* \setminus U)$ is a $(3\alpha, c)$ -local cut, Λ_{G_*} contains all possible U such that $(U, P_* \setminus U)$ is a minimum $(3\alpha, c)$ -local cut. We can see Λ_{G_*} as a priority queue that allows us to keep track of minimum $(3\alpha, c)$ -local cuts in G_* .

REMARK 4.1. In the definition of the one-level data structure, we actually store c' -cuts in the terminal sparsifier for $c' \geq c$. This is because in the terminal sparsifier update algorithm (Lemma 3.3), the edge connectivity parameter for the sparsifier reduces after applying an update sequence to the input graph. However, this does not affect the query result – if the minimum terminal cut has a size at most c , it is also a minimum c' -terminal cut. We will refer to these cuts as “minimum c -cuts in $\text{Sparsifier}_{c',\gamma}(G, \mathcal{P}, CC, F)$ ”.

REMARK 4.2. Because $G_*[P_*]$ is an expander, α can be set to any number greater than c/ϕ . This does not change the vertex sets stored in Λ_{G_*} . In fact, in Section 5.1.2, we will set α to be $\lceil c/\phi \rceil$ for a simpler complexity expression.

Below (Lemma 4.7) we show that the highest priority item in the priority queue gives us a minimum local cut in G_* , and thus a minimum local cut in G (by Lemma 4.5). The global minimum cut of G can be restored by comparing the size of the minimum terminal cut found in the terminal cut edge connectivity data structure and the minimum local cut found in Λ_{G_*} .

LEMMA 4.7. Let $\text{Sparsifier}_{c',\gamma}(G, \mathcal{P}, CC, F)$ be a one-level terminal edge connectivity sparsifier data structure with parameters c', γ such that $c' \geq c$ and $\gamma \geq c'$. Let Λ_{G_*} be the priority queue defined in Definition 4.3. Let C_1 be the cut in G induced by the cut-set of a minimum c -cut in $\text{Sparsifier}_{c',\gamma}(G, \mathcal{P}, CC, F)$ (if one such cut exists in $\text{Sparsifier}_{c',\gamma}(G, \mathcal{P}, CC, F)$); otherwise $C_1 = \perp$.

Let U_2 be a vertex set in Λ_{G_*} with highest priority. If U_2 exists, let P_* be the cluster in G_* that contains U_2 and P be the cluster in $\mathcal{P}$ corresponding to P_* . We use C_2 to denote cut $(U_2 \cap V, V \setminus U_2 \cap V)$. If U_2 does not exist, $C_2 = \perp$.

If C_1 and C_2 are both $\perp$, then G does not have a c -cut. Otherwise, the smaller one between C_1 and C_2 is a minimum c -cut of G .

Proof. By Lemma 4.4, any minimum cut of G is either a terminal cut with $\text{End}(\partial_G(\mathcal{P}))$ as terminals, or a local cut within a $G[P]$ for some $P \in \mathcal{P}$ with respect to $\text{End}(\partial_G(\mathcal{P})) \cap P$, $\alpha \geq \lceil c/\phi \rceil$. By Lemma 4.2, C_1 is a minimum terminal cut in G .

By definition of Λ_{G_*} , it contains all $(3\alpha, c)$ -local cuts $(U, P_* \setminus U)$ in $G_*[P_*]$ such that U induces a connected subgraph. By Lemma 4.6, the minimum $(3\alpha, c)$ -local cut in G_* must have a side of volume at most 3α that is also connected. Then by Lemma 4.5, $(U_2 \cap P, P \setminus U_2 \cap P)$ is a minimum local cut in $G[P]$. Therefore, $(U_2 \cap V, V \setminus U_2 \cap V)$ is a minimum local cut in $G[\mathcal{P}]$. $\square$

In summary, a one-level global minimum cut sparsifier data structure is a tuple $(G, \mathcal{P}, CC, F, \Lambda_{G_*})$, where $(G, \mathcal{P}, CC, F)$ is a one-level terminal edge connectivity sparsifier, and Λ_{G_*} is a priority queue of vertex sets.

Multi-level Sparsifier Definition A multi-level global minimum cut data structure

$$\left\{ (G^{(j)}, \mathcal{P}^{(j)}, CC^{(j)}, F^{(j)}, \Lambda_{G_*^{(j)}}) \right\}_{j=1}^{\ell}$$

contains a set of ℓ one-level global minimum cut data structures such that conditions (c1), (c2) and (c3) in Section 3.3 hold.

For a multi-level global minimum cut data structure, if we collect all one-level terminal edge connectivity data structures inside each one-level global minimum cut data structure for all the levels, these one-level terminal edge connectivity data structures together form a multi-level terminal edge connectivity data structure. Thus, the multi-level minimum cut data structure can be viewed as a multi-level terminal edge connectivity data structure with additional priority queues on each level.

Throughout this section, we assume that the parameters c, ϕ, c', γ , and α defining the one-level and multi-level global minimum cut data structure are set such that $c' \geq c$ always holds, γ is always greater than c' , and $\alpha \geq \lceil c/\phi \rceil$.

5 Initialization and Update Algorithms for the Global Minimum Cut Data Structure

With the new global minimum cut data structure defined, we proceed with presenting the algorithms to initialize and update it. In Sections 5.1 and 5.2, we will discuss the algorithms for one-level data structure and multi-level data structure, respectively. In Section 5.3, we will use these algorithms to prove Theorem 1.1.

5.1 One-level Initialization and Update Algorithms

5.1.1 Subroutine: Cut Enumeration We begin with a subroutine to find all cuts in which a given vertex belongs to a side with bounded volume. In Section 5.1.2, we will use this subroutine to find all the local cuts.

We define a subroutine called **Enumerate-Cuts**, which is invoked in the one-level global minimum cut data structure initialization and update algorithms. **Enumerate-Cuts** takes arguments $G, \mathcal{P}, G_*, v, c, \alpha$, where $\mathcal{P}$ is a partition of G , $G_* = (V_*, E_*)$ is the auxiliary graph of G w.r.t. $\mathcal{P}$ (Definition 4.2), v is a vertex in G_* , c and α are positive integers. The returned value is a list of subsets of V_* that can represent a cut in G_* . We will later use these subsets to construct Λ_{G_*} .

Algorithm 1 is the detailed implementation of **Enumerate-Cuts**.

Algorithm 1: Cut Enumeration

Input: $G_* = (V_*, E_*)$
 v : a vertex in G_* ,
 α, c : positive integers
Output: $\mathcal{C}$: a list of subsets of V_*

```

1 Function Enumerate-Cuts( $G_*, v, \alpha, c$ ):
2    $T_{\mathcal{P}} \leftarrow$  the set of special terminals added when constructing  $G_*$ , as per Definition 4.2;
3    $\mathcal{C} \leftarrow \emptyset$ ;
4   Cut-Enumeration-Recursive( $G_*, v, \alpha, c, \emptyset, \mathcal{C}$ );
5   return  $\mathcal{C}$ 

6 Function Cut-Enumeration-Recursive( $G_*, v, \alpha, c, F, \mathcal{C}, T_{\mathcal{P}}$ ):
7   if  $c < 0$  then return;
8   Run a DFS from  $v$  in  $G_*$  until the DFS stops or  $\text{vol}_G(U) > 3\alpha$ , where  $U$  is the set of visited vertices;
9   if  $F$  is exactly the set of edges between  $U$  and  $V \setminus U$  in  $G_*$ ,  $\sum_{(u,w,\beta) \in F} \beta \leq c$ , and
10     $\text{vol}_G(U) + \sum_{(u,w,\beta) \in F} \beta \leq 3\alpha$  then
11     $\mathcal{C} \leftarrow \mathcal{C} \cup \{U\}$ ;
12    return
13   for every edge  $(u, w, \beta)$  of  $G_*$  such that  $u, w \in U$  and  $u, v \notin T_{\mathcal{P}}$  do
14      $F \leftarrow F \cup \{(u, w, \beta)\}$ ;
15     Remove edge  $(u, w, \beta)$  from  $G[\mathcal{P}]$ 
16     Cut-Enumeration-Recursive( $G_*, v, \alpha, c - \beta, F, \mathcal{C}$ );
17     Add edge  $(u, w, \beta)$  back to  $G[\mathcal{P}]$ ;

```

LEMMA 5.1. Let $G = (V, E)$ be a graph, $\mathcal{P}$ be a partition of G , G_* be the auxiliary graph of G w.r.t. $\mathcal{P}$. For a given vertex v in G and two integers $\alpha, c > 0$, Algorithm 1 **Enumerate-Cuts** finds all subsets of V_* , denoted by U , such that

- There exists a $P \in \mathcal{P}$ such that the corresponding P_* in G_* (Definition 4.2) contains U
- $v \in U$
- U induces a connected subgraph in G_*
- $(U, P_* \setminus U)$ is a $(3\alpha, c)$ -local cut in $G_*[P_*]$ with respect to terminal set $\text{End}(\partial_G(\mathcal{P})) \cap P \cup \{t_P\}$

The running time of the algorithm is $O(\alpha)^c$.

Proof. We check the requirements one by one.

- Since the DFS is conducted on G_* and U is the set of visited vertices, by the definition of P_* (Definition 4.2), there exists a P_* such that $U \subseteq P_*$.

- Since the DFS (line 8) starts from v , each U contains v .
- Since each U is obtained from a DFS, $G_*[U]$ is connected.
- - Since the DFS stops when the visited volume reaches 3α , the volume of U in G_* is at most 3α .
 - Since the number of edges between U and $V_* \setminus U$ is at most c (line 9), the size of $(U, V_* \setminus U)$ is at most c in the returned pairs.
 - Fix a vertex $t \in \text{End}(\partial_G(\mathcal{P})) \cap P$. (t, t_P) is not in the cut-set of $(U, V_* \setminus U)$ (line 12), t and t_P are on the same side of cut $(U, V_* \setminus U)$. Therefore, all terminals in $\text{End}(\partial_G(\mathcal{P})) \cap P \cup \{t_P\}$ are on the same side of the cut.

To show that all U satisfying the constraints are added to $\mathcal{C}$, let U' be a set of vertices satisfying the desired conditions, E' be the edge set between U' and $V_* \setminus U'$. We look at the recursion tree of **Cut-Enumeration-Recursive**. Since U induces a connected subgraph in G_* , no proper subset of E' form another cut in G_* . Since $v \in U'$ and the volume of U' in G_* is at most 3α , by performing a DFS from v until the visited volume is strictly greater than 3α , it is guaranteed that there exists an edge $e \in E'$ such that both endpoints of e are visited.

Suppose after a depth i recursion, i edges of E' are added to F . $E' \setminus F$ is the cut-set of cut $(U', V_* \setminus U')$ in graph $G_* \setminus F$. The algorithm performs another DFS from v that stops when the volume of visited vertices is greater than 3α . Since $\mathbf{vol}_{G_* \setminus F}(U') \leq \mathbf{vol}_{G_*}(U') \leq 3\alpha$, it is still guaranteed that there is at least one edge $e \in E' \setminus F$ such that both endpoints of e are visited. By induction, at the end of this chain of recursion, $F = \partial_{G_*}(U')$. Then, after one more recursion, U' is added to $\mathcal{C}$.

Now we prove the time complexity. In each recursion, the DFS takes $O(\alpha)$ time. The conditions on line 9 can be checked by running another DFS and takes another $O(\alpha)$ time. Since the volume of the visited vertices is at most α , at most $O(\alpha)$ edges are enumerated in line 12. Therefore, the recursion tree is an α -ary tree of depth at most c , and can be completed in time $O(\alpha)^c$. $\square$

5.1.2 One-level Initialization We initialize the global minimum cut data structure by separately initializing the underlying terminal edge connectivity sparsifier and Λ_{G^*} .

Algorithm 2: One-level Minimum Cut Data Structure Initialization

Input: G : a graph,

 $c, \phi, c', \gamma, \alpha$: parameters.1 **Function** Min-Cut-One-Level-Initialization:2 | $(G, \mathcal{P}, CC, F) \leftarrow \text{Edge-Connectivity-One-Level-Initialization}(G, c', \phi, \gamma);$ 3 **for** *all* $P \in \mathcal{P}$ **do**

4	Add vertex t_P to G ;
---	---------------------------

5	for each $t \in \text{End}(\partial_G(\mathcal{P})) \cap P$ do
---	--

6	Add edge (t_P, t) to G ;
---	------------------------------

7	$G_* = (V_*, E_*) \leftarrow$ the resulting graph;
---	--

8 **for** *all* $v \in V_*$ **do**

9	$\mathcal{C} \leftarrow \text{Enumerate-Cuts}(G_*, v, \alpha, c);$
---	--

10	for $U \in \mathcal{C}$ do
----	--

11 Insert U into Λ_{G_*} with priority $|\partial_{G_*}(U)|$ // $|\partial_{G_*}(U)|$ can be computed and returned by Enumerate-Cuts. We do not need to find all edges in the cut-set here

12	return $(G, \mathcal{P}, CC, F, \Lambda_{G_*})$;
----	--

LEMMA 5.2. *For a graph G with parameters c, ϕ, γ, α satisfying $c' \geq c$, $\gamma \geq c'$, and $\alpha \geq \lceil c'/\phi \rceil$, Algorithm 2 computes a one-level global minimum cut data structure of G w.r.t. parameters c, ϕ, c', γ , and α in time $m^{1+o(1)}O(\alpha)^{O(c')}$.*

Proof. By Lemma 3.2, **Edge-Connectivity-One-Level-Initialization** initializes the edge connectivity sparsifier correctly. We only need to show that after the execution of Algorithm 2, Λ_{G_*} is correctly initialized.

Recall the definition of Λ_{G_*} . It should contain exactly all subsets U such that

- There exists a $P \in \mathcal{P}$ such that $U \subseteq P_*$, where $P_* = P \cup \{t_P\}$
- $(U, P_* \setminus U)$ is an $(3\alpha, c)$ -local cut in $G_*[P_*]$ for some $\alpha \geq \lceil c'/\phi \rceil$. The terminal set is $\text{End}(\partial_G(\mathcal{P})) \cap P \cup \{t_P\}$.

Fix such a U with corresponding P_* . Let v be an arbitrary vertex in U . By Lemma 5.1, U is in the list of subsets returned by **Enumerate-Cuts** (line 9).

For any U that is not supposed to be in Λ_{G_*} . Lemma 5.1 guarantees that U would not be returned by **Enumerate-Cuts**.

For the time complexity, by Lemma 3.2, **Edge-Connectivity-One-Level-Initialization** takes $m^{1+o(1)}c'^{O(c')}/\text{poly}(\phi)$ time. Constructing G_* takes at most $O(m)$ time (line 3 to 6). Each invocation of **Enumerate-Cuts** runs in $O(\alpha)^c$ time, and returns $O(\alpha)^c$ vertex subsets. Therefore, the total time needed to insert all subsets into Λ_{G_*} is at most $n \cdot O(\alpha)^c \cdot O(c \log \alpha + \log n)$.

Therefore, the overall time complexity is

$$m^{1+o(1)}c'^{O(c')}/\text{poly}(\phi) + n \cdot O(\alpha)^c \cdot O(c \log \alpha + \log n) = m^{1+o(1)}O(\alpha)^{O(c')}$$

□

5.1.3 One-level Update When updates occur, in addition to updating the terminal cut sparsifier, we need to also update Λ_{G_*} . To achieve this, we remove and add some vertex sets and make sure that after the update, Λ_{G_*} still satisfies Definition 4.3 for the updated graph.

To achieve this, we select a list of vertices S , remove from Λ_{G_*} all U that contain at least one vertex in S . Then we run **Enumerate-Cuts** from every vertex in S and insert the returned vertex sets back to Λ_{G_*} . We observe that it suffices to let S contain only the following vertices:

1. Terminals vertices that switch to non-terminals after the updates, or non-terminals vertices that switch to terminals after the update. These are also the vertices returned by the terminal cut sparsifier update algorithm **Edge-Connectivity-One-Level-Update** (Lemma 3.3).
2. Vertices that are endpoints of edges involved in the update sequence.

We formally state and prove the observation above with the following lemma.

LEMMA 5.3. *Let G be a graph, and Λ_{G_*} be the priority queue in the one-level global minimum cut data structure of G with parameters c, ϕ, c', γ and α . After an update sequence updates G to G' , suppose an expander decomposition $\mathcal{P}$ of G is updated to $\mathcal{P}'$ of G' such that for each $P' \in \mathcal{P}'$, P' either contains only a single vertex or is a subset of some $P \in \mathcal{P}$ such that $G[P'] = G'[P']$. Let S be a list of vertices v that satisfy one of the three following conditions:*

- v is a terminal vertex before the update and a non-terminal after the update
- v is a non-terminal vertex before the update, and a terminal after the update
- v is an endpoint of an edge involved in the update sequence.

Let $\Lambda'_{G_*} = \{U \in \Lambda_{G_*} : U \cap S \neq \emptyset\}$, let Λ''_{G_*} be the set of all vertex set U' such that

- (a) $U' \cap S \neq \emptyset$
- (b) There exists $P' \in \mathcal{P}'$, such that $U' \subseteq P'_*$, where P'_* is the cluster corresponding to P' in $\mathcal{P}'$, per Definition 4.2.
- (c) U' induces a connected subgraph of G'_*
- (d) $(U', P'_* \setminus U')$ is a $(3\alpha, c)$ -local cut in $G'_*[P'_*]$ with respect to terminal set $\text{End}(\partial_{G'}(\mathcal{P}')) \cup \{t_{P'}\}$

$\Lambda_{G'_*} = \Lambda_{G_*} \setminus \Lambda'_{G_*} \cup \Lambda''_{G_*}$, ordered by the size of each $(U', P'_* \setminus U')$, is a valid priority queue in the one-level global minimum cut data structure of G' w.r.t parameters c, ϕ, c', γ and α .

Proof. After comparing Definition 4.3 to conditions (b), (c) and (d), we essentially need to prove

1. For any U' that should be in $\Lambda_{G'_*}$, if $U' \notin \Lambda''_{G_*}$, then $U' \in \Lambda_{G_*}$ and $U' \notin \Lambda'_{G_*}$.
 Fix a $U' \in \Lambda_{G'_*} \setminus \Lambda''_{G_*}$. It has to be true that $U' \cap S = \emptyset$. In this case, there exists a $P' \in \mathcal{P}'$ and a $P \in \mathcal{P}$ such that $U' \subseteq P'_* \subseteq P_*$. U' has the same volume in $G_*[P_*]$ and $G'_*[P'_*]$, and thus $U' \in \Lambda_{G_*}$. Since $U' \cap S = \emptyset$, $U' \notin \Lambda'_{G_*}$.
2. For any $U'' \in \Lambda_{G_*}$ that does not satisfy all of conditions (a) (b) (c) and (d), $U'' \in \Lambda'_{G_*}$.
 Fix a $U'' \in \Lambda_{G_*}$ that should not be in $\Lambda_{G'_*}$. We only need to show that $U'' \cap S \neq \emptyset$. Assume to the contrary. There exists a $P' \in \mathcal{P}'$ and a $P \in \mathcal{P}$ such that $U'' \subseteq P'_* \subseteq P_*$. Therefore, U'' induces the same subgraph in G_* and G'_* and should also be in a pair in $\Lambda_{G'_*}$. We reached a contradiction.

This concludes the proof. $\square$

REMARK 5.1. Since the update of Λ_{G_*} does not involve the terminal cut sparsifier, c' and γ can be any number satisfying $c' \geq c$ and $\gamma \geq c'$. c' and γ do not have to be the same for the data structures before and after the update.

We have an update routine in Algorithm 3:

Algorithm 3: Update One Level Minimum Cut Data Structure

Input: $(G, \mathcal{P}, CC, F, \Lambda_{G_*})$: one-level minimum cut data structure, w.r.t. parameters $c, \phi, c', \gamma, \alpha$

UpdateSeq: update sequence for multigraph G

Output: $(G', \mathcal{P}', CC', F', \Lambda_{G'_*})$: one-level minimum cut data structure for the up-to-date graph, w.r.t parameters $c, \phi/2^{O(\log^{1/3} n \log^{2/3} \log n)}, c, \gamma, \alpha$
UpdateSeq': update sequence that should be applied to the next sparsifier

```

1 Function Minimum-Cut-One-Level-Update:
2    $(G', \mathcal{P}', CC', F'), \text{UpdateSeq}', S \leftarrow$ 
   Edge-Connectivity-One-Level-Update $((G, \mathcal{P}, CC, T), \text{UpdateSeq}, c', \phi, \gamma);$ 
3   Add all vertices involved in UpdateSeq to  $S$ ;
   // Update  $G_*$ 
4   for all  $P' \in \mathcal{P}'$  such that  $P' \cap S \neq \emptyset$  do
5      $P \leftarrow$  the cluster in  $\mathcal{P}$  such that  $P' \subseteq P$ , as per Lemma 3.3;
6     Remove  $t_P$  and all incident edges from  $G_*$ ;
7     Add vertex  $t_{P'}$  to  $G_*$ ;
8     for all  $t \in \text{End}(\partial_{G'}(\mathcal{P}')) \cap P'$  do
9       Add edge  $(t, t_{P'})$  to  $G_*$ ;
10   $G'_* = (V'_*, E'_*) \leftarrow$  the resulting graph;
   // Update  $\Lambda_{G_*}$ 
11  for all  $v \in S$  do
12     $P' \leftarrow$  the cluster in  $\mathcal{P}'$  that contains  $v$ ;
13    Remove from each  $\Lambda_{G_*}$  all  $U$  such that  $v \in U$ ;
14     $\mathcal{C} \leftarrow \text{Enumerate-Cuts}(G'_*, v, \alpha, c);$ 
15    for each  $U \in \mathcal{C}$  do
16       $j \leftarrow$  the size of  $(U, V \setminus U)$ ;
17      Insert  $U$  into  $\Lambda_{G_*}$  with priority  $j$ 
18  Change the name of  $\Lambda_{G_*}$  to  $\Lambda_{G'_*}$ 
19  return  $(G', \mathcal{P}', CC', F', \Lambda_{G'_*}), \text{UpdateSeq}'$ 

```

LEMMA 5.4. Given a one-level minimum cut data structure $(G, \mathcal{P}, CC, F, \Lambda_{G_*})$ with parameters $c, \phi, c', \gamma, \alpha$ satisfying the following inequalities

$$c' = c^2 + 2c, \gamma > c', \alpha \geq (c^2 + 2c) \cdot 2^{O(\log^{1/3} n \log \log n)} / \phi, \text{ and } \phi \leq \frac{1}{2^{\log^{3/4} n}},$$

and an update sequence **UpdateSeq** of G which consists of vertex/edge insertions/deletions, there is an algorithm **Minimum-Cut-One-Level-Update** of running time $O(|\text{UpdateSeq}|) \cdot O(\alpha)^{O(c^2)}$ which updates the data structure to $(G', \mathcal{P}', CC', F', \Lambda_{G'_*})$ with parameters $c, \phi/2^{O(\log^{1/3} n \log \log n)}$, c, γ, α , and outputs an update sequence **UpdateSeq'** satisfying the following properties:

1. G' is the resulted graph of applying **UpdateSeq** to G .
2. $\mathcal{P}'$ is a $\phi/2^{O(\log^{1/3} n \log \log n)}$ -expander decomposition of G' . Furthermore, each cluster $P' \in \mathcal{P}'$ either contains only a single vertex or is a subset of a cluster $P \in \mathcal{P}$ such that $G'[P'] = G[P]$.
3. CC' is a c -cut containment set for G' and $\mathcal{P}'$.
4. F' is an unweighted spanning forest of $G' \setminus CC'$.
5. The sparsifier maintained in the data structure is $\text{Sparsifier}_{c, \gamma}(G', \mathcal{P}', CC', F')$.
6. **UpdateSeq'** updates $\text{Sparsifier}_{c', \gamma}(G, \mathcal{P}, CC, F)$ to $\text{Sparsifier}_{c, \gamma}(G', \mathcal{P}', CC', F')$. The length of **UpdateSeq'** is at most $|\text{UpdateSeq}| (10c)^{O(c)}$.
7. $\Lambda_{G'_*}$ is a priority queue containing every vertex set U' such that
 - There exists a $P' \in \mathcal{P}'$ such that $U' \subseteq P'_*$, where P'_* is the corresponding cluster of P' in $G'_* = (V'_*, E'_*)$.
 - U' induces a connected subgraph in G'_* .
 - $(U', P'_* \setminus U')$ is an $(3\alpha, c)$ -local cut in $G'_*[P'_*]$ for some $\alpha \geq \lceil c/\phi \rceil$. The terminal set is $\text{End}(\partial_{G'}(P')) \cap P' \cup \{t_{P'}\}$.

The priority of vertex subsets in $\Lambda_{G'_*}$ is given by the size of $(U', V'_* \setminus U')$ in increasing order.

Proof. Properties 1, 2, 3, 4, 5, 6 are proven in Lemma 3.3. For a $P' \in \mathcal{P}'$, by 2, there is a $P \in \mathcal{P}$ such that $P' \subseteq P$. If P' does not contain any terminal vertex that is a non-terminal before the update, or any non-terminal vertex that is a terminal before the update, or any vertex that is involved in the update sequence, then $P' = P$ and P' and P have the same set of terminals. The c -cuts in P' are the same as those in P . Therefore, in order to update G_* and Λ_{G_*} , we only need to consider clusters $P' \in \mathcal{P}'$ that contains at least one vertex in S (line 4). For these clusters, we remove the old special terminal and add new special terminals for each of them (line 4 to 9). Since these are all of the clusters affected by the update, in line 10, G'_* is same as the graph obtained by directly adding special terminals and edges to the up-to-date graph $G'[\mathcal{P}']$.

Since $\Lambda_{G'_*}$ is obtained by removing from Λ_{G_*} all U such that U contains at least one vertex in S and then adding all new U' returned by running **Enumerate-Cuts** from vertices in S , by Lemma 5.3, the resulting data structures satisfy Definition 4.3.

For the time complexity, Lemma 3.3 shows that **Edge-Connectivity-One-Level-Update** runs in $O(|\text{UpdateSeq}| (c/\phi)^{O(c^2)})$ time (because c' is $O(c^2)$), and that the vertex set S returned by this subroutine contains at most $O(|\text{UpdateSeq}|)$ vertices. After adding vertices involved in **UpdateSeq** to S (line 3), S still has $O(|\text{UpdateSeq}|)$ vertices. Therefore, line 9 is executed for at most $O(|\text{UpdateSeq}|)$ times. The overall time for updating G_* is at most $O(|\text{UpdateSeq}|)$.

The block from line 11 to line 17 is executed for at most $O(|S|) = O(|\text{UpdateSeq}|)$ times. Since for a fixed v , every U that contains v can be enumerated by running **Enumerate-Cuts** again on a copy of the data structure before the update, line 13 takes $O(\alpha)^c \cdot O(c \log \alpha)$ time. Since $\alpha > c/\phi$ and $0 < \phi < 1$, $O(\alpha)^c \cdot O(c \log \alpha) = O(\alpha)^{O(c)}$. By Lemma 5.1, the invocation of **Enumerate-Cuts** on line 14 takes $O(\alpha)^{O(c)}$ time. By an argument similar to Lemma 5.2, the block from line 15 to 17 takes $O(\alpha)^{O(c)}$ time. Therefore, the block updating Λ_{G_*} takes $O(|\text{UpdateSeq}| O(\alpha)^{O(c)})$ time. Since $\alpha > c/\phi$, the overall time complexity of Algorithm 3 is $O(|\text{UpdateSeq}| O(\alpha)^{O(c^2)})$. $\square$

5.2 Multi-level Initialization and Update Algorithms With one-level initialization and update algorithms, we can define the multi-level initialization and update algorithms.

LEMMA 5.5. *For a graph G with n vertices and m edges and parameters $c, \phi, c', \gamma, \alpha$ satisfying the following inequalities*

$$c' > c, \gamma > c', \alpha \geq c/\phi, \text{ and } \phi \leq \frac{1}{2^{\log^{3/4} n}}$$

Algorithm 4 constructs an at most $O(\log^{1/10} n)$ -level global minimum cut data structure for G with parameters $c, \phi, c', \gamma, \alpha$ in time

$$m^{1+o(1)} \cdot O(\alpha)^{O(c')}$$

Algorithm 4: Multi-Level Minimum Cut Data Structure Initialization

Input: G : graph before update,

$c, \phi, c', \gamma, \alpha$: parameters.

Output: $\{(G^{(i)}, \mathcal{P}^{(i)}, CC^{(i)}, F^{(i)}, \Lambda_{G_*^{(i)}})\}_{i=1}^\ell$: multi-level minimum cut data structure

```

1 Function Multi-Level-Min-Cut-Initialization( $G, c, \phi, c', \alpha, \gamma$ ):
2    $\ell \leftarrow 0, G^{(0)} \leftarrow G$ ;
3   repeat
4      $(G^{(\ell)}, \mathcal{P}^{(\ell)}, CC^{(\ell)}, F^{(\ell)}, \Lambda_{G_*^{(\ell)}}) \leftarrow \text{Min-Cut-One-Level-Initialization}(G^{(\ell)}, c, \phi, c', \gamma, \alpha)$ ;
5      $G^{(\ell+1)} \leftarrow \text{Sparsifier}_{c', \gamma}(G^{(\ell)}, \mathcal{P}^{(\ell)}, CC^{(\ell)}, F^{(\ell)})$ ;
6      $\ell \leftarrow \ell + 1$ ;
7   until  $\mathcal{P}^{(\ell-1)}$  contains only one cluster;
8   return  $\{(G^{(i)}, \mathcal{P}^{(i)}, CC^{(i)}, F^{(i)}, \Lambda_{G_*^{(i)}})\}_{i=0}^{\ell-1}$ 

```

Proof. By the definition of the multi-level global minimum cut data structure and the algorithm, the output is a multi-level global minimum cut data structure for the input graph with parameters c, ϕ, γ , and α . By Theorem 3.1, ℓ is at most $O(\log^{1/10} n)$.

By Lemma 5.2, the initialization of a one-level minimum cut data structure takes

$$m^{1+o(1)} \cdot O(\alpha)^{O(c')}$$

time for a graph with m edges. Since the multi-level data structure has at most $O(\log^{1/10} n)$ levels, **Min-Cut-One-Level-Initialization** is called at most $O(\log^{1/10} n)$ times, the total time of executing **Min-Cut-One-Level-Initialization** is $O(m^{1+o(1)} \log n) \cdot O(\alpha)^{O(c')}$. Therefore, the total running time of the algorithm is

$$O(m^{1+o(1)}) \cdot O(\alpha)^{O(c')}$$

□

In our multi-level update algorithm, for a given update sequence, we use the one level update algorithm to update the minimum cut data structure for each level starting from level 0. After the update on level i , the algorithm returns a new update sequence for the level $i + 1$. If at some level, the update sequence contains too many updates such that the resulted sparsifier is too large, we reconstruct the sparsifier to make sure that all the sparsifiers get smaller as the level increases.

LEMMA 5.6. *Given a multi-level minimum cut data structure $\{(G^{(i)}, \mathcal{P}^{(i)}, CC^{(i)}, F^{(i)}, \Lambda_{G_*^{(i)}})\}_{i=0}^\ell$ with parameters $c, \phi, c', \alpha, \gamma$ satisfying the following inequalities*

$$c' = c^2 + 2c, \gamma > c', \alpha \geq (c^2 + 2c) \cdot 2^{O(\log^{1/3} n \log \log n)} / \phi, \text{ and } \phi \leq \frac{1}{2^{\log^{3/4} n}}$$

Algorithm 5: Multi-Level Minimum Cut Data Structure Update

Input: $\left\{G^{(i)}, \mathcal{P}^{(i)}, CC^{(i)}, F^{(i)}, \Lambda_{G_*^{(i)}}\right\}_{i=0}^{\ell}$: multi-level global cut data structure,
UpdateSeq: a sequence of update to graph $G^{(0)}$,
 $c, \phi, c' = c^2 + 2c, \gamma, \alpha$: parameters of the multi-level global cut data structure
Output: $\left\{G'^{(i)}, \mathcal{P}'^{(i)}, CC'^{(i)}, F'^{(i)}, \Lambda_{G_*'^{(i)}}\right\}_{i=0}^{\ell'}$: multi-level global cut data structure for the graph after applying **UpdateSeq**

```

1 Function Multi-level-Update():
2    $i \leftarrow 0, \text{UpdateSeq}^{(0)} \leftarrow \text{UpdateSeq};$ 
3   while  $\text{UpdateSeq}^{(i)}$  contains no more than  $n^{(i)} / \log n$  updates, where  $n^{(i)}$  is the number of vertices in
    $G^{(i)}$  do
4      $(G'^{(i)}, \mathcal{P}'^{(i)}, CC'^{(i)}, F'^{(i)}, \Lambda_{G_*'^{(i)}}), \text{UpdateSeq}^{(i+1)} \leftarrow \text{Minimum-Cut-One-Level-Update}$ 
        $((G^{(i)}, \mathcal{P}^{(i)}, CC^{(i)}, F^{(i)}, \Lambda_{G_*^{(i)}}), \text{UpdateSeq}^{(i)}, c, \phi, c^2 + 2c, \gamma, \alpha);$ 
5      $i \leftarrow i + 1;$ 
6      $\phi \leftarrow \phi / 2^{O(\log^{1/3} n \log^{2/3} \log n)};$ 
7      $\left\{G'^{(j)}, \mathcal{P}'^{(j)}, CC'^{(j)}, F'^{(j)}, \Lambda_{G_*'^{(j)}}\right\}_{j=i}^{\ell'} \leftarrow \text{Multi-Level-Min-Cut-Initialization}(G'^{(i)}, c, \phi, c, \gamma, \alpha);$ 
8   return  $\left\{G'^{(i)}, \mathcal{P}'^{(i)}, CC'^{(i)}, F'^{(i)}, \Lambda_{G_*'^{(i)}}\right\}_{i=0}^{\ell'};$ 

```

such that the data structure is updated using at most $O(\log \log n)$ update sequences since initialized, and an update sequence **UpdateSeq** of graph $G^{(0)}$, then Algorithm 5 update the multi-level minimum cut data structure of $O(\log^{1/10} n \log \log n)$ levels with respect to the given update sequence with parameters $c, \phi / 2^{O(\log^{1/3} n \log^{2/3} \log n)}, \gamma, \alpha$ in time

$$O(|\text{UpdateSeq}| O(\alpha)^{O(c^2 \log^{1/10} n \log \log n)})$$

Proof. The correctness of the algorithm is implied by Lemma 5.4 and Lemma 5.5. Since the multi-level data structure has $O(\log^{1/10} n)$ levels after initialization (Lemma 5.5), and after applying an update sequence, the number of levels increases by at most $O(\log^{1/10} n)$ (new layers are added in line 7 by running the initialization algorithm again), the total number of levels is $O(\log^{1/10} n \log \log n)$ after applying $O(\log \log n)$ update sequences to the graph.

Now we bound the running time. By Lemma 5.4, at level i , the update sequence has length at most $|\text{UpdateSeq}|(10c)^{O(i \cdot c)}$, and the running time of **Minimum-Cut-One-Level-Update** for level i is at most

$$O(|\text{UpdateSeq}|) \cdot (10c)^{O(i \cdot c)} \cdot O(\alpha)^{O(c^2)} = O(|\text{UpdateSeq}|) \cdot O(\alpha)^{O(i \cdot c^2)}$$

Together with the fact that the minimum cut data structure has at most $O(\log^{1/10} n \log \log n)$ levels, the overall running time is $O(|\text{UpdateSeq}| O(\alpha)^{O(c^2 \log^{1/10} n \log \log n)})$. $\square$

5.3 Returning the Global Minimum Cut and Proof of Theorem 1.1

LEMMA 5.7. For any $c = (\log n)^{o(1)}$, there is a fully dynamic algorithm **Dynamic-Min-Cut-DS** which maintains a set of multi-level global minimum cut data structure such that each maintained multi-level global minimum cut data structure has $O(\log^{1/10} n \log \log n)$ levels all the time, and after processing each update, the algorithm provides the access to one of the maintained multi-level data structure $\left\{G^{(i)}, \mathcal{P}^{(i)}, CC^{(i)}, F^{(i)}, \Lambda_{G_*^{(i)}}\right\}_{i=0}^{\ell}$ with parameters c, ϕ, c', γ , and α for the up-to-date graph satisfying the following conditions:

$$\phi = 1/n^{o(1)}, c' \geq c, \gamma > c', \text{ and } \alpha \geq c'/\phi$$

The initialization time of the algorithm is $m^{1+o(1)}$, and the update time of the algorithm is $n^{o(1)}$ per update.

Proof. Suppose parameter ζ is defined as

$$\zeta =: \left\lfloor \log \left(\frac{\log \log n / 100}{\log(4c)} \right) \right\rfloor.$$

Since $c = (\log n)^{o(1)}$, we have $\zeta = \omega(1)$ and $\zeta = O(\log \log n)$.

We define parameters c_i, ϕ_i for all $0 \leq i \leq \zeta$, and γ, α such that the parameters for a multi-level minimum cut data structure are $c, \phi_0, c_0, \gamma, \alpha$ after initialization, and after executing Algorithm **Multi-level-Update** on the multi-level minimum cut data structure i times, the parameters for the multi-level minimum cut data structure are $c, \phi_i, c_i, \gamma, \alpha$. We set c_i as follows

$$c_\zeta =: c \text{ and } c_i =: c_{i+1}(c_{i+1} + 2) \text{ for } 0 \leq i < \zeta.$$

We set ϕ_i as follows

$$\phi_0 =: \frac{1}{2^{\log^{3/4} n}}, \phi_i =: \frac{\phi_{i-1}}{2^{\delta \log^{1/3} n \log^{2/3} \log n}} \text{ for } 1 \leq i \leq \zeta, \text{ where } \delta \text{ is a constant.}$$

We set γ, α as

$$\gamma =: c_0 + 1, \alpha =: \lceil c_0 / \phi_\zeta \rceil$$

We have the following observations for our parameters

- $c_i < (4c)^{2^\zeta} = \log^{1/100} n$ for any $0 \leq i \leq \zeta$.
- $\phi_i \leq \frac{1}{2^{\log^{3/4} n}}$ and $\phi_i = \frac{1}{2^{O(\log^{3/4} n)}}$ for any $0 \leq i \leq \zeta$.
- $\gamma > c_i, \alpha \geq c_i / \phi_i$ for any $0 \leq i \leq \zeta$.

By Lemma 5.5 and Lemma 5.6, if we have a multi-level minimum cut data structure initialized by Algorithm **Multi-Level-Min-Cut-Initialization** with parameters $c, \phi_0, c_0, \gamma, \alpha$, and the data structure has been updated i times by Algorithm **Multi-level-Update** for some $i \leq \zeta - 1$ such that after j -th for each $1 \leq j \leq i$, the parameters of the data structures are $c, \phi_j, c_j, \gamma, \alpha$, then if we use Algorithm **Multi-level-Update** to update the data structure again, the parameters of the data structures are $c, \phi_{i+1}, c_{i+1}, \gamma, \alpha$. By our choice of parameters and the time complexity of Lemma 5.5 and 5.6, since $c_i < \log^{1/100} n$ and $\phi_i = 1/2^{O(\log^{3/4} n)}$, $O(\alpha)^{O(c^2 \log^{1/10} n \log \log n)}$ is always $n^{o(1)}$. Therefore, **Multi-Level-Min-Cut-Initialization** can initialize the data structure in $m^{1+o(1)}$ time and **Multi-level-Update** can handle a sequence of updates with $n^{o(1)}$ time per update.

By our selection of parameters and Lemma C.1, we obtained a fully dynamic algorithm with $m^{1+o(1)}$ initialization time and $n^{o(1)}$ update time. $\square$

Now we prove Theorem 1.1 using Algorithm **Fully-Dynamic-Min-Cut** (Algorithm 6). In our dynamic algorithm, we maintain a set of multi-level minimum cut data structures such that after processing each update, the algorithm returns the access to one of the maintained multi-level minimum cut data structure with parameters $c, \phi, c', \alpha, \gamma$ which is for the up-to-date graph. Then for each one-level data structure $(G^{(i)}, \mathcal{P}^{(i)}, CC^{(i)}, F^{(i)}, \Lambda_{G_*^{(i)}})$, one of the following cases hold:

1. The cut-set of any minimum cut of size c for graph $G^{(i)}$ is also the cut-set of a minimum cut for G .
2. The minimum cut of $G^{(i)}$ is of size greater than c .

Furthermore, by Lemma 4.7, the minimum cut of $G^{(i)}$ can be obtained by finding the smallest cut from

- The cut induced by the cut-set of a minimum cut in $\text{Sparsifier}_{c, \gamma}(G^{(i)}, \mathcal{P}^{(i)}, CC^{(i)}, F^{(i)})$, which is also $G^{(i+1)}$
- A cut $(U \cap V, V_* \setminus U \cap V)$, where U is the vertex set in $\Lambda_{G_*^{(i)}}$ with highest priority.

Recall that terminals in one level minimum cut data structures are boundary vertices of the expander decomposition. When a terminal cut exists in the one-level minimum cut data structure for level i , there are more than one cluster in the expander decomposition in level i . Therefore, we can retrieve the minimum cut from sparsifiers of the current level, i.e., the one-level minimum cut data structure for level $i + 1$. We have the following algorithm to return the minimum cut using the multi-level minimum cut data structure.

Algorithm 6: Fully Dynamic Minimum Cut

Input: **Update:** a multigraph vertex/edge insertion/deletion update

```
1 Function Fully-Dynamic-Min-Cut:
2    $\left\{G^{(i)}, \mathcal{P}^{(i)}, CC^{(i)}, F^{(i)}, \Lambda_{G_*^{(i)}}\right\}_{i=0}^{\ell} \leftarrow \text{Dynamic-Min-Cut-DS}(\text{UpdateSeq});$ 
3   for  $i \leftarrow \ell$  down to 0 do
4      $U \leftarrow$  highest priority item in  $\Lambda_{G_*^{(i)}}$ ;
5      $A_i \leftarrow$  edges between  $U$  and  $V \setminus U$  or  $\perp$  if it doesn't exist;
6     if  $i < \ell$  and ( $A_i = \perp$  or  $|A_i| > |A_{i+1}|$ ) then
7        $A_i \leftarrow A_{i+1}$ ;
8   if  $A_0 = \perp$  then
9     return  $\emptyset$ 
10  else
11    return  $A_0$ 
```

Proof. [Proof of Theorem 1.1] We first prove the correctness of Algorithm 6. By Lemma 5.7, the multi-level minimum cut data structure obtained is for the up-to-date graph with $\ell = O(\log^{1/10} n \log \log n)$ levels.

We use induction to show that for any $0 \leq i \leq \ell$, A_i is the cut-set of a minimum cut of $G^{(i)}$ if the minimum cut of $G^{(i)}$ is of size at most c , otherwise, A_i is $\perp$. Since $G^{(\ell)}$ only contains a single cluster, all the vertices of $G^{(\ell)}$ are non-terminal vertices. Therefore, if a minimum cut of size $j \leq c$ exists, a side of this cut is stored in $\Lambda_{G_*^\ell}$. Hence, A_ℓ is the cut-set of a minimum cut of $G^{(\ell)}$ if the minimum cut of $G^{(\ell)}$ is of size at most c , otherwise, A_ℓ is $\perp$.

For any $0 \leq i < \ell$, by Lemma 4.7, if the minimum cut of $G^{(i)}$ is of size at most c , then it is either obtained from a minimum cut in the sparsifier (A_{i+1} here), or a vertex set with the lowest priority in $\Lambda_{G_*^{(i)}}$ (line 4 and 5). In line 6, the algorithm sets A_i to be the smaller of the two cuts above. Therefore, A_i is the cut-set of a minimum cut of $G^{(i)}$ if the minimum cut of $G^{(i)}$ is of size at most c , otherwise, A_i is $\perp$.

To bound the running time, we note that for any $0 \leq i \leq \ell$, the minimum cut from $\Lambda_{G_*^{(i)}}$ can be obtained in $O(c \log n)$ time. Since the multi-level data structure has $O(\log^{1/10} n \log \log n)$ levels, the running time to find the cut-set of minimum c -cut is polylogarithmic. By Lemma 5.7, the processing time for each update is $n^{o(1)}$. $\square$

References

- [1] Anders Aaman, Adam Karczmarz, Jakub Lacki, Nikos Parotsidis, Peter MR Rasmussen, and Mikkel Thorup. Optimal decremental connectivity in non-sparse graphs. *arXiv preprint arXiv:2111.09376*, 2021.
- [2] Parinya Chalermsook, Syamantak Das, Yunbum Kook, Bundit Laekhanukit, Yang P Liu, Richard Peng, Mark Sellke, and Daniel Vaz. Vertex sparsification for edge connectivity. In *Symposium on Discrete Algorithms (SODA)*, pages 1206–1225. SIAM, 2021.
- [3] 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 *Symposium on Foundations of Computer Science (FOCS)*. IEEE, 2020.
- [4] David Eppstein, Zvi Galil, Giuseppe F Italiano, and Amnon Nissenzweig. Sparsification-a technique for speeding up dynamic graph algorithms. *Journal of the ACM (JACM)*, 44(5):669–696, 1997.
- [5] Greg N Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM Journal on Computing*, 14(4):781–798, 1985.
- [6] Greg N Frederickson. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. *SIAM Journal on Computing*, 26(2):484–538, 1997.
- [7] Harold N. Gabow. A matroid approach to finding edge connectivity and packing arborescences. *Journal of Computer and System Sciences*, 50(2):259–273, 1995.
- [8] Zvi Galil and Giuseppe F Italiano. Fully dynamic algorithms for 3-edge connectivity. *Unpublished manuscript*, 1991.

- [9] Zvi Galil and Giuseppe F Italiano. Fully dynamic algorithms for edge connectivity problems. In *Symposium on Theory of Computing (STOC)*, pages 317–327, 1991.
- [10] Gramoz Goranci, Monika Henzinger, Danupon Nanongkai, Thatchaphol Saranurak, Mikkil Thorup, and Christian Wulff-Nilsen. Fully dynamic exact edge connectivity in sublinear time. In *Symposium on Discrete Algorithms (SODA)*, pages 70–86. SIAM, 2023.
- [11] Gramoz Goranci, Harald Räcke, Thatchaphol Saranurak, and Zihan Tan. The expander hierarchy and its applications to dynamic graph algorithms. In *Symposium on Discrete Algorithms (SODA)*, pages 2212–2228. SIAM, 2021.
- [12] Frank Harary. *Graph theory*. Addison-Wesley, Reading, MA, 1969.
- [13] Monika R Henzinger and Valerie King. Randomized dynamic graph algorithms with polylogarithmic time per operation. In *Symposium on Theory of Computing (STOC)*, pages 519–527, 1995.
- [14] Monika R Henzinger and Valerie King. Fully dynamic 2-edge connectivity algorithm in polylogarithmic time per operation. *SRC Technical Note*, 4, 1997.
- [15] Monika R Henzinger and Valerie King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *Journal of the ACM (JACM)*, 46(4):502–516, 1999.
- [16] Monika R Henzinger and Mikkil Thorup. Sampling to provide or to bound: With applications to fully dynamic graph algorithms. *Random Structures & Algorithms*, 11(4):369–379, 1997.
- [17] Jacob Holm, Kristian de Lichtenberg, and Mikkil Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM (JACM)*, 48(4):723–760, 2001.
- [18] Jacob Holm, Eva Rotenberg, and Mikkil Thorup. Dynamic bridge-finding in $\tilde{O}(\log^2 n)$ amortized time. In *Symposium on Discrete Algorithms (SODA)*, pages 35–52. SIAM, 2018.
- [19] Wenyu Jin and Xiaorui Sun. Fully dynamic st edge connectivity in subpolynomial time. In *Symposium on Foundations of Computer Science (FOCS)*, pages 861–872. IEEE, 2021.
- [20] Bruce M Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *Symposium on Discrete Algorithms (SODA)*, pages 1131–1142. SIAM, 2013.
- [21] Casper Kejlberg-Rasmussen, Tsvi Kopelowitz, Seth Pettie, and Mikkil Thorup. Faster worst case deterministic dynamic connectivity. In *European Symposium on Algorithms (ESA)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.
- [22] Robert Krauthgamer and Inbal Rika. Mimicking networks and succinct representations of terminal cuts. In *Symposium on Discrete Algorithms (SODA)*, page 1789–1799. SIAM, 2013.
- [23] Danupon Nanongkai and Thatchaphol Saranurak. Dynamic spanning forest with worst-case update time: adaptive, Las Vegas, and $O(n^{1/2-\epsilon})$ -time. In *Symposium on Theory of Computing (STOC)*, pages 1122–1129. ACM, 2017.
- [24] Danupon Nanongkai, Thatchaphol Saranurak, and Christian Wulff-Nilsen. Dynamic minimum spanning forest with subpolynomial worst-case update time. In *Symposium on Foundations of Computer Science (FOCS)*, pages 950–961. IEEE, 2017.
- [25] Thatchaphol Saranurak. A simple deterministic algorithm for edge connectivity. In *Symposium on Simplicity in Algorithms (SOSA)*, pages 80–85. SIAM, 2021.
- [26] Thatchaphol Saranurak and Di Wang. Expander decomposition and pruning: Faster, stronger, and simpler. In *Symposium on Discrete Algorithms (SODA)*, pages 2616–2635. SIAM, 2019.
- [27] Mikkil Thorup. Near-optimal fully-dynamic graph connectivity. In *Symposium on Theory of Computing (STOC)*, pages 343–350. ACM, 2000.
- [28] Mikkil Thorup. Fully-dynamic min-cut. *Combinatorica*, 27(1):91–127, 2007.
- [29] Mikkil Thorup and David R Karger. Dynamic graph algorithms with applications. In *Scandinavian Workshop on Algorithm Theory*, pages 1–9. Springer, 2000.
- [30] Jeffery Westbrook and Robert E Tarjan. Maintaining bridge-connected and biconnected components on-line. *Algorithmica*, 7(1-6):433–464, 1992.
- [31] Christian Wulff-Nilsen. Faster deterministic fully-dynamic graph connectivity. In *Symposium on Discrete Algorithms (SODA)*, pages 1757–1769. SIAM, 2013.
- [32] Christian Wulff-Nilsen. Fully-dynamic minimum spanning forest with improved worst-case update time. In *Symposium on Theory of Computing (STOC)*, pages 1130–1143. ACM, 2017.

A Dynamic Simple Graph to Dynamic Multigraph with A Constant Number of Neighbors

Let $\overline{G} = (\overline{V}, \overline{E})$ be the original dynamic simple graph with an arbitrary degree. For a fixed positive integer c , we use the degree reduction technique [12] to transform $\overline{G}$ to a multigraph $G = (V, E)$ such that every vertex has at most a constant number of distinct neighbors as follows:

1. The vertex set V of G is

$$V = \{v_{u,w} : (u, w) \in \overline{E}\} \cup \{v_{u,u} : u \in \overline{V}\}.$$

2. For any edge $(u, w) \in \overline{E}$, add edge $(v_{u,w}, v_{w,u}, 1)$ to E .
3. For every vertex u of $\overline{G}$ with degree at least 1, let $w_{u,0} = u$ and $w_{u,1}, w_{u,2}, \dots, w_{u,\deg(u)}$ be the neighbors of u in $\overline{G}$. Add edge $(v_{u,w_{u,i}}, v_{u,w_{u,i+1}}, c+1)$ to E for every $0 \leq i < \deg(u)$.

To maintain the correspondence between $\overline{G}$ and G , for every $u \in \overline{V}$, we maintain the list of $w_{u,0}, \dots, w_{u,\deg(u)}$.

Suppose $\overline{E}' \subseteq \overline{E}$ is the cut-set of a cut $\overline{C}$ in $\overline{G}$ with cut size at most c . One can verify that the edge set

$$E' = \{(v_{u,w}, v_{w,u}, 1) : (u, w) \in \overline{E}'\}$$

is a subset of edges of E , and E' is the cut-set of a cut C in G . The cut size of C in G is the same as the cut size of $\overline{C}$ in $\overline{G}$.

Similarly, suppose E' is the cut-set of a cut C in G with cut size at most c . Then every edge in E' is an edge of form $(v_{u,w}, v_{w,u}, 1)$ for some $u, w \in \overline{V}$. Let $\overline{E}'$ be the edge subset defined as

$$\overline{E}' = \{(u, v) : (v_{u,w}, v_{w,u}, 1) \in E'\}.$$

$\overline{E}'$ is the cut-set of a cut $\overline{C}$ in $\overline{G}$. The cut size of $\overline{C}$ in $\overline{G}$ is the same as the cut size of C in G . Hence, there is a bijection between all the c -cuts in $\overline{G}$ and all the c -cuts in G .

If edge (u, w) is inserted to graph $\overline{G}$, then we update graph G and the neighbor lists for the vertices of $\overline{G}$ as follows: Let u' be the last element in the neighbor list of u , and w' be the last element in the neighbor list of w . We apply the following updates to multigraph G

$$\text{insert}(v_{u,w}), \text{insert}(v_{w,u}), \text{insert}(v_{u,w}, v_{w,u}, 1), \text{insert}(v_{u,w}, v_{u,u'}, c+1), \text{insert}(v_{w,u}, v_{w,w'}, c+1),$$

and add w to the end of the neighbor list of u , and add u to the end of the neighbor list of w .

If edge (u, w) is deleted from graph $\overline{G}$, then we update graph G and the neighbor lists for the vertices of $\overline{G}$ as follows: Let u' be the previous element of w in the neighbor list of u , u'' be the next element after w in the neighbor list of u if exists, w' be the previous element in the neighbor list of w , and w'' be the next element in the neighbor list of w if exists. We apply the following updates to multigraph G

$$\begin{aligned} &\text{delete}(v_{u,w}, v_{w,u}, 1), \text{delete}(v_{u,w}, v_{u,u'}, c+1), \text{delete}(v_{u,w}, v_{u,u''}, c+1), \text{delete}(v_{w,u}, v_{w,w'}, c+1), \\ &\text{delete}(v_{w,u}, v_{w,w''}, c+1), \text{insert}(v_{w,w'}, v_{w,w''}, c+1), \text{insert}(v_{u,u'}, v_{u,u''}, c+1), \text{delete}(v_{u,w}), \text{delete}(v_{w,u}), \end{aligned}$$

where the operations related to u'' (or w'') are omitted if u'' (or w'') does not exist. We also remove w from the neighbor list of u , and remove u from the neighbor list of w .

B Contraction Technique

In this section, we summarize the construction $\text{Contract}_K(F)$ for a given unweighted forest F and a set of terminals K .

We start by defining the *connecting paths* for a tree $T = (V, E)$ and a set of terminals $K \subseteq V$. The set of connecting paths of T with respect to K is the minimal collection of edge disjoint paths in T that connect vertices in K . Formally, given T and K , the set of connecting paths, denoted as $\text{Path}_K(T)$, is a set of edge disjoint paths of T satisfying the following conditions:

1. The union of all the paths in $\text{Path}_K(T)$ is a connected subtree of T .
2. For any $v \in K$, v is an end point of some path in $\text{Path}_K(T)$.
3. For any endpoint v of a path in $\text{Path}_K(T)$, v is either a vertex in K or v is an endpoint for at least three paths in $\text{Path}_K(T)$.

It was shown in [14] that for fixed T and K , $\text{Path}_K(T)$ is uniquely defined, and has $O(K)$ paths.

The contracted tree with respect to T and K , denoted as $\text{Contract}_K(T)$, is constructed as follows:

1. The vertices of $\text{Contract}_K(T)$ are the endpoints of paths in $\text{Path}_K(T)$.

2. There is an edge (u, v) in $\text{Contract}_K(T)$ iff u and v are the two endpoints of a path in $\text{Path}_K(T)$.

Without loss of generality, we assume if $|K| = 1$, $\text{Contract}_K(T)$ contains only the isolated vertex of K , and if $K = \emptyset$, $\text{Contract}_K(F)$ is an empty graph.

The contracted tree can be extended to the contracted forest as follows: For an unweighted forest $F = (V, E)$ and a set of terminals $K \subseteq V$, $\text{Contract}_K(F)$ is the union of $\text{Contract}_{K \cap V(T)}(T)$ for each tree T in F .

LEMMA B.1. *For an unweighted forest $F = (V, E)$ and a set of terminals $K \subseteq V$, $\text{Contract}_K(F)$ is a forest that contains at most $O(|K|)$ vertices and edges.*

C Fully Dynamic Algorithm From Offline Update Algorithm

The following lemma, which was implicitly given in [24], shows that for a data structure of a graph, if there is an offline update algorithm (i.e., the update algorithm can update the data structure with respect to a given update sequence of the graph) such that the data structure can be maintained by applying the update algorithm for a bounded number of update sequences since initialized, then there is a fully dynamic algorithm to maintain the data structure.

LEMMA C.1. (SECTION 5, [24]) *Let G be a dynamic graph, and $\mathcal{D}$ be a data structure of G . Let $\zeta, w, t_{\text{initialization}}$, and t_{update} be four parameters. If there is an initialization algorithm for $\mathcal{D}$ with running time $t_{\text{initialization}}$ and an offline update algorithm satisfying the following two conditions:*

1. *the data structure $\mathcal{D}$ can be maintained after applying the offline update algorithm with any ζ different update sequences sequentially;*
2. *for an update sequence UpdateSeq for G such that $|\text{UpdateSeq}| \leq w$, the running time of the update algorithm is at most $t_{\text{update}} \cdot |\text{UpdateSeq}|$,*

then for any $\xi \leq \zeta$ satisfying $w \geq 2 \cdot 6^\xi$, there is a fully dynamic algorithm with initialization time $O(2^\xi \cdot t_{\text{initialization}})$ and worst-case update time $O(4^\xi \cdot (t_{\text{initialization}}/w + w^{(1/\xi)} t_{\text{update}}))$ to maintain a set of $O(2^\xi)$ instances of the data structure $\mathcal{D}$ such that after each update, the update algorithm specifies one of the maintained data structure instances satisfying the following conditions

1. *The specified data structure instance is for the up-to-date graph.*
2. *The online-batch update algorithm is executed for at most ξ times on the specified data structure instance with each update batch of size at most w .*