Efficient Contractions of Dynamic Graphs – With Applications

Monika Henzinger
Institute of Science and Technology, Klosterneuburg, Austria

Evangelos Kosinas
Institute of Science and Technology, Klosterneuburg, Austria

Robin Münk
Technical University of Munich, Germany

Harald Räcke
Technical University of Munich, Germany

Abstract

A non-trivial minimum cut (NMC) sparsifier is a multigraph $\hat{G}$ that preserves all non-trivial minimum cuts of a given undirected graph G. We introduce a flexible data structure for fully dynamic graphs that can efficiently provide an NMC sparsifier upon request at any point during the sequence of updates. We employ simple dynamic forest data structures to achieve a fast from-scratch construction of the sparsifier at query time. Based on the strength of the adversary and desired type of time bounds, the data structure comes with different guarantees. Specifically, let G be a fully dynamic simple graph with n vertices and minimum degree δ. Then our data structure supports an insertion/deletion of an edge to/from G in $n^{o(1)}$ worst-case time. Furthermore, upon request, it can return w.h.p. an NMC sparsifier of G that has $O(n/\delta)$ vertices and $O(n)$ edges, in $\hat{O}(n)$ time. The probabilistic guarantees hold against an adaptive adversary. Alternatively, the update and query times can be improved to $\tilde{O}(1)$ and $\tilde{O}(n)$ respectively, if amortized-time guarantees are sufficient, or if the adversary is oblivious. Throughout the paper, we use $\tilde{O}$ to hide polylogarithmic factors and $\hat{O}$ to hide subpolynomial (i.e., $n^{o(1)}$) factors.

We discuss two applications of our new data structure. First, it can be used to efficiently report a cactus representation of all minimum cuts of a fully dynamic simple graph. Building this cactus for the NMC sparsifier instead of the original graph allows for a construction time that is sublinear in the number of edges. Against an adaptive adversary, we can with high probability output the cactus representation in worst-case $\hat{O}(n)$ time. Second, our data structure allows us to efficiently compute the maximal k-edge-connected subgraphs of undirected simple graphs, by repeatedly applying a minimum cut algorithm on the NMC sparsifier. Specifically, we can compute with high probability the maximal k-edge-connected subgraphs of a simple graph with n vertices and m edges in $\tilde{O}(m + n^2/k)$ time. This improves the best known time bounds for $k = \Omega(n^{1/8})$ and naturally extends to the case of fully dynamic graphs.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms; Theory of computation → Dynamic graph algorithms

Keywords and phrases Graph Algorithms, Cut Sparsifiers, Dynamic Algorithms

Digital Object Identifier 10.4230/LIPIcs.ESA.2025.36

Related Version *Full Version*: `https://arxiv.org/abs/2509.05157`

Funding *Monika Henzinger and Evangelos Kosinas*: This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (MoDynStruct, No. 101019564) and the Austrian Science Fund (FWF) grant DOI 10.55776/Z422 and grant DOI 10.55776/I5982.
Harald Räcke and Robin Münk: This project has received funding from the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 498605858.

1 Introduction

Graph sparsification is an algorithmic technique that replaces an input graph G by another graph $\hat{G}$, which has fewer edges and/or vertices than G, but preserves (or approximately preserves) a desired graph property. Specifically, for connectivity-based and flow-based problems, a variety of static sparsifiers exist, that approximately maintain cut- or flow-values in G [2, 3, 4, 8, 20, 26, 27].

Let n denote the number of vertices, m the number of edges, and δ the minimum degree of the input graph. In an undirected simple graph it is possible to reduce the number of vertices to $O(n/\delta)$ and the number of edges to $O(n)$ both in randomized and deterministic time $\tilde{O}(m)$ while preserving the value of all non-trivial minimum cuts *exactly* [10, 19, 21]. A minimum cut is considered *trivial* if one of its sides consists of a single vertex and we call the resulting multigraph a *non-trivial minimum cut sparsifier (NMC sparsifier)*.

Most sparsification algorithms assume a static graph. Maintaining an NMC sparsifier in a fully dynamic setting, where a sequence of edge insertions and deletions can be arbitrarily interleaved with requests to output the NMC sparsifier, was only recently studied: Goranci, Henzinger, Nanongkai, Saranurak, Thorup, and Wulff-Nilsen [11] show how to maintain an NMC sparsifier in a fully dynamic graph w.h.p. in $\tilde{O}(n)$ worst-case update and query time (as a consequence of Theorem 3.7 in [11]), under the assumption of an oblivious adversary. Additionally, Theorem 4.5 in [11] gives a deterministic algorithm that outputs an NMC sparsifier of size $\tilde{O}(m/\delta)$ in $\tilde{O}(m/\delta)$ worst-case query time with $\tilde{O}(\delta^3)$ amortized update time.

1.1 Our Results

In this paper, we present the first data structure for providing an NMC sparsifier of a fully dynamic graph that supports sublinear *worst-case* update and query time and works against an *adaptive* adversary. As a first application, we give an improved fully dynamic algorithm that outputs a cactus representation of *all* minimum cuts of the current graph upon request. Additionally, we use our data structure to compute the maximal k-edge connected subgraphs of an undirected simple graph with an improvement in running time for large values of k.

In more detail, we provide a data structure for a fully dynamic graph that can be updated in *worst-case* time $\hat{O}(1)$ and that allows (at any point during the sequence of edge updates) to construct an NMC sparsifier in worst-case time $\hat{O}(n)$. The probabilistic guarantees work against an *adaptive* adversary. If the update time is relaxed to be *amortized* or if the adversary is *oblivious*, the update time can be improved to $\tilde{O}(1)$ and the query time to $\tilde{O}(n)$.

Our basic approach is to maintain suitable data structures in a dynamically changing graph that allow the execution of the *static* NMC sparsifier algorithm based on random 2-out contractions proposed by Ghaffari, Nowicki, and Thorup [10] in time $\hat{O}(n)$ instead of $\tilde{O}(m)$. Our main insight is that this speedup can be achieved by maintaining

1. a dynamic spanning forest data structure (DSF) of the input graph G and
2. a dynamic cut set data structure (DCS), where the user determines which edges belong to a (not necessarily spanning) forest F of G, and, given a vertex v, the data structure returns an edge that leaves the tree of F that contains v (if it exists).

We show that these two data structures suffice to construct an NMC sparsifier of the current graph in the desired time bound. Put differently, we can avoid processing all edges of G in order to build the NMC sparsifier, and only spend time that is roughly proportional to the size of the sparsifier.

Note that the NMC sparsifier is computed from scratch every time it is requested and no information of a previously computed sparsifier is maintained throughout the dynamic updates of the underlying graph. This ensures the probabilistic guarantees hold against an adaptive adversary if the guarantees of the chosen DSF and DCS data structures do.

Our main result is the following theorem.

▶ **Theorem 1.** *Let G be a fully dynamic simple graph that currently has n nodes and minimum degree $\delta > 0$. There is a data structure that outputs an NMC sparsifier of G that has $O(n/\delta)$ vertices and $O(n)$ edges w.h.p. upon request. Each update and query takes either*

1. *worst-case $\hat{O}(1)$ and $\hat{O}(n)$ time respectively w.h.p., assuming an* adaptive *adversary, or*
2. *amortized $\tilde{O}(1)$ and $\tilde{O}(n)$ time respectively w.h.p., assuming an* adaptive *adversary, or*
3. *worst-case $\tilde{O}(1)$ and $\tilde{O}(n)$ time respectively w.h.p., assuming an* oblivious *adversary.*

Recall that $\tilde{O}$ hides polylogarithmic factors and $\hat{O}$ hides subpolynomial (i.e., $n^{o(1)}$) factors. The three cases of Theorem 1 result from using different data structures to realize our required DSF and DCS data structures. We show that with minimal overhead both data structures can be reduced to a dynamic minimum spanning forest data structure, for which many constructions have been proposed in the literature. For Case 1, we make use of the fully dynamic minimum spanning forest algorithm of Chuzhoy, Gao, Li, Nanongkai, Peng, and Saranurak [6] – the only spanning forest data structure known so far that can provide worst-case time guarantees against an adaptive adversary. Case 2 is the result of substituting the deterministic data structure of Holm, de Lichtenberg, and Thorup [15] and case 3 results from using the randomized data structure of Kapron, King, and Mountjoy [16] instead.

As a first application we can efficiently provide a cactus representation of the minimum cuts of a simple graph upon request in the fully dynamic setting. The cactus representation of all minimum cuts of a static graph is known to be computable in near linear time $\tilde{O}(m)$ using randomized algorithms [12, 18, 21]. With deterministic algorithms, the best known time bound is $m^{1+o(1)}$ [12]. We are not aware of any previous work on providing the cactus for fully dynamic graphs in sublinear time per query. Specifically, we show the following:

▶ **Theorem 2.** *Let G be a fully dynamic simple graph with n vertices. There is a data structure that w.h.p. reports a cactus representation of all minimum cuts of G. Each update and query takes either*

1. *worst-case $\hat{O}(1)$ and $\hat{O}(n)$ time respectively w.h.p., assuming an* adaptive *adversary, or*
2. *amortized $\tilde{O}(1)$ and $\tilde{O}(n)$ time respectively w.h.p., assuming an* adaptive *adversary, or*
3. *worst-case $\tilde{O}(1)$ and $\tilde{O}(n)$ time respectively w.h.p., assuming an* oblivious *adversary.*

Theorem 2 provides an improvement over one of the main technical components in [11]. Specifically, Goranci et al. [11], provide a method to efficiently maintain an NMC sparsifier of a dynamic simple graph, based on the 2-out contractions of Ghaffari et al. [10]. However, their main technical result (Theorem 3.7 in [11]) has several drawbacks. First, it needs to know an estimated upper bound $\hat{\delta}$ on the minimum degree of any graph that occurs during the sequence of updates of G. Second, they try to *maintain* the sparsifier during the updates. This results in an update time $\tilde{O}(\hat{\delta})$, and forces them to "hide" their sparsifier from an adversary, i.e., they can expose their sparsifier only if they work against an oblivious adversary. Thus, to make their minimum cut algorithm work against an adaptive adversary, they return only the *value* of the minimum cut. In contrast, our algorithm computes a sparsifier from scratch upon request, and can therefore provide a cactus representation of *all* minimum cuts, even against an adaptive adversary.

Notice that we provide a different trade-off in reporting the minimum cut: We have an update time of $\hat{O}(1)$ and a query time of $\hat{O}(n)$, whereas Theorem 1.1 of Goranci et al. [11] has an update time of $\tilde{O}(n)$ and query time $O(1)$. This trade-off is never worse (modulo

Table 1 Best known time bounds for computing the maximal k-edge-connected subgraphs in undirected graphs in the static setting. The $\tilde{O}$ expression hides polylogarithmic factors.

Algorithm	Time	Type	Range of k
Chechik et al., Forster et al. [5, 7]	$\tilde{O}(m + k^{O(k)}n^{3/2})$	Det.	$k \in \mathbb{N}$
Forster et al. [7]	$\tilde{O}(m + k^3 n^{3/2})$	Las Vegas Rnd.	$k \in \mathbb{N}$
Henzinger et al. [13]	$\tilde{O}(n^2)$	Det.	$k \in \mathbb{N}$
Thorup, Georgiadis et al. [28, 9]	$\tilde{O}(m + k^8 n^{3/2})$	Det.	$k = \log^{O(1)} n$
Saranurak and Yuan [25]	$O(m + kn^{1+o(1)})$	Det.	$k = \log^{o(1)} n$
Nalam and Saranurak [24]	$\tilde{O}(m \cdot \min\{m^{3/4}, n^{4/5}\})$	Monte Carlo Rnd.	$k \in \mathbb{N}$
This paper	$\tilde{O}(m + n^2/k)$	Monte Carlo Rnd.	$k \in \mathbb{N}$

the subpolynomial factors) if one caches the result of a query and answers queries from the cache if the graph did not change. Furthermore, upon request, we can provide a minimum cut explicitly, and not just its value.

As a second application of our main result, we improve the time bounds for computing the (vertex sets of the) maximal k-edge-connected subgraphs in a simple undirected graph. Specifically, we have the following:

▶ **Theorem 3.** *Let G be a simple graph with n vertices and m edges, and let k be a positive integer. We can compute the maximal k-edge-connected subgraphs of G w.h.p. in $\tilde{O}(m+n^2/k)$ time w.h.p.*

For comparison, Table 1 gives an overview of the best known algorithms for computing the maximal k-edge-connected subgraphs in undirected graphs. Thorup [28] does not deal with the computation of the maximal k-edge-connected subgraphs, but the result is a consequence of his fully dynamic minimum cut algorithm as observed in [9]. The algorithm in [24] is the only one that holds for weighted graphs (with integer weights), and it has no dependency on k in the time bound. Notice that our algorithm improves over all prior results for $k = \Omega(n^{1/8})$ and $m = \Omega(n^{8/7})$.

With a reduction to simple graphs, this implies the following bounds for computing the maximal k-edge-connected subgraphs of multigraphs.

▶ **Corollary 4.** *Let G be a multigraph with n vertices and m edges, and let k be a positive integer. We can compute the maximal k-edge-connected subgraphs of G w.h.p. in $\tilde{O}(m + k^2n + kn^2)$ time w.h.p.*

Notice that Corollary 4 provides an improvement compared to the other algorithms in Table 1, depending on k and the density of the graph. For example, if δ and ϵ are two parameters satisfying $1/4 \leq \delta \leq 1$, $16/15 \leq \epsilon \leq 2$ and $\delta \leq \epsilon - 6/5$, then we have an improvement in the regime where $m = \Theta(n^\epsilon)$ and $k = \Theta(n^\delta)$ against all previous algorithms in Table 1 that work for multigraphs (i.e., except for Henzinger et al. [13], which works only for simple graphs).

Finally, the method that we use in Theorem 3 for computing the maximal k-edge-connected subgraphs in static graphs extends to the case of dynamic graphs.

▶ **Theorem 5.** *Let G be a fully dynamic simple graph with n vertices. There is a data structure that can provide the maximal k-edge-connected subgraphs of G at any point in time, with high probability, for any integer $k < n$. Each update and query takes either*

1. *worst-case* $\hat{O}(1)$ *and* $\hat{O}(n^2/k)$ *time respectively w.h.p, assuming an* adaptive *adversary,*
2. *amortized* $\tilde{O}(1)$ *and* $\tilde{O}(n^2/k)$ *time respectively w.h.p, assuming an* adaptive *adversary, or*
3. *worst-case* $\tilde{O}(1)$ *and* $\tilde{O}(n^2/k)$ *time respectively w.h.p, assuming an* oblivious *adversary.*

The results by Aamand et al. [1] and Saranurak and Yuan [25] provide algorithms for maintaining the maximal k-edge-connected subgraphs in *decremental* graphs. Georgiadis et al. [9] provide a fully dynamic algorithm that given two vertices determines whether they belong to the same k-edge connected subgraph in time $O(1)$. Their worst-case update time is $\tilde{O}(T(n,k))$, where $T(n,k)$ is the running time of any algorithm for static graphs that is used internally by the algorithm (see values in the time column of Table 1 with the additive term m removed). Thus, as in the case for dynamic minimum cut, our algorithm provides a different trade-off, with fast updates and slower query time. Notice that Theorem 5 provides an improvement over [9] (for any function $T(n,k)$ corresponding to an algorithm from Table 1) when k is sufficiently large (i.e., when $k = \Omega(n^{1/8})$).

Due to the space constraints, more details on related work, all omitted proofs, and some further details can be found in the full version of this paper.

2 Preliminaries

In this paper, we consider only undirected, unweighted graphs. A graph is called *simple* if it contains no parallel edges, conversely it is a multigraph if it does. We use common graph terminology and notation that can be found e.g. in [23]. Let $G = (V, E)$ be a graph. Throughout, we use n and m to denote the number of vertices and edges of G, respectively. We use $V(G)$ and $E(G)$ to denote the set of vertices and edges, respectively, of G; that is, $V = V(G)$ and $E = E(G)$. A *subgraph* of G is a graph of the form (V', E'), where $V' \subseteq V$ and $E' \subseteq E$. A *spanning* subgraph of G is a subgraph of the form (V, E') that contains at least one incident edge to every vertex of G that has degree > 0. If X is a subset of vertices of G, we let $G[X]$ denote the induced subgraph of G on the vertex set X. Thus, $V(G[X]) := X$, and the edge set of $G[X]$ is $\{e \in E \mid \text{ both endpoints of } e \text{ are in } X\}$. If E' is a set of edges of G, we let $G[E'] := (V, E')$.

A connected component of G is a maximal connected subgraph of G. A set of edges of G whose removal increases the number of connected components of G is called a *cut* of G. The size $|C|$ is called the *value* of the cut. A cut C with minimum value in a connected graph G is called a *minimum cut* of G. In this case, $G \setminus C = (V, E \setminus C)$ consists of two connected components. If one of them is a single vertex, then C is called a *trivial minimum cut*.

An *NMC sparsifier* of G is a multigraph H on the same vertex set as G that preserves all non-trivial minimum cuts of G, in the sense that the number of edges leaving any non-trivial minimum cut is the same in H as it is in G.

Contractions of graphs. Let $E' \subseteq E$ be a set of edges of a graph $G = (V, E)$, and let $C_1, \ldots, C_t$ be the connected components of the graph $G' = (V, E')$. The *contraction* $\hat{G}$ induced by E' is the graph derived from G by contracting each C_i, for $i \in \{1, \ldots, t\}$, into a single node. We ignore possible self-loops, but we maintain distinct edges of G that have their endpoints in different connected components of G'. Hence, $\hat{G}$ may be a multigraph even though G is a simple graph. Furthermore, there is a natural injective correspondence from the edges of $\hat{G}$ to the edges of G. We say that an edge of G is *preserved* in $\hat{G}$ if its endpoints are in different connected components of G' (it corresponds to an edge of $\hat{G}$).

A *random 2-out contraction* of G is a contraction of G that is induced by sampling from every vertex v of G two edges incident to it (independently, with repetition allowed) and setting E' to be the edges sampled in this way. Thus, E' satisfies $|E'| \leq 2|V(G)|$. The importance of considering 2-out contractions is demonstrated in the following theorem of Ghaffari et al. [10].

▶ **Theorem 6** (rephrased weaker version of Theorem 2.3 in [10]). *A random 2-out contraction of a simple graph G with n vertices and minimum degree δ has $O(n/\delta)$ vertices, with high probability, and preserves any fixed non-trivial minimum cut of G with constant probability.*

Preserving small cuts via forest decompositions. Nagamochi and Ibaraki [22] have shown the existence of a sparse subgraph that maintains all cuts of the original graph with value up to k, where k is an integer ≥ 1. Specifically, given a graph $G = (V, E)$ with n vertices and m edges, there is a spanning subgraph $H = (V, E')$ of G with $|E'| \leq k(n-1)$ such that every set of edges $C \subseteq E$ with $|C| < k$ is a cut of G if and only if C is a cut of H. A graph H with this property is given by a *k-forest decomposition* of G, which is defined as follows. First, we let F_1 be a spanning forest of G. Then, for every $i \in \{2, \ldots, k\}$, we recursively let F_i be a spanning forest of $G \setminus (F_1 \cup \cdots \cup F_{i-1})$. Then we simply take the union of the forests $H = F_1 \cup \cdots \cup F_k$. A naive implementation of this idea takes time $O(k(n+m))$. However, this construction can be completed in linear time by computing a maximum adjacency ordering of G [23].

Maximal k-edge-connected subgraphs. Given a graph G and a positive integer k, a *maximal k-edge-connected subgraph* of G is a subgraph of the form $G[S]$, where: (1) $G[S]$ is connected, (2) the minimum cuts of $G[S]$ have value at least k, and (3) S is maximal with this property. The first two properties can be summarized by saying that $G[S]$ is k-edge-connected, which equivalently means that we have to remove at least k edges in order to disconnect $G[S]$. It is easy to see that the vertex sets $S_1, \ldots, S_t$ that induce the maximal k-edge-connected subgraphs of G form a partition of V.

A note on probabilistic guarantees. Throughout the paper we use the abbreviation w.h.p. (with high probability) to mean a probability of success at least $1 - O(\frac{1}{n^c})$, where c is a fixed constant chosen by the user and n denotes the number of vertices of the graph. The choice of c only affects the values of two parameters q and r that are used internally by the algorithm of Ghaffari et al. [10], which our result is based on. Note that the specification "w.h.p" may be applied either to the running time or to the correctness (or to both).

3 Outline of Our Approach

Our main contribution is a new data structure that outputs an NMC sparsifier of a fully dynamic graph upon request. The idea is to compute the NMC sparsifier from scratch every time it is requested. For this we adapt the construction by Ghaffari et al. [10] to compute a random 2-out contraction of the graph at the current time. We can achieve a speedup of the construction time by maintaining just two data structures for dynamic forests throughout the updates of the graph.

3.1 Updates: Data Structures for Dynamic Forests

As our fully dynamic graph G changes, we rely on efficient data structures for the following two problems, which we call the "dynamic spanning forest" problem (DSF) and the "dynamic cutset" problem (DCS). In DSF, the goal is to maintain a spanning forest F of a dynamic graph G. Specifically, the DSF data structure supports the following operations.

- `insert`(e). Inserts a new edge e to G. If e has endpoints in different trees of F, then it is automatically included in F, and this event is reported to the user.
- `delete`(e). Deletes the edge e from G. If e was a tree edge in F, then it is also removed from F, and a new replacement edge is selected among the remaining edges of G in order to reconnect the trees of F that got disconnected. If a replacement edge is found, it automatically becomes a tree edge in F, and it is output to the user.

The DCS problem is a more demanding variant of DSF. Here the goal is to maintain a (not necessarily spanning) forest F of the graph, but we must also be able to provide an edge that connects two different trees if needed. Specifically, the DCS data structure supports the following operations.

- $\texttt{insert}_G(e)$. Inserts a new edge e to G.
- $\texttt{insert}_F(e)$. Inserts an edge e to F, if e is an edge of G that has endpoints in different trees of F. Otherwise, F stays the same.
- $\texttt{delete}_G(e)$. Deletes the edge e from G. If e is a tree edge in F, it is also deleted from F.
- $\texttt{delete}_F(e)$. Deletes the edge e from F (or reports that e is not an edge of F).
- `find_cutedge`(v). Returns an edge of G (if it exists) that has one endpoint in the tree of F that contains v, and the other endpoint outside of that tree.

The main difference between DSF and DCS is that in DCS the user has absolute control over the edges of the forest F. In both problems, the most challenging part is finding an edge that connects two distinct trees of F. In DCS this becomes more difficult because the data structure must work with the forest that the user has created, whereas in DSF the spanning forest is managed internally by the data structure. Both of these problems can be solved by reducing them to the dynamic minimum spanning forest (MSF) problem, as shown in the following Lemma 7.

▶ **Lemma 7.** *The DSF problem can be reduced to the DCS problem within the same time bounds. The DCS problem can be reduced to dynamic MSF with an additive $O(\log n)$ overhead for every operation.*

To realize these two data structures deterministically with worst-case time guarantees, the only available solution at the moment is to make use of the reduction to the dynamic MSF problem given in Lemma 7 and then employ the dynamic MSF data structure of Chuzhoy et. al. [6], which supports every update operation in worst-case $\hat{O}(1)$ time. Alternatively, one can solve both DSF and DCS deterministically with *amortized* time guarantees using the data structures of Holm et. al. [15]. In that case, every update for DSF can be performed in amortized $O(\log^2 n)$ time, and every operation for DCS can be performed in amortized $O(\log^4 n)$ time, by reduction to dynamic MSF. If one is willing to allow randomness, one can solve both DSF and DCS in worst-case polylogarithmic time per operation, under the assumption of an oblivious adversary with the data structure by Kapron et al. [16].

These different choices for realizing the dynamic MSF data structure give the following lemma.

▶ **Lemma 8.** *The DSF and DCS data structures can be realized in either*
1. *deterministic worst-case* $\hat{O}(1)$ *update time, or*
2. *deterministic amortized* $\tilde{O}(1)$ *update time, or*
3. *worst-case* $\tilde{O}(1)$ *update time, assuming an* oblivious *adversary.*

To handle the updates on G, any insertion or deletion of an edge is also applied in both the DSF and the DCS data structure. In particular, for DCS we only use the $\texttt{insert}_G$ and $\texttt{delete}_G$ operations and keep its forest empty. At query time, we will build the DCS forest from scratch and then fully delete it again. In order for this to be efficient, DCS needs to have already processed all edges of G.

3.2 Queries: Efficiently Contracting a Dynamic Graph

The NMC sparsifier that we output for each query is a random 2-out contraction of the graph at the current time. For this construction, we use our maintained forest data structures and build upon the algorithm of Ghaffari et al. [10], who prove the following.

▶ **Theorem 9** (Weaker version of Theorem 2.1 of Ghaffari et al. [10])**.** *Let G be a simple graph with n vertices, m edges, and minimum degree δ. In $O(m \log n)$ time we can create a contraction $\hat{G}$ of G that has $O(n/\delta)$ vertices and $O(n)$ edges w.h.p., and preserves all non-trivial minimum cuts of G w.h.p.*

Note that in particular, the contracted graph $\hat{G}$ created by the above theorem is an NMC sparsifier, as desired. We first sketch the algorithm that yields Theorem 9 as described by Ghaffari et al. [10]. Then we outline how we can adapt and improve this construction for dynamic graphs, making use of the DSF and DCS data structures. Specifically, to answer a query we show that we can build $\hat{G}$ in time proportional to the size of $\hat{G}$, which may be much lower than $O(m \log n)$. The details and a formal analysis can be found in Section 4.

Random 2-out Contractions of Static Graphs

Ghaffari et al.'s algorithm [10] works as follows (see also Algorithm 1). First, it creates a collection of $q = O(\log n)$ random 2-out contractions $G_1, \ldots, G_q$ of G, where every G_i is created with independent random variables. Now, according to Theorem 2.4 in [10], each G_i for $i \in \{1, \ldots, q\}$, has $O(n/\delta)$ vertices w.h.p., and preserves any fixed non-trivial minimum cut of G with constant probability.

In a second step, they compute a $(\delta + 1)$-forest decomposition $\hat{G}_i$ of G_i, for every $i \in \{1, \ldots, q\}$, in order to ensure that $\hat{G}_i$ has $O(\delta \cdot (n/\delta)) = O(n)$ edges w.h.p. Because G has minimum degree δ, every non-trivial minimum cut has value at most δ. Hence, each $\hat{G}_i$ still maintains every fixed non-trivial minimum cut with constant probability.

Finally, they select a subset of edges $E_{\text{con}} \subseteq E(G)$ to contract by a careful *"voting"* process. Specifically, for every edge e of G, they check if it is an edge of at least r graphs from $\hat{G}_1, \ldots, \hat{G}_q$, where r is a carefully chosen parameter. If e does not satisfy this property, then it is included in E_{con}, the set of edges to be contracted. In the end, $\hat{G}$ is given by contracting all edges from E_{con}.

An Improved Construction using Dynamic Forests

We now give an overview of our algorithm for efficiently constructing the NMC sparsifier $\hat{G}$. It crucially relies on having the DSF and DCS data structures initialized to contain the edges of the current graph G. For a fully dynamic graph, this is naturally ensured at query

Algorithm 1 Construction of the contracted graph $\hat{G}$ in Theorem 9.

```
input: a simple graph G in adjacency list representation
choose parameters q, r = Θ(log n) according to [10]
compute the minimum degree δ of G
for i ← 1 to q do
    construct a 2-out contraction G_i of G
foreach i ∈ {1, ..., q} do
    construct a (δ + 1)-forest decomposition Ĝ_i of G_i
let E_con ← ∅ // a set of edges of G to contract
foreach edge e of G do
    if e is preserved in less than r graphs from Ĝ_1, ..., Ĝ_q then
        E_con ← E_con ∪ {e}
return the graph obtained from G by contracting the edges in E_con
```

time by maintaining the two forest data structures throughout the updates, as described in Section 3.1. Since the goal is to output a graph $\hat{G}$ with $O(n/\delta)$ vertices and $O(n)$ edges w.h.p., we aim for an algorithm that takes roughly $O(n)$ time. The process is broken up into three parts.

Part 1. First, we compute the 2-out contractions $G_1, \ldots, G_q$ for $q = O(\log n)$. Each G_i can be computed by sampling, for every vertex v of G, two random edges incident to v (independently, with repetition allowed). Since the graph G is dynamic, the adjacency lists of the vertices also change dynamically, and therefore this is not an entirely trivial problem. However, in the full version of this paper we provide an efficient solution that relies on elementary data structures. Specifically, we can support every graph update in worst-case constant time, and every query for a random incident edge to a vertex in worst-case $\tilde{O}(1)$ time. Notice that each G_i, for $i \in \{1, \ldots, q\}$, is given by contracting a set E_i of $O(n)$ edges of G.

Part 2. Next, we separately compute a $(\delta + 1)$-forest decomposition $\hat{G}_i$ of G_i, for every $i \in \{1, \ldots, q\}$. Each $\hat{G}_i$ is computed by only accessing the edges of E_i plus the edges of the output (which are $O(n)$ w.h.p.). For this, we rely on the DCS data structure. Since in DCS we have complete control over the maintained forest F, we can construct it in such a way, that every connected component of $G[E_i]$ induces a subtree of F. Notice that the connected components of $G[E_i]$ correspond to the vertices of G_i. This process of modifying F takes $O(|E_i|) = O(n)$ update operations on the DCS data structure. Then, we have established the property that the tree edges that connect vertices of different connected components of $G[E_i]$ correspond to a spanning tree of G_i. Afterwards, we just repeatedly remove the spanning tree edges, and find replacements using the DCS data structure. These replacements constitute a new spanning forest of G_i. Thus, we just have to repeat this process $\delta + 1$ times to construct the desired $(\delta + 1)$-forest decomposition. Note that every graph $\hat{G}_i$ is constructed in time roughly proportional to its size (which is $O(n)$ w.h.p.). Any overhead comes from the use of the DCS data structure.

Part 3. Finally, we construct the graph $\hat{G}$ by contracting the edges $E_{<r}$ of G that appear in less than r graphs from $\hat{G}_1, \ldots, \hat{G}_q$. From Ghaffari et al. ([10], Theorem 2.4) it follows that $|E \setminus E_{<r}| = |E_{\geq r}| = O(qn) = O(n \log n)$. In the following we provide an algorithm for constructing $\hat{G}$ with only $O(n + |E_{\geq r}|)$ operations of the DSF data structure.

We now rely on the spanning forest F of G that is maintained by the DSF data structure. We pick the edges of F one by one, and check for every edge whether it is contained in $E_{<r}$ (it is easy to check for membership in $E_{<r}$ in $O(\log n)$ time). If it is not contained in $E_{<r}$, then we store it as a *candidate edge*, i.e., an edge that possibly is in $\hat{G}$. In this case, we also remove it from F, and the DSF data structure will attempt to fix the spanning forest by finding a replacement edge. Otherwise, if $e \in E_{<r}$, then we "fix" it in F, and do not process it again.

In the end all "fixed" edges of F form a spanning forest of the connected components of $G[E_{<r}]$. Note that the algorithm makes continuous progress: in each step it either identifies an edge of $E_{\geq r}$, or it "fixes" an edge of F (which can happen at most $n-1$ times). Thus, it performs $O(n + |E_{\geq r}|) = \tilde{O}(n)$ DSF operations. Since we have arrived at a spanning forest of $G[E_{<r}]$, we can build $\hat{G}$ by contracting the candidate edges stored during this process.

4 Analysis

Our main technical tools are Propositions 10 and 11. Proposition 10 allows us to create a forest decomposition of a contracted graph without accessing all the edges of the original graph, but roughly only those that are included in the output, and those that we had to contract in order to get the contracted graph. Proposition 11 has a somewhat reverse guarantee: it allows us to create a contracted graph by accessing only the edges that are preserved during the contraction (plus only a small number of additional edges).

In this section we assume that G is a fully dynamic graph that currently has n vertices. We also assume that we have maintained a DCS and a DSF data structure on G, which support every operation in U_{CS} and U_{SF} time, respectively (cf. Section 3.1). The proofs of all theorems and propositions in this section can be found in the full version of this paper.

4.1 A k-Forest-Decomposition of the Contraction

Given a set of edges to contract, the following proposition shows how to compute a k forest decomposition of the induced contracted graph. Crucially, the contracted graph H does not have to be given explicitly, but it is sufficient to know only a set of edges whose contraction yields H. Thus, the running time of the construction is independent of the number of edges of H. The whole procedure is shown in Algorithm 2.

▶ **Proposition 10.** *Let H be a contraction of G that results from contracting a given set E_{con} of edges in G. Let further k be a positive integer and n_H be the number of vertices in H. Then we can construct a k-forest decomposition of H in time $O\big((n + kn_H) \cdot U_{\text{CS}} + |E_{con}| \log n\big)$.*

Note that the number of DCS operations depends only on (1) the number n of vertices of G, (2) the number n_H of vertices of H, and (3) the number k of forests that we want to build.

4.2 Building the Contracted Graph

A contracted graph can naturally be computed from its defining set of contraction edges E_{con} in time proportional to the size of this set $|E_{\text{con}}|$. Recall that in our case E_{con} is the result of the "voting" procedure across all generated δ-forest decompositions (cf. Section 3.2), which is rather expensive to compute. We hence use a different construction that does not need to know E_{con} explicitly. Instead, it relies on an efficient oracle to certify that a given edge is not contained in E_{con}.

Algorithm 2 Compute a k-forest decomposition of the graph that is formed by contracting a set of edges E' of G.

```
let F be the empty DCS forest
foreach edge e ∈ E' do
    if the endpoints of e belong to different trees of F then
        DCS.insert_F(e)
let V be a set that consists of one vertex from every tree of F
set S ← ∅
for i ← 1 to k do
    set L ← ∅ // L will contain the edges of the current spanning forest
    foreach v ∈ V do
        let e ← DCS.find_cutedge(v)
        while e ≠ ⊥ do
            DCS.insert_F(e), and append e to L
            e ← DCS.find_cutedge(v)
    foreach e ∈ L do
        DCS.delete_G(e), and append e to S
use DCS.delete_F to remove all the edges from F
foreach e ∈ S do
    DCS.insert_G(e) // restore DCS to its original state
return S
```

▶ **Proposition 11.** *Let H be a contraction of G that results from contracting a set E_{con} of edges in G and let E_{pre} be the set of edges of G that are preserved in H. Suppose that there is a set of edges E_{check} with $E_{pre} \subseteq E_{check}$ and $E_{check} \cap E_{con} = \emptyset$, for which we can check membership in time μ. Then we can construct H in time $O(n\mu + |E_{check}| \cdot (U_{\mathrm{SF}} + \mu))$.*

Note that the number of DSF operations is proportional to the number of edges in E_{check} (the set E_{con} is not required as input to the algorithm). Thus, this algorithm becomes more efficient if only few edges are contained in $E(G) \setminus E_{\mathrm{con}}$ (since $E_{\mathrm{check}} \subseteq E(G) \setminus E_{\mathrm{con}}$). In our application, we will have $\mu = O(\log n)$.

4.3 Constructing an NMC sparsifier

We can now state our result for computing an NMC sparsifier of a simple graph using Propositions 10 and 11. Compare this to Theorem 9 and note how Theorem 12 requires initialized DSF and DCS data structures but has a running time that is independent of the number of edges in G. An outline of this procedure is given in Section 3.2.

▶ **Theorem 12.** *Let G be a simple graph with n vertices that has minimum degree $\delta > 0$ and is maintained in a DSF and a DCS data structure. Then, with high probability we can construct an NMC sparsifier of G that has $O(n/\delta)$ vertices and $O(n)$ edges. W.h.p this takes $\tilde{O}(n \cdot (U_{\mathrm{SF}} + U_{\mathrm{CS}}))$ time.*

4.4 Fully Dynamic Graphs

The result of Theorem 12 can easily be extended to fully dynamic simple graphs by maintaining the DSF and DCS data structures throughout the updates of the graph. These forest data structures can be realized in different ways, as described in Section 3.1. Depending on this choice we get a different result, and this is how we derive Theorem 1.

If G is disconnected, all minimum cuts have value 0 and an NMC sparsifier H is by definition only required to have no edges between different connected components of G. Crucially, this implies that there is no guarantee that any information of the minimum cuts within each connected component of G is preserved in H. In this case, however, we can easily strengthen the result by applying Theorem 12 to each connected component of G individually.

5 Applications

5.1 A Cactus Representation of All Minimum cuts in Dynamic Graphs

It is well-known that a graph with n vertices has $O(n^2)$ minimum cuts, all of which can be represented compactly with a data structure of $O(n)$ size that has the form of a cactus graph (for more details, see [23]). As a first immediate application of our main Theorem 1, we show how the NMC sparsifier $\hat{G}$ of any fully dynamic simple graph G can be used to quickly compute this cactus representation. This result is summarized in Theorem 2; see the full version of this paper for a detailed construction.

To obtain just a single minimum cut of G, one can apply the deterministic minimum cut algorithms of [14, 19] on $\hat{G}$, which yields a minimum cut C of $\hat{G}$ in time $\tilde{O}(|\hat{G}|) = \tilde{O}(n)$. To transform C into a minimum cut of G, we compare its size with the minimum degree δ of any node in G: If $|C| \leq \delta$, then we get a minimum cut of G by simply mapping the edges from C back to the corresponding edges of G. Otherwise, a minimum cut of G is given by any vertex of G that has degree δ (which is easy to maintain throughout the updates).

5.2 Computing the Maximal k-Edge-Connected Subgraphs

The data structure of Theorem 1 can also be used to improve the time bounds for computing the maximal k-edge-connected subgraphs of a simple graph, in for cases where k is a sufficiently large polynomial of n. Specifically, we get an improvement for $k = \Omega(n^{1/8})$, c.f. Section 1.

A general strategy for computing these subgraphs is the following. Let G be a simple graph with n vertices and m edges, and let k be a positive integer. The basic idea is to repeatedly find and remove cuts with value less than k from G. First, as long as there are vertices with degree less than k, we repeatedly remove them from the graph. Now we are left with a (possibly disconnected) graph where every non-trivial connected component has minimum degree at least k. If we perform a minimum cut computation on a non-trivial connected component S of G, there are two possibilities: either the minimum cut is at least k, or we will have a minimum cut C with value less than k. In the first case, S is confirmed to be a maximal k-edge-connected subgraph of G. In the second case, we remove C from S, and thereby split it into two connected components S_1 and S_2. Then we recurse on both S_1 and S_2. Since the graph is simple and S has minimum degree at least k, it is a crucial observation that both S_1 and S_2 contain at least k vertices (see e.g. [19]). Therefore the number of nodes decreases by at least k with every iteration and hence the total recursion depth is $O(n/k)$.

The minimum cut computation takes time $T_{\text{mc}} = \tilde{O}(m)$ [17], hence the worst-case running time of this approach is $\Theta(n/k \cdot T_{\text{mc}}) = \tilde{O}(mn/k)$. We can use Theorem 1 to bring the time down to $\tilde{O}(m + n^2/k)$ w.h.p.

▶ **Theorem 3.** *Let G be a simple graph with n vertices and m edges, and let k be a positive integer. We can compute the maximal k-edge-connected subgraphs of G w.h.p. in $\tilde{O}(m+n^2/k)$ time w.h.p.*

Through a reduction to simple graphs, Theorem 3 implies Corollary 4, which is a similar result with slightly worse time bounds for undirected *multigraphs*. Finally, the method that establishes Theorem 3 naturally extends to the case of fully dynamic simple graphs, which yields Theorem 5.

References

1 Anders Aamand, Adam Karczmarz, Jakub Lacki, Nikos Parotsidis, Peter M. R. Rasmussen, and Mikkel Thorup. Optimal decremental connectivity in non-sparse graphs. In Kousha Etessami, Uriel Feige, and Gabriele Puppis, editors, *50th International Colloquium on Automata, Languages, and Programming, ICALP 2023, July 10-14, 2023, Paderborn, Germany*, volume 261 of *LIPIcs*, pages 6:1–6:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.ICALP.2023.6.

2 Joshua Batson, Daniel A. Spielman, and Nikhil Srivastava. Twice-ramanujan sparsifiers. *SIAM Journal on Computing*, 41(6):1704–1721, 2012. doi:10.1137/090772873.

3 András A. Benczúr and David R. Karger. Approximating s-t minimum cuts in $\tilde{O}(n^2)$ time. In Gary L. Miller, editor, *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 47–55. ACM, 1996. doi:10.1145/237814.237827.

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. doi:10.1137/070705970.

5 Shiri Chechik, Thomas Dueholm Hansen, Giuseppe F. Italiano, Veronika Loitzenbauer, and Nikos Parotsidis. Faster algorithms for computing maximal 2-connected subgraphs in sparse directed graphs. In Philip N. Klein, editor, *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 1900–1918. SIAM, 2017. doi:10.1137/1.9781611974782.124.

6 Julia Chuzhoy, Yu Gao, Jason Li, Danupon Nanongkai, Richard Peng, and Thatchaphol Saranurak. A deterministic algorithm for balanced cut with applications to dynamic connectivity, flows, and beyond. In Sandy Irani, editor, *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 1158–1167. IEEE, 2020. doi:10.1109/FOCS46700.2020.00111.

7 Sebastian Forster, Danupon Nanongkai, Liu Yang, Thatchaphol Saranurak, and Sorrachai Yingchareonthawornchai. Computing and testing small connectivity in near-linear time and queries via fast local cut algorithms. In Shuchi Chawla, editor, *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 2046–2065. SIAM, 2020. doi:10.1137/1.9781611975994.126.

8 Wai Shing Fung, Ramesh Hariharan, Nicholas J. A. Harvey, and Debmalya Panigrahi. A general framework for graph sparsification. *SIAM J. Comput.*, 48(4):1196–1223, 2019. doi:10.1137/16M1091666.

9 Loukas Georgiadis, Giuseppe F. Italiano, Evangelos Kosinas, and Debasish Pattanayak. On maximal 3-edge-connected subgraphs of undirected graphs. *CoRR*, abs/2211.06521, 2022. doi:10.48550/arXiv.2211.06521.

10 Mohsen Ghaffari, Krzysztof Nowicki, and Mikkel Thorup. Faster algorithms for edge connectivity via random 2-out contractions. In Shuchi Chawla, editor, *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 1260–1279. SIAM, 2020. doi:10.1137/1.9781611975994.77.

11 Gramoz Goranci, Monika Henzinger, Danupon Nanongkai, Thatchaphol Saranurak, Mikkel Thorup, and Christian Wulff-Nilsen. Fully dynamic exact edge connectivity in sublinear time. In Nikhil Bansal and Viswanath Nagarajan, editors, *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms, SODA 2023, Florence, Italy, January 22-25, 2023*, pages 70–86. SIAM, 2023. doi:10.1137/1.9781611977554.ch3.

12 Zhongtian He, Shang-En Huang, and Thatchaphol Saranurak. Cactus representation of minimum cuts: Derandomize and speed up. In David P. Woodruff, editor, *Proceedings of the 2024 ACM-SIAM Symposium on Discrete Algorithms, SODA 2024, Alexandria, VA, USA, January 7-10, 2024*, pages 1503–1541. SIAM, 2024. doi:10.1137/1.9781611977912.61.

13 Monika Henzinger, Sebastian Krinninger, and Veronika Loitzenbauer. Finding 2-edge and 2-vertex strongly connected components in quadratic time. In Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann, editors, *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part I*, volume 9134 of *Lecture Notes in Computer Science*, pages 713–724. Springer, 2015. doi:10.1007/978-3-662-47672-7_58.

14 Monika Henzinger, Satish Rao, and Di Wang. Local flow partitioning for faster edge connectivity. *SIAM J. Comput.*, 49(1):1–36, 2020. doi:10.1137/18M1180335.

15 Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, 2001. doi:10.1145/502090.502095.

16 Bruce M. Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In Sanjeev Khanna, editor, *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 1131–1142. SIAM, 2013. doi:10.1137/1.9781611973105.81.

17 David R. Karger. Minimum cuts in near-linear time. *J. ACM*, 47(1):46–76, 2000. doi:10.1145/331605.331608.

18 David R. Karger and Debmalya Panigrahi. A near-linear time algorithm for constructing a cactus representation of minimum cuts. In Claire Mathieu, editor, *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2009, New York, NY, USA, January 4-6, 2009*, pages 246–255. SIAM, 2009. doi:10.1137/1.9781611973068.28.

19 Ken-ichi Kawarabayashi and Mikkel Thorup. Deterministic edge connectivity in near-linear time. *J. ACM*, 66(1):4:1–4:50, 2019. doi:10.1145/3274663.

20 Yin Tat Lee and He Sun. Constructing linear-sized spectral sparsification in almost-linear time. *SIAM Journal on Computing*, 47(6):2315–2336, 2018. doi:10.1137/16M1061850.

21 On-Hei Solomon Lo, Jens M. Schmidt, and Mikkel Thorup. Compact cactus representations of all non-trivial min-cuts. *Discret. Appl. Math.*, 303:296–304, 2021. doi:10.1016/J.DAM.2020.03.046.

22 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. doi:10.1007/BF01758778.

23 Hiroshi Nagamochi and Toshihide Ibaraki. *Algorithmic Aspects of Graph Connectivity*, volume 123 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 2008. doi:10.1017/CBO9780511721649.

24 Chaitanya Nalam and Thatchaphol Saranurak. Maximal k-edge-connected subgraphs in weighted graphs via local random contraction. In Nikhil Bansal and Viswanath Nagarajan, editors, *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms, SODA 2023, Florence, Italy, January 22-25, 2023*, pages 183–211. SIAM, 2023. doi:10.1137/1.9781611977554.ch8.

25 Thatchaphol Saranurak and Wuwei Yuan. Maximal k-edge-connected subgraphs in almost-linear time for small k. In Inge Li Gørtz, Martin Farach-Colton, Simon J. Puglisi, and Grzegorz Herman, editors, *31st Annual European Symposium on Algorithms, ESA 2023, September 4-6, 2023, Amsterdam, The Netherlands*, volume 274 of *LIPIcs*, pages 92:1–92:9. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.ESA.2023.92.

26 Daniel A. Spielman and Nikhil Srivastava. Graph sparsification by effective resistances. *SIAM Journal on Computing*, 40(6):1913–1926, 2011. doi:10.1137/080734029.

27 Daniel A. Spielman and Shang-Hua Teng. Spectral sparsification of graphs. *SIAM Journal on Computing*, 40(4):981–1025, 2011. doi:10.1137/08074489X.

28 Mikkel Thorup. Fully-dynamic min-cut. *Comb.*, 27(1):91–127, 2007. doi:10.1007/s00493-007-0045-2.