Downloaded 06/06/17 to 69.136.137.32. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Fully Dynamic Connectivity in $O(\log n(\log\log n)^2)$ Amortized Expected Time*

Shang-En Huang, Univ. of Michigan
Dawei Huang, Univ. of Michigan
Tsvi Kopelowitz, Univ. of Michigan
Seth Pettie, Univ. of Michigan

Abstract

Dynamic connectivity is one of the most fundamental problems in dynamic graph algorithms. We present a new randomized dynamic connectivity structure with $O(\log n(\log\log n)^2)$ amortized expected update time and $O(\log n/\log\log\log n)$ query time, which comes within an $O((\log\log n)^2)$ factor of a lower bound due to Pătraşcu and Demaine. The new structure is based on a dynamic connectivity algorithm proposed by Thorup in an extended abstract at STOC 2000, which left out some important details.

1 Introduction

The *dynamic connectivity* problem is one of the most fundamental problems in dynamic graph algorithms. The goal is to support the following three operations on an undirected graph G with n vertices:

- `Insert`(u,v): Insert a new edge (u,v) into G.
- `Delete`(u,v): Delete edge (u,v) from G.
- `Conn?`(u,v): Return `true` if and only if u and v are in the same connected component in G.

In this paper we prove the following bound on the complexity of dynamic connectivity.

THEOREM 1.1. *There exists a Las Vegas randomized dynamic connectivity data structure, that supports insertions and deletions of edges in amortized expected $O(\log n(\log\log n)^2)$ time, and answers connectivity queries in worst case $O(\log n/\log\log\log n)$ time.*

Previous Results. The dynamic connectivity problem has been studied under both worst case and amortized measures of efficiency, and in deterministic, randomized Monte Carlo, and randomized Las Vegas models. We therefore have the opportunity to see six incomparably best algorithms! Luckily, there are currently *only* four. The best deterministic-worst case update time is $O(\sqrt{\frac{n(\log\log n)^2}{\log n}})$ [10], improving on the long-standing $O(\sqrt{n})$ bound [3, 1], and the best deterministic-amortized update time is $O(\log^2 n/\log\log n)$ [16], improving on earlier $O(\log^2 n)$-time algorithms [8, 14] (see also [6, 7]). Kapron et al. [9] designed a worst case randomized *Monte Carlo* algorithm with $O(\log^5 n)$ update time, that is, there is some $1/\operatorname{poly}(n)$ probability of answering a connectivity query incorrectly. The update time was recently improved to $O(\log^4 n)$ [4] for deletion and $O(\log^3 n)$ [15] for insersion. In all dynamic connectivity algorithms the update time determines the query time [6]: $O(t(n)\log n)$ update implies $O(\log n/\log t(n))$ query time; see Theorem 2.1.

Thorup [14], in an extended abstract presented at STOC 2000, proposed a *Las Vegas* randomized-amortized algorithm with update time $O(\log n(\log\log n)^3)$, that is, queries must be answered correctly with probability 1, and the total update time for m updates is a random variable, which is $m\cdot O(\log n(\log\log n)^3)$ in expectation. Unfortunately, the extended abstract [14] sketched or omitted a few critical data structural details. The problem of completing Thorup's research program has, over the years, evolved into an important open research problem in the area of dynamic graph algorithms. A bound of $O(\log n\operatorname{poly}(\log\log n))$ is substantially better than the best worst case and/or deterministic algorithms [10, 8, 9, 16], and comes within a tiny $\operatorname{poly}(\log\log n)$ factor of known cell-probe lower bounds [11, 12].

Pătraşcu and Demaine [11] showed that for $t(n) = \Omega(1)$, update time $O(t(n)\log n)$ implies query time $\Omega(\log n/\log t(n))$ and Pătraşcu and Thorup [12] showed that there is no similar tradeoff in the reverse direction, that update time $o(\log n)$ implies $\Omega(n^{1-o(1)})$ query time. Whether there is a dynamic connectivity structure supporting all operations in $O(\log n)$ time (even amortized) is one of the main open questions in this area. This bound has only been achieved on forests [13] and embedded planar graphs [2].

*Supported by NSF grants CCF-1217338, CNS-1318294, and CCF-1514383. Contact: sehuang@umich.edu, dwhuang9@gmail.com, kopelot@gmail.com, seth@pettie.net

Copyright © by SIAM
Unauthorized reproduction of this article is prohibited

Downloaded 06/06/17 to 69.136.137.32. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Our Contribution. Thorup [14] proposed a dynamic connectivity structure based on four innovative ideas: (1) using a single, hierarchical representation of the graph, (2) imposing an overlay network of *shortcuts* on this representation in order to navigate between certain nodes in $O(\log \log n)$ time, (3) using random sampling (as in [7, 6]) to find replacement edges after `Delete` operations, and (4) maintaining a system of *approximate counters* to facilitate efficient random sampling of edges. The interactions between these four elements is rather complex. Dynamic changes in the hierarchy (1) may require destroying and rebuilding the shortcuts in (2), and may invalidate the approximate counters in (4). In order for (3) to work correctly the approximate counters must be very accurate.

In this paper we use the same tools introduced by Thorup, but apply them differently in order to simplify parts of the algorithm, to accommodate a proof of correctness, and improve the expected amortized update time to $O(\log n(\log \log n)^2)$. Here is a summary of the technical differences.

- Thorup [14] (as in [8, 16]) assigns each edge a *depth* (aka level) between 1 and $\log n$ and maintains a spanning forest $\mathcal{F}$. Depths are non-decreasing over time, so we can charge each depth promotion (from i to $i+1$) $(\log \log n)^2$ units of work. The depths of $\mathcal{F}$-edges induce a hierarchy $\mathcal{H}$, which is then refined into a binary hierarchy, $\mathcal{H}^b$, by substituting "local trees" connecting each $\mathcal{H}$-node to its $\mathcal{H}$-children. One of the primitive operations supported by the hierarchy $\mathcal{H}$ is to return an almost uniformly random depth-i edge touching some component corresponding to an $\mathcal{H}$-node. To implement this random sampling efficiently one needs a system of shortcuts and approximate counters. However, it is not obvious how to efficiently *maintain* approximate counters after edge promotions. Our data structure uses a more complicated classification of edges, which simplifies how approximate counters are implemented and analyzed. Each edge has a depth, as before, and each edge is either a *witness* (in $\mathcal{F}$) or *non-witness*. The endpoints of a depth-i non-witness edge can be either *primary* or *secondary*. We only keep approximate counters for i-primary endpoints, and only sample i-primary endpoints. When an edge is promoted from depth $i-1$ to i, its endpoints are secondary, so there is no immediate need to update approximate counters for depth i. So long as good replacement edges can be found by sampling from the pool of i-primary endpoints we are happy, but if none can be found we are also happy to spend some time promoting depth-i edges to depth-$(i+1)$, and *upgrading* i-secondary endpoints to i-primary status. Since each edge's endpoints can be upgraded at most $2\log n$ times over the lifetime of the edge, each upgrade can also be charged $(\log \log n)^2$ units of work. Whenever we upgrade i-secondary endpoints to i-primary status, we are guaranteed that the number of promotions/upgrades is large enough to *completely rebuild* the system of approximate counters for a pool of i-primary endpoints.

- One of Thorup's [14] ideas was to maintain $\log n$ forests (one for each edge depth) on different subsets of the $\mathcal{H}^b$-nodes, via a system of *shortcuts*. However, to be efficient it is important that these forests share shortcuts whenever possible. We provide a new method for storing and updating shortcuts, that allows us to find the right shortcut at a $\mathcal{H}^b$-node in $O(1)$ time, and update information on *all* the shortcuts at a $\mathcal{H}^b$-node in $O(\log \log n)$ time.

- We give a simpler random sampling procedure, which can be regarded as a two-stage version of the "provide or bound" routine of [6]. Our random sampling procedure is necessarily somewhat different than [14] because of the classification of non-witness edges into *primary* and *secondary*. The routine must either (i) provide a replacement edge with an i-primary endpoint, or (ii) determine that the fraction of such edges is less than a certain constant, with high probability. In case (ii) the procedure has found (statistical) evidence that there will be enough promotions/upgrades to pay for converting i-secondary endpoints to i-primary, promoting depth-i edges to depth-$(i+1)$, and rebuilding i-primary approximate counters.

- The structure of $\mathcal{H}$ is uniquely determined by the depths of witness ($\mathcal{F}$) edges, and $\mathcal{H}^b$ is a binary refinement of $\mathcal{H}$. In Thorup's [14] system $\mathcal{H}^b$ is only modified in response to structural changes in $\mathcal{H}$, due to promotions of witness edges in $\mathcal{F}$. A key invariant maintained by our data structure is that certain approximate counters, once initialized, are only subject to decrements, never increments. Thus, to preserve this invariant we actually update $\mathcal{H}^b$ in response to non-witness edge promotions/upgrades, which necessarily have no effect on $\mathcal{H}$.

Organization of the Paper. In Section 2 we review several fundamental concepts of dynamic connectivity algorithms. Section 3 gives a detailed overview of the data structure invariants and its three main components: maintaining a binary *hierarchical* representation

Copyright © by SIAM
Unauthorized reproduction of this article is prohibited

Downloaded 06/06/17 to 69.136.137.32. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

of the graph, maintaining *shortcuts* for efficient navigation around the hierarchy, and maintaining a system of *approximate counters* to support $O(1)$-approximate random sampling. Each of these three main components is explained in great detail in the arXiv version of the paper[1].

2 Preliminaries

In this section we review some basic concepts and invariants used in prior dynamic connectivity algorithms [6, 5, 8, 14, 16].

Witness Edges, Witness Forests and Replacement Edges. A common method for supporting connectivity queries is to maintain a spanning forest $\mathcal{F}$ of G called the *witness forest*, together with a dynamic connectivity structure on $\mathcal{F}$. Each edge in the witness forest is called a *witness* edge and all others *non-witness* edges. Notice that deleting a non-witness edge does not change the connectivity. A dynamic connectivity data structure for $\mathcal{F}$ supports fast queries via Theorem 2.1.

THEOREM 2.1. (HENZINGER AND KING [5]) *For any function $t(n) = \Omega(1)$, there exists a dynamic connectivity data structure for forests with $O(t(n) \log n)$ update time and $O(\log n / \log t(n))$ query time.*

The difficulty in maintaining a dynamic connectivity data structure is to find a *replacement edge* e' when a witness edge $e \in \mathcal{F}$ is deleted, or determine that no replacement exists. To speed up the search for replacement edges we maintain Invariant 1 (below) governing edge *depths*.

Edge Depths. Each edge e has a depth $d_e \in [1, d_{max}]$, where $d_{max} = \lfloor \log n \rfloor$. Let E_i be the set of edges with depth i. All edges are inserted at depth 1 and depths are non-decreasing over time. Incrementing the depth of an edge is called a *promotion*. Since we are aiming for $O(\log n (\log \log n)^2)$ amortized time per update, if the actual time to promote an edge set S is $O(|S| \cdot (\log \log n)^2)$, the amortized time per promotion is zero. Promotions are performed in order to maintain Invariant 1. There are other at most $O(\log n)$ status changes that an edge will undergo, each affording $O((\log \log n)^2)$ work. Define $G_i = (V, \bigcup_{j \geq i} E_j)$.

INVARIANT 1. (THE DEPTH INVARIANT)

(1) (Spanning Forest Property) $\mathcal{F}$ is a maximum spanning forest of G with respect to the depths.

(2) (Weight Property) For each $1 \leq i \leq d_{max}$, each connected component in the subgraph G_i contains at most $n/2^{i-1}$ vertices.

[1] https://arxiv.org/abs/1609.05867

Hierarchy of connected components. Define $\hat{V}_i$ to be in one-to-one correspondence with the connected components of G_{i+1}, which are called $(i+1)$-*components*. If $u \in V$, let $u^i \in \hat{V}_i$ be the unique $(i+1)$-component containing u. Define $\hat{G}_i = (\hat{V}_i, \hat{E}_i)$ to be the *multigraph* (including parallel edges and loops) obtained by contracting edges with depth above i and discarding edges with depth below i, so $\hat{E}_i = \{(u^i, v^i) \mid (u, v) \in E_i\}$. The hierarchy $\mathcal{H}$ is composed of the undirected multi-graphs $\hat{G}_{d_{max}}, \hat{G}_{d_{max}-1}, \ldots, \hat{G}_0$. An edge $e = (u, v) \in E_i$ is said to be *touching* all nodes $x^j \in \hat{V}_j$ where either $u^j = x^j$ or $v^j = x^j$.

Let $F_i = E_i \cap \mathcal{F}$ be the set of *i-witness edges*; all other edges in $E_i - F_i$ are *i-non-witness edges*. By Invariant 1, F_i corresponds to a spanning forest of $\hat{G}_i$. The *weight* $w(u^i)$ of a node $u^i \in \hat{V}_i$ is the number of vertices in its component: $w(u^i) = |\{v \in V \mid v^i = u^i\}|$. The data structure explicitly maintains the exact weight of all hierarchy nodes. The weight property in Invariant 1 can be restated as $w(u^i) \leq n/2^i$.

Endpoints. The endpoints of an edge $e = (u, v)$ are the pairs $\langle u, e \rangle$ and $\langle v, e \rangle$. At one stage in our algorithm we sample a random endpoint from $E' \subset E$ incident to a set $V' \subset V$; this means that an edge $(u, v) \in E'$ is sampled with probability proportional to $|\{u, v\} \cap V'|$. An endpoint $\langle u, e \rangle$ is said to be *touching* the nodes $u^i \in \hat{V}_i$ for all $i \in [1, d_{max}]$.

3 Overview of the Data Structure

Following the key invariant in [8, 14, 16], the main goal is summarized as the following lemma:

LEMMA 3.1. *Invariant 1 is maintained throughout updates to G.*

In the rest of this section, we provide an overview of the data structure. The underlined parts of the text refer to primitive data structure operations supported by Lemma 3.2, presented in Section 3.3.

The data structure. The hierarchy $\mathcal{H}$ naturally defines a rooted forest (not to be confused with the spanning forest), which is called the *hierarchy forest*, and contains several *hierarchy trees*. We abuse notation and say that $\mathcal{H}$ refers to this hierarchy forest, together with several auxiliary data structures supporting operations on the forest. The nodes in $\mathcal{H}$ are the i-components for all $1 \leq i \leq d_{max}$. The roots of the hierarchy trees are nodes in $\hat{V}_0$, representing 1-components. The set of nodes at depth i is exactly $\hat{V}_i$. The set of children of a node v^i at depth i is $\{u^{i+1} \in \hat{V}_{i+1} \mid u^i = v^i\}$. The leaves are nodes in $\hat{V}_{d_{max}} = V$. See Figure 1 for an example. The nodes in $\mathcal{H}$ are called *$\mathcal{H}$-nodes*, and the roots are called *$\mathcal{H}$-roots*. Each non-leaf $\mathcal{H}$-node v is as-

Copyright © by SIAM
Unauthorized reproduction of this article is prohibited

Downloaded 06/06/17 to 69.136.137.32. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

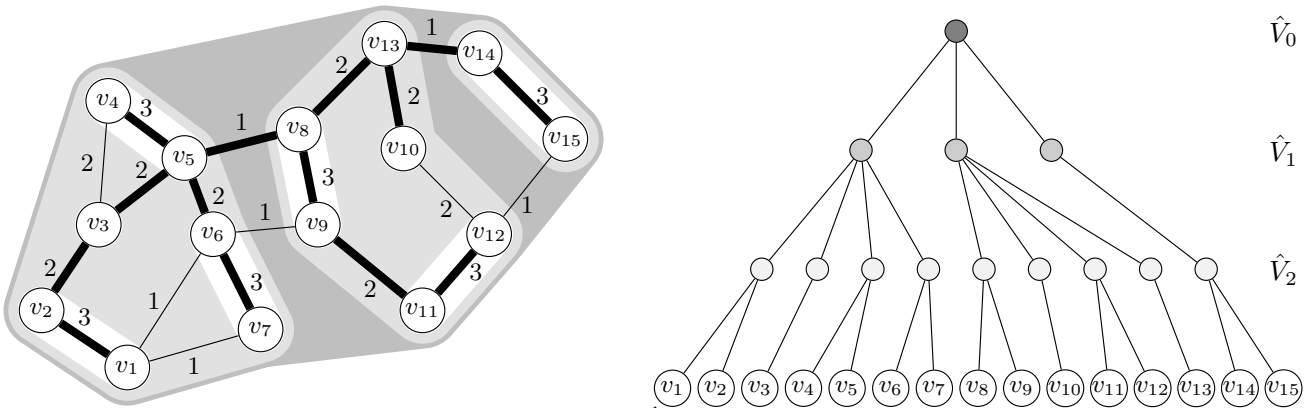

Figure 1: An illustration of a graph and the corresponding hierarchy forest $\mathcal{H}$, where $n = 15$ and $d_{max} = 3$. All thick edges are witness edges and the thin edges are non-witness edges.

sociated with a binary local tree, implicitly supporting operations between v and its $\mathcal{H}$-*children*.

3.1 Insertion. To execute an `insert`(u, v) operation, where $e = (u, v)$, the data structure first sets $d_e = 1$. If e connects two distinct components in G (which is verified by a connectivity query on $\mathcal{F}$), then the data structure accesses two $\mathcal{H}$-roots u^0 and v^0, merges u^0 and v^0 and e is inserted into $\mathcal{H}$ (and $\mathcal{F}$) as a 1-witness edge. Otherwise, e is inserted into $\mathcal{H}$ as a 1-non-witness edge.

3.2 Deletion. By the Spanning Forest Property of Invariant 1, the deletion of an edge e can only be replaced by edges of depth d_e or less. We always first look for a replacement edge at the same depth of the deleted edge. If we do not find a replacement edge at depth d_e then we *demote* e by setting $d_e \leftarrow d_e - 1$, which preserves Invariant 1, and continue looking for replacement edges at the new depth d_e. Demotion is merely conceptual; the deletion algorithm does not actually update d_e in the course of deleting e.

To execute a `delete`(u, v) operation, where $e = (u, v)$, the data structure first removes e from $\mathcal{H}$. If e is an i-non-witness edge, then the deletion process is done. If e is an i-witness edge, the deletion of e could split an i-component. Specifically, prior to the deletion, the edge (u^i, v^i) connected two $(i + 1)$-components, u^i and v^i, which, possibly together with some additional i-witness edges and $(i+1)$-components, formed a single i-component $u^{i-1} = v^{i-1}$ in $\hat{G}_i$. If no i-non-witness replacement edge exists, then deleting (u, v) splits u^{i-1} into two i-components. In order to establish if this is the case, the data structure first accesses u^i, v^i and u^{i-1} in $\mathcal{H}$ and implicitly splits the i-component u^{i-1} into two connected components c_u and c_v in $\hat{F}_i = (\hat{V}_i, \{(u^i, v^i) \mid (u, v) \in F_i\})$ where $u^i \in c_u$ and $v^i \in c_v$ (we define c_u and c_v but without context to the subscripts, see Figure 2.*a*). The rest of the deletion process focuses on finding a replacement edge to reconnect c_u and c_v into one i-component. This process has two parts, explained in detail below: (1) establishing the two components c_u and c_v, and (2) finding a replacement edge. Notice that c_u and c_v do not correspond to $\mathcal{H}$-nodes.

3.2.1 Establishing Two Components. To establish the two components c_u and c_v created by the deletion of e, the data structure executes in parallel two depth first searches (DFS) on $\hat{F}_i - \{(u^i, v^i)\}$, one DFS starting from u^i and one DFS starting from v^i. To implement a DFS, the data structure repeatedly enumerates all i-witness edge endpoints touching an $(i+1)$-component. The DFSs are carried out in parallel until one of the connected components is fully scanned. By fully scanning one component, the weights of both components are determined (since $w(u^{i-1}) = w(c_u) + w(c_v)$). Without lost of generality, assume that $w(c_u) \leq w(c_v)$, and so by Invariant 1, $w(c_u) \leq w(u^{i-1})/2 \leq n/2^i$.

Witness Edge Promotions. The data structure promotes all i-witness edges touching nodes in c_u and merges all $(i+1)$-components contained in c_u into one $(i+1)$-component with weight $w(c_u)$. This is permitted by Invariant 1, since $w(c_u) \leq w(u^{i-1})/2 \leq n/2^i$. The merged $(i+1)$-component has the node u^{i-1} as its parent in $\mathcal{H}$. See Figure 2.*b*.

To differentiate between versions of components before and after the merges, we use a convention where bold notation refers to the components after the merges take place. Thus, we deonte the $(i + 1)$-component contracted from all $(i+1)$-components inside c_u by $\mathbf{u^i}$.

Copyright © by SIAM
Unauthorized reproduction of this article is prohibited

Downloaded 06/06/17 to 69.136.137.32. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

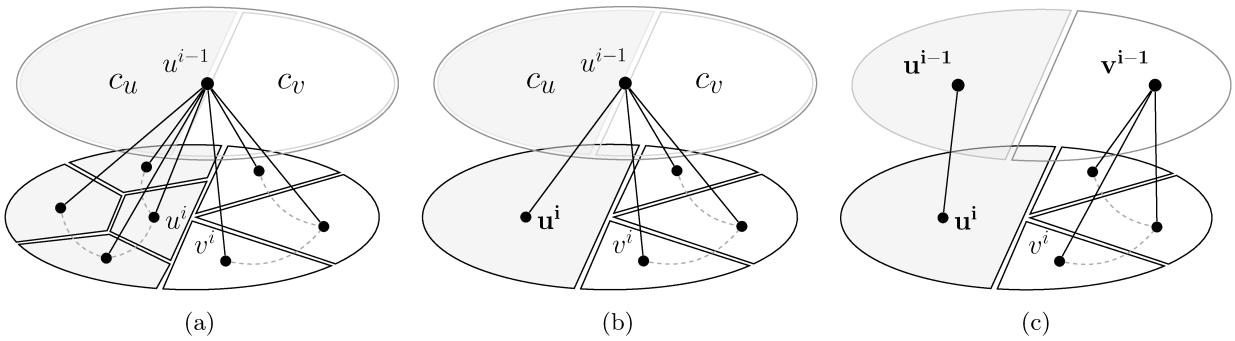

Figure 2: Illustration of the hierarchy of components at depth $i-1$ and i: (a) After identifying two components c_u and c_v, it turns out that c_u has smaller weight although it has more $(i+1)$-components. (b) After merging all $(i+1)$-components in the smaller weight component. (c) If no replacement edge is found, then c_u and c_v are two actual connected components in $\hat{G}_i$ and hence $\mathbf{u^i}$ is split.

Similarly, the graph $\hat{G}_i$ after merging some of its nodes is denoted by $\hat{\mathbf{G}}_\mathbf{i}$.

Having contracted the $(i+1)$-components inside c_u into $\mathbf{u^i}$, we now turn our attention to identifying whether the deletion of e disconnects $\mathbf{u^i}$ from c_v in $\hat{\mathbf{G}}_\mathbf{i}$. This task reduces to determining whether there exists an edge in $\hat{\mathbf{G}}_\mathbf{i}$ that reconnects $\mathbf{u^i}$ to any $(i+1)$-components in $u^{i-1} \setminus \mathbf{u^i}$.

3.2.2 Finding a Replacement Edge. Notice that by definition of $\hat{\mathbf{G}}_\mathbf{i}$ and u^{i-1}, a depth i edge is a replacement edge in E if and only if it is an i-non-witness edge with exactly one endpoint $x \in V$ such that $x^i = \mathbf{u^i}$. To find a replacement edge, the data structure executes one or both of the following two auxiliary procedures: the *sampling procedure* and the *enumeration procedure*.

Intuition. Consider these two situations. In Situation A at least a constant fraction of the i-non-witness edges touching $\mathbf{u^i}$ have exactly one endpoint touching $\mathbf{u^i}$, and are therefore eligible replacement edges. In Situation B a small ϵ fraction (maybe zero) of these edges have exactly one endpoint in $\mathbf{u^i}$. If we magically *knew* which situation we were in and could sample i-non-witness endpoints uniformly at random then the problem would be easy. In Situation A we would iteratively sample an i-non-witness endpoint and test whether the other endpoint was in u^i; each test takes $O(\log n \log\log n)$ time. The expected number of samples required to find a replacement edge is $O(1)$ and this cost would be charged to the deletion operation. In Situation B we would enumerate and mark every i-non-witness endpoint touching $\mathbf{u^i}$. Any edge with one mark is a replacement edge and any with two marks can be promoted to depth $i+1$. Since a majority of the edges will end up being promoted, the amortized cost of the enumeration procedure is zero, so long as the enumeration and promotion cost is $O((\log\log n)^2)$ per endpoint.

There are two technical difficulties with implementing this idea. First, the set of i-non-witness edges incident to $\mathbf{u^i}$ is a dynamically changing set, and supporting (almost-)uniformly random sampling on this set is a very tricky problem. Second, we do not know which situation, A or B, we are in. Note that it is insufficient to take $O(1)$ random samples and, if no replacement edges are found, deduce that we are in Situation B. Because the cost of enumeration is so high, we cannot afford to *mistakenly* think we are in Situation B unless the probability of error is inversely proportional to the cost of enumeration.

Thorup [14] addresses the first difficulty by maintaining a system of approximate counters and two layers of overlay networks,[2] and solves the second difficulty by using the "provide or bound" sampling procedure of [7].

Primary and Secondary Endpoints — A Simpler Sampling Method. The difficulty with supporting random sampling is dynamic updates: i-non-witness edges are inserted and deleted from the pool due to promotions, and we want to update various counters in response to each insertion/deletion. However, the number of counters that need to be updated turns out to be too large. Our solution is to maintain two endpoint types for i-non-witness edges: *primary* and *secondary*. A newly promoted i-non-witness edge has two *i-secondary endpoints* and when an i-secondary end-

[2]The first overlay network, which we also use, supports navigation to the $\mathcal{H}$-leaves incident to i-non-witness edges. The second overlay network, which is sketched in [14], is derived from a heavy-path decomposition of the first overlay network in order to guarantee some degree of balance. The second overlay network is used to facilitate dynamic updates to the approximate counters.

Copyright © by SIAM
Unauthorized reproduction of this article is prohibited

Downloaded 06/06/17 to 69.136.137.32. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

point is enumerated (see below), the data structure *upgrades* that endpoint into an *i-primary endpoint*. The set of i-secondary endpoints is subject to individual insertions, but we never sample from the i-secondary endpoints. The set of i-primary endpoints is subject to bulk inserts/deletes, which are sufficiently large to pay for completely rebuilding the counters necessary to support random sampling. Rather than use the full power of [7], we give a simpler two-stage sampling procedure that either provides a replacement edge or states that, with high enough probability, the fraction of i-primary endpoints touching u^i that belongs to replacement edges is small. If so, we enumerate all i-primary and i-secondary endpoints touching u^i, upgrading i-secondary endpoints at replacement edges to i-primary endpoints, and promoting non-replacement i-non-witness edges to $(i+1)$-non-witness edges. The cost for rebuilding the i-primary sampling structure is amortized $O((\log\log n)^2)$ time per promotion or upgrade, and therefore paid for.

We now give a more detailed description of the sampling procedure.

The Sampling Procedure. This is the only procedure that uses randomness. The procedure uses subroutines for $(1+o(1))$-approximating the number s of i-primary endpoints touching $\mathbf{u^i}$, for $(1+o(1))$-uniformly sampling an endpoint of an i-primary edge touching $\mathbf{u^i}$, and for enumerating every i-primary/i-secondary endpoint touching $\mathbf{u^i}$. The sampling procedure can be viewed as a two-stage version of Henzinger and Thorup [7]. The data structure first estimates s up to a constant factor and then invokes the batch sampling test, which $(1+o(1))$-uniformly samples $O(\log\log s)$ i-primary endpoints touching $\mathbf{u^i}$. If an endpoint of a replacement edge is sampled, then the sampling procedure is terminated, returning one of the replacement edges. Otherwise, the data structure invokes the batch sampling test, which $(1+o(1))$-uniformly samples $O(\log s)$ i-primary endpoints touching $\mathbf{u^i}$. The purpose of this step is *not* to find a replacement edge, but to *increase our confidence* that there are actually few replacement edges. If more than half of these endpoints belong to replacement edges, the sampling procedure is terminated and one replacement edge is returned. Otherwise, the data structure concludes that the fraction of the non-replacement edges touching $\mathbf{u^i}$ is at least a constant, and invokes the *enumeration procedure*.

The Enumeration Procedure. The data structure first upgrades all i-secondary endpoints touching $\mathbf{u^i}$ to i-primary endpoints, enumerates all i-primary endpoints touching $\mathbf{u^i}$ and establishes for each such edge how many of its endpoints touch $\mathbf{u^i}$ (either one or both). An edge is a replacement edge if and only if exactly one of its endpoints is enumerated. Each non-replacement edge encountered by the enumeration procedure has both endpoints in an $(i+1)$-component, namely $\mathbf{u^i}$, and can therefore be promoted to be a depth $(i+1)$-non-witness edge (making both endpoints secondary), without violating Invariant 1. After all promotions and upgrades are completed, the sampling structure for i-primary endpoints touching $\mathbf{u^i}$ is rebuilt.

3.2.3 Iteration and Conclusion. If a replacement edge e' exists, then u^{i-1} is still an i-component and the data structure converts e' from an i-non-witness edge to an i-witness edge. Otherwise, c_u and c_v form two distinct i-components in $\hat{\mathbf{G}}_\mathbf{i}$. In this case, depending on i, the data structure splits u^{i-1} into two sibling nodes or two $\mathcal{H}$-roots: a new node $\mathbf{u^{i-1}}$ representing c_u whose only child is $\mathbf{u^i}$, and $\mathbf{v^{i-1}}$ representing c_v whose children are the rest of the $(i+1)$-components in c_v. Recall that while there may not be an i-non-witness replacement edge for e, there may be one at a lower depth, by the Spanning Forest Property. Therefore, if $i=1$ then we are done. Otherwise, we set $i=i-1$, conceptually demoting e, and repeat the procedure as if e were deleted at depth $i-1$.

3.3 The Backbone of the Data Structure. Lemma 3.2 summarizes the primitive operations required to execute and `Insert` or `Delete`. Remember that the possible depths are integers in $[1, d_{max}]$, and that the possible endpoint types are WITNESS, PRIMARY and SECONDARY.

LEMMA 3.2. *There exists a data structure supporting the following operations on $\mathcal{H}$ with the following amortized time complexities (in parentheses):*

(1) Add or remove an edge with a given edge depth and endpoint type $\big(O(\log n(\log\log n)^2)\big)$.

(2) Given a set S of sibling $\mathcal{H}$-nodes or $\mathcal{H}$-roots, merge them into a single node $\mathbf{u^i}$, and then promote all i-witness edges touching $\mathbf{u^i}$ into $(i+1)$-witness edges. $(O(k(\log\log n)^2+1)$, *where k is the number of i-witness edges touching $\mathbf{u^i}$).*

(3) Given an $\mathcal{H}$-node $v^i \in \hat{V}_i$, upgrade all i-secondary endpoints associated with v^i to i-primary endpoints $(O((p+s)(\log\log n)^2+1)$, *where p and s denote the number of i-primary endpoints and i-secondary endpoints touching v^i prior to the upgrade).*

(4) Given an $\mathcal{H}$-node $v^i \in \hat{V}_i$ and a subset of i-primary endpoints associated with v^i, promote them to $(i+1)$-secondary endpoints. $(O(k(\log\log n)^2+1)$, *where k is the number of* **all** *i-primary endpoints associated with v^i).*

Copyright © by SIAM
Unauthorized reproduction of this article is prohibited

Downloaded 06/06/17 to 69.136.137.32. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

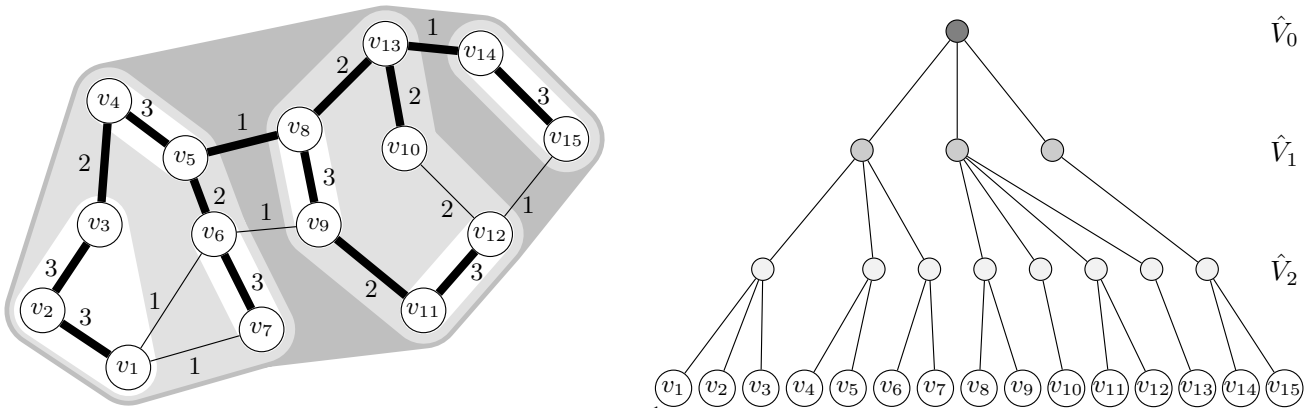

Figure 3: After deletion of (v_3, v_5) (See Figure 1.) By identifying $\{v_1, v_2, v_3\}$ to be the smaller weight component, the witness edge (v_2, v_3) is promoted and the corresponding nodes in $\hat{V}_2$ is merged. The edge (v_3, v_4) is the replacement edge.

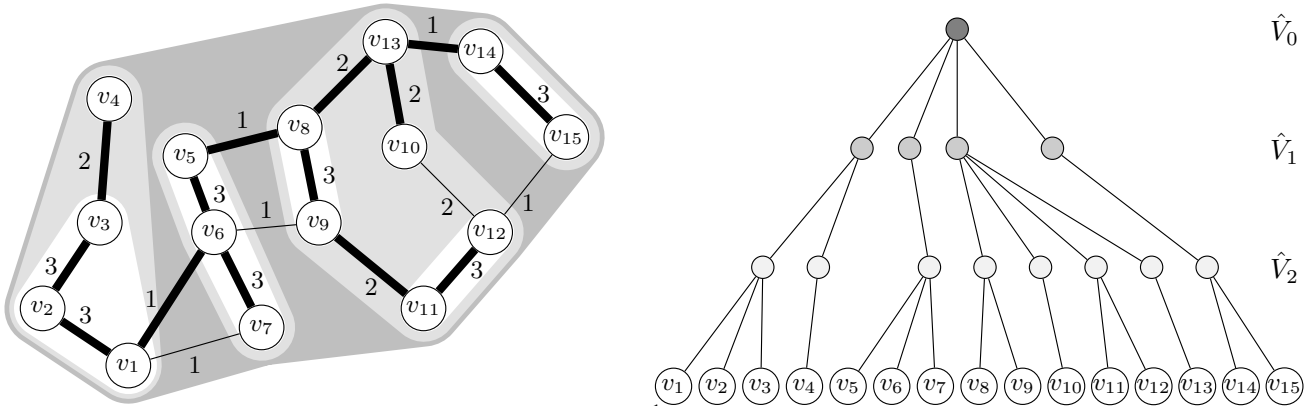

Figure 4: After deletion of (v_4, v_5): (1) Split the node in $\hat{V}_2$ associated with v_4 and v_5. (2) Identify that $\{v_5, v_6, v_7\}$ is the smaller weight component. (3) Merge nodes v_5^2 and $v_6^2 = v_7^2$. (4) Split the node v_5^1. (5) Found replacement edge (v_1, v_6).

(5) Convert a given i-non-witness edge into an i-witness edge $\big(O(\log n(\log\log n)^2)\big)$.

(6) Given two $\mathcal{H}$-nodes u^{i-1} and u^i where u^i is an $\mathcal{H}$-child of u^{i-1}, split u^{i-1} into two sibling $\mathcal{H}$-nodes: one with u^i as a single $\mathcal{H}$-child and the other with the rest of u^{i-1}'s former $\mathcal{H}$-children as its $\mathcal{H}$-children $\big(O((\log\log n)^2)\big)$.

(7) Given an $\mathcal{H}$-node $v^i \in \hat{V}_i$ and a given endpoint type, enumerate all endpoints $\langle u, e\rangle$ where e is of the given endpoint type, $d_e = i$, and $u^i = v^i$. $(O(k\log\log n + 1)$*, where k is the number of enumerated endpoints*$)$.

(8) Given v^i, return v^{i-1} $\big(O(\log(w(v^{i-1})) - \log(w(v^i)) + \log\log n)\big)$.

(9) Given an $\mathcal{H}$-node $v^i \in \hat{V}_i$, return a $(1 + o(1))$-approximation to the number of i-primary endpoints touching v^i $(O(1))$.

(10) (Batch Sampling Test) Given an $\mathcal{H}$-node $v^i \in \hat{V}_i$ and an integer k, independently sample k i-primary endpoints touching v^i $(1 + o(1))$-uniformly at random, and establish for each sampled endpoint whether the other endpoint is also in v^i. $(O(\min(k\log n\log\log n, k + (p+s)\log\log n))$*, where p and s are the number of i-primary and i-secondary endpoints touching v^i, respectively*$)$.

Notice that each data structure operation stated in Lemma 3.2 on its own does not guarantee that the data structure maintains Property 1. However, given the use of Lemma 3.2 in the description of the algorithm above, the proof of Lemma 3.1 is straightforward.

3.4 The Main Modules of the Data Structure. To support Lemma 3.2, the data structure utilizes five

Copyright © by SIAM
Unauthorized reproduction of this article is prohibited

Downloaded 06/06/17 to 69.136.137.32. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

main modules, some of which depend on each other: (1) the $\mathcal{H}$-leaf data structure (2) the notion of induced (i,t)-forests (3) the shortcut infrastructure (4) approximate counters, and (5) local trees. The $\mathcal{H}$-leaf data structure is fairly straightforward and is described in detail in Section 3.4.1. Then we define the notion of overlaying $O(\log n)$ forests on $\mathcal{H}$ in Section 3.4.2. A brief overview of the other modules is described in Sections 3.4.3, 3.4.4, and 3.4.5. In the arXiv version of the paper we provide a detailed explanation of each module.

The data structure also uses lookup tables in several modules. We describe in Section 3.5 a way to amortize the cost constructing the lookup tables. The general operations involving multiple modules, as well as the proof of Lemma 3.2 are described and analyzed in detail in the arXiv version.

3.4.1 The $\mathcal{H}$-Leaf Data Structure. The $\mathcal{H}$-leaf data structure supports the following operations: (1) Given an endpoint with a specified edge depth and endpoint type, insert or delete an edge with an endpoint at the leaf. (2) Given a depth and type, enumerate all edge endpoints incident to the leaf with that depth and type. (3) Return a uniformly random endpoint among the set of edge endpoints with a given depth and type.

To support these operations, each leaf maintains a dynamic array of endpoints for each edge depth $1 \leq i \leq d_{max}$ and each endpoint type $t \in \{\text{WITNESS}, \text{PRIMARY}, \text{SECONDARY}\}$. Hence the three operations are supported in worst case $O(1)$ time.

3.4.2 The Induced (i,t)-forest. For a given edge depth $i \in [1, d_{max}]$ and endpoint type $t \in \{\text{WITNESS}, \text{PRIMARY}, \text{SECONDARY}\}$, an $\mathcal{H}$-leaf v is an (i,t)-*leaf* if v has an endpoint with depth i and type t. An $\mathcal{H}$-node $v^i \in \hat{V}_i$ having an (i,t)-leaf in its subtree is an (i,t)-*root*. For each (i,t) pair, consider the induced forest $\mathfrak{F}$ on $\mathcal{H}$ by taking the union of the paths from each (i,t)-leaf to the corresponding (i,t)-root. An $\mathcal{H}$-node v in $\mathfrak{F}$ is an (i,t)-*node* if

- v is an (i,t)-leaf,
- v is an (i,t)-root,
- v has more than one child in $\mathfrak{F}$. In this case we call v an (i,t)-*branching node*, or
- v is an $\mathcal{H}$-child of an (i,t)-branching node but has only one $\mathcal{H}$-child in $\mathfrak{F}$. In this case we call v an *single-child* (i,t)-*node*.

Notice that an (i,t)-root may or may not be an (i,t)-branching node.

For each (i,t)-node other than an (i,t)-root, define its (i,t)-*parent* to be the nearest ancestor on $\mathfrak{F}$ that is also an (i,t)-node. An (i,t)-*child* is defined accordingly. The (i,t)-parent/child relation implicitly defines an (i,t)-*forest*, which consists of (i,t)-*trees* rooted at $\hat{V}_i$ nodes. The single-child (i,t)-nodes play a crucial role in the efficiency of traversing an (i,t)-tree. An $\mathcal{H}$-node v has an (i,t)-*status* if v is an (i,t)-node.

Storing (i,t)-status. Each node in $v \in \mathcal{H}$ stores two bitmaps of size $O(\log n)$ each, indicating whether v is an (i,t)-node, and if so then indicating whether v is an (i,t)-branching node or not.

Operations on (i,t)-forests. A key idea introduced by Thorup [14] is that edges between an (i,t)-node and its (i,t)-parent or (i,t)-children do not need to be maintained explicitly. The two components that simulate these edges are the *shortcut infrastructure*, and the *local trees* (which also use a relaxed version of the shortcut infrastructure). In particular, the shortcut infrastructure supports efficient traversals from a single-child (i,t)-node to its unique (i,t)-child, while the local trees support efficient enumeration of all (i,t)-children of an (i,t)-branching node. Lemma 3.3 summarizes the operations on (i,t)-forests which are implemented via the shortcut infrastructure and local trees, together with their corresponding time cost. We emphasize that our implementation of the operations in Lemma 3.3 imply an $O(\log\log n)$ factor improvement in time cost over the system of Thorup [14].

LEMMA 3.3. *There exists a data structure on $\mathcal{H}$ supporting the following operations:*

- *Given an $\mathcal{H}$-leaf x, make x an (i,t)-leaf* $\big(O(\log n(\log\log n)^2)\big)$.
- *Given an (i,t)-leaf x, remove the (i,t)-leaf status from x* $(O(\log n \log\log n))$.
- *Given an (i,t)-node v, return the (i,t)-parent of v.* $(O(\log\log n))$.
- *Given an (i,t)-node v, enumerate the (i,t)-children of v.* $(O(k\log\log n + 1))$ *where k is the number of enumerated (i,t)-children.*
- *Given an (i,t)-tree $\mathcal{T}$ rooted at v, an integer $i \leq i' \leq d_{max}$, an endpoint type t', and two subsets of (i,t)-leaves S^- and S^+ (these subsets need not be disjoint), update $\mathcal{H}$ so that all of the leaves in S^- lose their (i,t)-leaf status, and all leaves in S^+ gain (i',t')-leaf status (if they did not have it before)* $\big(O(|\mathcal{T}|(\log\log n)^2 + 1)\big)$.

3.4.3 The Shortcut Infrastructure. The purpose of shortcuts is to simulate a traversal from a single-child (i,t)-node to its only $\mathcal{H}$-child. This traversal costs amortized $O(\log\log n)$ time. The details and construction of

Copyright © by SIAM
Unauthorized reproduction of this article is prohibited

Downloaded 06/06/17 to 69.136.137.32. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

shortcuts is described in the arXiv version. Nevertheless, there are two main conceptual components which we introduce that allow for simplification of the shortcut system, and the improved runtime in Lemma 3.3.

Shared shortcuts and the local dictionary. Intuitively, a shortcut connects an $\mathcal{H}$-node u and a descendant v of u in $\mathcal{H}$. We say that such a shortcut *leaves* u and *enters* v. Since we are imposing $O(\log n)$ independent (i,t)-forests on $\mathcal{H}$, when $\mathcal{H}$-nodes merge or split, an inefficient implementation may necessitate updating information for several (i,t)-forests. However, notice that the paths between a single-child (i,t)-node to its (i,t)-child may overlap for several (i,t) pairs. To improve efficiency, a shortcut is shared between several (i,t)-forests, and is accessed through an $O(\log n)$ size array $\textsc{Down}_u$ with pointers to all shortcuts leaving u. Moreover, we employ a *local dictionary*, which is an array $\textsc{DownIdx}_u$ with a slot corresponding to each (i,t)-forest. Each location in $\textsc{DownIdx}_u$ stores an $O(\log\log n)$ bit index of the location in $\textsc{Down}_u$ containing the pointer to the shortcut for that specific (i,t) pair. With the local dictionary, the data structure efficiently accesses the shortcut for any specific (i,t) pair by two array lookups.

Lazy covers. One key aspect of shortcuts is that they do not *cross*, which means that if there is a shortcut between u and v, then there is no shortcut between a node in the internal path between u and v (exclusive) and a node that is either a proper descendent or proper ancestor of both u and v. Since shortcuts do not cross, they form a naturally partially ordered set (poset).

When structural changes take place in $\mathcal{H}$, all of the shortcuts that touch the nodes participating in these changes are removed. The cost for removing those shortcuts is amortized over the cost of creating them. However, once the structural changes are complete, we do not immediately return all the shortcuts back. Instead, the data structure partially recovers some of the shortcuts and employs a lazy approach in which shortcuts are only added when they are needed. We feel this method simplifies the description of the data structure.

3.4.4 Approximate Counters. Implementing the sampling operation in Lemma 3.2 reduces to being able to traverse from an $(i,\text{PRIMARY})$-branching node to one of its $(i,\text{PRIMARY})$-children v, where the probability is almost proportional to the number of i-primary endpoints touching v. The distribution over $(i,\text{PRIMARY})$-children of an $(i,\text{PRIMARY})$-branching node is supported by maintaining an approximate i-counter at each $(i,\text{PRIMARY})$-node. Notice that an $\mathcal{H}$-node could be an $(i,\text{PRIMARY})$-node for several i, so there are several i-counters maintained in an $\mathcal{H}$-node. An approximate i-counter at such a node v stores an $(1+o(1))$ approximation of the number of i-primary endpoints touching v. This quality of approximation provides the guarantees needed for the sampling operation. We emphasize that approximate i-counters are only stored for i-primary endpoints, *not* i-secondary endpoints.

Each approximate i-counter uses $O(\log\log n)$ bits, and its precision is relative to the depth and weight of the node. The i-counters are precisely maintained at the $(i,\text{PRIMARY})$-leaves. When the data structure sums i-counters together, the approximate i-counters may lose precision. However, this precision depends on the height of the arithmetic formula tree implicitly formed in the $(i,\text{PRIMARY})$-trees. The following property states the precision requirement in order to support accurate sampling:

Invariant 2. (Precision of Approximate Counters) *Let v be an $\mathcal{H}$-node and let $C_i(v)$ be the number of i-primary endpoints touching v. Let j be the depth of v and let*

$$H(v) = (d_{max} - j)\cdot O(\log\log n) + \lfloor \log(w(v)) \rfloor.$$

If v is an $(i,\text{PRIMARY})$-node then v stores an approximate i-counter $\hat{C}_i(v)$, where

$$\left(1 - (\log^{-2} n)\right)^{H(v)+1} C_i(v) \le \hat{C}_i(v) \le C_i(v).$$

The shortcut infrastructure and local trees together allow us to efficiently guarantee that Invariant 2 holds. This is captured by the following lemma.

Lemma 3.4. *There exists a data structure on $\mathcal{H}$ that maintains approximate i-counters and supports the following operations (the runtime is given in parenthesis):*

- *Update the approximate counters to support a change in the number of $(i,\text{PRIMARY})$-endpoints at a given $\mathcal{H}$-leaf.* $(O(\log n(\log\log n)^2))$
- *Given an $(i,\text{PRIMARY})$-root v^i, update the approximate i-counters for all $(i,\text{PRIMARY})$-nodes in the $(i,\text{PRIMARY})$-tree of v^i so that Invariant 2 holds for those nodes $(O(|\mathcal{T}|(\log\log n)^2+1)$, where $\mathcal{T}$ is the $(i,\text{PRIMARY})$-tree rooted at v^i).*
- *When merging two sibling $\mathcal{H}$-nodes, compute the approximate i-counters for all $i \in [1, d_{max}]$ at the merged node.* $(O(\log\log n))$.
- *When splitting an $\mathcal{H}$-node into two sibling $\mathcal{H}$-nodes, compute the approximate i-counters for all $i \in [1, d_{max}]$ at the two sibling nodes.* $(O(\log\log n))$.

Copyright © by SIAM
Unauthorized reproduction of this article is prohibited

Downloaded 06/06/17 to 69.136.137.32. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

3.4.5 The Local Trees. The local tree is a specially constructed binary tree, where the root is associated with an $\mathcal{H}$-node v and the leaves are the $\mathcal{H}$-children of v. The local trees support the following operations.

LEMMA 3.5. *There exists a data structure that supports the following operations between an $\mathcal{H}$-node v and its $\mathcal{H}$-children.*

- *Add a new $\mathcal{H}$-child x* $\big(O((\log\log n)^2)\big)$.
- *Delete an $\mathcal{H}$-child x* $\big(O((\log\log n)^2)\big)$.
- *Merge two sibling $\mathcal{H}$-nodes u and v* $\big(O((\log\log n)^2)\big)$.
- *Return the $\mathcal{H}$-parent v^{i-1} of $\mathcal{H}$-node v^i* $(O(\log w(v^{i-1}) - \log w(v^i) + \log\log n))$.
- *Enumerate all local tree leaves with an (i,t)-status* $(O(\log\log n)$ *per leaf).*
- *Add (i,t)-status to a local tree leaf* $(O((\log\log n)^2))$.
- *Given an $(i, \text{PRIMARY})$-branching node u^{j-1} and an edge depth i, sample an $(i, \text{PRIMARY})$-child u^j with probability at most*

$$\frac{\hat{C}_i(u^j)}{\hat{C}_i(u^{j-1})}(1-\log^{-2} n))^{-\left[\log(w(u^{j-1}))-\log(w(u^j))+O(\log\log n)\right]}.$$

- *Given an $\mathcal{H}$-node v, test whether there is a unique (i,t)-leaf in the local tree rooted at v. If yes, return that (i,t)-leaf* $(O(\log\log n))$.

3.5 Lookup Tables. There are several components of our data structure that use small lookup tables of size $O(n^\epsilon)$ for a constant $0 < \epsilon < 1$ for supporting fast operations on bit strings. By assuming that the initial graph is empty, the $O(n^\epsilon)$ sized lookup tables are built on-the-fly and their cost is amortized through the operations as follows. As long as the number of graph updates is $m \leq n$, all edge depths are at most $\lfloor \log m \rfloor$. Hence, for each $0 \leq r \leq \log\log n$, after the $m = 2^{2^r}$-th graph update, the data structure rebuilds the lookup tables of size $O(m^\epsilon)$. The time cost for building the lookup tables during the first m operations is bounded by

$$\sum_{i=0}^{\lceil \log\log m \rceil} m^{\frac{1}{2^i}\epsilon} = O(m^\epsilon).$$

This is amortized $o(1)$ per update.

References

[1] D. Eppstein, Z. Galil, G. Italiano, and A. Nissenzweig. Sparsification – a technique for speeding up dynamic graph algorithms. *J. ACM*, 44(5):669–696, 1997.

[2] D. Eppstein, G. F. Italiano, R. Tamassia, R. E. Tarjan, J. Westbrook, and M. Yung. Maintenance of a minimum spanning forest in a dynamic plane graph. *J. Algor.*, 13(1):33–54, 1992.

[3] G. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. Comput.*, 14(4):781–798, 1985.

[4] 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.

[5] M. R. Henzinger and V. King. Randomized dynamic graph algorithms with polylogarithmic time per operation. In *Proceedings of the Twenty-seventh Annual ACM Symposium on Theory of Computing*, STOC '95, pages 519–527, New York, NY, USA, 1995. ACM.

[6] M. R. Henzinger and V. King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *J. ACM*, 46(4):502–516, July 1999.

[7] M. R. Henzinger and M. Thorup. Sampling to provide or to bound: With applications to fully dynamic graph algorithms. *Random Structures & Algorithms*, 11(4):369–379, 1997.

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

[9] B. M. Kapron, V. King, and B. Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *Proceedings of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1131–1142, 2013.

[10] C. Kejlberg-Rasmussen, T. Kopelowitz, S. Pettie, and M. Thorup. Faster worst case deterministic dynamic connectivity. In *Proceedings 24th European Symposium on Algorithms (ESA)*, pages 53:1–53:15, 2016.

[11] M. Pătraşcu and E. Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM J. Comput.*, 35(4):932–963, 2006.

[12] M. Pătraşcu and M. Thorup. Don't rush into a union: take time to find your roots. In *Proceedings of the 43rd ACM Symposium on Theory of Computing (STOC)*, pages 559–568, 2011. Technical report available as arXiv:1102.1783.

Copyright © by SIAM
Unauthorized reproduction of this article is prohibited

[13] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983.

[14] M. 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.

[15] Z. Wang. An improved randomized data structure for dynamic graph connectivity. *CoRR*, abs/1510.04590, 2015.

[16] C. Wulff-Nilsen. Faster deterministic fully-dynamic graph connectivity. In *Proceedings of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1757–1769, 2013.

Downloaded 06/06/17 to 69.136.137.32. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Copyright © by SIAM
Unauthorized reproduction of this article is prohibited