# Incremental Exact Min-Cut in Polylogarithmic Amortized Update Time

GRAMOZ GORANCI and MONIKA HENZINGER, University of Vienna, Austria
MIKKEL THORUP, University of Copenhagen

We present a deterministic incremental algorithm for *exactly* maintaining the size of a minimum cut with $O(\log^3 n \log\log^2 n)$ amortized time per edge insertion and $O(1)$ query time. This result partially answers an open question posed by Thorup (2007). It also stays in sharp contrast to a polynomial conditional lower bound for the fully dynamic weighted minimum cut problem. Our algorithm is obtained by combining a sparsification technique of Kawarabayashi and Thorup (2015) or its recent improvement by Henzinger, Rao, and Wang (2017), and an exact incremental algorithm of Henzinger (1997).

We also study space-efficient incremental algorithms for the minimum cut problem. Concretely, we show that there exists an $O(n \log n/\varepsilon^2)$ space Monte Carlo algorithm that can process a stream of edge insertions starting from an empty graph, and with high probability, the algorithm maintains a $(1+\varepsilon)$-approximation to the minimum cut. The algorithm has $O((\alpha(n)\log^3 n)/\varepsilon^2)$ amortized update time and constant query time, where $\alpha(n)$ stands for the inverse of Ackermann function.

CCS Concepts: • **Theory of computation → Dynamic graph algorithms**; *Streaming, sublinear and near linear time algorithms*;

Additional Key Words and Phrases: Minimum cut, edge connectivity, space-efficient dynamic graph algorithms

**ACM Reference format:**
Gramoz Goranci, Monika Henzinger, and Mikkel Thorup. 2018. Incremental Exact Min-Cut in Polylogarithmic Amortized Update Time. *ACM Trans. Algorithms* 14, 2, Article 17 (March 2018), 21 pages.
https://doi.org/10.1145/3174803

## 1 INTRODUCTION

Computing a minimum cut of a graph is a fundamental algorithmic graph problem. Although most of the focus has been on designing static-efficient algorithms for finding a minimum cut, the dynamic maintenance of a minimum cut has also attracted increasing attention over the past two

A preliminary version of this work appeared in *Proceedings of the 24th European Symposium on Algorithms (ESA'16)* [14]. The research leading to these results received funding from the European Research Council under the European Union's 7th Framework Programme (FP/2007-2013)/ERC Grant Agreement No. 340506 for M. Henzinger and G. Goranci. M. Thorup's research was partly supported by Advanced Grant DFF-0602-02499B from the Danish Council for Independent Research under the Sapere Aude research career programme. This work was done in part while M. Henzinger and M. Thorup were visiting the Simons Institute for the Theory of Computing.
Authors' addresses: G. Goranci and M. Henzinger, Faculty of Computer Science, University of Vienna, Währinger Straße 29, Vienna, 1090, Austria; emails: {gramoz.goranci, monika.henzinger}@univie.ac.at; M. Thorup, Department of Computer Science, University of Copenhagen, Universitetsparken 1, Copenhagen East, 2100, Denmark; email: mikkel2thorup@gmail.com.

https://doi.org/10.1145/3174803

decades. The motivation for studying the dynamic setting is apparent as real-life networks such as the social or road network undergo constant and rapid changes.

Given an initial graph $G$, the goal of a dynamic graph algorithm is to build a data structure that maintains $G$ and supports update and query operations. Depending on the types of update operations we allow, dynamic algorithms are classified into three main categories: (i) *fully dynamic*, if update operations consist of both edge insertions and deletions; (ii) *incremental*, if update operations consist of edge insertions only; and (iii) *decremental*, if update operations consist of edge deletions only. In this article, we study incremental algorithms for maintaining the size of a minimum cut of an unweighted, undirected graph (denoted by $\lambda(G) = \lambda$) supporting the following operations:

- Insert $(u, v)$: Insert the edge $(u, v)$ in $G$.
- QuerySize: Return the exact (approximate) size of a minimum cut of the current $G$.

For any $\alpha \geq 1$, we say that an algorithm is an $\alpha$-approximation of $\lambda$ if QuerySize returns a positive number $k$ such that $\lambda \leq k \leq \alpha \cdot \lambda$. Our problem is characterized by two time measures: *query time*, which denotes the time needed to answer each query, and *total update time*, which denotes the time needed to process all edge insertions. We say that an algorithm has an $O(t(n))$ amortized update time if it takes $O(m(t(n)))$ total update time for $m$ edge insertions starting from an empty graph. We use $\widetilde{O}(\cdot)$ to hide polylogarithmic factors.

*Related work.* For more than a decade, the best-known static and deterministic algorithm for computing a minimum cut was due to Gabow [11], which runs in $O(m + \lambda^2 n \log n)$ time. Kawarabayashi and Thorup [22] devised an $O(m \log^{12} n)$ time algorithm that applies only to unweighted, undirected simple graphs. Recently, Henzinger et al. [16] improved the running time to $O(m \log^2 n \log\log^2 n)$. Randomized Monte Carlo algorithms in the context of static minimum cut were initiated by Karger [20]. The best-known randomized algorithm is due to Karger [21] and runs in $O(m \log^3 n)$ time.

Karger [19] was the first to study the dynamic maintenance of a minimum cut in its full generality. He devised a fully dynamic, albeit randomized, algorithm for maintaining a $\sqrt{1 + 2/\varepsilon}$-approximation of the minimum cut in $\widetilde{O}(n^{1/2+\varepsilon})$ expected amortized time per edge operation. In the incremental setting, he showed that the update time for the same approximation ratio can be further improved to $\widetilde{O}(n^{\varepsilon})$. Thorup and Karger [33] improved upon the preceding guarantees by achieving an approximation factor of $\sqrt{2 + o(1)}$ and an $\widetilde{O}(1)$ expected amortized time per edge operation.

Henzinger [17] obtained the following guarantees for the incremental minimum cut: for any $\varepsilon \in (0, 1]$, (i) an $O(1/\varepsilon^2)$ amortized update time for a $(2 + \varepsilon)$-approximation, (ii) an $O(\log^3 n/\varepsilon^2)$ expected amortized update time for a $(1 + \varepsilon)$-approximation, and (iii) an $O(\lambda \log n)$ amortized update time for the exact minimum cut.

For minimum cut up to some polylogarithmic size, Thorup [32] gave a fully dynamic Monte Carlo algorithm for maintaining exact minimum cut in $\widetilde{O}(\sqrt{n})$ time per edge operation. He also showed how to obtain an $1 + o(1)$-approximation of an arbitrary-size minimum cut with the same time bounds. In comparison to previous results, it is worth pointing out that his work achieves worst-case update times.

Lacki and Sankowski [25] studied the dynamic maintenance of the exact size of the minimum cut in planar graphs with arbitrary edge weights. They obtained a fully dynamic algorithm with $\widetilde{O}(n^{5/6})$ worst-case query and update time.

There has been a growing interest in proving conditional lower bounds for dynamic problems in the past few years [1, 15]. A recent result of Nanongkai and Saranurak [29] shows the following

conditional lower bound for the *exact weighted* minimum cut assuming the online matrix-vector multiplication conjecture: for any $\varepsilon > 0$, there are no fully dynamic algorithms with polynomial-time preprocessing that can simultaneously achieve $O(n^{1-\varepsilon})$ update time and $O(n^{2-\varepsilon})$ query time.

*Our results and technical overview.* We present two new incremental algorithms concerning the maintenance of the size of a minimum cut. Both algorithms apply to undirected, unweighted simple graphs.

Our first and main result, presented in Section 4, shows that there is a deterministic incremental algorithm for exactly maintaining the size of a minimum cut with $O(\log^3 n \log\log^2 n)$ amortized time per operation and $O(1)$ query time. This result allows us to partially answer in the affirmative a question regarding efficient dynamic algorithms for exact minimum cut posed by Thorup [32]. Additionally, it also stays in sharp contrast to the polynomial conditional lower bound for the fully dynamic weighted minimum cut problem of Nanongkai and Saranurak [29].

We obtain our result by heavily relying on a recent sparsification technique developed in the context of static minimum cut algorithms. Specifically, for a given simple graph $G$, Kawarabayashi and Thorup [22] (and subsequently Henzinger et al. [16]) designed an $\widetilde{O}(m)$ procedure that contracts vertex sets of $G$ and produces a multigraph $H$ with considerably fewer vertices and edges while preserving some family of cuts of size up to $(3/2)\lambda(G)$. Motivated by the properties of $H$, the crucial observation is that it is "safe" to work entirely with graph $H$ as long as the sequence of newly inserted edges do not increase the size of the minimum cut in $H$ by more than $(3/2)\lambda(G)$. If the latter occurs, we recompute a new multigraph $H$ for the current graph $G$. Since $\lambda(G) \leq n$, the number of such recomputations is $O(\log n)$. For maintaining the minimum-cut of $H$, we appeal to the exact incremental algorithm due to Henzinger [17]. Our main technical contribution is to skillfully combine these two algorithms and formally argue that such combination leads to our desirable guarantees.

Motivated by the recent work on space-efficient dynamic algorithms [5, 13], we also study the efficient maintenance of the size of a minimum cut using only $\widetilde{O}(n)$ space. Concretely, we present a $O(n \log n/\varepsilon^2)$ space Monte Carlo algorithm that can process a stream of edge insertions starting from an empty graph, and with high probability, it maintains a $(1+\varepsilon)$-approximation to the minimum cut in $O((\alpha(n)\log^3 n)/\varepsilon^2)$ amortized update time and constant query time.

Note that although the streaming model also allows only $\widetilde{O}(n)$ space, it is less constrained than the space-efficient dynamic model since streaming algorithms do not need to maintain an explicit sparsifier at every moment but just have enough information to construct one at the end of the stream. There have been several streaming algorithms [2, 3, 23, 24] for maintaining a cut sparsifier, and thus $(1+\varepsilon)$-approximating the minimum cut. The best bounds are due to Kyng et al. [24], who compute a stronger spectral sparsifier with $O(n \log n/\varepsilon^2)$ size and $O(\log^2 n)$ amortized update time. In comparison to our result, although our update time is slightly worse, we can achieve constant query time, whereas their algorithms require $\Omega(n)$ time to answer a query.

## 2 PRELIMINARY

Let $G = (V, E)$ be an undirected, unweighted multigraph with no self-loops. Two vertices $x$ and $y$ are *$k$-edge connected* if there exist $k$ edge-disjoint paths connecting $x$ and $y$. A graph $G$ is *$k$-edge connected* if every pair of vertices is $k$-edge connected. The *local edge connectivity* $\lambda(x, y, G)$ of vertices $x$ and $y$ is the largest $k$ such that $x$ and $y$ are $k$-edge connected in $G$. The *edge connectivity* $\lambda(G)$ of $G$ is the largest $k$ such that $G$ is $k$-edge connected.

For a subset $S \subseteq V$ in $G$, the *edge cut* $E_G(S, V \setminus S)$ is a set of edges that have one endpoint in $S$ and the other in $V \setminus S$. We may omit the subscript when clear from the context. Let $\lambda(S, G) = |E_G(S, V \setminus S)|$ be the size of the edge cut. If $S$ is a singleton, we refer to such cut as a *trivial* cut.

Two vertices $x$ and $y$ are *separated* by $E(S, V \setminus S)$ if they belong to different connected components of the graph induced by $E \setminus E(S, V \setminus S)$. A *minimum edge cut* of $x$ and $y$ is a cut of minimum size among all cuts separating $x$ and $y$. A *global minimum cut* $\lambda(G)$ for $G$ (or simply $\lambda$ when $G$ is clear from the context) is the minimum edge cut over all pairs of vertices. By Menger's theorem [26], (a) the size of the minimum edge cut separating $x$ and $y$ is $\lambda(x, y, G)$, and (b) the size of the global minimum cut is equal to $\lambda(G)$.

Let $n$, $m_0$, and $m_1$ be the number of vertices, initial edges, and inserted edges, respectively. The total number of edges $m$ is the sum of the initial and inserted edges. Moreover, let $\lambda$ and $\delta$ denote the size of the global minimum cut and the minimum degree in the final graph, respectively. Note that the minimum degree is always an upper bound on the edge connectivity (i.e., $\lambda \leq \delta$ and $m = m_0 + m_1 = \Omega(\delta n)$).

A subset $U \subseteq V$ is *contracted* if all vertices in $U$ are identified with some element of $U$ and all edges between them are discarded. For $G = (V, E)$ and a collection of vertex sets, let $H = (V(H), E(H))$ denote the graph obtained by contracting such vertex sets. Such contractions are associated with a mapping $h : V \to V(H)$. For an edge subset $N \subseteq E$, let $N_h = \{(h(a), h(b)) : (a, b) \in N\} \subseteq E(H)$ be its corresponding edge subset induced by $h$. Throughout, we will use the term with high probability (in short, w.h.p.) to denote the event that holds with probability at least $1 - 1/n^c$, for some positive constant $c$.

## 3 SPARSE CERTIFICATES

In this section, we review a useful sparsification tool introduced by Nagamochi and Ibaraki [27]. We first give the following definition from Benczúr and Karger [4], which also appeared implicitly in Nagamochi and Ibaraki [27].

*Definition 3.1.* A *sparse $k$-connectivity certificate*, or simply a *$k$-certificate*, for an unweighted graph $G$ with $n$ vertices is a subgraph $G'$ of $G$ such that

(1) $G'$ consists of at most $k(n-1)$ edges, and
(2) $G'$ contains all edges crossing cuts of size at most $k$.

Given an undirected graph $G = (V, E)$, a *(maximal) spanning forest decomposition (msfd)* $\mathcal{F}$ of order $k$ is a decomposition of $G$ into $k$ edge-disjoint spanning forests $F_i$, $1 \leq i \leq k$, such that $F_i$ is a (maximal) spanning forest of $G \setminus (F_1 \cup F_2 \ldots \cup F_{i-1})$. Note that $G_k = (V, \bigcup_{i \leq k} F_i)$ is a $k$-certificate. An msfd fulfills the following property, whose proof we defer to the appendix.

Lemma 3.2 ([28]). *Let $\mathcal{F} = (F_1, \ldots, F_m)$ be an* msfd *of order $m$ of a graph $G = (V, E)$, and let $k$ be an integer with $1 \leq k \leq m$. Then for any nonempty and proper subset $S \subset V$,*

$$\lambda(S, G_k) \begin{cases} \geq k, & \text{if } \lambda(S, G) \geq k \\ = \lambda(S, G) & \text{if } \lambda(S, G) \leq k - 1. \end{cases}$$

As $G_k$ is a subgraph of $G$, $\lambda(G_k) \leq \lambda(G)$. This implies that $\lambda(G_k) \geq \min(k, \lambda(G))$.

Nagamochi and Ibaraki [27] presented an $O(m + n)$ time algorithm (which we call a *decomposition algorithm* (DA)) to construct a special msfd, which we refer to as DA-msfd.

## 4 INCREMENTAL EXACT MINIMUM CUT

In this section, we present a deterministic incremental algorithm that exactly maintains $\lambda(G)$. The algorithm has $O(\log^3 n \log\log^2 n)$ update time and $O(1)$ query time, and it applies to any undirected, unweighted simple graph $G = (V, E)$. The result is obtained by carefully combining a recent static min-cut algorithm by Kawarabayashi and Thorup [22] or its recent improvement due to Henzinger et al. [16], and the incremental min-cut algorithm of Henzinger [17]. We start

by describing the maintenance of nontrivial cuts—that is, cuts with at least two vertices on both sides.

*Maintaining nontrivial cuts.* Kawarabayashi and Thorup [22] devised a near-linear time algorithm that contracts vertex sets of a simple input graph $G$ and produces a sparse multi-graph $H$ preserving all nontrivial minimum cuts of $G$. We refer to such a graph $H$ as a KT-Sparsifier. Recently, Henzinger et al. [16] improved the running time for constructing $H$ and provided better bounds on the size of $H$. We next define a slightly generalized version of a KT-Sparsifier and then state the bounds achieved by these two algorithms.

*Definition 4.1 (KT-Sparsifier).* Let $G = (V, E)$ be an undirected, unweighted simple graph with $n$ vertices, $m$ edges, and min-cut $\lambda$. A multigraph $H = (V(H), E(H))$ is a KT-Sparsifier of $G$ if the following holds:

- $H$ has $n_H = \widetilde{O}(n/\lambda)$ vertices and $m_H = \widetilde{O}(m/\lambda)$ edges
- $H$ preserves all nontrivial cuts of size up to $(3/2)\lambda$ in $G$
- $H$ is obtained by contracting vertex sets in $G$.

Theorem 4.2 ([22]). *Given an undirected, unweighted simple graph $G = (V, E)$, there is an $O(m \log^{12} n)$ time algorithm to construct a KT-Sparsifier $H$ of $G$ such that $H$ has $O(n \log^4 n/\lambda)$ vertices and $O(m \log^4 n/\lambda)$ edges.*

In what follows, whenever we invoke the algorithm that constructs a KT-Sparsifier, we mean to invoke the algorithm from the following theorem.

Theorem 4.3 ([16]). *Given an undirected, unweighted simple graph $G = (V, E)$, there is an $O(m \log^2 n \log\log^2 n)$ time algorithm to construct a KT-Sparsifier $H$ of $G$ such that $H$ has $O(n \log n/\lambda)$ vertices and $O(m \log n/\lambda)$ edges.*

As far as nontrivial cuts are concerned, Theorem 4.3 implies that it is safe to work on $H$ instead of $G$ as long as the sequence of newly inserted edges satisfies $\lambda_H \leq (3/2)\lambda$. To incrementally maintain the correct $\lambda_H$, we apply Henzinger's algorithm [17] on top of $H$. The basic idea to verify the correctness of the solution is to compute and store all min-cuts of $H$. Clearly, a solution is correct as long as an edge insertion does not increase the size of all min-cuts. If all min-cuts have increased, a new solution is computed using information about the previous solution. The preceding steps can be performed efficiently by making use of the cactus tree representation, which we will define shortly. The crucial observation is that whenever $\lambda_H$ increases (and assuming that we can efficiently check this), instead of recomputing the cactus tree from scratch, we update intermediate structures that remained from the previous cactus tree. We next show a precise implementation of these steps.

The minimum edge cuts are stored using the *cactus tree* representation introduced by Dinitz et al. [7] (see Fleiner and Frank [9] for a concise proof). A cactus tree of a graph $G = (V, E)$ is a weighted graph $G_c = (V_c, E_c)$ defined as follows. There is a mapping $\phi : V \to V_c$ such that

(1) Every node in $V$ maps to exactly one node in $V_c$ and every node in $V_c$ corresponds to a (possibly empty) subset of $V$.
(2) $\phi(x) = \phi(y)$ if and only if $x$ and $y$ are $(\lambda(G) + 1)$-edge connected.
(3) Every min-cut in $G_c$ corresponds to a min-cut in $G$, and every min-cut in $G$ corresponds to at least one min-cut in $G_c$.
(4) If $\lambda$ is odd, every edge of $E_c$ has weight $\lambda$ and $G_c$ is a tree. If $\lambda$ is even, no two simple cycles of $G_c$ intersect in more than one node. Furthermore, edges that belong to a cycle have weight $\lambda/2$, whereas those not belonging to a cycle have weight $\lambda$.

Dinitz and Westbrook [8] showed that given a cactus tree, we can use the data structures from Galil and Italiano [12] and La Poutré [30] to efficiently maintain the cactus tree for fixed minimum cut size $\lambda$ under edge insertions. This implies that this data structure can be used to efficiently test whether min-cut has increased its value during edge insertions. The result is summarized in the following theorem.

THEOREM 4.4 ([8]). *Given a cactus tree, there is an algorithm that maintains the cactus tree for fixed minimum cut size $\lambda$ under $u$ edge insertions, reporting when the minimum cut size increase to $\lambda + 1$ in $O(u + n)$ total time.*

We now turn our attention to the efficient construction and update of the cactus tree representation of a given multigraph $G$. To construct the cactus tree, we use an algorithm due to Gabow [10], which proceeds as follows. It first computes a subgraph of $G$, called a *complete $\lambda$-intersection* or $I(G, \lambda)$, with at most $\lambda n$ edges, and then uses $I(G, \lambda)$ to compute the cactus tree. In the following theorem, we state the running time for the cactus tree construction dependent on the time for computing $I(G, \lambda)$.

THEOREM 4.5 ([10]). *Let $G = (V, E)$ be an undirected, unweighted multigraph, and assume that there is an algorithm that computes $I(G, \lambda)$ in $O(T(m, n))$ time. Given $I(G, \lambda)$, the cactus tree representation of $G$ can be constructed in $O(m)$ time. Hence, the total time for constructing the cactus tree of $G$ is bounded by $O(T(m, n) + m)$.*

Gabow [11] devised an algorithm to compute $I(G, \lambda)$ in $O(m + \lambda^2 n \log n)$ time. Moreover, his algorithm is incremental in the sense that whenever $I(G, \lambda)$ is given as an input, the new $I(G, \lambda + 1)$ can be computed more efficiently rather than just recomputing it from scratch. The precise statement and bounds are given in the following theorem.

THEOREM 4.6 ([11]). *Given an undirected, unweighted multigraph $G = (V, E)$, there is an algorithm that computes $I(G, \lambda)$ in $O(m + \lambda^2 n \log n)$ time. Moreover, given $I(G, \lambda)$ and a sequence of edge insertions that increase the minimum cut by 1, the new $I(G, \lambda + 1)$ can be computed in $O(m' \log n)$ time, where $m'$ is the number of edges in the current graph.*

Note that by combining Theorems 4.5 and 4.6, we get that the cactus tree for the initial graph can be computed in $O(m_0 + \lambda^2 n \log n)$ time, and the new cactus tree for some current graph whose minimum cut has increased can be computed in $O(m' \log n)$ time.

*Maintaining trivial cuts.* We remark that the multigraph $H$ from Theorem 4.3 preserves only nontrivial cuts of $G$. If $\lambda = \delta$, then we also need a way to keep track of a trivial minimum cut. We achieve this by maintaining a minimum heap $\mathcal{H}_G$ on the vertices, where each vertex is stored with its degree. When an edge insertion is performed, the values of the edge endpoints are updated accordingly in the heap. It is well known that constructing $\mathcal{H}_G$ takes $O(n)$ time. The supported operations MIN($\mathcal{H}_G$) and UPDATEENDPOINTS($\mathcal{H}_G, e$) can be implemented in $O(1)$ and $O(\log n)$ time, respectively (see Cormen et al. [6]).

This leads to Algorithm 1.

*Correctness.* Let $G$ be the current graph throughout the execution of the algorithm, and let $H$ be the corresponding multigraph maintained by the algorithm. Recall that $H$ preserves some family of cuts from $G$. We say that $H$ is *useful* if and only if there exists a minimum cut from $G$ that is contained in the union of (a) all trivial cuts of $G$ and (b) all cuts in $H$. Note that we consider $H$ to be useful even in the `Special Step` (i.e., when $\lambda_H > (3/2)\lambda^*$), where $H$ is not updated anymore since we are certain that the smallest trivial cut is smaller than any cut in $H$.

To prove the correctness of the algorithm, we will show that (1) it correctly maintains a trivial min-cut at any time, (2) as long as $\lambda_H \leq (3/2)\lambda^*$, the algorithm correctly maintains all cuts of size

**ALGORITHM 1:** Incremental Exact Minimum Cut

1: Compute the size $\lambda_0$ of the min-cut of $G$ and set $\lambda^* = \lambda_0$.
Build a heap $\mathcal{H}_G$ on the vertices, where each vertex stores its degree as a key.
Compute a KT-sparsifier $H$ of $G$ and a mapping $h : V \to V_H$.
Compute the size $\lambda_H$ of the min-cut of $H$, a DA-msfd $F_1, \ldots, F_m$ of order $m$ of $H$,
$I(H, \lambda_H)$, and a cactus-tree of $\bigcup_{i \leq \lambda_H + 1} F_i$.
2: Set $N_h = \emptyset$.
`// Use the data-structure from Theorem 4.4 to maintain the cactus tree`
**while** there is at least one minimum cut of size $\lambda_H$ **do**
**Receive the next operation**.
**if** it is a query **then return** $\min\{\lambda_H, \textsc{Min}(\mathcal{H}_G)\}$
**else** it is the insertion of an edge $(u, v)$, **then**
update the cactus tree according to the insertion of the new edge $(h(u), h(v))$,
add the edge $(h(u), h(v))$ to $N_h$ and update the degrees of $u$ and $v$ in $\mathcal{H}_G$.
**endif**
**endwhile**
Set $\lambda_H = \lambda_H + 1$.
3: **if** $\min\{\lambda_H, \textsc{Min}(\mathcal{H}_G)\} > (3/2)\lambda^*$ **then**
`// Full Rebuild Step`
Compute $\lambda(G)$ and set $\lambda^* = \lambda(G)$.
Compute a KT-sparsifier $H$ of the current graph $G$.
Update $\lambda_H$ to be the min-cut of $H$, compute a DA-msfd $F_1, \ldots, F_m$ of order $m$ of $H$,
and then $I(H, \lambda_H)$ and a cactus tree of $\bigcup_{i \leq \lambda_H + 1} F_i$.
**else if** $\lambda_H \leq (3/2)\lambda^*$ **then**
// `Partial Rebuild Step`
Compute a DA-msfd $F_1, \ldots, F_m$ of order $m$ of $\bigcup_{i \leq (3/2)\lambda^* + 1} F_i \cup N_h$ and
call the resulting forests $F_1, \ldots, F_m$.
// `Update the cactus tree using Theorems 4.5 and 4.6`
Let $H' = (V(H), E')$ be a graph with $E' = I(H, \lambda_H - 1) \cup \bigcup_{i \leq \lambda_H + 1} F_i$.
Compute $I(H', \lambda_H)$, a cactus tree of $H'$ and set $H = H'$.
**else** // `Special Step`
**while** $\textsc{Min}(\mathcal{H}_G) \leq (3/2)\lambda^*$ **do**
**if** the next operation is a query **then return** $\textsc{Min}(\mathcal{H}_G)$
**else** update the degrees of the edge endpoints in $\mathcal{H}_G$.
**endif**
**endwhile**
**Goto** 3.
**endif**
**Goto** 2.

up to $(3/2)\lambda^* + 1$ of $H$, and (3) $H$ is useful as long as $\min\{\textsc{Min}(\mathcal{H}_G), \lambda_H\} \leq (3/2)\lambda^*$ (note that when this condition fails, we rebuild $H$).

Lemma 4.7. *The algorithm correctly maintains a trivial min-cut in $G$.*

Proof. This follows directly from the min-heap property of $\mathcal{H}_G$. □

To simplify the notation, in the following we will refer to Step 1 as a `Full Rebuild Step` (namely the initial `Full Rebuild Step`). Let $G = (V, E)$ be the current graph, and let $H$ be the multigraph obtained by invoking KT-sparsifier on $G$, at the time of a `Full Rebuild Step`. Now, as long as $\lambda_H \leq (3/2)\lambda^*$, suppose that the graph $G$ and its corresponding multigraph $H$ have

undergone a sequence of edge insertions that triggered $k$ executions of `Partial Rebuild Steps` (including Step 2) for some $k \geq 0$. Note that no `Full Rebuild Step` is executed as long as $\lambda_H \leq (3/2)\lambda^*$.

Let $H^{(k)} = (V(H), E(H^{(k)}))$ be the multigraph $H$ *after* the $k$-th partial rebuild, and let $H^{(0)} = H$. Let $N_h^{(k)} \subseteq E(H^{(k)})$ be the set of inserted edges in $H$ that the algorithm maintains during the execution of the *while* loop in Step 2, after the $(k-1)$-st and before the $k$-th partial rebuild. Define $\tilde{H}^{(k)} = (V(H), \bigcup_{i \leq (3/2)\lambda^*+1} F_i^{(k)}) \cup N_h^{(k)})$ to be the sparsified graph that the algorithm maintains, where $F_1^{(k)}, \ldots, F_m^{(k)}$ is a DA-msfd for the graph $\tilde{H}^{(k-1)}$, and let $\tilde{H}^{(0)} = H$ be the multigraph right after the last full rebuild. We next show that $\tilde{H}^{(k)}$ preserves all cuts of size up to $(3/2)\lambda^* + 1$ of $H^{(k)}$.

Lemma 4.8. *For $k \geq 0$, let $H^{(k)}$ and $\tilde{H}^{(k)}$ be the multigraphs defined earlier. Then for any nonempty and proper subset $S \subset V(H)$,*

$$\lambda(S, \tilde{H}^{(k)}) \begin{cases} \geq (3/2)\lambda^* + 1, & \text{if } \lambda(S, H^{(k)}) \geq (3/2)\lambda^* + 1, \\ = \lambda(S, H^{(k)}) & \text{if } \lambda(S, H^{(k)}) \leq (3/2)\lambda^*. \end{cases}$$

Proof. We proceed by induction on the number $k$ of partial rebuilds. We give the inductive step; the base case ($k = 0$) follows from the fact that $\tilde{H}^{(0)} = H = H^{(0)}$.

Fix any cut $(S, V(H) \setminus S)$ in $H^{(k)}$, and note that $H^{(k)} = (V(H), E(H^{(k-1)}) \cup N_h^{(k)})$. Define $A := E_{H^{(k)}}(S, V(H) \setminus S) \cap N_h^{(k)}$ and $B := E_{H^{(k)}}(S, V(H) \setminus S) \cap E(H^{(k-1)})$ such that $E_{H^{(k)}}(S, V(H) \setminus S) = A \uplus B$. Letting $F' = \bigcup_{i \leq (3/2)\lambda^*+1} F_i^{(k)}$, we similarly define edge sets $\tilde{A}$ and $\tilde{B}$ partitioning the edges $E_{\tilde{H}^{(k)}}(S, V(H) \setminus S)$ that cross the cut $(S, V(H) \setminus S)$ in $\tilde{H}^{(k)}$. Note that $A = \tilde{A}$ since edges of $N_h^{(k)}$ are always included in $\tilde{H}^{(k)}$ and $\lambda(S, H^{(k)}) = |A| + |B|$, $\lambda(S, \tilde{H}^{(k)}) = |\tilde{A}| + |\tilde{B}|$. We distinguish two cases.

First, assume that $\lambda(S, H^{(k)}) \leq (3/2)\lambda^*$. Then, since $H^{(k-1)} \subseteq H^{(k)}$ and by construction of $H^{(k)}$, $\lambda(S, H^{(k-1)}) = |B|$, we get that $\lambda(S, H^{(k-1)}) \leq (3/2)\lambda^*$. By the induction hypothesis, it follows that $\lambda(S, \tilde{H}^{(k-1)}) = \lambda(S, H^{(k-1)}) \leq (3/2)\lambda^*$. The latter along with Lemma 3.2 implies that $|\tilde{B}| = \lambda(S, \tilde{H}^{(k-1)})$, and thus $\lambda(S, \tilde{H}^{(k)}) = |\tilde{A}| + |\tilde{B}| = |A| + |B| = \lambda(S, H^{(k)})$.

Second, assume that $\lambda(S, H^{(k)}) \geq (3/2)\lambda^* + 1$. Then either $\lambda(S, H^{(k-1)}) \leq (3/2)\lambda^*$ or $\lambda(S, H^{(k-1)}) \geq (3/2)\lambda^* + 1$. In the first case, by the induction hypothesis, it follows that $\lambda(S, \tilde{H}^{(k-1)}) = \lambda(S, H^{(k-1)}) \leq (3/2)\lambda^*$. This along with Lemma 3.2 implies that $|\tilde{B}| = \lambda(S, \tilde{H}^{(k-1)})$, and thus $\lambda(S, \tilde{H}^{(k)}) = |\tilde{A}| + |\tilde{B}| = |A| + |B| = \lambda(S, H^{(k)}) \geq (3/2)\lambda^* + 1$. In the second case, by the induction hypothesis, it follows that $\lambda(S, \tilde{H}^{(k-1)}) \geq (3/2)\lambda^* + 1$. The latter along with Lemma 3.2 imply that $|\tilde{B}| \geq (3/2)\lambda^* + 1$, and thus $\lambda(S, \tilde{H}^{(k)}) = |\tilde{A}| + |\tilde{B}| \geq (3/2)\lambda^* + 1$, which completes the proof. □

We now show that the multigraphs $H^{(k)}$ and $\tilde{H}^{(k)}$ share the same set of minimum cuts.

Lemma 4.9. *Assume that $\lambda(H^{(k)}) \leq (3/2)\lambda^*$. Then a cut is a min-cut in $H^{(k)}$ if and only if it is a min cut in $\tilde{H}^{(k)}$.*

Proof. We first show that every non–min cut in $H^{(k)}$ is a non–min cut in $\tilde{H}^{(k)}$. By contrapositive, we get that a min-cut in $\tilde{H}^{(k)}$ is a min-cut in $H^{(k)}$.

To this end, let $(S, V(H) \setminus S)$ be a cut with $\lambda(S, H^{(k)}) \geq \lambda(H^{(k)}) + 1$ in $H^{(k)}$. Note that by assumption $\lambda(H^{(k)}) \leq (3/2)\lambda^*$. By Lemma 4.8, we distinguish two cases. First, if $\lambda(S, H^{(k)}) \leq (3/2)\lambda^*$, then $\lambda(S, \tilde{H}^{(k)}) = \lambda(S, H^{(k)}) \geq \lambda(H^{(k)}) + 1$. Second, if $\lambda(S, H^{(k-1)}) \geq (3/2)\lambda^* + 1$, then $\lambda(S, \tilde{H}^{(k)}) \geq (3/2)\lambda^* + 1 \geq \lambda(H^{(k)}) + 1$. The preceding cases along with $\lambda(H^k) \geq \lambda(\tilde{H}^{(k)})$ give that $\lambda(S, \tilde{H}^{(k)}) \geq \lambda(\tilde{H}^{(k)}) + 1$, which in turn implies that $(S, V(H) \setminus S)$ cannot be a min-cut in $\tilde{H}^{(k)}$.

For the other direction, consider a min-cut $(D, V(H) \setminus D)$ of size $\lambda(D, \tilde{H}^{(k)})$ in $\tilde{H}^{(k)}$. Considering the cut in $H^{(k)}$, we know that $\lambda(D, H^{(k)}) \geq \lambda(H^{(k)})$. Then, similarly as before, Lemma 4.8 implies

that $\lambda(D, \tilde{H}^{(k)}) \geq \lambda(H^{(k)})$. Since $(D, V(H) \setminus D)$ was chosen arbitrarily, we get that $\lambda(\tilde{H}^{(k)}) \geq \lambda(H^{(k)})$ must hold. The latter along with $\lambda(\tilde{H}^{(k)}) \leq \lambda(H^{(k)})$ imply that $\lambda(\tilde{H}^{(k)}) = \lambda(H^{(k)})$.

Now, let $(S, V(H) \setminus S)$ be a min-cut in $H^{(k)}$. Since $\tilde{H}^{(k)}$ is a subgraph of $H^{(k)}$, we know that $\lambda(S, \tilde{H}^{(k)}) \leq \lambda(S, H^{(k)})$. The latter along with $\lambda(\tilde{H}^{(k)}) = \lambda(H^{(k)})$ imply that

$$\lambda(S, \tilde{H}^{(k)}) \leq \lambda(S, H^{(k)}) = \lambda(H^{(k)}) = \lambda(\tilde{H}^{(k)}),$$

or $\lambda(S, \tilde{H}^{(k)}) \leq \lambda(\tilde{H}^{(k)})$. It follows that the inequality must hold with equality since $\lambda(\tilde{H}^{(k)})$ is the value of min-cut in $\tilde{H}^{(k)}$. Thus, $(S, V(H) \setminus S)$ is also a min-cut in $\tilde{H}^{(k)}$. □

Lemma 4.10. *For some current graph $G$, let $H$ be the current multigraph maintained by the algorithm and assume that $\lambda_H \leq (3/2)\lambda^*$, where $\lambda^*$ denotes the min-cut of $G$ at the last* `Full Rebuild Step`. *Then the value $\lambda_H$ maintained by the algorithm satisfies $\lambda_H = \lambda(H)$.*

Proof. Let $\lambda(H^{(k)})$ be the value of $\lambda_H$ *after* the $k$-th execution of partial rebuild step for $k \geq 0$. Since $\lambda(H^{(k)}) = \lambda(H)$, it suffices to show that $\lambda(H^{(k)})$ is correct. We proceed by induction on the number $k$ of partial rebuilds since the last full rebuild.

We first consider the base case $k = 0$ (i.e., the time right after the last full rebuild). At the beginning of a full rebuild, the algorithm computes a KT-sparsifier $H$ of $G$ that preserves all nontrivial min-cuts of $G$. The value of $\lambda_H$ is updated to $\lambda(H)$, a DA-msfd $F_1, \ldots, F_m$ is computed for $H$, and a cactus tree is constructed for $F' = \bigcup_{i \leq \lambda_H + 1} F_i$. Lemma 3.2 shows that a cut is a min-cut in $H$ if and only if it is a min-cut in $F'$. The latter implies that since the cactus tree preserves the min-cuts of $F'$, it also preserves those of $H$. The fact that the cactus tree algorithm correctly tells us when to increment $\lambda_H$ in Step 2, we conclude that the value of $\lambda_H$ after a full rebuild is set correctly.

We next give the inductive step. By the induction hypothesis, assume that $\lambda(H^{(k-1)})$ is correct. By Lemma 4.9, we get that a cut is a min-cut in $H^{(k-1)}$ if and only if it is a min-cut in $\tilde{H}^{(k-1)}$. Now, let $F_1^{(k)}, \ldots, F_m^{(k)}$ be the DA-msfd computed on $\tilde{H}^{(k-1)}$ during the $k$-th partial rebuild, and define $\tilde{F}^{(k)} = \bigcup_{i \leq \lambda(H^{(k-1)})+1} F_i^{(k)}$. Lemma 3.2 shows that a cut is min-cut in $\tilde{H}^{(k-1)}$ if and only if it is a min-cut in $\tilde{F}^{(k)}$. The two preceding equivalences give that every min-cut in $H^{(k-1)}$ is a min-cut $\tilde{F}^{(k)}$, and thus the graph $H'^{(k)}$ (as defined in Algorithm 1) correctly preserves all min-cuts of $H^{(k-1)}$. Given the correctness of $\lambda(H^{(k-1)})$, the properties of the cactus trees, and the fact that the incremental cactus tree algorithm correctly tells us when to increment $\lambda(H^{(k-1)})$ in Step 2, we conclude that $\lambda(H^{(k)})$ is the correct min-cut value for the graph $H^{(k)} = (V(H), E(H^{(k-1)}) \cup N_h^{(k)})$ after the $k$-th partial rebuild. □

Note that when $\lambda_H > (3/2)\lambda^*$, the preceding lemma is not guaranteed to hold, as the algorithm does not execute a `Partial Rebuild Step` in this case. However, we will show in the following that this is not necessary for the correctness of the algorithm. The fact that we do not need to execute a `Partial Rebuild Step` in this setting is crucial for achieving our time bound.

Lemma 4.11. *If $\min\{\textsc{Min}(\mathcal{H}_G), \lambda_H\} \leq 3/2\lambda^*$, then $H$ is useful.*

Proof. Let $(S', V \setminus S')$ be any nontrivial cut in $G$ that is not in $H$. Such a cut must have cardinality strictly greater than $(3/2)\lambda^*$, as otherwise it would be contained in $H$. We show that $(S', V \setminus S')$ cannot be a minimum cut as long as $\min\{\textsc{Min}(\mathcal{H}_G), \lambda_H\} \leq (3/2)\lambda^*$ holds. We distinguish two cases:

(1) If $\lambda_H \leq (3/2)\lambda^*$, then by Lemma 4.10 the algorithm maintains $\lambda_H$ correctly. Since $H$ is obtained from $G$ by contracting vertex sets, there is a cut $(S, V_H, S)$ in $H$, and thus in $G$, of value $\lambda_H$. It follows that $(S', V \setminus S')$ cannot be a minimum cut of $G$ since $|E(S', V \setminus S')| > (3/2)\lambda^* \geq \lambda_H = \lambda(H) \geq \lambda(G)$, where the last inequality follows from the fact that $H$ is a contraction of $G$.

(2) If $\textsc{Min}(\mathcal{H}_G) \leq (3/2)\lambda^*$, then by Lemma 4.7 there is a cut of size $\textsc{Min}(\mathcal{H}_G) = \delta$ in $G$. Similarly, $(S', V \setminus S')$ cannot be a minimum cut of $G$ since $|E(S', V \setminus S')| > (3/2)\lambda^* \geq \delta \geq \lambda(G)$.

Appealing to the preceding cases, we conclude that $H$ is useful since a min-cut of $G$ is either contained in $H$ or is a trivial cut of $G$.

Lemma 4.12. *Let $G$ be some current graph. Then the algorithm correctly maintains $\lambda(G)$.*

Proof. Let $G$ be some current graph and $H$ be the current multigraph maintained by the algorithm. We will argue that $\lambda(G) = \min\{\textsc{Min}(\mathcal{H}_G), \lambda_H\}$.

If $\min\{\textsc{Min}(\mathcal{H}_G), \lambda_H\} \leq (3/2)\lambda^*$, then by Lemma 4.11, $H$ is useful—that is, there exists a minimum cut of $G$ that is contained in the union of all trivial cuts of $G$ and all cuts in $H$. Lemma 4.7 guarantees that the algorithm correctly maintains $\textsc{Min}(\mathcal{H}_G)$ (i.e., the trivial minimum cut of $G$). If $\lambda_H \leq (3/2)\lambda^*$, then Lemma 4.10 ensures that $\lambda_H = \lambda(H)$, and thus $\min\{\textsc{Min}(\mathcal{H}_G), \lambda_H\} = \lambda(G)$. If, however, $\lambda_H > (3/2)\lambda^*$ but $\min\{\textsc{Min}(\mathcal{H}_G), \lambda_H\} \leq (3/2)\lambda^*$, then $\lambda_H > \min\{\textsc{Min}(\mathcal{H}_G), \lambda_H\}$, which implies that $\min\{\textsc{Min}(\mathcal{H}_G), \lambda_H\} = \textsc{Min}(\mathcal{H}_G) = \lambda(G)$. As we argued earlier, the algorithm correctly maintains $\textsc{Min}(\mathcal{H}_G)$ at any time. Thus, it follows that the algorithm correctly maintains $\lambda(G)$ in this case as well.

The only case that remains to consider is $\min\{\textsc{Min}(\mathcal{H}_G), \lambda_H\} > (3/2)\lambda^*$. But whenever this happens, the algorithm performs a full rebuild step. After this full rebuild, $\lambda(G) = \min\{\textsc{Min}(\mathcal{H}_G), \lambda_H\}$ trivially holds.

*Running time analysis.*

Theorem 4.13. *Let $G$ be a simple graph with $n$ nodes and $m_0$ edges. Then the total time for inserting $m_1$ edges and maintaining a minimum edge cut of $G$ is*

$$O((m_0 + m_1)\log^3 n \log\log^2 n).$$

*If we start with an empty graph, the amortized time per edge insertion is $O(\log^3 n \log\log^2 n)$. The size of a minimum cut can be answered in constant time.*

Proof. We first analyze Step 1. Note that building the heap $\mathcal{H}_G$ and computing $\lambda_0$ take $O(n)$ and $O(m_0 \log^2 n \log\log^2 n)$ time, respectively. Recall that $m_0 \geq \lambda_0 n$. The total running time for constructing $H$, $I(H, \lambda_H)$ and the cactus tree is dominated by $O((m_0 + \lambda_0^2 \cdot (n/\lambda_0))\log^2 n \log\log^2 n) = O(m_0 \log^2 n \log\log^2 n)$. Thus, the total time for Step 1 is $(m_0 \log^2 n \log\log^2 n)$.

Let $\lambda_H^0, \ldots, \lambda_H^f$ be the values that $\lambda_H$ assumes in Step 2 during the execution of the algorithm in increasing order. We define Phase $i$ to be all steps executed after Step 1 while $\lambda_H = \lambda_H^i$, excluding `Full Rebuild Steps` and `Special Steps`. Additionally, let $\lambda_0^*, \ldots, \lambda_{O(\log n)}^*$ be the values that $\lambda^*$ assumes during the algorithm. We define *Superphase $j$* to consist of the $j$-th `Full Rebuild Step` along with all steps executed until the next `Full Rebuild Step`, i.e., while $\min\{\textsc{Min}(\mathcal{H}_G), \lambda_H\} \leq (3/2)\lambda_j^*$, where $\lambda_j^*$ is the value of $\lambda(G)$ at the $j$-th `Full Rebuild Step`. Note that a superphase consists of a sequence of phases and potentially a final `Special Step`. Moreover, the algorithm executes a phase if $\lambda_H \leq (3/2)\lambda^*$.

We say that $\lambda_H^i$ *belongs* to superphase $j$ if the $i$-th phase is executed during superphase $j$ and $\lambda_H^i \leq (3/2)\lambda_j^*$. We remark that the number of vertices in $H$ changes only at the beginning of a superphase and remains unchanged during its lifespan.

Let $n_j$ denote the number of vertices in some superphase $j$. We bound this quantity as follows.

Proposition 4.14. *Let $j$ be a superphase during the execution of the algorithm. Then we have*

$$n_j = O((n \log n)/\lambda_H^i) \text{ for all } \lambda_H^i \text{ belonging to superphase } j.$$

Proof. From Step 3 and Theorem 4.3, we know that $n_j = O((n \log n)/\lambda_j^*)$. Moreover, observe that $\lambda_j^* \leq \lambda_H^i$ and a phase is executed whenever $\lambda_H^i \leq (3/2)\lambda_j^*$. Thus, for all $\lambda_H^i$'s belonging to superphase $j$, we get the following relation:

$$\lambda_j^* \leq \lambda_H^i \leq (3/2)\lambda_j^*, \tag{1}$$

which in turn implies that $n_j = O((n \log n)/\lambda_j^*) = O((n \log n)/\lambda_H^i)$. □

For the remaining steps, we divide the running time analysis into two parts: one part corresponding to phases and the other to superphases.

*Part 1.* For some superphase $j$, the $i$-th phase consists of the $i$-th execution of a `Partial Rebuild Step` followed by the execution of Step 2. Let $u_i$ be the number of edge insertions in Phase $i$. By Theorem 4.4 and the fact that heap insertions are performed in $O(\log n)$ time, it follows that the total time for Step 2 during the $i$-th phase is $O(n_j + u_i \log n) = O((n + u_i) \log n)$. Since $n_j = O((n \log n)/\lambda_j^*)$, we observe that $\bigcup_{i \leq (3/2)\lambda_j^*+1} F_i \cup N_h$ has size $O(u_{i-1} + \lambda_j^* n_j) = O((u_{i-1} + n) \log n)$. Thus, the total time for computing DA-msfd in a `Partial Rebuild Step` is $O((u_{i-1} + n) \log n)$. Using Proposition 4.14, note that $H'$ has $O(\lambda_H^i n_j) = O(n \log n)$ edges, and thus it takes $O(n \log^2 n)$ time to compute $I(H', \lambda_H^i)$ and the new cactus tree.

The total time spent in Phase $i$ is $O((u_{i-1} + u_i + n) \log^2 n)$. Let $\lambda$ and $\lambda_H$ denote the size of the minimum cut in the final graph and its corresponding multigraph, respectively. Note that $\sum_{i=1}^{\lambda} u_i \leq m_1$, $\lambda n \leq m_0 + m_1$ and recall Equation (1). This gives that the total work over all phases is

$$\sum_{i=1}^{\lambda_H} O((u_{i-1} + u_i + n) \log^2 n) = \sum_{i=1}^{\lambda} O((u_{i-1} + u_i + n) \log^2 n) = O((m_0 + m_1) \log^2 n).$$

*Part 2.* The $j$-th superphase consists of the $j$-th execution of a `Full Rebuild Step` along with a possible execution of a `Special Step`, depending on whether the condition is met. In a `Full Rebuild Step`, computing $\lambda(G)$ takes $O((m_0 + m_1) \log^2 n \log\log^2 n)$ time. The total running time for constructing $H$, $I(H, \lambda_j^*)$ and the cactus tree is dominated by $O((m_0 + m_1 + (\lambda_j^*)^2 \cdot (n/\lambda_j^*)) \log^2 n \log\log^2 n) = O((m_0 + m_1) \log^2 n \log\log^2 n)$. The running time of a `Special Step` is $O(m_1 \log n)$.

Throughout its execution, the algorithm begins a new superphase whenever $\lambda(G) = \min\{\textsc{Min}(\mathcal{H}_G), \lambda_H\} > (3/2)\lambda^*$. This implies that $\lambda(G)$ must be at least $(3/2)\lambda^*$, where $\lambda^*$ is the value of $\lambda(G)$ at the last `Full Rebuild Step`. Thus, a new superphase begins whenever $\lambda(G)$ has increased by a factor of 3/2 (i.e., only $O(\log n)$ times over all insertions). This gives that the total time over all superphases is $O((m_0 + m_1) \log^3 n \log\log^2 n)$. □

## 5 INCREMENTAL $(1 + \varepsilon)$ MINIMUM CUT WITH $\widetilde{O}(n)$ SPACE

In this section, we present two $\widetilde{O}(n)$ space incremental Monte Carlo algorithms that w.h.p. maintain the size of a min-cut up to a $(1 + \varepsilon)$ factor. Both algorithms have $\widetilde{O}(1)$ update time and $\widetilde{O}(1)$, respectively $O(1)$, query-time. The first algorithm uses $O(n \log^2 n/\varepsilon^2)$ space, whereas the second one improves the space complexity to $O(n \log n/\varepsilon^2)$.

### 5.1 An $O(n \log^2 n/\varepsilon^2)$ Space Algorithm

Our first algorithm follows an approach that was used in several previous works [17, 32, 33]. The basic idea is to maintain the min-cut up to some size $k$ using small space. We achieve this by maintaining a sparse $(k + 1)$ certificate and incorporating it into the incremental exact min-cut

**ALGORITHM 2:** Incremental Exact Min-Cut Up to Size $k$

1: Set $\lambda = 0$, initialize $k+1$ union-find data structures $\mathcal{F}_1, \ldots, \mathcal{F}_{k+1}$,
$k+1$ empty forests $F_1, \ldots, F_{k+1}$, $I(G, \lambda)$, and an empty cactus tree.
2: **while** there is at least one minimum cut of size $\lambda$ **do**
**Receive the next operation**.
**if** it is a query **then return** $\lambda$
**else** it is the insertion of an edge $e$, **then**
Set $i = (k+1)$-Connectivity$(e)$.
**if** $i \neq$ NULL **then**
Set $F_i = F_i \cup \{e\}$.
Update the cactus tree according to the insertion of the edge $e$.
**endif**
**endif**
**endwhile**
3: Set $\lambda = \lambda + 1$.
Let $G' = (V, E')$ be a graph with $E' = I(G, \lambda - 1) \cup \bigcup_{i \leq \lambda+1} F_i$.
Compute $I(G', \lambda)$ and a cactus tree of $G'$.
**Goto** 2.

algorithm due to Henzinger [17], as described in Section 4. Finally, we apply the well-known randomized sparsification result due to Karger [20] to obtain our result.

*Maintaining min-cut up to size k using $O(kn)$ space.* We incrementally maintain an msfd for an unweighted graph $G$ using $k+1$ union-find data structures $\mathcal{F}_1, \ldots, \mathcal{F}_{k+1}$ (see Cormen et al. [6]). Each $\mathcal{F}_i$ maintains a spanning forest $F_i$ of $G$. Recall that $F_1, \ldots, F_{k+1}$ are edge disjoint. When a new edge $e = (u, v)$ is inserted into $G$, we define $i$ to be the first index such that $\mathcal{F}_i$.Find$(u) \neq \mathcal{F}_i$.Find$(v)$. If we find such an $i$, we append the edge $e$ to the forest $F_i$ by setting $\mathcal{F}_i$.Union$(u, v)$ and return $i$. If such an $i$ cannot be found after $k+1$ steps, we simply discard edge $e$ and return NULL. We refer to such procedure as $(k+1)$-Connectivity$(e)$.

It is easy to see that the forests maintained by $(k+1)$-Connectivity$(e)$ for every newly inserted edge $e$ are indeed edge disjoint. Combining this procedure with techniques from Henzinger [17] leads to Algorithm 2.

The correctness of the preceding algorithm is immediate from Lemmas 4.8 and 4.10. The running time and query bounds follow from Theorem 8 of Henzinger [17]. For the sake of completeness, we provide a full proof here.

Corollary 5.1. *For $k > 0$, there is an $O(kn)$ space algorithm that processes a stream of edge insertions starting from any empty graph $G$ and maintains an exact value of $\min\{\lambda(G), k\}$. Starting from an empty graph, the total time for inserting $m$ edges is $O(km\alpha(n) \log n)$ and queries can be answered in constant time, where $\alpha(n)$ stands for the inverse of Ackermann function.*

Proof. We first analyze Step 1. Initializing $k+1$ union-find data structures takes $O(kn)$ time. The running time for constructing $I(G, \lambda)$ and building an empty cactus tree is also dominated by $O(kn)$. Thus, the total time for Step 1 is $O(kn)$.

Let $\lambda_0, \ldots, \lambda_f$, where $\lambda_f \leq k$, be the values that $\lambda$ assumes in Step 2 during the execution of the algorithm in increasing order. We define Phase $i$ to be all steps executed while $\lambda = \lambda_i$. For $i \geq 1$, we can view Phase $i$ as the $i$-th execution of Step 3 followed by the execution of Step 2. Let $u_i$ denote the number of edge insertion in Phase $i$. The total time for testing the $(k+1)$-connectivity of the endpoints of the newly inserted edges and updating the cactus tree in Step 2 is dominated by $O(n + k\alpha(n)u_i)$. Since the graph $G'$ in Step 3 has always at most $O(kn)$ edges, the running time

**ALGORITHM 3:** $(1+\varepsilon)$-Min-Cut With $O(n\log^2 n/\varepsilon^2)$ Space

1: **For** $i = 0, \ldots, \lfloor \log n \rfloor$, let $G_i$ be an initially empty sampled subgraph.
2: **Receive the next operation**.
  **if** it is a query **then**
    Find the minimum $j$ such that $\lambda(G_j) \leq k$ and **return** $2^j\lambda(G_j)/(1-\varepsilon)$.
  **else** it is the insertion of an edge $e$, **then**
    Include edge $e$ to each $G_i$ with probability $1/2^i$.
    Maintain the exact min cut of each $G_i$ up to size $k = 48\log n/\varepsilon^2$ using Algorithm 2.
  **endif**
3: **Goto** 2.

to compute $I(G', \lambda)$ and the cactus tree of $G'$ is $O(kn\log n)$. Combining the preceding bounds, the total time spent in Phase $i$ is $O(k(\alpha(n)u_i + n\log n))$. Thus, the total work over all phases is $O(km\alpha(n)\log n)$.

The space complexity of the algorithm is only $O(kn)$, as we always maintain at most $k+1$ spanning forests during its execution. □

*Dealing with min-cuts of arbitrary size.* We observe that Corollary 5.1 gives polylogarithmic amortized update time only for min-cuts up to some polylogarithmic size. For dealing with min-cuts of arbitrary size, we use the well-known sampling technique due to Karger [20]. This allows us to get an $(1+\varepsilon)$-approximation to the value of a min-cut with high probability.

Lemma 5.2 ([20]). *Let $G$ be any graph with minimum cut $\lambda$, and let $p \geq 12(\log n)/(\varepsilon^2\lambda)$. Let $G(p)$ be a subgraph of $G$ obtained by including each of edge of $G$ to $G(p)$ with probability $p$ independently. Then the probability that the value of any cut of $G(p)$ has value more than $(1+\varepsilon)$ or less than $(1-\varepsilon)$ times its expected value is $O(1/n^4)$.*

For some integer $i \geq 1$, let $G_i$ denote a subgraph of $G$ obtained by including each edge of $G$ to $G_i$ with probability $1/2^i$ independently. We now have all necessary tools to present our incremental algorithm.

Theorem 5.3. *There is an $O(n\log^2 n/\varepsilon^2)$ space randomized algorithm that processes a stream of edge insertions starting from an empty graph $G$ and maintains a $(1+\varepsilon)$-approximation to a min-cut of $G$ with high probability. The amortized update time per operation is $O(\alpha(n)\log^3 n/\varepsilon^2)$, and queries can be answered in $O(\log n)$ time.*

Proof. We first prove the correctness of the algorithm. For an integer $t \geq 0$, let $G^{(t)} = (V, E^{(t)})$ be the graph after the first $t$ edge insertions. Further, let $\lambda(G^{(t)})$ denote the min-cut of $G^{(t)}$, $p^{(t)} = 12(\log n)/(\varepsilon^2\lambda^{(t)})$ and $\lambda(S, G) = |E_G(S, V\setminus S)|$ for some cut $(S, V\setminus S)$. For any integer $i \leq \lfloor \log_2 1/p^{(t)} \rfloor$, Lemma 5.2 implies that for any cut $(S, V\setminus S)$, $((1-\varepsilon)/2^i)\lambda(S, G^{(t)}) \leq \lambda(S, G_i^{(t)}) \leq ((1+\varepsilon)/2^i)\lambda(S, G^{(t)})$ with probability $1 - O(1/n^4)$. Let $(S^*, V\setminus S^*)$ be a min-cut of $G_i^{(t)}$ (i.e., $\lambda(S^*, G_i^{(t)}) = \lambda(G_i^{(t)})$). Setting $i = \lfloor \log_2 1/p^{(t)} \rfloor$, we get that

$$\mathbb{E}\left[\lambda\left(G_i^{(t)}\right)\right] \leq \lambda\left(G^{(t)}\right)/2^i \leq 2p^{(t)}\lambda\left(G^{(t)}\right) \leq 24\log n/\varepsilon^2.$$

The latter along with the implication of Lemma 5.2 give that for any $\varepsilon \in (0,1)$, the size of the minimum cut in $G_i^{(t)}$ is at most $(1+\varepsilon)24\log n/\varepsilon^2 \leq 48\log n/\varepsilon^2$ with probability $1 - O(1/n^4)$. Thus, $j \leq \lfloor \log_2 1/p^{(t)} \rfloor$ with probability $1 - O(1/n^4)$. Additionally, we observe that the algorithm returns a $(1+O(\varepsilon))$-approximation to a min-cut of $G^{(t)}$ w.h.p. since by Lemma 5.2,

$2^i\lambda(G_i^{(t)})/(1-\varepsilon) \leq (1+\varepsilon)/(1-\varepsilon)\lambda(G^{(t)}) = (1+O(\varepsilon))\lambda(G^{(t)})$ w.h.p. Note that for any $t$, $\lfloor \log_2 1/p^{(t)} \rfloor \leq \lfloor \log n \rfloor$, and thus it is sufficient to maintain only $O(\log n)$ sampled subgraphs.

Since our algorithm applies to unweighted simple graphs, we know that $t \leq O(n^2)$. Now applying union bound over all $t \in \{1, \ldots O(n^2)\}$ gives that the probability that the algorithm does not maintain a $1+O(\varepsilon)$-approximation is at most $O(1/n^2)$.

The total expected time for maintaining a sampled subgraph is $O(m\alpha(n)\log^2 n/\varepsilon^2)$, and the required space is $O(n \log n/\varepsilon^2)$ (Corollary 5.1). Maintaining $O(\log n)$ such subgraphs gives an $O(\alpha(n)\log^3 n/\varepsilon^2)$ amortized time per edge insertion and an $O(n\log^2 n/\varepsilon^2)$ space requirement. The $O(\log n)$ query time follows, as in the worst case we scan at most $O(\log n)$ subgraphs, each answering a min-cut query in constant time. □

### 5.2 Improving the Space to $O(n \log n/\varepsilon^2)$

We next show how to bring down the space requirement of the previous algorithm to $O(n \log n/\varepsilon^2)$ without degrading its running time. The main idea is to keep a single sampled subgraph instead of $O(\log n)$ of them.

Let $G = (V, E)$ be an unweighted undirected graph, and assume that each edge is given some random weight $p_e$ chosen uniformly from $[0, 1]$. Let $G^w$ be the resulting weighted graph. For any $p > 0$, we denote by $G(p)$ the unweighted subgraph of $G$ that consists of all edges that have weight at most $p$. We state the following lemma due to Karger [18].

Lemma 5.4. *Let $k = 48 \log n/\varepsilon^2$. Given a connected graph $G$, let $p$ be a value such that $p \geq k/(4\lambda(G))$. Then with high probability, $\lambda(G(p)) \leq k$ and $\lambda(G(p))/p$ is an $(1+\varepsilon)$-approximation to a min-cut of $G$.*

Proof. Since the weight of every edge is uniformly distributed, the probability that an edge has weight at most $p$ is exactly $p$. Thus, $G(p)$ is a graph that contains every edge of $G$ with probability $p$. The claim follows from Lemma 5.2. □

For any graph $G$ and some appropriate weight $p \geq k/(4\lambda(G))$, the preceding lemma tells us that the min-cut of $G(p)$ is bounded by $k$ with high probability. Thus, instead of considering the graph $G$ along with its random edge weights, we build a collection of $k+1$ minimum edge-disjoint spanning forests (using those edge weights). We note that such a collection is an msfd of order $k+1$ for $G$ with $O(kn)$ edges, and by Lemma 4.8, it preserves all minimum cuts of $G$ up to size $k$.

Our algorithm uses the following two data structures.

(1) NI-Sparsifier($k$) data-structure. Given a graph $G$, where each edge $e$ is assigned some weight $p_e$ and some parameter $k$, we devise an insertion-only data structure that maintains a collection of $k+1$ minimum edge-disjoint spanning forests $F_1, \ldots, F_{k+1}$ with respect to the edge weights. Let $F = \bigcup_{i \leq k+1} F_i$. Since we are in the incremental setting, it is known that the problem of maintaining a single minimum spanning forest can be solved in time $O(\log n)$ per insertion using the dynamic tree structure of Sleator and Tarjan Sleator and Tarjan [31]. Specifically, we use this data structure to determine for each pair of nodes $(u, v)$ the maximum weight of an edge in the cycle that the edge $(u, v)$ induces in the minimum spanning forest $F_i$. Let max-weight$(F_i(u, v))$ denote such a maximum weight. The update operation works as follows. When a new edge $e = (u, v)$ is inserted into $G$, we first use the dynamic tree data structure to test whether $u$ and $v$ belong to the same tree. If no, we link their two trees with the edge $(u, v)$ and return the pair (TRUE, NULL) to indicate that $e$ was added to $F_i$ and no edge was evicted from $F_i$. Otherwise, we check whether $p_e >$ max-weight$(F_i(e))$. If the latter holds, we make no changes in the forest and return (FALSE, $e$). Otherwise, we replace one of the maximum edges, say $e'$, on the path between $u$ and $v$ in the tree by $e$ and return (TRUE, $e'$). The Boolean value that is returned indicates whether $e$ belongs to

$F_i$ or not, and the second value that is returned gives an edge that does not (or no longer) belong to $F_i$. Note that each edge insertion requires $O(\log n)$ time. We refer to this insert operation as Insert-MSF$(F_i, e, p_e)$.

Now, the algorithm that maintains the weighted minimum spanning forests implements the following operations:

- Initialize-NI $(k)$: Initializes the data structure for $k+1$ empty minimum spanning forests.
- Insert-NI $(e, p_e)$: Set $i = 1$, $e' = e$, taken = FALSE.

  **while** $((i \leq k+1)$ and $e' \neq$ NULL$)$ **do**
  Set $(t', e'')$ = Insert-MSF$(F_i, e', p_{e'})$.
  **if** $(e' = e)$ **then** set taken $= t'$ **endif**
  Set $e' = e''$ and $i = i + 1$.
  **endwhile**
  **if** $(e' \neq e)$ **then return** (taken, $e'$) **else return** (taken, NULL).

The Boolean value that is returned indicates whether $e$ belongs to $F$ or not, and the second value returns an edge that is removed from $F$, if any.

Recall that $F = \bigcup_{i \leq k+1} F_i$. We use the abbreviation NI-Sparsifier$(k)$ to refer to this data structure. Throughout the algorithm, we will associate a weight with each edge in $F$ and use $F^w$ to refer to this weighted version of $F$.

Lemma 5.5. *For $k > 0$ and any graph $G$, NI-Sparsifier$(k)$ maintains a weighted mfsd of order $k+1$ of $G$ under edge insertions. The algorithm uses $O(kn)$ space, and the total time for inserting $m$ edges is $O(km \log n)$.*

Proof. We first show that NI-Sparsifier$(k)$ maintains a forest decomposition such that (1) the forests are edge disjoint and (2) each forest is maximal. We proceed by induction on the number $m$ of edge insertions.

For $m = 0$, the forest decomposition is empty. Thus the edge disjointness and maximality of forests trivially hold. For $m > 0$, consider the $m$-th edge insertion, which inserts an edge $e$. Let $F'$ (respectively $F$) denote the union of forests before (respectively after) the insertion of edge $e$. By the inductive assumption, $F'$ satisfies (1) and (2). If $F = F'$ (i.e., the edge $e$ was not added to any of the forests when Insert-NI$(e, p_e)$ was called), then $F$ also satisfies (1) and (2). Otherwise, $F \neq F'$, and note that by construction that $e$ is appended to exactly one forest. Let $F'_j$ (respectively $F_j$) denote such maximal forest before (respectively after) the insertion of $e$. We distinguish two cases. If $e$ links two trees of $F'_j$, then $F_j$ is also a maximal forest and forests of $F$ are edge disjoint. Thus, $F$ satisfies (1) and (2). Otherwise, the addition of $e$ results in the deletion of another edge $e' \in F'_j$. It follows that $F_j$ is maximal and the current forests are edge disjoint. Applying a similar argument to the addition of edge $e'$ in the remaining forests, we conclude that $F$ satisfies (1) and (2).

We next argue about time and space complexity. The dynamic tree data structure can be implemented in $O(n)$ space, where each query regarding max-weight$(F_i(u, v))$ can be answered in $O(\log n)$ time. Since the algorithm maintains $k+1$ such forests, the space requirement is $O(kn)$. The total running time follows since insertion of an edge can result in at most $k+1$ executions of the Insert-MSF$(F_i, e, p_e)$ procedures, each running in $O(\log n)$ time.

(2) Limited Exact Min-Cut$(k)$ data structure. We use Algorithm 2 to implement the following operations for any unweighted graph $G$ and parameter $k$:

- Insert-Limited $(e)$: Executes the insertion of edge $e$ using Algorithm 2
- Query-Limited (): Returns $\lambda$

**ALGORITHM 4:** $(1+\varepsilon)$-Min-Cut With $O(n \log n/\varepsilon^2)$ Space

1: Set $k = 48 \log n/\varepsilon^2$.
Set $p = 12 \log n/\varepsilon^2$.
Let $H$ and $F^w$ be empty graphs.
2: Initialize-Limited$(H, k)$.
**while** Query-Limited() $< k$ **do**
**Receive the next operation**.
**if** it is a query **then return** Query-Limited()/ $\min\{1, p\}$.
**else** it is the insertion of an edge $e$, **then**
Sample a random weight from $[0, 1]$ for the edge $e$ and denote it by $p_e$.
**if** $p_e \leq p$ **then** Insert-Limited$(e)$ **endif**
Set (taken, $e'$) = Insert-NI$(e, p_e)$.
**if** taken **then**
Insert $e$ into $F^w$ with weight $p_e$.
**if** ($e' \neq$ NULL) **then** remove $e'$ from $F^w$.
**endif**
**endif**
**endwhile**
3: // `Rebuild Step`
Set $p = p/2$.
Let $H$ be the unweighted subgraph of $F^w$ consisting of all edges of weight at most $p$.
**Goto** 2.

- Initialize-Limited $(G, k)$: Builds a data structure for $G$ with parameter $k$ by executing Step 1 of Algorithm 2 and then Insert-Limited$(e)$ for each edge $e$ in $G$.

We use the abbreviation Lim$(k)$ to refer to this type of data structure.

Combining the preceding data structures leads to Algorithm 4.

*Correctness and running time analysis.* Throughout the execution of Algorithm 4, $F$ corresponds exactly to the msfd of order $k+1$ of $G$ maintained by NI-Sparsifier$(k)$. In the following, let $H$ be the graph that is given as input to Lim$(k)$. Thus, by Corollary 5.1, Query-Limited() returns $\min\{k, \lambda(H)\}$—that is, it returns $\lambda(H)$ as long as $\lambda(H) \leq k$. We now formally prove the correctness.

Lemma 5.6. *Let $\epsilon \leq 1, k = 48 \log n/\varepsilon^2$, and assume that the algorithm is started on an empty graph. As long as $\lambda(G) < k$, we have $H = G$, $p = k/4$, and Query-Limited() returns $\lambda(G)$. The first rebuild step is triggered after the first insertion that increases $\lambda(G)$ to $k$, and at that time, it holds that $\lambda(G) = \lambda(H) = k$.*

Proof. The algorithm starts with an empty graph $G$ (i.e., initially $\lambda(G) = 0$). Throughout the sequence of edge insertions, $\lambda(G)$ never decreases. We show by induction on the number $m$ of edge insertions that $H = G$ and $p = k/4$ as long as $\lambda(G) < k$.

Note that $k/4 \geq 1$ by our choice of $\epsilon$. For $m = 0$, the graphs $G$ and $H$ are both empty graphs, and $p$ is set to $k/4$. For $m > 0$, consider the $m$-th edge insertion, which inserts an edge $e$. Let $G$ and $H$ denote the corresponding graphs after the insertion of $e$. By the inductive assumption, $p = k/4$ and $G \setminus \{e\} = H \setminus \{e\}$. As $p \geq 1$, $e$ is added to $H$, and thus it follows that $G = H$. Hence, $\lambda(H) = \lambda(G)$. If $\lambda(G) < k$ but $\lambda(G \setminus \{e\}) < k$, no rebuild is performed and $p$ is not changed. If $\lambda(G) = k$, then the last insertion was exactly the insertion that increased $\lambda(G)$ from $k-1$ to $k$. As $H = G$ before the rebuild, Query-Limited() returns $k$, triggering the first execution of the rebuild step. □

We next analyze the case that $\lambda(G) \geq k$. In this case, both $H$ and $p$ are random variables, as they depend on the randomly chosen weights for the edges. Let $F(p)$ be the unweighted subgraph of $F^w$ that contains all edges of weight at most $p$.

LEMMA 5.7. *Let $N_h(p)$ be the graph consisting of all edges that were inserted after the last rebuild and have weight at most $p$, and let $F^{old}(p)$ be $F(p)$ right after the last rebuild. Then it holds that $H = F^{old}(p) \cup N_h(p)$.*

PROOF. Up to the first rebuild, $N_h = G$ and $p \geq 1$. Thus, $N_h(p) = N_h = G$. Lemma 5.6 shows that until the first rebuild, $H = G$. As $F^{\text{old}}(p) = \emptyset$, it follows that $H = G = N_h(p) \cup F^{\text{old}}(p)$ up to the first rebuild.

Immediately after each rebuild step, $N_h = \emptyset$ and $H$ is set to be $F(p)$, and thus the claim holds. After each subsequent edge insertion that does not trigger a rebuild, the newly inserted edge is added to $N_h(p)$ and to $H$ if and only if its weight is at most $p$. Thus, both $N_h(p)$ and $H$ change in the same way, which implies that $H = F^{\text{old}}(p) \cup N_h(p)$. □

LEMMA 5.8. *At the time of a rebuild, $F(p)$ is an msfd of order $k+1$ of $G(p)$.*

PROOF. NI-SPARSIFIER maintains a maximal spanning forest decomposition based on minimum-weight spanning forests $F_1, \ldots F_{k+1}$ of $G$ using the weights $p_e$. Now consider the hierarchical decomposition $F_1(p), \ldots, F_{k+1}(p)$ of $G(p)$ induced by taking only the edges of weight at most $p$ of each forest $F_i$. Note that NI-SPARSIFIER would return exactly the same hierarchy $F_1(p), \ldots, F_{k+1}(p)$ if only the edges of $G(p)$ were inserted into NI-SPARSIFIER. Thus, $F_1(p), \ldots, F_{k+1}(p)$ is an msfd of order $k+1$ of $G(p)$. □

To show that $\lambda(H)/\min\{1,p\}$ is an $(1+\varepsilon)$-approximation of $\lambda(G)$ with high probability, we need to show that if $\lambda(G) \geq k$, then (a) the random variable $p$ is at least $k/(4\lambda(G))$ w.h.p., which implies that $\lambda(G(p))$ is a $(1+\varepsilon)$-approximation of $\lambda(G)$ w.h.p. and (b) that $\lambda(H) = \lambda(G(p))$ (by Lemma 5.4).

LEMMA 5.9. *Let $\varepsilon \leq 1$. If $\lambda(G) \geq k$, then (1) $p \geq k/(4\lambda(G))$ with probability $1 - O(\log n/n^4)$ and (2) $\lambda(H) = \lambda(G(p))$.*

PROOF. For any $i \geq 0$, after the $i$-th rebuild we have $p = p^{(i)} := 12\log n/(2^i\varepsilon^2)$. Let $\ell = \lfloor \log(12\log n/\varepsilon^2) \rfloor$ denote the index of the last rebuild at which $p^{(i)} \geq 1$. For any $i \geq \ell + 1$, we will show by induction on $i$ that (1) $p^{(i)} = 12\log n/(2^i\varepsilon^2) \geq 12\log n/(\varepsilon^2\lambda(G))$ with probability $1 - O((i-1-\ell)/n^4)$, which is equivalent to showing that $\lambda(G) \geq 2^i$, and that (2) at any point between the $(i-1)$-st and the $i$-th rebuild, $\lambda(H) = \lambda(G(p^{(i-1)}))$.

Once we have shown this, we can argue that the number of rebuild steps is small, thus giving the claimed probability in the lemma. Indeed, note that $\lambda(G) \leq n$ since $G$ is unweighted. Additionally, from earlier, we get that after the $i$-th rebuild, $\lambda(G) \geq 2^i$ with high probability. Combining these two bounds yields $i \leq O(\log n)$ w.h.p. (i.e., the number of rebuild steps is at most $O(\log n)$).

We first analyze $i = \ell + 1$. Note that $\ell + 1$ is the index of the first rebuild at which $p^{(i)} < 1$. Assume that the insertion of some edge $e$ caused the first rebuild. Lemma 5.6 showed that (1) at the first rebuild $\lambda(G) = k$ and (2) that up to the first rebuild $G(p) = G = H$. We observe that (1) and (2) remain true up to the $(\ell+1)$-st rebuild. In addition, $\lambda(G) = k \geq 24\log n/\varepsilon^2 \geq 2^i$, which implies that $p^{(i)} \geq 1/2$. This shows the base case.

For the induction step ($i > \ell + 1$), we inductively assume that (1) at the $(i-1)$-st rebuild, $p^{(i-1)} \geq 12\log n/(\varepsilon^2\lambda(G^{\text{old}}))$ with probability $1 - O((i-2-\ell)/n^4)$, where $G^{\text{old}}$ is the graph $G$ right before the insertion that triggered the $i$-th rebuild (i.e., at the last point in time when QUERY-LIMITED() returned a value less than $k$), and (2) that $\lambda(H) = \lambda(G(p^{(i-2)}))$ at any time between the $(i-2)$-nd and the $(i-1)$-st rebuild. Let $e$ be the edge whose insertion caused the $i$-th rebuild.

Define $G^{\text{new}} = G^{\text{old}} \cup \{e\}$. By the induction hypothesis, with probability $1 - O((i-2-\ell)/n^4)$, $p^{(i-1)} \geq 12 \log n/(\varepsilon^2 \lambda(G^{\text{old}})) \geq 12 \log n/(\varepsilon^2 \lambda(G^{\text{new}}))$, as $\lambda(G^{\text{old}}) \leq \lambda(G^{\text{new}})$. Thus, by Lemma 5.4, we get that $\lambda(G^{\text{new}}(p^{(i-1)}))/p^{(i-1)} \leq (1+\varepsilon)\lambda(G^{\text{new}})$ with probability $1 - O(1/n^4)$. Applying an union bound, we get that the two previous statements hold simultaneously with probability $1 - O((i-1-\ell)/n^4)$.

We show in the following that $\lambda(G^{\text{new}}(p^{(i-1)})) = \lambda(H^{\text{new}})$, where $H^{\text{new}}$ is the graph stored in Lim($k$) right before the $i$-th rebuild. Thus, $\lambda(H^{\text{new}}) = k$, which implies that

$$\begin{aligned}\lambda(G^{\text{new}}(p^{(i-1)})) = k = 48 \log n/\varepsilon^2 &\leq (1+\varepsilon)\lambda(G^{\text{new}}) \cdot p^{(i-1)} \\ &= (1+\varepsilon)\lambda(G^{\text{new}}) \cdot 12 \log n/(2^{i-1}\varepsilon^2),\end{aligned}$$

with probability $1 - O((i-1-\ell)/n^4)$. This in turn implies that with probability $1 - O((i-1-\ell)/n^4)$, $\lambda(G^{\text{new}}) \geq 2^{i+1}/(1+\varepsilon) \geq 2^i$ by our choice of $\varepsilon$.

It remains to show that $\lambda(G^{\text{new}}(p^{(i-1)})) = \lambda(H^{\text{new}})$. Note that this is a special case of (2), which claims that at any point between that $(i-1)$-st and the $i$-th rebuild $\lambda(H) = \lambda(G(p^{(i-1)}))$, where $H$ and $G$ are the current graphs. Thus, to complete the proof of the lemma, it suffices to show (2).

As $H$ is a subgraph of $G(p^{(i-1)})$, we know that $\lambda(G(p^{(i-1)})) \geq \lambda(H)$. Thus, we only need to show that $\lambda(G(p^{(i-1)})) \leq \lambda(H)$. Let $G^{i-1}$ (respectively $F^{i-1}$, respectively $H^{i-1}$) be the graph $G$ (respectively $F$, respectively $H$) right after rebuild $i-1$, and let $N_h$ be the set of edges inserted since, for instance, $G = G^{(i-1)} \cup N_h$. As we showed in Lemma 5.7, $H = F^{i-1}(p^{(i-1)}) \cup N_h(p^{(i-1)})$. Thus, $H^{i-1} = F^{i-1}(p^{(i-1)})$. Additionally, by Lemma 5.8, $F^{i-1}(p^{(i-1)})$ is an msfd of order $k+1$ of $G^{i-1}(p^{(i-1)})$. Thus, by Lemma 3.2, for every cut $(A, V \setminus A)$ of value at most $k$ in $H^{i-1}$, $\lambda(A, H^{i-1}) = \lambda(F^{i-1}(p^{(i-1)}), A) = \lambda(A, G^{i-1}(p^{(i-1)}))$, where $\lambda(A, G) = |E_G(A, V \setminus A)|$. Now assume toward contradiction that $\lambda(G(p^{(i-1)})) > \lambda(H)$ and consider a minimum cut $(A, V \setminus A)$ in $H$ (i.e., $\lambda(H) = \lambda(A, H)$). We know that at any time, $k \geq \lambda(H)$. Thus, $k \geq \lambda(H) = \lambda(A, H)$, which implies that $k \geq \lambda(A, H^{i-1})$. By Lemma 3.2, it follows that $\lambda(A, H^{i-1}) = \lambda(A, G^{i-1}(p^{(i-1)}))$. Note that $H = H^{i-1} \cup N_h(p^{(i-1)})$ and $G(p^{(i-1)}) = G^{i-1}(p^{(i-1)}) \cup N_h(p^{(i-1)})$. Let $x$ be the number of edges of $N_h(p^{(i-1)})$ that cross the cut $(A, V \setminus A)$. Then $\lambda(H) = \lambda(H, A) = \lambda(A, H^{i-1}) + x = \lambda(A, G^{i-1}(p^{(i-1)})) + x = \lambda(A, G(p^{(i-1)}))$, which contradicts the assumption that $\lambda(G(p^{(i-1)})) > \lambda(H)$. □

Since our algorithm is incremental and applies only to unweighted graphs, we know that there can be at most $O(n^2)$ edge insertions. The preceding lemma implies that for any current graph $G$, Algorithm 4 returns a $(1+\varepsilon)$-approximation to a min-cut of $G$ with probability $1 - O(\log n/n^4)$. Applying an union bound over $O(n^2)$ possible different graphs gives that the probability that the algorithm does not maintain a $(1+\varepsilon)$-approximation is at most $O(\log n/n^2) = O(1/n)$. Thus, at any time, we return a $(1+\varepsilon)$-approximation with probability $1 - O(1/n)$.

Theorem 5.10. *There is an $O(n \log n/\varepsilon^2)$ space randomized algorithm that processes a stream of edge insertions starting from an empty graph $G$ and maintains a $(1+\varepsilon)$-approximation to a min-cut of $G$ with high probability. The total time for inserting $m$ edges is $O(m\alpha(n) \log^3 n/\varepsilon^2)$, and queries can be answered in constant time.*

Proof. The space requirement is $O(n \log n/\varepsilon^2)$ since at any point of time, the algorithm keeps $H$, $F^w$, Lim($k$), and NI-Sparsifier ($k$), each of size at most $O(n \log n/\varepsilon^2)$ (Corollary 5.1 and Lemma 5.5).

When Algorithm 4 executes a `Rebuild Step`, only the Lim($k$) data structure is rebuilt, but not NI-Sparsifier($k$). During the whole algorithm, $m$ Insert-NI operations are performed. Thus, by Lemma 5.5, the total time for all operations involving NI-Sparsifier($k$) is $O(m \log^2 n/\varepsilon^2)$.

It remains to analyze Steps 2 and 3. By Corollary 5.1, Initialize-Limited($H, k$) takes at most $O(m\alpha(n) \log^2 n/\varepsilon^2)$ total time (Step 2). The running time of Step 3 is $O(m)$ as well. Since the

number of `Rebuild Steps` is at most $O(\log n)$, it follows that the total time for all INITIALIZE-LIMITED$(H, k)$ calls in Step 2 and the total time of Step 3 throughout the execution of the algorithm is $O(m\alpha(n)\log^3 n/\varepsilon^2)$.

We are left with analyzing the remaining part of Step 2. Each query operation executes one QUERY-LIMITED() operation, which takes constant time. Each insertion executes one INSERT-NI$(e, p_e)$ operation, which takes amortized time $O(\log^2 n/\varepsilon)$. We maintain the edges of $F^w$ in a balanced binary tree so that each insertion and deletion takes $O(\log n)$ time. As there are $m$ edge insertions, the remaining part of Step 2 takes total time $O(m\log^2 n/\varepsilon^2)$. Combining the preceding bounds gives the theorem. □

## APPENDIX

## A MISSING PROOFS

Next we show a proof for Lemma 3.2 in Section 3. The arguments closely follow the work of Nagamochi and Ibaraki [28]. We first present the following helpful lemma.

LEMMA A.1. *Let $\mathcal{F} = (F_1, \ldots, F_m)$ be an* msfd *of order $m$ of a graph $G = (V, E)$. Then for any edge $(u, v) \in F_j$ and any $i \leq j$, it holds that $\lambda(u, v, \bigcup_{l\leq i} F_l) \geq i$.*

PROOF. Fix some edge $e = (u, v) \in F_j$. We first argue that for each $i = 1, \ldots, j-1$, the forest $(V, F_i)$ contains some $(u, v)$-path. Indeed, by the maximality of the forest $(V, F_i)$, the graph $(V, F_i \cup \{e\})$ must have some cycle $C$ that contains $e$. Thus, $P = C \setminus e$ is the $(u, v)$-path in the forest $(V, F_i)$. It follows that $(V, \bigcup_{l\leq i} F_l)$ has $i$ edge-disjoint paths. Next, observe that $G_j = (V, \bigcup_{l\leq j} F_l)$ has $j$ edge-disjoint paths, namely the $j-1$ edge-disjoint paths in $G_{j-1}$ (which does not contain the edge $(u, v)$) and the 1-edge path consisting of the edge $(u, v)$. Hence, $\lambda(u, v, \bigcup_{l\leq i} F_l) \geq i$. □

PROOF OF LEMMA 3.2. Assume that $\lambda(S, G) \leq k-1$. Then by definition of $G_k$, we know that $G_k$ preserves any cut $S$ of size up to $k$. Thus, $\lambda(S, G_k) = \lambda(S, G)$.

For the other case, $\lambda(S, G) \geq k$ and assume that $\lambda(S, G_k) < \lambda(S, G)$ (otherwise the lemma follows). Then there is an edge $e = (u, v) \in E_G(S, V \setminus S) \setminus E_{G_k}(S, V \setminus S)$. Since $e \notin \bigcup_{i\leq k} F_i$, this means that $e$ belongs to some forest $F_j$ with $j > k$. By Lemma A.1, we have that $\lambda(u, v, G_k) \geq k$. Since $(S, V \setminus S)$ separates $u$ and $v$ in $G_k$, it follows that $\lambda(S, G_k) = |E_{G_k}(S, V \setminus S)| \geq \lambda(u, v, G_k) \geq k$. □

## ACKNOWLEDGMENTS

We thank the two anonymous reviewers for their suggestions and comments, which improved the quality of the article.

## REFERENCES

[1] Amir Abboud and Virginia Vassilevska Williams. 2014. Popular conjectures imply strong lower bounds for dynamic problems. In *Proceedings of the 55th Symposium on Foundations of Computer Science (FOCS'14)*. IEEE, Los Alamitos, CA, 434–443. DOI: https://doi.org/10.1109/FOCS.2014.53

[2] Kook Jin Ahn and Sudipto Guha. 2009. Graph sparsification in the semi-streaming model. In *Proceedings of the 36th International Colloquium on Automata, Languages, and Programming (ICALP'09)*. 328–338. DOI: https://doi.org/10.1007/978-3-642-02930-1_27

[3] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. 2012. Graph sketches: Sparsification, spanners, and subgraphs. In *Proceedings of the 32nd Symposium on Principles of Database Systems (PODS'12)*. ACM, New York, NY, 5–14. DOI: https://doi.org/10.1145/2213556.2213560

[4] András A. Benczúr and David R. Karger. 2015. Randomized approximation schemes for cuts and flows in capacitated graphs. *SIAM Journal on Computing* 44, 2, 290–319. DOI: https://doi.org/10.1137/070705970

[5] Sayan Bhattacharya, Monika Henzinger, Danupon Nanongkai, and Charalampos E. Tsourakakis. 2015. Space- and time-efficient algorithm for maintaining dense subgraphs on one-pass dynamic streams. In *Proceedings of the 47th Symposium on Theory of Computing (STOC'15)*. ACM, New York, NY, 173–182. DOI: https://doi.org/10.1145/2746539.2746592

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (3rd ed.). MIT Press, Cambridge, MA.

[7] E. A. Dinitz, A. V. Karzanov, and M. V. Lomonosov. 1976. On the structure of a family of minimum weighted cuts in a graph. *Studies in Discrete Optimization* 1976, 290–306.

[8] Yefim Dinitz and Jeffery Westbrook. 1998. Maintaining the classes of 4-edge-connectivity in a graph on-line. *Algorithmica* 20, 3, 242–276. DOI: https://doi.org/10.1007/PL00009195

[9] Tamás Fleiner and András Frank. 2009. A quick proof for the cactus representation of mincuts. Retrieved February 10, 2018, from http://web.cs.elte.hu/~frank/cikkek/FrankR3.pdf.

[10] Harold N. Gabow. 1991. Applications of a poset representation to edge connectivity and graph rigidity. In *Proceedings of the 32nd Symposium on Foundations of Computer Science (FOCS'91)*. IEEE, Los Alamitos, CA, 812–821. DOI: https://doi.org/10.1109/SFCS.1991.185453

[11] Harold N. Gabow. 1995. A matroid approach to finding edge connectivity and packing arborescences. *Journal of Computer and System Sciences* 50, 2, 259–273. DOI: https://doi.org/10.1006/jcss.1995.1022

[12] Zvi Galil and Giuseppe F. Italiano. 1993. Maintaining the 3-edge-connected components of a graph on-line. *SIAM Journal on Computing* 22, 1, 11–28. DOI: https://doi.org/10.1137/0222002

[13] David Gibb, Bruce M. Kapron, Valerie King, and Nolan Thorn. 2015. Dynamic graph connectivity with improved worst case update time and sublinear space. arXiv:1509.06464.

[14] Gramoz Goranci, Monika Henzinger, and Mikkel Thorup. 2016. Incremental exact min-cut in poly-logarithmic amortized update time. In *Proceedings of the 24th European Symposium on Algorithms (ESA'16)*. 46:1–46:17. DOI: https://doi.org/10.4230/LIPIcs.ESA.2016.46

[15] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. 2015. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *Proceedings of the 47th Symposium on Theory of Computing (STOC'15)*. ACM, New York, NY, 21–30. DOI: https://doi.org/10.1145/2746539.2746609

[16] Monika Henzinger, Satish Rao, and Di Wang. 2017. Local flow partitioning for faster edge connectivity. In *Proceedings of the 28th Symposium on Discrete Algorithms (SODA'17)*. 1919–1938. DOI: https://doi.org/10.1137/1.9781611974782.125

[17] Monika Rauch Henzinger. 1997. A static 2-approximation algorithm for vertex connectivity and incremental approximation algorithms for edge and vertex connectivity. *Journal of Algorithms* 24, 1, 194–220. DOI: https://doi.org/10.1006/jagm.1997.0855

[18] David Karger. 1994. *Random Sampling in Graph Optimization Problems*. Ph.D. Dissertation. Stanford University, Stanford, CA.

[19] David R. Karger. 1994. Using randomized sparsification to approximate minimum cuts. In *Proceedings of the 5th Symposium on Discrete Algorithms (SODA'94)*. 424–432.

[20] David R. Karger. 1999. Random sampling in cut, flow, and network design problems. *Mathematics of Operations Research* 24, 2, 383–413. DOI: https://doi.org/10.1287/moor.24.2.383

[21] David R. Karger. 2000. Minimum cuts in near-linear time. *Journal of the ACM* 47, 1, 46–76. DOI: https://doi.org/10.1145/331605.331608

[22] Ken-Ichi Kawarabayashi and Mikkel Thorup. 2015. Deterministic global minimum cut of a simple graph in near-linear time. In *Proceedings of the 47th Symposium on Theory of Computing (STOC'15)*. ACM, New York, NY, 665–674. DOI: https://doi.org/10.1145/2746539.2746588

[23] Jonathan A. Kelner and Alex Levin. 2013. Spectral sparsification in the semi-streaming setting. *Theory of Computing Systems* 53, 2, 243–262. DOI: https://doi.org/10.1007/s00224-012-9396-1

[24] Rasmus Kyng, Jakub Pachocki, Richard Peng, and Sushant Sachdeva. 2017. A framework for analyzing resparsification algorithms. In *Proceedings of the 28th Symposium on Discrete Algorithms (SODA'17)*. 2032–2043. DOI: https://doi.org/10.1137/1.9781611974782.132

[25] Jakub Lacki and Piotr Sankowski. 2011. Min-cuts and shortest cycles in planar graphs in $O(n \log \log n)$ time. In *Proceedings of the 19th European Symposium on Algorithms (ESA'11)*. 155–166. DOI: https://doi.org/10.1007/978-3-642-23719-5_14

[26] Karl Menger. 1927. Zur allgemeinen kurventheorie. *Fundamenta Mathematicae* 1, 10, 96–115. DOI: https://doi.org/10.4064/fm-10-1-96-115

[27] Hiroshi Nagamochi and Toshihide Ibaraki. 1992. A linear-time algorithm for finding a sparse $k$-connected spanning subgraph of a $k$-connected graph. *Algorithmica* 7, 5–6, 583–596. DOI: https://doi.org/10.1007/BF01758778

[28] Hiroshi Nagamochi and Toshihide Ibaraki. 2008. *Algorithmic Aspects of Graph Connectivity*. Cambridge University Press, New York, NY. DOI: https://doi.org/10.1017/CBO9780511721649

[29] Danupon Nanongkai and Thatchaphol Saranurak. 2016. Dynamic cut oracle. Under submission.

[30] Johannes A. La Poutré. 2000. Maintenance of 2- and 3-edge-connected components of graphs II. *SIAM Journal on Computing* 29, 5, 1521–1549. DOI: https://doi.org/10.1137/S0097539793257770

[31] Daniel Dominic Sleator and Robert Endre Tarjan. 1983. A data structure for dynamic trees. *Journal of Computer and System Sciences* 26, 3, 362–391. DOI: https://doi.org/10.1016/0022-0000(83)90006-5

[32] Mikkel Thorup. 2007. Fully-dynamic min-cut. *Combinatorica* 27, 1, 91–127. DOI: https://doi.org/10.1007/s00493-007-0045-2

[33] Mikkel Thorup and David R. Karger. 2000. Dynamic graph algorithms with applications. In *Proceedings of the 7th Scandinavian Workshop on Algorithm Theory*. 1–9. DOI: https://doi.org/10.1007/3-540-44985-X_1

Received November 2016; revised September 2017; accepted December 2017