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

Link to article, DOI:
10.1137/1.9781611975031.3

Publication date:
2018

Document Version
Publisher's PDF, also known as Version of record

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

Jacob Holm† Eva Rotenberg†‡ Mikkel Thorup†

November 2, 2017

Abstract

We present a deterministic fully-dynamic data structure for maintaining information about the bridges in a graph. We support updates in $O((\log n)^2)$ 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 $O((\log n)^3)$ 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 $O((\log n)^3)$ amortized time algorithm. Combining it with Thorup's bittrick, we get down to the claimed $O((\log n)^2)$ amortized time.

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

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 $\mathcal{O}(m + n\log n\log\log n)$.

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

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 [20], this is equivalent to saying that a pair of connected vertices are 2-edge connected if there exist two edge-disjoint paths between them. By edge-disjoint it 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 [6] and Eppstein et al. [5]. Later, Frederickson [7] and Eppstein et al. [5] gave a data structure with $\mathcal{O}(\sqrt{n})$ update time for 2-edge connectivity. Henzinger and King achieved poly-logarithmic expected amortized time [12], 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 [11], $\mathcal{O}((\log n)^5)$ 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. announced in [13], see [14] for a journal version; 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 [24]. Sacrificing determinism, an $\mathcal{O}(\log n(\log\log n)^3)$ structure for connectivity was presented by Thorup [23], and later improved to $\mathcal{O}(\log n(\log\log n)^2)$ by Huang et al. [15]. In the same paper, Thorup obtains an update time of $\mathcal{O}((\log n)^3\log\log n)$ for deterministic 2-edge connectivity. Interestingly, Kapron et al. [16] 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. This was later improved by Gibbs et al. [10] to $\mathcal{O}((\log n)^4)$ worst case update time and sublinear $O(n\log^2 n)$ space. We know of no similar worst-case result for bridge finding. The same paper [10] also gives the first sublinear-space

*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.

†jaho@di.ku.dk, eva@rotenberg.dk, mthorup@di.ku.dk, Department of Computer Science, University of Copenhagen, Denmark

‡Department of Applied Mathematics and Computer Science, Technical University of Denmark, Denmark

$O(n \log^2 n)$ space data structure for (amortized) 2-edge connectivity, by using the sublinear-space connectivity data structure to maintain a sparse subgraph preserving 2-edge connectivity and then using the existing 2-edge connectivity data structure from Holm et al. [14] as a black box on that subgraph.

The best lower bound known is by Pătraşcu et al. [21], 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 1.1. *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.*
- *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 2-edge connected exactly when there is no bridge separating them, we have the following corollary:

COROLLARY 1.1. *There exists a data structure for dynamic multigraphs in the word RAM model with $\Omega(\log n)$ word size, that can answer 2-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. [21]

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

THEOREM 1.2. *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 [14]. 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 Although our data structure is deterministic and uses linear space, it entails an improvement of the current best sublinear-space data structure. Namely, the Monte-Carlo randomized sublinear-space 2-edge connectivity data structure by Gibbs et al. [10] uses the data structure from [14] as a black box: For each update the data structure uses worst case $O(\log^5 n)$ time by itself, and makes $O(\log^2 n)$ updates in the sparse graph seen by the black box. Thus, with the $O(\log^4 n)$ amortized update time from [14], this gives a sublinear-space data structure with amortized $O(\log^6 n)$ update time. Using the data structure from [23] or our new purely combinatorial data structure, this drops to $\mathcal{O}((\log n)^5 \log\log n)$ amortized time. With our new $\tilde{O}(\log^2 n)$ update time data structure, this improves to $\mathcal{O}(\log^5 n)$ amortized time (and the bottleneck is now in the reduction).

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 time algorithm if implemented together with a decremental 2-edge connectivity data structure with poly-logarithmic update time:

THEOREM 1.3. (A. KOTZIG '59 [19]) *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 [9] is straightforward: 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 [22] which states that any cubic, 2-edge connected graph contains a perfect matching. An algorithm by Biedl et al. [3] finds a

perfect matching in such graphs in $\mathcal{O}(n \log^4 n)$ time, by using the Holm et al 2-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 2-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 [4] improved Biedl et al.'s algorithm for perfect matchings in 2-edge connected cubic graphs, by having it rely only on dynamic connectivity, not 2-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 [2] 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. [14] 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. [14], 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 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 [23]. 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 [14] fundamentally solves 2-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 [2]. 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 1.2. 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 1.1. 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 [14], 2-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. (Called when an edge was inserted at level $i = 0$ or had its level incresed to $i > 0$.)
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. (Called when the knowledge we had about whether the edges on the tree path from v to w were covered at level $\leq i$ is no longer valid because some edge was deleted. This may temporarily set some cover levels too low, but the algorithm fixes that using subsequent calls to Cover.)
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. (Find a bridge, anywhere in the tree.)
9. MinCoveredEdge(v, w). Returns a tree-edge on the path from v to w whose cover level is CoverLevel(v, w). (Find a bridge on the given path.)
10. AddLabel(v, l, i). Associate the *user label* l to the vertex v at level i. (Insert a non-tree edge.)
11. RemoveLabel(l). Remove the user label l from its vertex vertex(l). (Delete a non-tree edge.)
12. FindFirstLabel(v, w, i). Find a user label at level i such that the associated vertex u has[1] CoverLevel$(u, \text{meet}(u, v, w)) \geq i$; among such user labels, return the one that minimizes the distance from v to meet(u, v, w). (Find the first candidate witness that part of the path $v \cdots w$ is covered on level i, or a non-tree edge to swap with when deleting a covered tree edge.)
13. FindSize(v, w, i). Find the number of vertices u such that CoverLevel$(u, \text{meet}(u, v, w)) \geq i$. (Determine the size of the 2-edge connected component at level i that would result from increasing the level of (v, w) to i, or for $v = w$ just find the size of the 2-edge connected component at level i that contains $v = w$. Thus the size of the 2-edge component of v in the whole graph is FindSize$(v, v, 0)$.). Note that FindSize$(v, v, -1)$ is just the number of vertices in the tree containing v (which is also the size of the connected component of v).

LEMMA 2.1. (ESSENTIALLY FROM [14]) *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, and size queries using a single call to FindSize.

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

The algorithm in [14] 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 [23] 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 2.2. (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 $u \in U$,*

[1] meet(u, v, w) is defined as the unique vertex that is on all simple paths beween any two of u, v, and w.

#	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)					
3	Connected(v, w)					$\frac{\log n}{\log f(n)}$
4	Cover(v, w, i)					
5	Uncover(v, w, i)					
6	CoverLevel(v)					
7	CoverLevel(v, w)					
8	MinCoveredEdge(v)					
9	MinCoveredEdge(v, w)					
10	AddLabel(v, l, i)	-	-	$\log n \log\log n$	-	-
11	RemoveLabel(l)					
12	FindFirstLabel(v, w, i)					
13	FindSize(v, w, i)	-	$(\log n)^2 \log\log n$	-	$\log n(\log\log n)^2$	-
	FindSize$(v, v, -1)$	$\log n$	$\log n$	$\log n$	$\log n$	$\frac{\log n}{\log f(n)}$
		Space cost, using structure in section				
natively		n	$n \log n$	$m + n$	$n \log\log n$	n
when modified as in Section 10			n		n	

Table 1: 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.

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 1.2. 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 2.2. Then the reduction from Lemma 2.1 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 1.1. 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 2.2. Then the reduction from Lemma 2.1 gives the required bounds. □

3 Top trees

A *top tree* is a data structure for maintaining information about each tree of a dynamic forest. Let T be a tree, and let ∂T be an arbitrary set of 1 or 2 vertices of T, which we will call the *external boundary vertices* of T. For any subgraph S of T, define the *boundary vertices* of S (denoted $\partial_{(T,\partial T)} S$ or just ∂S) as the set of vertices in S that are either in ∂T or are incident to an edge not in S. A cluster C is a connected subgraph of T with 1 or 2 boundary vertices[2]. A top tree $\mathcal{T}$ is a rooted tree representing a recursive partition of T into clusters. The root of $\mathcal{T}$ corresponds to all of T, and each non-leaf node is an edge-disjoint union of the clusters of its children. The leaves of $\mathcal{T}$ are called *base clusters* and (usually[3]) correspond to the edges of T.

For every cluster C the *cluster path* of C, denoted $\pi(C)$, is the tree path in T connecting ∂C. If $|\partial C| = 2$ then $\pi(C)$ contains at least one edge, and we call C a *path cluster*. Otherwise $|\pi(C)| = 1$ and we call C a *point cluster*. If $|\partial C| = 1$ then $\pi(C)$ is the trivial path consisting of the single boundary vertex.

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.

A path cluster D is called a *path child* of its parent C if $\pi(D) \subseteq \pi(C)$. Note that for binary top trees, a path cluster D is a path child if and only if its parent C is also a path cluster. But for non-binary top trees, even if C and D are both path clusters, ∂D may intersect ∂C only in a point, or not at all.

The top forest supports dynamic changes to the forest: insertion (link) or deletion (cut) of edges. Fur-

[2]Note that this deviates from the existing literature, which introduces a special class of cluster with 0 boundary vertices, which can only be present in the root [2]

[3]We will look at generalized top trees where this is not the case in Section 8

thermore, 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 external boundary vertices. 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 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 they are edge-disjoint and their union is a cluster (i.e. is connected and has a boundary of size at most 2).

THEOREM 3.1. (THEOREM 1 IN [2]) *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 essentially the same as in [13,14] (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:

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

Then

$$\begin{aligned}
\text{CoverLevel}(v) &= \text{globalcover}_C \\
\text{MinCoveredEdge}(v) &= \text{minglobaledge}_C
\end{aligned}$$

where C is the point cluster returned by Expose(v), and

$$\begin{aligned}
\text{CoverLevel}(v, w) &= \text{cover}_C \\
\text{MinCoveredEdge}(v, w) &= \text{minpathedge}_C
\end{aligned}$$

where C 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:

$$\text{cover}_C = \max\{\text{cover}_C, i\} \quad \text{cover}^+_C = \max\{\text{cover}^+_C, i\}$$

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 $\text{cover}_C \leq i$:

$$\text{cover}_C = -1 \quad \text{cover}^+_C = -1 \quad \text{cover}^-_C = \max\{\text{cover}^-_C, i\}$$

Together, cover^-_C and cover^+_C represent the fact that for each path descendant D of C, if $\text{cover}_D \leq \max\{\text{cover}^-_C, \text{cover}^+_C\}$[4], 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 $\max\{\text{cover}_D, \text{cover}^-_D\} \leq \text{cover}^-_C$ we need to set

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

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

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

Note that only cover_D is affected. None of globalcover_D, minpathedge_D, or minglobaledge_D depend directly on the lazy information.

Now suppose we have k clusters[5] $A_1, \ldots, A_k$ that we want to merge into a single new cluster C. For $1 \leq i \leq k$ define

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

[4] In [13, 14] this condition is erroneously stated as $\text{cover}_D \leq \text{cover}^-_C$.

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

Note that for a point-cluster A_i, 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:

$$\text{cover}_C = \begin{cases} \min\limits_{\substack{1\leq i<k \\ \partial A_i \subseteq \pi(C)}} \text{cover}_{A_i} & \text{if } |\partial C| = 2 \\ \ell_{\max} & \text{otherwise} \end{cases}$$

$$\text{minpathedge}_C = \begin{cases} \text{minpathedge}_{A_j} & \text{if } |\partial C| = 2 \\ \textbf{nil} & \text{otherwise} \end{cases}$$

$$\text{where } j = \operatorname*{arg\,min}_{\substack{1\leq i<k \\ \partial A_i \subseteq \pi(C)}} \text{cover}_{A_i}$$

$$\text{globalcover}_C = \min_{1\leq i<k} \text{globalcover}'_{C,A_i}$$

$$\text{minglobaledge}_C = \text{minglobaledge}'_{C,A_j}$$

$$\text{where } j = \operatorname*{arg\,min}_{1\leq i<k} \text{globalcover}'_{C,A_i}$$

$$\text{cover}^-_C = -1$$

$$\text{cover}^+_C = -1$$

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 $\mathcal{O}(n)$.

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.

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

$$\text{pointset}_{C,v,i} := \left\{ u \in C \;\middle|\; \begin{array}{c} \pi(C) \cap u \cdots v = \{v\} \\ \wedge\ \text{CoverLevel}(u,v) \geq i \end{array} \right\}$$

Intuitively, $\text{pointset}_{C,v,i}$ is the set of vertices in C whose path to v is covered at level $\geq i$ independently of the cover levels on $\pi(C)$. Information (such as the size or the existence of certain marked vertices) about this set stays constant for as long as C exists, no matter what happens with the lazy information in the ancestors to C. In this section we only care about the size

$$\text{pointsize}_{C,v,i} := \left|\text{pointset}_{C,v,i}\right|$$

For convenience, we will combine all the $\mathcal{O}(\log n)$ levels together into a single vector[6]

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

Then we can define the vector

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

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 size_C vector is what we want to maintain.

The main difficulty turns out be computing the size_C vector for the heterogeneous point clusters. To help with that we will for each cluster C and boundary vertex $v \in \partial C$ break $\pi(C)$ into $\ell_{\max}+2$ *parts*. For each $-1 \leq i \leq \ell_{\max}$ define

$$\text{partpath}_{C,v,i} := \{u \in \pi(C) \mid \text{CoverLevel}(u,v) = i\}$$

Then $\text{partpath}_{C,v,i}$ (if nonempty) is a contiguous subset of the vertices on $\pi(C)$. Furthermore, $\text{partpath}_{C,v,\ell_{\max}} = \{v\}$, $\partial C \setminus \{v\} \subseteq \text{partpath}_{C,v,-1}$, and for all $0 \leq i < \ell_{\max}$ the set $\text{partpath}_{C,v,i}$ lies between the closest edge e to v with $c(e) \leq i$ and the closest edge e' to v with $c(e') < i$. In addition to the size_C vector, we will maintain the following two size vectors for each part:

$$\text{partsize}_{C,v,i} := \sum_{u\in\text{partpath}_{C,v,i}} \text{pointsize}_{C,u}$$

$$\text{diagsize}_{C,v,i} := M(i) \cdot \text{partsize}_{C,v,i}$$

Where $M(i)$ is a diagonal matrix whose entries are defined by[7]

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

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

[7] Here, $[P] = \begin{cases} 1 & \text{if } P \text{ is true} \\ 0 & \text{otherwise} \end{cases}$ is the *Iverson Bracket* (see [17]).

The $M(i)$ matrix is purely a notational convenience whose purpose is to "zero out" some elements in a vector. In particular, for $0 \le j < \ell_{\max}$

$$\begin{aligned}\text{diagsize}_{C,v,i,j} &= (M(i) \cdot \text{partsize}_{C,v,i})_j \\ &= \begin{cases} \text{partsize}_{C,v,i,j} & \text{if } j \le i \\ 0 & \text{otherwise} \end{cases}\end{aligned}$$

Note that these vectors do not take cover_C^- and cover_C^+ (as defined in Section 4) into account. The corresponding "clean" vectors are not explicitly stored, but computed when needed as follows

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

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 [14].

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 both types of homogeneous merges (two point or two path clusters), as well as the heterogeneous merge (one point and one path cluster) where the result is a path cluster. The only type of merge not covered is the heterogeneous merge resulting in a point cluster, which is handled in the next section. 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

$$\text{size}_C = \text{size}_A + \text{size}_B$$

$$\text{partsize}_{C,a,i} = \begin{cases} \text{partsize}'_{A,a,i} & \text{if } i > \text{cover}_A \\ \text{partsize}'_{A,a,i} & \text{if } i = \text{cover}_A \\ \quad + \sum_{j=i}^{\ell_{\max}} \text{partsize}'_{B,c,j} & \\ \text{partsize}'_{B,c,i} & \text{if } i < \text{cover}_A \end{cases}$$

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

The formulas for $\text{partsize}_{C,b,i}$ and $\text{diagsize}_{C,b,i}$ are analoguous. The important thing to note is that if we have already computed and stored the $\text{partsize}'_{A,a,i}$, $\text{partsize}'_{B,c,i}$, $\text{diagsize}'_{A,a,i}$, and $\text{diagsize}'_{B,c,i}$ vectors for all i, then the only new value we need to compute is for $i = \text{cover}_A$. The rest can be inherited.

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

$$\text{size}_C = \left(\sum_{i=-1}^{\ell_{\max}} \text{diagsize}'_{A,a,i}\right) + M(\text{cover}_A) \cdot \text{size}_B$$

$$\text{partsize}_{C,a,i} = \begin{cases} \text{size}_C & \text{if } i = \ell_{\max} \\ \vec{0} & \text{otherwise} \end{cases}$$

$$\text{diagsize}_{C,a,i} = \text{partsize}_{C,a,i}$$

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. [18, 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 jth 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 [13, 14, 23]. Notice that

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

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 [2].

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[8]

$$\begin{aligned}
&\operatorname{pointincident}_{C,v} \\
&\quad := \left(\left[\exists v \in \operatorname{pointset}_{C,v,i} : \begin{matrix} v \text{ has labels} \\ \text{at level } i \end{matrix} \right] \right)_{\{0 \le i < \ell_{\max}\}} \\
&\operatorname{incident}_C \\
&\quad := \bigvee_{u \in \pi(C)} \operatorname{pointincident}_{C,u}
\end{aligned}$$

Maintaining the $\operatorname{incident}_C$ bit vectors, and the corresponding $\operatorname{partincident}_{C,v}$ and $\operatorname{diagincident}_{C,v}$ 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 $v = \operatorname{vertex}(l)$ and $\ell(l) = i$ can be done by first calling $\operatorname{detach}(C)$, then updating the linked lists containing the user labels and setting

$$\begin{aligned}
\operatorname{incident}_C &= \left(\begin{bmatrix} v \text{ has labels} \\ \text{at level } j \end{bmatrix} \right)_{\{0 \le j < \ell_{\max}\}} \\
\operatorname{partincident}_{C,v,i} &= \begin{cases} \operatorname{incident}_C & \text{if } i = \ell_{\max} \\ \vec{0} & \text{otherwise} \end{cases} \\
\operatorname{diagincident}_{C,v} &= \operatorname{partincident}_C
\end{aligned}$$

and then reattaching C. Finally FindFirstLabel(v,w,i) can be implemented in the way already described, by examining $\operatorname{pointincident}_{C,v,i}$ 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

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

In general, let $A_1, \ldots, A_k$ 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

$$\begin{aligned}
&\operatorname{FindFirstLabel}((A_1, \ldots, A_k); v, w, i) \\
&\quad = \begin{cases} \operatorname{userlabels}_{v_x,i} & \text{if } A_x \text{ is a vertex label} \\ \operatorname{FindFirstLabel}(\operatorname{Split}(A_x); v_x, w_x, i) & \\ \qquad \text{otherwise} & \end{cases}
\end{aligned}$$

where for $1 \le j \le k$

$$\begin{aligned}
v_j &= \arg\min_{u \in \partial A_j} \operatorname{dist}(v, u) \\
w_j &= \arg\max_{u \in \partial A_j} \operatorname{dist}(v, u)
\end{aligned}$$

and

$$\begin{aligned}
I &= \left\{ 1 \le j \le k \;\middle|\; \begin{matrix} \operatorname{CoverLevel}(v, v_j) \ge i \\ \wedge\ \operatorname{pointincident}_{A_j, v_j, i} = 1 \end{matrix} \right\} \\
x &= \arg\min_{j \in I} \left(3 \cdot \operatorname{dist}(v, \operatorname{meet}(v_j, v, w)) \right. \\
&\qquad \left. + |\partial A_j \cap v \cdots w| \right)
\end{aligned}$$

and finally

$$\begin{aligned}
&\operatorname{FindFirstLabel}(v, w, i) \\
&\quad = \operatorname{FindFirstLabel}(\operatorname{Expose}(v, w); v, w, i)
\end{aligned}$$

[8] Again, using the Iverson bracket.

What happens here is that, for each j, the vertices v_j and w_j are the boundary vertices of A_j closest to v, and farthest from v, respectively. Thus, if A_j is a path cluster, $\partial A_j = \{v_j, w_j\}$, otherwise $\partial A_j = \{v_j\} = \{w_j\}$. The set I defined above is the set of indices of the clusters that contain labels at level i. Then, x is picked from I to minimize $D(x) = \text{dist}(v, \text{meet}(v_x, v, w))$. If there are more than one cluster minimizing $D(x)$, we prefer clusters with at most one boundary vertex on $\pi(C)$, since any vertex u in such a cluster will have $\text{dist}(v, \text{meet}(u, v, w)) = D(x)$, which is minimal. A path cluster A_x with both boundary vertices on $\pi(C)$ is only picked if it is the only cluster minimizing $D(x)$. In either case, we know that A_x contains a vertex u with the desired label, and that any vertex in A_x minimizing $\text{dist}(v_x, \text{meet}(u, v_x, w_x))$ will suffice. Now if A_x is a vertex label, it has only one vertex, and it stores the desired user label. Otherwise, we simply split A_x and recurse

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 [23], 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 $s' \le s \le e^\epsilon s'$, 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 $(1+2^{-b})^h \le e^{2^{-b}h}$, 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 $h \le h(n) \cdot \ell_{\max}$, 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:

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

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 $\mathcal{O}(m + n(\log n)^2)$ space algorithm from [13, 14] gives the $\mathcal{O}((\log n)^3 \log n)$ time and $\mathcal{O}(m + n \log n \log\log n)$ space algorithm briefly mentioned in [23].

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 [2].

8.1 Level-based top trees, labels, and fat-bottomed trees As described in [2], 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. [2, 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 [2], we need not have single edges on the bottom most level. We may generalize this to instead have clusters of *size* $\leq Q$, that is, with at most Q edges, as the leaves of the top tree.

THEOREM 8.1. (THEOREM 16 IN [2]) *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, ε 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 8.1. *Given any $Q \geq 1$ and $B \geq 2$, one can maintain top trees of degree B and height $\mathcal{O}(\log n/\log B)$ with base clusters of size at most Q. 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 $\mathcal{O}(B(\log n/\log B) + Q)$ time.*

Proof. Given a binary levelled top tree $\mathcal{T}_2$ of height h with base clusters of size at most Q as in Theorem 8.1, 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 $\mathcal{O}(B(\log n/\log B) + Q)$. □

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. For convenience, we will call any $b(n)$-ary top tree data structure that can be implemented using the top trees from Lemma 8.1 *well-behaved*. Loosely speaking, any well-behaved top tree data structure can be modified to use linear space.

The properties of the technique are captured in the following:

LEMMA 8.2. *Suppose we have a well-behaved $b(n)$-ary top tree data structure, that uses $s(n)$ space per cluster, and spends $t(n)$ worst-case time per merge or split. Suppose further that there exists an algorithm that takes any subgraph of size q that forms a cluster, say, C, and calculates the complete information for C in $t_0(q,n)$ time, and suppose that the complete information for C has size at most $s_0(q,n)$. Finally, suppose 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 data structure maintaining the same information in top trees of height $h = O(\log n/\log b(n))$, such that the top trees use linear space in total, and have $\mathcal{O}(t(n)\cdot h(n) + t_0(q(n),n))$ update time for link, cut, and expose.

Proof. This follows directly from Lemma 8.1 by setting $Q = q(n)$ and $B = b(n)$. 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 or splits and $\mathcal{O}(1)$ create and destroy per link cut and expose. These take $t(n)$ and $t_0(q(n),n)$ time respectively. □

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 $b(n) = (\log n)^\epsilon \in \mathcal{O}(w/\log \ell_{\max})$. As noted in Lemma 8.1, 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 have each parent cluster store a *buffer* for each of its children, containing all the cover, cover^-, cover^+ and globalcover values. 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 standard bit tricks[9] to operate on the buffers for all children of a node in parallel. We will show how to implement Cover, Uncover, CoverLevel, and MinCoveredEdge, such that each of them only touches $\mathcal{O}(h(n))$ nodes and the buffers stored in those nodes.

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 $\max\{\text{packedcover}_{C,i}, \text{packedcover}^-_{C,i}\} \leq \text{cover}^-_C$, set:

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

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

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

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

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

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

  We can then compute the data for C from the buffer

[9] See e.g. [8] or [1].

as follows:

$$\text{cover}_C = \begin{cases} \min\limits_{\substack{1\le i<k \\ \partial A_i \subseteq \pi(C)}} \text{packedcover}_{C,i} & \\ & \text{if } |\partial C| = 2 \\ \ell_{\max} & \text{otherwise} \end{cases}$$

$$\text{minpathedge}_C = \begin{cases} \text{minpathedge}_{A_j} & \text{if } |\partial C| = 2 \\ \textbf{nil} & \text{otherwise} \end{cases}$$

$$\text{where } j = \operatorname*{arg\,min}_{\substack{1\le i<k \\ \partial A_i \subseteq \pi(C)}} \text{packedcover}_{C,i}$$

$$\text{globalcover}_C = \min_{1\le i<k} \text{packedglobalcover}'_{C,i}$$

$$\text{minglobaledge}_C = \text{minglobaledge}'_{C,j}$$

$$\text{where } j = \operatorname*{arg\,min}_{1\le i<k} \text{packedglobalcover}'_{C,i}$$

$$\text{cover}^-_C = -1$$

$$\text{cover}^+_C = -1$$

This can be computed in constant time, because $(\text{packedglobalcover}'_{C,i})_{\{1\le i\le k\}}$ 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($C; A_1, \ldots, A_k$) 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

$$\text{CoverLevel}(v) = \text{globalcover}_C$$

$$\text{MinCoveredEdge}(v) = \text{minglobaledge}_C$$

$$\text{where } C \text{ is the point cluster returned by root}(v)$$

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 w 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) = 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 whose parent has v or w as an internal vertex.
- 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 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[10]

$$\text{CoverLevel}(v, w) = \min_{1\le i\le k} \text{CoverLevel}(v_i, w_i)$$

$$\text{MinCoveredEdge}(v, w) = \text{MinCoveredEdge}(v_j, w_j)$$

$$\text{where } j = \operatorname*{arg\,min}_{1\le i\le k} \text{CoverLevel}(v_i, w_i)$$

$$\text{MinCoveredEdge}(v_j, w_j) = \text{minpathedge}_X$$

$$\text{where } X = \operatorname*{arg\,min}_{Y \text{ path child of } C_{f_j}} \text{cover}_Y$$

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

Cover and Uncover We are now ready to handle Cover(v, w, i) and 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,

[10] Recall that a *path child* of C is defined as a child that contains at least one edge of $\pi(C)$.

Cover(v, w, i) needs to set

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

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

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

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 Cover(v, w, i) and 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 $\mathcal{O}(n)$.

10 Saving space

We now apply the space-saving trick from Lemma 8.2 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 10.1. *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 $e \in C$.
- cover$_C$, cover^-_C, cover^+_C, globalcover$_C$, size$_C$.
- partsize$_{C,v,i}$ and 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 10.2. *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:

```
Vector x ← 0⃗
Initialize empty max-queue Q
j ← ℓ_max
for w ← each vertex in the fragment that is on π(C) do
    Mark w as visited
    x_j ← x_j + 1
    for e ← each edge incident to w not on π(C) do
        if c(e) ≥ 0 then
            Add e to Q with key c(e)
while Q is not empty do
    e ← EXTRACT-MAX(Q)
    while c(e) < j do
        x_{j−1} = x_j
        j ← j − 1
    w ← the unvisited vertex at the end of e
    Mark w as visited
    x_j ← x_j + 1
    for e ← each edge (w, u) with u unvisited do
        if c(e) ≥ 0 then
            Add e to Q with key c(e)
partsize_{C,v,i} ← x
diagsize_{C,v,i} ← M(i) · x
```

If the ith 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 8.2 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] S. Albers and T. Hagerup, *Improved parallel integer sorting without concurrent writing*, Inf. Comput., 136 (1997), pp. 25–51.

[2] S. Alstrup, J. Holm, K. D. Lichtenberg, and M. Thorup, *Maintaining information in fully dynamic trees with top trees*, ACM Trans. Algorithms, 1 (2005), pp. 243–264.

[3] T. C. Biedl, P. Bose, E. D. Demaine, and A. Lubiw, *Efficient algorithms for petersen's matching theorem*, Journal of Algorithms, 38 (2001), pp. 110 – 134.

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

[5] D. Eppstein, Z. Galil, and G. F. Italiano, *Improved sparsification*, tech. rep., 1993.

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

[7] ———, *Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees*, SIAM J. Comput., 26 (1997), pp. 484–538.

[8] M. L. Fredman and D. E. Willard, *Surpassing the information theoretic bound with fusion trees*, J. Comput. Syst. Sci., 47 (1993), pp. 424–436.

[9] H. N. Gabow, H. Kaplan, and R. E. Tarjan, *Unique maximum matching algorithms*, J. Algorithms, 40 (2001), pp. 159–183. Announced at STOC '99.

[10] D. Gibb, B. M. Kapron, V. King, and N. Thorn, *Dynamic graph connectivity with improved worst case update time and sublinear space*, CoRR, abs/1509.06464 (2015).

[11] M. R. Henzinger and V. King, *Fully dynamic 2-edge connectivity algorithm in polylogarithmic time per operation*, 1997.

[12] M. R. Henzinger and V. King, *Maintaining minimum spanning trees in dynamic graphs*, Springer Berlin Heidelberg, Berlin, Heidelberg, 1997, pp. 594–604.

[13] J. Holm, K. de Lichtenberg, and M. 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, New York, NY, USA, 1998, ACM, pp. 79–89.

[14] ———, *Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity*, J. ACM, 48 (2001), pp. 723–760.

[15] S.-E. Huang, D. Huang, T. Kopelowitz, and S. 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, Philadelphia, PA, USA, 2017, Society for Industrial and Applied Mathematics, pp. 510–520.

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

[17] D. E. Knuth, *Two notes on notation*, The American Mathematical Monthly, 99 (1992), pp. 403–422.

[18] D. 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.

[19] A. Kotzig, *On the theory of finite graphs with a linear factor II.*, 1959.

[20] K. Menger, *Zur allgemeinen Kurventheorie*, Fundamenta Mathematicae, 10 (1927).

[21] M. Patrascu and E. D. Demaine, *Logarithmic lower bounds in the cell-probe model*, SIAM Journal on Computing, 35 (2006), pp. 932–963.

[22] J. Petersen, *Die Theorie der regulären graphs*, Acta Math., 15 (1891), pp. 193–220.

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

[24] C. Wulff-Nilsen, *Faster deterministic fully-dynamic graph connectivity*, in Encyclopedia of Algorithms, 2016, pp. 738–741.

A Details of the high level algorithm

Lemma 2.1. (Essentially from [14]) *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, and size queries using a single call to FindSize.

Proof. The only part of the high level algorithm from [14] 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 nontree 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. □

```
1: function 2-EDGE-CONNECTED(v, w)
2:     return T.CONNECTED(v, w) ∧
           T.COVERLEVEL(v, w)≥ 0
3: function FINDBRIDGE(v)
4:     if T.COVERLEVEL(v)= −1 then
5:         return T.MINCOVEREDEDGE(v)
6:     else
7:         return nil
8: function FINDBRIDGE(v, w)
9:     if T.COVERLEVEL(v, w)= −1 then
10:        return T.MINCOVEREDEDGE(v, w)
11:    else
12:        return nil
13: function SIZE(v)
14:    return T.FINDSIZE(v,v,−1)
15: function 2-SIZE(v)
16:    return T.FINDSIZE(v,v,0)
17: function INSERT(v, w, e)
18:    if ¬T.CONNECTED(v, w) then
19:        T.LINK(v, w, e)
20:        ℓ(e) ← ℓmax
21:    else
22:        T.ADDLABEL(v, e.label1, 0)
23:        T.ADDLABEL(w, e.label2, 0)
24:        ℓ(e) ← 0
25:        T.COVER(v, w, 0)
26: function DELETE(e)
27:    (v, w) ← e
28:    α ← ℓ(e)
29:    if α = ℓmax then
30:        α ← T.COVERLEVEL(v, w)
31:        if α = −1 then
32:            T.CUT(e)
33:            return
34:        SWAP(e)
35:    T.REMOVELABEL(e.label1)
36:    T.REMOVELABEL(e.label2)
37:    T.UNCOVER(v, w, α)
38:    for i ← α, . . . , 0 do
39:        RECOVER(w,v,i)
40: function SWAP(e)
41:    (v, w) ← e
42:    α ← T.COVERLEVEL(v, w)
43:    T.CUT(e)
44:    e′ ←FINDREPLACEMENT(v,w,α)
45:    (x, y) ← e′
46:    T.REMOVELABEL(e′.label1)
47:    T.REMOVELABEL(e′.label2)
48:    T.LINK(x, y, e′)
49:    ℓ(e′) ← ℓmax
50:    T.ADDLABEL(v,e.label1, α)
51:    T.ADDLABEL(w,e.label2, α)
52:    ℓ(e) ← α
53:    T.COVER(v,w,α)
54: function FINDREPLACEMENT(v,w,i)
55:    s_v ← T.FINDSIZE(v, v, i)
56:    s_w ← T.FINDSIZE(w, w, i)
57:    if s_v ≤ 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 ← ⌊T.FINDSIZE(v, w, i)/2⌋
63:    RECOVERPHASE(v,w,i,s)
64:    RECOVERPHASE(w,v,i,s)
65: function RECOVERPHASE(v, w, i, s)
66:    l ← T.FINDFIRSTLABEL(v, w, i)
67:    while l ≠ nil do
68:        e ← l.edge
69:        (q, r) ← e
70:        if ¬T.CONNECTED(q, r) then
71:            return e
72:        if T.FINDSIZE(q, r, i + 1) ≤ 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:            ℓ(e) = i + 1
78:            T.COVER(q,r,i + 1)
79:        else
80:            T.COVER(q,r,i)
81:            return nil
82:        l ← T.FINDFIRSTLABEL(v, w, i)
83:    return nil
```

B Pseudocode for the CoverLevel structure

```
1: function CL.COVER(v,w,i)
2:     C ← TOPTREE.EXPOSE(v, w)
3:     cover_C ← max {cover_C, i}
```

#	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.

```
    cover_C^+ ← max{cover_C^+, i}
function CL.Uncover(v,w,i)
    C ← 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, ..., A_k)
    cover_C ← ℓ_max
    minpathedge_C ← nil
    globalcover_C ← ℓ_max
    minglobaledge_C ← nil
    for i ← 1, ..., k do
        if ∂A_i ⊆ π(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
    globalcover_C ← −1
    if C is a point cluster then
        minpathedge_C ← nil
        minglobaledge_C ← e
    else
        minpathedge_C ← e
        minglobaledge_C ← nil
    cover_C^− ← −1
    cover_C^+ ← −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.

function FS.FindSize(v, w, i)
  $C \leftarrow$ TopTree.Expose(v, w)
  return $\text{size}_{C,i}$
function FS.Merge(C; A, B)
  $\{c\} \leftarrow \partial A \cap \partial B$
  if $c \in \pi(C)$ **then** ▷ Merge 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 $b \in \partial B$.
    $\text{size}_C \leftarrow \text{size}_A + \text{size}_B$
    for $(x, X) \leftarrow (a, A), (b, B)$ **do**
      if $x = c$ **then**
        $\text{tree}'_{X,x} \leftarrow \text{tree}_{X,x}$
        $\text{undo}'_{X,x} \leftarrow$ **nil**
      else
        for $v \leftarrow x, c$ **do**
          $\ell \leftarrow \max\left\{\text{cover}^-_X, \text{cover}^+_X\right\}$
          $s \leftarrow (\text{tree}_{X,v}).\,\text{partsizesum}$
          $d \leftarrow M(\text{cover}^+_X) * s$
          $\text{tree}'_{X,v} \leftarrow \text{tree}_{X,v,\{i>\ell\}}$
          $\text{undo}'_{X,v} \leftarrow \text{tree}_{X,v,\{i\le\ell\}}$
          $\text{tree}'_{X,v} \leftarrow \text{tree}'_{X,v} + [\text{cover}^+_X : s, d]$
    for $(x, X, y, Y) \leftarrow (a, A, b, B), (b, B, a, A)$ **do**
      $s \leftarrow (\text{tree}'_{Y,c,\{\text{cover}_X \le i \le \ell_{\max}\}}).\,\text{partsizesum}$
      $p \leftarrow (\text{tree}'_{X,x,\text{cover}_X}).\,\text{partsize} + s$
      $d \leftarrow (\text{tree}'_{X,x,\text{cover}_X}).\,\text{diagsize}$
      $d \leftarrow d + M(\text{cover}_X) * s$
      if $x = c$ **then**
        $\text{tree}''_{X,x} \leftarrow [\ell_{\max} : \text{size}_X, \text{size}_X]$
        $\text{undo}''_{X,x} \leftarrow$ **nil**
      else
        $\text{tree}''_{X,x} \leftarrow \text{tree}'_{X,x,\{i>\text{cover}_X\}}$
        $\text{undo}''_{X,x} \leftarrow \text{tree}'_{X,x,\{i\le\text{cover}_X\}}$
      if $y = c$ **then**
        $\text{tree}'''_{Y,c} \leftarrow$ **nil**
        $\text{undo}'''_{Y,c} \leftarrow [\ell_{\max} : \text{size}_Y, \text{size}_Y]$
      else
        $\text{tree}'''_{Y,c} \leftarrow \text{tree}'_{Y,c,\{i<\text{cover}_X\}}$
        $\text{undo}'''_{Y,c} \leftarrow \text{tree}'_{Y,c,\{i\ge\text{cover}_X\}}$
      $\text{tree}_{C,x} \leftarrow \text{tree}''_{X,x} + [\text{cover}_X : p, d] + \text{tree}'''_{Y,c}$
  else ▷ Merge off path
    $\{a\} \leftarrow \partial C \setminus \{c\}$
    if $a \notin \partial A$ **then**
      Swap A and B
    $\ell \leftarrow \max\left\{\text{cover}^-_A, \text{cover}^+_A\right\}$
    $d \leftarrow (\text{tree}_{A,a,\{\ell<i\le\ell_{\max}\}}).\,\text{diagsizesum}$
    $p \leftarrow (\text{tree}_{A,a,\{-1\le i\le\ell\}}).\,\text{partsizesum}$
    $\text{size}_C \leftarrow d + M(\text{cover}^+_A) * p + M(\text{cover}_A) * \text{size}_B$
    $\text{tree}_{C,a} \leftarrow [\ell_{\max} : \text{size}_C, \text{size}_C]$
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**
      $\text{tree}''_{X,x} \leftarrow \text{tree}_{C,x,\{i>\text{cover}_X\}}$
      $\text{tree}'''_{Y,c} \leftarrow \text{tree}_{C,x,\{i<\text{cover}_X\}}$
      if $y \neq c$ **then**
        $\text{tree}'_{Y,c} \leftarrow \text{tree}'''_{Y,c} + \text{undo}'''_{Y,c}$
      if $x \neq c$ **then**
        $\text{tree}'_{X,x} \leftarrow \text{tree}''_{X,x} + \text{undo}''_{X,x}$
    for $(x, X) \leftarrow (a, A), (b, B)$ **do**
      if $x \neq c$ **then**
        for $v \leftarrow x, c$ **do**
          $\text{tree}_{X,v} \leftarrow \text{tree}'_{X,v,\{i>\text{cover}^+_X\}} + \text{undo}'_{X,v}$
function FS.Create(C; edge e)
  $\text{size}_C \leftarrow \vec{0}$
  for $v \in \partial C$ **do**
    $\text{tree}_{C,v} \leftarrow [\ell_{\max} : \vec{0}, \vec{0}]$
function FS.Create(C; vertex label l)
  $\text{size}_C \leftarrow (1)_{\{0\le i<\ell_{\max}\}}$
  for $v \in \partial C$ **do**
    $\text{tree}_{C,v} = [\ell_{\max} : \text{size}_C, \text{size}_C]$