Optimal Offline Dynamic 2, 3-Edge/Vertex Connectivity

Richard Peng[1], Bryce Sandlund[2(✉)], and Daniel D. Sleator[3]

[1] School of Computer Science, Georgia Institute of Technology, Atlanta, GA, USA
rpeng@cc.gatech.edu
[2] Cheriton School of Computer Science, University of Waterloo, Waterloo, ON, Canada
bcsandlund@uwaterloo.ca
[3] Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA
sleator@cs.cmu.edu

Abstract. We give offline algorithms for processing a sequence of 2- and 3-edge and vertex connectivity queries in a fully-dynamic undirected graph. While the current best fully-dynamic online data structures for 3-edge and 3-vertex connectivity require $O(n^{2/3})$ and $O(n)$ time per update, respectively, our per-operation cost is only $O(\log n)$, optimal due to the dynamic connectivity lower bound of Patrascu and Demaine. Our approach utilizes a divide and conquer scheme that transforms a graph into smaller equivalents that preserve connectivity information. This construction of equivalents is closely-related to the development of vertex sparsifiers, and shares important connections to several upcoming results in dynamic graph data structures, including online models.

1 Introduction

Dynamic graph data structures seek to answer queries on a graph as it undergoes edge insertions and deletions. Perhaps the simplest and most fundamental query to consider is connectivity. A connectivity query asks for the existence of a path connecting two vertices u and v in the current graph. As the insertion or deletion of a single edge may have large consequences to connectivity across the entire graph, constructing an efficient dynamic data structure to answer connectivity queries has been a challenge to the data structure community. A number of solutions have been developed, achieving a wide variety of runtime tradeoffs in a number of different models [10–12,18,19,22–24,30,34,36]

The models typically addressed are *online*: each query must be answered before the next query or update is given. A less demanding variant is the *offline* setting, where the entire sequence of updates and queries is provided as input to the algorithm. While an online data structure is more general, there are many scenarios in which the entire sequence of operations is known in advance. This is often the case when a data structure is used in a subroutine of an algorithm [6, 28], one specific example being the use of dynamic trees in the near-linear time minimum cut algorithm of Karger [25].

Z. Friggstad et al. (Eds.): WADS 2019, LNCS 11646, pp. 553–565, 2019.
https://doi.org/10.1007/978-3-030-24766-9_40

In exchange for the loss of flexibility, one can hope to obtain faster and simpler algorithms in the offline setting. This has been shown to be the case in the dynamic minimum spanning tree problem. While an online fully-dynamic minimum spanning tree data structure requires about $O(\log^4 n)$ time per update [20], the offline algorithm of Eppstein requires only $O(\log n)$ time per update [10].

In this paper, we show similar, but stronger, performance gains for higher versions of connectivity. In particular, we consider the problems of 2, 3-edge/vertex connectivity on a fully-dynamic undirected graph. An extension of connectivity, c-edge connectivity asks for the existence of c edge-disjoint paths between two vertices u and v in the current graph. Vertex connectivity requires vertex-disjoint paths instead of edge-disjoint paths. Current online fully-dynamic 2-edge/vertex connectivity data structures require update time $\tilde{O}(\log^2 n)$[1] [19] and $\tilde{O}(\log^3 n)$[2] [34], respectively, and current online fully-dynamic 3-edge/vertex connectivity data structures require update time $O(n^{2/3})$ and $O(n)$, respectively [11]. In contrast, our offline algorithms for 2, 3-edge/vertex connectivity require only $O(\log n)$ time per operation. As the lower bound on dynamic connectivity [32], as well as most lower bounds in general [1–3, 7], also apply in the offline model, our algorithms are optimal up to a constant factor. This paper further shows that any lower bound attempting to show hardness stronger than $\Omega(\log n)$ time per operation for online fully-dynamic 2, 3-edge/vertex connectivity must make use of the online model.

As a straightforward application of our results, one can consider the use of our algorithms when data regarding a dynamic network is collected, but not analyzed, until a later point in time. For example, to diagnose an issue of network latency across key routing hubs, or determine viability of a dynamically-changing network of roads, our algorithms can answer a batch of queries in time $O(t \log n)$, where t is the total number of updates and queries. Since online fully-dynamic algorithms for higher versions of connectivity are significantly slower, namely, $O(n^{2/3})$ and $O(n)$ time update for 3-edge and 3-vertex connectivity, respectively, our offline data structure makes these computations practical for large data sets when they would otherwise be prohibitively expensive.

Related to our work are papers by Łącki and Sankowski [30] and Karczmarz and Łącki [24], which also apply to the above applications but for lower versions of connectivity. Their work considers a fixed sequence of graph updates, given in advance, and is then able to answer connectivity queries regarding intervals of this sequence, online. This is more general than the model we consider because the queries need not be supplied in advance and data regarding an interval of time is richer than information from a specific point in time. For connectivity and 2-edge connectivity, Karczmarz and Łącki achieve $O(\log n)$ time per operation [24]. Both 2-vertex connectivity and 3-edge/vertex connectivity queries are not supported.

[1] The $\tilde{O}(\cdot)$ notation hides $\log \log n$ factors.

[2] This complexity is claimed in Thorup's STOC 2000 [34] result. As noted by Huang et al. [22], the paper provides few details, deferring to a journal version that has since not appeared. The best complexity for online fully-dynamic biconnectivity prior to this claim was $O(\log^5 n)$ by Holm and Thorup [18].

In competitive programming, the idea of using divide and conquer as an offline algorithm for connectivity is known. The authors are aware of several contest problems[3] that are solved with similar techniques to Eppstein's minimum spanning tree algorithm [10], as we do here. The master's thesis of Sergey Kopeliovich, a member of the competitive programming community, describes such an offline algorithm for fully-dynamic 2-edge connectivity, also achieving about $O(\log n)$ time per operation [26]. Unfortunately, the thesis only appears in Russian, but we speculate that the ideas used are similar.

The techniques developed in this paper may be of independent interest. Our work has close connections with recent developments in vertex sparsification, particularly vertex sparsification-based dynamic graph data structures [4,5,8,9,12–17,27]. In particular, the equivalent graphs at the core of our algorithms are akin to vertex sparsifiers, with the main difference that 2- and 3-connectivity require preserving far less information than the more general definitions of vertex sparsifiers [15,27]. A promising step in this direction is very recent work of Goranci et al. [17], which suggests the notion of a *local sparsifier*. This is a generalization of the sparsifier that we consider here, and leads to efficient incremental algorithms in the *online* setting.

Indeed, offline algorithms haven proven useful for the development of online counterparts in the past. One such example is recent development in the maintenance of dynamic effective resistance. Recent work in fully-dynamic data structures for maintaining effective resistances online [8] relied heavily upon ideas from earlier data structures for maintaining effective resistances in offline [9,29] or offline-online hybrid [9,28] settings.

The results of this paper were previously published online in the open access journal arXiv [33] and have recently been extended to offline 4- and 5-edge connectivity [31]. This new work achieves about $O(\sqrt{n})$ time complexity per operation.

The rest of this paper will be dedicated to proving the following theorem:

Theorem 1. *Given a sequence of t updates/queries on a graph of the form:*

- *Insert edge (u, v),*
- *Delete edge (u, v),*
- *Query if a pair of vertices u and v are 2-edge connected/3-edge connected/biconnected/tri-connected in the current graph,*

there exists an algorithm that answers all queries in $O(t \log n)$ time.

For simplicity, we will assume the graph is empty at the start and end of the sequence, but the results discussed are easily modified to start with an initial graph G, at the cost of an additive $O(m)$ term in the running time, where m is

[3] See https://codeforces.com/blog/entry/15296 and https://codeforces.com/gym/100551/problem/A, for example.

the number of edges of G. Further, we assume a fixed vertex set of size n. Any update sequence with arbitrary vertex endpoints can be modified to one on a fixed set of vertices, where the size of the fixed set is equal to the largest number of non-isolated vertices in any graph achieved in the given update sequence. Finally, we consider the graph G to be a multigraph, since at times during our constructions and definitions, we will need to work with multigraphs.

The paper is organized as follows. We describe our offline framework for reducing graphs to smaller equivalents in Sect. 2. We show how simple techniques can be used to create such equivalents for 2-edge connectivity and bi-connectivity in Sect. 3. In Sect. 4 we extend these constructions to 3-edge connectivity. Our most technical section is 3-vertex connectivity, where constructing equivalent graphs requires careful manipulation of SPQR trees. Unfortunately, due to page limits, this will only appear in the full version of this paper.

2 Offline Framework

The main idea of our offline framework is to perform divide and conquer on the input sequence, similar to what is done in Eppstein's offline minimum spanning tree algorithm [10]. Consider the full sequence of updates and queries $x_1, \ldots, x_t$, where each x_i is either an edge insertion, edge deletion, or query. Call each x_i an event.

Assume each inserted edge has a unique identity. Then for each inserted edge e, we may associate an interval $[I(e), D(e)]$, indicating that edge e was inserted at time $I(e)$ and removed at time $D(e)$. Plotting time along the x-axis and edges on the y-axis as in Fig. 1 gives a convenient way to view the sequence of events.

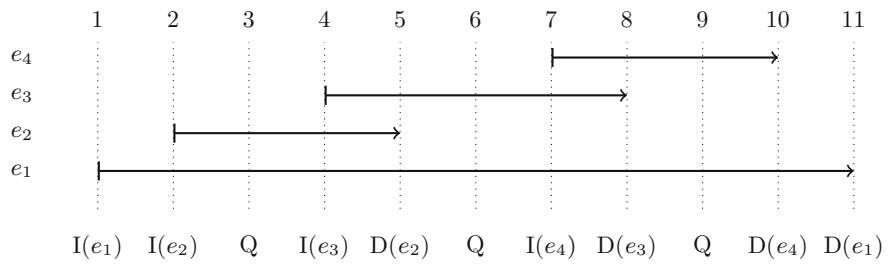

Fig. 1. A timeline diagram of four edge insertions(I)/deletions(D) and three queries(Q), with time on the x-axis and edges on the y-axis.

Fix some subinterval $[l, r]$ of the sequence of events. Let us classify all edges present at any point of time in the sequence $x_l, \ldots, x_r$ as one of two types.

1. Edges present throughout the duration $(i_e \leq l \leq r \leq d_e)$, we call *permanent* edges.
2. Edges affected by an event in this range (one or both of $I(e), D(e)$ is in (l, r)), we call *non-permanent* edges.

While there may be a large number of permanent edges, the number of non-permanent edges is limited by the number of time steps, $r - l + 1$. Therefore, the graph can be viewed as a large static graph on which a smaller number of events take place.

Our goal will be to reduce this graph of permanent edges to one whose size is a small function of the number of events in the subinterval. If we may do so without affecting the answers to the queries, we can recursively apply the technique to achieve an efficient divide and conquer algorithm for the original dynamic c-connectivity sequence.

We will work in the dual-view, considering cuts instead of edge-disjoint or vertex-disjoint paths. Two vertices u and v are c-edge connected if there does not exist a cut of $c-1$ edges separating them; further, u and v are c-vertex connected if there does not exist a cut of $c-1$ vertices that separates them.

We need the following definition.

Definition 1. *Given a graph $G = (V_G, E_G)$ with vertex subset $W \subseteq V_G$ and a graph $H = (V_H, E_H)$ with $W \subseteq V_H$, we say that H and G are c-edge equivalent if, for any partition (A, B) of W, the size of a minimum cut separating A and B is the same in G and H whenever either of these sizes is less than c. Similarly, we say H and G are c-vertex equivalent if, for any partition (A, B, C) of W with $|C| < c$, the size of a minimum vertex cut D separating A and B such that $C \subseteq D$ and $D \cap A = \emptyset$, $D \cap B = \emptyset$, is the same in G and H whenever either of these sizes is less than c.*

This gives the following.

Lemma 1. *Suppose $G = (V_G, E_G)$ and $H = (V_H, E_H)$ are c-edge/c-vertex equivalent on vertex set W. Let E_W denote any set of edges between vertices of W. Then $H' = (V_H, E_H \cup E_W)$[4] and $G' = (V_G, E_G \cup E_W)$ are c-edge/c-vertex equivalent.*

Proof. We first show c-edge equivalence. Let (A, B) be any partition of W and consider the minimum cuts separating A and B in G' and H'. Since the edges in E_W are between vertices of W, they must cross the separation (A, B) in the same way. Therefore, if the minimum cut separating A and B had size less than c in either G or H, the minimum cuts separating A and B will have equivalent size in G' and H'. Further, if the minimum cuts separating A and B had size larger or equal to c in both G and H, the minimum cuts separating A and B will also have size larger or equal to c in G' and H', since we only add edges to G' and H'. Thus G' and H' are c-edge equivalent.

We now consider c-vertex equivalence. Consider a partition (A, B, C) of W. As with edge connectivity, if no vertex subset D exists satisfying the conditions of Definition 1, the introduction of additional edges between any vertices of W will not change the existence of such a set D in G' or H'. Furthermore, if an

[4] We take $\cup$ here to be in the multigraph sense; an edge $uv \in E_W$ is added regardless if there is already a uv edge in E_H or E_G.

edge of E_W connects a vertex of A to a vertex of B, no vertex cut separates A and B in G' and H'. Now suppose none of these cases is true, and there exists a vertex set D satisfying the conditions of Definition 1 such that the removal of D disconnects A and B in G and H and further that no edge of E_W connects a vertex of A to a vertex of B. Then the removal of vertex set D still disconnects A and B in G' and H'. Thus c-vertex equivalence of G' and H' follows from c-vertex equivalence of G and H.

Now consider the graph G of permanent edges for the subinterval $x_l, \ldots, x_r$ of events. Let W be the set of vertices involved in any event in the subinterval (that is, W is the set of endpoints of all non-permanent edges in the subinterval, as well as vertices involved in a query). We will refer to these vertices as *active* vertices, and all other vertices of G not in W as *inactive* vertices. Lemma 1 says that if we reduce G to a c-edge/ c-vertex equivalent graph H on set W, the result of all queries in $x_l, \ldots, x_r$ on H will be the same as on G. This is because all cuts in H and G that affect the queries (therefore of size less than c) are of equivalent size, even after the addition of non-permanent edges in H and G.

This idea can lead to a divide and conquer algorithm if we can produce such equivalent graphs H of small size efficiently. Specifically:

Lemma 2. *Given a graph G with m edges and vertex set W of size k, if there is an $O(m)$ time algorithm that produces a graph H of size $O(k)$ that is c-edge/c-vertex equivalent to G on W, then there is an algorithm that can answer all c-edge/c-vertex connectivity queries in a sequence of events $x_1, \ldots, x_t$ in $O(t \log n)$ time.*

Proof. We perform divide and conquer on the sequence of events. We take the sequence of events $x_1, \ldots, x_t$ and divide it in half. Over each half, we will take the graph of permanent edges, which we denote G, and reduce it to a c-edge/ c-vertex equivalent graph H. We repeat the scheme recursively. As the subintervals get smaller, non-permanent edges become permanent and are absorbed by the production of equivalent graphs. Eventually, we reduce to subintervals with a constant number of events, which can be answered by any algorithm of our choice on a graph of constant size.

Consider the sizes of the graphs in each step of recursion. The graph G is the graph produced in the previous level plus the edges that became permanent in this interval. The graph produced at the previous level has size linear in the number of events at the current level, and since we reduce the number of events by a factor 2 in each step of recursion, the number of edges that become permanent is also linear. It follows that the divide and conquer satisfies the recurrence $T(t) = 2T(t/2) + O(t)$, which solves to $T(t) = O(t \log t)$. If t is polynomially-bounded by n, $T(t) = O(t \log n)$. If not, we may first break the sequence of events $x_1, \ldots x_t$ into blocks of size, say, n^2. Since the size of the graph G cannot be more than $O(n^2)$ in any subinterval, we can therefore handle each block separately and answer all queries in $O(t \log n)$ time. This proves the lemma.

The remainder of the paper will show the construction of 2-edge, 2-vertex, 3-edge, and 3-vertex equivalent graphs.

3 Equivalent Graphs for 2-Edge Connectivity and Bi-connectivity

We now show offline algorithms for dynamic 2-edge connectivity and bi-connectivity by constructing 2-edge and 2-vertex equivalents needed by Lemma 2. These two properties ask for the existence of a single edge/vertex whose removal separates query vertices u and v. Since these cuts can affect at most one connected component, it suffices to handle each component separately.

The underlying structure for 2-edge connectivity and bi-connectivity is tree-like. This is perhaps more evident for 2-edge connectivity, where vertices on the same cycle belong to the same 2-edge connected component. We will first describe the reductions that we will make to this tree in Sect. 3.1, and adapt them to bi-connectivity in Sect. 3.2.

At times we will make use of the term "equivalent cut". By this we mean that a cut C' is equivalent to C if it has the same size and separates the vertices of W in the same way.

3.1 2-Edge Connectivity

Using depth-first search [21], we can identify all cut-edges in the graph and the 2-edge connected components that they partition the graph into. The case of edge cuts is slightly simpler conceptually, since we can combine vertices without introducing new cuts. Specifically, we show that each 2-edge connected component can be shrunk to single vertex.

Lemma 3. *Let S be a 2-edge connected component in G. Then contracting all vertices in S to a single vertex s in H[5], and endpoints of edges correspondingly, creates a 2-edge equivalent graph.*

Proof. The only cuts that we need to consider are ones that remove cut edges in G or H. Since we only contracted vertices in a component, there is a one-to-one mapping of these edges from G to H. Since S is 2-edge connected, all vertices in it will be on the same side of one of these cuts. Furthermore, removing the same edge in H leads to a cut with s instead. Therefore, all active vertices in S are mapped to s, and are therefore on the same side of the cut.

This allows us to reduce G to a tree H, but the size of this tree can be much larger than k. Therefore we need to prune the tree by removing inactive leaves and length 2 paths whose middle vertex is inactive.

[5] Here we slightly abuse our requirement $W \subseteq V_H$, where V_H are the vertices of H. A map of W onto V_H that preserves the cuts needed by c-edge/c-vertex equivalence suffices.

Lemma 4. *If G is a tree, the following two operations lead to 2-edge equivalent graphs H.*

- *Removing an inactive leaf.*
- *Removing an inactive vertex with degree 2 and adding an edge between its two neighbors.*

Proof. In the first case, the only cut in G that no longer exist in H is the one that removes the cut edge connecting the leaf with its unique neighbor. However, this places all active vertices in one component and thus does not separate W and need not be represented in H.

In the second case, if a cut removes either of the edges incident to the degree 2 vertex, removing the new edge creates an equivalent cut since the middle vertex is inactive. Also, for a cut that removes the new edge in H, removing either of the two original edges in G leads to an equivalent cut.

This allows us to bound the size of the tree by the number of active vertices, and therefore finish the construction.

Lemma 5. *Given a graph G with m edges and k active vertices W, a 2-edge equivalent of G of size $O(k)$, H, can be constructed in $O(m)$ time.*

Proof. We can find all the cut edges and 2-edge connected components in $O(m)$ time using depth-first search [21], and reduce the resulting structure to a tree H using Lemma 3. On H, we repeatedly apply Lemma 4 to obtain H'.

In H', all leaves are active, and any inactive internal vertex has degree at least 3. Therefore the number of such vertices can be bounded by $O(k)$, giving a total size of $O(k)$.

3.2 Bi-connectivity

All cut-vertices (articulation points) can also be identified using DFS, leading to a structure known as the block-tree. However, several modifications are needed to adapt the ideas from Sect. 3.1. The main difference is that we can no longer replace each bi-connected component with a single vertex in H, since cutting such vertices corresponds to cutting a much larger set in G. Instead, we will need to replace the bi-connected components with simpler bi-connected graphs such as cycles.

Lemma 6. *Replacing a bi-connected component with a cycle containing all its cut-vertices and active vertices gives a 2-vertex equivalent graph.*

Proof. As this mapping maintains the bi-connectivity of the component, it does not introduce any new cut-vertices. Therefore, G and H have the same set of cut vertices and the same block-tree structure. Note that the actual order the active vertices appear in does not matter, since they will never be separated. The claim follows similarly to Lemma 3.

The block-tree also needs to be shrunk in a similar manner. Note that the fact that blocks are connected by shared vertices along with Lemma 6 implies the removal of inactive leaves. Any leaf component with no active vertices aside from its cut vertex can be reduced to the cut vertex, and therefore be removed. The following is an equivalent of the degree two removal part of Lemma 4.

Lemma 7. *Two bi-connected components C_1 and C_2 with no active vertices that share cut vertex w and are only incident to one other cut vertex each, u and v respectively, can be replaced by an edge connecting u and v to create a 2-vertex equivalent graph.*

Proof. As we have removed only w, any cut vertex in H is also a cut vertex in G. As C_1 and C_2 contain no active vertices, this cut would induce the same partition of active vertices.

For the cut given by removing w in G, removing u in H gives the same cut since C_1 has no active vertices (which in turn implies that u is not active). Note that the removal of u may break the graph into more pieces, but our definition of cuts allows us to place these pieces on two sides of the cut arbitrarily.

Note that Lemma 6 may need to be applied iteratively with Lemma 7 since some of the cut vertices may no longer be cut vertices due to the removal of components attached to them.

Lemma 8. *Given a graph G with m edges and k active vertices W, a 2-vertex equivalent of G of size $O(k)$, H, can be constructed in $O(m)$ time.*

Proof. We can find all the initial block-trees using depth-first search [21]. Then we can apply Lemmas 6 and 7 repeatedly until no more reductions are possible. Several additional observations are needed to run these reduction steps in $O(m)$ time. As each cut vertex is removed at most once, we can keep a counter in each component about the number of cut vertices on it. Also, the second time we run Lemma 6 on a component, it's already a cycle, so the reductions can be done without examining the entire cycle by tracking it in a doubly linked list and removing vertices from it.

It remains to bound the size of the final block-tree. Each leaf in the block-tree has at least one active vertex that's not its cut vertex. Therefore, the block-tree contains at most $O(k)$ leaves and therefore at most $O(k)$ internal components with 3 or more cut vertices, as well as $O(k)$ components containing active vertices. If these components are connected by paths with 4 or more blocks in the block tree, then the two middle blocks on this path meet the condition of Lemma 7 and should have been removed by the above procedure. This gives a bound of $O(k)$ on the number of blocks, which in turn implies an $O(k)$ bound on the number of cut vertices. The edge count then follows from the fact that Lemma 6 replaces each component with a cycle, whose number of edges is linear in the number of vertices, and that the bi-connected components themselves are arranged in a tree.

4 3-Edge Connectivity

We now extend our algorithms to 3-edge connectivity. Our starting point is a statement similar to Lemma 3, namely that we can contract all 3-edge connected components. Though of no consequence to our algorithms, we note that unlike 2-edge or biconnected components, 3-edge connected components need not be connected.

Lemma 9. *Let S be a 3-edge connected component in G. Then contracting all vertices in S to a single vertex s in H, and endpoints of edges correspondingly, creates a 3-edge equivalent graph.*

Proof. A two-edge cut will not separate a 3-edge connected component. Therefore all active vertices in S fall on one side of the cut, to which vertex s may also fall. The proof follows analogously to Lemma 3.

Such components can also be identified in $O(m)$ time using depth-first search [35], so the preprocessing part of this algorithm is the same as with the 2-connectivity cases. However, the graph after this shrinking step is no longer a tree. Instead, it is a cactus, which in its simplest terms can be defined as:

Definition 2. *A cactus is an undirected graph where each edge belongs to at most one cycle.*

On the other hand, cactuses can also be viewed as a tree with some of the vertices turned into cycles[6]. Such a structure essentially allows us to repeat the same operations as in Sect. 3 after applying the initial contractions.

Lemma 10. *A connected undirected graph with no nontrivial 3-edge connected component is a cactus.*

Due to space restrictions, we save the proof for Appendix A.

With this structural statement, we can then repeat the reductions from the 2-edge equivalent algorithm from Sect. 3.1 to produce the 3-edge equivalent graph.

Lemma 11. *Given a graph G with m edges and k active vertices W, a 3-edge equivalent of G of size $O(k)$, H, can be constructed in $O(m)$ time.*

Proof. Lemma 10 means that we can reduce the graph to a cactus after $O(m)$ time preprocessing.

First consider the tree where the cycles are viewed as vertices. Note that in this view, a vertex that's not on any cycle is also viewed as a cycle of size 1. This can be pruned in a manner analogous to Lemma 4:

1. Cycles containing no active vertices and incident to 1 or 2 other cycles can be contracted to a single vertex.
2. Inactive single-vertex cycles incident to 1 other cycle can be removed.

[6] Some 'virtual' edges are needed in this construction, because a vertex can still belong to multiple cycles.

This procedure takes $O(m)$ time and produces a graph with at most $O(k)$ leaves. Correctness of the first rule follows by replacing a cut of the two edges within an inactive cycle by a cut of the single contracted vertex with one of its neighbors. The second rule does not affect any cuts separating W. It remains to reduce the length of degree 2 paths and the sizes of the cycles themselves.

As in Lemma 4, all inactive vertices of degree 2 can be replaced by an edge between its two neighbors. This bounds the length of degree 2 paths and reduces the size of each cycle to at most twice its number of incidences with other cycles. This latter number is in turn bounded by the number of leaves of the tree of cycles. Hence, this contraction procedure reduces the total size to $O(k)$.

We remark that this is not identical to iteratively removing inactive vertices of degrees at most 2. With that rule, a cycle can lead to a duplicate edge between pairs of vertices, and a chain of such cycles needs to be reduced in length.

A Omitted Proofs

Proof (Proof of Lemma 10). We prove by contradiction. Let G be a graph with no nontrivial 3-edge connected component. Suppose there exists two simple cycles a and b in G with more than one vertex, and thus at least one edge, in common.

Call the vertices in the first simple cycle $a_1, \ldots, a_n$ and the second simple cycle $b_1, \ldots, b_m$, in order along the cycle.

Since these cycles are not the same, there must be some vertex not common to both cycles. Without loss of generality, assume (by flipping a and b) that b is not a subset of a, and (by shifting b cyclically) that b_1 is only in b and not a.

Now let b_{first} be the first vertex after b_1 in b that is common to both cycles, so

$$first \stackrel{\text{def}}{=} \min_i b_i \in a. \tag{1}$$

and let b_{last} be the last vertex in b common to both cycles

$$last \stackrel{\text{def}}{=} \max_i b_i \in a. \tag{2}$$

The assumption that these two cycles have more than 1 vertex in common means that

$$first < last. \tag{3}$$

We claim b_{first} and b_{last} are 3-edge connected.

We show this by constructing three edge-disjoint paths connecting b_{first} and b_{last}. Since both b_{first} and b_{last} occur in a, we may take the two paths formed by cycle a connecting b_{first} and b_{last}, which are clearly edge-disjoint.

By construction, vertices

$$b_{last+1}, \ldots, b_m, b_1, \ldots, b_{first-1} \tag{4}$$

are not shared with a. Thus they form a third edge-disjoint path connecting b_{first} and b_{last}, and so the claim follows. Therefore, a graph with no 3-edge connected vertices, and thus no nontrivial 3-edge connected component has the property that two simple cycles have at most one vertex in common.

References

1. Abboud, A., Dahlgaard, S.: Popular conjectures as a barrier for dynamic planar graph algorithms. In: IEEE 57th Annual Symposium on Foundations of Computer Science, pp. 477–486, Nov 2016
2. Abboud, A., Williams, V.V.: Popular conjectures imply strong lower bounds for dynamic problems. In: Proceedings of the 2014 IEEE 55th Annual Symposium on Foundations of Computer Science, FOCS 2014, pp. 434–443 (2014)
3. Abboud, A., Williams, V.V., Yu, H.: Matching triangles and basing hardness on an extremely popular conjecture. In: Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, pp. 41–50 (2015)
4. Abraham, I., Durfee, D., Koutis, I., Krinninger, S., Peng, R.: On fully dynamic graph sparsifiers. In: 2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS), pp. 335–344, Oct 2016
5. Assadi, S., Khanna, S., Li, Y., Tannen, V.: Dynamic sketching for graph optimization problems with applications to cut-preserving sketches. In: FSTTCS (2015)
6. Bringmann, K., Kunnemann, M., Nusser, A.: Frechet distance under translation: conditional hardness and an algorithm via offline dynamic grid reachability. CoRR abs/1810.10982 (2018). https://arxiv.org/abs/1810.10982
7. Dahlgaard, S.: On the hardness of partially dynamic graph problems and connections to diameter. In: 43rd International Colloquium on Automata, Languages, and Programming (2016)
8. Durfee, D., Gao, Y., Goranci, G., Peng, R.: Fully dynamic spectral vertex sparsifiers and applications. In: Proceedings of the thirtieth annual ACM symposium on Theory of computing, STOC 2019. ACM (2019)
9. Durfee, D., Kyng, R., Peebles, J., Rao, A.B., Sachdeva, S.: Sampling random spanning trees faster than matrix multiplication. In: Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, pp. 730–742 (2017)
10. Eppstein, D.: Offline algorithms for dynamic minimum spanning tree problems. J. Algorithms **17**(2), 237–250 (1994)
11. Eppstein, D., Galil, Z., Italiano, G.F., Nissenzweig, A.: Sparsification-a technique for speeding up dynamic graph algorithms. J. ACM **44**(5), 669–696 (1997)
12. Eppstein, D., Galil, Z., Italiano, G.F., Spencer, T.H.: Separator-based sparsification II: edge and vertex connectivity. SIAM J. Comput. **28**(1), 341–381 (2006)
13. Fafianie, S., Hols, E.M.C., Kratsch, S., Quyen, V.A.: Preprocessing under uncertainty: matroid intersection. In: 41st International Symposium on Mathematical Foundations of Computer Science, MFCS 2016, vol. 58, pp. 35:1–35:14 (2016)
14. Fafianie, S., Kratsch, S., Quyen, V.A.: Preprocessing under uncertainty. In: 33rd Symposium on Theoretical Aspects of Computer Science, STACS 2016, vol. 47, pp. 33:1–33:13 (2016)
15. Goranci, G., Henzinger, M., Peng, P.: The power of vertex sparsifiers in dynamic graph algorithms. In: European Symposium on Algorithms (ESA), pp. 45:1–45:14 (2017)
16. Goranci, G., Henzinger, M., Peng, P.: Dynamic effective resistances and approximate schur complement on separable graphs. In: 26th Annual European Symposium on Algorithms, ESA 2018, vol. 112, pp. 40:1–40:15 (2018)
17. Goranci, G., Henzinger, M., Saranurak, T.: Fast incremental algorithms via local sparsifiers. CoRR (2018). https://drive.google.com/file/d/1SJrbzuz_szMwsBfeBZfGUWkDEbZKAGD5/view

18. Holm, J., de Lichtenberg, K., Thorup, M.: Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. In: Proceedings of the thirtieth annual ACM symposium on Theory of computing, STOC 1998, pp. 79–89. ACM, New York (1998)
19. Holm, J., Rotenberg, E., Thorup, M.: Dynamic bridge-finding in $\tilde{O}(\log^2 n)$ amortized time. In: Symposium on Discrete Algorithms (SODA) (2018)
20. Holm, J., Rotenberg, E., Wulff-Nilsen, C.: Faster fully-dynamic minimum spanning forest. In: Bansal, N., Finocchi, I. (eds.) ESA 2015. LNCS, vol. 9294, pp. 742–753. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48350-3_62
21. Hopcroft, J.E., Tarjan, R.E.: Dividing a graph into triconnected components. SIAM J. Comput. **2**(3), 135–158 (1973)
22. Huang, S.E., Huang, D., Kopelowitz, T., Pettie, S.: Fully dynamic connectivity in $O(\log n(\log \log n)^2)$ amortized expected time. In: Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 510–520 (2017)
23. Kapron, B., King, V., Mountjoy, B.: Dynamic graph connectivity in polylogarithmic worst case time. In: SODA (2013)
24. Karczmarz, A., Łącki, J.: Fast and simple connectivity in graph timelines. In: Dehne, F., Sack, J.-R., Stege, U. (eds.) WADS 2015. LNCS, vol. 9214, pp. 458–469. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21840-3_38
25. Karger, D.R.: Minimum cuts in near-linear time. J. ACM **47**(1), 46–76 (2000)
26. Kopeliovich, S.: Offline solution of connectivity and 2-edge-connectivity problems for fully dynamic graphs. Master's thesis, Saint Petersburg State University (2012)
27. Kratsch, S., Wahlstrom, M.: Representative sets and irrelevant vertices: New tools for kernelization. In: Proceedings of the 2012 IEEE 53rd Annual Symposium on Foundations of Computer Science, FOCS 2012, pp. 450–459 (2012)
28. Li, H., Patterson, S., Yi, Y., Zhang, Z.: Maximizing the number of spanning trees in a connected graph. CoRR abs/1804.02785 (2018). https://arxiv.org/abs/1804.02785
29. Li, H., Zhang, Z.: Kirchhoff index as a measure of edge centrality in weighted networks: nearly linear time algorithms. In: Symposium on Discrete Algorithms (SODA), pp. 2377–2396 (2018)
30. Łącki, J., Sankowski, P.: Reachability in graph timelines. In: ITCS (2013)
31. Molina, A., Sandlund, B.: Personal communication
32. Patracscu, M., Demaine, E.D.: Lower bounds for dynamic connectivity. In: Proceedings of the thirty-sixth annual ACM symposium on Theory of computing, STOC 2004, pp. 546–553. ACM, New York (2004)
33. Peng, R., Sandlund, B., Sleator, D.D.: Offline dynamic higher connectivity. CoRR abs/1708.03812 (2017). http://arxiv.org/abs/1708.03812
34. Thorup, M.: Near-optimal fully-dynamic graph connectivity. In: Proceedings of the Thirty-Second Annual ACM Symposium on Theory of omputing, STOC 2000, pp. 343–350, ACM, New York (2000)
35. Tsin, Y.H.: Yet another optimal algorithm for 3-edge-connectivity. J. Discrete Algorithms **7**(1), 130–146 (2009)
36. Wulff-Nilsen, C.: Fully-dynamic minimum spanning forest with improved worst-case update time. In: Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC, pp. 1130–1143 (2017)