Incremental Exact Min-Cut in Poly-logarithmic Amortized Update Time*

Gramoz Goranci[1], Monika Henzinger[†2], and Mikkel Thorup[‡3]

1 **University of Vienna, Faculty of Computer Science, Vienna, Austria**
gramoz.goranci@univie.ac.at
2 **University of Vienna, Faculty of Computer Science, Vienna, Austria**
monika.henzinger@univie.ac.at
3 **Faculty of Computer Science, University of Copenhagen, Copenhagen, Denmark**
mikkel2thorup@gmail.com

Abstract

We present a deterministic incremental algorithm for *exactly* maintaining the size of a minimum cut with $\widetilde{O}(1)$ amortized time per edge insertion and $O(1)$ query time. This result partially answers an open question posed by Thorup [Combinatorica 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 recent sparsification technique of Kawarabayashi and Thorup [STOC 2015] and an exact incremental algorithm of Henzinger [J. of Algorithm 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 $\widetilde{O}(1)$ amortized update-time and constant query-time.

1998 ACM Subject Classification G.2.2 Graph Theory

Keywords and phrases Dynamic Graph Algorithms, Minimum Cut, Edge Connectivity

Digital Object Identifier 10.4230/LIPIcs.ESA.2016.46

1 Introduction

Computing a minimum cut of a graph is a fundamental algorithmic graph problem. While 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 last two decades. The motivation for studying the dynamic setting is apparent, as important networks such as 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

* This work was done in part while M. Henzinger and M. Thorup were visiting the Simons Institute for the Theory of Computing.
† The research leading to these results has 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.
‡ M. Thorup's research is partly supported by Advanced Grant DFF-0602-02499B from the Danish Council for Independent Research under the Sapere Aude research career programme.

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 paper, 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 size of a minimum cut of the current G.

For any $\alpha \geq 1$, we say that an algorithm is an α-approximation of λ 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 poly-logarithmic factors.

Related Work

For over a decade, the best known static and deterministic algorithm for computing a minimum cut was due to Gabow [10] which runs in $O(m + \lambda^2 \log n)$ time. Recently, Kawarabayashi and Thorup [19] devised a $\widetilde{O}(m)$ time algorithm which applies only to simple, unweighted and undirected graphs. Randomized Monte Carlo algorithms in the context of static minimum cut were initiated by Karger [17]. The best known randomized algorithm is due to Karger [18] and runs in $O(m \log^3 n)$ time.

Karger [16] 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 [28] improved upon the above guarantees by achieving an approximation factor of $\sqrt{2 + o(1)}$ and an $\widetilde{O}(1)$ expected amortized time per edge operation.

Henzinger [14] 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 poly-logarithmic size, Thorup [27] 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 sized 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.

Łącki and Sankwoski [21] 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 $\widetilde{O}(n^{5/6})$ query and update time.

There has been a growing interest in proving conditional lower bounds for dynamic problems in the last few years [1, 13]. A recent result of Nanongkai and Saranurak [24] 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.

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 $\widetilde{O}(1)$ 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 [27]. Meanwhile, it also stays in sharp contrast to the recent polynomial conditional lower-bound for the fully-dynamic weighted minimum cut problem [24].

We obtain our result by heavily relying on a recent sparsification technique developed in the context of static minimum cut. Specifically, for some given simple graph G, Kawarabayashi and Thorup [19] designed an $\widetilde{O}(m)$ procedure that contracts vertex sets of G and produces a multigraph H with considerably less vertices and edges while preserving some family of cuts of size up to $3/2\lambda(G)$. Motivated by the properties of H, we crucially observe that it is "safe" to escape from G and work entirely with graph H as long as the sequence of newly inserted edges have not increased the size of a minimum cut in H by more than $3/2\lambda(G)$. If the latter occurs, we then recompute a new multigraph H for the current graph G. Since $\lambda(G) \leq n$, we note the number of such re-computations can be at most $O(\log n)$. For maintaining the minimum-cut of H, we appeal to the exact incremental algorithm due to Henzinger [14]. Though the combination of this two algorithms might seem immediate at first sight, we remark that it is not alone sufficient for achieving the claimed bounds. Our main contribution is to overcome some technical obstacles and formally argue that such combination indeed leads to our desirable guarantees.

Motivated by the recent work on *space-efficient* dynamic algorithms [5, 12], we next 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 algorithms that can process a stream of edge insertions starting from an empty graph, and with high probability, the algorithm maintains an $(1+\varepsilon)$-approximation to the minimum cut in $O(\alpha(n) \log^3 n/\varepsilon^2)$ amortized update-time and constant query-time. Note that none of the existing streaming algorithms for $(1+\epsilon)$-approximate minimum cut [2, 20, 3] achieve these update and query times.

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-connected* if every pair of vertices is k-edge connected. The *local edge connectivity* $\lambda(G, x, y)$ 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 $S \subseteq V$, the *edge cut* $E(S, V \setminus S)$ is a set of edges that have one endpoint in S and the other in $V \setminus S$. If S is a singleton, we refer to such cut as *trivial* cut. Two vertices x and y are *separated* from $E(S, V \setminus S)$ if they do not belong to the same connected component induced by the edge cut. 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 is the minimum edge cut over all pairs of vertices. By Menger's Theorem [22], (a) the size of the minimum edge cut separating x and y is $\lambda(G, x, y)$, 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

λ and δ 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 from U and all edges between them are discarded. Note that this may not correspond to edge contractions, since we do not know whether U is connected. 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.

3 Sparse certificates

In this section we review a useful sparsification tool, introduced by Nagamochi and Ibaraki [23].

▶ **Definition 1** ([4]). 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)* 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 $G \setminus (F_1 \cup F_2 \ldots \cup F_{i-1})$. If we let $G' = (V, \bigcup_{i \leq k} F_i)$, then G' is a k-certificate. An msfd that fulfills the following additional properties is called a DA-msfd of order k: For a multigraph G, (1) for all $1 \leq i \leq k$, if x and y are connected in F_i, then they are i-edge connected in G; (2) G is k-edge connected iff G' is k-edge connected; (3) for any $1 \leq i \leq k$ and $x, y \in V$, $\lambda(\bigcup_{j \leq i} F_j, x, y) \geq \min(\lambda(G, x, y), i)$. As G' is a subgraph of G, $\lambda(G') \leq \lambda(G)$. This implies that $\lambda(G') = \min(k, \lambda(G))$. Nagamochi and Ibaraki [23] presented a $O(m + n)$ time algorithm to construct a DA-msfd, of order k.

4 Incremental Exact Minimum Cut

In this section we present a deterministic incremental algorithm that exactly maintains $\lambda(G)$. The algorithm has an $\widetilde{O}(1)$ update-time, an $O(1)$ query time and it applies to any undirected, unweighted, simple graph $G = (V, E)$. The result is obtained by carefully combining a recent result of Kawarabayashi and Thorup [19] on static min-cut and the incremental exact min-cut algorithm of Henzinger [14]. We start by describing the maintenance of non-trivial cuts, that is, cuts with at least two vertices on both sides.

Maintaining non-trivial cuts

Kawarabayashi and Thorup [19] devised a near-linear time algorithm that contracts vertex sets of a simple input graph G and produces a sparse multi-graph preserving all non-trivial minimum cuts of G. In the following theorem, we state a slightly generalized version of this algorithm.

▶ **Theorem 2** (KT-Sparsifier [19]). *Given an undirected, unweighted graph G with n vertices, m edges, and min-cut λ, in $\widetilde{O}(m)$ time, we can contract vertex sets and produce a multigraph H which consists of only $m_H = \widetilde{O}(m/\lambda)$ edges and $n_H = \widetilde{O}(n/\lambda)$ vertices, and which preserves all non-trivial minimum cuts along with the non-trivial cuts of size up to $3/2\lambda$ in G.*

As far as non-trivial cuts are concerned, the above theorem implies that it is safe to abandon G and work on H as long as the sequence of newly inserted edges satisfies $\lambda_H \leq 3/2\lambda$. To incrementally maintain the correct λ_H, we apply Henzinger's algorithm [14] on top of H. The basic idea to verify the correctness of the solution is to compute and store all min-cuts. 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. We next show how to do this efficiently.

To store all minimum edge cuts we use the *cactus tree* representation by Dinitz, Karzanov and Lomonosov [7]. 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. $\phi(x) = \phi(y)$ iff x and y are $(\lambda(G) + 1)$-edge connected.
3. Every minimum 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 λ is odd, every edge of E_c has weight λ and G_c is a tree. If λ is even, G_c consists of paths and simple cycles sharing at most one vertex, where edges that belong to a cycle have weight $\lambda/2$ while those not belonging to a cycle have weight λ.

Dinitz and Westbrook [8] showed that given a cactus tree, we can use the data structures from [11, 25] to maintain the cactus tree for minimum cut size λ under u insertions, reporting when the minimum cut size increases to $\lambda + 1$ in $O(u + n)$ total time.

To quickly compute and update the cactus tree representation of a given multigraph G, we use an algorithm due to Gabow [9]. The algorithm computes first a subgraph of G, called a *complete λ-intersection* or $I(G, \lambda)$, with at most λn edges, and uses $I(G, \lambda)$ to compute the cactus tree. Given some initial graph with m_0 edges, the algorithm computes $I(G, \lambda)$ and the cactus tree in $\widetilde{O}(m_0 + \lambda^2 n)$ time. Moreover, given $I(G, \lambda)$ and a sequence of edge insertions that increase the minimum cut by 1, the new $I(G, \lambda)$ and the new cactus tree can be computed in $\widetilde{O}(m')$, where m' is the number of edges in the current graph (this corresponds to one execution of Round Robin subroutine [10]).

Maintaining trivial cuts

We remark that the multigraph H from Theorem 2 preserves only non-trivial 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. If 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 $\textsc{Min}(\mathcal{H}_G)$ and $\textsc{UpdateEndpoints}(\mathcal{H}_G, e)$ can be implemented in $O(1)$ and $O(\log n)$ time, respectively (see [6]).

This leads to the following Algorithm 1.

Correctness

Let G be some 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 *correct* 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 correct even in the `Special Step` (i.e., when $\lambda_H > 3/2\lambda^*$), where H is not

Algorithm 1 Incremental Exact Minimum Cut

1: Compute the size λ_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 multigraph H by running KT-sparsifier on G and a mapping $h : V \to V_H$.
Compute the size λ_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 $N_h = \emptyset$.
while there is at least one minimum cut of size λ_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 multigraph H by running KT-sparsifier on the current graph G.
Update λ_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 $\bigcup_{i \leq \lambda_H + 1} F_i \cup N_h$ and call the resulting forests $F_1, \ldots, F_m$.
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 $I(H', \lambda_H)$ and a cactus tree of 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
endif
Goto 2.

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) H is correct as long as $\min\{\textsc{Min}(\mathcal{H}_G), \lambda_H\} \leq 3/2\lambda^*$ (and when this condition fails we rebuild H), and (3) as long as $\lambda_H \leq 3/2\lambda^*$, the algorithm correctly maintains all cuts of size up to $\lambda_H + 1$ of H.

Let N be the set of recently inserted edges in G that the algorithm maintains during the execution of the **while** loop in Step 2. Similarly, let N_h be the corresponding edge set in H.

▶ **Lemma 3.** *Let $H = (V_H, E_H)$ be a multigraph with minimum cut λ_H and let N_h be a set with $N_h \subseteq E_H$. Further, let $F_1, \ldots, F_m$ be a DA-msfd of order $m \geq \lambda_H + 1$ of $H \setminus N_h$, and let $H' = (V_H, E')$ be a graph with $E' = N_h \cup \bigcup_{i \leq \lambda_H + 1} F_i$. Then, a cut is a min-cut in H' iff it is a min-cut in H.*

Proof. We first show that every non-min cut in H is a non-min cut in H'. By contrapositive, we get that a min-cut in H' is a min-cut in H.

To this end, let $(S, V_H \setminus S)$ be a cut with $|E_H(S, V_H \setminus S)| \geq \lambda_H + 1$ in H. Define $E_H(S, V_H \setminus S) \cap N_h = S_{N_h}$ and $E_H(S, V_H \setminus S) \cap (E_H \setminus N_h) = S_{H \setminus N_h}$ such that $E_H(S, V_H \setminus S) = S_{N_h} \uplus S_{H \setminus N_h}$ and $|E_H(S, V_H \setminus S)| = |S_{N_h}| + |S_{H \setminus N_h}|$. Letting $F' = \bigcup_{i \leq \lambda_H + 1} F_i$, we similarly define edge sets S'_{N_h} and $S'_{F'}$ partitioning the edges $E'(S, V_H \setminus S)$ that cross the cut $(S, V_H \setminus S)$ in H'. First, observe that $|S_{N_h}| = |S'_{N_h}|$ since edges of N_h are always included in H'. In addition, by second property of Definition 1, we know that F' preserves all cuts of $H \setminus N_h$ up to size $\lambda_H + 1$. Thus, if $|S_{H \setminus N_h}| \leq \lambda_H + 1$, we get that $|E'(S, V_H \setminus S)| = |E_H(S, V_H \setminus S)| \geq \lambda_H + 1$. If $|S_{H \setminus N_h}| > \lambda_H + 1$, then F' must contain at least $\lambda_H + 1$ edges crossing such cut and thus $|S'_{F'}| \geq \lambda_H + 1$. The latter implies that $|E'(S, V_H \setminus S)| \geq \lambda_H + 1$. But H' being a subgraph of H implies that $\lambda(H') \leq \lambda_H$, thus $(S, V_H \setminus S)$ cannot be a min-cut in H'.

For other direction, let $(S, V_H \setminus S)$ be a min-cut in H. Since H' is a subgraph of H, we know that $|E'(S, V_H \setminus S)| \leq \lambda_H$. Therefore, showing that $|E'(S, V_H \setminus S)| \geq \lambda_H$ implies that $(S, V_H \setminus S)$ is also a min cut in H'. Fix x, y and consider a min-cut $(D, V_H \setminus D)$ of size $\lambda(H', x, y)$ separating x and y. Using the above notation and considering the cut $(D, V_H \setminus D)$ in H, we know that $|E_H(D, V_H \setminus D)| = |D_{N_h}| + |D_{H \setminus N_h}| \geq \lambda_H$. We first note that $|D_{N_h}| = |D'_{N_h}|$ since edges of N_h are always included in H'. Then, similarly as above, by second property of Definition 1 we know that if $|D_{H \setminus N_h}| \leq \lambda_H + 1$, then $|E'(D, V_H \setminus D)| = |E_H(D, V_H \setminus D)| \geq \lambda_H$. If $|D_{H \setminus N_H}| > \lambda_H + 1$, then F' must contain at least $\lambda_H + 1$ edges crossing such cut and thus $|E'(D, V_H \setminus D)| \geq \lambda_H + 1$. Combining both bounds we obtain that $\lambda(H', x, y) = |E'(D, V_H \setminus D)| \geq \lambda_H$. Since the later is valid for any x and y, we get that $\lambda(H') \geq \lambda_H$ must hold and in particular, $|E'(S, V_H \setminus S)| \geq \lambda_H$. ◀

▶ **Lemma 4.** *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 our notation, in the following we will refer to Step 1 as a `Full Rebuild Step` (namely the initial `Full Rebuild Step`).

▶ **Lemma 5.** *For some current graph G, let H be the multigraph obtained from G and assume that $\lambda_H \leq 3/2\lambda^*$, where λ^* denotes the value of min-cut at the last* `Full Rebuild Step`*. Then the algorithm correctly maintains $\lambda_H = \lambda(H)$.*

Proof. At the time of the last `Full Rebuild Step`, the algorithm applies KT-SPARSIFIER on G, which yields a multigraph H that preserves all non-trivial min-cuts of G. The value of

λ_H is updated to $\lambda(H)$ and a DA-msfd and a cactus tree are constructed for H. The latter preserve all cuts of H of size up to $\lambda_H + 1$. Thus, the value of λ_H is correct at this step.

Now, suppose that the graph after the last `Full Rebuild Step` has undergone a sequence of edge insertions, which resulted in the current graph G. During these insertions, as long as $\lambda_H \leq 3/2\lambda^*$, a sequence of k `Partial Rebuild Steps` is executed, for some $k \geq 1$. Let $\lambda_H^{(i)}$ be the value of λ_H after the i-th execution of `Partial Rebuild Step`, where $1 \leq i \leq k$. Since, $\lambda_H^{(k)} = \lambda(H)$, it suffices to show that $\lambda_H^{(k)}$ is correct. We proceed by induction.

For the base case, we show that $\lambda_H^{(1)}$ is correct. First, using the fact that λ_H and the cactus tree are correct at the last `Full Rebuild Step` and that the incremental cactus tree algorithm correctly tell us when to increment λ_H, we conclude that incrementing the value of λ_H in Step 2 is valid. Thus, $\lambda_H^{(1)}$ is correct. Next, in a `Partial Rebuild Step`, the algorithm sparsifies the graph while preserving all cuts of size up to $\lambda_H^{(1)} + 1$ and producing a new cactus tree for the next insertions. The correctness of the sparsification follows from Lemma 3.

For the induction step, let us assume that $\lambda_H^{(k-1)}$ is correct. Then, similarly to the base case, the correctness of $\lambda_H^{(k-1)}$, the cactus tree from the $(k-1)$-th `Partial Rebuild Step` and the correctness of the incremental cactus tree algorithm give that incrementing the value of $\lambda_H^{(k-1)}$ in Step 2 is valid and yields a correct $\lambda_H^{(k)}$. ◀

Note that when $\lambda_H > 3/2\lambda^*$, the above lemma is not guaranteed to hold. However, we will show below that this is not necessary for the correctness of the algorithm. The fact that we do not need to update the cactus tree in this setting is crucial for achieving our time bound.

▶ **Lemma 6.** *If* $\min\{\textsc{Min}(\mathcal{H}_G), \lambda_H\} \leq 3/2\lambda^*$*, then* H *is correct.*

Proof. Let C' be any non-trivial cut in G that is not in H. Such a cut must have cardinality strictly greater than $3/2\lambda^*$ since otherwise it would be contained in H. We show that C' 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 5 the algorithm maintains λ_H correctly. Since H is obtained from G by contracting vertex sets, there is a cut C in H, and thus in G, of value λ_H. It follows that C' cannot be a minimum cut of G since $|C'| > 3/2\lambda^* \geq \lambda_H = \lambda(H) \geq \lambda(G)$, 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 there is a cut of size $\textsc{Min}(\mathcal{H}_G) = \delta$ in G. Similarly, C' cannot be a minimum cut of G since $|C'| > 3/2\lambda^* \geq \delta \geq \lambda(G)$.

Appealing to the above cases, we conclude H is correct since a min-cut of G is either contained in H or it is a trivial cut of G. ◀

▶ **Lemma 7.** *The algorithm correctly maintains* $\lambda(G)$*, i.e.,* $\lambda(G) = \min\{\textsc{Min}(\mathcal{H}_G), \lambda_H\}$.

Proof. Let G be some current graph. If $\min\{\textsc{Min}(\mathcal{H}_G), \lambda_H\} \leq 3/2\lambda^*$, then by Lemma 6, H is correct. Thus, if $\lambda_H \leq 3/2\lambda^*$, then Lemma 5 ensures that λ_H is also maintained correctly and, hence, $\min\{\textsc{Min}(\mathcal{H}_G), \lambda_H\} = \lambda(G)$. 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 $\min\{\textsc{Min}(\mathcal{H}_G), \lambda_H\} = \textsc{Min}(\mathcal{H}_G)$. As the algorithm correctly maintains $\textsc{Min}(\mathcal{H}_G)$ at any time by Lemma 4, it follows that the algorithm maintains λ correctly in this case as well.

The only case that remains to consider is $\textsc{Min}(\mathcal{H}_G) > 3/2\lambda^*$ and $\lambda_H > 3/2\lambda^*$. But this implies that $\min\{\textsc{Min}(\mathcal{H}_G), \lambda_H\} > 3/2\lambda^*$, and the algorithm computes a H and $\lambda(G)$ from scratch and sets λ_H correctly. After this full rebuild $\lambda(G) = \min\{\textsc{Min}(\mathcal{H}_G), \lambda_H\}$ trivially holds. ◀

Running Time Analysis

▶ **Theorem 8.** *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 $\widetilde{O}(m_0 + m_1)$. If we start with an empty graph, the amortized time per edge insertion is $\widetilde{O}(1)$. The size of the minimum cut can be answered in constant time.*

Proof. We first analyse Step 1. Building the heap $\mathcal{H}_G$ and computing λ_0 take $O(n)$ and $\widetilde{O}(m_0)$ time, respectively. The total running time for constructing H, $I(H, \lambda_H)$ and the cactus tree is dominated by $\widetilde{O}(m_0 + \lambda_0^2 \cdot (n/\lambda_0)) = \widetilde{O}(m_0)$. Thus, the total time for Step 1 is $\widetilde{O}(m_0)$.

Let $\lambda_H^0, \ldots, \lambda_H^f$ be the values that λ_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 λ^* assumes during the algorithm. We define *Superphase j* to consist of the j-th `Full Rebuild Step` along with all steps executed while $\min\{\textsc{Min}(\mathcal{H}_G), \lambda_H\} \leq 3/2\lambda_j^*$, where λ_j^* is the value of $\lambda(G)$ at the `Full Rebuild Step`. Note that a superphase consists of a sequence of phases and potentially a final `Special Step`. Moreover, the algorithm runs a phase if $\lambda_H \leq 3/2\lambda^*$.

We say that λ_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:

▶ **Fact 9.** *Let j be a superphase during the execution of the algorithm. Then, we have*

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

Proof. From Step 3 we know that $n_j = \widetilde{O}(n/\lambda_j^*)$. 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 λ_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 $n_j = \widetilde{O}(n/\lambda_j^*) = \widetilde{O}(n/\lambda_H^i)$. ◀

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. The total time for Step 2 is $O(n_j + u_i \log n) = \widetilde{O}(n + u_i)$. Using Fact 9, we observe that $\bigcup_{i \leq \lambda_H + 1} F_i \cup N_h$ has size $O(u_{i-1} + \lambda_H^i n_j) = \widetilde{O}(u_{i-1} + n)$. Thus, the total time for computing DA-msfd in a `Partial Rebuild Step` is $\widetilde{O}(u_{i-1} + n)$. Similarly, since H' has $O(\lambda_H^i n_j) = \widetilde{O}(n)$ edges, it takes $\widetilde{O}(n)$ time to compute $I(H', \lambda_H^i)$ and the new cactus tree.

The total time spent in Phase i is $\widetilde{O}(u_{i-1} + u_i + n)$. Let λ and λ_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 Eqn. (1). This gives that the total work over all phases is

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

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`, the total running time for constructing H, $I(H, \lambda_j^*)$ and the cactus tree is dominated by $\widetilde{O}(m_0 + m_1 + (\lambda_j^*)^2 \cdot (n/\lambda_j^*)) = \widetilde{O}(m_0 + m_1)$. The running time of a Special Step is $\widetilde{O}(m_1)$.

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 λ^* 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 $\widetilde{O}(m_0 + m_1)$. ◀

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 $\widetilde{O}(1)$, resp. $O(1)$ query-time.

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 [14, 28, 27], where the space requirement is not considered. 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-certificate and incorporating it into the incremental exact min-cut algorithm due to Henzinger [14], as described in Section 4. Finally we apply the well-known randomized sparsification result due to Karger [17] to obtain our result.

Maintaining min-cut up to size k using $O(kn)$ space

We incrementally maintain a DA-msfd for an unweighted multigraph G using k union-find data structures $\mathcal{F}_1, \ldots, \mathcal{F}_k$ (see [6]). Each $\mathcal{F}_i$ maintains a spanning forest F_i of G. Recall that $F_1, \ldots, F_k$ are edge-disjoint. When a new edge $e = (u, v)$ is inserted into G, we define i to be the first index such that $\mathcal{F}_i.\textsc{Find}(u) \neq \mathcal{F}_i.\textsc{Find}(v)$. If we found such an i, we append the edge e to the forest F_i by setting $\mathcal{F}_i.\textsc{Union}(u, v)$ and return i. If such an i cannot be found after k steps, we simply discard edge e and return NULL. We refer to such procedure as k-$\textsc{Connectivity}(e)$.

It is easy to see that the forests maintained by k-$\textsc{Connectivity}(e)$ for every newly inserted edge e are indeed edge-disjoint. Combining this procedure with techniques from Henzinger [14] leads to the following Algorithm 2.

The space requirement of the above algorithm is only $O(kn)$, since we always maintain at most k spanning forests during its execution. The total running time for testing the k-connectivity of the endpoints of the newly inserted edges in Step 2 is $O(km\alpha(n))$, where

Algorithm 2 Incremental Exact Min-Cut up to size k

1: Set $\lambda = 0$, initialize k union-find data structures $\mathcal{F}_1, \ldots, \mathcal{F}_k$,
k empty forests $F_1, \ldots, F_k$, $I(\lambda)$, and an empty cactus tree.
2: **while** there is at least one minimum cut of size λ **do**
Receive the next operation.
if it is a query **then return** λ
else it is the insertion of an edge e, then
Set $i = k$-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 $E' = I(\lambda - 1) \cup \bigcup_{i \leq \lambda+1} F_i$.
Compute $I(\lambda)$ and a cactus tree of G'.
Goto 2.

$\alpha(n)$ stands for the inverse of Ackermann function. These guarantees combined with the arguments from Theorem 8 of Henzinger [14] give the following corollary.

▶ **Corollary 10.** *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 $\min\{\lambda(G), k\}$. The total time for inserting m edges is $O(km\alpha(n)\log n)$ and queries can be answered in constant time.*

Dealing with min-cuts of arbitrary size

We observe that Corollary 10 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 [17]. This allows us to get an $(1+\varepsilon)$-approximation to the value of min-cut with high probability.

▶ **Lemma 11** ([17]). *Let G be any graph with minimum cut λ and let $p \geq 10(\log n)/(\varepsilon^2\lambda)$. Let $S(p)$ be a subgraph of G obtained by including each of edge of G to $S(p)$ with probability p independently. Then the probability that the value of any cut of $S(p)$ has value more than $(1+\varepsilon)$ or less than $(1-\varepsilon)$ times its expected value is $O(1/n^3)$.*

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:

1. For $i = 0, \ldots, \lfloor \log n \rfloor$, let G_i be the initially empty sampled subgraphs.
2. If an edge e is inserted into G, include e to each G_i with probability $1/2^i$ and maintain the exact minimum cut of G_i up to size $k = 40 \log n/\varepsilon^2$ using Algorithm 2.
3. If the operation is a query, find the minimum j such that the min-cut of G_j is at most k. Return $2^j\lambda(G_j)$.

▶ **Theorem 12.** *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 the 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 $G^{(t)}$ and $p^{(t)} = 10(\log n)/(\varepsilon^2 \lambda^{(t)})$. For any integer $i \leq \lfloor \log_2 1/p^{(t)} \rfloor$, Lemma 11 implies that $2^i \lambda(G_i^{(t)})$ is an $(1 \pm \varepsilon)$-approximation to $\lambda(G^{(t)})$. Setting $i = \lfloor \log_2 1/p^{(t)} \rfloor$, we get that:

$$\mathbb{E}[\lambda(G_i^{(t)})] \leq \lambda(G^{(t)})/2^i \leq 2p^{(t)} \lambda(G^{(t)}) \leq 20 \log n/\varepsilon^2.$$

The later along with Lemma 11 imply that for any $\varepsilon \in (0, 1)$, the size of the minimum cut in $G_i^{(t)}$ is at most $(1+\varepsilon)20 \log n/\varepsilon^2 \leq 40 \log n/\varepsilon^2$ with probability $1 - O(1/n^3)$. Thus, $j \leq \lfloor \log_2 1/p^{(t)} \rfloor$ and the algorithm returns a $(1 \pm \varepsilon)$-approximation to the minimum cut of $G^{(t)}$ with probability $1 - O(1/n^3)$. 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 $(1 \pm \varepsilon) \leq 1 + O(\varepsilon)$-approximation is at most $O(1/n)$.

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 10). 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 each edge is given some random weight p_e chosen uniformly from $[0, 1]$. We call the resulting weighted graph G^w. 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 [15]:

▶ **Lemma 13.** *Let $k = 40 \log n/\varepsilon^2$. 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 the 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, $G(p)$ can be viewed as taking G and including each edge with probability p. The claim follows from Lemma 11. ◀

For any graph G and some appropriate weight p, the above 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 minimum edge-disjoint spanning forests (using those edge weights). We note that such a collection is a DA-msfd of order k for G with $O(kn)$ edges and by Lemma 3, 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 maintain an insertion-only data-structure that maintains a collection of k minimum edge-disjoint spanning forests $S_1, \ldots, S_k$ with respect to the edge weights. Let $S = \bigcup_{i=1}^{k} S_i$. 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 [26]. 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 S_i. Let max-weight$(S_i(u, v))$ 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 S_i and no edge was evicted from S_i. Otherwise, we check whether $p_e >$ max-weight$(S_i(e))$. 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 S_i or not, the second value that is returned gives an edge that does not (or no longer) belong to S_i. Note that each edge insertion requires $O(\log n)$ time. We refer to this insert operation as Insert-MSF(S_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 empty minimum spanning forests.
- Insert-NI(e, p_e): Set $i = 1$, $e' = e$, taken = FALSE.
  while $((i \leq k)$ and $e' \neq$ NULL$)$ **do**
  Set $(t'$, $e'') =$ Insert-MSF$(S_i, e', p_{e'})$.
  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).

Recall that $S = \bigcup_{i \leq k} S_i$. We use the abbreviation NI-Sparsifier(k) to refer to this data-structure. By slight abuse of notation we will associate a weight with each edge in S and use S^w to refer to this weighted version of S.

▶ **Lemma 14.** *For $k > 0$ and any graph G,* NI-Sparsifier(k) *maintains a weighted* DA-msfd of order k of G under edge insertions. The algorithm uses $O(kn)$ space and the total time for inserting m edges is $O(km \log n)$.

(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 into Algorithm 2.
- Query-Limited$()$: returns λ
- Initialize-Limited(G, k): builds a data structure for G with parameter k by calling Insert-Limited(e) for each edge e in G.

We use the abbreviation Lim(k) to refer to such data-structure.

Combining the above data-structures leads to the following algorithm:

Correctness and Running Time Analysis

Let S denote the unweighted version of S^w. Throughout the execution of Algorithm 3, S corresponds exactly to the DA-msfd of order k of G maintained by NI-Sparsifier(k). In

Algorithm 3 $(1+\varepsilon)$-Min-Cut with $O(n \log n/\varepsilon^2)$ Space

```
1: Set k = 40 log n/ε^2.
   Set p = 10 log n/ε^2.
   Let H and S^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 ≤ p then Insert-Limited(e) endif
      Set (taken, e') = Insert-NI(e, p_e).
          if taken then
            Insert e into S^w with weight p_e.
            if (e' ≠ NULL) then remove e' from S^w.
          endif
      endif
   endwhile
3: // Rebuild Step
   Set p = p/2.
   Let H be the unweighted subgraph of S^w consisting of all edges of weight at most p.
   Goto 2.
```

the following, let H be the graph that is given as input to Lim(k). Thus, by Corollary 10, Query-Limited() returns $\min\{k, \lambda(H)\}$, i.e., it returns $\lambda(H)$ as long as $\lambda(H) \leq k$. We now formally prove the correctness.

▶ **Lemma 15.** *Let $\epsilon \leq 1$. If $\lambda(G) < k$, then $H = G$, $p = k/4$, and* Query-Limited() *returns $\lambda(G)$. The first rebuild step is triggered after the first insertion that increases $\lambda(G)$ to k and let $\lambda(G) = \lambda(H) = k$ at that time.*

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 ϵ. 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 $\lambda(G) < k$, 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 $S(p)$ be the unweighted subgraph of S^w that contains all edge of weight at most p.

▶ **Lemma 16.** *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 $S^{old}(p)$ be $S(p)$ right after the last rebuild. Then the graph $H = S^{old}(p) \cup N_h(p)$.*

▶ **Lemma 17.** *At the time of a rebuild $S(p)$ is a DA-msfd of order k of $G(p)$.*

By Lemma 13, in order 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 $\lambda(H) = \lambda(G(p))$.

▶ **Lemma 18.** *Let $\varepsilon \leq 1$. If $\lambda(G) \geq k$, then (1) $p \geq k/(4\lambda(G))$ w.h.p. and (2) $\lambda(H) = \lambda(G(p))$.*

Proof. For any $i \geq 0$, after the i-th rebuild we have $p = p^{(i)} := 10\log n/(2^i\varepsilon^2)$. We will show by induction on i that (1) $p^{(i)} = 10\log n/(2^i\varepsilon^2) \geq 10\log n/(\varepsilon^2\lambda(G))$ with high probability, 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, $\lambda(H) = \lambda(G(p^{(i-1)}))$.

We first analyse $i = 1$. Assume that the insertion of edge e caused the first rebuild. Lemma 15 showed that (1) at the first rebuild $\lambda(G) = k \geq 2^1 = 2$ and (2) that up to the first rebuild $G(p) = G = H$.

For the induction step ($i > 1$), we inductively assume that (1) at the $(i-1)$-st rebuild, $p^{(i-1)} \geq 10\log n/\varepsilon^2\lambda(G^{\text{old}})$ with high probability, where G^{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 $G^{\text{new}} = G^{\text{old}} \cup \{e\}$. Note that w.h.p. $p^{(i-1)} \geq 10\log n/(\varepsilon^2\lambda(G^{\text{old}})) \geq 10\log n/(\varepsilon^2\lambda(G^{\text{new}}))$ as $\lambda(G^{\text{old}}) \leq \lambda(G^{\text{new}})$. Thus, by Lemma 13, we get that $\lambda(G^{\text{new}}(p^{(i-1)}))/p^{(i-1)} \leq (1+\varepsilon)\lambda(G^{\text{new}})$ with high probability.

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

$$\begin{aligned}\lambda(G^{\text{new}}(p^{(i-1)})) = k = 40\log n/\varepsilon^2 &\leq (1+\varepsilon)\lambda(G^{\text{new}})\cdot p^{(i-1)}\\ &= (1+\varepsilon)\lambda(G^{\text{new}})\cdot 10\log n/(2^{i-1}\varepsilon^2),\end{aligned}$$

w.h.p.. This in turn implies that $\lambda(G^{\text{new}}) \geq 2^{i+1}/(1+\varepsilon) \geq 2^i$ w.h.p. by our choice of ε.

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}, resp. S^{i-1}, resp. H^{i-1}, be the graph G, resp. S, resp. H, right after rebuild $i-1$ and let N_h be the set of edges inserted since, i.e., $G = G^{(i-1)} \cup N_h$. As we showed in Lemma 16, $H = S^{i-1}(p^{(i-1)}) \cup N_h(p^{(i-1)})$. Thus, $H^{i-1} = S^{i-1}(p^{(i-1)})$. Additionally, by Lemma 17, $S^{i-1}(p^{(i-1)})$ is a DA-msfd of order k of $G^{i-1}(p^{(i-1)})$. Thus by Property (3) of a DA-msfd of order k, for every cut $(A, V\setminus A)$ of value at most k in H^{i-1}, $\lambda(H^{i-1}, A) = \lambda(S^{i-1}(p^{(i-1)}), A) = \lambda(G^{i-1}(p^{(i-1)}), A)$, where $\lambda(G, A)$ denotes the number of edges crossing from A to $V\setminus A$ in G. Now assume by contradiction that $\lambda(G(p^{(i-1)})) > \lambda(H)$ and consider a minimum cut $(A, V\setminus A)$ in H, i.e., $\lambda(H) = \lambda(H, A)$. We know that at any time $k \geq \lambda(H)$. Thus $k \geq \lambda(H) = \lambda(H, A)$, which implies $k \geq \lambda(H^{i-1}, A)$. By Property (3) of DA-msfd it follows that $\lambda(H^{i-1}, A) = \lambda(G^{i-1}(p^{(i-1)}), A)$. Note that $H = H^{i-1} \cup N_h(p^{(i-1)})$ and $G(p^{(i-1)}) = G^{i-1}(p^{(i-1)}) \cup E_H(p^{(i-1)})$. Let x be the number of edges of $N_h(p^{(i-1)})$ that cross the cut $(A, V\setminus A)$. Then $\lambda(H) = \lambda(H, A) = \lambda(H^{i-1}, A) + x = \lambda(G^{i-1}(p^{(i-1)}), A) + x = \lambda(G(p^{(i-1)}), A)$, 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. Thus, by the above lemma and an union bound over these $O(n^2)$ different graphs, we get that throughout its execution, our algorithm maintains a $(1+\varepsilon)$-approximation to the min cut with high probability.

▶ **Theorem 19.** *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 the min-cut of G with high probability. The total time for insertiong 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, S^w, Lim(k), and NI-Sparsifier (k), each of size at most $O(n \log n/\varepsilon^2)$ (Corollary 10 and Lemma 14).

When Algorithm 3 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 14, 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. In Step 2, Initialize-Limited(H, k) takes at most $O(m\alpha(n)\log^2 n/\varepsilon^2)$ total time (Corollary 10). 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 Steps 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 S^w in a 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 above bounds gives the theorem. ◀

References

1 Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *Proc. of the 55th FOCS*, pages 434–443. IEEE, 2014.

2 Kook Jin Ahn and Sudipto Guha. Graph sparsification in the semi-streaming model. In *Proc. of the 36th ICALP*, pages 328–338, 2009.

3 Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Graph sketches: sparsification, spanners, and subgraphs. In *Proc. of the 32nd PODS*, pages 5–14, 2012.

4 András A. Benczúr and David R. Karger. Randomized approximation schemes for cuts and flows in capacitated graphs. *SIAM J. Comput.*, 44(2):290–319, 2015.

5 Sayan Bhattacharya, Monika Henzinger, Danupon Nanongkai, and Charalampos E. Tsourakakis. Space- and time-efficient algorithm for maintaining dense subgraphs on one-pass dynamic streams. In *Proc. of the 47th STOC*, pages 173–182, 2015.

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

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

8 Yefim Dinitz and Jeffery Westbrook. Maintaining the classes of 4-edge-connectivity in a graph on-line. *Algorithmica*, 20(3):242–276, 1998.

9 Harold N. Gabow. Applications of a poset representation to edge connectivity and graph rigidity. In *Proc. of the 32nd FOCS*, pages 812–821, 1991.

10 Harold N. Gabow. A matroid approach to finding edge connectivity and packing arborescences. *J. Comput. Syst. Sci.*, 50(2):259–273, 1995.

11 Zvi Galil and Giuseppe F. Italiano. Maintaining the 3-edge-connected components of a graph on-line. *SIAM J. Comput.*, 22(1):11–28, 1993.

12 David Gibb, Bruce M. Kapron, Valerie King, and Nolan Thorn. Dynamic graph connectivity with improved worst case update time and sublinear space. *CoRR*, abs/1509.06464, 2015.

13 Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *Proc. of the 47th STOC*, pages 21–30, 2015.

14 Monika Rauch Henzinger. A static 2-approximation algorithm for vertex connectivity and incremental approximation algorithms for edge and vertex connectivity. *Journal of Algorithms*, 24(1):194–220, 1997.

15 David Karger. *Random Sampling in Graph Optimization Problems.* PhD thesis, Stanford University, Stanford, 1994.

16 David R. Karger. Using randomized sparsification to approximate minimum cuts. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms. 23-25 January 1994, Arlington, Virginia.*, pages 424–432, 1994.

17 David R. Karger. Random sampling in cut, flow, and network design problems. *Math. Oper. Res.*, 24(2):383–413, 1999.

18 David R. Karger. Minimum cuts in near-linear time. *J. ACM*, 47(1):46–76, 2000.

19 Ken-ichi Kawarabayashi and Mikkel Thorup. Deterministic global minimum cut of a simple graph in near-linear time. In *Proc. of the 47th STOC*, pages 665–674, 2015.

20 Jonathan A. Kelner and Alex Levin. Spectral sparsification in the semi-streaming setting. *Theory Comput. Syst.*, 53(2):243–262, 2013.

21 Jakub Lacki and Piotr Sankowski. Min-cuts and shortest cycles in planar graphs in $O(n \log \log n)$ time. In *Proc. of the 19th ESA*, pages 155–166, 2011.

22 Karl Menger. Zur allgemeinen kurventheorie. *Fundamenta Mathematicae*, 1(10):96–115, 1927.

23 Hiroshi Nagamochi and Toshihide Ibaraki. A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph. *Algorithmica*, 7(5&6):583–596, 1992.

24 Danupon Nanongkai and Thatchaphol Saranurak. Dynamic cut oracle. under submission, 2016.

25 Johannes A. La Poutré. Maintenance of 2- and 3-edge-connected components of graphs II. *SIAM J. Comput.*, 29(5):1521–1549, 2000.

26 Daniel Dominic Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983.

27 Mikkel Thorup. Fully-dynamic min-cut. *Combinatorica*, 27(1):91–127, 2007.

28 Mikkel Thorup and David R Karger. Dynamic graph algorithms with applications. In *Algorithm Theory-SWAT 2000*, pages 1–9. Springer, 2000.