Practical Fully Dynamic Minimum Cut Algorithms[*]

Monika Henzinger[†] Alexander Noe [‡] Christian Schulz [§]

Abstract

We present a practically efficient algorithm for maintaining a global minimum cut in large dynamic graphs under both edge insertions and deletions. While there has been theoretical work on this problem, our algorithm is the first implementation of a fully-dynamic algorithm. The algorithm uses the theoretical foundation and combines it with efficient and finely-tuned implementations to give an algorithm that can maintain the global minimum cut of a graph with rapid update times. We show that our algorithm gives up to multiple orders of magnitude speedup compared to static approaches both on edge insertions and deletions.

1 Introduction

We consider the problem of maintaining a (global) *minimum cut* of a graph under *edge insertions and deletions*, also known as the *fully-dynamic minimum cut problem.* A *minimum cut* in an edge-weighted graph is a partition of the vertices into two sets so that the total weight of edges connecting the sets is minimized. In the fully dynamic setting, the algorithm has to process a sequence of edge insertions and deletions and has to be able to return a minimum cut at any point in this sequence.

The minimum cut problem has applications in many fields, such as network reliability [27, 47], VLSI design [33], graph drawing [25], as a subproblem in the branch-and-cut algorithm for solving the travelling salesperson problem and other combinatorial problems [46], and as a subproblem in connectivity-based data reductions for problems such as cluster editing [3]. Most real-world networks are continuously changing and evolving [7, 10, 52] and thus, dynamic algorithms that maintain a solution for a changing graph are of utmost importance for large-scale graph applications.

There has been a large body of research for the static minimum cut problem starting in 1961 [15]. The randomized algorithm of Karger [26] with a running time of $\mathcal{O}(m\log^3 n)$ is the first algorithm with a quasi-linear running time. Kawarabayashi and Thorup [32] give the first deterministic quasi-linear algorithm, later improved by Henzinger et al. [22] to a running time of $\mathcal{O}(m\log^2 n\log\log^2 n)$, which is the fastest deterministic minimum cut algorithm for unweighted simple graphs. Nagamochi et al. [40, 44] give an algorithm for the minimum cut problem, which is based on edge contractions instead of maximum flows. Their algorithm has a worst case running time of $\mathcal{O}(nm + n^2\log n)$ but performs far better in practice on many graph classes [4, 24, 20]. Gawrychowski et al. [12] give a randomized algorithm that finds a minimum cut in an undirected weighted graph G with high probability in $\mathcal{O}(m\log^2 n)$ time, which is currently the fastest asymptotic running time for the static minimum cut problem. Mukhopadhyay and Nanongkai [39] give a randomized algorithm with running time $\mathcal{O}\left(m\frac{\log^2 n}{\log\log n} + n\log^6 n\right)$. Li and Panigrahi [37] give a deterministic algorithm that runs in time $\mathcal{O}(m^{1+\epsilon})$ plus polylog(n) maximum flow computations for any constant $\epsilon > 0$. Recently, Li [36] gave a deterministic algorithm with running time $\mathcal{O}(m^{1+\epsilon})$ for any constant $\epsilon > 0$.

In the field of dynamic graph algorithms, Henzinger [23] gives the first incremental minimum cut algorithm, which maintains the exact minimum cut with an amortized time of $\mathcal{O}(\lambda\log n)$ per edge insertion, where λ is the value of the minimum cut. Goranci et al. [16] manage to remove the dependence on λ from the update time and give an incremental algorithm with $\mathcal{O}(\log^3 n\log\log^2 n)$ amortized time per edge insertion. Both algorithms maintain a compact data structure of all minimum cuts called *cactus graph* and invalidate minimum cuts whose weight was increased due to an edge insertion. If there are no remaining minimum cuts, they recompute all minimum cuts from scratch. For minimum cut values up to polylogarithmic size, Thorup [51] gives a fully dynamic algorithm with $\tilde{\mathcal{O}}(\sqrt{n})$ worst-case running time. Note that all of these algorithms are limited to unweighted graphs. The algorithm of Thorup is based on greedy tree packings us-

[*]The full version of the paper can be accessed at `https://arxiv.org/abs/2101.05033`.

[†]University of Vienna, Faculty of Computer Science, Vienna, Austria, `monika.henzinger@univie.ac.at`

[‡]University of Vienna, Faculty of Computer Science, Vienna, Austria, `alexander.noe@univie.ac.at`

[§]Heidelberg University, Heidelberg, Germany, `christian.schulz@informatik.uni-heidelberg.de` ing top trees. Implementations [4, 2] of static greedy tree packing algorithms give experimental results that are significantly slower than implementations of minimum cut algorithms based on edge contraction [40, 44] or maximum flows [17] on similar graphs [4, 24, 20].

For *planar* graphs with arbitrary edge-weights, Łącki and Sankowski [34] give a fully-dynamic algorithm with $\mathcal{O}\left(n^{5/6}\log^{5/2} n\right)$ time per update. To the best of our knowledge, there exists no implementation of any of these dynamic algorithms.

An important subproblem for many dynamic minimum cut algorithms is finding all minimum cuts. Even though a graph can have up to $\binom{n}{2}$ minimum cuts [26], there is a compact representation of all minimum cuts of a graph called *cactus graph* with at most $\mathcal{O}(n)$ vertices and edges. A cactus graph is a graph in which each edge belongs to at most one simple cycle. Nagamochi and Kameda [41] give a representation of all minimum cuts separating two vertices s and t in a so-called (s,t)-cactus representation. Based on this (s,t)-cactus representation, Nagamochi et al. [43] give an algorithm that finds all minimum cuts and gives the minimum cut cactus in $\mathcal{O}\left(nm + n^2\log n + n^* m \log n\right)$, where n^* is the number of vertices in the cactus. Karger and Stein [29] give a randomized algorithm to find all minimum cuts in $\mathcal{O}\left(n^2\log^3 n\right)$ time by contracting random edges. Based on the algorithm of Karzanov and Timofeev [31] and its parallel variant given by Naor and Vazirani [45] they show how to give the cactus representation of the graph in the same asymptotic time. Karger and Panigrahi [28] give a near-linear time algorithm that constructs a cactus representation of all minimum cuts. Ghaffari et al. [13] give an algorithm that finds a compact representation of all *non-trivial minimum cuts* of a simple unweighted graph in $\mathcal{O}\left(m\log^2 n\right)$ time. Using the techniques of Karger and Stein the algorithm can trivially give the cactus representation of all minimum cuts in $\mathcal{O}\left(n^2\log n\right)$. Recently, Henzinger et al. [21] developed an algorithm that combines various data reductions with an efficient implementation of the algorithm of Nagamochi et al. [43] and can find all minimum cuts in graphs with up to billions of edges and millions of minimum cuts in a few minutes. In each step, the algorithm of Nagamochi et al. [43] selects a random edge $e = (u,v)$ and computes $\lambda(u,v)$, the smallest cut separating them. If $\lambda(u,v) = \lambda$, the edge is *critical* and at least one minimum cut separates u from v. The set of minimum u-v-cuts can be described as a set of vertex sets $(V_1,\dots,V_k)$ with $u \in V_1$ and $v \in V_k$, where for each $i \in [1,k-1] : (V_1 \cup \dots \cup V_i, V_{i+1} \cup \dots \cup V_k)$ is a minimum cut. The algorithm then creates one subproblem for each vertex set V_i, where $V \setminus V_i$ is contracted into a single vertex and combines the cacti when leaving the recursion.

Our Results In this paper, we give the first implementation of a *fully-dynamic algorithm* for the *minimum cut problem* in a weighted graph. Our algorithm maintains an exact global minimum cut under edge insertions and deletions. For edge insertions, we use the approach of Henzinger [23] and Goranci et al. [16], who maintain a compact data structure of all minimum cuts in a graph and invalidate only the minimum cuts that are affected by an edge insertion. We hereby use the recent algorithm of Henzinger et al. [21] to compute all minimum cuts in a graph. For edge deletions, we use the push-relabel algorithm of Goldberg and Tarjan [14] to certify whether the previous minimum cut is still a minimum cut. As we only need to certify whether an edge deletion changes the value of the minimum cut, we can perform optimizations that significantly improve the speed of the push-relabel algorithm for our application. In particular, we develop a fast initial labeling scheme and terminate early when the connectivity value is certified.

2 Basic Concepts

Let $G = (V, E, c)$ be a weighted undirected simple graph with vertex set V, edge set $E \subset V \times V$ and non-negative edge weights $c : E \to \mathbb{N}$. We extend c to a set of edges $E' \subseteq E$ by summing the weights of the edges; that is, let $c(E') := \sum_{e=(u,v)\in E'} c(u,v)$ and let $c(u)$ denote the sum of weights of all edges incident to vertex v. The *weighted degree* of a vertex is the sum of the weights of its incident edges. For brevity, we simply call this the *degree* of the vertex. Let $n = |V|$ be the number of vertices and $m = |E|$ be the number of edges in G. The *neighborhood* $N(v)$ of a vertex v is the set of vertices adjacent to v. For a set of vertices $A \subseteq V$, we denote by $E[A] := \{(u,v) \in E \mid u \in A, v \in V \setminus A\}$; that is, the set of edges in E that start in A and end in its complement. A cut $(A, V \setminus A)$ is a partitioning of the vertex set V into two non-empty *partitions* A and $V \setminus A$, each being called a *side* of the cut. The *capacity* or *weight* of a cut $(A, V \setminus A)$ is $c(A) = \sum_{(u,v)\in E[A]} c(u,v)$. A *minimum cut* is a cut $(A, V \setminus A)$ that has smallest capacity $c(A)$ among all cuts in G. We use $\lambda(G)$ (or simply λ, when its meaning is clear) to denote the value of the minimum cut over all $A \subset V$. For two vertices s and t, we denote $\lambda(G,s,t)$ as the capacity of the smallest cut of G, where s and t are on different sides of the cut. $\lambda(G,s,t)$ is also known as the *minimum s-t-cut* of the graph. If all edges have weight 1, $\lambda(G,s,t)$ is also called the *connectivity* of vertices s and t. The connectivity $\lambda(G,e)$ of an edge $e = (s,t)$ is defined as $\lambda(G,s,t)$, the connectivity of its incident vertices. At any point in the execution of a minimum cut algorithm, $\hat{\lambda}(G)$ (or simply $\hat{\lambda}$) denotes the smallest upper bound of the minimum cut that the algorithm discovered until that point. For a vertex $u \in V$ with minimum vertex degree, the size of the *trivial cut* $(\{u\}, V \setminus \{u\})$ is equal to the vertex degree of u. Many algorithms tackling the minimum cut problem use *graph contraction.* Given an edge $e = (u, v) \in E$, we define $G/(u, v)$ (or G/e) to be the graph after *contracting edge* (u, v). In the contracted graph, we delete vertex v and all edges incident to this vertex. For each edge $(v, w) \in E$, we add an edge (u, w) with $c(u, w) = c(v, w)$ to G or, if the edge already exists, we give it the edge weight $c(u, w) + c(v, w)$.

A graph with n vertices can have up to $\Omega(n^2)$ minimum cuts [26]. To see that this bound is tight, consider an unweighted cycle with n vertices. Each set of 2 edges in this cycle is a minimum cut of G. This yields a total of $\binom{n}{2}$ minimum cuts. However, all minimum cuts can be represented by a cactus graph $\mathcal{C}$ with up to $2n$ vertices and $\mathcal{O}(n)$ edges [43]. A cactus graph is a connected graph, in which any two simple cycles have at most one vertex in common. In a cactus graph, each edge belongs to at most one simple cycle.

To represent all minimum cuts of a graph G in an edge-weighted cactus graph $\mathcal{C} = (V(\mathcal{C}), E(\mathcal{C}))$, each vertex of $\mathcal{C}$ represents a possibly empty set of vertices of G and each vertex in G belongs to a vertex in $\mathcal{C}$. Let Π be a function that assigns a cactus vertex $V(\mathcal{C})$ to each vertex in G. Then every cut $(S, V(\mathcal{C})\backslash S)$ in $\mathcal{C}$ corresponds to a minimum cut $(A, V\backslash A)$ in G where $A = \cup_{x \in S}\Pi(x)$. In $\mathcal{C}$, all edges that do not belong to a cycle have weight λ and all cycle edges have weight $\frac{\lambda}{2}$. A minimum cut in $\mathcal{C}$ consists of either one tree edge or two edges of the same cycle. We denote by n^* the number of vertices in $\mathcal{C}$ and m^* the number of edges in $\mathcal{C}$. The weight $c(v)$ of a vertex $v \in \mathcal{C}$ is equal to the number of vertices in G that are assigned to v.

In this work we use and adapt the push-relabel algorithm of Goldberg and Tarjan [14]. In the full version of this paper [19] we give a brief summary of the push-relabel algorithm, for more details we refer the reader to the original work.

3 Fully Dynamic Minimum Cut

In this section we develop an efficient fully-dynamic algorithm for the global minimum cut. For this, we use techniques from a multitude of original works combined with new and improved algorithmic solutions to engineer an algorithm that is able to solve the dynamic minimum cut problem by orders of magnitude faster than a static recomputation of the solution for a wide variety of graphs. An important observation for dynamic minimum cut algorithms is that graphs often have a large set of global minimum cuts [21]. Thus, dynamic minimum cut algorithms can avoid costly recomputation by storing a compact data structure representing all minimum cuts [23, 16] and only invalidate changed cuts in edge insertion. The data structure we use is a *cactus graph*, i.e. a graph in which every edge is part of at most one cycle. A minimum cut in the cactus graph is represented by either a tree edge or two edges of the same cycle [21]. For a graph with multiple connected components, i.e. a graph whose minimum cut value $\lambda = 0$, the cactus graph $\mathcal{C}$ has an empty edge set and one vertex corresponding to each connected component.

The rest of this section is organized as follows: we start by explaining the incremental minimum cut algorithm, followed by a description of the decremental minimum cut algorithm and conclude by showing how to combine the routines to a fully dynamic minimum cut algorithm. Our fully dynamic algorithm is a composition of our incremental and decremental algorithms.

3.1 Incremental Minimum Cut For incremental minimum cut, our algorithm is closely related to the exact incremental dynamic algorithms of Henzinger [23] and Goranci et al. [16]. On initialization of the algorithm with graph G, we run the recent algorithm of Henzinger et al. [21] on G to find the weight of the minimum cut λ and the cactus graph $\mathcal{C}$ representing all minimum cuts in G. Each minimum cut in $\mathcal{C}$ corresponds to a minimum cut in G and each minimum cut in G corresponds to one or more minimum cuts in $\mathcal{C}$ [23].

The insertion of an edge $e = (u, v)$ with positive weight $c(e) > 0$ increases the weight of all cuts in which u and v are in different partitions, i.e. in different vertices of the cactus graph $\mathcal{C}$. The weight of cuts in which u and v are in the same partition remains unchanged. As edge weights are non-negative, no cut weight can be decreased by inserting additional edges.

If $\Pi(u) = \Pi(v)$, i.e. both vertices are mapped to the same vertex in $\mathcal{C}$, there is no minimum cut that separates u and v and all minimum cuts remain intact. If $\Pi(u) \neq \Pi(v)$, i.e. the vertices are mapped to different vertices in $\mathcal{C}$, we need to invalidate the affected minimum cuts by contracting the corresponding edges in $\mathcal{C}$.

Path Contraction Dinitz [9] shows that for a connected graph with $\lambda > 0$ the minimum cuts that are affected by the insertion of (u, v) correspond to the minimum cuts on the path between $\Pi(u)$ and $\Pi(v)$. We find the path using alternating breadth-first searches from $\Pi(u)$ and $\Pi(v)$. For this path-finding algorithm, imagine the cactus graph $\mathcal{C}$ as a tree graph in which each cycle is contracted into a single vertex. On this tree, there is a unique path from $\Pi(u)$ to $\Pi(v)$.

For every cycle in $\mathcal{C}$ that contains at least two vertices of the path between $\Pi(u)$ and $\Pi(v)$, the cycle is "squeezed" by contracting the first and last path vertex in the cycle, thus creating up to two new cycles. For details and correctness proofs we refer the reader to the work of Dinitz [9]. The intuition is that due to the insertion of the new edge, all cactus vertices in the path from $\Pi(u)$ and $\Pi(v)$ are now connected with a value $> \lambda$, as their previous connection was λ and the newly introduced edge increased it. For any cycle in the path, this also includes the first and last cycle vertices x and y in the path, as these two vertices now have a higher connectivity $\lambda(x, y)$. The minimum cuts that are represented by edges in this cycle that have x and y on the same side are unaffected, as all vertices in the path from $\Pi(u)$ and $\Pi(v)$ are on the same side of this cut. As this is not true for cuts that separate x and y, we merge x and y (as well as the rest of the path from $\Pi(u)$ to $\Pi(v)$), which "squeezes" the cycle and creates up to two new cycles.

If the graph has multiple connected components, i.e. the graph has a minimum cut value $\lambda = 0$, $\mathcal{C}$ is a graph with no edges where each connected component is mapped to a vertex. The insertion of an edge between different connected components $\Pi(u)$ and $\Pi(v)$ merges the two vertices representing the connected components, as they are now connected.

If $\mathcal{C}$ has at least two non-empty vertices after the edge insertion, there is at least one minimum cut of value λ remaining in the graph, as all minimum cuts that were affected by the insertion of edge e were just removed from the cactus graph $\mathcal{C}$. As an edge insertion cannot decrease any connectivities, λ remains the value of the minimum cut. If $\mathcal{C}$ only has a single non-empty vertex, we need to recompute the cactus graph $\mathcal{C}$ using the algorithm of Henzinger et al. [21].

Checking the set affiliation Π of u and v can be done in constant time. If $\Pi(u) = \Pi(v)$ and the cactus graph does not need to be updated, no additional work needs to be done. If $\Pi(u) \neq \Pi(v)$, we perform breadth-first search on $\mathcal{C}$ with $n^* := |V(\mathcal{C})|$ and $m^* := |E(\mathcal{C})|$ which has a asymptotic running time of $\mathcal{O}(n^* + m^*) = \mathcal{O}(n^*)$, contract the path from $\Pi(u)$ to $\Pi(v)$ in $\mathcal{O}(n^*)$ and then update the set affiliation of all contracted vertices. This update has a worst-case running time of $\mathcal{O}(n)$, however, contracting all vertices of the path from $\Pi(u)$ to $\Pi(v)$ into the cactus graph vertex that already corresponds to the most vertices of G, we often only need to update the affiliation of a few vertices. Both the initial computation and a full recomputation of the minimum cut cactus have a worst-case running time of $\mathcal{O}\big(nm + n^2 \log n + n^* m \log n\big)$.

3.2 Decremental Minimum Cut

The deletion of an edge $e = (u, v)$ with positive weight $c(e) > 0$ decreases the weight of all cuts in which u and v are in different partitions. This might lead to a decrease of the minimum cut value λ and thus the invalidation of the minimum cuts in the existing minimum cut cactus $\mathcal{C}$. The value of the minimum cut $\lambda(G, u, v)$ that separates vertices u and v is equal to the maximum flow between them and can be found by a variety of algorithms [8, 11, 14]. In order to check whether λ is decreased by this edge deletion, we need to check whether $\lambda(G - e, u, v) < \lambda(G)$. For this purpose, we use the push-relabel algorithm of Goldberg and Tarjan [14] which aims to push flow from u to v until there is no more possible path. In the following we introduce modifications to their algorithm that make it significantly faster in our application.

We terminate the algorithm as soon as $\lambda(G)$ units of flow reached v. If $\lambda(G)$ units of flow from u reached v, we know that $\lambda(G-e, u, v) \geq \lambda(G)$, i.e. the connectivity of u and v on $G - e$ is at least as large as the minimum cut on G, the minimum cut value λ remains unchanged. Note that iff $\lambda(G - e, u, v) = \lambda(G)$, the deletion of e introduces one or more new minimum cuts. We do not introduce these new cuts to $\mathcal{C}$. The trade-off hereby is that we are able to terminate the push-relabel algorithm earlier and do not need to perform potentially expensive operations to update the cactus, but do not necessarily keep all cuts and have to recompute the cactus earlier. As most real-world graphs have a large number of minimum cuts [21], there are far more edge deletions than recomputations of $\mathcal{C}$.

Each edge deletion calls the push-relabel algorithm using the lowest-label selection rule with a worst-case running time of $\mathcal{O}\big(n^2 m\big)$ [14]. The lowest-label selection rule picks the active vertices whose distance label is lowest, i.e. a vertex that is close to the sink v. Using highest-level selection would improve the worst-case running time to $\mathcal{O}\big(n^2 \sqrt{m}\big)$, but we aim to push as much flow as possible to the sink early to be able to terminate the algorithm early as soon as λ units of flow reach the sink. Using lowest-level selection prioritizes the vertices close to the sink and thus increases the amount of flow which reaches the sink at a given point in time. Preliminary experiments show faster running times using the lowest-level selection rule.

3.2.1 Decremental Rebuild of Cactus Graph

If the push-relabel algorithm finishes with a value of $< \lambda(G)$, we update the minimum cut value $\lambda(G - e)$ to $\lambda(G - e, u, v)$. As the minimum cut value changed by the deletion of e and this deletion only affects cuts which contain e, we know that all minimum cuts of the updated graph $G - e$ separate u and v. We use this information to significantly speed up the cactus construction. Instead of running the algorithm of Henzinger et al. [21], we run only the subroutine which is used to compute the (u, v)-cactus, i.e. the cactus graph which contains all cuts that separate u and v, as we know that all minimum cuts of $G - e$ separate u and v. This routine, developed by Nagamochi and Kameda [42], finds a u-v-cactus a running time of $\mathcal{O}(n + m)$.

Note that the routine of Nagamochi and Kameda [42] only guarantees to find all minimum u-v-cuts if an edge $e = (u, v)$ with $c(e) > 0$ exists ([42, Lemma 3.4]). As this edge was just deleted in $G - e$ and therefore does not exist, it is possible that *crossing* u-v-cuts $(X, \overline{X})$ and $(Y, \overline{Y})$ with $u \in X$ and $u \in Y$ exist. Two cuts are *crossing*, if both $(\overline{X} \cap Y)$ and $(X \cap \overline{Y})$ are not empty. As we only find one cut in a pair of crossing cuts, the u-v-cactus is not necessarily maximal. However, the operation is significantly faster than recomputing the complete minimum cut cactus in which almost all edges are not part of any minimum cut. While it is not guaranteed that the decremental rebuild algorithm finds all minimum cuts in $G-e$, every cut of size $\lambda(G - e, u, v)$ that is found is a minimum cut. As we build the minimum cut cactus out of minimum cuts, it is a valid (but potentially incomplete) minimum cut cactus and the algorithm is correct.

3.2.2 Local Relabeling Many efficient implementations of the push-relabel algorithm use the global relabeling heuristic [5] in order to direct flow towards the sink more efficiently. The push-relabel algorithm maintains a distance label d for each vertex to indicate the distance from that vertex to the sink using only edges that can receive additional flow. The global relabeling heuristic hereby periodically performs backward breadth-first search to compute distance labels on all vertices.

This heuristic can also be used to set the initial distance labels in the flow network for a flow problem with source u and sink v. This has a running time of $\mathcal{O}(n + m)$ but helps lead the flow towards the sink. As our algorithm terminates the push-relabel algorithm early, we try to avoid the $\mathcal{O}(m)$ running time while still giving the flow some guidance. Thus, we perform *local relabeling* with a *relabeling depth* of γ for $\gamma \in [0, n)$, where we set $d(v) = 0$, $d(u) = n$ and then perform a backward breadth-first search around the sink v, in which we set $d(x)$ to the length of the shortest path between x and v (at this point, there is no flow in the network, so every edge in G is admissible). Instead of setting the distance of every vertex, we only explore the neighborhoods of vertices x with $d(x) < \gamma$, thus we only set the distance-to-sink for vertices with $d(x) \leq \gamma$. For every vertex y with a higher distance, we set $d(y) = (\gamma + 1)$. This results in a running time for setting the distance labels of $\mathcal{O}(n)$ plus the time needed to perform the bounded-depth breadth-first search.

This process creates a "funnel" around the sink to lead flow towards it, without incurring a running time overhead of $\Theta(m)$ (if γ is set sufficiently low). Note that this is useful because the push-relabel algorithm is terminated early in many cases and thus initializing the distance labels faster can give a large speedup. We give experimental results for different relabeling depths γ for local relabeling in our application in Section 4.1. We now show correctness of this local relabeling scheme.

Goldberg and Tarjan show that each push and relabel operation in the push-relabel algorithm preserve a *valid labeling* [14]. A valid labeling is a labeling d, where in a given preflow f and corresponding residual graph G_f, for each edge $e = (u, v) \in E_f$, $d(u) \leq d(v)+1$. We therefore need to show that the labeling d that is given by the initial local relabeling is a valid labeling.

Lemma 3.1. *Let $G = (V, E, c)$ be a flow-graph with source s and sink t and let d be the vertex labeling given by the local relabeling algorithm. The vertex labeling d is a valid labeling.*

Proof. The vertex labeling d is generated using breadth-first search. Thus, for every edge $e = (u, v)$ where $u \neq s$ and $v \neq s$, $|d(u) - d(v)| \leq 1$. We prove this by contradiction. W.l.o.g. assume that $d(u)-d(v) > 1$. As $u \neq s$ and s is the only vertex with $d(s) > \gamma$, $d(u) \leq \gamma+1$ and $d(v) < \gamma$. Thus, at some point of the breadth-first search, we set the distance labels of all neighbors of v that do not yet have a distance label to $d(v)+1$. As edge $e = (u, v)$ exists, u and v are neighbors and the labeling sets $d(u) = d(v) + 1$. This contradicts $d(u) - d(v) > 1$.

This shows that the labeling is valid for every edge not incident to the source s, as distance labels of incident non-source vertices differ by at most 1. The only edges we need to check are edges incident to s. In the initialization of the push-relabel algorithm, all outgoing edges of the source s are fully saturated with flow and are thus no outgoing edge of s is in E_f. For ingoing edges $e = (v, s)$, we know that $0 \leq d(v) \leq \gamma + 1 = n$ and thus know that $d(v) \leq d(s)$. Thus e respects the validity of labeling d. ∎

Lemma 3.1 shows that local relabeling gives a valid labeling; which is upheld by the operations in the push-relabel algorithm [14]. Thus, correctness of the modified algorithm follows from the correctness proof of Goldberg and Tarjan.

Resetting the vertex data structures can be performed in $\mathcal{O}(n)$, however there are m edges whose current flow needs to be reset to 0. Using early termination we hope to solve some problems very fast in practice, as we can sometimes terminate early without exploring large parts of the graph. Thus, resetting of the edge flows in $\mathcal{O}(m)$ is a significant problem and is avoided using implicit resetting as described in the following paragraph.

Each flow problem that is solved over the course of the dynamic minimum cut algorithm is given a unique ID, starting at an arbitrary integer and incrementing from there. In addition to the current flow on an edge, we also store the ID of the last problem which accessed the flow on this edge. When the flow of an edge is read or updated in a flow problem, we check whether the ID of the last access equals the ID of the current problem. If they are equal, we simply return or update the flow value, as the edge has already been accessed in this flow problem and does not need to be reset. Otherwise, we need to reset the edge flow to 0 and set the problem ID to the ID of the current problem and then perform the operation on the updated edge. Thus, we implicitly reset the edge flow on first access in the current problem. As we increment the flow problem ID after every flow problem, no two flow problems share the same ID.

Using this implicit reset of the edge flows saves $\mathcal{O}(m)$ overhead but introduces a constant amount of work on each access and update of the edge flow. It is therefore useful in practice if the problem terminates with significantly fewer than m flow updates due to early termination. It does not affect the worst-case running time of the algorithm, as we only perform a constant amount of work on each edge update. The running time of the initialization of the implementation is improved from $\mathcal{O}(n+m)$ to $\mathcal{O}(n)$, as we do not explicitly reset the flow on each edge.

3.3 Fully Dynamic Minimum Cut

Based on the incremental and decremental algorithm, we now describe our fully dynamic algorithm. As the operations in the previous section each output the minimum cut $\lambda(G)$ and a corresponding cut cactus $\mathcal{C}$ that stores a set of minimum cuts for G, the algorithm gives correct results on all operations. However, there are update sequences in which every insertion or deletion changes the minimum cut value and, thus, triggers a recomputation of the minimum cut cactus $\mathcal{C}$. One such example is the repeated deletion and reinsertion of an edge that belongs to a minimum cut. In the following paragraphs we describe a technique that is used to mitigate such worst-case instances. Nevertheless, it is still possible to construct update sequences in which the minimum cut cactus $\mathcal{C}$ needs to be recomputed every $\mathcal{O}(1)$ edge updates and thus the worst-case asymptotic running time per update is equal to the running time of the static algorithm.

3.3.1 Cactus Cache

Computing the minimum cut cactus $\mathcal{C}$ is expensive if there is a large set of minimum cuts and the cactus is therefore large. Thus, it is beneficial to reduce the amount of recomputations to speed up the process. On some fully dynamic workloads, the minimum cut often jumps between values λ_1 and λ_2 with $\lambda_1 > \lambda_2$, where the minimum cut cactus for cut value λ_1 is large and thus expensive to recompute whenever the cut value changes.

A simple example workload is a large unweighted cycle, which has a minimum cut of 2. If we delete any edge, the minimum cut value changes to 1, as the incident vertices have a degree of 1. By reinserting the just-deleted edge, the minimum cut changes to a value of 2 again and the minimum cut cactus is equal to the cactus prior to the edge deletion. Thus we can save a significant amount of work by caching and reusing the previous cactus graph when the minimum cut is increased to 2 again.

Reuse Cactus Graph from Cache Whenever the deletion of an edge e from graph G decreases the minimum cut value from λ_1 to λ_2, we cache the previous cactus $\mathcal{C}$. After this point, we also remember all edge insertions, as these can invalidate minimum cuts in $\mathcal{C}$. If at a later point the minimum cut is again increased from λ_2 to λ_1 and the number of edge insertions divided by the number of vertices in $\mathcal{C}$ is smaller than a parameter δ, we recreate the cactus graph from the cache instead of recomputing it. The default value for δ is 2. The algorithm does not store the intermediate edge deletion, as these can only lower connectivities and by computing the minimum cut value we know that there is no cut of value $< \lambda_1$ and thus all cuts of value λ_1 are global minimum cuts.

Note that this process does not necesarily maintain all minimum cuts in $\mathcal{C}$. For each edge insertion since caching we perform the edge insertion operation from Section 3.1 to eliminate all cuts that are invalidated by the edge insertion. All cuts that remain in $\mathcal{C}$ are still minimum cuts. If there are only a small amount of edge insertions since the cactus was cached, this is significantly faster than recomputing the cactus from scratch. As we do not remember edge deletions, the cactus might not contain all minimum cuts and thus require slightly earlier recomputation.

4 Experiments and Results

We now perform an experimental evaluation of the proposed algorithms. This is done in the following order.

We first describe the instances used in our experiments. In our experiments, we use a wide variety of static and dynamic graph instances. These are social graphs, web graphs, co-purchase matrices, cooperation networks and some generated instances. All instances are undirected. If the original graph is directed, we generate an undirected graph by removing edge directions and then removing duplicate edges. In our experiments, we use three families of graphs.

Family A consists of 28 graphs obtained from the work of Henzinger et al. on the global minimum cut problem [20], originally from the 10th DIMACS Implementation challenge [1] and the SuiteSparse Matrix Collection [6]. As finding *any* minimum cut is easy if the minimum cut is equal to the minimum degree, these graphs are *k-cores* of large social networks in which the minimum cut is strictly smaller than the minimum degree. A *k-core* of a graph is a subgraph in which we iteratively remove all vertices of degree $< k$ until the graph has a minimum degree of k. If the k-core has multiple connected components, we use the largest of them. In this graph family, there are generally only one or a few minimum cuts on each graph and the minimum cut is strictly smaller than the minimum degree. In Table 1 we report both the minimum cut λ and the minimum degree δ. In the dataset there are 7 different graphs, each with 4 different values of $\delta = k$ with $\lambda < \delta$ in every instance.

Family B consists of 65 graphs obtained from the work of Henzinger et al. on finding all minimum cuts of a graph [21], originally from from the 10th DIMACS Implementation challenge [1], the SuiteSparse Matrix Collection [6] and the Walshaw Graph Partitioning Archive [50]. They represent a wide variety of real-world graphs from different fields and applications. In contrast to Family A, most graphs in this graph family have a large number of minimum cuts and generally the minimum cut is equal to the minimum degree. As most real-world graphs have some vertices of very low degree and therefore also a low minimum cut, we also create instances with higher minimum cut by computing beforehand the largest subgraph G_x of G that does not contain any minimum cut of size $\lambda(G)$.

Family C consists of a set of 36 dynamic graphs from Network Repository [48, 49]. These graphs consist of a sequence of edge insertions and deletions. While edges are inserted and deleted, all vertices are static and remain in the graph for the whole time. Each edge update has an associated timestamp, a set of updates with the same timestamp is called a *batch*. Most of the graphs in this dataset have multiple connected components, i.e. their minimum cut λ is 0.

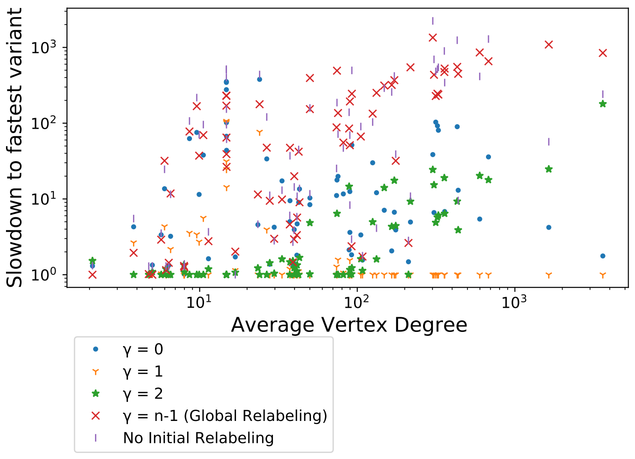

Figure 1: Effect of local relabeling depth on running time of delete operations.

In Section 4.1 we analyze the impact of local relabeling on the static preflow-push algorithm to determine which value of the relabeling depth to use in the experiments on dynamic graphs. Then (Sections 4.3 and 4.2) we evaluate our dynamic algorithms on a wide variety of instances. We then generate a set of worst-case problems and use these to evaluate the performance of our algorithm on instances that were created in order to be difficult for them. Additionally, we examine how often the dynamic algorithm finds the most balanced minimum cut, i.e. the minimum cut which has the highest number of vertices in the smaller partition.

Experimental Setup and Methodology We implemented the algorithms using C++-17 and compiled all code using g++ version 8.3.0 with full optimization (`-O3`). Our experiments are conducted on a machine with two Intel Xeon Gold 6130 processors with 2.1GHz with 16 CPU cores each and 256 GB RAM in total. All codes in this work are sequential. In this section we first describe our experimental methodology. Afterwards, we evaluate different algorithmic choices in our algorithm and then we compare our algorithm to the state of the art. When we report a mean result we give the geometric mean as problems differ significantly in cut size and time. Our code is freely available under the permissive MIT license[1].

4.1 Local Relabeling In order to examine the effects of local relabeling with different values of relabel-

[1]`https://github.com/VieCut/VieCut`

Graph Family A					
Graph	n	m	λ	δ	n^*
`com-orkut`	$2.4M$	$112M$	14	16	2
	114.190	$18M$	89	95	2
	107.486	$17M$	76	98	2
	103.911	$17M$	70	100	2
`eu-2005`	605.264	$15M$	1	10	63
	271.497	$10M$	2	25	3
	58.829	$3.7M$	29	60	2
	5.289	464.821	19	100	2
`gsh-2015-host`	$25M$	$1.3B$	1	10	175
	$5.3M$	$944M$	1	50	32
	$2.6M$	$778M$	1	100	16
	98.275	$188M$	1	1.000	3
`hollywood-2011`	$1.3M$	$109M$	1	20	13
	576.111	$87M$	6	60	2
	328.631	$71M$	77	100	2
	138.536	$47M$	27	200	2
`twitter-2010`	$13M$	$958M$	1	25	2
	$10M$	$884M$	1	30	3
	$4.3M$	$672M$	3	50	3
	$3.5M$	$625M$	3	60	2
`uk-2002`	$9M$	$226M$	1	10	1.940
	$2.5M$	$115M$	1	30	347
	783.316	$51M$	1	50	138
	98.275	$11M$	1	100	20
`uk-2007-05`	$68M$	$3.1B$	1	10	3.202
	$16M$	$1.7B$	1	50	387
	$3.9M$	$862M$	1	100	134
	223.416	$183M$	1	1.000	2

Graph Family B					
`amazon`	64.813	153.973	1	1	10.068
`auto`	448.695	$3.31M$	4	4	43
	448.529	$3.31M$	5	5	102
	448.037	$3.31M$	6	6	557
	444.947	$3.29M$	7	7	1.128
	437.975	$3.24M$	8	8	2.792
	418.547	$3.10M$	9	9	5.814
`caidaRouterLevel`	190.914	607.610	1	1	49.940
`cfd2`	123.440	$1.48M$	7	7	15
`citationCiteseer`	268.495	$1.16M$	1	1	43.031
	223.587	$1.11M$	2	2	33.423
	162.464	862.237	3	3	23.373
	109.522	435.571	4	4	16.670
	73.595	225.089	5	5	11.878
	50.145	125.580	6	6	8.770
`cnr-2000`	325.557	$2.74M$	1	1	87.720
	192.573	$2.25M$	2	2	33.745
	130.710	$1.94M$	3	3	11.604
	110.109	$1.83M$	4	4	9.256
	94.664	$1.77M$	5	5	4.262
	87.113	$1.70M$	6	6	5.796
	78.142	$1.62M$	7	7	3.213
	73.070	$1.57M$	8	8	2.449
`coAuthorsDBLP`	299.067	977.676	1	1	45.242
`cs4`	22.499	43.858	2	2	2
`delaunay_n17`	131.072	393.176	3	3	1.484
`fe_ocean`	143.437	409.593	1	1	40
`kron-logn16`	55.319	$2.46M$	1	1	6.325
`luxembourg`	114.599	239.332	1	1	23.077
`vibrobox`	12.328	165.250	8	8	625
`wikipedia`	35.579	495.357	1	1	2.172

Graph Family B (continued)					
Graph	n	m	λ	δ	n^*
`amazon-2008`	735.323	$3.52M$	1	1	82.520
	649.187	$3.42M$	2	2	50.611
	551.882	$3.18M$	3	3	35.752
	373.622	$2.12M$	5	5	19.813
	145.625	582.314	10	10	64.657
`coPapersCiteseer`	434.102	$16.0M$	1	1	6.372
	424.213	$16.0M$	2	2	7.529
	409.647	$15.9M$	3	3	7.495
	379.723	$15.5M$	5	5	6.515
	310.496	$13.9M$	10	10	4.579
`eu-2005`	862.664	$16.1M$	1	1	52.232
	806.896	$16.1M$	2	2	42.151
	738.453	$15.7M$	3	3	21.265
	671.434	$13.9M$	5	5	18.722
	552.566	$11.0M$	10	10	23.798
`hollywood-2009`	$1.07M$	$56.3M$	1	1	11.923
	$1.06M$	$56.2M$	2	2	17.386
	$1.03M$	$55.9M$	3	3	21.890
	942.687	$49.2M$	5	5	22.199
	700.630	$16.8M$	10	10	19.265
`in-2004`	$1.35M$	$13.1M$	1	1	278.092
	909.203	$11.7M$	2	2	89.895
	720.446	$9.2M$	3	3	45.289
	564.109	$7.7M$	5	5	33.428
	289.715	$5.1M$	10	10	12.947
`uk-2002`	$18.4M$	$261.6M$	1	1	$2.5M$
	$15.4M$	$254.0M$	2	2	$1.4M$
	$13.1M$	$236.3M$	3	3	938.319
	$10.6M$	$207.6M$	5	5	431.140
	$7.6M$	$162.1M$	10	10	298.716
	657.247	$26.2M$	50	50	24.139
	124.816	$8.2M$	100	100	3.863

Graph Family C					
Dynamic Graph	n	Insertions	Deletions	Batches	λ
`aves-weaver-social`	445	1.423	0	23	0
`ca-cit-HepPh`	28.093	$4.60M$	0	2.337	0
`ca-cit-HepTh`	22.908	$2.67M$	0	219	0
`comm-linux-kernel-r`	63.399	$1.03M$	0	839.643	0
`copresence-InVS13`	987	394.247	0	20.129	0
`copresence-InVS15`	1.870	$1.28M$	0	21.536	0
`copresence-LyonS`	1.922	$6.59M$	0	3.124	0
`copresence-SFHH`	1.924	$1.42M$	0	3.149	0
`copresence-Thiers`	1.894	$18.6M$	0	8.938	0
`digg-friends`	279.630	$1.73M$	0	$1.64M$	0
`edit-enwikibooks`	134.942	$1.16M$	0	$1.13M$	0
`fb-wosn-friends`	63.731	$1.27M$	0	736.675	0
`ia-contacts_dublin`	10.972	415.912	0	76.944	0
`ia-enron-email-all`	87.273	$1.13M$	0	214.908	0
`ia-facebook-wall`	46.952	855.542	0	847.020	0
`ia-online-ads-c`	$15.3M$	133.904	0	56.565	0
`ia-prosper-loans`	89.269	$3.39M$	0	1.259	0
`ia-stackexch-user`	545.196	$1.30M$	0	1.154	1
`ia-sx-askubuntu-a2q`	515.273	257.305	0	257.096	0
`ia-sx-mathoverflow`	88.580	390.441	0	390.051	0
`ia-sx-superuser`	567.315	$1.11M$	0	$1.10M$	0
`ia-workplace-cts`	987	9.827	0	7.104	0
`imdb`	150.545	296.188	0	7.104	0
`insecta-ant-colony1`	113	111.578	0	41	4.285
`insecta-ant-colony2`	131	139.925	0	41	3.742
`insecta-ant-colony3`	160	241.280	0	41	1.539
`insecta-ant-colony4`	102	81.599	0	41	1.838
`insecta-ant-colony5`	152	194.317	0	41	6.671
`insecta-ant-colony6`	164	247.214	0	39	2.177
`mammalia-voles-kcs`	1.218	4.258	0	64	0
`SFHH-conf-sensor`	1.924	70.261	0	3.509	0
`soc-epinions-trust`	131.828	717.129	123.670	939	0
`soc-flickr-growth`	$2.30M$	$33.1M$	0	134	0
`soc-wiki-elec`	8.297	83.920	23.093	101.014	0
`soc-youtube-growth`	$3.22M$	$12.2M$	0	203	0
`sx-stackoverflow`	$2.58M$	392.515	0	384.680	0

Table 1: Statistics of static and dynamic graphs used in experiments.

ing depth γ, we run experiments using all static graph instances (Graph Family A and Graph Family B) from Table 1, in which we delete 1000 random edges in random order. We report the total time spent executing delete operations. We compare a total of 5 variants, one that does not run initial relabeling, three variants with relabeling depth $\gamma = 0, 1, 2$ and one variant which performs global relabeling in the initialization process, i.e. local relabeling with depth $\gamma = (n - 1)$. Local relabeling with $\gamma = 0$ is very similar to no relabeling, however the distance value of non-sink vertices are set to $(\gamma + 1) = 1$ and not to 0.

In Figure 1 we report the slowdown to the fastest variant for all static graph instances from Table 1. The x-axis shows the average vertex degree for the instances. On most instances, the fastest variant is local relabeling with $\gamma = 1$. Depending on the graph instance, this variant spends $25 - 90\%$ of the deletion time in the initialization (including initial relabeling). An increase in labeling depth increases the initialization running time, but decreases the subsequent algorithm running time. Thus we aim to find a labeling depth value that maintains some balance between initial labeling and the subsequent algorithm execution. On some instances, it is outperformed by local relabeling with $\gamma = 2$, which is slower by a factor of $3 - 10$x on most instances, with $90 - 99\%$ of the total running time spent in the initialization of the algorithm. We can see that in instances with a higher average degree, local relabeling with $\gamma = 1$ performs better. This is an expected result, as the larger local relabeling is more expensive in higher-average-degree graphs, as the 2-neighborhood of a vertex is much larger. Local relabeling with $\gamma = 2$ spends $90 - 99\%$ of the total running time in initialization and initial relabeling. The same effect is even more pronounced for the variant which performs global relabeling in initialization. On vertices with a low average degree, we can perform global relabeling in reasonable time, which makes the variant competitive with the local relabeling variants. However, in high average degree instances, the excessive running time of a global relabeling step causes the variant to have slowdowns of up to 1000x compared to the fastest variant. On all instances, the vast majority of running time is spent in initialization including initial global relabeling.

One graph family where local relabeling with $\gamma = 1$ performs badly are the graph instances based on `auto` [30], a 3D finite element mesh graph. These graphs are rather sparse (average degree 15) and planar. On these graphs, the value of the minimum cut divided by the average degree is very large, as they do not contain any vertices of degree 1, 2, 3. Thus, the variants which perform only minor local relabeling do not guide the flow enough and therefore the push-relabel algorithm takes a long time. On most other instances in our test set, local relabeling with $\gamma = 1$ is enough to guide at least λ flow to the sink quickly.

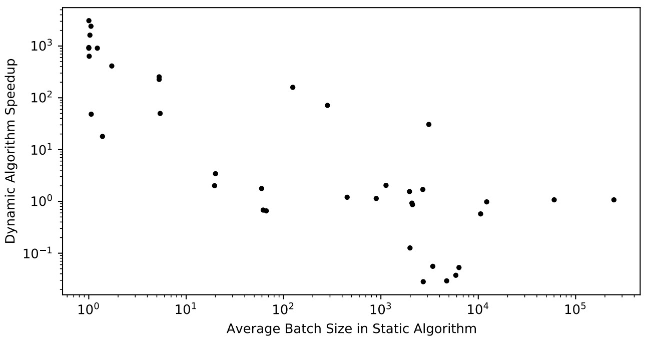

Figure 2: Speedup of Dynamic Algorithm.

Local relabeling with a relabeling depth $\gamma = 0$ (i.e. we set the distance of the sink to 0, the source to n and all other vertices to 1) has a slowdown factor of $10 - 100$x with only $1 - 10\%$ of the running time spent in the initialization. The slowdown factor is generally increasing for larger values of the minimum cut λ and average degree, which indicates that "the lack of guidance towards the sink" causes the algorithm to send flow to regions of the graph that are far away from the source. For graphs with large minimum cut value λ, the algorithm does not terminate early and needs to perform a significant amount of push and relabel steps. In variants that perform more relabeling at initialization, the flow is guided towards the sink by the distance labels and the termination trigger is reached faster. The variant which does not include any relabeling in the initialization phase has similar issues with an even larger slowdown factor of $10 - 2000$x, as even flow that is already incident to the sink does not necessarily flow straight to the sink.

On most instances, local relabeling with depth $\gamma = 1$ performed best, as it helps guide the flow towards the sink with additional work (compared to no relabeling) only equal to the degree of the sink. While performing more relabeling can increase this guidance even further, it comes with a trade-off in additional time spent in the initialization. Note that this is not a general observation for the push-relabel algorithm and can only be applied to our application, in which the push-relabel algorithm is terminated early as soon as λ units of flow reach the sink vertex. Based on these experiments, we use local relabeling with $\gamma = 1$ for edge deletions in all following experiments.

4.2 Dynamic Graphs

Figure 2 shows experimental results on the dynamic graph instances from Graph

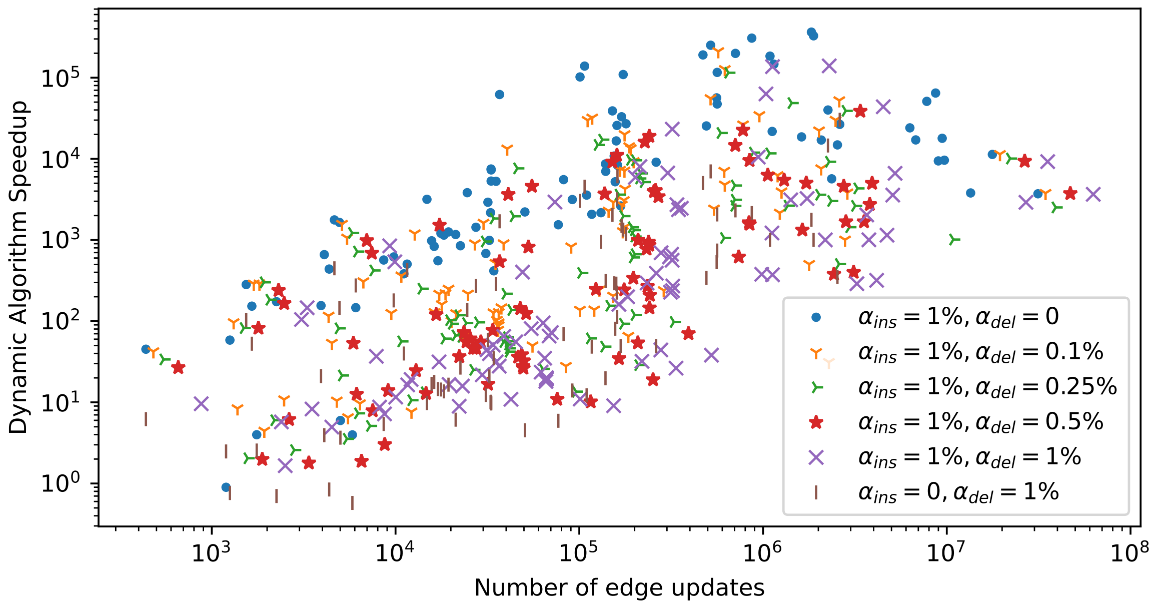

Figure 3: Speedup of Dynamic Algorithm on Random Insertions and Deletions from Static Graphs.

Family C in Table 1. These graph instances are mostly incremental with some being fully dynamic and most instances have multiple connected components, i.e. a minimum cut value $\lambda = 0$, even after all insertions. On these incremental graphs with multiple connected components, our algorithm behaves similar to a simple union-find based connected components algorithm that for edge insertion checks whether the incident vertices already belong to the same connected component and merges their connected components if they are different.

In this section we compare our dynamic minimum cut algorithm to an efficient implementation [18] of the static algorithm of Nagamochi et al. [44], which has been shown to be one of the fastest sequential algorithms for the minimum cut problem [4, 20]. As our algorithm is sequential, we restrict the static algorithm to a single core as well. Our algorithm uses a modified version [21] of this static algorithm, which finds all minimum cuts. As finding any minimum cut is faster than finding all of them, we compare our algorithm to the faster [18]. The static algorithm performs the updates batch-wise, i.e. the static algorithm is not called inbetween multiple edge updates with equal timestamp. In Figure 2, we show the dynamic speedup in comparison to the average batch size. As expected, there is a large speedup factor of up to 1000x for graphs with small batch sizes; and the speedup decreases for increasing batch sizes. The family of instances in which the dynamic algorithm is outperformed by the static algorithm is the `insecta-ant-colony` graph family [38]. These graphs have a very high minimum cut value and fewer batches than changes in the minimum cut value. Therefore, the dynamic algorithm which updates on every edge insertion needs to recompute the minimum cut cactus more often than the static algorithm is run and, thus, takes a longer time.

As these dynamic instances do not have sufficient diversity, we also perform experiments on static graphs in graph family B in which a subset of edges is inserted or removed dynamically. We report on this experiment in the following section.

4.3 Random Insertions and Deletions from Static Graphs Figure 3 shows results for dynamic edge insertions and deletions from all graphs in Graph Family A and B from Table 1. These graphs are static, we create a dynamic problem from graph $G = (V, E, c)$ as follows: let $\alpha_{ins} \in (0, 1)$ and $\alpha_{del} \in (0, 1)$ with $\alpha_{ins} + \alpha_{del} < 1$ be the edge insertion and deletion rate. We randomly select edge lists E_{ins} and E_{del} with $|E_{ins}| = \alpha_{ins} \cdot |E|$, $|E_{del}| = \alpha_{del} \cdot |E|$ and $E_{ins} \cap E_{del} = \emptyset$. For every vertex $v \in V$, we make sure that at least one edge incident to v is neither in E_{ins} nor in E_{del}, so that the minimum degree of $(V, E \backslash (E_{ins} \cap E_{del}), c)$ is strictly greater than 0 at any point in the update sequence.

We initialize the graph as $(V, E \backslash E_{ins}, c)$ and create a sequence of edge updates E_u by concatenating E_{ins} and E_{del} and randomly shuffling the combined list. Then we perform edge updates one after another and compute the minimum cut - either statically using the algorithm of Nagamochi et al. [44] or by performing an update in the dynamic algorithm - after every update. We report the total running time of either variant and give the speedup of the dynamic algorithm over the static algorithm as a function of the number of edge updates performed. For each graph we create problems with $\alpha_{ins} = 1\%$ and $\alpha_{del} \in \{0, 0.1\%, 0.25\%, 0.5\%, 1\%\}$; and additionally a decremental problem with $\alpha_{ins} = 0$ and $\alpha_{del} = 1\%$. We set the timeout for the static algorithm to 1 hour, if the algorithm does not finish before timeout, we approximate the total running time of the static algorithm by performing 100 or 1000 updates in batch.

Dynamic edge insertions are generally much faster than edge deletions, as most real-world graphs have large sets that are not separated by any global minimum cut. When inserting an edge where both incident vertices are in the same set in $\mathcal{C}$, the edge insertion only requires two array accesses; if they are in different sets, it requires a breadth-first search on the relatively small cactus graph $\mathcal{C}$ and only if there are no minimum cuts remaining, an edge insertion requires a recomputation. In contrast to that, every edge deletion requires solving of a flow problem and therefore takes significantly more time in average. Therefore, the average speedup is larger on problems with a higher rate of edge insertions.

Generally, the speedup of the dynamic algorithm increases with larger problems and more edge updates. For larger graphs with $\geq 10^6$ edge updates, the average speedup is more than four orders of magnitude for instances with $\alpha_{del} = 0$ and still more than two orders of magnitude for large instances when $\alpha_{del} = \alpha_{ins} = 1\%$. Note that in this experiment, the number of edge updates is a function of the number of edges, thus instances with more updates directly correspond to graphs with more edges. The average running time of an edge insertion in these instances is $37\mu s$, the average running time of an edge deletion is $942\mu s$. For decremental instances with $\alpha_{ins} = 0$, the speedup is generally lower, but still reaches multiple orders of magnitude in larger instances.

4.4 Worst-case instances On random edge insertions, there is a high chance that the vertices incident to the newly inserted edge were not separated by a minimum cut and therefore require no update of the cactus graph $\mathcal{C}$. In this experiment we aim to generate instances that aim to maximize the work performed by the dynamic algorithm. We initialize the graph as $G = (V, E, c)$ and add random unit-weight edges $e = (u, v)$ where $\Pi(u) \neq \Pi(v)$ for every newly added edge. Then we randomly select $|E_{ins}| = 1000$ edges to add so that for each such edge (u, v), $\Pi(u) \neq \Pi(v)$ before inserting (u, v), and select a subset $E_{del} \subseteq E_{ins}$ to delete. For each graph we create 5 problems, with $|E_{del}| \in \{0, 100, 250, 500, 1000\}$. We randomly shuffle the edge updates while making sure that an edge deletion is only performed after the respective edge has been added to the graph, but still interspersing edge insertions and deletions to create true worst-case instances for the dynamic algorithm, as each edge deletion or insertion affects one or multiple minimum cuts in the graph.

Figure 4 shows the results of this experiment. Each low-alpha dot shows the speedup of the dynamic algorithm on a single problem, the black line gives the geometric mean speedup. As indicated in previous experiments, we can see that the average speedup decreases when the ratio of deletions is increased. However, even on these worst-case instances, the mean speedup factor is still 7.46x for $|E_{ins}| = |E_{del}| = 1000$ up to 79.2x for the purely incremental instances on instances where both algorithms finished before timeout at one hour. Similar to previous experiments, the speedup factor increases with the graph size.

On these problem instances we can see interesting effects. Especially in instances with $|E_{del}| = 500$ we can see many instances where the minimum cut fluctuates between two different values in more than half of all edge updates. As the larger of the values usually has a large cactus graph $\mathcal{C}$, this would result in expensive recomputation on almost every update. However, using the cactus caching technique detailed in Section 3.3.1 we can save this overhead and simply reuse the almost unchanged previous cactus graph. In some cases, this reduces the number of calls to the algorithm of Henzinger et al. [21] by more than a factor of 10.

We also find some instances where the static graph has few minimum cuts, but there is a large set of cuts slightly larger than lambda. One such example are planar graphs derived from Delaunay triangulation [35] that have a few vertices of minimal degree near the edges of the triangulated object, but a large number of vertices with a slightly larger degree. If we now add edges to increase the degree of the minimum-degree vertices, the resulting graph has a huge number of minimum cuts and computing all minimum cuts is significantly more expensive than computing just a single minimum cut. In these instances the dynamic algorithm is actually slower than rerunning the static algorithm on every edge update. The dynamic algorithm is slower than the static algorithm in 3.9% of the worst-case instances.

4.5 Most Balanced Minimum Cut Henzinger et al. [21] show that given the cactus graph $\mathcal{C}$ we can compute the most balanced minimum cut,

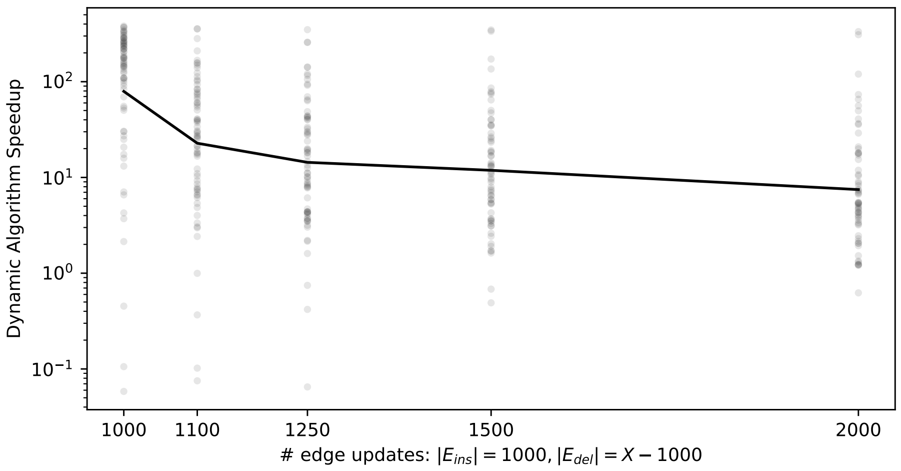

Figure 4: Speedup of Dynamic Algorithm on Worst-case Insertions and Deletions from Static Graphs.

i.e. the minimum cut which has the highest number of vertices in the smaller partition, in $\mathcal{O}(n^*)$ time. In our algorithm for the dynamic minimum cut problem we also compute a cactus graph of minimum cuts, however this cactus graph does not necessarily contain all minimum cuts in G, as we do not introduce new minimum cuts added by edge deletions.

We use the algorithm of Henzinger et al. [21] to find the most balanced minimum cut for all instances of Graph Family B every 1000 edge updates and compare it to the most balanced minimum cut found by our algorithm. In instances that are not just decremental, in 97.3% of all cases where there is a nontrivial minimum cut (i.e. smaller side contains multiple vertices), both algorithms give the same result, i.e. our algorithm can almost always output the most balanced minimum cut. In the instances that are purely decremental, i.e. $|E_{ins}| = 0$, we only find the most balanced minimum cut in 25.4% of cases where there is a non-trivial minimum cut. This is the case because an increase of the minimum cut prompts a full recomputation of a cactus graph that represents all (potentially many) minimum cuts, thus also the most balanced minimum cut. Only if this cut in particular is affected by an edge update, the dynamic algorithm "loses" it. In the purely decremental case, the minimum cut value only decreases. Thus, the dynamic algorithm only knows one or a few minimum cuts. All cuts that reach the same value λ in later edge deletions are not in $\mathcal{C}$, as we do not add cuts of the same value to it. As these decremental instances do not have any edge insertions that can increase the value of these cuts, there is eventually a large set of minimum cuts of which the algorithm only knows a few. If maintaining a balanced minimum cut is a requirement, this can easily be achieved by occasionally recomputing the entire cactus graph $\mathcal{C}$ from scratch.

5 Conclusion

In this work, we presented the first implementation of a fully-dynamic algorithm that maintains the minimum cut of a graph under both edge insertions and deletions. Our algorithm combines ideas from the theoretical foundation with efficient and fine-tuned implementations to give an algorithm that outperforms static approaches by up to five orders of magnitude on large graphs. In our experiments, we show the performance of our algorithm on a wide variety of graph instances.

Future work includes maintaining all global minimum cuts also under edge deletions and employing shared-memory or distributed parallelism to further increase the performance of our algorithm.

6 Acknowledgments

Research supported by the Austrian Science Fund (FWF) and netIDEE SCIENCE project P 33775-N. Partially supported by DFG grant SCHU 2567/1-2.

References

[1] David A Bader, Henning Meyerhenke, Peter Sanders, Christian Schulz, Andrea Kappes, and Dorothea Wagner. Benchmarking for graph clustering and partitioning. *Encyclopedia of Social Network Analysis and Mining*, pages 73–82, 2014.

[2] Nalin Bhardwaj, Antonio Molina Lovett, and Bryce Sandlund. A simple algorithm for minimum cuts in near-linear time. In Susanne Albers, editor, *17th Scandinavian Symposium and Workshops on Algorithm*

Theory, SWAT 2020, June 22-24, 2020, Tórshavn, Faroe Islands, volume 162 of *LIPIcs*, pages 12:1–12:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.SWAT.2020.12.

[3] Sebastian Böcker, Sebastian Briesemeister, and Gunnar W. Klau. Exact algorithms for cluster editing: Evaluation and experiments. *Algorithmica*, 60(2):316–334, Jun 2011. doi:10.1007/s00453-009-9339-7.

[4] Chandra S. Chekuri, Andrew V. Goldberg, David R. Karger, Matthew S. Levine, and Cliff Stein. Experimental study of minimum cut algorithms. In *Proc. 8th Symp. on Discrete Algorithms (SODA '97)*, pages 324–333. SIAM, 1997.

[5] Boris V Cherkassky and Andrew V Goldberg. On implementing the push—relabel method for the maximum flow problem. *Algorithmica*, 19(4):390–410, 1997.

[6] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Mathematical Software (TOMS)*, 38(1):1, 2011.

[7] Camil Demetrescu, David Eppstein, Zvi Galil, and Giuseppe F Italiano. Dynamic graph algorithms. In *Algorithms and theory of computation handbook: general concepts and techniques*, pages 9–9. 2010.

[8] Efim A Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. In *Soviet Math. Doklady*, volume 11, pages 1277–1280, 1970.

[9] Yefim Dinitz. Maintaining the 4-edge-connected components of a graph on-line. In *[1993] The 2nd Israel Symposium on Theory and Computing Systems*, pages 88–97. IEEE, 1993.

[10] David Eppstein, Zvi Galil, and Giuseppe F Italiano. *Dynamic graph algorithms.* Citeseer, 1998.

[11] Lester R. Ford and Delbert R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8(3):399–404, 1956.

[12] Paweł Gawrychowski, Shay Mozes, and Oren Weimann. Minimum cut in o (m $\log^2$ n) time. In *47th International Colloquium on Automata, Languages, and Programming (ICALP 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[13] Mohsen Ghaffari, Krzysztof Nowicki, and Mikkel Thorup. Faster algorithms for edge connectivity via random 2-out contractions. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1260–1279. SIAM, 2020.

[14] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM*, 35(4):921–940, 1988.

[15] Ralph E. Gomory and Tien Chung Hu. Multi-terminal network flows. *Journal of the Society for Industrial and Applied Mathematics*, 9(4):551–570, 1961.

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

[17] Jianxiu Hao and James B. Orlin. A faster algorithm for finding the minimum cut in a graph. In *Proc. of the 3rd ACM-SIAM Symp. on Discrete Algorithms*, pages 165–174. Society for Industrial and Applied Mathematics, 1992.

[18] Monika Henzinger, Alexander Noe, and Christian Schulz. Shared-memory Exact Minimum Cuts. *Proc. 33rd Intl. Parallel and Distributed Processing Symp. (IPDPS)*, 2019.

[19] Monika Henzinger, Alexander Noe, and Christian Schulz. Practical fully dynamic minimum cut algorithms. *arXiv preprint 2101.05033*, 2021.

[20] Monika Henzinger, Alexander Noe, Christian Schulz, and Darren Strash. Practical minimum cut algorithms. *ACM Journal of Experimental Algorithmics*, 23, 2018. doi:10.1145/3274662.

[21] Monika Henzinger, Alexander Noe, Christian Schulz, and Darren Strash. Finding all global minimum cuts in practice. In *28th Annual European Symposium on Algorithms (ESA 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[22] Monika Henzinger, Satish Rao, and Di Wang. Local flow partitioning for faster edge connectivity. In *Proc. of the 28th ACM-SIAM Symp. on Discrete Algorithms*, pages 1919–1938. SIAM, 2017.

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

[24] Michael Jünger, Giovanni Rinaldi, and Stefan Thienel. Practical performance of efficient minimum cut algorithms. *Algorithmica*, 26(1):172–195, 2000.

[25] Goossen Kant. *Algorithms for drawing planar graphs.* PhD thesis, 1993.

[26] David R Karger. Minimum cuts in near-linear time. *Journal of the ACM*, 47(1):46–76, 2000.

[27] David R. Karger. A randomized fully polynomial time approximation scheme for the all-terminal network reliability problem. *SIAM Review*, 43(3):499–522, 2001.

[28] David R Karger and Debmalya Panigrahi. A near-linear time algorithm for constructing a cactus representation of minimum cuts. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 246–255. SIAM, 2009.

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

[30] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

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

[32] Ken-ichi Kawarabayashi and Mikkel Thorup. Deterministic global minimum cut of a simple graph in near-linear time. In *Proc. of the 47th ACM Symp. on Theory of Computing*, pages 665–674. ACM, 2015.

[33] Balakrishnan Krishnamurthy. An improved min-cut algorithm for partitioning VLSI networks. *IEEE Trans. on Computers*, 33(5):438–446, 1984.

[34] Jakub Łącki and Piotr Sankowski. Min-cuts and shortest cycles in planar graphs in o (n loglogn) time. In *European Symposium on Algorithms*, pages 155–166. Springer, 2011.

[35] Der-Tsai Lee and Bruce J Schachter. Two algorithms for constructing a delaunay triangulation. *International Journal of Computer & Information Sciences*, 9(3):219–242, 1980.

[36] Jason Li. Deterministic mincut in almost-linear time. 2020.

[37] Jason Li and Debmalya Panigrahi. Deterministic mincut in poly-logarithmic max-flows. 2020.

[38] Danielle P Mersch, Alessandro Crespi, and Laurent Keller. Tracking individuals shows spatial fidelity is a key regulator of ant social organization. *Science*, 340(6136):1090–1093, 2013.

[39] Sagnik Mukhopadhyay and Danupon Nanongkai. Weighted min-cut: sequential, cut-query, and streaming algorithms. In Konstantin Makarychev, Yury Makarychev, Madhur Tulsiani, Gautam Kamath, and Julia Chuzhoy, editors, *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*, pages 496–509. ACM, 2020. `doi:10.1145/3357713.3384334`.

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

[41] Hiroshi Nagamochi and Tiko Kameda. Canonical cactus representation for minimum cuts. *Japan Journal of Industrial and Applied Mathematics*, 11(3):343–361, 1994.

[42] Hiroshi Nagamochi and Tiko Kameda. Constructing cactus representation for all minimum cuts in an undirected network. *Journal of the Operations Research Society of Japan*, 39(2):135–158, 1996.

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

[44] Hiroshi Nagamochi, Tadashi Ono, and Toshihide Ibaraki. Implementing an efficient minimum capacity cut algorithm. *Math. Prog.*, 67(1):325–341, 1994.

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

[46] Manfred Padberg and Giovanni Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Review*, 33(1):60–100, 1991.

[47] Aparna Ramanathan and Charles J. Colbourn. Counting almost minimum cutsets with reliability applications. *Mathematical Programming*, 39(3):253–261, 1987.

[48] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015. URL: `http://networkrepository.com`.

[49] Ryan A. Rossi and Nesreen K. Ahmed. An interactive data repository with visual analytics. *SIGKDD Explor.*, 17(2):37–41, 2016. URL: `http://networkrepository.com`.

[50] Alan J Soper, Chris Walshaw, and Mark Cross. A combined evolutionary search and multilevel optimisation approach to graph-partitioning. *Journal of Global Optimization*, 29(2):225–241, 2004.

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

[52] Aya Zaki, Mahmoud Attia, Doaa Hegazy, and Safaa Amin. Comprehensive survey on dynamic graph models. *International Journal of Advanced Computer Science and Applications*, 7(2):573–582, 2016.