# Dynamic bridge-finding in õ(log2 n) amortized time

**Holm, Jacob; Rotenberg, Eva; Thorup, Mikkel**

*Published in:*
Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms

*Publication date:*
2018

*Document Version*
Peer reviewed version

# Dynamic Bridge-Finding in $\widetilde{O}(\log^2 n)$ Amortized Time

Jacob Holm[*], Eva Rotenberg, and Mikkel Thorup[*]

University of Copenhagen (DIKU),
jaho@di.ku.dk, roden@di.ku.dk, mthorup@di.ku.dk

August 23, 2017

**Abstract**

We present a deterministic fully-dynamic data structure for maintaining information about the bridges in a graph. We support updates in $\widetilde{O}((\log n)^2)$ amortized time, and can find a bridge in the component of any given vertex, or a bridge separating any two given vertices, in $\mathcal{O}(\log n/\log\log n)$ worst case time. Our bounds match the current best for bounds for deterministic fully-dynamic connectivity up to $\log\log n$ factors.

The previous best dynamic bridge finding was an $\widetilde{O}((\log n)^3)$ amortized time algorithm by Thorup [STOC2000], which was a bittrick-based improvement on the $\mathcal{O}((\log n)^4)$ amortized time algorithm by Holm et al.[STOC98, JACM2001].

Our approach is based on a different and purely combinatorial improvement of the algorithm of Holm et al., which by itself gives a new combinatorial $\widetilde{O}((\log n)^3)$ amortized time algorithm. Combining it with Thorup's bittrick, we get down to the claimed $\widetilde{O}((\log n)^2)$ amortized time.

Essentially the same new trick can be applied to the biconnectivity data structure from [STOC98, JACM2001], improving the amortized update time to $\widetilde{O}((\log n)^3)$.

We also offer improvements in space. We describe a general trick which applies to both of our new algorithms, and to the old ones, to get down to linear space, where the previous best use $\mathcal{O}(m + n\log n\log\log n)$. Finally, we show how to obtain $\mathcal{O}(\log n/\log\log n)$ query time, matching the optimal trade-off between update and query time.

Our result yields an improved running time for deciding whether a unique perfect matching exists in a static graph.

[*]This research is supported by Mikkel Thorup's Advanced Grant DFF-0602-02499B from the Danish Council for Independent Research under the Sapere Aude research career programme.

## 1 Introduction

In graphs and networks, connectivity between vertices is a fundamental property. In real life, we often encounter networks that change over time, subject to insertion and deletion of edges. We call such a graph *fully dynamic*. Dynamic graphs call for dynamic data structures that maintain just enough information about the graph in its current state to be able to promptly answer queries.

Vertices of a graph are said to be *connected* if there exists a path between them, and *$k$-edge connected* if no sequence of $k-1$ edge deletions can disconnect them. A *bridge* is an edge whose deletion would disconnect the graph. In other words, a pair of connected vertices are 2-edge connected if they are not separated by a bridge. By Menger's Theorem [17], this is equivalent to saying that a pair of connected vertices are two-edge connected if there exist two edge-disjoint paths between them. By edge-disjoint is meant that no edge appears in both paths.

For dynamic graphs, the first and most fundamental property to be studied was that of dynamic connectivity. In general, we can assume the graph has a fixed set of $n$ vertices, and we let $m$ denote the current number of edges in the graph. The first data structure with sublinear $\mathcal{O}(\sqrt{n})$ update time is due to Frederickson [5] and Eppstein et al. [4]. Later, Frederickson [6] and Eppstein et al. [4] gave a data structure with $\mathcal{O}(\sqrt{n})$ update time for two-edge connectivity. Henzinger and King achieved poly-logarithmic expected amortized time [8], that is, an expected amortized update time of $\mathcal{O}((\log n)^3)$, and $\mathcal{O}(\log n/\log\log n)$ query time for connectivity. And in [9], $\mathcal{O}((\log n)^5)$ expected amortized update time and $\mathcal{O}(\log n)$ worst case query time for 2-edge connectivity. The first polylogarithmic deterministic result was by Holm et al in [10]; an amortized deterministic update time of $\mathcal{O}((\log n)^2)$ for connectivity, and $\mathcal{O}((\log n)^4)$ for 2-edge connectivity. The update time for deterministic dynamic connectivity has later been improved to $\mathcal{O}((\log n)^2/\log\log n)$ by Wulff-Nilsen [21]. Sacrificing determinism, an $\mathcal{O}(\log n(\log\log n)^3)$ structure for connectivity was presented by Thorup [20], and later improved to $\mathcal{O}(\log n(\log\log n)^2)$ by Huang et al. [12]. In the same paper, Thorup obtains an update time of $\mathcal{O}((\log n)^3\log\log n)$ for deterministic two-edge connectivity. Interestingly, Kapron et al. [13] gave a Monte Carlo-style randomized data structure with polylogarithmic worst case update time for dynamic connectivity, namely, $\mathcal{O}((\log n)^4)$ per edge insertion, $\mathcal{O}((\log n)^5)$ per edge deletion, and $\mathcal{O}(\log n/\log\log n)$ per query. We know of no similar result for bridge finding. The best lower bound known is by Pătraşcu et al. [18], which shows a trade-off between update time $t_u$ and query time $t_q$ of $t_q\lg\frac{t_u}{t_q}=\Omega(\lg n)$ and $t_u\lg\frac{t_q}{t_u}=\Omega(\lg n)$.

### 1.1 Our results

We obtain an update time of $\mathcal{O}((\log n)^2(\log\log n)^2)$ and a query time of $\mathcal{O}(\log n/\log\log n)$ for the bridge finding problem:

**Theorem 1.** *There exists a deterministic data structure for dynamic multigraphs in the word RAM model with $\Omega(\log n)$ word size, that uses $\mathcal{O}(m+n)$ space, and can handle the following updates, and queries for arbitrary vertices $v$ or arbitrary connected vertices $v,u$:*

- *insert and delete edges in $\mathcal{O}((\log n)^2(\log\log n)^2)$ amortized time,*
- *find a bridge in $v$'s connected component or determine that none exists, or find a bridge that separates $u$ from $v$ or determine that none exists. Both in $\mathcal{O}(\log n/\log\log n)$ worst-case time for the first bridge, or $O(\log n/\log\log n+k)$ worst case time for the first $k$ bridges.*

- *find the size of $v$'s connected component in $\mathcal{O}(\log n/\log\log n)$ worst-case time, or the size of its 2-edge connected component in $\mathcal{O}(\log n(\log\log n)^2)$ worst-case time.*

Since a pair of connected vertices are two-edge connected exactly when there is no bridge separating them, we have the following corollary:

**Corollary 2.** *There exists a data structure for dynamic multigraphs in the word RAM model with $\Omega(\log n)$ word size, that can answer two-edge connectivity queries in $\mathcal{O}(\log n/\log\log n)$ worst case time and handle insertion and deletion of edges in $\mathcal{O}((\log n)^2(\log\log n)^2)$ amortized time, with space consumption $\mathcal{O}(m+n)$.*

Note that the query time is optimal with respect to the trade-off by Pătraşcu et al. [18]

As a stepping stone on the way to our main theorem, we show the following:

**Theorem 3.** *There exists a combinatorial deterministic data structure for dynamic multigraphs on the pointer-machine without the use of bit-tricks, that uses $\mathcal{O}(m+n)$ space, and can handle insertions and deletions of edges in $\mathcal{O}((\log n)^3\log\log n)$ amortized time, find bridges and determine connected component sizes in $\mathcal{O}(\log n)$ worst-case time, and find 2-edge connected component sizes in $\mathcal{O}((\log n)^2\log\log n)$ worst-case time.*

Our results are based on modifications to the 2-edge connectivity data structure from [11]. Applying the analoguous modification to the biconnectivity data structure from the same paper yields a structure with $\mathcal{O}((\log n)^3(\log\log n)^2)$ amortized update time and $\mathcal{O}((\log n)^2(\log\log n)^2)$ worst case query time. The details of this modification are beyond the scope of this paper.

## 1.2 Applications

While dynamic graphs are interesting in their own right, many algorithms and theorem proofs for static graphs rely on decremental or incremental graphs. Take for example the problem of whether or not a graph has a unique perfect matching? The following theorem by Kotzig immediately yields a near-linear algorithm if implemented together with a decremental two-edge connectivity data structure with poly-logarithmic update time:

**Theorem 4** (A. Kotzig '59 [16])**.** *Let $G$ be a connected graph with a unique perfect matching $M$. Then $G$ has a bridge that belongs to $M$.*

The near-linear algorithm for finding a unique perfect matching by Gabow, Kaplan, and Tarjan [7] is straight-forward: Find a bridge and delete it. If deleting it yields connected components of odd size, it must belong to the matching, and all edges incident to its endpoints may be deleted—if the components have even size, the bridge cannot belong to the matching. Recurse on the components. Thus, to implement Kotzig's Theorem, one has to implement three operations: One that finds a bridge, a second that deletes an edge, and a third returning the size of a connected component.

Another example is Petersen's theorem [19] which states that any cubic, two-edge connected graph contains a perfect matching. An algorithm by Biedl et al. [2] finds a perfect matching in such graphs in $\mathcal{O}(n\log^4 n)$ time, by using the Holm et al two-edge connectivity data structure as a subroutine. In fact, one may implement their algorithm and obtain running time $\mathcal{O}(nf(n))$, by using as subroutine a data structure for amortized decremental two-edge connectivity with update-time $f(n)$. Here, we thus improve the running time from $\mathcal{O}(n(\log n)^3\log\log n)$ to $\mathcal{O}(n(\log n)^2(\log\log n)^2)$.

In 2010, Diks and Stanczyk [3] improved Biedl et al.'s algorithm for perfect matchings in two-edge connected cubic graphs, by having it rely only on dynamic connectivity, not two-edge connectivity, and thus obtaining a running time of $\mathcal{O}(n(\log n)^2/\log\log n)$ for the deterministic version, or $\mathcal{O}(n\log n(\log\log n)^2)$ expected running time for the randomized version. However, our data structure still yields a direct improvement to the original algorithm by Biedl et al.

Note that all applications to static graphs have in common that it is no disadvantage that our running time is amortized.

## 1.3 Techniques

As with the previous algorithms, our result is based on top trees [1] which is a hierarchical tree structure used to represent information about a dynamic tree — in this case, a certain spanning tree of the dynamic graph. The original $\mathcal{O}((\log n)^4)$ algorithm of Holm et al. [11] stores $\mathcal{O}((\log n)^2)$ counters with each top tree node, where each counter represent the size of a certain subgraph. Our new $\mathcal{O}((\log n)^3)$ algorithm applies top trees the same way, representing the same $\mathcal{O}((\log n)^2)$ sizes with each top tree node, but with a much more efficient implicit representation of the sizes.

Reanalyzing the algorithm of Holm et al. [11], we show that many of the sizes represented in the top nodes are identical, which implies that that they can be represented more efficiently as a list of actual differences. We then need additional data structures to provide the desired sizes, and we have to be very careful when we move information around as the the top tree changes, but overall, we gain almost a log-factor in the amortized time bound, and the algorithm remains purely combinatorial.

Our combinatorial improvement can be composed with the bittrick improvement of Thorup [20]. Thorup represents the same sizes as the original algorithm of Holm et al., but observes that we don't need the exact sizes, but just a constant factor approximation. Each approximate size can be represented with only $\mathcal{O}(\log\log n)$ bits, and we can therefore pack $\Omega(\log n/\log\log n)$ of them together in a single $\Omega(\log n)$-bit word. This can be used to reduce the cost of adding two $\mathcal{O}(\log n)$-dimensional vectors of approximate sizes from $\mathcal{O}(\log n)$ time to $\mathcal{O}(\log\log n)$ time. It may not be obvious from the current presentation, but it was a significant technical difficulty when developing our $\mathcal{O}((\log n)^3\log\log n)$ algorithm to make sure we could apply this technique and get the associated speedup to $\mathcal{O}((\log n)^2(\log\log n)^2)$.

The "natural" query time of our algorithm is the same as its update time. In order to reduce the query time, we observe that we can augment the main algorithm to maintain a secondary structure that can answer queries much faster. This can be used to reduce the query time for the combinatorial algorithm to $\mathcal{O}(\log n)$, and for the full algorithm to the optimal $\mathcal{O}(\log n/\log\log n)$.

The secondary structure needed for the optimal $\mathcal{O}(\log n/\log\log n)$ query time uses top trees of degree $\mathcal{O}(\log n/\log\log n)$. While the use of non-binary trees is nothing new, we believe we are the first to show that such top trees can be maintained in the "natural" time.

Finally, we show a general technique for getting down to linear space, using top trees whose base clusters have size $\Theta(\log^c n)$.

## 1.4 Article outline

In Section 2, we recall how [11] fundamentally solves two-edge connectivity via a reduction to a certain set of operations on a dynamic forest. In Section 3, we recall how top trees can be used to maintain information in a dynamic forest, as shown in [1]. In Sections 4, 5, and 6, we describe how to support the operations on a dynamic tree needed to make a combinatorial $\mathcal{O}((\log n)^3 \log\log n)$ algorithm for bridge finding, as stated in Theorem 3. Then, in Section 7, we show how to use Approximate Counting to get down to $\mathcal{O}((\log n)^2(\log\log n)^2)$ update time, thus, reaching the update time of Theorem 1. We then revisit top trees in Section 8, and introduce the notion of $B$-ary top trees, as well as a general trick to save space in complex top tree applications. We proceed to show how to obtain the optimal $\Theta(\log n/\log\log n)$ query time in Section 9. Finally, in Section 10, we show how to achieve optimal space, by only storing cluster information with large clusters, and otherwise calculating it from scratch when needed.

## 2 Reduction to operations on dynamic trees

In [11], two-edge connectivity was maintained via operations on dynamic trees, as follows. For each edge $e$ of the graph, the algorithm explicitly maintains a *level*, $\ell(e)$, between 0 and $\ell_{\max} = \lfloor \log_2 n \rfloor$ such that the edges at level $\ell_{\max}$ form a spanning forest $T$, and such that the 2-edge-connected components in the subgraph induced by edges at level at least $i$ have at most $\lfloor n/2^i \rfloor$ vertices. For each edge $e$ in the spanning forest, define the *cover level*, $c(e)$, as the maximum level of an edge crossing the cut defined by removing $e$ from $T$, or $-1$ if no such edge exists. The cover levels are only maintained implicitly, because each edge insertion and deletion can change the cover levels of $\Omega(n)$ edges. Note that the bridges are exactly the edges in the spanning forest with cover level $-1$. The algorithm explicitly maintains the spanning forest $T$ using a dynamic tree structure supporting the following operations:

1. Link$(v, w)$. Add the edge $(v, w)$ to the dynamic tree, implicitly setting its cover level to $-1$.
2. Cut$(v, w)$. Remove the edge $(v, w)$ from the dynamic tree.
3. Connected$(v, w)$. Returns **true** if $v$ and $w$ are in the same tree, **false** otherwise.
4. Cover$(v, w, i)$. For each edge $e$ on the tree path from $v$ to $w$ whose cover level is less than $i$, implicitly set the cover level to $i$.
5. Uncover$(v, w, i)$. For each edge $e$ on the tree path from $v$ to $w$ whose cover level is at most $i$, implicitly set the cover level to $-1$.
6. CoverLevel$(v)$. Return the minimal cover level of any edge in the tree containing $v$.
7. CoverLevel$(v, w)$. Return the minimal cover level of an edge on the path from $v$ to $w$. If $v = w$, we define CoverLevel$(v, w) = \ell_{\max}$.
8. MinCoveredEdge$(v)$. Return any edge in the tree containing $v$ with minimal cover level.
9. MinCoveredEdge$(v, w)$. Returns a tree-edge on the path from $v$ to $w$ whose cover level is CoverLevel$(v, w)$.
10. AddLabel$(v, l, i)$. Associate the *user label* $l$ to the vertex $v$ at level $i$.
11. RemoveLabel$(l)$. Remove the user label $l$ from its vertex vertex$(l)$.
12. FindFirstLabel$(v, w, i)$. Find a user label at level $i$ such that the associated vertex $u$ has CoverLevel$(u, \text{meet}(u, v, w)) \geq i$ and minimizes the distance from $v$ to meet$(u, v, w)$.
13. FindSize$(v, w, i)$. Find the number of vertices $u$ such that CoverLevel$(u, \text{meet}(u, v, w)) \geq i$. Note that FindSize$(v, v, -1)$ is just the number of vertices in the tree containing $v$.

**Lemma 5** (Essentially the high level algorithm from [11])**.** *There exists a deterministic reduction for dynamic graphs with $n$ nodes, that, when starting with an empty graph, supports any sequence of $m$ Insert or Delete operations using:*

- $\mathcal{O}(m)$ *calls to Link, Cut, Uncover, and CoverLevel.*
- $\mathcal{O}(m \log n)$ *calls to Connected, Cover, AddLabel, RemoveLabel, FindFirstLabel, and FindSize.*

*And that can answer FindBridge queries using a constant number of calls to Connected, CoverLevel, and MinCoveredEdge.*

*Proof.* See Appendix A for a proof and pseudocode. □

| # | Operation | Asymptotic worst case time per call, using structure in section | | | | |
|---|---|---|---|---|---|---|
| | | 4 | 5 | 6 | 7 | 9 |
| 1 | Link$(v, w, e)$ | $\log n$ | $(\log n)^2 \log\log n$ | $\log n \log\log n$ | $\log n (\log\log n)^2$ | $\frac{f(n)\log n}{\log f(n)}$ |
| 2 | Cut$(e)$ | $\log n$ | $(\log n)^2 \log\log n$ | $\log n \log\log n$ | $\log n (\log\log n)^2$ | $\frac{f(n)\log n}{\log f(n)}$ |
| 3 | Connected$(v, w)$ | $\log n$ | $(\log n)^2 \log\log n$ | $\log n \log\log n$ | $\log n (\log\log n)^2$ | $\frac{\log n}{\log f(n)}$ |
| 4 | Cover$(v, w, i)$ | $\log n$ | $(\log n)^2 \log\log n$ | $\log n \log\log n$ | $\log n (\log\log n)^2$ | $\frac{\log n}{\log f(n)}$ |
| 5 | Uncover$(v, w, i)$ | $\log n$ | $(\log n)^2 \log\log n$ | $\log n \log\log n$ | $\log n (\log\log n)^2$ | $\frac{\log n}{\log f(n)}$ |
| 6 | CoverLevel$(v)$ | $\log n$ | $(\log n)^2 \log\log n$ | $\log n \log\log n$ | $\log n (\log\log n)^2$ | $\frac{\log n}{\log f(n)}$ |
| 7 | CoverLevel$(v, w)$ | $\log n$ | $(\log n)^2 \log\log n$ | $\log n \log\log n$ | $\log n (\log\log n)^2$ | $\frac{\log n}{\log f(n)}$ |
| 8 | MinCoveredEdge$(v)$ | $\log n$ | $(\log n)^2 \log\log n$ | $\log n \log\log n$ | $\log n (\log\log n)^2$ | $\frac{\log n}{\log f(n)}$ |
| 9 | MinCoveredEdge$(v, w)$ | $\log n$ | $(\log n)^2 \log\log n$ | $\log n \log\log n$ | $\log n (\log\log n)^2$ | $\frac{\log n}{\log f(n)}$ |
| 10 | AddLabel$(v, l, i)$ | - | - | $\log n \log\log n$ | - | - |
| 11 | RemoveLabel$(l)$ | - | - | $\log n \log\log n$ | - | - |
| 12 | FindFirstLabel$(v, w, i)$ | - | - | $\log n \log\log n$ | - | - |
| 13 | FindSize$(v, w, i)$ | - | $(\log n)^2 \log\log n$ | - | $\log n (\log\log n)^2$ | - |
| | FindSize$(v, v, -1)$ | $\log n$ | $\log n$ | $\log n$ | $\log n$ | $\frac{\log n}{\log f(n)}$ |

Table 1: Overview of the worst case times achieved for each tree operation by the data structures presented in this paper. In the last column, $f(n) \in \mathcal{O}(\frac{\log n}{\log\log n})$ can be chosen arbitrarily.

The algorithm in [11] used a dynamic tree structure supporting all the operations in $\mathcal{O}((\log n)^3)$ time, leading to an $\mathcal{O}((\log n)^4)$ algorithm for bridge finding. Thorup [20] showed how to improve the time for the dynamic tree structure to $\mathcal{O}((\log n)^2 \log\log n)$ leading to an $\mathcal{O}((\log n)^3 \log\log n)$ algorithm for bridge finding.

Throughout this paper, we will show a number of data structures for dynamic trees, implementing various subsets of these operations while ignoring the rest (See Table 1). Define a *CoverLevel* structure to be one that implements operations 1–9, and a *FindSize* structure to be a CoverLevel structure that additionally implements the FindSize operation. Finally, we define a *FindFirstLabel* structure to be one that implements operations 1–12 (all except for FindSize).

The point is that we can get different trade-offs between the operation costs in the different structures, and that we can combine them into a single structure supporting all the operations using the following

**Lemma 6** (Folklore)**.** *Given two data structures $S$ and $S'$ for the same problem consisting of a set $U$ of update operations and a set $Q$ of query operations. If the respective update times are $f_u(n)$ and $f'_u(n)$ for $u \in U$, and the query times are $g_q(n)$ and $g'_q(n)$ for $q \in Q$, we can create a combined data* *structure running in $\mathcal{O}(f_u(n) + f'_u(n))$ time for update operation $u \in U$, and $\mathcal{O}(\min\{g_q(n), g'_q(n)\})$ time for query operation $q \in Q$.*

*Proof.* Simply maintain both structures in parallel. Call all update operations on both structures, and call only the fastest structure for each query. □

*Proof of Theorem 3.* Use the CoverLevel structure from Section 4, the FindSize structure from Section 5, and the FindFirstLabel structure from Section 6, and combine them into a single structure using Lemma 6. Then the reduction from Lemma 5 gives the correct running times but uses $\mathcal{O}(m + n \log n)$ space. To get linear space, modify the FindSize and FindFirstLabel structures as described in Section 10. □

*Proof of Theorem 1.* Use the CoverLevel structure from Section 9, the FindSize structure from Section 5, as modified in Section 7 and 10, and the FindFirstLabel structure from Section 6, and combine them into a single structure using Lemma 6. Then the reduction from Lemma 5 gives the required bounds. □

## 3 Top trees

A *top tree* is a data structure for maintaining information about a dynamic forest. Given a tree $T$, a top tree $\mathcal{T}$ is a rooted tree over subtrees of $T$, such that each non-leaf node is the union of its children. The root of $\mathcal{T}$ is $T$, its leaves are the edges of $T$, and its nodes are *clusters*, which we will define in two steps. For any subgraph $H$ of a graph $G$, the boundary $\partial H$ consists of the vertices of $H$ that have a neighbour in $G \setminus H$. A *cluster* is a connected subgraph with a boundary of size no larger than 2. We denote them by *point clusters* if the boundary has size $\leq 1$, and *path clusters* otherwise. For a path cluster $C$ with boundary $\partial C = \{u, v\}$, denote by $\pi(C)$ the tree path between $u$ and $v$, also denoted *the cluster path of* $C$. Similarly, for a point cluster $C$ with boundary vertex $v$, $\pi(C)$ is the trivial path consisting solely of $v$. The top forest supports dynamic changes to the forest: insertion (link) or deletion (cut) of edges. Furthermore, it supports the *expose* operation: expose($v$), or expose($v_1, v_2$), returns a top tree where $v$, or $v_1, v_2$, are considered boundary vertices of every cluster containing them, including the root cluster. All operations are supported by performing a series of *destroy*, *create*, *split*, and *merge* operations: *split* destroys a node of the top tree and replaces it with its two children, while merge creates a parent as a union of its children. Destroy and create are the base cases for split and merge, respectively. Note that clusters can only be merged if their union has a boundary of size at most 2.

A top tree is *binary* if each node has at most two children. We call a non-leaf node *heterogeneous* if it has both a point cluster and a path cluster among its children, and *homogeneous* otherwise.

**Theorem 7** (Alstrup, Holm, de Lichtenberg, Thorup [1]). *For a dynamic forest on $n$ vertices we can maintain binary top trees of height $\mathcal{O}(\log n)$ supporting each link, cut or expose with a sequence of $\mathcal{O}(1)$ calls to create or destroy, and $\mathcal{O}(\log n)$ calls to merge or split. These top tree modifications are identified in $\mathcal{O}(\log n)$ time. The space usage of the top trees is linear in the size of the dynamic forest.*

## 4 A CoverLevel structure

In this section we show how to maintain a top tree supporting the CoverLevel operations. This part is is essentially the same as in [10,11] (with minor corrections), but is included here for completeness because the rest of the paper builds on it. Pseudocode for maintaining this structure is given in Appendix B.

For each cluster $C$ we want to maintain the following two integers and up to two edges:

$$\begin{aligned}
\text{cover}_C &:= \min\{c(e) \mid e \in \pi(C)\} \cup \{\ell_{\max}\} \\
\text{globalcover}_C &:= \min\{c(e) \mid e \in C \setminus \pi(C)\} \cup \{\ell_{\max}\} \\
\text{minpathedge}_C &:= \operatorname*{arg\,min}_{e\in\pi(C)} c(e) \;\text{ if } |\partial C| = 2, \text{ and } \textbf{nil} \text{ otherwise} \\
\text{minglobaledge}_C &:= \operatorname*{arg\,min}_{e\in C\setminus\pi(C)} c(e) \text{ if } C \neq \pi(C), \text{ and } \textbf{nil} \text{ otherwise}
\end{aligned}$$

Then

$$\left.\begin{aligned}
\text{CoverLevel}(v) &= \text{globalcover}_C \\
\text{MinCoveredEdge}(v) &= \text{minglobaledge}_C
\end{aligned}\right\} \text{where } C \text{ is the point cluster returned by Expose}(v)$$

$$\left.\begin{aligned}
\text{CoverLevel}(v,w) &= \text{cover}_C \\
\text{MinCoveredEdge}(v,w) &= \text{minpathedge}_C
\end{aligned}\right\} \text{where } C \text{ is the path cluster returned by Expose}(v,w)$$

The problem is that when handling Cover or Uncover we cannot afford to propagate the information all the way down to the edges. When these operations are called on a path cluster $C$, we instead implement them directly in $C$, and then store "lazy information" in $C$ about what should be propagated down in case we want to look at the descendants of $C$. The exact additional information we store for a path cluster $C$ is

$$\begin{aligned}
\text{cover}_C^- &:= \text{max level of a pending Uncover, or } -1 \\
\text{cover}_C^+ &:= \text{max level of a pending Cover, or } -1
\end{aligned}$$

We maintain the invariant that $\text{cover}_C \geq \text{cover}_C^+$, and if $\text{cover}_C \leq \text{cover}_C^-$ then $\text{cover}_C = \text{cover}_C^+$.

This allows us to implement Cover$(v, w, i)$ by first calling Expose$(v, w)$, and then updating the returned path cluster $C$ as follows:

$$\text{cover}_C = \max\{\text{cover}_C, i\} \qquad\qquad \text{cover}_C^+ = \max\left\{\text{cover}_C^+, i\right\}$$

Similarly, we can implement Uncover$(v, w, i)$ by first calling Expose$(v, w)$, and then updating the returned path cluster $C$ as follows if $\text{cover}_C \leq i$:

$$\text{cover}_C = -1 \qquad\qquad \text{cover}_C^+ = -1 \qquad\qquad \text{cover}_C^- = \max\left\{\text{cover}_C^-, i\right\}$$

Together, $\text{cover}_C^-$ and $\text{cover}_C^+$ represent the fact that for each path descendant $D$ of $C$, if $\text{cover}_D \leq \max\left\{\text{cover}_C^-, \text{cover}_C^+\right\}$[1], we need to set $\text{cover}_D = \text{cover}_C^+$. In particular whenever a path cluster $C$ is split, for each path child $D$ of $C$, if $\max\left\{\text{cover}_D, \text{cover}_D^-\right\} \leq \text{cover}_C^-$ we need to set

$$\text{cover}_D^- = \text{cover}_C^-$$

[1]In [10,11] this condition is erroneously stated as $\text{cover}_D \leq \text{cover}_C^-$.

Furthermore, if $\text{cover}_D \leq \max\left\{\text{cover}_C^-, \text{cover}_C^+\right\}$ we need to set

$$\text{cover}_D = \text{cover}_C^+ \qquad\qquad \text{cover}_D^+ = \text{cover}_C^+$$

Note that only $\text{cover}_D$ is affected. None of $\text{globalcover}_D$, $\text{minpathedge}_D$, or $\text{minglobaledge}_D$ depend directly on the lazy information.

Now suppose we have $k$ clusters[2] $A_1, \ldots, A_k$ that we want to merge into a single new cluster $C$. For $1 \leq i \leq k$ define

$$\text{globalcover}'_{C,A_i} := \begin{cases} \text{globalcover}_{A_i} & \text{if } \partial A_i \subseteq \pi(C) \text{ or } \text{globalcover}_{A_i} \leq \text{cover}_{A_i} \\ \text{cover}_{A_i} & \text{otherwise} \end{cases}$$

$$\text{minglobaledge}'_{C,A_i} := \begin{cases} \text{minglobaledge}_{A_i} & \text{if } \partial A_i \subseteq \pi(C) \text{ or } \text{globalcover}_{A_i} \leq \text{cover}_{A_i} \\ \text{minpathedge}_{A_i} & \text{otherwise} \end{cases}$$

Note that for a point-cluster $A_i$, $\text{globalcover}_{A_i}$ is always $\leq \text{cover}_{A_i} = l_{\max}$.

We then have the following relations between the data of the parent and the data of its children:

$$\begin{aligned}
\text{cover}_C &= \ell_{\max} \text{ if } |\partial C| < 2, \text{ otherwise } \min_{1\leq i<k, \partial A_i \subseteq \pi(C)} \text{cover}_{A_i} \\
\text{minpathedge}_C &= \textbf{nil} \text{ if } |\partial C| < 2, \text{ otherwise } \text{minpathedge}_{A_j} \text{ where } j = \operatorname*{arg\,min}_{1\leq i<k, \partial A_i \subseteq \pi(C)} \text{cover}_{A_i} \\
\text{globalcover}_C &= \min_{1\leq i<k} \text{globalcover}'_{C,A_i} \\
\text{minglobaledge}_C &= \text{minglobaledge}'_{C,A_j} \quad \text{where } j = \operatorname*{arg\,min}_{1\leq i<k} \text{globalcover}'_{C,A_i} \\
\text{cover}_C^- &= -1 \\
\text{cover}_C^+ &= -1
\end{aligned}$$

**Analysis** For any constant-degree top tree, Merge and Split with this information takes constant time, and thus, all operations in the CoverLevel structure in this section take $\mathcal{O}(\log n)$ time. Each cluster uses $\mathcal{O}(1)$ space, so the total space used is $\mathcal{O}(n)$.

Note that we can extend this so for each cluster $C$, if all the least-covered edges (on or off the cluster path) lie in the same child of $C$, we have a pointer to the closest descendant $D$ of $C$ that is either a base cluster or has more than one child containing least-covered edges. We can use this structure to find the first $k$ bridges in $\mathcal{O}(\log n + k)$ time.

## 5 A FindSize Structure

We now proceed to show how to extend the CoverLevel structure from Section 4 to support FindSize in $\mathcal{O}(\log n \log\log n)$ time per Merge and Split. Later, in Section 7 we will show how to reduce this to $\mathcal{O}((\log\log n)^2)$ time per Merge and Split. See Appendix C for pseudocode.

We will use the idea of having a single *vertex label* for each vertex, which is a point cluster with no edges, having that vertex as boundary vertex and containing all relevant information about the vertex. The advantage of this is that it simplifies handling of the common boundary vertex during a merge by making sure it is uniquely assigned to (and accounted for by) one of the children.

[2] $k = 2$ for now, but we will reuse this in section 9 with a higher-degree top tree.

Let $C$ be a cluster in $T$, let $v$ be a vertex in $C$, and let $0 \leq i < \ell_{\max}$. Define

$$\text{pointsize}_{C,v,i} := \bigl|\{u \in C \mid \text{CoverLevel}(u,v) \geq i\}\bigr|$$

For convenience, we will combine all the $\mathcal{O}(\log n)$ levels together into a single vector[3]

$$\text{pointsize}_{C,v} := \bigl(\text{pointsize}_{C,v,i}\bigr)_{\{0 \leq i < \ell_{\max}\}}$$

Let $(C_v)_{\{v \in \pi(C)\}}$ be the point clusters that would result from deleting the edges of $\pi(C)$ from $C$. Then we can define the vector

$$\text{size}_C := \sum_{m \in \pi(C)} \text{pointsize}_{C_m,m}$$

Note that with this definition, if $\partial C = \{v\}$ then $\text{pointsize}_{C,v} = \text{size}_C$ so even when $v = w$ we have

$$\text{FindSize}(v,w,i) = \text{size}_{C,i} \qquad \text{where } C = \text{Expose}(v,w)$$

So for any cluster $C$, the $\text{size}_C$ vector is what we want to maintain.

The main difficulty turns out be computing the $\text{size}_C$ vector for the heterogeneous point clusters. To help with that we will for each cluster $C$ and boundary vertex $v \in \partial C$ additionally maintain the following two size vectors for each $-1 \leq i \leq \ell_{\max}$:

$$\text{partsize}_{C,v,i} := \sum_{\substack{m \in \pi(C) \\ \text{CoverLevel}(v,m)=i}} \text{pointsize}_{C_m,m} \qquad\qquad \text{diagsize}_{C,v,i} := M(i) \cdot \text{partsize}_{C,v,i}$$

Where $M(i)$ is a diagonal matrix whose entries are defined (using Iverson brackets, see [14]) by

$$M(i)_{jj} = [i \geq j]$$

Note that these vectors are independent of $\text{cover}_C^-$ and $\text{cover}_C^+$ as defined in Section 4. The corresponding "clean" vectors are not explicitly stored, but computed when needed as follows

$$\left.\begin{aligned}
\text{partsize}'_{C,v,i} &= \begin{cases} \text{partsize}_{C,v,i} & \text{if } i > \ell \\ \sum_{j=-1}^{\ell} \text{partsize}_{C,v,j} & \text{if } i = \text{cover}_C^+ \\ 0 & \text{otherwise} \end{cases} \\
\text{diagsize}'_{C,v,i} &= \begin{cases} \text{diagsize}_{C,v,i} & \text{if } i > \ell \\ M(i) \cdot \sum_{j=-1}^{\ell} \text{partsize}_{C,v,j} & \text{if } i = \text{cover}_C^+ \\ 0 & \text{otherwise} \end{cases}
\end{aligned}\right\} \text{where } \ell = \max\left\{\text{cover}_C^-, \text{cover}_C^+\right\}$$

The point of these definitions is that each path cluster inherits most of its partsize and diagsize vectors from its children, and we can use this fact to get an $\mathcal{O}(\ell_{\max}/\log \ell_{\max}) = \mathcal{O}(\log n/\log\log n)$ speedup compared to [11].

[3]All vectors and matrices in this section have indices ranging from 0 to $\ell_{\max} - 1$.

**Merging along a path (the general case)** Let $A, B$ be clusters that we want to merge into a new cluster $C$, and suppose $\partial A \cup \partial B \subseteq \pi(C)$. This covers all types of merge in a normal binary top tree, except for the heterogeneous point clusters. Let $\partial A \cap \partial B = \{c\}$. If $|\partial C| = 1$, let $a = b = c$, otherwise let $\partial C = \{a, b\}$ with $a \in \partial A$, $b \in \partial B$. Then

$$\begin{aligned}
\text{size}_C &= \text{size}_A + \text{size}_B \\
\text{partsize}_{C,a,i} &= \begin{cases} \text{partsize}'_{A,a,i} & \text{if } i > \text{cover}_A \\ \text{partsize}'_{A,a,i} + \sum_{j=i}^{\ell_{\max}} \text{partsize}'_{B,c,j} & \text{if } i = \text{cover}_A \\ \text{partsize}'_{B,c,i} & \text{if } i < \text{cover}_A \end{cases} \\
\text{diagsize}_{C,a,i} &= \begin{cases} \text{diagsize}'_{A,a,i} & \text{if } i > \text{cover}_A \\ \text{diagsize}'_{A,a,i} + M(i) \cdot \sum_{j=i}^{\ell_{\max}} \text{partsize}'_{B,c,j} & \text{if } i = \text{cover}_A \\ \text{diagsize}'_{B,c,i} & \text{if } i < \text{cover}_A \end{cases}
\end{aligned}$$

**Merging off the path (heterogeneous point clusters)** Now let $A$ be a path cluster with $\partial A = \{a, b\}$, let $B$ be a point cluster with $\partial B = \{b\}$, and suppose we want to merge $A, B$ into a new point cluster $C$ with $\partial C = \{a\}$. Then

$$\begin{aligned}
\text{size}_C &= \left( \sum_{i=-1}^{\ell_{\max}} \text{diagsize}'_{A,a,i} \right) + M(\text{cover}_A) \cdot \text{size}_B \\
\text{partsize}_{C,a,i} &= \begin{cases} \text{size}_C & \text{if } i = \ell_{\max} \\ 0 & \text{otherwise} \end{cases} \\
\text{diagsize}_{C,a,i} &= \text{partsize}_{C,a,i}
\end{aligned}$$

**Analysis** The advantage of our new approach is that each merge or split is a *constant* number of splits, concatenations, searches, and sums over $\mathcal{O}(\ell_{\max})$-length lists of $\ell_{\max}$-dimensional vectors. By representing each list as an augmented balanced binary search tree (see e.g. [15, pp. 471–475]), we can implement each of these operations in $\mathcal{O}(\ell_{\max} \log \ell_{\max})$ time, and using $\mathcal{O}(\ell_{\max})$ space per cluster, as follows. Let $C$ be a cluster and let $v \in \partial C$. The tree has one node for each key $i, -1 \le i \le \ell_{\max}$ such that $\text{partsize}_{C,v,i}$ is nonzero, augmented with the following additional information:

$$\begin{aligned}
\text{key} &:= i \\
\text{partsize} &:= \text{partsize}_{C,v,i} \\
\text{diagsize} &:= \text{diagsize}_{C,v,i} \\
\text{partsizesum} &:= \sum_{j \text{ descendant of } i} \text{partsize}_{C,v,j} \\
\text{diagsizesum} &:= \sum_{j \text{ descendant of } i} \text{partsize}_{C,v,j}
\end{aligned}$$

Each split, concatenate, search, or sum operation can be implemented such that it touches $\mathcal{O}(\log \ell_{\max})$ nodes, and the time for each node update is dominated by the time it takes to add two $\ell_{\max}$-dimensional vectors, which is $\mathcal{O}(\ell_{\max})$. The total time for each Cover, Uncover, Link, Cut, or FindSize is therefore $\mathcal{O}(\log n \cdot \ell_{\max} \cdot \log \ell_{\max}) = \mathcal{O}((\log n)^2 \log \log n)$, and the total space used for the structure is $\mathcal{O}(n \cdot \ell_{\max}) = \mathcal{O}(n \log n)$.

**Comparison to previous algorithms** For any path cluster $C$ and vertex $v \in \partial C$, let $S_{C,v}$ be the matrix whose $j$th column $0 \le j < \ell_{\max}$ is defined by

$$(S_{C,v}^T)_j := \sum_{k=j}^{\ell_{\max}} \text{partsize}'_{C,v,k}$$

Then $S_{C,v}$ is essentially the size matrix maintained for path clusters in [10, 11, 20]. Notice that

$$\text{diag}(S_{C,v}) = \sum_{k=-1}^{\ell_{\max}} \text{diagsize}'_{C,v,k}$$

which explains our choice of the "diag" prefix.

## 6 A FindFirstLabel Structure

We will show how to maintain information that allows us to implement FindFirstLabel; the function that allows us to inspect the replacement edge candidates at a given level. The implementation uses a "destructive binary search, with undo" strategy, similar to the non-local search introduced in [1].

The idea is to maintain enough information in each cluster to determine if there is a result. Then we can start by using Expose$(v, w)$, and repeatedly split the root containing the answer until we arrive at the correct label. After that, we simply undo the splits (using the appropriate merges), and finally undo the Expose.

Just as in the FindSize structure, we will use vertex labels to store all the information pertinent to a vertex. We store all the added *user labels* for each vertex in the label object for that vertex in the base level of the top tree. For each level where the vertex has an associated user label, we keep a doubly linked list of those labels, and we keep a singly-linked list of these nonempty lists. Thus, FindFirstLabel$(v, w, i)$ boils down to finding the first vertex label that has an associated user label at the right level. Once we have that vertex label, the desired user label can be found in $\mathcal{O}(\ell_{\max})$ time.

Let $C$ be a cluster in $T$, and let $v \in \partial C$. Define bit vectors[4]

$$\text{pointincident}_{C,v} := \left( \left[ \exists \text{label } l \in C\colon \begin{array}{c} \text{CoverLevel}(v, \text{vertex}(l)) = i \\ \wedge \quad \ell(l) = i \end{array} \right] \right)_{\{0 \le i < \ell_{\max}\}}$$

$$\text{incident}_C := \bigvee_{m \in \pi(C)} \text{pointincident}_{C_m, m}$$

Maintaining the incident$_C$ bit vectors, and the corresponding partincident$_{C,v}$ and diagincident$_{C,v}$ bit vectors, can be done completely analogous to the way we maintain the size vectors used for FindSize, with the minor change that we use bitwise OR on bit vectors instead of vector addition.

Updating the vertex label cluster $C$ in the top tree during AddLabel$(v, l, i)$, or a RemoveLabel$(l)$ where $v = \text{vertex}(l)$ and $\ell(l) = i$ can be done by first calling detach$(C)$, then updating the linked

[4]Here, $[P] = \begin{cases} 1 & \text{if } P \text{ is true} \\ 0 & \text{otherwise} \end{cases}$ is the *Iverson Bracket* (see [14]), and $\vee$ denotes bitwise OR.

lists containing the user labels and setting

$$\begin{aligned}
\text{incident}_C &= ([v \text{ has associated labels at level } j])_{\{0\le j<\ell_{\max}\}} \\
\text{partincident}_{C,\text{vertex}(l),i} &= \begin{cases} \text{incident}_C & \text{if } i = \ell_{\max} \\ 0 & \text{otherwise} \end{cases} \\
\text{diagincident}_{C,\text{vertex}(l)} &= \text{partincident}_C
\end{aligned}$$

and then reattaching $C$. Finally FindFirstLabel(v,w,i) can be implemented in the way already described, by examining $\text{pointincident}_{C,v,i}$ for each cluster. Note that even though we don't explicitly maintain it, for any cluster $C$ and any $v \in \partial C$ we can easily compute

$$\begin{aligned}
\text{pointincident}_{C,v} &= \bigvee_{i=-1}^{\ell_{\max}} \text{diagincident}'_{C,v,i} \\
&= \left(\bigvee_{i=\ell+1}^{\ell_{\max}} \text{diagincident}_{C,v,i}\right) + M(\text{cover}^+_C) \cdot \left(\bigvee_{i=-1}^{\ell} \text{partincident}_{C,v,i}\right) \\
&\quad \text{Where } \ell := \max\left\{\text{cover}^-_C, \text{cover}^+_C\right\}
\end{aligned}$$

In general, let $A_1, \dots A_k$ be the clusters resulting from an expose or split, let $v, w \in \bigcup_{i=1}^k \partial A_i$ (not necessarily distinct). Then we can define

$$\begin{aligned}
\text{FindFirstLabel}((A_1, \cdots, A_k); v, w, i) &= \begin{cases} A_x & \text{if } A_x \text{ is a label} \\ \text{FindFirstLabel}(\text{Split}(A_x); v_x, w_x, i) & \text{otherwise} \end{cases} \\
&\quad \text{where for } 1 \le j \le k \\
&\qquad v_j = \operatorname*{arg\,min}_{u\in\partial A_j} \text{dist}(v, u) \\
&\qquad w_j = \operatorname*{arg\,max}_{u\in\partial A_j} \text{dist}(u, w) \\
&\quad \text{and} \\
&\qquad I = \left\{ 1 \le j \le k \;\middle|\; \begin{array}{c} \text{CoverLevel}(v, v_j) \ge i \\ \wedge \quad \text{pointincident}_{A_j,v_j,i} = 1 \end{array} \right\} \\
&\qquad x = \operatorname*{arg\,min}_{j\in I} \left(3 \cdot \text{dist}(v, \text{meet}(v_j, v, w)) + |\partial A_j \cap v \cdots w|\right) \\
\text{FindFirstLabel}(v, w, i) &= \text{FindFirstLabel}(\text{Expose}(v, w); v, w, i)
\end{aligned}$$

**Analysis** By the method described in this section, AddLabel, RemoveLabel, and FindFirstLabel are maintained in $\mathcal{O}(\log n \cdot \ell_{\max} \cdot \log \ell_{\max}) = \mathcal{O}((\log n)^2 \log\log n)$ worst-case time.

This can be reduced to $\mathcal{O}(\log n \cdot \log \ell_{\max}) = \mathcal{O}(\log n \log\log n)$ by realizing that each $\ell_{\max}$-dimensional bit vector fits into $\mathcal{O}(1)$ words, and that each bitwise OR therefore only takes constant time.

The total space used for a FindFirstLabel structure with $n$ vertices and $m$ labels is $\mathcal{O}(m + n)$ plus the space for $\mathcal{O}(n)$ bit vectors. If we assume a word size of $\Omega(\log n)$, this is just $\mathcal{O}(m + n)$ in total. If we disallow bit packing tricks, we may have to use $\mathcal{O}(m + n \cdot \ell_{\max}) = \mathcal{O}(m + n \log n)$ space.

## 7 Approximate counting

As noted in [20], we don't need to use the exact component sizes at each level. If $s$ is the actual correct size, it is sufficient to store an approximate value $s'$ such that $s' \leq s \leq e^{\epsilon} s'$, for some constant $0 < \epsilon < \ln 2$. Then we are no longer guaranteed that component sizes drop by a factor of $\frac{1}{2}$ at each level, but rather get a factor of $\frac{e^{\epsilon}}{2}$. This increases the number of levels to $\ell_{\max} = \lfloor \ln n / (\ln 2 - \epsilon) \rfloor$ (which is still $\mathcal{O}(\log n)$), but leaves the algorithm otherwise unchanged. Suppose we represent each size as a floating point value with a $b$-bit mantissa, for some $b$ to be determined later. For each addition of such numbers the relative error increases. The relative error at the root of a tree of additions of height $h$ is $(1 + 2^{-b})^h \leq e^{2^{-b} h}$, thus to get the required precision it is sufficient to set $b = \log_2 \frac{h}{\epsilon}$. In our algorithm(s) the depth of calculation is clearly upper bounded by $h \leq h(n) \cdot \ell_{\max}$, where $h(n) = \mathcal{O}(\log n)$ is the height of the top tree. It follows that some $b \in \mathcal{O}(\log \log n)$ is sufficient. Since the maximum size of a component is $n$, the exponent has size at most $\lceil \log_2 n \rceil$, and can be represented in $\lceil \log_2 \lceil \log_2 n \rceil \rceil$ bits. Thus storing the sizes as $\mathcal{O}(\log \log n)$ bit floating point values is sufficient to get the required precision. Assuming a word size of $\Omega(\log n)$ this lets us store $\mathcal{O}(\frac{\log n}{\log \log n})$ sizes in a single word, and to add them in parallel in constant time.

**Analysis** We will show how this applies to our FindSize structure from Section 5. The bottlenecks in the algorithm all have to do with operations on $\ell_{\max}$-dimensional size vectors. In particular, the amortized update time is dominated by the time to do $\mathcal{O}(\log n \cdot \log \ell_{\max})$ vector additions, and $\mathcal{O}(\log n)$ multiplications of a vector by the $M(i)$ matrix. With approximate counting, the vector additions each take $\mathcal{O}(\log \log n)$ time. Multiplying a size vector $x$ by $M(i)$ we get:

$$(M(i) \cdot x)_j = \begin{cases} x_j & \text{if } i \geq j \\ 0 & \text{otherwise} \end{cases}$$

And clearly this operation can also be done on $\mathcal{O}(\frac{\log n}{\log \log n})$ sizes in parallel when they are packed into a single word. With approximate counting, each multiplication by $M(i)$ therefore also takes $\mathcal{O}(\log \log n)$ time. Thus the time per operation is reduced to $\mathcal{O}(\log n (\log \log n)^2)$.

The space consumption of the data structure is $\mathcal{O}(n)$ plus the space needed to store $\mathcal{O}(n)$ of the $\ell_{\max}$-dimensional size vectors. With approximate counting that drops to $\mathcal{O}(\log \log n)$ per vector, or $\mathcal{O}(n \log \log n)$ in total.

**Comparison to previous algorithms** Combining the modified FindSize structure with the CoverLevel structure from Section 4 and the FindFirstLabel structure from Section 6 gives us the first bridge-finding structure with $\mathcal{O}((\log n)^2 (\log \log n)^2)$ amortized update time. This structure uses $\mathcal{O}(m + n \log \log n)$ space, and uses $\mathcal{O}(\log n)$ time for FindBridge and Size queries, and $\mathcal{O}(\log n (\log \log n)^2)$ for 2-size queries.

For comparison, applying this trick in the obvious way to the basic $\mathcal{O}((\log n)^4)$ time and $\mathcal{O}(m + n(\log n)^2)$ algorithm from [10,11] gives the $\mathcal{O}((\log n)^3 \log n)$ time and $\mathcal{O}(m + n \log n \log \log n)$ algorithm briefly mentioned in [20].

## 8 Top trees revisited

We can combine the tree data structures presented so far to build a data structure for bridge-finding that has update time $\mathcal{O}((\log n)^2 (\log \log n)^2)$, query time $\mathcal{O}(\log n)$, and uses $\mathcal{O}(m + n \log \log n)$ space.

In order to get faster queries and linear space, we need to use top-trees in an even smarter way. For this, we need the full generality of the top trees described in [1].

## 8.1 Level-based top trees, labels, and fat-bottomed trees

As described in [1], we may associate a level with each cluster, such that the leaves of the top tree have level 0, and such that the parent of a level $i$ cluster is on level $i+1$. As observed in Alstrup et al. [1, Theorem 5.1], one may also associate one or more *labels* with each vertex. For any vertex, $v$, we may handle the label(s) of $v$ as point clusters with $v$ as their boundary vertex and no edges. Furthermore, as described in [1], we need not have single edges on the bottom most level. We may generalize this to instead have clusters of size $Q$ as the leaves of the top tree.

**Theorem 8** (Alstrup, Holm, de Lichtenberg, Thorup [1]). *Consider a fully dynamic forest and let $Q$ be a positive integer parameter. For the trees in the forest, we can maintain levelled top trees whose base clusters are of size at most $Q$ and such that if a tree has size $s$, it has height $h = \mathcal{O}(\log s)$ and $\lceil \mathcal{O}(s/(Q(1+\varepsilon)^i)) \rceil$ clusters on level $i \leq h$. Here, $\varepsilon$ is a positive constant. Each link, cut, attach, detach, or expose operation is supported with $\mathcal{O}(1)$ creates and destroys, and $\mathcal{O}(1)$ joins and splits on each positive level. If the involved trees have total size $s$, this involves $\mathcal{O}(\log s)$ top tree modifications, all of which are identified in $\mathcal{O}(Q + \log s)$ time. For a composite sequence of $k$ updates, each of the above bounds are multiplied by $k$. As a variant, if we have parameter $S$ bounding the size of each underlying tree, then we can choose to let all top roots be on the same level $H = \mathcal{O}(\log S)$.*

## 8.2 High degree top trees

Top trees of degree two are well described and often used. However, it turns out to be useful to also consider top trees of higher degree $B$, especially for $B \in \omega(1)$.

**Lemma 9.** *Given any $B \geq 2$, one can maintain top trees of degree $B$ and height $\mathcal{O}(\log n/\log B)$. Each expose, link, or cut is handled by $\mathcal{O}(1)$ calls to create or destroy and $\mathcal{O}(\log n/\log B)$ calls to split or merge. The operations are identified in $\mathcal{O}(B(\log n/\log B))$ time.*

*Proof.* Given a binary levelled top tree $\mathcal{T}_2$ of height $h$, we can create a $B$-ary levelled top tree $\mathcal{T}_B$, where the leaves of $\mathcal{T}_B$ are the leaves of $\mathcal{T}_2$, and where the clusters on level $i$ of $\mathcal{T}_B$ are the clusters on level $i \cdot \lfloor \log_2 B \rfloor$ of $\mathcal{T}_2$. Edges in $\mathcal{T}_B$ correspond to paths of length $\lfloor \log_2 B \rfloor$ in $\mathcal{T}_2$. Thus, given a binary top tree, we may create a $B$-ary top tree bottom-up in linear time.

We may implement link, cut and expose by running the corresponding operation in $\mathcal{T}_2$. Each cut, link or expose operation will affect clusters on a constant number of root-paths in $\mathcal{T}_2$. There are thus only $\mathcal{O}(\log n/\log B)$ calls to split or merge of a cluster on a level divisible by $\lfloor \log_2 B \rfloor$. Thus, since each split or merge in $\mathcal{T}_B$ corresponds to a split or merge of a cluster in $\mathcal{T}_2$ whose level is divisible by $\lfloor \log_2 B \rfloor$, we have only $\mathcal{O}(\log n/\log B)$ calls to split and merge in $\mathcal{T}_B$.

However, since there are $\mathcal{O}(B)$ clusters whose parent pointers need to be updated after a merge, the total running time becomes $\mathcal{O}(B(\log n/\log B))$. □

## 8.3 Saving space with fat-bottomed top trees

In this section we present a general technique for reducing the space usage of a top tree based data structure to linear. The properties of the technique are captured in the following:

**Lemma 10.** *Given a top tree data structure of height $h(n) \in \mathcal{O}(\log n)$ that uses $s(n)$ space per cluster, and $t(n)$ worst case time per merge or split.*

*Suppose that the complete information for a cluster of size $q$, including information that is shared with its children, has total size $s_0(q,n)$ and can be computed directly in time $t_0(q,n)$. Suppose further that there exists a function $q$ of $n$ such that $s(n) < s_0(q(n),n) \in \mathcal{O}(q(n))$.*

*Then there exists a top tree data structure, maintaining the same information, that uses linear space in total and has $\mathcal{O}(t(n) \cdot h(n) + t_0(q(n),n))$ update time for link, cut, and expose.*

*Proof.* This follows directly from Theorem 8 by setting $Q = q(n)$. Then the top tree will have $\mathcal{O}(n/q(n))$ clusters of size at most $s_0(q(n),n) = \mathcal{O}(q(n))$ so the total size is linear. The time per update follows because the top tree uses $\mathcal{O}(h(n))$ merges of split and $\mathcal{O}(1)$ create and destroy per link cut and expose. These take $t(n)$ and $t_0(q(n),n)$ time respectively. $\square$

## 9 A Faster CoverLevel Structure

If we allow ourselves to use bit tricks, we can improve the CoverLevel data structure from Section 4. The main idea is, for some $0 < \epsilon < 1$, to use top trees of degree $b(n) = (\log n)^\epsilon \in \mathcal{O}(w/\log \ell_{\max})$. Such top trees have height $h(n) \in \mathcal{O}(\frac{\log n}{\epsilon \log\log n})$, and finding the sequence of merges and splits for a given link, cut or expose takes $\mathcal{O}(b(n) \cdot h(n)) \in \mathcal{O}(\frac{(\log n)^{1+\epsilon}}{\epsilon \log\log n}) \subseteq o((\log n)^{1+\epsilon})$ time.

The high-level algorithm makes at most a constant number of calls to link and cut for each insert or delete, so we are fine with the time for these operations. However, we can no longer use Expose to implement Cover, Uncover, CoverLevel and MinCoveredEdge, as that would take too long.

In this section, we will show how to overcome this limitation by working directly with the underlying tree.

**The data** The basic idea is to maintain a *buffer* with all the cover, cover$^-$, cover$^+$ and globalcover values one level up in the tree, in the parent cluster. Since the degree is $\mathcal{O}(w/\log \ell_{\max})$, and each value uses at most $\mathcal{O}(\log \ell_{\max})$ bits, these fit into a constant number of words, and so we can use bit tricks to operate on the values for all children of a node in parallel.

Let $C$ be a cluster with children $A_1, \ldots, A_k$. Since $k \leq w/\log \ell_{\max}$, we can define the following vectors that each fit into a constant number of words.

$$\begin{aligned}
\text{packedcover}_C &:= (\text{cover}_{A_i})_{\{1 \leq i \leq k\}} \\
\text{packedcover}^-_C &:= (\text{cover}^-_{A_i})_{\{1 \leq i \leq k\}} \\
\text{packedcover}^+_C &:= (\text{cover}^+_{A_i})_{\{1 \leq i \leq k\}} \\
\text{packedglobalcover}_C &:= (\text{globalcover}_{A_i})_{\{1 \leq i \leq k\}}
\end{aligned}$$

The description of Split and Merge from Section 4 still apply, if we think of the "packed" values as a separate layer of degree 1 clusters between each pair of "real" clusters.

For concreteness, let $C$ be a cluster with children $A_1, \ldots, A_k$, and define operations

- CleanToBuffer($C$). For each $1 \leq i \leq k$: If $A_i$ is a path child of $C$ and $\max\left\{\text{packedcover}_{C,i}, \text{packedcover}^-_{C,i}\right\} \leq \text{cover}^-_C$, set:

$$\text{packedcover}^-_{C,i} = \text{cover}^-_C$$

Then if $\text{packedcover}_{C,i} \leq \max\left\{\text{cover}^-_C, \text{cover}^+_C\right\}$ set

$$\begin{aligned}\text{packedcover}_{C,i} &= \text{cover}^+_C \\ \text{packedcover}^+_{C,i} &= \text{cover}^+_C\end{aligned}$$

After updating all $k$ children, set $\text{cover}^-_C = \text{cover}^+_C = -1$. Note that this can be done in parallel for all $1 \leq i \leq k$ in constant time using bit tricks.

- CleanToChild$(C, i)$. If $A_i$ is a path child of $C$ and $\max\left\{\text{cover}_{A_i}, \text{cover}^-_{A_i}\right\} \leq \text{packedcover}^-_{C,i}$, set

$$\text{cover}^-_{A_i} = \text{packedcover}^-_{C,i}$$

Then if $\text{cover}_{A_i} \leq \max\left\{\text{packedcover}^-_{C,i}, \text{packedcover}^+_{C,i}\right\}$ set

$$\begin{aligned}\text{cover}_{A_i} &= \text{packedcover}^+_{C,i} \\ \text{cover}^+_{A_i} &= \text{packedcover}^+_{C,i}\end{aligned}$$

Finally set $\text{packedcover}^-_{C,i} = \text{packedcover}^+_{C,i} = -1$. Again, note that this takes constant time.

- ComputeFromChild$(C, i)$. Set

$$\begin{aligned}\text{packedcover}_{C,i} &= \text{cover}_{A_i} \\ \text{packedcover}^-_{C,i} &= -1 \\ \text{packedcover}^+_{C,i} &= -1 \\ \text{packedglobalcover}_{C,i} &= \text{globalcover}_{A_i}\end{aligned}$$

- ComputeFromBuffer$(C)$. For $1 \leq i \leq k$ define

$$\text{packedglobalcover}'_{C,i} = \begin{cases} \text{packedglobalcover}_{C,i} & \text{if } \partial A_i \subseteq \pi(C) \\ & \text{or } \text{packedglobalcover}_{C,i} \leq \text{packedcover}_{C,i} \\ \text{packedcover}_{C,i} & \text{otherwise} \end{cases}$$

$$\text{minglobaledge}'_{C,i} = \begin{cases} \text{minglobaledge}_{A_i} & \text{if } \partial A_i \subseteq \pi(C) \\ & \text{or } \text{globalcover}_{A_i} \leq \text{cover}_{A_i} \\ \text{minpathedge}_{A_i} & \text{otherwise} \end{cases}$$

We can then compute the data for $C$ from the buffer as follows:

$$\begin{aligned}
\mathrm{cover}_C &= \begin{cases} \displaystyle\min_{\substack{1\le i<k\\ \partial A_i\subseteq\pi(C)}} \mathrm{packedcover}_{C,i} & \text{if } |\partial C|=2\\ \ell_{\max} & \text{otherwise}\end{cases}\\
\mathrm{minpathedge}_C &= \begin{cases} \mathrm{minpathedge}_{A_j} & \text{if } |\partial C|=2\\ \quad\text{where } j=\displaystyle\arg\min_{\substack{1\le i<k\\ \partial A_i\subseteq\pi(C)}} \mathrm{packedcover}_{C,i} & \\ \mathbf{nil} & \text{otherwise}\end{cases}\\
\mathrm{globalcover}_C &= \min_{1\le i<k} \mathrm{packedglobalcover}'_{C,i}\\
\mathrm{minglobaledge}_C &= \mathrm{minglobaledge}'_{C,j}\\
&\qquad\text{where } j=\arg\min_{1\le i<k} \mathrm{packedglobalcover}'_{C,i}\\
\mathrm{cover}^-_C &= -1\\
\mathrm{cover}^+_C &= -1
\end{aligned}$$

This can be computed in constant time, because $(\mathrm{packedglobalcover}'_{C,i})_{\{1\le i\le k\}}$ fits into a constant number of words that can be computed in constant time using bit tricks, and thus each "min" or "arg min" is taken over values packed into a constant number of words.

Then Split$(C)$ can be implemented by first calling CleanToBuffer$(C)$, and then for each $1\le i\le k$ calling CleanToChild$(C,i)$. This ensures that all the lazy cover information is propagated down correctly. Similarly, Merge$(C;A_1,\dots,A_k)$ can be implemented by first calling ComputeFromChild$(C,i)$ for each $1\le i\le k$, and then calling ComputeFromBuffer$(C)$. Thus Split and Merge each take $\mathcal{O}(b(n))$ time.

**Computing** CoverLevel$(v)$ **and** MinCoveredEdge$(v)$ With the data described in the previous section, we can now answer the "global" queries as follows

$$\begin{aligned}
\mathrm{CoverLevel}(v) &= \mathrm{globalcover}_C\\
\mathrm{MinCoveredEdge}(v) &= \mathrm{minglobaledge}_C\\
&\qquad\text{where } C \text{ is the point cluster returned by root}(v)
\end{aligned}$$

Note that, for simplicity, we assume the top tree always has a single vertex exposed. This can easily be arranged by a constant number of calls to Expose after each link or cut, without affecting the asymptotic running time. Computing CoverLevel$(v)$ or MinCoveredEdge$(v)$ therefore takes $\mathcal{O}(h(n))$ worst case time.

**Computing** CoverLevel$(v,w)$ **and** MinCoveredEdge$(v,w)$ Since we can no longer use Expose to implement Cover and Uncover, we need a little more machinery.

What saves us is that all the information we need to find CoverLevel$(v,w)$ is stored in the $\mathcal{O}(h(n))$ clusters that have $v$ or $v$ as internal vertices, and that once we have that, we can find a single child $X$ of one of these clusters such that MinCoveredEdge$(v,w)=\mathrm{minpathedge}_X$.

Before we get there, we have to deal with the complication of cover$^-$ and cover$^+$. Fortunately, all we need to do is make $\mathcal{O}(h(n))$ calls to CleanToBuffer and CleanToChild, starting from the root and going down towards $v$ and $w$. Since each of these calls take constant time, we use only $\mathcal{O}(h(n))$ time on cleaning.

Now, the path $v \cdots w$ consists of $\mathcal{O}(h(n))$ edge-disjoint fragments, such that:

- Each fragment $f$ is associated with, and contained in, a single cluster $C_f$.
- For each fragment $f$, the endpoints are either in $\{v, w\}$ (and then $C_f$ is a base cluster) or are boundary vertices of children of $C_f$.

We can find the fragments in $\mathcal{O}(h(n))$ time, and for each fragment $f$, we can in constant time find its cover level by examining $\text{packedcover}_{C_f}$.

Let $f_1, \ldots, f_k$ be the fragments of the path, and for $1 \le i \le k$ let $v_i, w_i$ be the endpoints of the fragment closest to $v, w$ respectively. Then

$$\begin{aligned}
\text{CoverLevel}(v,w) &= \min_{1\le i\le k} \text{CoverLevel}(v_i, w_i) \\
\text{MinCoveredEdge}(v,w) &= \text{MinCoveredEdge}(v_j, w_j) \\
&\quad \text{where } j = \arg\min_{1\le i\le k} \text{CoverLevel}(v_i, w_i) \\
\text{MinCoveredEdge}(v_j,w_j) &= \text{minpathedge}_X \\
&\quad \text{where } X = \arg\min_{Y \text{ path child of } C_{f_j}} \text{cover}_Y
\end{aligned}$$

So computing $\text{CoverLevel}(v, w)$ or $\text{MinCoveredEdge}(v, w)$ takes $\mathcal{O}(h(n))$ worst case time.

**Cover and Uncover** We are now ready to handle $\text{Cover}(v, w, i)$ and $\text{Uncover}(v, w, i)$. First we make $\mathcal{O}(h(n))$ calls to CleanToBuffer and CleanToChild. Then let $f_1, \ldots, f_k$ be the fragments of the $v \cdots w$ path, and for $1 \le i \le k$ let $v_i, w_i$ be the endpoints of the fragment closest to $v, w$ respectively. Then for each $f \in f_1, \ldots, f_k$, and each path child $A_j$ of $C_f$, $\text{Cover}(v, w, i)$ needs to set

$$\begin{aligned}
\text{packedcover}_{C_f,j} &= \max\left\{\text{packedcover}_{C_f,j}, i\right\} \\
\text{packedcover}^+_{C_f,j} &= \max\left\{\text{packedcover}_{C_f,j}, i\right\}
\end{aligned}$$

Similarly, for each $f \in f_1, \ldots, f_k$, and for each path child $A_j$ of $C_f$, if $\text{packedcover}_{C_f,j} \le i$, $\text{Uncover}(v, w, i)$ needs to set

$$\begin{aligned}
\text{packedcover}_{C_f,j} &= -1 \\
\text{packedcover}^+_{C_f,j} &= -1 \\
\text{packedcover}^-_{C_f,j} &= \max\left\{\text{packedcover}^-_{C_f,j}, i\right\}
\end{aligned}$$

In each case, we can use bit tricks to make this take constant time per fragment. Finally, we need to update all the $\mathcal{O}(h(n))$ ancestors to the clusters we just changed. We can do this bottom-up using $\mathcal{O}(h(n))$ calls to ComputeFromChild and ComputeFromBuffer.

We conclude that $\text{Cover}(v, w, i)$ and $\text{Uncover}(v, w, i)$ each take worst case $\mathcal{O}(h(n))$ time.

**Analysis** Choosing any $b(n) \in \mathcal{O}(w/\log \ell_{\max})$ we get height $h(n) \in \mathcal{O}(\frac{\log n}{\log b(n)})$, so Link and Cut take worst case $\mathcal{O}(\frac{b(n)\log n}{\log b(n)})$ time with this CoverLevel structure. The remaining operations, Connected, Cover, Uncover, CoverLevel and MinCoveredEdge all take $\mathcal{O}(\frac{\log n}{\log b(n)})$ worst case time. For the purpose of our main result, choosing $b(n) \in \Theta(\sqrt{\log n})$ is sufficient. Each cluster uses $\mathcal{O}(1)$ space, so the total space used is $\mathcal{O}(n)$.

Note that the pointers that allow us to find the first $k$ least-covered edges can still be maintained in $\mathcal{O}(h)$ time per update, and allows us to find the first $k$ least-covered edges in $\mathcal{O}(h+k)$ time.

## 10 Saving Space

We now apply the space-saving trick from Lemma 10 to the FindSize structures from Section 5 and 7. Let $D$ be the number of words used for each size vector in our FindSize structure. This is $\mathcal{O}(\log n)$ for the purely combinatorial version, and $\mathcal{O}(\log\log n)$ in the version using approximate counting. As shown previously these use $s(n) = \mathcal{O}(D)$ space per cluster and $t(n) = \mathcal{O}(\log n \cdot D)$ worst case time per merge and split.

**Lemma 11.** *The complete information for a cluster of size $q$ in the FindSize structure, including information that would be shared with its children, has total size $s_0(q,n) = \mathcal{O}(q + \ell_{\max} \cdot D)$.*

*Proof.* The complete information for a cluster $C$ with $|C| = q$ consists of

- $c(e)$ for all $e \in C$.
- $\text{cover}_C$, $\text{cover}_C^-$, $\text{cover}_C^+$, $\text{globalcover}_C$, $\text{size}_C$.
- $\text{partsize}_{C,v,i}$ and $\text{diagsize}_{C,v,i}$ for $v \in \partial C$ and $-1 \leq i \leq \ell_{\max}$.

The total size for all of these is $s_0(q,n) = \mathcal{O}(q + \ell_{\max} \cdot D)$ □

Note that when keeping $n$ fixed, this is clearly $\mathcal{O}(q)$. In particular, we can choose $q(n) \in \Theta(\ell_{\max} \cdot D)$ such that $s(n) < s_0(q(n), n) \in \mathcal{O}(q(n))$.

**Lemma 12.** *The complete information for a cluster of size $q$ in the FindSize structure, including information that would be shared with its children, can be computed directly in time $t_0(q,n) = \mathcal{O}(q \log q + \ell_{\max} \cdot D)$.*

*Proof.* Let $C$ be the cluster of size $|C| = q$. For each $v \in \partial C$, we can in $\mathcal{O}(q)$ time find and partition the cluster path into the at most $\ell_{\max}$ parts such that in part $i$, each vertex $m$ on the cluster path have CoverLevel$(v,m) = i$. For each part $i$, run the following algorithm:

```
1: Vector $x \leftarrow 0$
2: Initialize empty max-queue $Q$
3: $j \leftarrow \ell_{\max}$
4: for $w \leftarrow$ each vertex in the fragment that is on $\pi(C)$ do
5:     Mark $w$ as visited
6:     $x_j \leftarrow x_j + 1$
7:     for $e \leftarrow$ each edge incident to $w$ that is not on $\pi(C)$ do
8:         if $c(e) \geq 0$ then
9:             Add $e$ to $Q$ with key $c(e)$
```

```
10: while $Q$ is not empty do
11:     $e \leftarrow$ EXTRACT-MAX$(Q)$
12:     while $c(e) < j$ do
13:         $x_{j-1} = x_j$
14:         $j \leftarrow j - 1$
15:     $w \leftarrow$ the unvisited vertex at the end of $e$
16:     Mark $w$ as visited
17:     $x_j \leftarrow x_j + 1$
18:     for $e \leftarrow$ each edge incident to $w$ that has an unvisited end do
19:         if $c(e) \geq 0$ then
20:             Add $e$ to $Q$ with key $c(e)$
21: partsize$_{C,v,i} \leftarrow x$
22: diagsize$_{C,v,i} \leftarrow M(i) \cdot x$
```

If the $i$th part has size $q_i$ than it can be processed this way in $\mathcal{O}(q_i \log q_i + D)$ time. Summing over all $\mathcal{O}(\ell_{\max})$ parts gives the desired result. □

**Analysis** Applying Lemma 10 with the $s(n)$, $t(n)$, $s_0(q, n)$, $t_0(q, n)$ and $q(n)$ derived in this section immediately gives a FindSize structure with $\mathcal{O}(\log n \cdot D \cdot \log \ell_{\max})$ worst case time per operation and using $\mathcal{O}(n)$ space. A completely analogous argument shows that we can convert the bitpacking-free version of the FindFirstLabel structure from $\mathcal{O}(\log n \cdot \ell_{\max} \cdot \log \ell_{\max})$ time and $\mathcal{O}(m + n \cdot \ell_{\max})$ space to one using linear space. (If bitpacking is allowed the structure already used linear space). In either case is the same time per operation as the original versions, so using the modified version here does not affect the overall running time, but reduces the total space of each bridge-finding structure to $\mathcal{O}(m + n)$.

Note that we can explicitly store lists with all the least-covered edges for these large base clusters, so this does not change the time to report the first $k$ least-covered edges.

## References

[1] Stephen Alstrup, Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. Maintaining information in fully dynamic trees with top trees. *ACM Trans. Algorithms*, 1(2):243–264, October 2005.

[2] Therese C. Biedl, Prosenjit Bose, Erik D. Demaine, and Anna Lubiw. Efficient algorithms for petersen's matching theorem. *Journal of Algorithms*, 38(1):110 – 134, 2001.

[3] Krzysztof Diks and Piotr Stanczyk. *Perfect Matching for Biconnected Cubic Graphs in $O(n \log^2 n)$ Time*, pages 321–333. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[4] David Eppstein, Zvi Galil, and Giuseppe F. Italiano. Improved sparsification. Technical report, 1993.

[5] Greg N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM Journal on Computing*, 14(4):781–798, 1985.

[6] Greg N. Frederickson. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. *SIAM J. Comput.*, 26(2):484–538, 1997.

[7] Harold N. Gabow, Haim Kaplan, and Robert Endre Tarjan. Unique maximum matching algorithms. *J. Algorithms*, 40(2):159–183, 2001. Announced at STOC '99.

[8] Monika R. Henzinger and Valerie King. *Maintaining minimum spanning trees in dynamic graphs*, pages 594–604. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

[9] Monika Rauch Henzinger and Valerie King. Fully dynamic 2-edge connectivity algorithm in polylogarithmic time per operation, 1997.

[10] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. 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 '98, pages 79–89, New York, NY, USA, 1998. ACM.

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

[12] Shang-En Huang, Dawei Huang, Tsvi Kopelowitz, and Seth Pettie. 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*, SODA '17, pages 510–520, Philadelphia, PA, USA, 2017. Society for Industrial and Applied Mathematics.

[13] Bruce M. Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *Proceedings of the Twenty-fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '13, pages 1131–1142, Philadelphia, PA, USA, 2013. Society for Industrial and Applied Mathematics.

[14] Donald E. Knuth. Two notes on notation. *The American Mathematical Monthly*, 99(5):403–422, 1992.

[15] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[16] Anton Kotzig. *On the theory of finite graphs with a linear factor II.* 1959.

[17] Karl Menger. Zur allgemeinen Kurventheorie. *Fundamenta Mathematicae*, 10, 1927.

[18] Mihai Patrascu and Erik D Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM Journal on Computing*, 35(4):932–963, 2006.

[19] Julius Petersen. Die Theorie der regulären graphs. *Acta Math.*, 15:193–220, 1891.

[20] Mikkel Thorup. Near-optimal fully-dynamic graph connectivity. In *Proceedings of the Thirty-second Annual ACM Symposium on Theory of Computing*, STOC '00, pages 343–350, New York, NY, USA, 2000. ACM.

[21] Christian Wulff-Nilsen. Faster deterministic fully-dynamic graph connectivity. In *Encyclopedia of Algorithms*, pages 738–741. 2016.

## A Details of the high level algorithm

**Lemma 5** (Essentially the high level algorithm from [11])**.** *There exists a deterministic reduction for dynamic graphs with $n$ nodes, that, when starting with an empty graph, supports any sequence of $m$ Insert or Delete operations using:*

- *$\mathcal{O}(m)$ calls to Link, Cut, Uncover, and CoverLevel.*
- *$\mathcal{O}(m \log n)$ calls to Connected, Cover, AddLabel, RemoveLabel, FindFirstLabel, and FindSize.*

*And that can answer FindBridge queries using a constant number of calls to Connected, CoverLevel, and MinCoveredEdge.*

*Proof.* The only part of the high level algorithm from [11] that does not directly and trivially translate into a call of the required dynamic tree operations (see pseudocode below) is in the Swap method where given a tree edge $e = (v, w)$ we need to find a nontree edge $e'$ covering $e$ with $\ell(e') = i = \text{CoverLevel}(e)$. We can find this $e'$ by using FindFirstLabel and increasing the level of each non-tree edge we examine that does not cover $e$. For at least one side of $(v, w)$, all non-tree edges at level $i$ incident to that side will either cover $e$ or can safely have their level increased without violating the size invariant. So we can simply search the side where the level $i$ component is smallest until we find the required edge (which must exist since $e$ was covered on level $i$). The amortized cost of all operations remain unchanged with this implementation. Counting the number of operations (see Table 2) gives the desired bound. □

| # | Operation | #Calls during | | | | |
|---|---|---|---|---|---|---|
| | | Insert+Delete | FindBridge($v$) | FindBridge($v,w$) | Size(v) | 2-Size(v) |
| 1 | Link($v, w, e$) | 1 | 0 | 0 | 0 | 0 |
| 2 | Cut($e$) | 1 | 0 | 0 | 0 | 0 |
| 3 | Connected($v, w$) | $\log n$ | 0 | 1 | 0 | 0 |
| 4 | Cover($v, w, i$) | $\log n$ | 0 | 0 | 0 | 0 |
| 5 | Uncover($v, w, i$) | 1 | 0 | 0 | 0 | 0 |
| 6 | CoverLevel($v$) | 0 | 1 | 0 | 0 | 0 |
| 7 | CoverLevel($v, w$) | 1 | 0 | 1 | 0 | 0 |
| 8 | MinCoveredEdge($v$) | 0 | 1 | 0 | 0 | 0 |
| 9 | MinCoveredEdge($v, w$) | 0 | 0 | 1 | 0 | 0 |
| 10 | AddLabel($v, l, i$) | $\log n$ | 0 | 0 | 0 | 0 |
| 11 | RemoveLabel($l$) | $\log n$ | 0 | 0 | 0 | 0 |
| 12 | FindFirstLabel($v, w, i$) | $\log n$ | 0 | 0 | 0 | 0 |
| 13 | FindSize($v, w, i$) | $\log n$ | 0 | 0 | 0 | 1 |
| | FindSize($v, v, -1$) | 0 | 0 | 0 | 1 | 0 |

Table 2: Overview of how many times each tree operation is called for each graph operation, ignoring constant factors. The "Insert+Delete" column is amortized over any sequence starting with an empty set of edges. The remaining columns are worst case.

```
1: function 2-EDGE-CONNECTED($v$, $w$)
2:     return T.CONNECTED($v$, $w$) $\wedge$ T.COVERLEVEL($v$, $w$)$\geq 0$
3: function FINDBRIDGE($v$)
4:     if T.COVERLEVEL($v$)$= -1$ then
```

```
        return T.MinCoveredEdge($v$)
    else
        return nil
function FindBridge($v$, $w$)
    if T.CoverLevel($v$, $w$)$= -1$ then
        return T.MinCoveredEdge($v$, $w$)
    else
        return nil
function Size($v$)
    return T.FindSize($v$,$v$,$-1$)
function 2-Size($v$)
    return T.FindSize($v$,$v$,0)
function Insert($v$, $w$, $e$)
    if $\neg$T.Connected($v$, $w$) then
        T.Link($v$, $w$, $e$)
        $\ell(e) \leftarrow \ell_{\max}$
    else
        T.AddLabel($v$, $e$.label1, 0)
        T.AddLabel($w$, $e$.label2, 0)
        $\ell(e) \leftarrow 0$
        T.Cover($v$, $w$, 0)
function Delete($e$)
    $(v, w) \leftarrow e$
    $\alpha \leftarrow \ell(e)$
    if $\alpha = \ell_{\max}$ then
        $\alpha \leftarrow$ T.CoverLevel$(v, w)$
        if $\alpha = -1$ then
            T.Cut($e$)
            return
        Swap($e$)
    T.RemoveLabel($e$.label1)
    T.RemoveLabel($e$.label2)
    T.Uncover$(v, w, \alpha)$
    for $i \leftarrow \alpha, \ldots, 0$ do
        Recover($w$,$v$,$i$)
function Swap($e$)
    $(v, w) \leftarrow e$
    $\alpha \leftarrow$ T.CoverLevel$(v, w)$
    T.Cut($e$)
    $e' \leftarrow$FindReplacement($v$,$w$,$\alpha$)
    $(x, y) \leftarrow e'$
    T.RemoveLabel($e'$.label1)
    T.RemoveLabel($e'$.label2)
    T.Link$(x, y, e')$
```

```
49:     $\ell(e') \leftarrow \ell_{\max}$
50:     T.AddLabel($v$,$e$.label1, $\alpha$)
51:     T.AddLabel($w$,$e$.label2, $\alpha$)
52:     $\ell(e) \leftarrow \alpha$
53:     T.Cover($v$,$w$,$\alpha$)
54: function FindReplacement($v$,$w$,$i$)
55:     $s_v \leftarrow$ T.FindSize($v, v, i$)
56:     $s_w \leftarrow$ T.FindSize($w, w, i$)
57:     if $s_v \leq s_w$ then
58:         return RecoverPhase($v, v, i, s_v$)
59:     else
60:         return RecoverPhase($w, w, i, s_w$)
61: function Recover($v$,$w$,$i$)
62:     $s \leftarrow \lfloor$T.FindSize($v, w, i$)$/2\rfloor$
63:     RecoverPhase($v$,$w$,$i$,$s$)
64:     RecoverPhase($w$,$v$,$i$,$s$)
65: function RecoverPhase($v, w, i, s$)
66:     $l \leftarrow$ T.FindFirstLabel($v, w, i$)
67:     while $l \neq$ nil do
68:         $e \leftarrow l$.edge
69:         $(q, r) \leftarrow e$
70:         if $\neg$T.Connected($q$, $r$) then
71:             return $e$
72:         if T.FindSize($q, r, i+1$) $\leq s$ then
73:             T.RemoveLabel($e$.label1)
74:             T.RemoveLabel($e$.label2)
75:             T.AddLabel($q$, $e$.label1, $i+1$)
76:             T.AddLabel($r$, $e$.label2, $i+1$)
77:             $\ell(e) = i+1$
78:             T.Cover($q$,$r$,$i+1$)
79:         else
80:             T.Cover($q$,$r$,$i$)
81:             return nil
82:         $l \leftarrow$ T.FindFirstLabel($v, w, i$)
83:     return nil
```

## B Pseudocode for the CoverLevel structure

```
1: function CL.Cover($v$,$w$,$i$)
2:     $C \leftarrow$ TopTree.Expose($v$, $w$)
3:     $\text{cover}_C \leftarrow \max\{\text{cover}_C, i\}$
4:     $\text{cover}^+_C \leftarrow \max\{\text{cover}^+_C, i\}$
5: function CL.Uncover($v$,$w$,$i$)
6:     $C \leftarrow$ TopTree.Expose($v$, $w$)
```

```
    cover$_C$ ← −1
    cover$_C^+$ ← −1
    cover$_C^-$ ← max {cover$_C^-$, $i$}
function CL.CoverLevel($v$)
    $C$ ← TopTree.Expose($v$)
    return globalcover$_C$
function CL.CoverLevel($v$, $w$)
    $C$ ← TopTree.Expose($v$, $w$)
    return cover$_C$
function CL.MinCoveredEdge($v$)
    $C$ ← TopTree.Expose($v$)
    return minglobaledge$_C$
function CL.MinCoveredEdge($v$, $w$)
    $C$ ← TopTree.Expose($v$, $w$)
    return minpathedge$_C$
function CL.Split($C$)
    for each path child $D$ of $C$ do
        if max {cover$_D$, cover$_D^-$} ≤ cover$_C^-$ then
            cover$_D^-$ ← cover$_C^-$
        if cover$_D$ ≤ max {cover$_D^-$, cover$_D^+$} then
            cover$_D$ ← cover$_C^+$
            cover$_D^+$ ← cover$_C^+$
function CL.Merge($C$; $A_1, \dots, A_k$)
    cover$_C$ ← $\ell_{\max}$
    minpathedge$_C$ ← nil
    globalcover$_C$ ← $\ell_{\max}$
    minglobaledge$_C$ ← nil
    for $i \leftarrow 1, \dots, k$ do
        if $\partial A_i \subseteq \pi(C)$ then
            if cover$_{A_i}$ < cover$_C$ then
                cover$_C$ ← cover$_{A_i}$
                minpathedge$_C$ ← minpathedge$_{A_i}$
        else
            if cover$_{A_i}$ < globalcover$_C$ then
                globalcover$_C$ ← cover$_{A_i}$
                minglobaledge$_C$ ← minpathedge$_{A_i}$
        if globalcover$_{A_i}$ < globalcover$_C$ then
            globalcover$_C$ ← globalcover$_{A_i}$
            minglobaledge$_C$ ← minglobaledge$_{A_i}$
    cover$_C^-$ ← −1
    cover$_C^+$ ← −1
function CL.Create($C$; edge $e$)
    cover$_C$ ← −1
```

```
50:     globalcover$_C \leftarrow -1$
51:     if $C$ is a point cluster then
52:         minpathedge$_C \leftarrow$ nil
53:         minglobaledge$_C \leftarrow e$
54:     else
55:         minpathedge$_C \leftarrow e$
56:         minglobaledge$_C \leftarrow$ nil
57:     cover$^-_C \leftarrow -1$
58:     cover$^+_C \leftarrow -1$
```

## C Pseudocode for the FindSize structure

In the following, we use the notation

$$[\text{key} : \text{partsize}, \text{diagsize}]$$

to denote the root of a new tree consisting of a single node with the given values. And for a given tree root and given $x, y$

$$(\text{tree}_{\{x \le i \le y\}})$$

is the root of the subtree consisting of all nodes whose keys are in the given range. Similarly, for any given $i$, let

$$(\text{tree}_i)$$

denote the node in the tree having the given key.

```
1: function FS.FindSize($v$, $w$, $i$)
2:     $C \leftarrow$ TopTree.Expose($v$, $w$)
3:     return size$_{C,i}$
4: function FS.Merge($C$; $A$, $B$)
5:     $\{c\} \leftarrow \partial A \cap \partial B$
6:     if $c \in \pi(C)$ then                                   ▷ Merge along path
7:         if $|\partial C| <= 1$ then
8:             $a \leftarrow c$, $b \leftarrow c$
9:         else
10:            $\{a, b\} \leftarrow \partial C$ with $a \in \partial A$ and $b \in \partial B$.
11:        size$_C \leftarrow$ size$_A +$ size$_B$
12:        for $(x, X) \leftarrow (a, A), (b, B)$ do
13:            if $x = c$ then
14:                tree$'_{X,x} \leftarrow$ tree$_{X,x}$, undo$'_{X,x} \leftarrow$ nil
15:            else
16:                for $v \leftarrow x, c$ do
17:                    $\ell \leftarrow \max\{\text{cover}^-_X, \text{cover}^+_X\}$
18:                    $s \leftarrow (\text{tree}_{X,v}).\text{partsizesum}$
19:                    $d \leftarrow M(\text{cover}^+_X) * s$
```

```
                    $\mathrm{tree}'_{X,v} \leftarrow \mathrm{tree}_{X,v,\{i>\ell\}}$, $\mathrm{undo}'_{X,v} \leftarrow \mathrm{tree}_{X,v,\{i\leq\ell\}}$
                    $\mathrm{tree}'_{X,v} \leftarrow \mathrm{tree}'_{X,v} + [\mathrm{cover}^{+}_{X} : s, d]$
        for $(x, X, y, Y) \leftarrow (a, A, b, B), (b, B, a, A)$ do
            $s \leftarrow (\mathrm{tree}'_{Y,c,\{\mathrm{cover}_X \leq i \leq \ell_{\max}\}}).\,\mathrm{partsizesum}$
            $p \leftarrow (\mathrm{tree}'_{X,x,\mathrm{cover}_X}).\,\mathrm{partsize} + s$
            $d \leftarrow (\mathrm{tree}'_{X,x,\mathrm{cover}_X}).\,\mathrm{diagsize} + M(\mathrm{cover}_X) * s$
            if $x = c$ then
                $\mathrm{tree}''_{X,x} \leftarrow [\ell_{\max} : \mathrm{size}_X, \mathrm{size}_X]$, $\mathrm{undo}''_{X,x} \leftarrow$ nil
            else
                $\mathrm{tree}''_{X,x} \leftarrow \mathrm{tree}'_{X,x,\{i>\mathrm{cover}_X\}}$, $\mathrm{undo}''_{X,x} \leftarrow \mathrm{tree}'_{X,x,\{i\leq\mathrm{cover}_X\}}$
            if $y = c$ then
                $\mathrm{tree}'''_{Y,c} \leftarrow$ nil, $\mathrm{undo}'''_{Y,c} \leftarrow [\ell_{\max} : \mathrm{size}_Y, \mathrm{size}_Y]$
            else
                $\mathrm{tree}'''_{Y,c} \leftarrow \mathrm{tree}'_{Y,c,\{i<\mathrm{cover}_X\}}$, $\mathrm{undo}'''_{Y,c} \leftarrow \mathrm{tree}'_{Y,c,\{i\geq\mathrm{cover}_X\}}$
            $\mathrm{tree}_{C,x} \leftarrow \mathrm{tree}''_{X,x} + [\mathrm{cover}_X : p, d] + \mathrm{tree}'''_{Y,c}$
    else                                                              ▷ Merge off path
        $\{a\} \leftarrow \partial C \setminus \{c\}$
        if $a \notin \partial A$ then
            Swap $A$ and $B$
        $\ell \leftarrow \max\left\{\mathrm{cover}^{-}_{A}, \mathrm{cover}^{+}_{A}\right\}$
        $d \leftarrow (\mathrm{tree}_{A,a,\{\ell<i\leq\ell_{\max}\}}).\,\mathrm{diagsizesum}$
        $p \leftarrow (\mathrm{tree}_{A,a,\{-1\leq i\leq\ell\}}).\,\mathrm{partsizesum}$
        $\mathrm{size}_C \leftarrow d + M(\mathrm{cover}^{+}_{A}) * p + M(\mathrm{cover}_A) * \mathrm{size}_B$
        $\mathrm{tree}_{C,a} \leftarrow [\ell_{\max} : \mathrm{size}_C, \mathrm{size}_C]$
function FS.SPLIT($C$)
    $A, B \leftarrow$ the children of $C$
    $\{c\} \leftarrow \partial A \cap \partial B$
    if $c \in \pi(C)$ then                                            ▷ Split along path
        if $|\partial C| <= 1$ then
            $a \leftarrow c$, $b \leftarrow c$
        else
            $\{a, b\} \leftarrow \partial C$ with $a \in \partial A$ and $b \in \partial B$.
        for $(x, X, y, Y) \leftarrow (a, A, b, B), (b, B, a, A)$ do
            $\mathrm{tree}''_{X,x} \leftarrow \mathrm{tree}_{C,x,\{i>\mathrm{cover}_X\}}$, $\mathrm{tree}'''_{Y,c} \leftarrow \mathrm{tree}_{C,x,\{i<\mathrm{cover}_X\}}$
            if $y \neq c$ then
                $\mathrm{tree}'_{Y,c} \leftarrow \mathrm{tree}'''_{Y,c} + \mathrm{undo}'''_{Y,c}$
            if $x \neq c$ then
                $\mathrm{tree}'_{X,x} \leftarrow \mathrm{tree}''_{X,x} + \mathrm{undo}''_{X,x}$
        for $(x, X) \leftarrow (a, A), (b, B)$ do
            if $x \neq c$ then
                for $v \leftarrow x, c$ do
                    $\mathrm{tree}_{X,v} \leftarrow \mathrm{tree}'_{X,v,\left\{i>\mathrm{cover}^{+}_{X}\right\}} + \mathrm{undo}'_{X,v}$
```

```
62: function FS.CREATE($C$; edge $e$)
63:     $\mathrm{size}_C \leftarrow 0$
64:     for $v \in \partial C$ do
65:         $\mathrm{tree}_{C,v} \leftarrow [\ell_{\max} : 0, 0]$
66: function FS.CREATE($C$; vertex label $l$)
67:     $\mathrm{size}_C \leftarrow (1)_{\{0 \le i < \ell_{\max}\}}$
68:     for $v \in \partial C$ do
69:         $\mathrm{tree}_{C,v} = [\ell_{\max} : \mathrm{size}_C, \mathrm{size}_C]$
```