Dynamic graph connectivity in polylogarithmic worst case time

Bruce M. Kapron * Valerie King * Ben Mountjoy *

Abstract
The dynamic graph connectivity problem is the following: given a graph on a fixed set of \( n \) nodes which is undergoing a sequence of edge insertions and deletions, answer queries of the form \( q(a,b) \): “Is there a path between nodes \( a \) and \( b \)?” While data structures for this problem with polylogarithmic amortized time per operation have been known since the mid-1990’s, these data structures have \( \Theta(n) \) worst case time. In fact, no previously known solution has worst case time per operation which is \( o(\sqrt{n}) \).

We present a solution with worst case times \( O(\log^4 n) \) per edge insertion, \( O(\log^5 n) \) per edge deletion, and \( O(\log n / \log \log n) \) per query. The answer to each query is correct if the answer is “yes” and is correct with high probability if the answer is “no”. The data structure is based on a simple novel idea which can be used to quickly identify an edge in a cutset.

Our technique can be used to simplify and significantly speed up the preprocessing time for the emergency planning problem while matching previous bounds for an update, and to approximate the sizes of cutsets of dynamic graphs in time \( \tilde{O}(\min\{|S|, |V \setminus S|\}) \) for an oblivious adversary.

1 Introduction
The dynamic connectivity problem is as follows: We are given an undirected graph \( G = (V,E) \), where \( V \) is a fixed set of \( n \) nodes, and an online sequence of updates and queries, of the following form:

- Delete(\( e \)): Delete edge \( e \) from \( E \).
- Insert(\( e \)): Insert edge \( e \) into \( E \).
- Query(\( a, b \)): Is there a path between nodes \( a \) and \( b \)?

The goal is to build a data structure so as to minimize the cost per operation. The previously known state of the art for this problem is \( O(\sqrt{n}) \) worst case update time, due to Frederickson (1985) [5], as improved from \( O(\sqrt{n}) \) through use of the sparsification technique of the Eppstein, et.al. [4]. In the 1990’s data structures with polylogarithmic amortized update time were developed, but these algorithms have worst case time \( \tilde{O}(n) \). Each of these algorithms have query time \( \tilde{O}(1) \).

Main Result: For the dynamic connectivity problem, our data structure is initialized in time \( O(n \log^4 n + m \log^3 n) \) where \( n \) is the number of nodes and \( m \) is the number of edges in the initial graph. Each insertion of an edge requires worst case \( O(\log^4 n) \) time; each deletion of an edge requires worst case \( O(\log^5 n) \) time. Each query requires \( O(\log n / \log \log n) \) worst case time. If the answer to a query is “yes,” then the answer is always correct; if it is “no,” then it is correct with probability \( 1 - 1/n^c \) for \( c \) any constant. We assume the adversary knows the edges in the graph.
but constructs the sequence of updates and queries with no knowledge of the random bits used by the algorithm, other than whether the queries have been answered correctly or not. The edges of the spanning forest used by the data structure are not revealed to the adversary.

**Other results:** Several dynamic graph problems can easily be reduced to the problem of maintaining connectivity. See [6]. Thus our algorithm yields a dynamic graph algorithm for determining the weight of a minimum spanning tree in a graph with $k$ different edge weights, where the cost of updates is increased by a factor of $k$. It also can be used to maintain the property of bipartiteness with only constant factor increase in cost. See [1].

The emergency planning problem is to preprocess a graph so that after a single batch of updates, queries about the connectivity of the resulting graph can be answered quickly. The emergency planning problem can be solved by using any dynamic connectivity data structure with good worst case time performance. We can show that for this particular problem our Monte Carlo data structure can be streamlined to match or even improve upon the update costs of [11] but uses a much faster and simpler preprocessing method.

Given a fully dynamic graph, and a cutset determined by a partition of the nodes $(S, V \setminus S)$, a version of our data structure can also be used to approximate w.h.p., to within a factor of 2, the size of the cutset in time $\tilde{O}(\min\{|S|, |V \setminus S|\})$ with an oblivious adversary. By oblivious, we mean an adversary whose choices do not depend on previous answers.

In Section 2, we give related work. In Section 3 we give the basic data structure for finding an edge in a cutset. In Section 4, we show how it can be used to solve dynamic connectivity. In Section 5, we give the other applications. A discussion follows in Section 6.

### 2 Related work

**Dynamic connectivity:** In 1985, Frederickson introduced topology trees to give a worst case update time of $O(\sqrt{m})$ [5]. His data structure is based on an initial decomposition of a spanning forest into a hierarchical structure of subtrees, in which an edge is maintained between pairs of subtrees and extra nodes are added so that each node has small degree. A 1992 sparsification technique of Eppstein, et. al., [3], followed by an improved 1997 version [4], improved the update time of topology trees to $O(\sqrt{n})$. Henzinger and King [6] gave the first polylogarithmic amortized expected update time $O(\log^2 n)$. Their data structure constructs a decomposition of the graph dynamically as either replacement edges are found or sparse cuts are detected. Henzinger and Thorup [7] improved this to $O(\log^2 n)$ expected time. In 1998, Holm, de Lichtenberg and Thorup [8] gave a deterministic data structure which constructs a different decomposition of the graph dynamically, in $O(\log^2 n)$ per update. In 2000, Thorup [15] gave a randomized data structure with $O(\log n (\log \log n)^3)$ amortized update time. All of these algorithms have $O(1)$ worst case query time. All the amortized polylogarithmic time algorithms have instances in which a single update takes $\Theta(n)$ time.

**ET-trees** were first presented by Henzinger and King in [6] and are perhaps the simplest method of maintaining dynamic trees for the most of this paper. Other dynamic tree data structures such as top trees can be used as well. A 2010 study by Tarjan and Werneck compares performance of dynamic trees [14].

There is a lower bound for randomized dynamic connectivity (Demaine and Pătraşcu [12]) which gives tradeoffs between the amortized running time of an update and the amortized running time of a query including the case of Monte Carlo algorithms like ours with high probability of success. If $t_u$ is the amortized time of an update and $t_q$ is the amortized time of a query then $t_u \cdot \lg(t_u/t_q) = \Omega(\lg n)$ and $t_u \cdot \lg(t_q/t_u) = \Omega(\lg n)$.

**Emergency Planning:** The emergency planning problem, posed by Pătraşcu and Thorup [11], is to preprocess a graph so that after a single batch of updates, queries about the connectivity of the resulting graph can be answered quickly. They give a deterministic algorithm which makes use of a “polynomial” number of calls to a weighted sparsest cut algorithm in the preprocessing phase. Since this problem is NP-Hard, they rely on $\gamma$-approximation algorithms for preprocessing. $O(\gamma d \log^2 n \log \log n)$ worst case update time, and $O(\log \log n)$ query time, and somewhat slower randomized method with expected update time $O(\gamma d \log^3 n \log \log n)$ if the spanning forest is maintained.

**Graph Sketching:** Graph sketching was introduced in 2012 by Ahn, Guha, and McGregor and described as “algorithms that use a linear number of measurements of a representation of a graph to determine the properties of the graph” [1]. The main concern is that the representation use $< n^2$ space and possibly support dynamic data, but the time required for individual updates is not considered. Our work was done completely independently of [1], though our work can be seen as providing a sketching technique which bears a remark-
able similarity to the one presented there. In particular, our solution to the approximate cutset problem is quite close to what the method in [1] would have yielded, if this question had been addressed [10].

3 Cutset data structure

In this section, we define the cutset problem. Consider a forest $F$ of disjoint trees, not necessarily maximal (spanning) trees, in a graph $G$. $G$ and $F$ are dynamic, that is, they are undergoing updates in the form of edge insertions and deletions of edges. In addition, an edge in $G$ which links two trees may be updated to become a tree edge which would then cause the two trees to be joined as one. Similarly a tree edge may be changed to a non-tree edge. The goal is to maintain, for each non-maximal tree $T \in F$, an edge in the cutset of $(T, V \setminus T)$, where the graph updates are oblivious to the choice of these edges. In other words, we may consider the sequence of updates to be fixed in advance, but updates are revealed to the algorithm one at a time.

We use ET-trees to store each tree $T \in F$. ET-trees are based on the observation that when a tree is linked with another tree, the sequence given by the Euler tour of the new tree can be gotten by a constant number of links and cuts to the sequences of the old trees. The sequence is stored in a balanced ordered binary tree.

**ET-tree properties:** If each node in the graph has a value stored in $x$ words of length $O(\log n)$ then: (1) a tree edge can be deleted or inserted in $O(x \log n)$ time; (2) the sum of the values stored in the nodes of a given tree can be returned in $O(x)$ time; (3) a value at a node can be updated in $O(x \log n)$ time per word; (4) given a node $v$, the name of the tree $T(v)$ containing $v$ can be returned in $O(\log n)$ time. (5) The name of the tree can be obtained more quickly if one keeps an ET-tree of degree $\log n$ solely for this purpose. That is, it can be obtained in $O(\log n / \log \log n)$ if one is willing to spend $O(\log^2 n)$ time doing an insertion or deletion of a tree edge.

3.1 Dealing with cutsets with exactly one edge

To illustrate the main idea of our data structure, we will first explain a simplified deterministic version which works when there is exactly one edge in each cutset.

Each edge $(x, y)$, $0 \leq x < y < n$, is assigned a label $x_b \cdot y_b$ where $x_b$ and $y_b$ are the $\lceil \log n \rceil$ bit vectors containing the binary representations of nodes $x$ and $y$ respectively and $x_b \cdot y_b$ denotes the $2\lceil \log n \rceil$ bit vector $x_b$ followed by $y_b$. Each node will have a value equal to the bitwise XOR (i.e., sum mod 2) of the vectors assigned to each of its incident edges. For each tree $T \in F$, the ET-tree for $T$ maintains the bitwise XOR of the values stored at $T$’s nodes. At times we may refer to the XOR of two values as their addition or subtraction, with the understanding that both these operations are equivalent to the bitwise XOR of the two values. The data structure supports operations as follows:

- **Insert an edge** $(x, y)$: Add the edge to $G$, by adding vector $x_b \cdot y_b$ to the value at nodes $x$ and $y$.
- **Make an edge** $(x, y)$ a tree edge: Use $(x, y)$ to link the ET-trees $T(x)$ and $T(y)$ in $F$.
- **Delete an edge** $(x, y)$: Subtract vector $x_b \cdot y_b$ from the value at the nodes $x$ and $y$. If the edge is in $F$, remove it from $F$.
- **Find an edge in the cutset** $(T, V \setminus T)$ where $T$ is a tree in $F$: Let $z = z_1z_2...z_{2\lceil \log n \rceil}$ be the sum of the values in $T$. If $z \neq 0$ then return the edge given by $(z^1, z^2)$, where $z^1$ is the first $\lceil \log n \rceil$ bits of $z$ and $z^2$ is the second half of $z$.

It is easy to observe the following

**Lemma 3.1.** The sum of the vectors in any tree $T$ is 0 when all the edges in $G$ incident to the tree have both endpoints in $T$, and if exactly one edge has only one endpoint in the tree, the sum of the vectors for the tree will reveal the name of the edge.

3.2 Extension to cutsets with more than one edge

To extend our data structure to work with high probability when the cutset of $(T, V \setminus T)$ has more than one edge, we have $2\lceil \log n \rceil$ levels $i$ for each of $c'\lceil \log n \rceil$ versions $j$. Let $s_{i,j}(x)$ denote the value on level $i$ in version $j$ for node $x$. When edge $(x, y)$ is inserted into $G$, for each $j = 1, 2, ..., c'\lceil \log n \rceil$ and each $i = 0, 1, ..., 2\lceil \log n \rceil - 1$ with probability $1/2^i$ we add $x_b \cdot y_b$ to both $s_{i,j}(x)$ and $s_{i,j}(y)$. So for any one version $j$, when the size of the cutset is about $2^i$, with constant probability there is exactly one edge from the cutset with the property that, for some $x$ in $T$ the representation of the edge is added to $s_{i,j}(x)$. Repeating this experiment for $c'\lceil \log n \rceil$ versions gives a high probability that this condition is true for at least one version. We keep a table $A(x, y, i, j)$ to record when $x_b \cdot y_b$ has been added to $s_{i,j}(x)$ and $s_{i,j}(y)$. We implement operations as follows:

- **Insert an edge** $(x, y)$: for versions $j = 1, ..., c'\lceil \log n \rceil$: for $i = 0, 1, 2, ... , 2\lceil \log n \rceil - 1$ with probability $1/2^i$ add vector $x_b \cdot y_b$ to value $s_{i,j}$ at the nodes $x$ and $y$ and update $A(x, y, i, j)$.
- **Make an edge** $(x, y)$ a tree edge: Use $(x, y)$ to link the
ET-trees $T(x)$ and $T(y)$ in $F$.

Delete an edge \{x, y\} for versions $j = 1, ..., c' \log n$: for $i = 0, 1, 2, ..., \lceil 2 \lg n \rceil - 1$: if $x_i \cdot y_i$ was added to $s_{i,j}(x)$ and $s_{i,j}(y)$ then subtract it, and update $A(x, y, i, j)$.

If the edge is in $F$, remove it.

**Proof.**

Find an edge in the cutset $(T, V \setminus T)$, where $T$ is a tree in $F$ for versions $j = 1, ..., c' \log n$: for $i = 0, 1, 2, ..., \lceil 2 \lg n \rceil - 1$: while an edge in the cutset $(T, V \setminus T)$ has not been found, test $z_{i,j}$ where $z_{i,j}$ is the sum of the $s_{i,j}$ in $T$ as follows: Let \{z, z\} be the edge given by $z_{i,j}$. If \{z, z\} is in $E$ and exactly one of $T(z^1), T(z^2)$ is $T$, then it is an edge in the cutset and returned.

### 3.3 Proof of correctness and analysis

**Observation 1.** Let $G_0, ..., G_t$ be the sequence of graphs from the start to update $t$. If the sequence of updates is fixed, that is, independent of the random bits of the algorithm, the distribution of the values in the arrays $s_{i,j}$ at time $t$ is identical to the distribution generated by a sequence of updates consisting of insertions of the edges of $G_t$ into the empty data structure.

**Proof.** We show this as follows: Consider the array of values $(s_{i,j}(v))_{v \in V}$ where each entry is a bit string of length $\lceil \lg n \rceil$, initialized to zero. Suppose we have a fixed sequence of Insert and Delete operations. Let $S'$ be the distribution on arrays that result from executing the algorithm for this sequence of updates. Now suppose we modify the sequence so that every subsequence of the form Insert$(e)$, $O$, Delete$(e)$, where $O$ is a sequence of updates not affecting edge $e$, is replaced by $O$. Let $S'$ be the resulting distribution on arrays. Since the deletion simply undoes the modifications to the array which resulted from the corresponding insertion, $S'$ and $S$ are identically distributed. But this means that the distribution $S$ is determined only by the edges of the graph $G_t$.

**Lemma 3.2.** Let $c' = -c \log \frac{2}{5}$ be the constant in the data structure. For any cut $(T, V \setminus T)$ of $G$, $T \in F$, with probability $1 - 1/n^c$, there are indices $i, j$, such that $s_{i,j} = \sum_{x \in T} s_{i,j}(x)$ is the name of an edge in its cutset. $s_{i,j} = 0$ if there is no edge in the cut.

**Proof.** Fix $j$. Let $E_j$ be the event that for some $i$, $\sum_{v \in T} s_{i,j}(v)$ identifies an edge in the cutset. We first show $Pr(E_j) > 1/9$.

Let $C$ be the cutset. $E_j$ is true if the name of exactly one edge $e \in C$ has been added to the $s_{i,j}(x)$ over all nodes $x \in T$. We note that for any edges $e' \notin C$, either both its endpoints are in $T$ or not in $T$ and consequently contribute 0 to $z_{i,j}$.

Let $k = |C|$ and $i' = \lceil \lg k \rceil$. For $k = 1$, $E_j$ is always true. For $k > 1$, $Pr(E_j)$ is just the probability that exactly 1 out of $k$ coinflips is heads, where each heads has probability $1/2i$. Let $i'' = \lceil \lg k \rceil$. Then

$$Pr(E_j) = \binom{k}{1} \left( \frac{1}{2i} \right) \left( \frac{2^{i'} - 1}{2^{i''}} \right)^{k-1} > \left( \frac{2^{i'} - 1}{2^{i''}} \right)^{2^{i'}+1}$$

The first product exceeds $1/9$ for $k = 2, 3$. The second approaches $1/e^2$ and is greater than $1/9$ for $k \geq 4$. Hence $Pr(E_j) \geq 1/9$ for all values of $k$. Since the $j$ versions are generated independently, these events are independent, and the probability that all are false is at most $(1 - 1/9)^{-c \log \frac{2}{5} \log n} \leq n^{-c}$.

**Theorem 3.1.** If there is an update sequence of length $t$ which is independent of the random bits then the data structure described above will succeed for every update with probability $1 - 2t/n^c$ for $c' = 9c$. The cost of the data structure is $O(\log^2 n)$ worst case time for each edge update. The cost of preprocessing a graph with $m$ edges and $n$ nodes is $O(m \log^2 n + n \log^3 n)$.

**Proof.** From Lemma 3.2, the probability of failure for a single cut is less than $1/n^c$. Each update affects at most two trees and their respective cuts. By the union bound, for a sequence of $t$ updates, the error bound is $2t/n^c$. Each insertion of an edge requires $O(\log^2 n)$ coin flips and the adjustment of $O(\log^2 n)$ words at two nodes, for a total cost of $O(\log^2 n)$ to process the ET-trees containing its endpoints. Each deletion requires the same. Each insertion or deletion of a tree edge requires $O(\#\text{words} \log n) = O(\log^2 n)$ time to link and cut a constant number of trees. Each tree which is affected by an update requires the testing of $O(\log^2 n)$ values, each of which takes $O(\log n / \log \log n)$ to process, for a total cost of $O(\log^3 n / \log \log n)$.

### 3.4 Why this cutset problem does not solve dynamic connectivity

Ideally, we would like to maintain a spanning forest. When a tree edge is deleted, we would query the cutset induced by one component of the broken tree and use this to find a replacement edge. This may not work because the probabilistic analysis we present will be erroneous if the possible cutset queries and the edge updates are correlated with the random bits in the array. That is, if the random bits in the array cause certain edges to be inserted into a spanning forest, this will induce certain cutsets to be possibly considered later. It may be that the success in finding
an edge in the cutset is inversely correlated with the probability that this cutset is selected. A particularly clear example is if the adversary is told the replacement edge then the adversary can cause the algorithm to fail by simply deleting the replacement edge after every update. After $O(\log^2 n)$ deletions, a cut would have no more replacement edges in the data structure and the algorithm would fail.

We develop an algorithm which circumvents this problem by making sure the randomness which determines a cut is different from the randomness needed to find an edge in the cutset.

4 Fully dynamic connectivity

First, let us recall Borůvka’s parallel algorithm for building a minimum spanning tree [2]. We may view the algorithm as constructing a sequence of tiers. On tier 0 are the set of nodes in the graph; on tier $\ell$ are the trees formed when for each tier $\ell - 1$ an edge is picked linking it to another $\ell - 1$ tree, while edges which form cycles are discarded. If we represent each tree on each tier $\ell$ as a node, and make a node a parent of the trees from tier $\ell - 1$ which form it, we have what has been called the Borůvka tree (or forest if the graph is not connected) [9].

Our data structure has similar structure. Let $top = [\lceil \lg n \rceil]$. For each tier $\ell = 0, 1, ..., top$, we maintain a cutset data structure for $G$ with a forest $F_\ell$. For each tier $\ell$, there is an independently generated random bit array. As the graph is updated, the random bits on tier $\ell$ are used to pick edges to join the trees from tier $\ell - 1$ to form trees on tier $\ell + 1$. We represent each tree $T$ in each $F_\ell$ by a tier $\ell$ node in a Borůvka tree structure as described above, and call $T$ a tier $\ell$ component of the Borůvka tree.

Given an edge $e$, let $\ell$ be the minimum tier such that $e \in F_\ell$. Then $e$ is a tier $\ell$ edge. If a node in the Borůvka tree has no sibling and has not been marked as maximal, i.e., a tree which spans a maximally connected component of $G$, it is unmatched.

We maintain the following Invariants of the Borůvka tree:

1. The tier 0 nodes of the Borůvka tree (or forest) are the vertices of $G$;
2. On each tier $\ell$, $F_\ell \subseteq F_{\ell+1}$. A tier $\ell + 1$ node in the Borůvka tree represents the tree obtained by linking the trees of its children with tier $\ell + 1$ edges.
3. Every internal node of the Borůvka tree that does not correspond to the spanning tree of a maximally connected component has at least one sibling;
4. The structure of the forest on tier $\ell$ is independent of the random bits in tiers $\ell$ and higher.

5. The top nodes of the Borůvka tree comprise a spanning forest of $G$.

**Lemma 4.1.** Invariant (5) is implied by invariants (1)-(4).

**Proof.** It is straightforward to show by induction that every vertex in $G$ is contained in one tree in each tier and every tree in tier $k$ is either of size $2^k$ or is a spanning tree. Hence at tier $top$, every tree is a spanning tree of a maximally connected component.

4.1 Algorithms For each tier $\ell$ of the tree, we create an array $S^\ell = s_{i,j}^\ell(x)$ for $x$ a vertex of $G$, $0 \leq i \leq 2\lg n$ and $1 \leq j \leq c\lg n$. Below, $insert(e, S^\ell)$ refers to the insert edge operation in the cutset data structure which updates $s_{i,j}^\ell$ values with respect to $e$, while $search(T, S^\ell)$ refers to the search operation in the cutset data structure which finds an edge crossing the cut induced by the tree $T$ on tier $\ell$, using the randomness in array $S^\ell$.

4.1.1 Initialization We initialize by first creating, for each tier $\ell = 0, 1, ..., top$, a cutset data structure where $F_\ell$ contains $n$ trees consisting of the single vertices in $G$. To start with any initial graph $G$, we insert each edge in $G$ using the procedure for handling edge insertions described below. Note that after initialization, every $F_\ell, \ell > 0$ is identical for all $\ell$ and consists of a spanning forest. Thus at the start, in the Borůvka tree, every node has no siblings except the tier 0 nodes.

4.1.2 Handling insertions To insert an edge $\{x, y\}$, we first insert it into the cutset data structure on each tier. If it connects two unconnected trees in $F_{top}$, add it to $F_\ell$ for all $\ell > 0$.

4.1.3 Handling queries Given a query $\{x, y\}$, if $T(x) = T(y)$ in $F_{top}$, return “yes”, else return “no”.

4.1.4 Handling deletions The main idea is as follows: To handle deletions, we wish to maintain the invariants as described above, so that we can continue to find replacement edges on a given tier without “adulterating” the random bits that are there. In order to do this, we must preserve the property that the edge cuts which are searched on a given tier are fixed by the random bits in the tiers below. We also continue to preserve the height of the tree by making sure each nonmaximal node has a sibling. We show that for each update there are no more than two searches on each tier. Hence over $n^2$ updates, fewer than $2n^2$ events occur on any
tier and with high probability every search is successful.

Delete(\{x,y\}) is called when an edge \{x,y\} is deleted. First, \{x,y\} is removed from the cutset data structures which contain it. This may cause a violation of invariant (3), i.e., a component containing \(x\) and/or a component containing \(y\) on tier \(\ell\) or higher may become unmatched. Delete finds the lowest ancestor \(A\) of \(x\) in the Boruvka tree which has become unmatched on its tier and then calls Reconnect\((A,k)\) to link it to a sibling on its tier \(k\). It repeats until there are no longer unmatched ancestors of \(x\). Then it does the same for \(y\).

Reconnect\((A,k)\) calls the function search\((A,S^k)\) in the cutset data structure for tier \(k\) to find a tier \(k+1\) edge \(e = \{v,w\}\) across the cut induced by an unmatched component \(A\) on tier \(k\). If no edge is found, then \(A\) is marked as maximal. Else Reconnect determines if there is already a path from \(v\) to \(w\) created in some forest \(F_{k''}\), \(k'' > k + 1\). In this case, there is at least one edge of tier \(k''\) on the path between \(v\) and \(w\). It is removed and \(e\) is inserted into \(F_{k+1}\) (and every \(F_{k''}\), \(k'' > k + 1\)) and \(A\) now has a sibling.

Algorithm 1 Delete(\{x,y\})

1. delete \{x,y\} from all trees containing it.
2. for \(u \in \{x,y\}\) do
3. while \(u\) has an unmatched ancestor in the Boruvka tree do
4. \(A \leftarrow\) the lowest unmatched ancestor of \(u\)
5. \(k \leftarrow\) (tier of \(A\))
6. Reconnect\((A,k)\)
7. end while
8. end for

Algorithm 2 Reconnect\((A,k)\)

1. \(e = \{v,w\} \leftarrow\) search\((A,S^k)\) (assume that \(v\) is the endpoint of \(e\) in \(A\))
2. if \(e = null\) then
3. Mark \(A\) as maximal
4. else
5. if there is a path from \(v\) to \(w\) in \(F_{top}\) then
6. \(e' \leftarrow\) an edge of maximum tier on the path between \(v\) and \(w\).
7. Remove \(e'\) from all \(F_\ell\) that contain it
8. end if
9. Add \(e\) to \(F_{k''}\) for all \(k'' > k\)
10. end if

4.2 Proof of correctness

Lemma 4.2. The Invariants of the Boruvka tree are maintained by each update, assuming that all calls to search on line 1 of Reconnect return a replacement edge iff a replacement edge exists in \(G\).

Proof. Invariant (1) is not affected by updates, and Invariant (5) is implied by Invariants 1-4. So we must show that Invariants 2-4 are maintained by updates.

Effect of insertions: If an insertion causes an edge to be added to a \(F_\ell\), it will be added to every \(F_\ell, \ell > 0\), so that Invariant (2) is maintained. Such an addition takes place only if the insertion of the edge into \(G\) causes two components in \(F_{top}\) to be connected. Since the components of \(F_{top}\) are a spanning forest of \(G\), whether two components become connected depends only on the structure of \(G\), and not on the random bits from any tier, so that Invariant (4) is maintained.

The insertion of an edge which is not added to any \(F_\ell\) cannot cause a matched non-maximal component to become unmatched, but it may cause a maximal component to become non-maximal. However, any component which was maximal before the insertion of the edge was also a component in \(F_{top}\). This means that the inserted edge links two previously unconnected trees in \(F_{top}\) which would result in its addition to every \(F_\ell\). Thus no non-maximal unmatched components remain, and Invariant (3) is maintained.

Effect of deletions: For Invariant (2): If an edge is made into a tree edge at tier \(\ell\) then it is made into a tree edge at all higher tiers; if it is removed from \(F_\ell\) then it is removed from all forests which contain it, thus for each \(\ell, F_\ell \subseteq F_{\ell+1}\) is maintained.

For Invariant (3), we observe that the deletion of \(\{x,y\}\) from \(G\) can result in components in the Boruvka tree becoming unmatched only if they contain the vertices \(x\) and/or \(y\). We first show that after Reconnect\((A,k)\) is called, component \(A\) is no longer unmatched, any unmatched components containing \(x\) are on tiers higher than \(k\), and components which do not contain \(x\) do not become unmatched. If a tier \(k + 1\) edge \(e = \{v,w\}\) is added by Reconnect\((A,k)\) then there are three cases:

(i) \(e\) relinks the components previously joined by \(\{x,y\}\) and the Boruvka tree is completely repaired.

(ii) \(e\) links \(x\)'s component to another component in tier \(k\) and does not induce a cycle.

(iii) \(e\) induces a cycle within some \(F_{k''}, k'' > k' + 1\).

If case (i) occurs, no more needs to be done. If case (ii) occurs, Delete with \(u = y\) will take care of the problem. If case (iii) occurs, there is an edge \(e'\) which is a tier
of a query makes updates consistent with what A would do if the queries so far had been answered correctly. A’ is oblivious to the random bits of the algorithm. By Lemma 4.3 the algorithm will maintain invariants (1)-(5) and hence answer all queries correctly with probability at least $1 - \frac{c}{n^{c+2}}$. As long as the algorithm answers correctly, A will behave like A’, the algorithm will maintain the invariants, and answer the next query for A, with the same probability as it would for A’. By Observation 1, insertions do not increase the probability an incorrect answer.

4.3 Implementation details and running time

Analysis We do not explicitly represent the Borůka trees. We represent the top + 1 cutset data structures as described previously indexed by tier. Determining the lowest unmatched ancestor of a vertex can be implemented by checking each tier to determine if the tree containing u in that tier is unmatched. One way to determine if a tree is unmatched is to maintain the number of vertices in the tree and check if its parent has the same size, which can be done with ET-trees.

ST-trees provide a means of finding the maximum weight edge on a path between two nodes in a dynamic tree, in $O(\log n)$ time per tree edge link, cut, and the find operation. If ST-trees [13] are used to represent the top tier forest $F$ in which each tree edge is labelled with its tier number, Reconnect can find a maximum weighted edge on the path from $v$ to $w$ in $O(\log n)$ time.

Lemma 4.5. The total cost of initialization without any edges is $O(n \cdot \text{top})$. The total cost of a deletion update is $O(\log^5 n)$. The cost of an insertion update is $O(\log^3 n)$. The cost of a query is $O(\log n / \log \log n)$.

Proof. For the deletion of a tree edge, there may be two Reconnects per tier. Each Reconnect requires one search, and up to $\log n$ links and cuts; the first costs $O(\log n)$. Each link and cut of a tree storing $O(\log^2 n)$ words per node costs $O(\log^3 n)$ so the total cost of linking and cutting per Reconnect is $O(\log^4 n)$. Each search requires a total cost of $O(\log^3 n / \log \log n)$, since it involves the testing of up to $\log n$ versions each containing $\log n$ edges. Each test requires $\log n / \log \log n$ time to check. Since there are no more than two Reconnects per tier, the overall cost is $O(\log^5 n)$.

An edge insertion requires an insertion into $\log n$ cutset data structures. Each costs $O(\log^3 n)$ for a total of $O(\log^4 n)$. If the edge becomes a tree edge, it is inserted into trees in $\log n$ tiers, for a cost of $O(\log^3 n)$ per tier or $O(\log^4 n)$ overall.

The maintainence of the ST-trees costs $O(\log n)$ per change to the forest or $O(\log^2 n)$ overall for a deletion.
update, since $O(\log n)$ tree edges may change. The maintenance of the ET-trees of degree $\log n$ for the purpose of answering queries is $O(\log^2 n)$ per change to the forest or $O(\log^3 n)$ overall for a deletion update.

The query time is $O(\log n/\log \log n)$ using degree $\log n$ ET-trees.

4.4 Handling arbitrarily long sequences of updates We partition the update sequence into intervals of $n^2$ deletions. We use two data structures, $D_0$ and $D_1$. Before the first interval, $D_1$ is initialized. During the first interval $D_1$ processes the updates and answers the queries accordingly. In general, during interval $i$, $D_{(i \mod 2)}$ is in charge of processing the updates and answering queries. In parallel $D_{(i+1 \mod 2)}$ performs a reinitialization during the first half of interval $i$, using the edgset at the start of the interval $i$. During the second half of interval $i$, it simulates the updates on its data structure that have occurred during interval $i$, at double time until it is caught up to the current update, by the end of the interval $i$. At this point the next interval starts, and $D_1$ and $D_0$ swap roles. The probability of correctness at any given point is at least the probability of correctness of the initialization and the subsequent $O(n^2)$ updates which follow that initialization. This changes the cost per operation by no more than a constant factor. Thus Lemmas 4.4 and 4.5 imply:

**Theorem 4.1.** Given an initial graph with $m$ edges and $n$ nodes, for any constant $c$ and any sequence $\sigma_1, \ldots, \sigma_r$ of updates and queries, the algorithm answers each query correctly with probability at least $1 - 1/n^c$ with worst case cost $O(\log^c n)$ per insertion, $O(\log^c n)$ per deletion, and $O((\log n)/\log \log n)$ per query. The preprocessing cost of the algorithm is $O(m \log^c n + n \log^c n)$.

Furthermore, for any $1 \leq i < j \leq r$, if $\sigma_i, \ldots, \sigma_j$ contains at most $n^c$ deletions, then with probability $1 - 1/n^c$, every query in $\sigma_i, \ldots, \sigma_j$ is answered correctly.

5 Applications of techniques

Here we give a couple of applications of the basic Monte Carlo technique.

5.1 Approximating Cutset Size

An oblivious adversary is an adversary whose choices of updates and queries are independent of the query answers previously obtained. We show:

**Theorem 5.1.** Let $G = (V, E)$ be an undirected graph with $n$ nodes. Given any sequence of edge insertions and deletions, and queries, generated by an oblivious adversary, where queries are of the form: “What is the size of the cutset of cut $(U, V \setminus U)$?” there exists an algorithm which returns an answer to each query which is correct within a factor of 2 with probability $1 - 1/n^c$, for any constant $c > 0$, in worst case time $O(\log^2 n)$ per edge update and $O(|U| \log^2 n)$ per query.

The algorithm inserts and deletes edges as in the cutset problem, except that instead of adding the name of the edge $(x, y)$ to $x$ and to $y$, we either add $+1$ to $x$ and $-1$ to $y$ or $-1$ to $x$ and $+1$ to $y$. We keep $O(\log n)$ versions and $O(\log n)$ levels as in the cutset problem. When a query is given, we find the lowest level $i$ with more than a threshold number of versions with value 0 at level $i$ and return $3 \times 2^{i-2}$. We amplify the probability of success by using a constant number of extra trials per level and version to determine if the value really should be considered 0.

5.1.1 Algorithm For $i = 0, \ldots, 2 \log n$, $j = 1, \ldots, c \log n$ and $k = 1, \ldots, 7$, and for each node $x \in V$, we maintain an $O(\log n)$ bit number $S_{ijk}(v)$ and a table $A_{ijk} \in \{1, 0, 1\}$ to track the values added to the endpoints of each edge. $S$ and $A$ are initially 0. The constant $c'$ depends on $c$ in the theorem.

**Algorithm 3** Add edge $(x, y)$, $x < y$

1: for $j = 1, \ldots, c' \log n$ do
2: for $i = 1, \ldots, 2 \log n$ do
3: Set $c(ij) = \text{heads with probability } 1/2^i \text{ and tails otherwise}$
4: if $\text{coin } c(ij) \text{ is heads then}$
5: for $k = 1, \ldots, 7$ do
6: Set $\text{coin } c(ijk) = \text{heads with probability } 1/2 \text{ and tails otherwise}$
7: if $\text{coin } c(ijk) \text{ is heads then}$
8: $S_{ijk}(x) \leftarrow S_{ijk}(x) + 1$
9: $S_{ijk}(y) \leftarrow S_{ijk}(y) - 1$
10: $A_{ijk}(x, y) \leftarrow 1$
11: else
12: $S_{ijk}(x) \leftarrow S_{ijk}(x) - 1$
13: $S_{ijk}(y) \leftarrow S_{ijk}(y) + 1$
14: $A_{ijk}(x, y) \leftarrow -1$
15: end if
16: end for
17: end if
18: end for

**NOTE:** Bits are incrementally refreshed so that they have been replaced every $n^c$ queries to maintain the probability of correctness.

5.1.2 Proof of correctness First we observe:

**Lemma 5.1.** For a fixed $i, j$, if some $c(ij)$ was heads for some edge in the cutset of $(U, V \setminus U)$ then the probability
Proof of Theorem 5.1:

Algorithm 4 Delete edge \((x, y), x < y\)

1: for \(j = 1, \ldots, c' \ln n\) do
2: for \(i = 1, \ldots, 2 \lg n\) do
3: for \(k = 1, \ldots, 7\) do
4: \(S_{ijk}(x) \leftarrow S_{ijk}(x) - A_{ijk}(x, y)\)
5: \(S_{ijk}(y) \leftarrow S_{ijk}(y) + A_{ijk}(x, y)\)
6: \(A_{ijk}(x, y) \leftarrow 0\)
7: end for
8: end for
9: end for

Algorithm 5 Estimate Cut \(U\)

1: \(T \leftarrow 0.3528\)
2: Initialize \(X = \text{Count} = 0\)
3: for all \(i, j\) do
4: if for any \(k\), \(\sum_{v \in U} S_{ij,k}(v) \neq 0\) then
5: \(X_{ij}(U) \leftarrow 1\)
6: end if
7: end for
8: \(\text{Count}_0(U) \leftarrow \sum_{i,j} X_{ij}(U)\)
9: Find the smallest \(i\) such that \(\text{Count}_i \leq (1 - T)(c' \ln n)\)
10: if \(\text{Count}_0 = 0\) then
11: return 0
12: else if \(i \leq 1\) then
13: return 1
14: else
15: return \(3 \cdot 2^{i-2}\)
16: end if

that for all \(k = 1, \ldots, 7, \sum_{v \in U} S_{ijk}(v) = 0\) is no greater than \(1/2^7\).

Proof. If \(\sum_{v \in U} S_{ijk}(v) = 0\), then for the \(c_{ijk}\) which were tossed for edges crossing the cutset, there were an equal number of heads and tails, which happens with probability \(\leq 1/2\). Hence the probability there are an equal number of heads and tails for all \(k\) trials is at most \(1/2^7\).

Proof of Theorem 5.1: Let \(K\) be the size of the cutset of the cut \((U, V \setminus U)\). We show that given a series of insertions, deletions and cut size queries; if the cutset queried is chosen independently of previous queries then Algorithm 5 returns an approximation \(x\) of the cutset such that \(K/2 \leq x \leq 2K\) with high probability.

Let \(t = [\lg K]\). Let \(t = c' \ln n\). The algorithm will only fail in the following two cases:

1. \(K \geq 3 \cdot 2^{l-1}\) and \(\text{Count}_l(U) < (1 - T)t\) for some \(i \leq l - 1\) or both \(\text{Count}_l(U) > (1 - T)t\) and \(\text{Count}_{l+1}(U) > (1 - T)t\).

2. \(K \geq 3 \cdot 2^{l-1}\) and \(\text{Count}_l(U) < (1 - T)t\) for some \(i \leq l - 1\) or both \(\text{Count}_l(U) > (1 - T)t\) and \(\text{Count}_{l+1}(U) > (1 - T)t\).

For Case (1), we consider the probability that \(\text{Count}_l(U) < (1 - T)t\) for \(i \leq l - 1\). First we consider \(X_{ij}\) for a fixed \(j\). \(X_{ij} = 0\) only if for every edge in the cutset \(c_{ij}\) is tails and/or all \(k\) trials end up with an equal number of heads and tails. By the observation above, the latter happens with probability \(1/2^7\). By a union bound,

\[
\Pr(X_{ij}(U) = 0) \leq \Pr(\text{no heads is tossed by a } c_{ij}) \leq \Pr(\text{coin for an edge in the cutset}) + 1/2^7 \\
\leq \left(1 - \frac{1}{2^{l-1}}\right)^2 + 0.01 \leq e^{-2} + 0.01 \leq .1454.
\]

Since each version \(j\) is independent, we can use a Chernoff bound to bound \(\Pr(\text{Count}_l < (1 - T)t) = \Pr(\sum_j X_{ij}(U) \geq Tt) = \Pr(\sum_j X_{ij}(U) \geq (1 + \delta).1454t) \leq e^{-\delta^2/(1454^2)/3}\) where \((1 + \delta).1454t = Tt\). Substituting \(T = .3528\) and solving for \(\delta\) gives \(\delta = 1.426\). Recall that \(t = c' \ln n\). For \(c'\) a sufficiently large constant, this yields \(1/n^{c+1}\). The probability that this occurs for any \(i \leq l - 1\) is no greater than \((l - 1)/n^{c+2} < 1/n^{c+1}\).

Next we consider the probability that \(\text{Count}_l(U) > (1 - T)t\) and \(\text{Count}_{l+1}(U) > (1 - T)t\). The probability of both of these events occurring is bounded no greater than the probability of the second one occurring. We fix \(j\) and lower bound \(\Pr(X_{ij(1)} = 0)\), given that \(K < 3 \cdot 2^{l-1}\).

If \(j = 1\), \(\Pr(X_{i1,j(1)} = 0) \leq \left(1 - \frac{1}{2^{l+1}}\right)^{3 \cdot 2^{l-1} - 1} = (1 - \frac{1}{2^{l+1}})^{2} > 0.5625\)

If \(j \geq 2\) then

\[
\Pr(X_{i1,j(1)} = 0) = \left(1 - \frac{1}{2^{l+1}}\right)^{3 \cdot 2^{l-1} - 1} \\
\geq \exp\left\{-\frac{1}{2^{l+1}} - \frac{3}{2^{l+2}}\right\}^{3 \cdot 2^{l-1} - 1} \\
= \exp\left\{-\frac{3}{2^{l+1}} - \frac{3}{2^{l+2}}\right\} \\
\geq 0.4950 \quad (\text{since } j \geq 2)
\]

Therefore, \(\Pr(X_{i1,j(1)} = 0) > 0.4950t\). Using Chernoff bounds, we can show an upper bound on \(\Pr(\text{Count}_{l+1}(U) > (1 - T)t) = \Pr(\sum_j X_{l+1,j}(U) < .3528)\) which is \(1/n^{c+1}\) similarly to the analysis above. The probability that case (1) fails is thus \(\leq 2/n^{c+1}\) by a union bound.
The analysis of case (2) is similar; we leave this to the reader, which by a union bound, yields a probability of error of less than $1/n^c$.

### 5.2 Emergency Planning

Given a fixed graph $G = (V, E)$, the Emergency Planning problem is to be able to preprocess $G$ so that for any one-time batch of online insertions and deletions, structural information about the resulting graph can be easily obtained. In particular, one can make connectivity queries, and augment the data structure to answer questions like the size of maximal connected components.

Any dynamic connectivity data structure can be used to solve the Emergency Planning problem. The challenge is whether we can match the update and query bounds of \cite{11}. That paper gives a deterministic algorithm which makes use of a “polynomial” number of calls to a weighted sparsest cut algorithm in the preprocessing phase. Since this problem is NP-Hard, the algorithm relies on $\gamma$-approximation algorithms for preprocessing. The algorithm gives an $O(\gamma d \log^2 n \log \log n)$ worst case update time, where $d$ is the number of edges in the batch, and $O(\log \log n)$ query time. A somewhat slower randomized method which maintains the spanning forest is presented with expected update time $O(\gamma d \log^3 n \log \log n)$.

A straightforward implementation using our data structure would be to construct a spanning forest, store this in a balanced complete ET tree and insert every nontree edge using the dynamic connectivity edge insertion algorithm. When the batch of up to $d$ edge deletions are given, we delete each edge from the ET-trees, breaking these into as many as $d + 1$ subtrees (“broken pieces”). We could then rebuild the spanning forest by running the initialization algorithm, but starting with the broken pieces on the lowest tier. This method would result in a cost of $O(d \log^3 n \log d)$ for the batch update and $O(\log n)$ query time.

**Squeezing out the small factors:** To reduce the query time to $O(\log \log n)$ we follow the ideas sketched in \cite{11}. We store the original ET-tree in an array which, after the update, enables each node to find an ancestor on a given level in constant time, and hence a representative of its broken piece in $\log \log n$ time. Then following \cite{11}, when we stitch together the broken pieces, we can use a union-find data structure so as to quickly find the new component which contains a particular broken piece.

To enable each node to locate the name of its broken piece in time $O(\log \log n)$: When the batch update occurs, identify the leftmost and rightmost elements in the ET-tree which belong to each broken piece (the “boundary elements”). Label each ancestor of each boundary element with the name of its broken piece. If an ancestor is ancestor of boundary elements from more than one broken piece, label it $11X^\pi$. Note that once a node is labelled $X$, so are its ancestors. To locate the name of a node's broken piece, binary search its ancestors until one is found with exactly one name.

We also observe that we could cut down the expected amount of work considerably in this algorithm. The expected number of versions needed to find a replacement edge is $O(1)$. Hence finding a replacement edge in one cutset requires an expected constant number of versions checked, rather than $O(\log n)$. Now each version has $O(\log n)$ levels. We normally would check all of these, but ideally we just want to find the highest one which contains an edge by itself. We can binary search these levels. We omit the details here.

### 6 Discussion

The methods here are surprisingly simple yet quite different from other known techniques for dynamic graph algorithms. As noted in the related work section, this work can be seen as a technique for adaptive graph sketching with a concern for fast update and query time. What update and query times can be achieved for other graph problems in the graph sketching model or when space is an issue? It is also interesting to ask about these questions in a parallel streaming model such as MapReduce.

Classic dynamic graph problems which remain open and which still seem hard are dynamic minimum spanning tree in $o(\sqrt{n})$ worst case time and Las Vegas or deterministic graph connectivity with $o(\sqrt{n})$ worst case cost.

It is not hard to see that the technique described here can be made deterministic with an additional $O(k)$ factor in the update time if we know the cuts are of size no greater than $k$, through the use of combinatorial designs. Hence any lower bounds for deterministic algorithms should make use of large cuts.

**Acknowledgement:** The authors would like to thank Mikkel Thorup for pointing out \cite{1}.

*This paper is dedicated to the memory of Mihai Pătrașcu.*

### References

Appendix
The figures on the following page depict the processing of an edge deletion for the fully dynamic connectivity algorithm.
Figure 1: An edge is deleted from $F_{k+1}$.

Figure 2: Two components become unmatched in $F_k$ and $F_{k+2}$.

Figure 3: A new $k+1$ edge across the cut induced by an unmatched component in $F_k$ is found, but adding causes a cycle in $F_{k+3}$ so an edge in $F_{k+3}$ is removed.

Figure 4: A component in $F_{k+2}$ becomes unmatched.

Figure 5: An edge is added to $F_{k+3}$ which crosses the cut induced by the unmatched component.

Figure 6: There are no more unmatched components.