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

Shang-En Huang [a] ✉ ⓘ

Dawei Huang [a] ✉ ⓘ

Tsvi Kopelowitz [b] ✉ ⓘ

Seth Pettie [a] ✉ ⓘ

Mikkel Thorup [c] ✉ ⓘ

a Computer Science and
Engineering, University of
Michigan

b Department of Computer
Science, Bar-Ilan University

c Department of Computer
Science, University of
Copenhagen

ABSTRACT. Dynamic connectivity is one of the most fundamental problems in dynamic
graph algorithms. We present a randomized Las Vegas dynamic connectivity data structure
with $O(\log n(\log\log n)^2)$ amortized expected update time and $O(\log n/\log\log\log n)$ worst case
query time, which comes very close to the cell probe lower bounds of Pătraşcu and Demaine
(2006) and Pătraşcu and Thorup (2011).

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
$G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges, where E is initially empty.

— Insert(u, v): Set $E \leftarrow E \cup \{\{u, v\}\}$.

— Delete(u, v): Set $E \leftarrow E - \{\{u, v\}\}$.

— Conn?(u, v): Return `true` if and only if u and v are in the same connected component in G.

Dynamic connectivity has been studied in numerous models, under both worst case
and amortized notions of time, with deterministic, randomized Las Vegas, and randomized
Monte Carlo guarantees, and with both *public* and *private* witnesses of connectivity. Las Vegas

algorithms always answer Conn? queries correctly but their running time is a random variable. In contrast, the running time of a Monte Carlo algorithm is guaranteed deterministically, but it only answers Conn? queries correctly with high probability. All known dynamic connectivity algorithms maintain a spanning forest F of G as a sparse certificate of connectivity. If F is *public* then the sequence of Insert and Delete operations may depend on F, and may therefore depend on random bits generated earlier by the data structure. When F is *private* the Insert/Delete sequence is selected obliviously.

In this paper we prove near-optimal bounds on the *amortized* complexity of dynamic connectivity in the *Las Vegas* randomzed model, with a *public* connectivity witness.

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. The time bounds hold even if the adversary is aware of the internal state of the data structure. In particular, the data structure maintains a public spanning forest as a connectivity witness.*

1.1   A Brief History of Dynamic Connectivity Data Structures

Worst Case Time. Frederickson [8] developed a dynamic connectivity structure in the strictest model—deterministic worst case time—with $O(\sqrt{m})$ update time and $O(1)$ query time. Eppstein, Galil, Italiano, and Nissenzweig [6] showed that the update times for many dynamic graph algorithms could be made to depend on n rather than m, provided they maintain an $O(n)$-edge witness of the property being maintained, e.g., a spanning forest in the case of dynamic connectivity. Together with [8], Eppstein et al.'s reduction implied an $O(\sqrt{n})$ update time for dynamic connectivity, a bound which stood for many years. Kejlberg, Kopelowitz, Pettie, and Thorup [15] simplified Frederickson's data structure, and improved the update time of [8, 6] to $O\left(\sqrt{\frac{n(\log \log n)^2}{\log n}}\right)$. Recently Chuzhoy, Gao, Li, Nanongkai, Peng, and Saranurak [5] improved the worst case update time to $n^{o(1)}$.

Kapron, King, and Mountjoy [14] gave a Monte Carlo randomized structure with update time $O(c \log^5 n)$ and one-sided error[1] probability n^{-c}. Their data structure maintains a *private* connectivity witness, i.e., it keeps a spanning tree, but the adversary controlling Insert and Delete operations does not have access to the spanning tree. The update time was later improved to $O(c \log^4 n)$ independently by Gibb et al. [9] and Wang [21], and Wang further reduced the time for Insert to $O(c \log^3 n)$. Nanongkai, Saranurak, and Wulff-Nilsen [17] discovered a Las Vegas randomized structure with $n^{o(1)}$ update time that maintains a public connectivity witness. This data structure was recently derandomized [5], leading to a deterministic $n^{o(1)}$ dynamic connectivity algorithm maintaining a public witness.

---

1   An error occurs from reporting that two vertices are disconnected when they are actually connected.

Amortized Time. By allowing amortization in the running time, dynamic connectivity can be solved even faster. Henzinger and King [10] proved that with Las Vegas randomization, dynamic connectivity could be solved with amortized expected $O(\log^3 n)$ update time. This was subsequently improved to amortized expected $O(\log^2 n)$ update time by Henzinger and Thorup [11] using a more sophisticated sampling routine. Holm, de Lichtenberg, and Thorup [12] discovered a *deterministic* data structure with amortized $O(\log^2 n)$ update time. Thorup [20] later improved the *space* of [12] from $O(m + n \log n)$ to optimal $O(m)$, and Wulff-Nilsen [22] further improved [12, 20] to have amortized $O(\log^2 n/\log \log n)$ update time using $O(m)$ space.

At *STOC 2000*, Thorup [20] presented a Las Vegas randomized data structure with amortized expected $O(\log n(\log \log n)^3)$ update time and worst case $O(\log n/\log \log \log n)$ query time. At *SODA 2017*, Huang, Huang, Kopelowitz, and Pettie [13] improved the update time of [20] to $O(\log n(\log \log n)^2)$, and substantiated several claims that were only sketched in [20]. The data structures presented in [20, 13] are especially notable in light of the lower bounds of Pătraşcu and Demaine [1] and Pătraşcu and Thorup [2]. The first shows that any (amortized or randomized) dynamic connectivity structure with $O(t(n) \log n)$ update time, $t(n) = \Omega(1)$, requires $\Omega(\log n/\log t(n))$ query time. In particular, the maximum of update and query time is $\Omega(\log n)$. The second shows that any dynamic connectivity structure with $o(\log n)$ update time requires $n^{1-o(1)}$ query time. Thus, any data structure with $O(\log n(\log \log n)^2)$ update time must have $\Omega(\log n/\log \log \log n)$ query time, and for *any* reasonable query time, we cannot improve our update time by more than a $(\log \log n)^2$ factor. On certain restricted classes of inputs, e.g., trees [19] and planar graphs [7], both updates and queries can be supported in $O(\log n)$ worst case time.

Contribution. This paper should be considered the successor and full version of both the *STOC 2000* and the *SODA 2017* extended abstracts [20, 13], improving the complexity of dynamic connectivity from amortized $O(\log^2 n/\log \log n)$ update time [22] to the near-optimal amortized expected $O(\log n(\log \log n)^2)$ update time.

Organization of the Paper. In Section 2 we review several fundamental concepts of dynamic connectivity algorithms. Section 3 gives a detailed overview of the algorithm, and lists the primitive operations from the data structure that implements the algorithm. In Section 4 we describe the main modules of the data structure. The main modules include: maintaining a binary *hierarchical* representation of the graph, maintaining *shortcuts* for efficient navigation around the hierarchy, and maintaining a system of *approximate counters* to support nearly-uniform random sampling. Each of these modules is explained in detail in Sections 5–9. Finally, we piece up all the modules from the data structures and describe how primitive operations listed in Section 3 (Lemma 3.1) are implemented and analyzed in Section 10. We make some concluding remarks in Section 11.

2. Preliminaries

In this section we review some basic concepts and invariants used in prior dynamic connectivity algorithms [10, 12, 20, 22].

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 data structure on $\mathcal{F}$ (see Theorem 2.1 below). Each edge in the witness forest is called a *witness* edge, and all other edges are called *non-witness* edges. Deleting a non-witness edge does not change the connectivity.

Update Time and Query Time. When describing the dynamic connectivity data structure we only focus on the (amortized) running time of the update operations. Once this time bound is fixed, Theorem 2.1 provides a fast query time, which, according to Pătrașcu and Demaine [1], cannot be unilaterally improved.

THEOREM 2.1 (Henzinger and King [10]). *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.*

PROOF SKETCH. Maintain an Euler tour of each tree in the witness forest and a balanced $t(n)$-ary rooted tree over the Euler tour elements. The height of each rooted tree is $O(\log_{t(n)} n)$. A witness edge insertion/deletion imposes $O(1)$ changes to the Euler tour, which necessitates $O(t(n) \log_{t(n)} n)$ time to update the rooted trees. A query Conn?(u, v) finds the representative copies of u and v in the Euler tours, walks up to their respective roots, and checks if they are equal. ∎

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 edge exists. To speed up the search for replacement edges we maintain Invariant 2.2 (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 2.2.

INVARIANT 2.2 (The Depth Invariant). *Define $G_i = (V, \bigcup_{j \geq i} E_j)$.*

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

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

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 larger than i and discarding edges with depth less than 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*. It follows from Invariant 2.2 that F_i corresponds to a spanning forest of $\hat{G}_i$, if one maps the endpoints of F_i-edges to the contracted vertices 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 2.2 can be restated as $w(u^{i-1}) \le n/2^{i-1}$ since u^{i-1} corresponds to the connected component containing u in G_i.

Endpoints. The *endpoints* of an edge $e = \{u, v\}$ are the pairs $\langle u, e \rangle$ and $\langle v, e \rangle$. At some stage in our algorithm we sample a random endpoint from a set S of endpoints incident to some $V' \subset V$. An edge $\{u, v\}$ with $u, v \in V'$ could contribute zero, one, or two endpoints to S, i.e., the endpoints of an edge are often treated independently. An endpoint $\langle u, e \rangle$ is said to be *touching* the nodes $u^i \in \hat{V}_i$ for all $i \in [1, d_{max}]$.

2.1 Computational Model and Lookup Tables

We assume a standard $O(\log n)$-bit word RAM with the usual repertoire of AC^0 instructions. The data structure uses some non-standard operations on packed sequences of $O(\log \log n)$-bit floating point numbers, which we can simulate by building small lookup tables with size $O(n^\epsilon)$, for some $\epsilon \in (0, 1)$. Since the initial graph is empty, the $O(n^\epsilon)$ sized lookup tables can be built on-the-fly, with their cost amortized through the operations.[2]

---

2   As long as the number of graph updates is $m \le n$, all edge depths are at most $\lfloor \log m \rfloor$. Hence, for each $0 \le r \le \log \log n$, 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 $\sum_{i=0}^{\lceil \log \log m \rceil} m^{\frac{1}{2^i}\epsilon} = O(m^\epsilon)$, which is amortized $o(1)$ per update.

2.2 Miscellaneous

Almost Uniform Sampling. We say that an algorithm samples from a set X $(1+o(1))$-*uniformly at random*, if, for any element $x \in X$, the probability of x being returned is $(1 + o(1))/|X|$. (In our algorithm, the $o(1)$ term is roughly $1/\log n$, and $|X|$ is at most polynomial in n.)

Mergeable Balanced Binary Trees. Some parts of our data structure (see Section 7) use off-the-shelf mergeable balanced binary trees. They should support leaf-insertion and leaf-deletion on T in $O(\log |T|)$ time, and the *merger* of two trees T_1, T_2 in $O(\log |T_1| + \log |T_2|)$ time. The merge operation may create and delete internal nodes as necessary to ensure balance. These trees do *not* store elements from a totally ordered set, and do not need to support a *search* function.

3. Overview of the Algorithm

As in [12, 22], our goal is to restore Invariant 2.2 after each update operation.

In the rest of this section, we provide an overview of the algorithm. The underlined parts of the text refer to primitive data structure operations supported by Lemma 3.1, presented in Section 3.3.

The Data Structure. The hierarchy $\mathcal{H}$ naturally defines a rooted forest (not to be confused with the maximum 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 hierarchy forest. The nodes in $\mathcal{H}$ are the i-components for all $i \in [1, d_{max}]$. The roots of the hierarchy trees are nodes in $\hat{V}_0$, representing 1-components. The set of nodes at depth i in $\mathcal{H}$ is exactly $\hat{V}_i$. The set of children of a node $v^i \in \hat{V}_i$ 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*.

3.1 Insertion

To execute an $\mathtt{Insert}(u, v)$ 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

To execute a $\mathtt{Delete}(u, v)$ operation, where $e = \{u, v\}$, the data structure first removes e from $\mathcal{H}$. Let $i = d_e$. 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. In this case, the deletion algorithm first
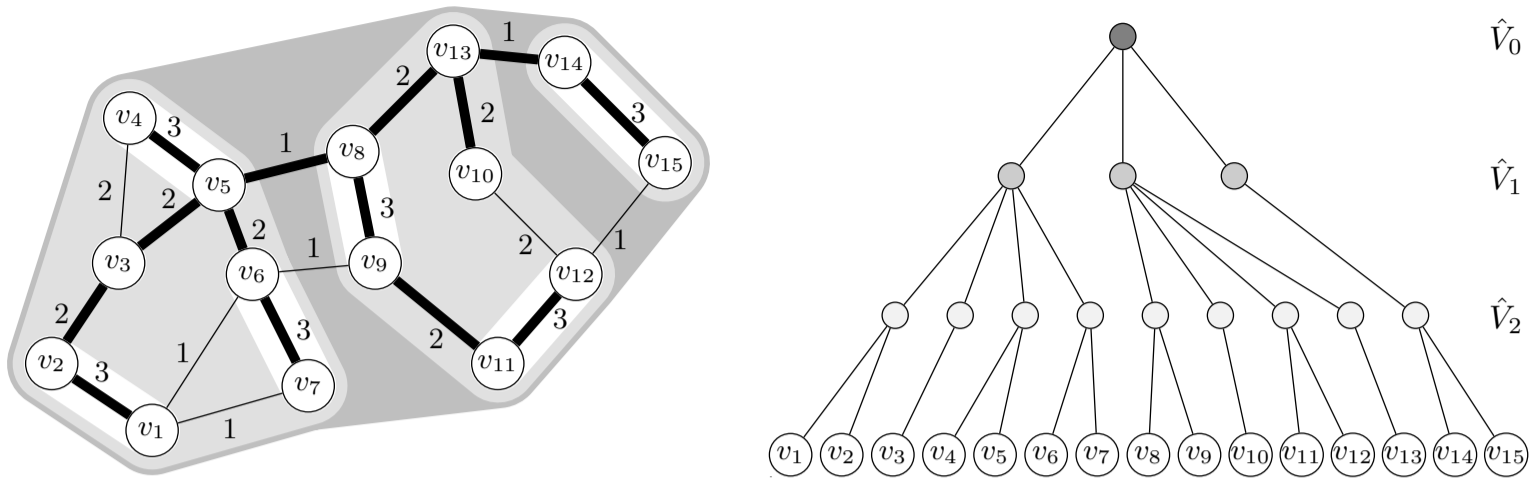
Figure 1. An illustration of a graph and the corresponding hierarchy forest $\mathcal{H}$, where $n = 15$ and $d_{max} = 3$. All thick edges are witness edges and the thin edges are non-witness edges. Components in $\hat{v}_1$ have size at most $\lfloor \frac{15}{2} \rfloor = 7$; those in $\hat{v}_2$ have size at most $\lfloor \frac{15}{4} \rfloor = 3$.

focuses on finding a replacement edge that has depth $i = d_e$. In Section 3.2.3 we extend our algorithm to find a replacement edge of any depth, while preserving the Maximum Spanning Forest Property of Invariant 2.2.

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 $v^i \in c_v$; 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 necessarily 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 graph searches on $\hat{F}_i - \{\{u^i, v^i\}\}$ starting from u^i and v^i. To implement a search, we mark u^i unexplored and insert it into a queue. We repeatedly remove any unexplored $(i + 1)$-component x from the queue, mark it explored, and enumerate all i-witness edges with one endpoint in x. All new $(i + 1)$-components touching these edges are marked unexplored and inserted into the queue. The two searches 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 loss of generality, assume that $w(c_u) \leq w(c_v)$, and so by Invariant 2.2, $w(c_u) \leq w(u^{i-1})/2 \leq n/2^i$.

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 2.2, since $w(c_u) \leq n/2^i$. The merged $(i + 1)$-component has the node u^{i-1} as its parent in $\mathcal{H}$. See Figure 2.b.
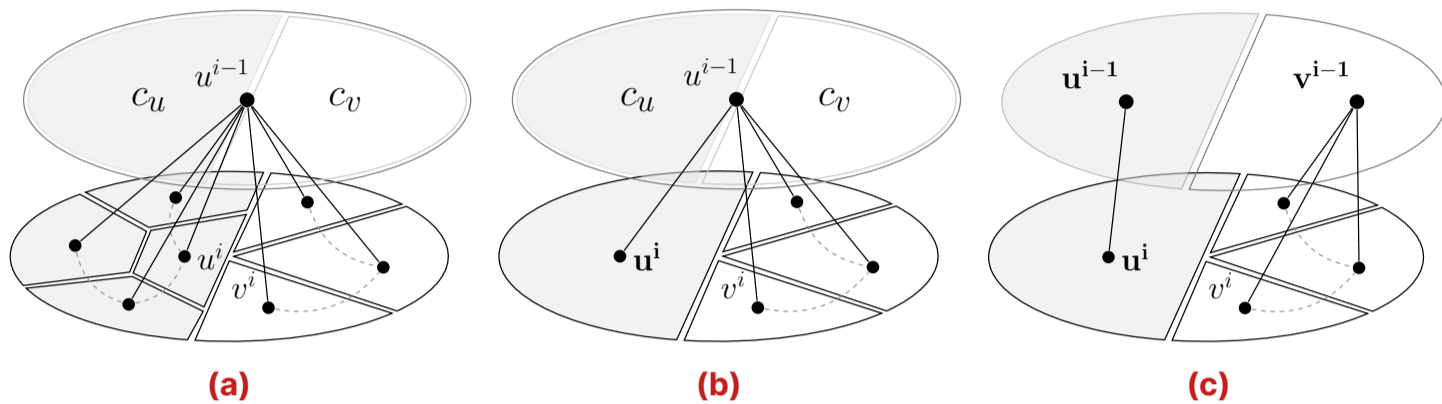

(a)   (b)   (c)

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 u^{i-1} is split.

To differentiate between versions of components before and after the merges, we use bold fonts to refer to components after the merges take place. Thus, the $(i+1)$-component contracted from all $(i + 1)$-components inside c_u is denoted $\mathbf{u^i}$. Similarly, the graph $\hat{G}_i$ after merging some of its nodes is denoted by $\mathbf{\hat{G}_i}$.

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 $\mathbf{\hat{G}_i}$.

3.2.2   Finding a Replacement Edge

Notice that by definition of $\mathbf{\hat{G}_i}$ 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 two auxiliary procedures: the *sampling procedure* and the *enumeration procedure*.

Intuition. Consider the following 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 the algorithm magically *knew* which situation the algorithm is in and could sample i-non-witness endpoints uniformly at random then the problem is straightforward to solve: In Situation A the algorithm iteratively samples an i-non-witness endpoint and tests whether the other endpoint is in $\mathbf{u^i}$. As we will see, each test costs $O(\log n(\log \log n))$ time. The

expected number of samples used to find a replacement edge in this situation is $O(1)$ and so the time cost is charged (in an amortized sense) to the deletion operation. In Situation B the algorithm enumerates and marks every i-non-witness endpoint touching $\mathbf{u^i}$. Any edge with one mark is a replacement edge and any with two marks may be promoted to depth $i + 1$ without violating Invariant 2.2. Since a constant fraction 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 fast (almost-)uniformly random sampling on this set is a very tricky problem. Second, the algorithm does not know in advance whether the current situation is Situation A or Situation B. Notice that it is insufficient to draw $O(1)$ random samples and, if no replacement edge is found, to deduce that the algorithm is in Situation B. Since the cost of enumeration is so high, the algorithm cannot afford to *mistakenly* think that it is in Situation B.

Primary and Secondary Endpoints. The difficulty with supporting random sampling is dynamic updates: when i-non-witness edges are inserted and deleted from the pool due to promotions, the algorithm responds to each such insertion/deletion with updating $\Omega(\log n)$ parts of the data structure that enables fast random sampling. Thus, the cost of updating each part needs to be relatively low in order to obtain the desired time bounds.

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 endpoint $\langle u, e \rangle$ is enumerated (see the enumeration procedure below), the data structure *upgrades* $\langle u, e \rangle$ to an *i-primary endpoint*. The motivation for using two types of endpoints is that the algorithm never samples from the set of i-secondary endpoints, which are only subject to individual insertions, but only the set of i-primary endpoints, which are subject to *bulk* inserts/deletes. The bulk updates to i-primary endpoints are sufficiently large (in an amortized sense) to pay for the changes made to the part of the data structure that supports random sampling.

Notice that each edge undergoes up to d_{max} promotions and up to $2d_{max}$ endpoint upgrades. Since our goal is to obtain an $O(\log n (\log \log n)^2)$ amortized insertion cost, we are able to charge each promotion or upgrade $O((\log \log n)^2)$ units of time.

The Sampling Procedure. This is the only procedure in our algorithm that uses randomness. The sampling procedure can be viewed as a two-stage version of Henzinger and Thorup [11], with some complications due to primary and secondary types. We give a simple 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 belong to replacement edges is small.

Let p be the number of i-primary endpoints touching $\mathbf{u^i}$. The data structure first estimates p up to a constant factor and then invokes the batch sampling test, which $(1 + o(1))$-uniformly samples $O(\log\log p)$ 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 second batch sampling test, which $(1 + o(1))$-uniformly samples $O(\log p)$ 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. (Since otherwise it is hard to obtain good amortized cost.) If more than half of these endpoints belong to replacement edges, then the sampling procedure is terminated and one of the replacement edges is returned. Otherwise, the algorithm 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, then enumerates all i-primary endpoints touching $\mathbf{u^i}$ and establishes for each such edge the number of its endpoints touching $\mathbf{u^i}$ (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 a depth $(i + 1)$-non-witness edge (making both endpoints secondary), without violating Invariant 2.2. As part of the promote and upgrade operations, the algorithm completely rebuilds the part of the data structure supporting random sampling on the i-primary endpoints touching $\mathbf{u^i}$.

Since the enumeration procedure is only invoked when the algorithm concludes that (before the enumeration process) the fraction of the non-replacement edges touching $\mathbf{u^i}$ is at least a constant, the cost of rebuilding the data structure component supporting random sampling is charged to promoting the (sufficiently large number of) non-replacement edges.

3.2.3 Iteration and Conclusion

By the Maximum Spanning Forest Property of Invariant 2.2, the deletion of an edge e can only be replaced by edges of depth d_e or less. The algorithm always first looks for a replacement edge at the same depth as the deleted edge. If the algorithm does not find a replacement edge at depth d_e then the algorithm conceptually *demotes* e by setting $d_e \leftarrow d_e - 1$ in order to preserve the Maximum Spanning Forest Property of Invariant 2.2, and continues looking for a replacement edge at the new depth d_e. The demotion is merely conceptual; the deletion algorithm does not actually update d_e in the course of deleting e.

Implementation. If a depth-i replacement edge e' exists, then u^{i-1} is still an i-component and the algorithm 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, the data structure splits u^{i-1} into two sibling nodes (or two $\mathcal{H}$-roots, if $i = 1$): a new node $\mathbf{u}^{\mathbf{i-1}}$ representing c_u whose only child is $\mathbf{u}^{\mathbf{i}}$, and $\mathbf{v}^{\mathbf{i-1}}$ representing c_v whose children are the rest of the $(i + 1)$-components in c_v. If $i = 1$ then the algorithm is done. Otherwise, the algorithm sets $i \leftarrow i - 1$, conceptually demoting e, and repeats the procedure as if e were deleted at depth $i - 1$.
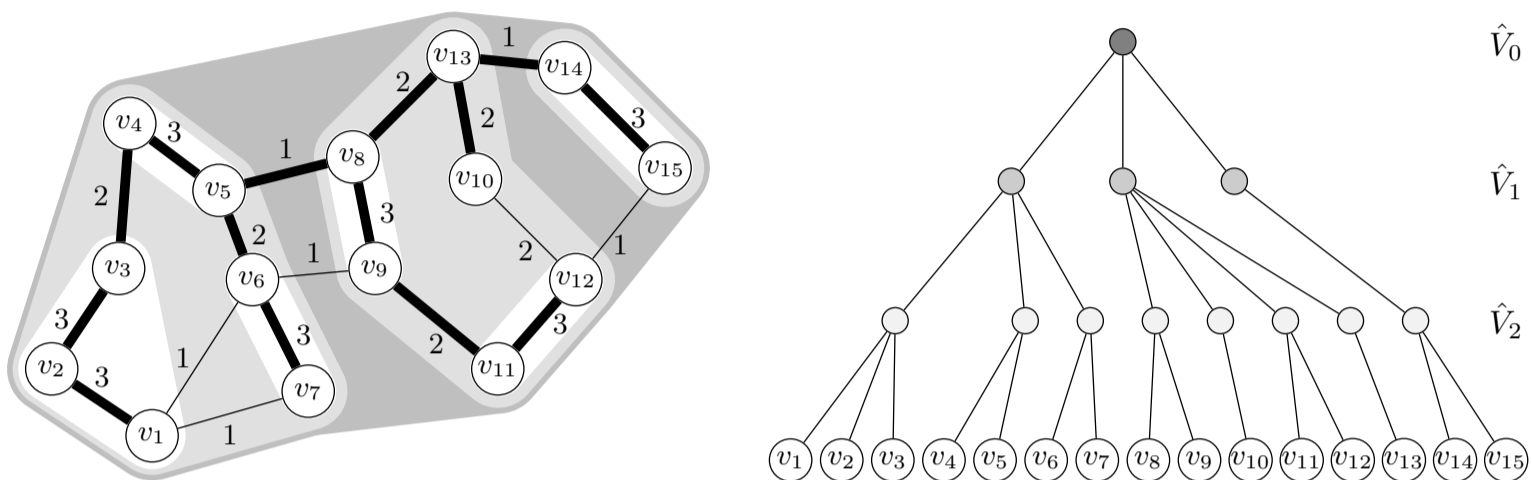
Figure 3. After deletion of $\{v_3, v_5\}$ (See Figure 1.) By identifying $\{v_1, v_2, v_3\}$ to be the smaller weight component, the witness edge $\{v_2, v_3\}$ is promoted to depth 3 and the corresponding nodes in $\hat{V}_2$ are merged. The edge $\{v_3, v_4\}$ is the replacement edge.
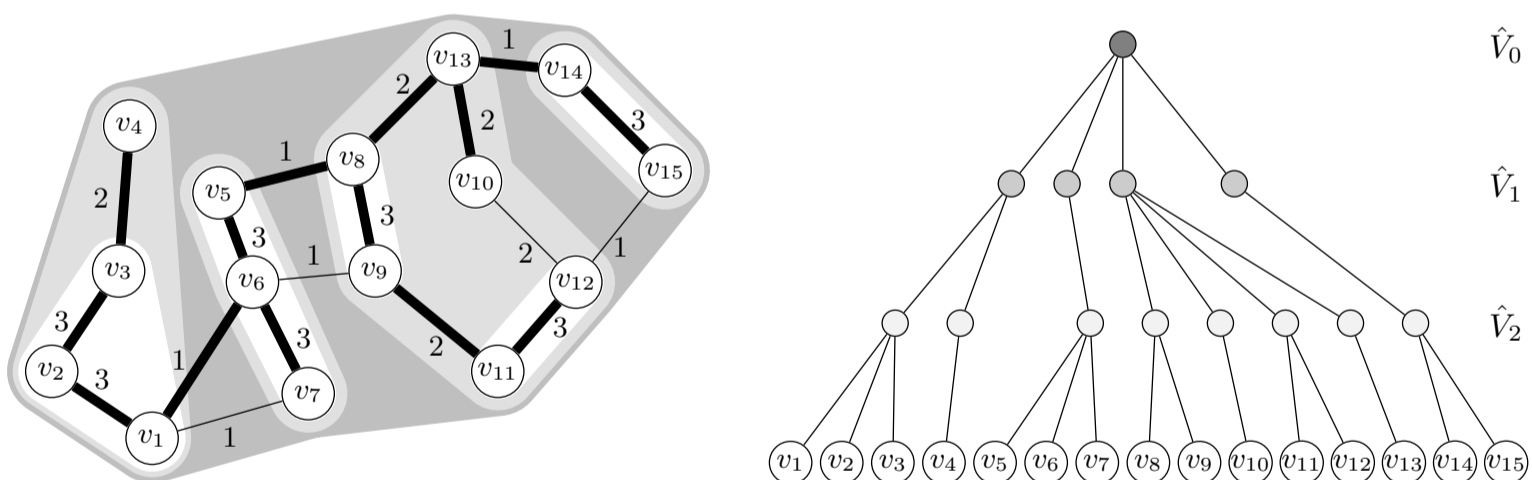
Figure 4. After deletion of $\{v_4, v_5\}$ we do the following steps. (1) Split the node in $\hat{V}_2$ associated with v_4 and v_5. (2) Identify that $\{v_5, v_6, v_7\}$ is the smaller weight component. (3) Promote the edge $\{v_5, v_6\}$ to depth 3, merging nodes v_5^2 and $v_6^2 = v_7^2$. (4) Fail to find a replacement edge at depth 2, and split the node v_5^1. (5) Find a replacement edge $\{v_1, v_6\}$ at depth 1 and designate it a witness edge.

3.3   The Backbone of the Data Structure

Lemma 3.1 summarizes the primitive operations required to execute `Insert` or `Delete`. Recall that the possible depths are integers in $[1, d_{max}]$, and the possible endpoint types are witness (for endpoints of an i-witness edge for some i), primary, and secondary.

LEMMA 3.1. *There exists a data structure that supports the following operations on $\mathcal{H}$ with the following amortized time complexities (given in parentheses).*

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

(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 $\mathbf{u^i}$ to $(i+1)$-witness edges $\left(-\Omega((|S|-1)(\log\log n)^2)\right)$.*

(3) *Given an $\mathcal{H}$-node $v^i \in \hat{V}_i$, upgrade all i-secondary endpoints touching v^i to i-primary endpoints $(-\Omega((s-p)(\log\log n)^2)$, 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 S of i-primary endpoints touching v^i, promote the endpoints in S to $(i+1)$-secondary endpoints $(-\Omega((12|S|-p)(\log\log n)^2)$, where p is the total number of i-primary endpoints touching v^i).*

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

(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 takes u^i as a single $\mathcal{H}$-child and the other takes the rest of u^{i-1}'s former $\mathcal{H}$-children as its $\mathcal{H}$-children $\left(O((\log\log n)^2)\right)$.*

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

(8) *Given v^i, return its $\mathcal{H}$-parent v^{i-1} $\left(O\left(\log\log n + \log\left(\frac{w(v^{i-1})}{w(v^i)}\right)\right)\right)$.*

(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 also touches v^i (see Remark 3.2).*

REMARK 3.2. It should be noted that the time bounds of Lemma 3.1 only apply if the operations are used to correctly maintain Invariant 2.2. For example, if we use Operation (5) to create a new i-witness edge but the set $\mathcal{F}$ (the set of witness edges) now contains a cycle, then all bets are off. Moreover, the worst case cost of the Batch Sampling Test operation is $O(\min(k\log n\log\log n, k + (p+s)\log\log n))$ time, where p and s are the number of i-primary and i-secondary endpoints touching v^i, respectively. We analyze the amortized cost of this operation only when it is used to find replacement edges and maintain Invariant 2.2; see Section 8.1.

REMARK 3.3. The amortized costs of these operations are with respect to a potential function (Section 9). The most important part of the potential function is that every upgrade and promotion releases $O((\log \log n)^2)$ units of potential. Observe that Operations (2,3,4) can have *negative* amortized cost. Negative amortized costs are not contradictory, and they are in fact helpful for paying for the (positive) costs of operations that occur in conjunction with Operations (2,3,4); see Section 10.1.

The proof of Theorem 1.1 uses Lemma 3.1 to restore Invariant 2.2. The proof itself is mostly a technical recapitulation of the algorithm described in Section 3; for the sake of completeness we provide a full proof in Section 10.1.

4.   The Main Modules of the Data Structure

To support Lemma 3.1, the data structure utilizes five main modules, some of which depend on each other: (1) the $\mathcal{H}$-leaf data structure, (2) local trees, (3) the notion of an induced (i, t)-forest, (4) shortcut infrastructure, and (5) approximate counters. The $\mathcal{H}$-leaf data structure is fairly straightforward and is described in detail in Section 4.1. We define induced (i, t)-forests in Section 4.3. A brief overview of the other modules is described in Sections 4.2, 4.4, and 4.5. Sections 5–9 provide a detailed explanation of each module. The general operations involving multiple modules, as well as the proof of Lemma 3.1 are described and analyzed in detail in Section 10.

4.1   The $\mathcal{H}$-Leaf Data Structure

The $\mathcal{H}$-leaf data structure supports several operations that act on an individual vertex. Let v be a vertex (an $\mathcal{H}$-leaf), $i \in [1, d_{max}]$ be a depth, and $t \in \{\text{witness}, \text{primary}, \text{secondary}\}$ be an endpoint type. The $\mathcal{H}$-leaf data structure supports insertion or deletion of an endpoint (of an edge incident to v) with depth i and type t. Moreover, the $\mathcal{H}$-leaf data structure supports enumeration of all endpoints incident to v with depth i and type t, and selecting one such endpoint uniformly at random.

Supporting these operations in $O(1)$ amortized time (plus time linear in the output) is straightforward. Simply pack the endpoints with depth and type (i, t) in a dynamic array. Dynamic arrays can be implemented deterministically to support incrementing/decrementing the length of the array in $O(1)$ amortized time.

4.2   The Local Trees

A *local tree* is a specially constructed binary tree, whose root is associated with an $\mathcal{H}$-node v and whose leaves are associated with the $\mathcal{H}$-children of v. By composing the local trees with $\mathcal{H}$, we can view the result as a single binary tree of height at most $O(\log n \log \log n)$. The purpose

of this binarization is to provide an efficient infrastructure for supporting navigation within $\mathcal{H}$. The local tree operations are detailed in Section 7 and summarized in Lemma 7.8.

4.3   The Induced (i, t)-Forest

The purpose of the (i, t)-forests is to support efficient enumeration of all endpoints of a given type that touch a given $\mathcal{H}$-node. For a given edge depth $i \in [1, d_{max}]$ and endpoint type $t \in \{\text{witness, primary, secondary}\}$, an $\mathcal{H}$-leaf v is an (i, t)-*leaf* if v has an incident 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 either

— v is an (i, t)-leaf,

— v is an (i, t)-root,

— v has more than one child in $\mathfrak{F}$ (so v is called an (i, t)-*branching node*), or

— v is an $\mathcal{H}$-child of an (i, t)-branching node but v has only one $\mathcal{H}$-child in $\mathfrak{F}$. In this case we call v a *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 in $\mathfrak{F}$ 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. An $\mathcal{H}$-node v has (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 $3d_{max} = O(\log n)$ each. The first indicates for each (i, t) pair whether v is an (i, t)-node, and if so, the second indicates whether v is an (i, t)-branching node or not.

Operations on (i, t)-forests. A conceptual edge between an (i, t)-node and its (i, t)-parent or (i, t)-child need not be maintained explicitly. The two components of our data structure that simulate these edges are the *shortcut infrastructure* and the *local trees*. In particular, the shortcut infrastructure supports efficient traversal from a single-child (i, t)-node to its unique (i, t)-child, while the local trees support efficient enumeration of all the (i, t)-children of an (i, t)-branching node. Since the implementation of traversal and navigation operations on (i, t)-forests utilizes local trees which are introduced and defined in Section 7, we defer the discussion of (i, t)-forests and their detailed implementation to Section 8.2 (see Lemma 8.1).

4.4   The Shortcut Infrastructure

The purpose of shortcuts is to facilitate a faster 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 shortcuts are described in Section 5.

4.5   Approximate Counters

Implementing the sampling operation in Lemma 3.1 reduces to being able to traverse from an $(i, \text{primary})$-branching node to one of its $(i, \text{primary})$-children v, where the choice of an $(i, \text{primary})$-child is random with probability that is approximately proportional to the number of i-primary endpoints touching v. The implementation of the random choice 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 could be several approximate i-counters maintained in an $\mathcal{H}$-node. The advantages of using approximate i-counters, as opposed to precise counters, are two-fold. First, each approximate i-counter uses only $O(\log \log n)$ bits, and so $O(\log n / \log \log n)$ approximate i-counters can be packed into a single machine word and be collectively manipulated in $O(1)$ time. Second, approximate counters can only take on $(\log n)^{O(1)}$ values, and hence a decrement-only counter can only generate $(\log n)^{O(1)}$ total work throughout its lifetime. The maintenance of approximate i-counters and the sampling algorithm are explained in Section 6 and Section 7.4.

5.   Shortcut Infrastructure

As described in Section 4.4, the purpose of shortcuts is to allow for efficient navigation between a single-child (i, t)-node u and its only (i, t)-child v. If the graph is static, a direct pointer between u and v could be stored in the data structure so that v can be directly accessed from u. The challenge is to maintain useful shortcuts in the midst of structural updates to $\mathcal{H}$.

$\mathcal{H}$-shortcuts. An $\mathcal{H}$-shortcut $u \leftrightharpoons v$ is a data structure connecting an ancestor u to a descendant v in $\mathcal{H}$. $\mathcal{H}$-shortcuts are only stored between a subset of eligible pairs of ancestor-descendant pairs. The eligible pairs are determined as follows. For a positive integer ℓ, define its *least significant bit index*, denoted by $\text{LSBIndex}(\ell)$, to be the minimum integer b such that 2^b divides ℓ but 2^{b+1} does not. For an $\mathcal{H}$-node u, let $\text{depth}_{\mathcal{H}}(u)$ be the distance from u to the root of the tree in $\mathcal{H}$ that contains u. The *power* of a pair of nodes u and v is defined as

$$\mathcal{P}(u, v) = \min(\text{LSBIndex}(\text{depth}_{\mathcal{H}}(u) + 1), \text{LSBIndex}(\text{depth}_{\mathcal{H}}(v) + 1)).^{[3]}$$

---

[3]   The "+1" is included because `LSBIndex` is not well defined at zero.

In order for an $\mathcal{H}$-*shortcut* to exist between u and v, any intermediate node x on the path from u to v must have $\mathrm{LSBIndex}(\mathrm{depth}_{\mathcal{H}}(x) + 1) < \mathcal{P}(u, v)$. If v is the $\mathcal{H}$-child of u, then $\mathcal{P}(u, v) = 0$ and $u \leftrightharpoons v$ is an eligible $\mathcal{H}$-shortcut, which is called a *fundamental $\mathcal{H}$-shortcut*.

The following lemma states that the set of $\mathcal{H}$-shortcuts on an ancestor-descendant path do not cross each other.

LEMMA 5.1. *For any four distinct $\mathcal{H}$-nodes x_1, x_2, x_3, x_4 along a root-to-leaf path in $\mathcal{H}$, it is impossible to have two $\mathcal{H}$-shortcuts $x_1 \leftrightharpoons x_3$ and $x_2 \leftrightharpoons x_4$.*

PROOF. For $j \in \{1, 2, 3, 4\}$ let $h_j = \mathrm{depth}_{\mathcal{H}}(x_j) + 1$, so $h_1 < h_2 < h_3 < h_4$. Assume the claim is false, and so there exist two $\mathcal{H}$-shortcuts $x_1 \leftrightharpoons x_3$ and $x_2 \leftrightharpoons x_4$. By definition this implies $\mathrm{LSBIndex}(h_2) < \mathrm{LSBIndex}(h_3)$ and $\mathrm{LSBIndex}(h_3) < \mathrm{LSBIndex}(h_2)$, a contradiction.   ∎

The covering relationships of $\mathcal{H}$-shortcuts and the poset. We say that $a \leftrightharpoons b$ *covers* $c \leftrightharpoons d$ if c and d are on the path P_{ab} from a to b in $\mathcal{H}$. Notice that a shortcut covers itself. Define $\geq$ to be the covering partial order:

$$(a \leftrightharpoons b) \geq (c \leftrightharpoons d) \text{ iff } a \leftrightharpoons b \text{ covers } c \leftrightharpoons d.$$

For any uv-path P_{uv} on $\mathcal{H}$, the *maximal covering set* of P_{uv}, denoted by $\mathrm{COVER}^{\mathcal{H}}(u, v)$, is the set of maximal $\mathcal{H}$-shortcuts (with respect to $\geq$) among all $\mathcal{H}$-shortcuts having both endpoints on P_{uv}. Figure 5 illustrates $\mathrm{COVER}^{\mathcal{H}}(v_5, v_{14})$ in bold.
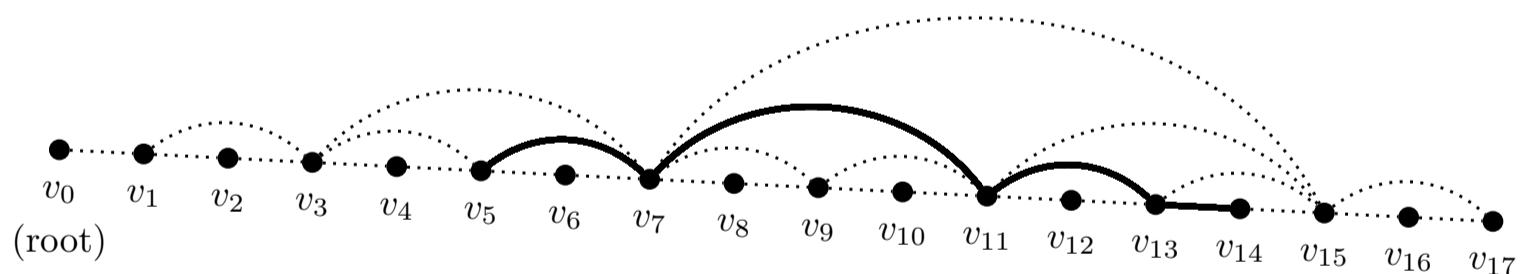
Figure 5. The figure above shows $\mathrm{Cover}^{\mathcal{H}}(v_5, v_{14})$ as an example, where v_i has $\mathrm{depth}_{\mathcal{H}}(v_i) = i$. The dotted edges are the set of all possible shortcuts.

The following lemma bounds the size of $\mathrm{COVER}^{\mathcal{H}}(u, v)$.

LEMMA 5.2. *For any two nodes $u, v \in \mathcal{H}$ with u an ancestor of v, all $\mathcal{H}$-shortcuts in $\mathrm{COVER}^{\mathcal{H}}(u, v)$ form a path connecting u and v, and $|\mathrm{COVER}^{\mathcal{H}}(u, v)| = O(\log \log n)$.*

PROOF. All $\mathcal{H}$-shortcuts on P_{uv} form a poset, and all fundamental $\mathcal{H}$-shortcuts on P_{uv} form the path between u and v. By Lemma 5.1, $\mathrm{COVER}^{\mathcal{H}}(u, v)$ forms a path connecting u and v.

The $\mathcal{H}$-shortcuts in $\text{COVER}^{\mathcal{H}}(u, v)$ can be partitioned into two sequences: one with strictly increasing powers and one with strictly decreasing powers. To see this, notice that for any sequence of consecutive integers, there is a unique largest LSBIndex value among the sequence. For any $\mathcal{H}$-node let $q(x) = \text{LSBIndex}(\text{depth}_{\mathcal{H}}(x) + 1)$. Let v^* be the unique $\mathcal{H}$-node on P_{uv} such that $q(v^*) > q(x)$ for all $x \in P_{uv} \backslash \{v^*\}$. It is straightforward to see that no $\mathcal{H}$-shortcut on P_{uv} crosses v^* and hence $\text{COVER}^{\mathcal{H}}(u, v) = \text{COVER}^{\mathcal{H}}(u, v^*) \cup \text{COVER}^{\mathcal{H}}(v^*, v)$.

Now we claim the following: let $P_{v'v}$ be an ancestor-descendant path such that $q(v') > q(x)$ for all $x \in P_{v'v} \backslash \{v'\}$. Then $\text{COVER}^{\mathcal{H}}(v', v)$ consists of $\mathcal{H}$-shortcuts with strictly decreasing powers. We prove this claim by induction. In the base cases the claim is trivially true, when $v' = v$ or v' is the $\mathcal{H}$-parent of v. In general, let v'' be the unique node on the path $P_{v'v}$ such that $q(v'') > q(x)$ for all $x \in P_{v'v} \backslash \{v', v''\}$. The shortcut $v' \leftrightharpoons v''$ must be in $\text{COVER}^{\mathcal{H}}(v', v)$ since the power of $v' \leftrightharpoons v''$ is strictly greater than the power of any shortcut on $P_{v''v}$. By the induction hypothesis on $P_{v''v}$, the claim holds. Thus, all $\mathcal{H}$-shortcuts in $\text{COVER}^{\mathcal{H}}(v^*, v)$ have distinct and decreasing powers. By symmetry, all $\mathcal{H}$-shortcuts in $\text{COVER}^{\mathcal{H}}(u, v^*)$ also have distinct and increasing powers. Since the maximum depth is $d_{max} = \lfloor \log n \rfloor$, the largest possible power of an $\mathcal{H}$-shortcut is $\lceil \log \log n \rceil - 1$. As a consequence, we have $|\text{COVER}^{\mathcal{H}}(u, v)| = O(\log \log n)$. ∎

(i, t)-shortcuts. Let u be a single-child (i, t)-node and let v be the (i, t)-child of u, which by definition must be either an (i, t)-branching node or an (i, t)-leaf. The purpose of maintaining $\mathcal{H}$-shortcuts is to allow one to quickly move from u to v. Ideally, the data structure will traverse the $O(\log \log n)$ $\mathcal{H}$-shortcuts in $\text{COVER}^{\mathcal{H}}(u, v)$. However, forcing all of the $\mathcal{H}$-shortcuts in $\text{COVER}^{\mathcal{H}}(u, v)$ to be maintained by the data structure seems to complicate the process of updating $\mathcal{H}$-shortcuts as $\mathcal{H}$ changes. In particular, when an i-witness edge $\{u, v\}$ is deleted, $\mathcal{H}$ goes through several structural changes by merging an ancestor u^i (or v^i) with a subset of the $\mathcal{H}$-siblings of u^i.[4] All $\mathcal{H}$-shortcuts that were connected to u^i (or v^i) and those $\mathcal{H}$-siblings need to be updated at the same time. Since we are fine with $O(\log \log n)$ *amortized* time for the traversal, the process of updating shortcuts (due to changes in the hierarchy or the corresponding (i, t)-forests) becomes simpler by allowing a weaker invariant governing which shortcuts are actually present.

INVARIANT 5.3 $((i, t)$-**Shortcuts**). *Let u be a single-child (i, t)-node and let v be the (i, t)-child of u. The (i, t)-shortcuts on P_{uv} that are stored by the data structure form a path connecting u and v.*

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

---

4    $\mathcal{H}$-*siblings* are $\mathcal{H}$-nodes sharing the same $\mathcal{H}$-parent.

immediately return all the shortcuts back. Instead, the data structure partially recovers enough[5] shortcuts to maintain Invariant 5.3, and then employs a lazy approach in which shortcuts are only added (via a *covering* process) when they are needed.
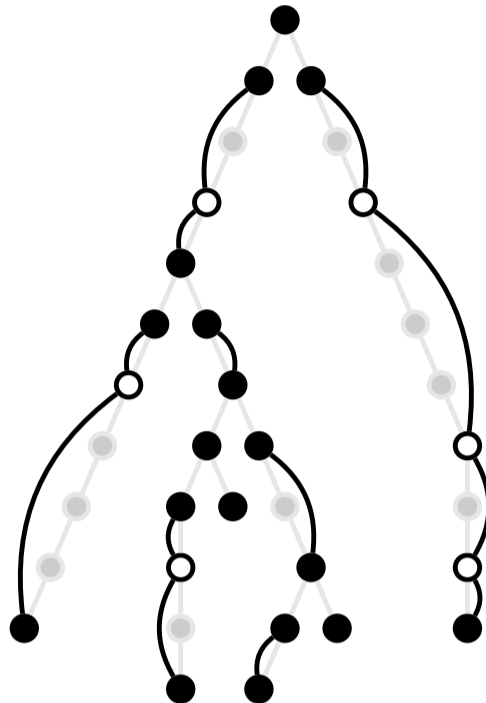

Figure 6. An example of an (i, t)-tree and its corresponding (i, t)-shortcuts: filled circles are (i, t)-nodes, and the curved line segments are (i, t)-shortcuts.

Covering and uncovering. Assume Invariant 5.3 holds. Suppose that the algorithm traverses downward from a single-child (i, t)-node u to its (i, t)-child v. If the set of shortcuts used is precisely $\text{COVER}^{\mathcal{H}}(u, v)$ then this traversal costs $O(\log \log n)$ time. If not, then the algorithm repeatedly *covers* consecutive (i, t)-shortcuts (see Section 5.1 for implementation details) until the set of (i, t)-shortcuts between u and v is exactly $\text{COVER}^{\mathcal{H}}(u, v)$. We use a potential argument to prove that the amortized cost of traversing from u to v is $O(\log \log n)$ time; see Section 9.2.

There are also certain cases where the structure of $\mathcal{H}$ does not change, but some (i, t)-forests do change (for example, whenever an $\mathcal{H}$-leaf gains or loses an (i, t)-status). To support structural changes in $\mathcal{H}$ or in (i, t)-forests, the data structure will at times *uncover* an (i, t)-shortcut s of power p by removing s and adding the two consecutive (i, t)-shortcuts of power $p - 1$ that were covered by s. In order to accommodate an efficient uncovering operation, during a covering operation the data structure continues to store the covered $\mathcal{H}$-shortcuts so that they are readily available when a subsequent *uncover* operation occurs. The $\mathcal{H}$-shortcuts stored by the data structure that are strictly covered by some (i, t)-shortcuts are called *supporting $\mathcal{H}$-shortcuts*; these supporting shortcuts *do not* have (i, t)-status. The $\mathcal{H}$-shortcut $u \leftrightarrows v$ is always

---

5   Notice that Invariant 5.3 implies that the shortcut data structure is not required to store all of $\text{Cover}^{\mathcal{H}}(u, v)$ in order for the invariant to hold.

directly accessible from v (the deeper node), but not necessarily from u (the shallower node). From the perspective of v, $u \leftrightharpoons v$ is called an *upward $\mathcal{H}$-shortcut*, while from the perspective of u, $u \leftrightharpoons v$ is called a *downward $\mathcal{H}$-shortcut*.

An upper bound on the number of $\mathcal{H}$-shortcuts that need to be stored at each $\mathcal{H}$-node is captured by the following straightforward corollary. (Recall that the algorithm <u>does not</u> store shortcuts between an (i,t)-branching node and its (i,t)-children.)

COROLLARY 5.4. *Assume Invariant 5.3 holds for all pairs of nodes in $\mathcal{H}$. Then for each node $v \in \mathcal{H}$, and each (i,t) pair, there is at most one downward (i,t)-shortcut and at most one upward (i,t)-shortcut at v.*

Sharing shortcuts. An $\mathcal{H}$-shortcut $u \leftrightharpoons v$ that is an (i,t)-shortcut could also be an (i',t')-shortcut when $(i,t) \neq (i',t')$. Similarly, a supporting shortcut for some (i,t)-shortcut could also be an (i',t')-shortcut when $(i,t) \neq (i',t')$. The data structure stores at most one copy of any $\mathcal{H}$-shortcut even if there are many (i,t) pairs that use it. The maximum number of distinct $\mathcal{H}$-shortcuts touching a given ancestor-descendant path is bounded by the following lemma. (Recall that a *stored shortcut* is either an (i,t)-shortcut for some (i,t), or a supporting shortcut, which may have no (i,t)-status.)

LEMMA 5.5. *Consider any node v in $\mathcal{H}$. The total number of stored shortcuts joining an ancestor of v to another node is $O(\log n \log \log n)$. In particular, the number of distinct fundamental (i,t)-shortcuts having one endpoint at an ancestor of v is $O(\log n)$. Moreover, the number of $\mathcal{H}$-shortcuts having both endpoints at ancestors of v is $O(\log n)$.*

PROOF. For a given path P, an $\mathcal{H}$-shortcut $u \leftrightharpoons v$ is said to be *deviating* if exactly one of its endpoints is on P.

Let P be the path from $v \in \mathcal{H}$ to its $\mathcal{H}$-root. For each edge depth i and type t, at most one (i,t)-shortcut is deviating from P, and each such shortcut has at most $O(\log \log n)$ supporting shortcuts with exactly one endpoint on P. (Recall that (i,t)-shortcuts form paths from *single-child (i,t)-nodes* to their (i,t)-child. Branching (i,t)-nodes have no (i,t)-shortcuts leading to descendants.) Thus, for each (i,t) pair, at most one *fundamental (i,t)-shortcut* deviates from P. All $\mathcal{H}$-shortcuts connecting $\mathcal{H}$-nodes on P form a laminar set, and so there are at most $2d_{max} = O(\log n)$ such $\mathcal{H}$-shortcuts. Thus, the total number of stored shortcuts with one endpoint in P is $O(\log n \log \log n)$, and the total number of distinct fundamental (i,t)-shortcuts with one endpoint on P is $O(\log n)$. ∎

In the rest of this section, we describe how $\mathcal{H}$-shortcuts are stored.

5.1 The $\mathcal{H}$-shortcut data structure

Information stored at $\mathcal{H}$-nodes. Due to Corollary 5.4, every node in $\mathcal{H}$ has at most $3d_{max}+1 = O(\log n)$ downward (i, t)-shortcuts at any given time. Each node u stores an array DOWN_u of size at most $3d_{max} + 1$ storing all downward (i, t)-shortcuts, together with a bitmap OCC_u indicating which array slots of DOWN_u are in use.[6] The size of DOWN_u is chosen to be exactly enough for storing pointers to (i, t)-shortcuts for all possible (i, t) pairs as well as one additional slot for temporary use during promotions/upgrades. However, a single shortcut may be shared by many (i, t) pairs. In order to support fast access from u to its downward (i, t)-shortcut, each node stores a *local dictionary* which is an array DOWNIDX_u storing, for each (i, t) pair, a $(\log\log n + 2)$-bit index to the location in DOWN_u of the appropriate downward $\mathcal{H}$-shortcut, i.e.,

$$\text{DOWN}_u[\text{DOWNIDX}_u[i, t]] \text{ points to an } (i, t)\text{-shortcut leaving } u, \text{ if such a shortcut exists.}$$

Notice that for an $\mathcal{H}$-node and a power p, there is at most one upward $\mathcal{H}$-shortcut from v with power p. Thus, each node v maintains an array UP_v of $O(\log\log n)$ pointers to shortcuts, sorted by power, to the upward supporting $\mathcal{H}$-shortcuts of v. Moreover, at each node v the data structure stores a $(3d_{max} + 1)$-length array UPIDX_v of $O(\log\log\log n)$-bit integers for each (i, t) pair. Thus, the upward (i, t)-shortcut $x \leftrightarrows v$ is accessed in $O(1)$ time, i.e.,

$$\text{UP}_v[\text{UPIDX}_v[i, t]] \text{ points to an } (i, t)\text{-shortcut entering } v, \text{ if such a shortcut exists.}$$

Notice that each entry in the DOWNIDX_u and UPIDX_v arrays is represented with $O(\log\log n)$ bits, and there are $O(\log n)$ (i, t) pairs. These entries are packed into $O(\log\log n)$ memory words so that the data structure is able to update the entire array efficiently via lookup tables in $O(\log\log n)$ time.

The following lemma summarizes how shortcuts are used to support various operations needed locally in one $\mathcal{H}$-node.

LEMMA 5.6. *The following operations are supported via shortcut information stored at nodes (worst case time in parentheses).*

— *Given $u \leftrightarrows v$ and a bitmap b of length $3d_{max} + 1$, add $u \leftrightarrows v$ as an (i, t)-shortcut for all (i, t) pairs indicated by b ($O(\min\{|b| + 1, \log\log n\})$ where $|b|$ is the number of 1s in b).*

— *Given $u \leftrightarrows v$ and a bitmap b of length $3d_{max} + 1$, remove the (i, t)-shortcut status from $u \leftrightarrows v$ for all (i, t) pairs indicated by b ($O(\min\{|b| + 1, \log\log n\})$).*

— *Given $u \in \mathcal{H}$ and an (i, t) pair, return the (i, t)-downward $\mathcal{H}$-shortcut at u or report that such a shortcut does not exist ($O(1)$).*

— *Given $v \in \mathcal{H}$ and an (i, t) pair, return the (i, t)-upward $\mathcal{H}$-shortcut at v or report that such a shortcut does not exist ($O(1)$).*

---

6   Notice that when the data structure allocates the array Down_u, it is assumed to contain arbitrary values. One can only tell which values are meaningful and how to interpret them via the Occ_u and DownIdx_u arrays.

— *Given $u \in \mathcal{H}$, return the index of an empty slot in DOWN_u ($O(1)$).*

— *Given $u \in \mathcal{H}$, enumerate all indices of used locations in DOWN_u ($O(k + 1)$ where k is the number of the enumerated indices).*

PROOF SKETCH. The proof of the lemma is straightforward using bitwise operations or $O(n^\epsilon)$-size lookup tables for operations on DOWN_u, DOWNIDX_u, OCC_u, UPIDX_v, and UP_v. For example, OCC_u is a $(3 \log n + 1)$-bit vector. We partition it into $3\epsilon^{-1}$ segments of $\epsilon \log n$ bits, and can search for a zero in each segment in $O(1)$ time with a table lookup. ∎

Information stored at shortcuts. An $\mathcal{H}$-shortcut $u \leftrightharpoons v$ is an $O(1)$-word data structure storing the following information:

— $\mathcal{P}(u, v)$: the *power* of the shortcut,

— Pointers to u and v,

— The index j in DOWN_u where $u \leftrightharpoons v$ is stored, or $\perp$ if $u \leftrightharpoons v$ is not stored in DOWN_u,

— A $3d_{max} + 1$ length bitmap $b_{u \leftrightharpoons v}$ containing one bit for each (i, t) pair (called the (i, t)-bit) indicating whether $u \leftrightharpoons v$ is an (i, t)-shortcut, and

— If $\mathcal{P}(u, v) > 0$ then $u \leftrightharpoons v$ stores pointers to the two supporting shortcuts with power $\mathcal{P}(u, v) - 1$ that $u \leftrightharpoons v$ covers.

LEMMA 5.7. *The $\mathcal{H}$-shortcut data structure supports the following operations (worst case time in parenthesis):*

1. *(Uncovering) Given an (i, t) pair and an (i, t)-shortcut $u \leftrightharpoons v$ that is not a fundamental $\mathcal{H}$-shortcut, uncover $u \leftrightharpoons v$ and convert the two supporting shortcuts of $u \leftrightharpoons v$ into (i, t)-shortcuts ($O(1)$).*

2. *(Traversal and Covering) Assume Invariant 5.3 holds for all $\mathcal{H}$-nodes with depth $\geq i$. Given a single-child (i, t)-node u whose (i, t)-child is v, traverse from u to v via (i, t)-shortcuts while guaranteeing that after the traversal is completed, the set of (i, t)-shortcuts between u and v is exactly $\text{COVER}^{\mathcal{H}}(u, v)$, preserving Invariant 5.3 for all $\mathcal{H}$-nodes with depth $\geq i$ ($O(k + \log \log n)$, where k is the number of (i, t)-shortcuts covered during the traversal).*

PROOF. Part 1. Suppose the algorithm uncovers a given (i, t)-shortcut $u \leftrightharpoons v$ that is not fundamental, meaning $u \leftrightharpoons v$ has power $p > 0$. The algorithm sets $b_{u \leftrightharpoons v}[i, t] = 0$, follows the two pointers from $u \leftrightharpoons v$ to its supporting power-$(p - 1)$ $\mathcal{H}$-shortcuts $u \leftrightharpoons x$ and $x \leftrightharpoons v$, and sets $b_{u \leftrightharpoons x}[i, t] = b_{x \leftrightharpoons v}[i, t] = 1$. The algorithm also updates in a straightforward manner some local information in all affected nodes $\{u, v, x\}$ in $O(1)$ time. To be specific, the algorithm (i) checks whether $u \leftrightharpoons x$ and $x \leftrightharpoons v$ are already stored in DOWN_u and DOWN_x by inspecting $u \leftrightharpoons x$ and $x \leftrightharpoons v$. (ii) If not, the algorithm finds empty slots in DOWN_u and/or DOWN_x via the bitmaps OCC_u and OCC_x, which indicate which slots in DOWN_u and DOWN_x are available, and updates $\text{DOWNIDX}_u[i, t]$ and/or $\text{DOWNIDX}_x[i, t]$. (iii) The algorithm sets $\text{DOWN}_u[\text{DOWNIDX}_u[i, t]] = u \leftrightharpoons x$;

$\text{DOWN}_x[\text{DOWNIDX}_x[i, t]] = x \leftrightharpoons v$; $\text{UPIDX}_x[i, t] = \mathcal{P}(u, x)$; and $\text{UPIDX}_v[i, t] = \mathcal{P}(x, v)$. (iv) If $b_{u \leftrightharpoons v}$ is all 0, i.e., $u \leftrightharpoons v$ is no longer an (i', t')-shortcut for any (i', t') pair, the algorithm frees the slot storing $u \leftrightharpoons v$ in DOWN_u by unsetting the corresponding bit in OCC_u, then updates $u \leftrightharpoons v$ to reflect that $u \leftrightharpoons v$ is no longer stored in DOWN_u.

REMARK 5.8. After step (iv), it may be that $b_{u \leftrightharpoons v} = 0$ *and* $u \leftrightharpoons v$ is not a supporting shortcut for any higher-power (i', t')-shortcut. If this is the case, it is fine to delete $u \leftrightharpoons v$ (and update UP_v and UPIDX_v appropriately). In our implementation the algorithm has no means to check whether $u \leftrightharpoons v$ is a necessary supporting shortcut, and so the algorithm keeps $u \leftrightharpoons v$ allocated. Notice that in the worst case there are $\Theta(n \log \log n)$ stored shortcuts, so keeping spurious shortcuts around does not affect the overall space usage of the data structure.

Continuing with the proof (Part 2), we can move from u to its (i, t)-child by starting at u and following downward (i, t)-shortcuts until an (i, t)-node is reached. During this traversal, if there are two consecutive (i, t)-shortcuts $x \leftrightharpoons y'$ and $y' \leftrightharpoons y$ with the same power p and

$$\text{LSBIndex}(\text{depth}_{\mathcal{H}}(y') + 1) < \min\Big(\text{LSBIndex}(\text{depth}_{\mathcal{H}}(x) + 1), \ \text{LSBIndex}(\text{depth}_{\mathcal{H}}(y) + 1)\Big),$$

then the data structure covers the two shortcuts with the $\mathcal{H}$-shortcut $x \leftrightharpoons y$ having power $p + 1$. This is done as follows.

First the algorithm checks whether $x \leftrightharpoons y$ already exists, by testing if $\text{UP}_y[\mathcal{P}(x, y)]$ stores a pointer to $x \leftrightharpoons y$ or not. If $x \leftrightharpoons y$ already exists then $x \leftrightharpoons y$ is accessed through UP_y, and if not then $x \leftrightharpoons y$ is created and a pointer to $x \leftrightharpoons y$ is added to UP_y.

Next, the algorithm sets the (i, t)-bit in $b_{x \leftrightharpoons y}$ to 1 and sets the (i, t)-bits in $b_{x \leftrightharpoons y'}$ and $b_{y' \leftrightharpoons y}$ to 0. If $b_{x \leftrightharpoons y'} = 0$ (resp. $b_{y' \leftrightharpoons y} = 0$), the algorithm removes its index from DOWN_x (resp. $\text{DOWN}_{y'}$) by unsetting the corresponding bits in OCC_x (resp. $\text{OCC}_{y'}$). The algorithm also updates DOWN_x and DOWNIDX_x so that $\text{DOWN}_x[\text{DOWNIDX}_x[i, t]] = x \leftrightharpoons y$ is accessible from x.

Covering $x \leftrightharpoons y$ may create the opportunity to cover another shortcut $x' \leftrightharpoons y$ of the next higher power. The data structure uses UP_x to access the upwards shortcut $x' \leftrightharpoons x$ with power $p + 1$. If $x' \leftrightharpoons x$ exists and is also an (i, t)-shortcut then the data structure covers $x' \leftrightharpoons x$ and $x \leftrightharpoons y$ with $x' \leftrightharpoons y$, and recursively looks to see if there are more shortcuts to cover at power $p + 2$, and so on.

It is straightforward to verify that at the end of the traversal the set of (i, t)-shortcuts connecting u and v is exactly $\text{COVER}^{\mathcal{H}}(u, v)$. The time for traversing the path is $O(k + |\text{COVER}^{\mathcal{H}}(u, v)|)$, which is $O(k + \log \log n)$ where k is the number of (i, t)-shortcuts being covered during the traversal. ∎

In Section 9.2 we show that by defining the potential function to be the number of all (i, t)-shortcuts that *could* be covered but are not yet covered, this operation has amortized cost $O(\log \log n)$ time.

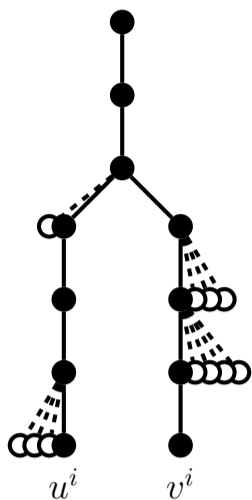## 5.2   Maintaining Invariant 5.3 Through Structural Changes to $\mathcal{H}$

Figure 7. After deleting an i-witness edge $\{u, v\}$, all affected $\mathcal{H}$-nodes are on at most two paths. In the course of looking for a replacement for $\{u, v\}$, we will merge a collection of siblings into one at each depth between i and the depth where the replacement edge is found. The dashed lines illustrate the effect of merging siblings.

The shortcut infrastructure is very sensitive to the merge operation (e.g., Operation (2) in Lemma 3.1). In particular, when an i-witness edge $\{u, v\}$ is deleted, $\mathcal{H}$ goes through several structural changes by merging an ancestor of u^i (or v^i) with a subset of its $\mathcal{H}$-siblings. These merges require updating the shortcut infrastructure, which seems to be a very complicated task when supporting these types of changes. Specifically, we need to employ a special strategy that ensures Invariant 5.3 holds after the entire `Delete` operation.

In order to provide an efficient implementation, observe that during such a single deletion, all merged $\mathcal{H}$-nodes (and their appropriate $\mathcal{H}$-siblings) end up being on the paths between u^i and v^i and their respective $\mathcal{H}$-roots. See Figure 7. Thus, we are able to employ the following strategy.

First, at the beginning of the delete operation, the algorithm completely uncovers and removes *all* $\mathcal{H}$-shortcuts that touch $\mathcal{H}$-nodes on the two paths. In particular, by Lemma 5.5, the algorithm removes (1) $O(\log n)$ fundamental shortcuts, (2) $O(\log n)$ shortcuts with both endpoints on the path, and (3) $O(\log \log n)$ deviating shortcuts from each path for each (i, t) pair. Recall that deviating shortcuts have one endpoint on the $\mathcal{H}$-path in question.

After removing these $\mathcal{H}$-shortcuts, Invariant 5.3 no longer holds for pairs of $\mathcal{H}$-nodes where at least one node is on the affected paths. However, these are shortcuts with (i', t')-status for some $i' < i$, and so during the deletion operation at depth i we never use such shortcuts. Hence, removing them does not affect the other operations that take place during the edge deletion process at depth i.

Lemma 5.9 summarizes the operations that remove and restore shortcuts along paths in $\mathcal{H}$, which are used to guarantee that Invariant 5.3 holds after the deletion operation terminates. Since the implementation requires interaction with the local trees, we defer its proof to Section 9.1.

LEMMA 5.9. *The data structure supports the following operations on $\mathcal{H}$ with amortized time cost (in parenthesis). Given an $\mathcal{H}$-node v:*

— *Uncover and remove every $\mathcal{H}$-shortcut that is touching any node that is an ancestor of v ($O(\log n (\log \log n)^2)$).*

— *Given v, its $\mathcal{H}$-parent u, and a bitmap b, add a fundamental $\mathcal{H}$-shortcut $u \leftrightharpoons v$ for all (i, t) pairs indicated by b ($O(\log \log n)$).*

— *Add all fundamental $\mathcal{H}$-shortcuts between consecutive ancestors of v that are (i, t)-shortcuts for at least one (i, t) pair ($O(\log n \log \log n)$).*

— *Assume Invariant 5.3 holds. For all (i, t) pairs, cover all (i, t)-shortcuts having both endpoints at ancestors of v ($O(\log n \log \log n)$).*

6. Implementation of Approximate Counters

In this section, we describe how approximate i-counters are implemented. Without loss of generality we assume that the input graph G is simple. Hence, all approximate i-counters are only required to represent a $(1 + o(1))$-approximation of integers in the range $[0, n^2]$.

6.1 Approximate Counters

Each $(i, \text{primary})$-leaf ℓ maintains the exact number of $(i, \text{primary})$-endpoints touching ℓ. The precise number of $(i, \text{primary})$-endpoints in a subtree of any $(i, \text{primary})$-node v could be computed *exactly* using a formula tree defined by the induced $(i, \text{primary})$-tree rooted at v where the value at each vertex is the sum of the values of its children. (Because the local trees are binary, the induced tree is also binary, and has height $O(\log n \log \log n)$.) If one were to use such a strategy, then every $\mathcal{H}$-node has the potential of storing $O(\log n)$ counters, where each counter uses $O(\log n)$ bits, for a total of $O(\log n)$ words. Thus, splitting and merging vertices may cost $\Theta(\log n)$ time each, which is too expensive for our purposes.

Instead, the data structure efficiently maintains *approximate i-counters* for nodes in $\mathcal{H}$ with a multiplicative approximation guarantee of $(1 + o(1))$ using only $O(\log \log n)$ bits per approximate i-counter.

The structure of an approximate counter. Let $\beta = 2$ be a parameter that controls the quality of the approximation. Each approximate counter $\hat{C}$ is defined by a pair (m, e) composed of a *mantissa* $m \in \{0, 1\}^{\beta \log \log n}$ and an *exponent* $e \in \{0, 1\}^{\log \log n + 1}$. The *floating point representation*

of $\hat{C}$ concatenates m and e into a length $(\beta + 1) \log \log n + 1$ bit string. The *integer representation* of $\hat{C}$ is $m2^e$, where we treat the mantissa part and the exponent part as unsigned integers. Notice that an approximate counter represents up to $2(\log n)^{\beta+1}$ different integers. From the definition above, an integer $C \in [0, n^2]$ is approximated by $\hat{C} = (m, e)$ where m is the first $\beta \log \log n$ bits of the binary representation of C and e is the number of truncated bits.

Special addition operation. When computing the addition of two values a and b represented by two approximate counters, the result $a + b$ is rounded *down* to the nearest possible approximate counter value. Notice that this kind of addition is not associative. We denote the operation of adding two approximate counters by $a \boxplus b$. The precision guarantee of $\boxplus$ is summarized in the following lemma.

LEMMA 6.1. *Let a and b be two approximate counter values represented by approximate counters. Then $a \boxplus b$ satisfies:*

$$(1 - \log^{-\beta} n)(a + b) \le a \boxplus b \le a + b.$$

PROOF. Let $C = a + b$. Then by definition $\hat{C} = (m, e)$ keeps the first $\beta \log \log n$ bits of the binary representation of C. The difference between C and $\hat{C}$ is therefore strictly less than $(\log^{-\beta} n)C$. Thus, $\hat{C} \ge (1 - \log^{-\beta} n)C$. ∎

Approximation guarantee and the formula tree. Using approximate counters with the $\boxplus$ operation instead of exact counters creates a loss in precision which depends on the height of the arithmetic formula tree. Recall that the height of a formula tree is always bounded by $O(\log n \log \log n)$ where the $\log \log n$ factor is due to the local trees. In order to bound the loss of precision we use a function $H(v)$ which expresses the *maximum possible* height of v in any formula tree. See Section 7.4 for more on why $H(\cdot)$ is defined this way.

DEFINITION 6.2. Let v be an $\mathcal{H}$-node. Let j be the depth of v in $\mathcal{H}$. Then

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

Notice that $H(v) = O(\log n \log \log n)$. The following invariant relates the precision of approximate counters to the function H. The maintenance of Invariant 6.3 is addressed in Lemma 6.5, which is proved in Section 8.3.

INVARIANT 6.3 (Precision of Approximate Counters). *Let v be an $\mathcal{H}$-node and let $C_i(v)$ be the precise number of i-primary endpoints touching v. If v is an $(i, \text{primary})$-node then v stores an approximate i-counter $\hat{C}_i(v)$, where*

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

Thus, if Invariant 6.3 holds with $\beta = 2$, then for any $\mathcal{H}$-node v,

$$\hat{C}_i(v) \geq \left(1 - (\log^{-2} n)\right)^{H(v)} C_i(v) = \left(1 - (\log^{-2} n)\right)^{O(\log n \log \log n)} C_i(v) = (1 - o(1))C_i(v),$$

and so $\hat{C}_i(v)$ gives the desired approximation.

Packing $O(\log n)$ Approximate Counters. Each node in $\mathcal{H}$ stores $d_{max} = \log n$ approximate counters. These counters are stored in $O(\log \log n)$ words by packing $O(\log n / \log \log n)$ approximate counters in the floating pointer representation into each word. With the aid of lookup tables of size $O(n^\epsilon)$, the following lemma is straightforward.

LEMMA 6.4. *The following operations are supported on approximate counters (worst case time in parentheses):*

— *Given an $\mathcal{H}$-node v and a depth i, update/return the approximate i-counter stored at v ($O(1)$).*
— *Given the floating point representation of an approximate counter, return its integer representation ($O(1)$).*
— *Given the integer representation of an approximate counter, return its floating point representation ($O(1)$).*
— *Given two approximate counters a and b, return $a \boxplus b$ ($O(1)$).*
— *Given two arrays of $O(\log n)$ approximate counters packed into $O(\log \log n)$ words, return their coordinate-wise sum, packed into $O(\log \log n)$ words ($O(\log \log n)$).*

PROOF SKETCH. The first four operations use bitwise operations in a straightforward manner. The fifth operation uses $O(n^\epsilon)$-size lookup tables to support a query in $O(\log \log n)$ time; see Section 2.1. ∎

Summary of operations. The main lemma summarizing operations related to approximate i-counters is given next.

LEMMA 6.5. *There exists a data structure that maintains approximate i-counters on $\mathcal{H}$ while maintaining Invariant 6.3 and supporting the following operations with the following amortized time complexities (in parentheses):*

— *Update the approximate counters to reflect 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 $(i, \text{primary})$-tree $\mathcal{T}$ rooted at v^i, rebuild approximate i-counters for all $(i, \text{primary})$-nodes in $\mathcal{T}$ to restore Invariant 6.3 for those nodes $(O(|\mathcal{T}|(\log \log n)^2))$.*
— *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 $(O(\log \log n))$.*

The proof of Lemma 6.5 depends on the implementation of local trees, which we provide in Section 7. Thus, the proof of Lemma 6.5 is deferred to Section 8.3.

7.   Local Trees

The purpose of the local tree $\mathcal{L}(v)$ is to connect an $\mathcal{H}$-node v with its $\mathcal{H}$-children while supporting various operations. A local tree is composed of a three-layer binary tree and a special binary tree called the *buffer tree*. The three-layer binary tree is composed of a *top layer*, a *middle layer* and a *bottom layer*. See Figure 8 for an illustration.
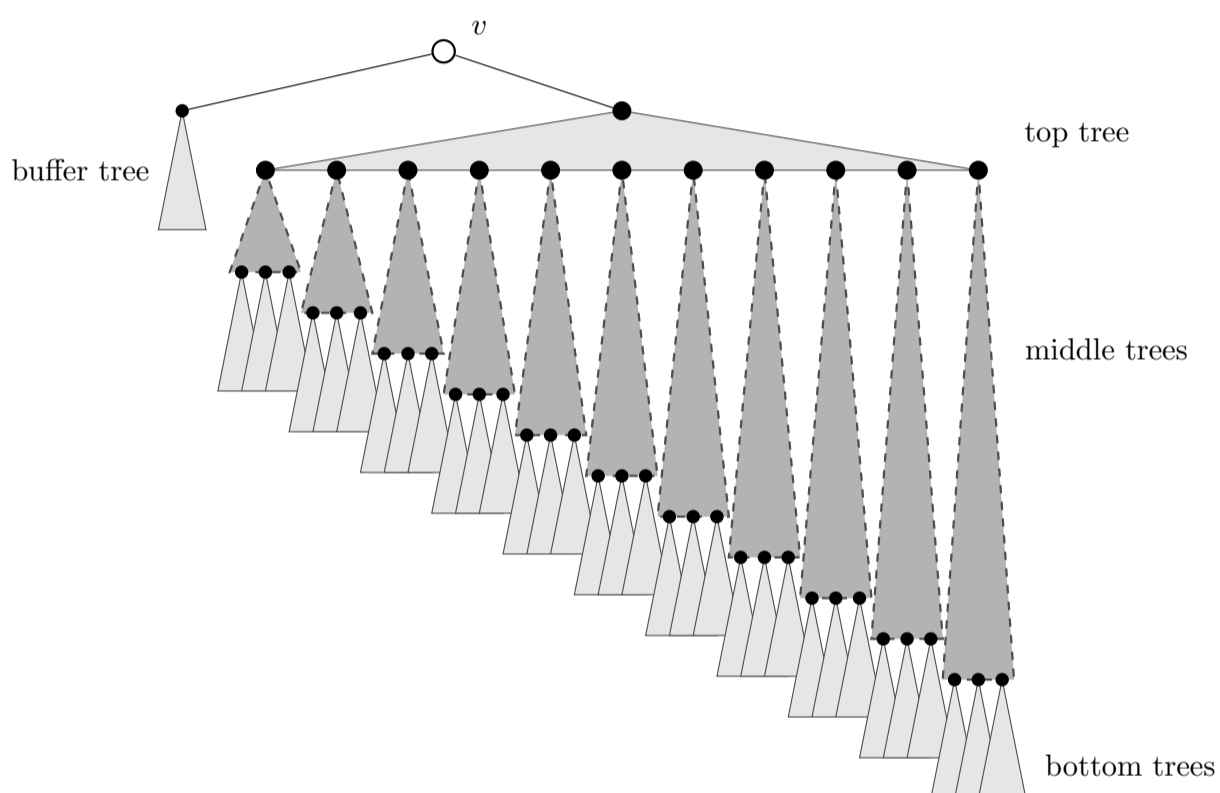

Figure 8. An example to a local tree $\mathcal{L}(v)$ associated with v.

— The bottom layer is composed of *bottom trees*, each having at most $2 \log^\alpha n$ leaves and height $O(\log \log n)$, $\alpha = O(1)$ to be calculated later.
— The middle layer is composed of *middle trees* such that all bottom tree roots are middle tree leaves. The *weight* of a node x in $\mathcal{L}(v)$, denoted by $w(x)$, is defined to be the sum of all weights of $\mathcal{H}$-children of v in the subtree of x, and the *rank* of x is defined to be $\text{rank}(x) = \lfloor \log w(x) \rfloor$. The weights are explicitly maintained only for nodes in either bottom or buffer trees. The middle trees are weight balanced binary trees with respect to $w(\cdot)$. The algorithm maintains the invariant that there are never more than $O(\log n)$ middle tree roots in a local tree.

— The top tree[7] is a mergeable, $O(\log \log n)$-height tree whose leaves are middle tree roots. Its purpose is merely to gather up all middle trees within a single tree, while increasing the overall height of the local tree by only $O(\log \log n)$.

Local tree roots and local tree leaves. The root of $\mathcal{L}(v)$ has two children: the root of the buffer tree and the root of the top tree. The root of $\mathcal{L}(v)$ also has a pointer pointing to v in $\mathcal{H}$. When a new $\mathcal{H}$-node $v \in \mathcal{H}$ is created, $\mathcal{L}(v)$ is initially empty.

$\mathcal{H}$-node representatives. Each $\mathcal{H}$-child x of v is not in $\mathcal{L}(v)$ as such, but is present through a *representative* ℓ_x, which is a leaf in $\mathcal{L}(v)$. We distinguish x from ℓ_x because they have different characteristics and store different information.

The local tree leaf ℓ_x stores a pointer to $x \in \mathcal{H}$, the weight of x, a parent pointer, approximate counters, and a bitmap maintaining <u>local</u> (i, t)-*status* of the leaf ℓ_x, where the (i, t)-bit in the bitmap is set to 1 if and only if x and v are both (i, t)-nodes but the fundamental $\mathcal{H}$-shortcut $v \leftrightarrows x$ is *not* an (i, t)-shortcut. In a quiescent state, this only occurs when v is an (i, t)-branching node or (i, t)-root and x is an (i, t)-node.[8] However, in the middle of a Delete operation we may temporarily uncover and remove a fundamental (i, t)-shortcut $v \leftrightarrows x$, which can cause v, x to temporarily become (i, t)-nodes and $\ell_x \in \mathcal{L}(v)$ to temporarily become a *local* (i, t)-node.

Local (i, t)-trees. Consider a local tree $\mathcal{L}(v)$. The *local* (i, t)-*nodes* comprise all leaves of $\mathcal{L}(v)$ with local (i, t)-status, as well as those internal nodes $z \in \mathcal{L}(v)$ satisfying at least one of the following.

— z is the root of $\mathcal{L}(v)$, having at least one leaf-descendant with local (i, t)-status.
— $z \in \mathcal{L}(v)$ is a bottom tree node, a buffer tree node, or a top tree node having a descendant with local (i, t)-status. (Because of their dual membership, middle tree roots and leaves are also included in this category.)
— $z \in \mathcal{L}(v)$ is a middle tree node whose children both have descendants with (i, t)-status. (It is a *local* (i, t)-*branching* node.)
— $z \in \mathcal{L}(v)$ is a child of a middle tree (i, t)-branching node.

A *local* (i, t)-*tree* is defined in a similar fashion as the (i, t)-forest on $\mathcal{H}$; each $z \in \mathcal{L}(v)$ maintains a bitmap indicating for which (i, t)-pairs it is a local (i, t)-node. Whereas the (i, t)-forest can have arbitrary branching factor, every local (i, t)-tree is binary since $\mathcal{L}(v)$ is itself binary. Navigating from a local (i, t)-node z to its child is straightforward when z is in a bottom, buffer, or top tree, since the (i, t)-bits are stored explicitly at every node in these trees, and these trees are binary. *Local* (i, t)-shortcuts are used for faster navigation in the middle layer; these

---

7   Not to be confused with the *top tree* dynamic tree data structure of Alstrup, Holm, Lichtenberg, and Thorup [**4**].

8   Notice that if x is an (i, t)-branching node but v is not, then $x \in \mathcal{H}$ has (i, t)-status but $\ell_x \in \mathcal{L}(v)$ does not have *local* (i, t)-status. This is the main reason for notationally distinguishing x from ℓ_x.

are defined in Section 7.2. Each local $(i, \text{primary})$-tree node in $\mathcal{L}(v)$ maintains an approximate i-counter.

In Sections 7.1–7.3 we describe the operations of the bottom/buffer layer, the middle layer, and the top layer in isolation. In particular, Lemmas 7.1, 7.5, and 7.6 state the *worst case* cost of operations, without regard to side effects on other layers. The interaction between the layers and the amortization of costs is addressed in Section 7.5, Lemma 7.8.

7.1   Bottom Trees and the Buffer Tree

The algorithm attaches new $\mathcal{H}$-node representatives only to the buffer tree, while deletions of $\mathcal{H}$-node representatives can take place in both buffer and bottom trees. The buffer tree can be regarded as a bottom tree under construction.

Each buffer tree and bottom tree has at most $2 \log^\alpha n$ local tree leaves, where α is a constant to be determined in Section 7.8. Whenever the buffer tree size exceeds $\log^\alpha n$, either from merging two $\mathcal{H}$-nodes or from inserting a new local tree leaf, the buffer tree becomes mature and is converted to a bottom tree. The data structure adds this bottom tree into the bottom layer and creates a new empty buffer.

The buffer and bottom trees are $O(\log \log n)$ height mergeable binary trees. Each node stores a weight, a vector of approximate counters, pointers to its parent and children, and a bitmap indicating for each (i, t) pair, whether there is a local tree leaf in its subtree with local (i, t)-status.

LEMMA 7.1. *The buffer tree and bottom trees support the following operations, with the following worst case time complexities (in parentheses):*
— *Detach a buffer/bottom tree leaf $(O((\log \log n)^2))$.*
— *Remove local (i, t)-status from a buffer/bottom tree leaf $(O(\log \log n))$.*
— *Given an edge depth $i \in [1, d_{max}]$, a buffer/bottom leaf x, and a value q, decrease the approximate i-counter at x to q $(O(\log \log n))$.*

In addition, the buffer tree supports the following operations:
— *Attach a buffer tree leaf $(O((\log \log n)^2))$.*
— *Merge two buffer trees of two sibling $\mathcal{H}$-nodes $(O((\log \log n)^2))$.*
— *Add local (i, t)-status to a buffer tree leaf $(O(\log \log n))$.*
— *Convert the buffer tree to a bottom tree $(O(1))$.*
— *Given an edge depth $i \in [1, d_{max}]$, a buffer leaf x, and a value q, set the approximate i-counter at v to be q $(O(\log \log n))$.*

PROOF. A buffer tree is implemented by an off-the-shelf mergeable binary tree with $O(\log \log n)$ worst case time for each attach, detach, and merge operation.[9] However, in order

to support updates to the vector of approximate counters, an $O(\log \log n)$ factor overhead is applied to each of the operations. See Lemma 6.4. Hence the worst case time cost for each operation is $O((\log \log n)^2)$. From these three operations, bottom trees are only subject to *detach*. Since we only require the height of a bottom tree to be $O(\log \log n)$, no rebalancing is necessary after detaching a leaf. In order to obtain correct rank(x), each attach, detach, and merge also updates the weight of the given buffer/bottom tree root.

To add (i, t)-status to a buffer tree leaf x, the data structure traverses up the buffer tree and sets the (i, t)-bit to 1 in all ancestors of x in the buffer tree. To remove (i, t)-status from a buffer/bottom leaf x, the data structure updates the (i, t)-bits at each ancestor of x. If a leaf x has local $(i, \text{primary})$ status, it carries an approximate i-counter. Such counters can be increased or decreased in $O(\log \log n)$ time by updating all ancestors of x in its buffer/bottom tree. ∎

REMARK 7.2. Observe that *only* the buffer tree can acquire new leaves, and only buffer tree nodes can *gain* local (i, t)-status and *increase* their approximate i-counters. In particular, this implies that when a bottom tree leaf has to acquire a local (i, t)-status, the algorithm removes the leaf from the bottom tree, updates its status and re-inserts the leaf into the buffer tree.

7.2 Middle Trees

All bottom tree roots are middle tree leaves. Middle trees respond to three types of updates at their leaves: a leaf losing (i, t)-status, decreasing its approximate i-counter, or decreasing its weight. Middle trees are maintained as weight-balanced binary trees satisfying Invariant 7.3.

INVARIANT 7.3. *If x is a middle tree leaf/bottom tree root it maintains $w(x)$ and* rank$(x) =$ *$\lfloor \log w(x) \rfloor$. If x is an internal middle tree node it maintains only* rank(x), *and if x has children x_L, x_R then* rank$(x_L) =$ rank$(x_R) =$ rank$(x) - 1$.

The operations described in Lemma 7.5 specifically maintain Invariant 7.3. As a consequence of Invariant 7.3, the path from any middle tree leaf x_B (bottom tree root) to the corresponding middle tree root x_M has length $\log(\frac{w(x_M)}{w(x_B)}) + O(1)$. This property is used to bound the number of local tree nodes traversed when walking from any $\mathcal{H}$-node to its $\mathcal{H}$-parent v via the local tree $\mathcal{L}(v)$. In accordance with Invariant 7.3, two middle tree roots with the same rank may *join*, and become children of a new middle node parent.

Local Shortcuts. Each of the middle trees maintains a *local shortcut infrastructure* in much the same way that shortcuts are maintained in $\mathcal{H}$. Let u and v be two nodes in the same middle tree such that u is a proper ancestor of v. Then $u \leftrightharpoons v$ is an eligible local shortcut if and only if

---

9   Note that all such off-the-shelf data structures are, in fact, binary *search* trees, but we do not impose any total order on the leaves, nor do we require any operation analogous to binary *search*.

for every internal node x on the path P_{uv},

$$\text{LSBIndex}(\text{rank}(x) + 1) < \min \Big(\text{LSBIndex}(\text{rank}(u) + 1), \ \text{LSBIndex}(\text{rank}(v) + 1)\Big).$$

Notice that the $\mathcal{H}$-shortcuts are defined from the depths of $\mathcal{H}$-nodes which increase along the path from an $\mathcal{H}$-root to an $\mathcal{H}$-leaf. In contrast, in middle trees the ranks of middle tree nodes decrease on the path from a middle tree root to a middle tree leaf. The definition of *power* is symmetric between u and v, so the increasing/decreasing direction here does not matter. Local shortcuts have the same properties as $\mathcal{H}$-shortcuts: they are non-crossing; all eligible local shortcuts naturally form a poset $\succeq$, and the maximal elements (w.r.t. $\succeq$) among shortcuts on a middle tree path P_{uv} form a path $\text{Cover}(u, v)$ with length $O(\log \log n)$. A local shortcut with power 0 is called a *trivial shortcut*, which coincides with a middle tree edge from a parent to one of its children.

Invariant 7.4 is a local tree analogue of Invariant 5.3.

INVARIANT 7.4. *Let u be a single-child local (i, t)-node and let v be the local (i, t)-child of u. Then the local (i, t)-shortcuts on P_{uv} that are stored by the data structure form a path connecting u and v.*

Lemma 7.5 lists the middle tree operations.

LEMMA 7.5. *The data structure supports the following operations on a collection of middle trees, maintaining Invariants 7.3 and 7.4, with the following worst case time complexities (in parentheses):*

— *Reduce the weight of a middle tree leaf* ($O(\log n \log \log n)$).
— *Remove (i, t)-status from a middle tree leaf* ($O(\log n)$).
— *Given an edge depth $i \in [1, d_{max}]$ and a middle tree leaf x_B, update the approximate i-counter at x_B* ($O(\log n)$).
— *Given a newly created bottom tree root, create a new middle tree leaf* ($O(1)$).
— *Join two middle trees with the same rank* ($O(\log \log n)$).

PROOF. We address each operation in turn.

Reducing ranks. When the weight of a middle tree leaf x_B is reduced (because its bottom tree suffered enough leaf deletions) it may cause a discrete reduction in its rank, which violates Invariant 7.3. If so, we destroy all middle tree nodes that are strict ancestors of x_B. We first uncover all local shortcuts touching the path from x_B to its middle tree root x_M. This procedure is the same as the uncovering procedure described in Section 9.1. In order to avoid redundancy, we do not provide details here. This costs $O(\log n \log \log n)$ time, and increases the number of middle trees in the collection. (Each new middle tree root is inserted into the top tree.)

Removing (i, t)-status. Similar to the (i, t)-forests, in the local (i, t)-tree the middle tree edges between a local (i, t)-branching node x and its (i, t)-children are *not* considered to be trivial (i, t)-shortcuts. To remove (i, t)-status from a bottom tree root/middle tree leaf x_B, we follow local upward (i, t)-shortcuts to find the one-child (i, t)-node ancestor x' of x_B. If $x' = x_M$ is the middle tree root of x_B then we remove (i, t)-status from x_M (triggering an update to the top tree; see Lemma 7.6). Otherwise, the parent of x', x'' is an (i, t)-branching node. We remove (i, t)-status from x' and all shortcuts from x_B to x', then add a trivial (i, t)-shortcut from x'' to the sibling of x'. This may cause x'' and/or the sibling of x' to lose (i, t)-status. Since the middle trees are weight balanced, removing (i, t)-status from a middle tree leaf costs worst case $O(\log n)$ time.

Update an approximate i-counter. If the approximate i-counter at x_B changes it invalidates the approximate i-counters at all ancestors on the path from x_B to its middle tree root x_M. Each can be updated in $O(1)$ time (Lemma 6.4), for a total of $O(\log n)$ time.

Create a new middle tree leaf. The buffer tree root maintains its weight and approximate i-counters. Thus, when the buffer is converted to a bottom tree, its root (the new middle tree leaf) can be inserted into the middle tree collection in $O(1)$ time. (As a new middle tree root, it is also inserted as a leaf in the top tree; this is accounted for in Lemma 7.6.)

Joining middle trees. To join roots x_L, x_R, we create a new middle tree parent x and compute its approximate i-counters in $O(\log \log n)$ time (Lemma 6.4) by adding the vectors at x_L, x_R. We set the bitmap of x to be the bitwise OR of bitmaps stored in x_L and x_R. In order to maintain Invariant 7.4, the data structure adds trivial (i, t)-shortcuts whenever x has an (i, t)-bit set to 1 and exactly one of x_L or x_R has its (i, t)-bit set to 1. This is done in $O(1)$ time using bitwise operations. ∎

7.3   Top Trees

The top tree is an $O(\log \log n)$-height mergeable binary tree. All middle tree roots are top tree leaves. As a consequence of the middle tree reduction procedure described below, each top tree has at most $4 \log n$ top tree leaves. Each top tree node x maintains pointers to its parent and children, approximate counters, and a bitmap of (i, t) pairs indicating whether a local tree leaf with (i, t)-status appears in the subtree of x.

Whenever we invoke the Middle Tree Reduction procedure, the entire top tree is rebuilt.

Middle Tree Reduction. There are at most $\log n$ possible ranks for a middle tree node. If there are at least $2 \log n$ middle trees in a local tree, then the data structure invokes the *middle tree reduction* procedure: (1) destroy the top tree, (2) repeatedly take two middle tree roots with

the same rank, and *join* the corresponding middle trees, then (3) rebuild the top tree on the remaining middle tree roots. The size of the top tree can be as large as $4 \log n$ immediately after merging the top trees of two sibling $\mathcal{H}$-nodes.

LEMMA 7.6. *The following operations are supported on the top trees, with the following worst case time complexities (in parentheses):*
— *Insert a middle tree root into the top tree* $(O((\log \log n)^2))$.
— *Remove a middle tree root from the top tree* $(O((\log \log n)^2))$.
— *Merge the top trees of two local trees* $(O((\log \log n)^2))$.
— *Given the list of all middle tree roots that are leaves of the top tree, perform a middle tree reduction and rebuild the top tree* $(O(\log n \log \log n))$.
— *Update approximate counters along the path from the given top tree leaf x_M to the top tree root x_T* $(O((\log \log n)^2))$.
— *Remove (i, t)-status of a given middle tree root* $(O(\log \log n))$.

PROOF. The top tree implements leaf-insertion, leaf-deletion, and merging the two top trees in $O((\log \log n)^2)$ time. Rebuilding the top tree costs time proportional to the number of middle trees (which is $O(\log n)$), multiplied by $O(\log \log n)$ for updating approximate counters at each node. ∎

7.4   Maintaining Precision when Sampling

Recall from Invariant 6.3 that $H(x^j)$ was defined as the maximum possible height of any arithmetic formula tree (summing up approximate counters) with $x^j \in \hat{V}_j$ at the root. We define a similar function for nodes inside local trees. If $v \in \mathcal{L}(x^j)$, define $H_\ell(v)$ as:

$$H_\ell(v) = (d_{max} - j - 1) \cdot O(\log \log n) + \lfloor \log(w(v)) \rfloor + h_{\mathrm{bot/top}}(v),$$

where $h_{\mathrm{bot/top}}(v) = O(\log \log n)$ is precisely the maximum number of top, bottom, and buffer trees nodes on a path from v to a leaf of $\mathcal{L}(x^j)$. With this definition, it is straightforward to see that when v_L, v_R are the children of v, that

$$H_\ell(v) = \max(H_\ell(v_L), H_\ell(v_R)) + 1.$$

We first prove that all nodes in a local tree have the correct precision in terms of $H_\ell(v)$.

Maintaining Invariant 6.3. Invariant 6.3 constrains the accuracy of approximate i-counters in terms of $H(\cdot)$. We prove that Invariant 6.3 is maintained, by analyzing the accuracy of approximate i-counters inside the local trees in terms of $H_\ell(\cdot)$.

Fix an edge depth i and a local $(i, \text{primary})$-branching node $x \in \mathcal{H}$. Assume, inductively, that every local $(i, \text{primary})$-leaf ℓ_y in $\mathcal{L}(x)$ representing the $(i, \text{primary})$-child y of x satisfies Invariant 6.3 and $\hat{C}_i(\ell_y) = \hat{C}_i(y)$. We now prove that Invariant 6.3 is satisfied at x as well.

Fix a local $(i, \text{primary})$-branching node $v \in \mathcal{L}(x)$, and let v_L, v_R be its $(i, \text{primary})$-children, so $\hat{C}_i(v) = \hat{C}_i(v_L) \boxplus \hat{C}_i(v_R)$. By induction on $H_\ell(v)$,

$$
\begin{aligned}
\hat{C}_i(v) &\geq \left(1 - \log^{-\beta} n\right)\left(\hat{C}_i(v_L) + \hat{C}_i(v_R)\right) \\
&\geq \left(1 - \log^{-\beta} n\right)^{\max(H_\ell(v_L), H_\ell(v_R))+1}\left(C_i(v_L) + C_i(v_R)\right) \\
&\geq \left(1 - \log^{-\beta} n\right)^{H_\ell(v)} C_i(v).
\end{aligned}
$$

On the other hand, by the definition of $\boxplus$ and the inductive hypothesis, $\hat{C}_i(v) \leq \hat{C}_i(v_L) + \hat{C}_i(v_R) \leq C_i(v_L) + C_i(v_R) = C_i(v)$. In addition, for any single-child *local* $(i, \text{primary})$-node u, the approximate i-counter $\hat{C}_i(u)$ is identical to the approximate i-counter value from its local $(i, \text{primary})$-child v. Since $H_\ell(u) \geq H_\ell(v)$, the precision requirement still holds.

Let z be the root of $\mathcal{L}(x)$. Then $H_\ell(z) \leq H(x)$ (provided the leading constants hidden by the $O(\log \log n)$ factors in the definitions of H_ℓ and H are set correctly) and Invariant 6.3 holds for x as well.

7.4.1   Sample an $(i, \text{primary})$-child

This section shows that an $(i, \text{primary})$-child can be efficiently sampled approximately proportional to its approximate i-counter.

LEMMA 7.7. *Given an $(i, \text{primary})$-branching $\mathcal{H}$-node u^{j-1}, we can sample an $(i, \text{primary})$-child u^j with probability at most*

$$
\frac{\hat{C}_i(u^j)}{\hat{C}_i(u^{j-1})} \cdot (1 - \log^{-\beta} n)^{-(H(u^{j-1}) - H(u^j))}
$$

in time $O(H(u^{j-1}) - H(u^j))$. Recall that $\beta = 2$ is constant.

The data structure begins at the root of $\mathcal{L}(u^{j-1})$, which is a local $(i, \text{primary})$ node, and walks down to a descendant leaf in $\mathcal{L}(u^{j-1})$ as follows. If we are at a local $(i, \text{primary})$-branching node x, let x_L and x_R be its local $(i, \text{primary})$-children. We randomly choose a child with probability proportional to $\hat{C}(x_L)$ and $\hat{C}(x_R)$, respectively, and navigate downward using local $(i, \text{primary})$-shortcuts to find the next local $(i, \text{primary})$-branching child. The process terminates when we reach a local leaf ℓ_{u^j} (representing u^j) with local $(i, \text{primary})$-status.

Let x_0 be the root of $\mathcal{L}(u^{j-1})$, and the sequence $x_1, x_2, \ldots, x_k$ be all local $(i, \text{primary})$-branching nodes which are on the path between x_0 and $x_{k+1} = \ell_{u^j}$. For all $t \in [0, k]$, let x'_t and x''_t be the two local $(i, \text{primary})$-children of x_t, with x'_t being the ancestor of x_{t+1}.[10] Then we have for all $t \in [0, k]$, $\hat{C}_i(x'_t) = \hat{C}_i(x_{t+1})$, and the probability that a particular $(i, \text{primary})$-child

---

10   In the case of $t = 0$, if the root is not a local $(i, \text{primary})$-branching node then we take $\hat{C}_i(t''_0)$ to be zero.

u^j is sampled is at most

$$\prod_{t=0}^{k} \frac{\hat{C}_i(x_t')}{\hat{C}_i(x_t') + \hat{C}_i(x_t'')} \leq \prod_{t=0}^{k} \left[\frac{\hat{C}_i(x_t')}{\hat{C}_i(x_t') \boxplus \hat{C}_i(x_t'')} (1 - \log^{-\beta} n)^{-1} \right]$$

$$= \prod_{t=0}^{k} \left[\frac{\hat{C}_i(x_{t+1})}{\hat{C}_i(x_t)} (1 - \log^{-\beta} n)^{-1} \right]$$

$$= \frac{\hat{C}_i(x_{k+1})}{\hat{C}_i(x_0)} (1 - \log^{-\beta} n)^{-(k+1)}$$

$$\leq \frac{\hat{C}_i(u^j)}{\hat{C}_i(u^{j-1})} (1 - \log^{-\beta} n)^{-(H(u^{j-1}) - H(u^j))}.$$

7.5   Local Tree Operations

Lemmas 7.1, 7.5, and 7.6 established worst case bounds on the elementary operations inside buffer, bottom, middle, and top trees. Lemma 7.8 lists the operations supported by the local tree as a whole, and analyzes their *amortized* time costs.

LEMMA 7.8. *There exists a data structure that supports the following operations between an $\mathcal{H}$-node v and its $\mathcal{H}$-children, with the following amortized time complexities (in parentheses):*
- *Attach a new $\mathcal{H}$-child x to v $\left(O((\log \log n)^2) \right)$.*
- *Detach an $\mathcal{H}$-child x of v $\left(O((\log \log n)^2) \right)$.*
- *Let S be a set of $\mathcal{H}$-children of v. Merge S into a single node x', which is a new $\mathcal{H}$-child of v. $\left(O(|S|(\log \log n)^2) \right)$.*
- *Given a non-root $\mathcal{H}$-node x, return its $\mathcal{H}$-parent v $(O(H(v) - H(x)))$.*
- *Given an $\mathcal{H}$-node v, enumerate all $\mathcal{H}$-children of v with (i, t)-status $(O(\log \log n)$ per child) or decide if v has a unique (i, t)-child $(O(\log \log n))$.*
- *Given an $\mathcal{H}$-node x and a bit vector b, add local (i, t)-status to the local tree leaf ℓ_x for all (i, t) pairs indicated by b $(O((\log \log n)^2))$.*
- *Given an $\mathcal{H}$-node x and a bit vector b, delete local (i, t)-status to the local tree leaf ℓ_x for all (i, t) pairs indicated by b $(O(\log \log n))$.*
- *Given an $(i, \text{primary})$-branching node v, sample an $(i, \text{primary})$-child x with probability at most*

$$\frac{\hat{C}_i(x)}{\hat{C}_i(v)} \cdot (1 - \log^{-2} n))^{-(H(v) - H(x))}. \qquad \text{(Time: } O(H(v) - H(x)))$$

- *Increase the i-counter of an $\mathcal{H}$-child x of v. $\left(O((\log \log n)^2) \right)$.*
- *Decrease the i-counter of an $\mathcal{H}$-child x of v. $(O(\log \log n))$.*

PROOF. We will address these operations one by one. The *sampling* operation was already established in Lemma 7.7, Section 7.4. We first describe the worst case cost of operations, and at the end of the proof we analyze the amortized cost.

Attach a new $\mathcal{H}$-child x. The local tree leaf ℓ_x is created and inserted into the buffer tree of $\mathcal{L}(v)$. By Lemma 7.1 the worst case cost of this operation is $O((\log \log n)^2)$. If the buffer tree is full, the algorithm converts the buffer tree into a bottom tree which costs $O(1)$ by Lemma 7.1, then creates a middle tree leaf which costs $O(1)$ time by Lemma 7.5, and possibly rebuilds the top tree which costs $O((\log \log n)^2)$ time by Lemma 7.6.

Detach an $\mathcal{H}$-child x. The local tree representitive ℓ_x is first removed from either the corresponding buffer tree or bottom tree, costing $O((\log \log n)^2)$ time by Lemma 7.1. In the case where the corresponding buffer/bottom tree root loses its local (i, t)-status, or in the case where the approximate i-counters are reduced, the entire ancestor path is updated in $O(\log n)$ time by Lemmas 7.5 and 7.6. In the case where the rank of the corresponding buffer/bottom tree root is reduced (costing $O(\log n \log \log n)$ time by Lemma 7.5), the middle tree reduction may be then invoked, costing $O(\log n \log \log n)$ time by Lemma 7.6. Notice that these $\log n$ worst case terms are amortized away at the end of this proof.

Merge sibling $\mathcal{H}$-nodes. For each node $x \in S$, we detach the representative ℓ_x in worst case $O((\log \log n)^2)$ time by Lemma 7.1, and then merge the local trees of S-nodes in pairs until there is only one node left. To merge local trees we first merge their buffer trees (costing $O((\log \log n)^2)$ time by Lemma 7.1), then merge their top trees (costing $O((\log \log n)^2)$ by Lemma 7.6). Then, if the merged buffer tree is full, make it a bottom tree (costing $O(1)$ by Lemma 7.5). Finally, if the top tree is full, call the middle tree reduction procedure (costing $O(\log n \log \log n)$ time by Lemma 7.6). Let x' be the node resulting from merging S. Its representative $\ell_{x'}$ is created, having weight that is the sum of the weights of the S-nodes, and reattached to the buffer tree of $\mathcal{L}(v)$, in $O((\log \log n)^2)$ time.

Return the $\mathcal{H}$-parent. Let $x \in \mathcal{H}$ be a non-root $\mathcal{H}$-node. We find the local representative $\ell_x \in \mathcal{L}(v)$, then walk up to the root of $\mathcal{L}(v)$ and return "v." The number of buffer, bottom, and top tree nodes traversed is $O(\log \log n)$ and the number of middle tree nodes traversed is $\log(\frac{w(v)}{w(x)}) + O(1)$. By the definition of $H(\cdot)$, this is bounded by $H(v) - H(x)$.

Enumerate all local tree leaves with local (i, t)-status. We perform a depth-first search from the local tree root. When the search encounters a top tree, a bottom tree, or a buffer tree node, the bitmaps in its children indicate whether the child contains a local tree leaf with an (i, t)-status or not. When the search encounters a middle tree node x, we examine Down[DownIdx[i, t]] to see whether there is a downward local (i, t)-shortcut leaving x or not. If there is no downward local (i, t)-shortcut leaving x, then x is a local (i, t)-branching node and the search proceeds recursively on both children. Otherwise, the search navigates downward from a local (i, t)-non-branching node x to its highest descendant (i, t)-node x'. The same navigation algorithm described in Section 5.1 is performed so that after the navigation all

(i, t)-shortcuts on the path P_{uv} are exactly local shortcuts in COVER(x, x'). (The cost of adding these shortcuts inside a local tree is amortized differently than how adding $\mathcal{H}$-shortcuts are amortized; see below.) All local tree leaves with (i, t)-status are enumerated in $O(\log \log n)$ amortized time per leaf.

To test whether there is a *unique* leaf with (i, t)-status, we navigate downward from the root z of $\mathcal{L}(v)$, following local (i, t)-shortcuts until reaching either a local (i, t)-leaf x (necessarily unique) or a local (i, t)-branching node x (indicating non-uniqueness). We then cover local (i, t)-shortcuts on the path from z to x as long as it is possible. As shown below, the amortized cost of this operation is $O(\log \log n)$.

Add local (i, t)-status to a local tree leaf. Recall that the *only* leaves that may gain local (i, t)-status are buffer tree leaves (Remark 7.2). Let ℓ_x be the local tree leaf gaining (i, t)-status. If ℓ_x resides in a bottom tree we detach it, reattach it to the buffer tree, and add (i, t)-status there. From the description above (the first two operations listed on Lemma 7.8), the time cost is amortized $O((\log \log n)^2)$ due to the potential detach/attach operation.

Delete local (i, t)-status from a local tree leaf. The algorithm first removes the local (i, t)-status from the local tree representative ℓ_x, costing $O(\log \log n)$ time by Lemma 7.1. If the corresponding bottom tree root loses some local (i, t)-status, the algorithm removes local (i, t)-status from the corresponding middle tree leaf, costing $O(\log n)$ time by Lemma 7.5. The $\log n$ worst case time cost will be amortized as described below.

Increase the approximate i-counter of an $\mathcal{H}$-child x of v. Let ℓ_x be the local tree leaf that represents x. The algorithm detaches ℓ_x, changes the i-counter value and then attaches ℓ_x to the buffer tree. The operations costs $O((\log \log n)^2)$ time from the first two operations listed on Lemma 7.8.

Decrease the approximate i-counter of an $\mathcal{H}$-child x of v. The algorithm sets the approximate i-counter at x to the new value, costing $O(\log \log n)$ time by Lemma 7.1. If ℓ_x is in a bottom tree and the corresponding bottom tree root has its approximate i-counter value changed, the algorithm invokes Lemma 7.5 and updates the approximate i-counter at the corresponding middle tree leaf, costing $O(\log n)$ worst case time and again can be amortized away by the description below.

Amortized Cost Analysis. We use a credit system. Every buffer tree leaf carries $\Theta(1)$ credits and every middle tree root carries $\Theta(\log \log n)$ credits. Suppose the buffer tree matures and becomes a bottom tree, say with root x_B. At this moment the tree has $\Theta(\log^\alpha n)$ credits, which will pay for all future costs associated with updating the middle and top tree ancestors of x_B. The following three types of events change the information stored at x_B.

1. x_B changes rank. Since the bottom tree is only subject to *detach* operations (see Remark 7.2), its weight is non-increasing. Therefore, this happens at most $\log n$ times.
2. x_B loses (i, t)-status. It can never regain (i, t)-status (Remark 7.2), so this happens at most $3d_{max} = O(\log n)$ times.
3. x_B's approximate i-counter changes. The approximate counters are non-increasing, and each such counter can take on $O(\log^{\beta+1} n)$ different values (Section 6). Since there are $\log n$ possible values for i, the total number of counter changes is $O(\log^{\beta+2} n)$.

Each of the above events requires that we update or delete the entire path form x_B to the local tree root, which can have length $\Theta(\log n)$. Events of type (1) take $O(\log n \log \log n)$ time to destroy the path and reinsert new middle tree roots into the top tree, each with $O(\log \log n)$ credits. Events of type (2) and (3) take $O(\log n)$ time to update the (i, t)-status or approximate i-counters of all ancestors of x_B. Since $\beta = 2$, the total cost for events of type (3) is the bottleneck. They take $O(\log^{\beta+3} n)$ time over the life of the bottom tree. We set $\alpha \geq \beta + 3 = 5$, so the credits of a bottom tree suffice to pay for all costs incurred over the lifetime of the bottom tree.

A middle tree reduction procedure is invoked if the leaf set S of the top tree has size $|S| \geq 2 \log n$. Thus, it begins with at least $2 \log n \cdot O(\log \log n)$ credits and ends with at most $\log n \cdot O(\log \log n)$ credits, which pays for rebuilding the top tree in $O(\log n \log \log n)$ time (Lemma 7.6).

The number of shortcuts removed is bounded by the number created, so it suffices to account for the cost of creating shortcuts. Local shortcuts are created in two ways: (i) in response to the creation of a middle tree node (joining two middle trees), and (ii) lazy covering. The cost of case (i) is ultimately paid for by the deletion of that middle tree node, which in turn is paid for by the bottom tree that triggered the deletion. The cost of case (ii) is attributed to the removal of (i, t)-status at some corresponding middle tree leaf with an (i, t)-status, which is accounted for in the cost of type (2) events.   ■

8.   Loose Ends

Some of the operations on the hierarchy $\mathcal{H}$ required the definition of (i, t)-forests (Section 5) and local trees (Section 7) and could not be described until now. In Section 8.1 we analyze the cost of searching for a replacement edge using the two-stage batch sampling test sketched in Section 3.2.2. In Section 8.2 we explain how to maintain (i, t)-forests (Invariant 5.3), and in particular, how to efficiently merge two such forests when doing batch promotions/upgrades. In Section 8.3 we prove Lemma 6.5 concerning approximate counters, and show that Invariant 6.3 is maintained.

8.1   The Batch Sampling Test

Recall from the deletion algorithm of Section 3.2.1 that $\mathbf{u^i}$ is the new $\mathcal{H}$-node resulting from merging a set of siblings. In this section we show how the data structure performs the batch sampling test among i-primary endpoints touching $\mathbf{u^i}$. Let p and s be the number of i-primary and i-secondary endpoints touching $\mathbf{u^i}$, and let $\hat{p} = \hat{C}_i(\mathbf{u^i})$ be a $(1 + o(1))$-approximation of p. (Retrieving $\hat{p}$ is Operation (9) from Lemma 3.1.)

Single Sample Test. To $(1 + o(1))$-uniformly sample one i-primary endpoint touching $\mathbf{u^i}$, the data structure sets $x = \mathbf{u^i}$ and iteratively performs the following step. Base case: If x is an $(i, \text{primary})$-leaf, then return an i-primary endpoint at x uniformly at random. General case: If x is an $(i, \text{primary})$-branching node, then use $\mathcal{L}(x)$ to sample an $(i, \text{primary})$-child x' of x with probability at most $\frac{\hat{C}_i(x')}{\hat{C}_i(x)}(1 - \log^{-\beta} n)^{-(H(x)-H(x'))}$ (Lemma 7.7). If x' is an $(i, \text{primary})$-branching node or leaf, we set $x = x'$ and repeat. Otherwise, we repeatedly follow the $(i, \text{primary})$-shortcuts leaving x' to its $(i, \text{primary})$-child x'', set $x = x''$, and repeat (Lemma 5.7).

Notice that with accurate counters this procedure picks a perfectly uniformly random i-primary endpoint. Let $\langle x, \{x, y\}\rangle$ be the sampled endpoint and $x_0 = \mathbf{u^i}, x_1, \ldots, x_k = x$ be the sequence of $(i, \text{primary})$-branching nodes on the path in $\mathcal{H}$ from $\mathbf{u^i}$ to x. Then the probability that $\langle x, \{x, y\}\rangle$ is sampled is at most

$$\frac{1}{C_i(x)} \prod_{j=0}^{k-1} \left[\frac{\hat{C}_i(x_{j+1})}{\hat{C}_i(x_j)}(1 - \log^{-\beta} n)^{-(H(x_j)-H(x_{j+1}))} \right]$$

$$= \frac{1}{C_i(x)} \left[\prod_{j=0}^{k-1} \frac{\hat{C}_i(x_{j+1})}{\hat{C}_i(x_j)} \right] (1 - \log^{-\beta} n)^{-H(x_0)}$$

$$= \frac{1}{C_i(x)} \frac{\hat{C}_i(x)}{\hat{C}_i(\mathbf{u^i})}(1 - \log^{-\beta} n)^{-H(x_0)}$$

$$= (1 - o(1))\frac{1}{\hat{C}_i(\mathbf{u^i})}(1 - \log^{-\beta} n)^{-O(\log n \log\log n)} \qquad (H(x_0) = O(\log n \log\log n))$$

$$\leq (1 + o(1))\frac{1}{C_i(\mathbf{u^i})}. \qquad (\beta = 2)$$

The $1/C_i(x)$ factor reflects the fact that once we reach the $\mathcal{H}$-leaf x, an endpoint touching x is selected (exactly) uniformly at random, without any approximation. To check whether $\{x, y\}$ is a replacement edge or not, it suffices to check whether $y^i = \mathbf{u^i}$. This can be accomplished by starting from y and repeatedly accessing the $\mathcal{H}$-parent until y^i is reached. Using local trees, the cost of computing $\mathcal{H}$-parents along a path telescopes to $H(y^i) = O(\log n \log\log n)$.

The Preprocessing Method. Another way to sample i-primary endpoints is to first enumerate <u>all</u> i-primary endpoints and <u>all</u> i-secondary endpoints touching $\mathbf{u^i}$ in $O((p + s) \log\log n)$ time, mark all enumerated endpoints and store all i-primary endpoints in an array. Then the data

structure samples an i-primary endpoint uniformly at random from all enumerated i-primary endpoints and checks whether the other endpoint is marked in $O(1)$ time.

Batch Sampling Test on k Samples. The data structure runs the two sampling methods in parallel and halts when the first finishes. Thus, the time to sample k $(i, \text{primary})$-endpoints is

$$O\left(\min\left\{(p + s)\log\log n + k, \ k\log n\log\log n\right\}\right).$$

8.1.1   Cost Analysis for Sampling Procedure

As described in Section 3.2.2, the sampling procedure either returns a replacement edge, or invokes the enumeration procedure. Roughly speaking, if no replacement edge is found, the cost is charged to either upgrades of $(i, \text{secondary})$ endpoints or promotions to $(i, \text{primary})$ endpoints. If any replacement edge is found, the data structure is willing to pay $O(\log n(\log\log n)^2)$ cost because this happens at most once per `Delete` operation.

If the enumeration procedure is invoked, the data structure upgrades all enumerated i-secondary endpoints touching $\mathbf{u^i}$ to i-primary endpoints, and then all i-primary endpoints touching $\mathbf{u^i}$ associated with non-replacement edges are promoted to $(i+1)$-secondary endpoints. The first batch sampling test, when $k = O(\log\log \hat{p}) = O(\log\log p)$, costs

$$T_1 = O(\min((p + s)\log\log n, \ \log\log p \cdot \log n\log\log n)).$$

The second batch sampling test ($k = O(\log p)$), if invoked, costs

$$T_2 = O(\min((p + s)\log\log n, \ \log p \cdot \log n\log\log n)).$$

The enumeration procedure, if invoked, costs

$$T_E = O((p + s)(\log\log n)^2).$$

Let ρ be the fraction of i-primary endpoints touching $\mathbf{u^i}$ associated with replacement edges before the execution of the sampling procedure. The rest of the analysis is separated into two cases:

Case 1. If $\rho \geq 3/4$, the probability that the first batch sampling test returns with a replacement edge is at least $1 - (1/4 + o(1))^{O(\log\log p)} > 1 - 1/\log p$.[11] The second batch sampling test, if invoked, returns a replacement edge if at least half the $O(\log p)$ endpoints sampled belong to replacement edges. By a standard Chernoff bound, the probability that the second batch *fails* to return a replacement edge and halt is $\exp(-\Omega(\log p)) < 1/p$.

---

11   It is $1/4 + o(1)$ because the sampling procedure is only $(1 + o(1))$-approximate.

The expected time cost is therefore

$$T_1 + (1/\log p)T_2 + (1/p)T_E = O\left(\left(\log n + \frac{p+s}{p}\right)(\log\log n)^2\right) = O((\log n + s)(\log\log n)^2)$$

We charge the `Delete` operation $O(\log n (\log\log n)^2)$, which covers the expected cost of the two batch sampling steps <u>and</u> the expected cost of dealing with *primary* endpoints if the enumeration step is reached. If the enumeration step is reached, endpoint upgrades pay for the $\Theta(s(\log\log n)^2)$ cost of dealing with secondary endpoints.

Case 2. Otherwise, $\rho < 3/4$. If the enumeration procedure is ultimately invoked, a $1 - \rho = \Omega(1)$ fraction of the i-primary endpoints touching $\mathbf{u^i}$ belong to non-replacement edges, which are promoted to depth $i + 1$, and all s i-secondary endpoints are upgraded to either i-primary or $(i + 1)$-secondary status. In this case the time cost is

$$T_1 + T_2 + T_E = O((p + s)(\log\log n)^2),$$

which is charged to the promoted edges/upgraded endpoints. We need to prove that the probability of terminating after the second batch sampling test is sufficiently small. If $\rho \geq 1/4$ then the probability of the first batch sampling test *not* returning a replacement edge is at most $(3/4 + o(1))^{O(\log\log p)} < 1/\log p$. In this case the expected cost is

$$T_1 + (1/\log p)T_2 = O(\log n (\log\log n)^2).$$

If $\rho < 1/4$ then, by a Chernoff bound, the probability that at least half the sampled endpoints belong to replacement edges is $\exp(-\Omega(\log p)) < 1/p$. Therefore the expected cost when the enumeration procedure is not invoked with $\rho < 1/4$ is at most

$$(1/p)(T_1 + T_2) = O(\log n \log\log n),$$

which is charged to the `Delete` operation.

8.2 Maintaining (i, t)-Forests

Lemma 8.1 summarizes the operations on (i, t)-forests which are implemented via the shortcut infrastructure and local trees, together with their corresponding time cost.

LEMMA 8.1. *There exists a data structure on $\mathcal{H}$ supporting the following operations with amortized time (in parenthesis):*

— *Given an $\mathcal{H}$-leaf x and an (i, t) pair, designate x an (i, t)-leaf $\left(O(\log n (\log\log n)^2)\right)$.*
— *Given an (i, t)-leaf x, remove its (i, t)-status $\left(O(\log n (\log\log n)^2)\right)$.*
— *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 $(O(1 + k \log\log n)$ where k is the number of enumerated (i, t)-children).*

— *Given an (i, t)-tree $\mathcal{T}$ rooted at v, an integer $i' \in [i, d_{max}]$, 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) $\left(O(|\mathcal{T}|(\log \log n)^2 + 1)\right)$.*

Each operation assumes that, prior to the execution of the operation, Invariant 5.3 holds for all $\mathcal{H}$-nodes of depth $\geq i$, where i is part of the input of the operation. Moreover, Invariant 5.3 is guaranteed to hold for all $\mathcal{H}$-node of depth $\geq i$ after each operation is completed.

The remainder of this section constitutes a proof of Lemma 8.1.

Add (i, t)-status to an $\mathcal{H}$-leaf. Let x be the $\mathcal{H}$-leaf. In order to identify the (i, t)-branching ancestor of x, the data structure climbs up $\mathcal{H}$ and finds the first $\mathcal{H}$-node x' that is *either* an (i, t)-node or has a downward (i, t)-shortcut $x' \leftrightharpoons x''$. If x' is an (i, t)-branching node, then since the $\mathcal{H}$-child of x' that is also an ancestor of x is not an (i, t)-node, x' is the (i, t)-branching ancestor of x. Otherwise, the data structure performs a binary search on the path $P_{x'x''}$ to find the (i, t)-branching ancestor as follows:

If $x' \leftrightharpoons x''$ is not a fundamental (i, t)-shortcut, the data structure uncovers $x' \leftrightharpoons x''$ into $x' \leftrightharpoons y$ and $y \leftrightharpoons x''$ and recurses to one of the two subpaths depending on whether y is an ancestor of x or not. Otherwise, $x' \leftrightharpoons x''$ is fundamental, and in this case x' is the branching node we are looking for. Let x''' be the ancestor of x that is a child of x'. We uncover $x' \leftrightharpoons x''$, give local (i, t)-status to $\ell_{x'''}$ and $\ell_{x''}$ in $\mathcal{L}(x')$, and then cover all shortcuts on the path $P_{x''',x}$, using Lemma 5.9 (See Section 9.1.) The cost for walking up these local trees telescopes to $O(\log n \log \log n)$ by Lemma 7.8. Now suppose that $t = $ primary. For every (i, t)-branching node y that is an ancestor of x, the data structure updates the approximate i-counter stored in y, using Lemma 7.8. Now, Invariant 5.3 is restored on all $\mathcal{H}$-nodes with depth $\geq i$ since all (i, t)-shortcuts between x''' and x form the path $P_{x''',x}$. Since there are at most $d_{max} = O(\log n)$ such (i, t)-branching nodes affected, the amortized cost is at most $O(\log n (\log \log n)^2)$.

Remove (i, t)-status from an (i, t)-leaf. Let x be the $\mathcal{H}$-leaf. The data structure navigates up from x by upward (i, t)-shortcuts until it reaches a single-child (i, t)-node q. The intermediate (i, t)-shortcuts are removed by setting their (i, t)-bits to 0.

The data structure then removes the local (i, t)-status of the local tree leaf ℓ_q representing q. If the (i, t)-parent p of q (which is also its $\mathcal{H}$-parent) now has only one (i, t)-child q', p is no longer an (i, t)-branching node. The data structure removes the (i, t)-status of q', removes *local* (i, t)-status of $\ell_{q'}$ in $\mathcal{L}(p)$ using Lemma 7.8, removes the (i, t)-branching status of p, and covers the fundamental (i, t)-shortcut $p \leftrightharpoons q'$ using Lemma 5.9. This may also cause p to lose its (i, t)-status.

Notice that this operation is equivalent to first performing the lazy covering on the (i,t)-shortcuts from x to its (i,t)-parent and then removing x. Hence, the time cost for removing (i,t)-status from x is amortized $O((\log\log n)^2)$. We can remove (i,t)-status from a group of leaves S^- in $O(|S^-|(\log\log n)^2)$ amortized time by repeating this procedure for every leaf. Notice that Invariant 5.3 holds for all $\mathcal{H}$-nodes with depth $\geq i$ because fundamental (i,t)-shortcuts are covered when $\mathcal{H}$-nodes lose their (i,t)-branching status.

Enumerating (i,t)-children. This is an operation of Lemma 7.8.

Given an (i,t)-tree $\mathcal{T}$ and a set of leaves S^+ in $\mathcal{T}$, add (i',t')-status to the leaves in S^+. First of all, the data structure creates a "dummy" tree induced from the set of leaves S^+ and the root of $\mathcal{T}$, by first copying the entire (i,t)-tree $\mathcal{T}$, enumerating all its leaves and removing all the leaves that do not belong to S^+.[12] Hence, without loss of generality, we now assume S^+ is the entire leaf set of $\mathcal{T}$ and that there are no potential shortcuts w.r.t. $\mathcal{T}$.

Notice that, after adding (i',t')-status to the leaves in $\mathcal{T}$, every (i,t)-branching node of depth at least i' in $\mathcal{T}$ is also an (i',t')-branching node. Moreover, for each such (i,t)-branching node, adding (i',t')-status to the node converts at most one $\mathcal{H}$-node into a new (i',t')-branching node.

Define $\mathcal{T}^*$ to be the subtree of $\mathcal{H}$ induced by <u>all</u> ancestors of leaves in $\mathcal{T}$ up to depth i. Our first task is to enumerate all nodes of $\mathcal{T}^*$ at depth i'; call them $r_1,\ldots,r_k$.

CLAIM 8.2. *The nodes $r_1,\ldots,r_k$ can be enumerated in worst case $O(k\log\log n)$ time.*

PROOF. We perform a depth first search of $\mathcal{T}$ looking for nodes at depth i'. Let x be the locus of the search; initially x is the root of $\mathcal{T}$. If x is at depth i' we output x and backtrack. If x is a $\mathcal{T}$-branching node we continue the search recursively on each $\mathcal{T}$-child of x. If x has a single downward $\mathcal{T}$-shortcut $x \leftrightharpoons x'$ and x' has depth *strictly* greater than i' we iteratively uncover the downward shortcut from x until it is $x \leftrightharpoons x''$, where x'' has depth at most i', and move the locus of the search to x''. If k nodes are output by this procedure, the number of shortcuts followed/uncovered is $k \cdot O(\log\log n)$. ∎

Let $\mathcal{T}_1,\ldots,\mathcal{T}_k$ be the subtrees of $\mathcal{T}$ rooted at $r_1,\ldots,r_k$ and let $\mathcal{W}_1,\ldots,\mathcal{W}_k$ be the (i',t')-trees rooted at these nodes. It may be that some r_l does not currently have (i',t')-status, in which case $\mathcal{W}_l$ is empty. In this case we simply traverse $\mathcal{T}_l$, giving each node encountered (i',t')-status. In Claim 8.3 we focus on the non-trivial problem of merging $(\mathcal{T}_l,\mathcal{W}_l)$ when r_l is an existing (i',t')-root. Here "$\mathcal{W}_l$-status" is synonymous with (i',t')-status.

---

12   This is the reason for having $3d_{max}+1$ slots in the Down arrays; the +1 is for creating a temporary dummy tree of this type.

CLAIM 8.3. *Let $\mathcal{T}_l, \mathcal{W}_l$ be two trees rooted at r_l, where all shortcuts are maximal. We can give $\mathcal{W}_l$-status to all leaves of $\mathcal{T}_l$ (and find all new $\mathcal{W}_l$-branching vertices) in amortized $O(|\mathcal{T}_l|(\log \log n)^2)$ time, independent of the size of $\mathcal{W}_l$.*
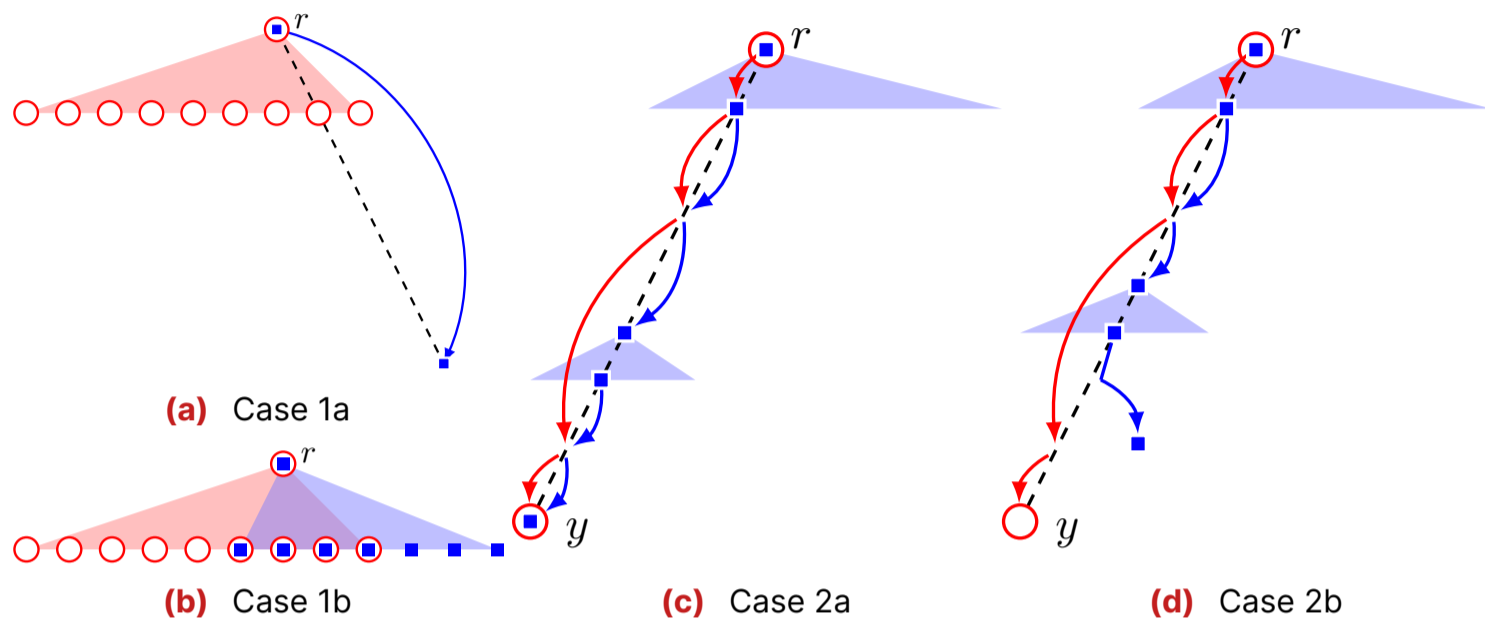

Figure 9. The examples to the four cases in the proof of Lemma 8.3. The red circle nodes are $\mathcal{T}$-nodes and the blue square nodes are $\mathcal{W}$-nodes.

PROOF. We merge $\mathcal{T}_l$ and $\mathcal{W}_l$ in a depth-first manner. Let r be the locus of the search; initially $r = r_l$. We maintain the invariant that r is both a $\mathcal{T}_l$-node and a $\mathcal{W}_l$-node. There are two main cases; Case 1 is when r is a branching $\mathcal{T}_l$-node and Case 2 is when r is a single-child $\mathcal{T}_l$-node. See Figure 9 for illustration.

Case 1a: *r is a branching $\mathcal{T}_l$-node but not a branching $\mathcal{W}_l$-node.* After the merging process r will become a branching $\mathcal{W}_l$-node, and therefore can have no downward $\mathcal{W}_l$-shortcut. We repeatedly uncover the $\mathcal{W}_l$-shortcut leaving r. In the final step we uncover a fundamental shortcut $r \leftrightharpoons x$, give ℓ_x local $\mathcal{W}_l$-status in $\mathcal{L}(r)$, and then designate r a branching $\mathcal{W}_l$-node. This reduces the situation to Case 1b.

Case 1b: *r is both a branching $\mathcal{T}_l$-node and branching $\mathcal{W}_l$-node.* Enumerate every $\mathcal{T}_l$-child r' of r. If r' does not have $\mathcal{W}_l$-status, traverse the entire subtree of $\mathcal{T}_l$ rooted at r', marking each node encountered as a $\mathcal{W}_l$-node, and give $\ell_{r'}$ local $\mathcal{W}_l$-status in $\mathcal{L}(r)$. Otherwise, move the locus of the search to r' and recursively merge the subtrees of $\mathcal{T}_l$ and $\mathcal{W}_l$ rooted at r'.

Case 2a: *r is a single-child $\mathcal{T}_l$-node and the $\mathcal{T}_l$-child of r is a $\mathcal{W}_l$-node or has a downward $\mathcal{W}_l$-shortcut.* Let y be the $\mathcal{T}_l$-child of r. If y is a $\mathcal{W}_l$-node then there are no *new* branching vertices on

the path from r to y (exclusive). In this case we move the locus of the search to y and continue recursively. If y is not a $\mathcal{W}_l$-node but has a downward $\mathcal{W}_l$-shortcut it becomes a branching $\mathcal{W}_l$-node. We repeatedly uncover its downward $\mathcal{W}_l$-shortcut, culminating in uncovering a fundamental shortcut $y \leftrightharpoons x$, then designate ℓ_x a local $\mathcal{W}_l$-node in $\mathcal{L}(y)$ and designate y a branching $\mathcal{W}_l$-node. Finally we move the locus of the search to y.

Case 2b: *r is a single-child $\mathcal{T}_l$-node, but its $\mathcal{T}_l$-child y is neither a $\mathcal{W}_l$-node nor has a $\mathcal{W}_l$-shortcut.* In this case, y will become a branching $\mathcal{W}_l$-node or $\mathcal{W}_l$-leaf. In addition, there may be a new branching $\mathcal{W}_l$-node on the path from r to y. We proceed to find the new branching node as follows. Initialize $x = r$ and let $x \leftrightharpoons x'$ refer to its current downward $\mathcal{T}_l$-shortcut. Whenever x' is a $\mathcal{W}_l$-node or has a $\mathcal{W}_l$-shortcut, we move the locus of the search to x', setting $x = x'$. Whenever x has a downward $\mathcal{T}_l$-shortcut $x \leftrightharpoons x'$ and a $\mathcal{W}_l$-shortcut $x \leftrightharpoons x''$ with $x' \neq x''$, we uncover the one with maximum power, or uncover both if they have the same power. If $x \leftrightharpoons x''$ does not exist because x is a branching $\mathcal{W}_l$-node then we repeatedly uncover $x \leftrightharpoons x'$. Eventually this process terminates when we uncover a fundamental $\mathcal{T}_l$-shortcut $x \leftrightharpoons x'$ (perhaps uncovering a fundamental $\mathcal{W}_l$-shortcut $x \leftrightharpoons x''$ at the same time). Then x is the new branching $\mathcal{W}_l$-node. We designate it as such, and explore the $\mathcal{T}_l$ subtree rooted at x', giving all $\mathcal{T}_l$-nodes and shortcuts encountered $\mathcal{W}_l$-status.

About Invariant 5.3. Notice that all new (i', t') branching nodes are correctly identified by the procedure described above, and that $i' \geq i$. Thus, Invariant 5.3 holds for all $\mathcal{H}$-nodes of depth $\geq i$.

Time Complexity. The time required to traverse $\mathcal{T}_l$ and identify all new branching nodes is $O(|\mathcal{T}_l| \log \log n)$. The running time is dominated by the cost of introducing up to $O(|\mathcal{T}_l|)$ new branching vertices and adding $\mathcal{W}_l$-status to $O(|\mathcal{T}_l|)$ nodes. The cost of adding $\mathcal{W}_l$-status is $O((\log \log n)^2)$ and the cost of uncovering a fundamental $\mathcal{W}_l$-shortcut, in Case 1a or Case 2b, is also $O((\log \log n)^2)$. In total the time is $O(|\mathcal{T}_l|(\log \log n)^2)$. ∎

8.3   Approximate Counters Operations — Proof of Lemma 6.5

Update ancestor approximate i-counters. The data structure updates the approximate i-counters from a given $\mathcal{H}$-leaf x to the corresponding $\mathcal{H}$-root. Let v be the current $(i, \text{primary})$-node. If v is a single-child $(i, \text{primary})$-node, then it adopts the approximate i-counter of its $(i, \text{primary})$-child. If v is the child of an $(i, \text{primary})$-branching node u, the data structure updates the approximate i-counters of v from $\mathcal{L}(u)$ using Lemma 7.8. At this point u adopts the approximate i-counter of the root of $\mathcal{L}(u)$. There are at most $\log n$ branching nodes on the path and each costs $O((\log \log n)^2)$ time to update an i-counter (Lemma 7.8), for a total of $O(\log n (\log \log n)^2)$ time.

Update approximate *i*-counters in an (*i*, primary)-tree $\mathcal{T}$ rooted at u^i. At the beginning of this operation, the approximate *i*-counters at all (*i*, primary)-leaves are accurate but those at internal nodes are presumed invalid. Beginning at the root u^i, the data structure traverses the (*i*, primary) tree $\mathcal{T}$ in a postorder fashion, setting approximate *i*-counters in this order. As in the analysis above, the cost is $O((\log \log n)^2)$ per node in $\mathcal{T}$, for a total of $O(|\mathcal{T}|(\log \log n)^2)$.

Update approximate counters at a merged/split $\mathcal{H}$-node x. Suppose $x = \mathbf{u^i}$ is the result of merging several siblings. We inspect the root of $\mathcal{L}(x)$ and retrieve the bitmap I indicating for which (*i*, primary)-pairs x is an (*i*, primary)-branching node. Using table lookups, in $O(\log \log n)$ time we make an $O(\log n \log \log n)$-bit mask and copy all the approximate *i*-counters from the root of $\mathcal{L}(x)$ to x. The case when x is the result of a split is handled in the same way.

9.   Amortized Analysis of Shortcut Maintenance

In this section, we describe how shortcuts are utilized and supported on $\mathcal{H}$. Moreover, we provide a potential function for $\mathcal{H}$-shortcuts that contributes to the amortized analysis for the `Delete` operation.

9.1   Covering All Shortcuts Touching Specified Paths — Proof of Lemma 5.9

The remainder of this section constitutes a proof of Lemma 5.9. Let P be a path from the given $\mathcal{H}$-node u^i to the corresponding $\mathcal{H}$-root u^0.

Uncover and remove all $\mathcal{H}$-shortcuts touching P. Removing a fundamental shortcut is a local tree operation that costs $O((\log \log n)^2)$ time. Uncovering a shortcut with both endpoints on the path costs $O(\log \log n)$ time by Lemma 5.6. (Such a shortcut may be an (i', t')-shortcut for multiple (i', t') pairs.) Uncovering a non-fundamental deviating (i, t)-shortcut costs $O(1)$ time, by setting the appropriate (i, t)-bits in the supporting shortcuts. Thus, the total cost of uncovering and removing all of the $\mathcal{H}$-shortcuts on the affected paths is $O(\log n(\log \log n)^2)$.

For each $\mathcal{H}$-node x iterated from u^0 to u^i, the data structure first enumerates all downward $\mathcal{H}$-shortcuts in Down_x. Then the data structure repeatedly uncovers the $\mathcal{H}$-shortcut with the largest power > 0 until every $\mathcal{H}$-shortcut leaving x is fundamental.

The data structure then uncovers each fundamental $\mathcal{H}$-shortcut leaving x by the following procedure. To uncover (remove) a fundamental $\mathcal{H}$-shortcut $x \leftrightharpoons y$, the data structure first detaches the local leaf ℓ_y in $\mathcal{L}(x)$ representing y and re-inserts ℓ_y into the buffer tree. Notice that this operation does not alter the structure of $\mathcal{H}$, so any $\mathcal{H}$-shortcut leaving y is not affected. Then the data structure adds local (i, t)-status to ℓ_y for all (i, t) pairs indicated in the bitmap $b_{x \leftrightharpoons y}$. This enables one to navigate from the root of $\mathcal{L}(x)$ to ℓ_y via *local* (i, t)-shortcuts in $\mathcal{L}(x)$. To preserve Invariant 5.3 (and thereby keep the whole (i, t)-forest in $\mathcal{H}$ navigable) we

designate x, y (i, t)-nodes for each (i, t)-bit indicated in $b_{x \leftrightharpoons y}$. By the local tree operations listed in Lemma 7.1, the time cost for uncovering (removing) a fundamental $\mathcal{H}$-shortcut is amortized $O((\log \log n)^2)$.

Adding a fundamental shortcut between an $\mathcal{H}$-node v and its $\mathcal{H}$-parent u for all (i, t) pairs indicated by the bit vector b. This can be done by first invoking Lemma 7.8, removing local (i, t)-status from ℓ_v, and then adding a shortcut $u \leftrightharpoons v$ via Lemma 5.6. The time cost is $O(\log \log n)$.

Adding all fundamental $\mathcal{H}$-shortcuts touching P shared by some (i, t) pairs. There are two types of fundamental $\mathcal{H}$-shortcuts touching P: (1) having both endpoints on P, and (2) deviating from P.

To add all fundamental $\mathcal{H}$-shortcuts touching P, the data structure checks for edge depth j iterated from i to 1 whether to add the fundamental shortcut $u^{j-1} \leftrightharpoons u^j$ or not. It should be added if, for some (i, t) pair, u_j is an (i, t)-node but u^{j-1} is *not* an (i, t)-branching node. To check this, the data structure first obtains a bitmap b stored in u^j indicating which (i, t) pairs have an (i, t)-status at u^j, and then accesses the path in the local tree $\mathcal{L}(u^{j-1})$ from ℓ_{u^j} to the root of $\mathcal{L}(u^{i-1})$. During this traversal, if we encounter a *local* (i, t)-branching node we set the corresponding (i, t)-bit in b to zero. When we reach the root of $\mathcal{L}(u^{j-1})$, if b is still non-zero, the data structure creates the fundamental $\mathcal{H}$-shortcut $u^{j-1} \leftrightharpoons u^j$ with $b_{u^{j-1} \leftrightharpoons u^j} = b$. Furthermore, for each (i, t)-bit set to 1 in b, the data structure removes local (i, t)-status from the local tree leaf ℓ_{u^j}. If u_j is not an (i, t)-branching node, we also remove (i, t)-status from u_j.

To handle the second case, notice that by Lemma 5.5, for each (i, t) pair there is at most one fundamental (i, t)-shortcut deviating from P. In particular, for an (i, t) pair, at most one deviating fundamental (i, t)-shortcut is added touching the unique $\mathcal{H}$-node u^{j-1} such that u^{j-1} belongs to an (i, t)-forest but u^j does not. The data structure forms the bitmap *diff* in $O(1)$ time indicating all such pairs. For each (i, t) in *diff* we check in $O(\log \log n)$ time whether $\mathcal{L}(u^{j-1})$ contains a single leaf ℓ_y with local (i, t)-status (Lemma 7.8). If so, we create a fundamental shortcut $u^{j-1} \leftrightharpoons y$, remove local (i, t)-status from ℓ_y, and remove (i, t)-status from y if it is not an (i, t)-branching node.

We now analyze the time cost. For (1), at most $O(\log n)$ $\mathcal{H}$-shortcuts are covered, and each covering involves multiple (i, t) pairs so each covering can be done in $O(H(u^j) - H(u^{j+1}))$ time (Lemma 7.8), which telescopes to $O(\log n \log \log n)$. Moreover, removing (i, t)-status on local tree leaves costs $O(\log \log n)$ time, by Lemma 7.8. For (2), there are $O(\log n)$ possible deviating fundamental shortcuts to be created. Each requires $O(\log \log n)$ amortized time, for a total of $O(\log n \log \log n)$ amortized time.

Cover all (i,t)-shortcuts having both endpoints on P. In addition to adding all of the fundamental shortcuts, the data structure adds back all of the $\mathcal{H}$-shortcuts on the path P from u^j to u^0. This is done by traversing $P \log\log n$ times. In the p-th traversal the data structure covers all possible $\mathcal{H}$-shortcuts of power $p+1$ that have both endpoints on the path. Each shortcut is covered in $O(\log\log n)$ time: to cover $x \leftrightharpoons y$ from $x \leftrightharpoons y'$ and $y' \leftrightharpoons y$, the data structure first adds the shortcut $x \leftrightharpoons y$ into Up_y. Then the data structure computes the bitwise AND of two bitmaps by setting $b_{x \leftrightharpoons y} \leftarrow b_{x \leftrightharpoons y'} \wedge b_{y' \leftrightharpoons y}$, and removes the bits in the covered shortcuts by setting $b_{x \leftrightharpoons y'} \leftarrow b_{x \leftrightharpoons y'} \oplus b_{x \leftrightharpoons y}$ and $b_{y' \leftrightharpoons y} \leftarrow b_{y' \leftrightharpoons y} \oplus b_{x \leftrightharpoons y}$. Finally, the data structure updates UpIdx_y, Down_x and DownIdx_x according to $b_{x \leftrightharpoons y}$, and if $b_{x \leftrightharpoons y'}$ and/or $b_{y' \leftrightharpoons y}$ becomes 0, updates Down_x, DownIdx_x, Occ_x, $\text{Down}_{y'}$, $\text{DownIdx}_{y'}$, $\text{Occ}_{y'}$ appropriately.

It is straightforward to see that, after $\log\log n$ passes, if there is any (i,t)-shortcut with at least one endpoint on the path that could be covered, the other endpoint must be outside of the path and hence is a deviating (i,t)-shortcut. Since there are a total of $O(d_{max}) = O(\log n)$ non-fundamental $\mathcal{H}$-shortcuts to consider, the total time cost is $O(\log n \log\log n)$.

9.2   Shortcut Cost Analysis

At first glance it seems sensible to charge the cost of deleting a shortcut to the creation of the shortcut, and therefore only account for their creation in the amortized analysis. This does not quite work because shortcuts are *shared* between many (i,t) pairs and the cost of deleting a shortcut depends on how broadly it is shared. The amortized analysis for $\mathcal{H}$-shortcuts focusses on *supporting potential shortcuts* defined as follows:

DEFINITION 9.1. Let u be a single-child (i,t)-node and v be the (i,t)-child of u. Then the *maximal potential (i,t)-shortcuts* are the maximal shortcuts with respect to the covering relation having both endpoints on the path P_{uv}. The *supporting potential (i,t)-shortcuts* are the $\mathcal{H}$-shortcuts that support some maximal potential (i,t)-shortcut.

Consider a supporting potential shortcut $u \leftrightharpoons v$ (which may or may not be *stored*) and define $k_{u \leftrightharpoons v}$ to be the number of (i,t) pairs for which $u \leftrightharpoons v$ is covered by a maximal potential (i,t)-shortcut but *is not* covered by a *stored* (i,t)-shortcut.[13] Define a function f as follows.

$$f(u \leftrightharpoons v) = \begin{cases} k_{u \leftrightharpoons v}, & \text{if } u \leftrightharpoons v \text{ is not a fundamental shortcut,} \\ 0, & \text{if } u \leftrightharpoons v \text{ is a fundamental shortcut.} \end{cases}$$

Let C be the set of all shortcuts defined over $\mathcal{H}$, C_{st} be the set of all stored non-fundamental shortcuts, and C_{f} be the set of all stored fundamental shortcuts. The potential Φ is defined as

---

13   The count $k_{u \leftrightharpoons v}$ also takes the dummy tree into account, as if it had a special (i,t)-status. Notice that the dummy tree only exists in the middle of the `Delete` operation; see Section 8.2.

follows.

$$\Phi = \underbrace{\left(\sum_{u \leftrightharpoons v \in C} f(u \leftrightharpoons v)(\log \log n + 1) \right)}_{\Phi_1} + \underbrace{|C_{\mathrm{st}}| \cdot \log \log n}_{\Phi_2} + \underbrace{|C_{\mathrm{f}}| \cdot (\log \log n)^2}_{\Phi_3}$$

Uncovering a fundamental shortcut could possibly cause a detach-reattach operation in the local tree, which costs $O((\log \log n)^2)$ time; see the proof of Lemma 5.9 in Section 9.1. This is the reason that we give more credit to a stored fundamental shortcut than to a non-fundamental shortcut. Throughout the algorithm execution, there are many places where the (i, t)-forests are modified. These structural changes affect the potential Φ so we list them in the following paragraphs.

Adding (i, t)-status to an $\mathcal{H}$-leaf. (Lemma 8.1) Adding (i, t)-status to an $\mathcal{H}$-leaf increases Φ by $O(\log n (\log \log n)^2)$ since all new shortcuts that need to be created lie on the path from the leaf to its (i, t)-parent. In particular, each of the $O(\log n)$ new fundamental shortcuts increases Φ_3 by $(\log \log n)^2$ each, and both Φ_1 and Φ_2 increase by at most $O(\log n \log \log n)$ each.

Removing (i, t)-status from an $\mathcal{H}$-leaf. (Lemma 8.1) Removing (i, t)-status from a leaf x increases Φ by $O((\log \log n)^2)$. Let y be the (i, t)-parent of x. If y loses its (i, t)-status and its $\mathcal{H}$-parent z is no longer an (i, t)-branching node, we will create one new fundamental shortcut from z to a sibling of y, increasing Φ_3 by $(\log \log n)^2$. All new supporting potential (i, t)-shortcuts will cover z and have distinct powers. Thus, the net increase of Φ_1 will be at most $(\log \log n + 1) \log \log n$. Φ_2 is unchanged.

Creating a dummy tree. (Lemma 8.1) Create a dummy tree $\mathcal{T}$ by copying a maximally covered (i, t)-tree. Recall that there are $3d_{max} + 1$ shortcut forests, one for every (i, t)-pair and 1 for the dummy forest; we will say its shortcuts have $\perp$-*status*. After creating the dummy tree $\mathcal{T}$ and giving its maximal shortcuts $\perp$-status, there is no change to Φ. Every potential $\perp$-shortcut is a stored shortcut, and was formerly stored before $\mathcal{T}$ was created.

Removing (i, t)-status from a subset of $\mathcal{H}$-leaves. (Lemma 8.1) The data structure removes (i, t)-status (or $\perp$-status) from a subset of leaves in an (i, t)-tree $\mathcal{T}$ (or dummy tree $\mathcal{T}$). There are $O(|\mathcal{T}|)$ leaves removed, and each removal increases Φ by at most $O((\log \log n)^2)$, for a total of $O(|\mathcal{T}|(\log \log n)^2)$.

Merging and destroying dummy trees. (Lemma 8.1) The data structure merges a maximally covered dummy tree $\mathcal{T}$ into an (i', t')-tree, and destroys $\mathcal{T}$. Observe that in the process of merging these trees, the (i', t')-tree acquires new branching nodes and the set of supporting potential (i', t')-shortcuts only loses elements. Thus Φ_1 does not increase. Every shortcut sup-

porting the merged tree was in at least one of the two original trees before the operation, so Φ_2 and Φ_3 are also non-increasing.

Lazy Covering. (Lemma 5.7) The lazy covering method only covers non-fundamental short-cuts, so each covering costs constant actual time. Suppose we have traversed (i, t)-shortcuts $x \leftrightharpoons y$ and $y \leftrightharpoons z$ and covered them with $x \leftrightharpoons z$. (Notice that $x \leftrightharpoons z$ may or may not have been previously *stored*.) This causes $f(x \leftrightharpoons z)$ to drop by at least 1 and hence Φ_1 to drop by $\log \log n + 1$. If $x \leftrightharpoons z$ was not already stored, Φ_2 increases by $\log \log n$. In any case, the net potential drop in Φ is at least 1, which pays for the covering.

The `Delete` Operation. (See also Section 10.1.2) At the beginning of a `Delete`(u, v) operation, the algorithm spends $O(\log \log n)$ time to uncover each $\mathcal{H}$-shortcut touching an ancestor of u^i or v^i, where i is the depth of $\{u, v\}$. Notice that these $\mathcal{H}$-shortcuts may be shared by many (i', t')-pairs, so the uncovering operation may *temporarily* increase Φ_1 by $\Omega(\log^2 n \log \log n)$. Fortunately, after the deletion operation most of these $\mathcal{H}$-shortcuts are covered back. As mentioned in Section 5.2, after a deletion the data structure covers every possible supporting potential (i', t')-shortcut with both endpoints at ancestors of u^i or v^i, as well as all necessary fundamental $\mathcal{H}$-shortcuts with at least one endpoint ancestral to u^i or v^i. We claim that after covering back all necessary $\mathcal{H}$-shortcuts on the two paths, the increase of Φ is upper bounded by $O(\log n (\log \log n)^2)$. Counting multiplicity, there are $O(\log n \log \log n)$ non-fundamental deviating shortcuts that the lazy covering method failed to restore after the `Delete` operation. Each contributes $\log \log n + 1$ to Φ_1, for a total of $O(\log n (\log \log n)^2)$. The number of non-fundamental shortcuts with both endpoints at ancestors of u^i or v^i is $O(\log n)$, and each contributes $\log \log n$ to Φ_2, for a total of $O(\log n \log \log n)$. Similarly, the $O(\log n)$ fundamental shortcuts each contribute $(\log \log n)^2$ to Φ_3, for a total of $O(\log n (\log \log n)^2)$. The increase in Φ due to these changes are charged to the `Delete` operation.

10.   Main Operations — Proof of Lemma 3.1

We review how each of the 10 operations of Lemma 3.1 can be implemented in the stated amortized running time.

Operation (1) — Add or remove an edge with depth *i* and endpoint type *t*. The data structure first adds (or removes) the given edge to the $\mathcal{H}$-leaf data structures of its endpoints; see Section 4.1. If the addition/removal changes the (i, t)-status of either endpoint, we update them with Lemma 8.1 and if t = primary we update the approximate i-counters using Lemma 6.5. The time cost is $O(\log n (\log \log n)^2)$.

Operation (2) — Merge a subset of $\mathcal{H}$-siblings into $\mathbf{u^i}$ and promote all i-witness edges touching $\mathbf{u^i}$. Given the subset S of $\mathcal{H}$-siblings at depth i, the algorithm first uncovers all $\mathcal{H}$-shortcuts that touch any $\mathcal{H}$-siblings in S (Lemma 5.7). We then invoke Lemma 7.8 to merge $\mathcal{H}$-siblings in S, two at a time, into a single $\mathcal{H}$-node $\mathbf{u^i}$. The amortized cost for uncovering and deleting all $\mathcal{H}$-shortcuts touching S is zero. (The cost for *restoring* necessary shortcuts is not part of this operation. It is paid for by the Delete itself; see Section 9.2.) Thus, by Lemma 7.8, the amortized cost so far is $O(|S|(\log\log n)^2)$.

The algorithm then traverses the $(i, \text{witness})$-tree rooted at $\mathbf{u^i}$, obtains the set of leaf-descendants with $(i, \text{witness})$-status and enumerates the $|S| - 1$ $(i, \text{witness})$-edges touching these vertices. By Lemmas 5.7 and 7.8, the amortized cost of the traversal is $O(|S| \log\log n)$. Now the data structure uses Lemma 8.1 (last bullet point) to promote all these $(i, \text{witness})$-edges to $(i + 1, \text{witness})$-status, which costs $O((|S| - 1)(\log\log n)^2)$ time.

Notice that every edge releases $\Omega((\log\log n)^2)$ units of potential upon promotion. As every unit of potential pays for some constant $\Theta(1)$ running time, the amortized cost of this operation can be made $-\Omega((|S| - 1)(\log\log n)^2)$ by choosing a sufficiently large constant.

Operation (3) — Upgrade all i-secondary endpoints touching $\mathbf{u^i}$. The data structure first traverses the $(i, \text{secondary})$-tree rooted at $\mathbf{u^i}$, enumerating its leaf-set S. By Lemma 8.1, enumerating S costs $O(|S| \log\log n)$ time. Let $s \geq |S|$ be the number of $(i, \text{secondary})$-endpoints stored at these leaves. We then use Lemma 8.1 to add $(i, \text{primary})$-status and remove $(i, \text{secondary})$-status from all leaves in S, in $O(|S|(\log\log n)^2)$ amortized time. Using the $\mathcal{H}$-leaf data structure, we can upgrade all s $(i, \text{secondary})$-endpoints to $(i, \text{primary})$-status in $O(s)$ time. At this point the approximate i-counters at S are accurate, but the approximate i-counters at ancestors of S are out of date. Using Lemma 6.5, we rebuild all approximate i-counters at descendants of $\mathbf{u^i}$ in $O(p(\log\log n)^2)$ time, where $p \geq |S|$ is the number of $(i, \text{primary})$-leaves descending from $\mathbf{u^i}$.

The s upgrades release $\Omega(s(\log\log n)^2)$ units of potential whereas the cost for traversing the $(i, \text{primary})$-tree and updating its counters is $O(p(\log\log n)^2)$. Thus, the amortized time of this operation is $-\Omega((s - p)(\log\log n)^2)$.

Operation (4) — Promote a subset of i-primary endpoints touching $\mathbf{u^i}$. Let R be the set of $(i, \text{primary})$ endpoints being promoted. The data structure first scans through R, forming two leaf sets: S^- are all $\mathcal{H}$-leaves whose $(i, \text{primary})$-endpoints are contained in R (these will lose $(i, \text{primary})$-status) and S^+ are all $\mathcal{H}$-leaves touched by at least one element of R (these will gain $(i + 1, \text{secondary})$-status, if they do not have it already). Both S^- and S^+ are leaves of the $(i, \text{primary})$-tree $\mathcal{T}$ rooted at $\mathbf{u^i}$. The data structure uses Lemma 8.1 to add $(i + 1, \text{secondary})$-status to all $\mathcal{H}$-leaves in S^+ and removes $(i, \text{primary})$-status from all $\mathcal{H}$-leaves in S^-. By Lemma 8.1 the time cost is $O(|\mathcal{T}|(\log\log n)^2 + 1)$. Let p be the number of i-primary

endpoints touching $\mathbf{u^i}$, including the ones that are not promoted. Since $|\mathcal{T}| \leq p$ we have that this operation costs $O(p(\log \log n)^2 + 1)$ time.

Since the promotions release $|R| \cdot \Omega((\log \log n)^2)$ units of potential, with the leading constants set properly the amortized cost of this operation is at most $-\Omega((12|R| - p)(\log \log n)^2)$.

Operation (5) — Convert an *i*-non-witness edge to an *i*-witness edge. The data structure changes the status of the endpoints of the converted edge to $(i, \text{witness})$ using the $\mathcal{H}$-leaf data structure. If either endpoint of the edge had $(i, \text{primary})$-status prior to the conversion, the approximate i-counters at all ancestors of the $\mathcal{H}$-leaf containing the endpoing may be invalid and the endpoints may lose $(i, \text{primary})$-status. The data structure updates the approximate i-counters at all $(i, \text{primary})$-ancestors, and removes $(i, \text{primary})$-status of the endpoints, if necessary. This costs $O(\log n(\log \log n)^2)$ time, by Lemmas 6.5 and 8.1.

Operation (6) — Split an $\mathcal{H}$-node u^{i-1} with a single child u^i. We are given pointers to u^{i-2} (if it exists), u^{i-1}, and u^i. The data structure first creates a new $\mathcal{H}$-node x, detaches u^i from $\mathcal{L}(u^{i-1})$, and makes u^i a child of x using Lemma 7.8. If $i = 1$, then x is an $\mathcal{H}$-root and we are done. Otherwise, the data structure attaches x to $\mathcal{L}(u^{i-2})$. By Lemma 7.8, the amortized time for all these operations is $O((\log \log n)^2)$.

Operation (7) — Enumerate all (i, t)-endpoints in the (i, t)-tree rooted at $\mathbf{u^i}$. The data structure traverses the (i, t)-tree. For each (i, t)-leaf, enumerate all the endpoints of depth i and type t from the $\mathcal{H}$-leaf data structure. By applying the operations of Lemma 8.1, the time cost is $O(l \log \log n + k) = O(k \log \log n)$, where l is the number of (i, t)-leaves and k is the number of enumerated endpoints.

Operation (8) — Accessing $\mathcal{H}$-parent v^{i-1} from v^i. This is a local tree operation. According to Lemma 7.8, the time cost is $O(H(v^{i-1}) - H(v^i))$.

Operation (9) — Accessing an approximate *i*-counter. The approximate i-counter is stored at the node in floating-point representation. It can be retrieved and converted to an integer (Lemma 6.4) in $O(1)$ time.

Operation (10) — Batch Sampling Test. From Section 8.1, the batch sampling test on k samples costs worst case time $O(\min((p+s) \log \log n + k, k \log n \log \log n))$ where p is the number of i-primary edges touching $\mathbf{u^i}$ and s is the number of i-secondary edges touching $\mathbf{u^i}$.

10.1   Proof of Theorem 1.1

The correctness of the data structure follows from Section 3.2's maintenance of Invariant 2.2, using Lemma 3.1 to maintain $\mathcal{H}$ and Theorem 2.1 to maintain the witness forest $\mathcal{F}$. In this section

we prove that the amortized time complexity is $O(\log n (\log \log n)^2)$ per `Insert` or `Delete` and $O(\log n / \log \log \log n)$ per `Conn?` query. Call $\mathfrak{D}_{\mathcal{H}}$ the data structure for $\mathcal{H}$ described in Lemma 3.1 and $\mathfrak{D}_{\mathcal{F}}$ the data structure for $\mathcal{F}$ from Theorem 2.1, fixing $t(n) = (\log \log n)^2$.

10.1.1   Insertion

To execute `Insert`(u, v), the algorithm makes a connectivity query to $\mathfrak{D}_{\mathcal{F}}$ in $O(\log n / \log t(n)) = O(\log n / \log \log \log n)$ time. Then, there are two cases:

— If u and v are already connected, then the algorithm invokes Operation (1) of Lemma 3.1 on the data structure $\mathfrak{D}_{\mathcal{H}}$, adding the edge $\{u, v\}$ with depth 1 and endpoint type secondary in amortized $O(\log n (\log \log n)^2)$ time.

— Otherwise, u and v are not connected. The algorithm then invokes Operation (8) $2d_{max}$ times, obtaining pointers to u^0 and v^0. Thus, the cost of Operation (8) telescopes to $O(\log n \log \log n)$ time. The algorithm then merges u^0 and v^0 using Operation (2) in amortized $O((\log \log n)^2)$ time. Finally, $\{u, v\}$ is added to the data structure $\mathfrak{D}_{\mathcal{H}}$ through Operation (1) as an edge with depth 1 and type witness, in amortized $O(\log n (\log \log n)^2)$ time. The algorithm also inserts $\{u, v\}$ into $\mathfrak{D}_{\mathcal{F}}$, in $O(\log n \cdot t(n)) = O(\log n (\log \log n)^2)$ time.

Hence, an `Insert`(u, v) operation costs amortized $O(\log n (\log \log n)^2)$ time.

10.1.2   Deletion

To execute a `Delete`(u, v) operation, where $e = \{u, v\}$, the algorithm first removes e from $\mathcal{H}$ through Operation (1), taking amortized $O(\log n (\log \log n)^2)$ time. If e is a non-witness edge, then the operation is done. Otherwise, the algorithm also removes e from $\mathfrak{D}_{\mathcal{F}}$ in $O(\log n \cdot t(n))$ time. Then, the algorithm attempts to find a replacement edge iteratively at depth $i = d_e, d_e - 1, \ldots, 1$.

Preparing Iterations. As mentioned in Section 5.2, before the iterations begin, all ancestors of $u^{i-1} = v^{i-1}$ are found and stored in a list, using Operation (8). The cost of Operation (8) telescopes to $O(\log n \log \log n)$ time. In addition, all stored $\mathcal{H}$-shortcuts touching the path from u^{i-1} to u^0 are uncovered, using Lemma 5.9. We note that Invariant 5.3 now holds only for all $\mathcal{H}$-nodes at depth $\geq i$, which validates all operations whose implementation depends on Lemma 8.1. Once the shortcuts have been removed, the iterations begin.

Establishing Two Components. On the iteration concerning depth i, the algorithm runs two parallel searches starting from u^i and v^i, obtaining the connected components c_u and c_v. Throughout the search, $\mathcal{H}$-siblings of u^i and v^i are found via i-witness edges enumerated by Operation (7). Let S_u be the set of $\mathcal{H}$-siblings in the same component c_u with u^i and S_v be the of $\mathcal{H}$-siblings for c_v with v^i. Notice that there are exactly $|S_u| - 1$ and $|S_v| - 1$ i-witness edges in c_u and c_v respectively, and each i-witness edge contributes 2 endpoints throughout the search.

Thus, the searches in parallel take amortized $O(\min\{|S_u|-1, |S_v|-1\}(\log\log n)+1)$ time until the first completes. At this point we can deduce which of c_u or c_v is the smaller weight component; suppose it is c_u.

The algorithm uncovers and removes all remaining downward shortcuts on the siblings of u^i that form c_u (Lemma 5.9), then performs Operation (2) to promote all $(i, \text{witness})$-edges in c_u to $(i+1, \text{witness})$ edges, with a negative amortized cost of $-\Omega((|S_u|-1)(\log\log n)^2)$, which pays for the cost of the two searches.

In conclusion, establishing two components costs amortized constant time.

Finding a Replacement Edge. Recall from Section 8.1 that ρ is the fraction of i-primary endpoints belonging to replacement edges and p and s are the number of primary and secondary endpoints. When $\rho > 3/4$ the search for a replacement edge halts after the first or second batch sampling test with probability $1 - 1/p$, and costs $O(\log n(\log\log n)^2)$ in expectation, which is charged to the Delete operation. Suppose that the enumeration procedure is invoked, which upgrades *all* $(i, \text{secondary})$ endpoints to $(i, \text{primary})$ status (Operation (3)), and then *some* of the $(i, \text{primary})$ endpoints to $(i+1, \text{secondary})$ status (Operation (4)). This procedure costs $O((p+s)\log\log n)$ time. The amortized time cost of Operation (3) is $-\Omega((s-p)(\log\log n)^2)$. At this point there are now $p' = p+s$ $(i, \text{primary})$ endpoints. Suppose that Operation (4) promotes s' of them to $(i+1, \text{secondary})$ status, at an amortized time cost of $-\Omega((12s'-p')(\log\log n)^2) = -\Omega((12(1-\rho)p-(p+s))(\log\log n)^2)$. (If $s' < p'$, then all the unpromoted endpoints belong to replacement edges.) Let the leading constants of the amortized costs of Operations (3) and (4) be c_0 and c_1 times that of the cost of the enumeration procedure. Then the amortized time cost of the enumeration procedure is proportional to

$$\left(\log\log n\right)^2 \left((p+s) - c_0(s-p) - c_1(12(1-\rho)p - (p+s))\right)$$
$$= \left(\log\log n\right)^2 \left(p(1 + c_0 - c_1(12(1-\rho) - 1)) + s(1 - c_0 + c_1)\right)$$

When $\rho < 3/4$, the contribution of original primary endpoints (p) is at most $p(1 + c_0 - 2c_1)(\log\log n)^2$, which is at most 0 when $c_1 \geq (1 + c_0)/2$. When $\rho > 3/4$ the enumeration procedure is invoked with probability at most $1/p$, and the expected time cost is $O((\log\log n)^2)$. Regardless of ρ, the contribution of original secondary endpoints is $s(1 - c_0 + c_1)(\log\log n)^2$, which is at most 0 when $c_0 \geq c_1 + 1$. Setting $c_0 = 3$ and $c_1 = 2$ satisfies both constraints.

In conclusion, *successfully* finding a replacement edge in the first or second batch sampling test costs $O(\log n(\log\log n)^2)$ expected time, which is charged to the Delete operation. If the enumeration procedure is invoked, then the search for a replacement edge may fail to find a replacement edge at level i. The amortized expected cost of the enumeration procedure at depth i is $O((\log\log n)^2)$, which is charged to the Delete operation.

Preparation for Next Iteration. If no replacement edge is found at the current depth i, the algorithm splits u^{i-1} into to $\mathcal{H}$-siblings $\mathbf{u^{i-1}}$ and $\mathbf{v^{i-1}}$, through Operation (6). The split operation costs amortized $O((\log \log n)^2)$ time. After the split, the algorithm restores all necessary downward shortcuts touching $\mathbf{u^{i-1}}$, $\mathbf{v^{i-1}}$, v^i, or $\mathbf{u^i}$, as described in Section 5.2 and Lemma 5.9. The covering of fundamental shortcuts ensures Invariant 5.3 to hold for all $\mathcal{H}$-nodes at depth $\geq i - 1$. By the same argument from Lemma 5.9, the total cost of covering these shortcuts is $O(\log n(\log \log n)^2)$, which is charged to the Delete operation.

The End of Iteration. Suppose we find a replacement edge at depth i. The algorithm ends by restoring all necessary shortcuts with one endpoint at an ancestor of u^i or v^i. By Lemma 5.9, this costs $O(\log n(\log \log n)^2)$ time. Furthermore, this restores Invariant 5.3 holds for all nodes in $\mathcal{H}$.

Combining the Costs. Summing all costs, the total amortized expected time for an edge deletion is $O(\log n(\log \log n)^2)$.

11. Conclusion

We have shown that the Las Vegas randomized amortized update time of dynamic connectivity is $O(\log n(\log \log n)^2)$, which leaves a small $(\log \log n)^2$ gap between the cell probe lower bounds of Pătraşcu and Demaine [1] and Pătraşcu and Thorup [2]. The main bottleneck in our approach is dealing with insertions in the buffer trees inside local trees. Each affects $O(\log \log n)$ local tree nodes, and the cost of updating such nodes involves adding $O(\log n)$ (floating point) approximate counters packed into $O(\log \log n)$ machine words. If this $(\log \log n)^2$ barrier were overcome, there would still be a $\log \log n$-factor bottleneck, which arises from the shortcut infrastructure and the height of the bottom, buffer, and top trees.

It may be possible to achieve $O(\log n)$ amortized time in the Monte Carlo model with a private connectivity witness, by using connectivity sketches [3, 14, 9, 21, 18].

References

[1] M. Pătraşcu and E. Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM J. Comput.* 35(4):932–963, 2006 (3, 4, 55).

[2] M. Pătraşcu and M. Thorup. Don't rush into a union: take time to find your roots. *Proceedings of the 43rd ACM Symposium on Theory of Computing (STOC)*, pages 559–568, 2011 (3, 55).

[3] K. J. Ahn, S. Guha, and A. McGregor. Analyzing graph structure via linear measurements. *Proceedings 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 459–467, 2012 DOI (55).

[4] S. Alstrup, J. Holm, K. de Lichtenberg, and M. Thorup. Maintaining information in fully dynamic trees with top trees. *ACM Trans. Algorithms*, 1(2):243–264, 2005 DOI (28).

[5] J. Chuzhoy, Y. Gao, J. Li, D. Nanongkai, R. Peng, and T. Saranurak. A deterministic algorithm for balanced cut with applications to dynamic connectivity, flows, and beyond. *Proceedings 61st IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1158–1167, 2020 DOI (2).

[6] 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).

[7] 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).

[8] G. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. Comput.* 14(4):781–798, 1985 (2).

[9] 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 (2, 55).

[10] M. R. Henzinger and V. King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *J. ACM*, 46(4):502–516, July 1999 DOI (3, 4).

[11] 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 (3, 9).

[12] 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 DOI (3, 4, 6).

[13] S.-E. Huang, D. Huang, T. Kopelowitz, and S. Pettie. Fully dynamic connectivity in $O(\log n(\log \log n)^2)$ amortized expected time. *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 510–520, 2017 DOI (1, 3).

[14] B. M. Kapron, V. King, and B. Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. *Proceedings of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1131–1142, 2013 (2, 55).

[15] C. Kejlberg-Rasmussen, T. Kopelowitz, S. Pettie, and M. Thorup. Faster worst case deterministic dynamic connectivity. *Proceedings 24th European Symposium on Algorithms (ESA)*, 53:1–53:15, 2016 (2).

[16] D. Nanongkai and T. Saranurak. Dynamic spanning forest with worst-case update time: adaptive, las vegas, and $O(n^{1/2-\epsilon})$-time. *Proceedings of the 49th Annual ACM Symposium on Theory of Computing (STOC)*, pages 1122–1129, 2017 DOI .

[17] D. Nanongkai, T. Saranurak, and C. Wulff-Nilsen. Dynamic minimum spanning forest with subpolynomial worst-case update time. *Proceedings 58th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 950–961, 2017 (2).

[18] J. Nelson and H. Yu. Optimal lower bounds for distributed and streaming spanning forest computation. *Proceedings 30th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1844–1860, 2019 DOI (55).

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

[20] M. Thorup. Near-optimal fully-dynamic graph connectivity. *Proceedings of the Thirty-second Annual ACM Symposium on Theory of Computing (STOC)*, pages 343–350, 2000 DOI (1, 3, 4).

[21] Z. Wang. An improved randomized data structure for dynamic graph connectivity. *CoRR*, abs/1510.04590, 2015 (2, 55).

[22] C. Wulff-Nilsen. Faster deterministic fully-dynamic graph connectivity. *Proceedings of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1757–1769, 2013 (3, 4, 6).

[23] C. Wulff-Nilsen. Fully-dynamic minimum spanning forest with improved worst-case update time. *Proceedings 49th Annual ACM Symposium on Theory of Computing (STOC)*, pages 1130–1143, 2017 DOI .