A Data Structure for Nearest Common Ancestors with Linking

HAROLD N. GABOW, University of Colorado at Boulder

45

Consider a forest that evolves via *link* operations that make the root of one tree the child of a node in another tree. Intermixed with *link* operations are *nca* operations, which return the nearest common ancestor of two given nodes when such exists. This article shows that a sequence of m such *nca* and *link* operations on a forest of n nodes can be processed online in time $O(m\alpha(m, n) + n)$. This was previously known only for a restricted type of *link* operation.

The special case where a *link* only extends a tree by adding a new leaf occurs in Edmonds' algorithm for finding a maximum weight matching on a general graph. Incorporating our algorithm into the implementation of Edmonds' algorithm in [9] achieves time $O(n(m + n \log n))$ for weighted matching, an arguably optimum asymptotic bound (n and m are the number of vertices and edges, respectively). Our data structure also provides a simple alternative implementation of the incremental-tree set merging algorithm of Gabow and Tarjan [11].

CCS Concepts: • **Theory of computation** → **Data structures design and analysis**; *Graph algorithms analysis*; • **Mathematics of computing** → *Trees*;

Additional Key Words and Phrases: Least common ancestors, matching algorithms, set merging, disjoint-set data structure

ACM Reference format:

Harold N. Gabow. 2017. A Data Structure for Nearest Common Ancestors with Linking. *ACM Trans. Algorithms* 13, 4, Article 45 (September 2017), 28 pages.
<https://doi.org/10.1145/3108240>

1 INTRODUCTION

Finding nearest common ancestors (*ncas*) is a basic data structure operation. It is used to solve many string problems via the suffix tree [4, 12, 18]. It arises in trees that grow by adding leaves, in the efficient implementation of algorithms for weighted matching and its generalizations [8, 9]. This application is the jumping-off point for this article.

We summarize our algorithms for dynamic *ncas* starting with the most general version of the problem. Recall that the *nearest common ancestor* of two nodes x and y in a rooted tree is the ancestor of x and y that has greatest depth. Consider a rooted forest F that is subject to two operations. In both operations, x and y are nodes of F :

link(x, y) – make y a child of x , where y is the root of a tree not containing x ;

Research supported in part by NSF Grant No. CCR-8815636.

A preliminary version of results in this article was presented at the First Annual ACM-SIAM Symposium on Discrete Algorithms [8].

Author's address: H. N. Gabow, Department of Computer Science, University of Colorado at Boulder, Boulder, Colorado 80309-0430, USA; email: hal@cs.colorado.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2017 ACM 1549-6325/2017/09-ART45 \$15.00

<https://doi.org/10.1145/3108240>

$nca(x, y)$ – return the nearest common ancestor of x and y ;
 if x and y are in different trees, return ϵ .

The problem of *nearest common ancestors with linking* is to process (online) an arbitrary sequence of m *link* and *nca* operations starting from an initial forest of n nodes ($m, n \geq 1$).

A number of data structures have been proposed for this problem and various special cases. See [13] for the early history, which dates back to the offline algorithm of Aho, Hopcroft, and Ullman [1]. The problem of *nearest common ancestors for static trees* is when F is given initially (equivalently, all *links* precede all *ncas*). Harel and Tarjan give an algorithm that answers each *nca* query in time $O(1)$, after $O(n)$ time to preprocess F [13]. Several algorithms that achieve the same asymptotic bounds but appear to be simpler have also been given (e.g., [15, 3, 2]). Harel and Tarjan also give an algorithm for the case where *link* and *nca* operations are intermixed but where both arguments of every *link* are roots [13]. The running time is $O(m\alpha(m, n) + n)$. In this article, we remove the restriction on links: We show that the general nearest common ancestors with linking problem can be solved in time $O(m\alpha(m, n) + n)$ and space $O(m + n)$. The previous best solution uses dynamic trees [17]. This data structure performs each operation in time $O(\log n)$, achieving total time $O(m \log n + n)$. This is not as fast as our algorithm, although dynamic trees have the advantage that they can process *cut* operations.

Dynamic *ncas* arise in Edmonds' algorithm for weighted matching on graphs [6, 9, 16]. The algorithm is based on the notion of "blossom," a type of subgraph that can be contracted in the search for an augmenting path. Blossoms get contracted using a trivial *nca* computation in a search graph. But only the "cheapest blossom" can be contracted in Edmonds' algorithm. An efficient matching algorithm must track all possible blossoms and contract only the best. This necessitates answering an *nca* query for every possible blossom. Furthermore, the query is made in the search graph, a forest that grows by adding new leaves as the search progresses.

To be precise, consider the following operations on a rooted tree T . x is a node already in T , and y is a new node not yet in T :

add_leaf (x, y) – add a new leaf y , with parent x , to T ;
add_root (y) – make the current root of T a child of new root y .

The problem of *incremental-tree nearest common ancestors* is to process (online) an arbitrary sequence of n *add_leaf* and *add_root* operations intermixed with m *nca* operations, starting with an empty tree (more precisely, T starts with one node, an artificial root). Edmonds' algorithm actually only uses *add_leaf* and *nca* operations. We give an algorithm that solves this problem in $O(m + n \log n)$ time. This achieves the desired time bound for Edmonds' algorithm: It finds a maximum weight matching in time $O(n(m + \log n))$ [8, 9].¹

We refine our *nca* algorithm to use time $O(m + n)$. This incremental-tree algorithm becomes the starting point of our algorithm for general *links*.

A part of our data structure can be used to implement the incremental-tree set merging algorithm of Gabow and Tarjan [11]. The new implementation does not change the asymptotic performance, but it uses simpler primitive operations that must be precomputed (specifically, most-significant-bit, powers of two, and logical *and*), which might afford advantages in practice.

After the conference version of this article Cole and Hariharan presented a more powerful algorithm for the incremental-tree *nca* problem. Our algorithm answers each *nca* query in worst-case time $O(1)$, but *add* operations are $O(1)$ only in an amortized sense. Cole and Hariharan achieve worst-case time $O(1)$ for all operations, even allowing insertion of internal nodes and deletion

¹This time bound is optimal in an appropriate model of computation [8, 9]. See [9] for a survey of previous work on weighted matching.

of nodes with ≤ 1 child [4]. The starting point of their algorithm is a version of our approach presented in Section 2.²

The model of computation throughout this article is a random access machine with a word size of $\log n$ bits. Harel and Tarjan [13] give a lower bound indicating that it is unlikely that our results for nearest common ancestors can be achieved on a pointer machine. However, it still might be possible to achieve our results for Edmonds' algorithm on a pointer machine.

The rest of the article is organized as follows. This section concludes with some terminology. Section 2 introduces the basic idea, a generalization of preorder numbering of trees. It solves the incremental-tree *nca* problem in $O(m + n \log^2 n)$ time. Section 3 improves the time bound to $O(m + n \log n)$. This is all that is needed to complete the implementation of Edmonds' search algorithm in time $O(m + n \log n)$. (Readers interested only in the application to matching need go no further.) The approach of Section 3 is extended in the next two sections: Section 4 achieves linear time for the incremental-tree *nca* problem. Section 5 achieves time $O(m\alpha(m, n) + n)$ to process m *nca* and *link* operations on a set of n nodes.

Some further details are presented in appendices: Appendix A shows how to compute the logarithms needed in Section 2. Appendix B proves a simple lemma for dynamic space allocation. Appendix C gives our algorithm for incremental-tree set merging. Appendix D proves simple properties of Ackermann's function used in Section 5. Appendix E extends our algorithm for *ncas* and *links* (Section 5) to allow *make_node* operations (i.e., the number of nodes is not known in advance).

Terminology. We use interval notation for sets of integers: for $i, j \in \mathbb{Z}$, $[i..j] = \{k \in \mathbb{Z} : i \leq k \leq j\}$. $\log n$ denotes logarithm to the base two.

As usual, we assume a RAM machine does truncating integer division. We compare two rational numbers a/b and c/d for positive integers a, b, c, d by comparing ad and bc . Assume that for a given integer $r \in [1..n]$ the value $\lfloor \log r \rfloor$ can be computed in $O(1)$ time. This can be done if we precompute these n values and store them in a table. The precomputation time is $O(n)$. More generally, for a fixed rational number $\beta > 1$ $\lfloor \log_\beta r \rfloor$ can be computed in $O(1)$ time (see Appendix A).

We use the following tree terminology. Let T be a tree. $V(T)$ denotes its vertex set. A *subtree* of T is a connected subgraph. The root of T is denoted $\rho(T)$. Let v be a node of T . The *ancestors* of v are the nodes on the path from v to $\rho(T)$. The ancestors are ordered as in this path. This indicates how to interpret expressions like "the first ancestor of v such that." The *descendants* of v are all nodes that have v as an ancestor. T_v denotes the subtree of all descendants of v . The *parent* of v is denoted $\pi(v)$. For any function f defined on nodes of a tree, we write f_T when the tree T is not clear from context (e.g., $\pi_T(v)$).

2 FAT PREORDER FOR DYNAMIC TREES

This section introduces the basic idea for dynamic trees, the fat preorder numbering that generalizes preorder numbering of trees. It starts with an algorithm to find nearest common ancestors on a tree that is given in advance. Then it extends that algorithm to solve the incremental-tree *nca* problem in $O(m + n \log^2 n)$ time.

Our main auxiliary tree is the compressed tree, so we begin by reviewing the basic definitions [13, 19]. Let T be a tree with root $\rho(T)$. The *size* $s(v)$ of a node v is the number of its descendants. (As usual, a node is a descendant of itself.) A child w of v is *light* if $s(w) \leq s(v)/2$; otherwise, it is *heavy*. Deleting each edge from a light child to its parent partitions the nodes of T into paths of nonnegative length, called the *heavy paths* of T . (Thus, an isolated node is considered a heavy

²The *reorganize* operation defined in Section 2 makes our time bound for *adds* amortized. Cole and Hariharan do reorganizations piecemeal in future operations, so each operation does constant work [4].

Fig. 1. Fat preorder interval for vertex v . The two empty intervals guard the preorder numbers of the vertices of T_v , which are in $[p(v), q(v))$.

path, and a heavy path has at most one node at each depth.) A node is an *apex* if it is not a heavy child (e.g., $\rho(T)$). Equivalently, an apex is the highest node on its heavy path.

To generalize this, consider an arbitrary partition of $V(T)$ into a family $\mathcal{P}$ of disjoint paths of nonnegative length. Length 0 paths are allowed, and we require that each path has at most one node at each depth. Call the highest node on each path its apex. The *compressed tree for T and $\mathcal{P}$* , $C(T, \mathcal{P})$, has nodes $V(T)$; its root is $\rho(T)$ and the parent of a node $v \neq \rho(T)$ is the first proper ancestor of v that is an apex. Any apex v has the same descendants in $C(T, \mathcal{P})$ and T , so in particular $s_{C(T, \mathcal{P})}(v) = s_T(v)$. When $\mathcal{P}$ consists of the heavy paths of T , we call $C(T, \mathcal{P})$ the *compressed tree (for T)*, denoted $C(T)$. As extreme examples, $C(T)$ is T if T is a complete binary tree, and $C(T)$ has height 1 if T is a path of one or more edges. (In a complete binary tree, every node is an apex; in a path the apexes are the two ends.)

Let T be an arbitrary tree and let C be its compressed tree $C(T)$. C has height $\leq \lfloor \log n \rfloor$. This follows from the fact that in C , the parent v of node w has $s_C(v) \geq 2s_C(w)$. (This is clear if w is a light child in T . If w is a heavy child then $s_C(w) = 1$ and $s_C(v) \geq 2$.)

For any nodes x, y in the given tree T , we compute $nca_T(x, y)$ by starting with the corresponding node $nca_C(x, y)$ in the compressed tree C . Harel and Tarjan [13] compute $nca_C(x, y)$ in $O(1)$ time by embedding C in a complete binary tree B ; $ncas$ in B are calculated using the binary expansion of the inorder numbers of the nodes. Schieber and Vishkin [15] use a similar approach. We now present a different strategy that seems to give simpler algorithms (see Section 3).

For simplicity, we begin by just discussing the algorithm for nca_C . We then extend that algorithm to also find the closely related “characteristic ancestors” (see later discussion).

Our main tool, the fat preordering, is defined for special trees C that generalize the compressed tree. Choose a real-valued constant $\beta > 1$ and integers $e > 1, c > 2$ such that

$$\frac{2}{\beta^{e-1} - 1} \leq c - 2 \leq \beta^e. \quad (1)$$

Note that the left inequality is satisfied when $\beta^{e-1} \geq 2$ and $c \geq 4$, so a convenient choice is $e = \beta = 2$ and $c = 4$. Let C be equipped with a function³ $\sigma : V(C) \rightarrow [1..n]$ such that every node v with child w satisfies

$$\sigma(v) \geq \beta \sigma(w). \quad (2)$$

For functions $p, q, \bar{p}, \bar{q} : V(C) \rightarrow \mathbb{Z}_+$, p is a *fat preorder numbering* of C if, for any node v , as illustrated in Figure 1,

- (i) the descendants of v in C are the nodes w with $p(w) \in [p(v), q(v))$;
- (ii) no node w has $p(w) \in [\bar{p}(v), p(v)) \cup [q(v), \bar{q}(v))$;
- (iii) $\bar{q}(v) - \bar{p}(v) = c\sigma(v)^e$ and $p(v) - \bar{p}(v) = \bar{q}(v) - q(v) = \sigma(v)^e$.

³In this definition, n is not necessarily equal to $|V(C)|$. Eventually, we will have n either equal to $|V(C)|$ with $\sigma = s_C$, $\beta = 2$ or $n \leq |V(C)|$ and σ a variant of s_C .

Note that (i) is equivalent to p being a preorder numbering. Also, the definition allows “guarding intervals” to overlap; for example, we may have $\bar{q}(w) \in [p(v), q(v))$ for w not descending from v . However, our algorithms will maintain the intervals $[\bar{p}(v), \bar{q}(v))$ as a laminar family. Without loss of generality, we can assume $\bar{p}(\rho(C)) = 0$, so $\bar{q}(\rho(C)) \leq cn^e$ and all p -numbers are in $[0, cn^e)$. The fat preorders that we construct take σ to be the size function s_C (for static trees) or a close variant of s_C (for dynamic trees).

Given a fat preordering, the following high-level algorithm returns $nca_C(x, y)$:

Let b be the first ancestor of x that has

$$(c - 2)\sigma(b)^e > |p(x) - p(y)|.$$

If b is an ancestor of y (i.e., $p(b) \leq p(y) < q(b)$), then return b ; else return $\pi_C(b)$ (the parent of b).

LEMMA 2.1. *The nca_C algorithm is correct.*

PROOF. We first show that any common ancestor d of x and y satisfies

$$(c - 2)\sigma(d)^e > |p(x) - p(y)|.$$

By (i) the interval $[p(d), q(d))$ contains both $p(x)$ and $p(y)$, so its length is at least $1 + |p(x) - p(y)|$. By (iii), its length is $(c - 2)\sigma(d)^e$. The preceding inequality follows.

Now, to prove the lemma we need only show that $b' = \pi_C(b)$ is a common ancestor. (Clearly, we can assume b is not the root.) This amounts to showing b' is an ancestor of y . By (i) – (iii), a nondescendant of b' and a descendant of b' differ in p -number by more than $\sigma(b')^e$. Using Equation (2) and the right inequality of Equation (1), $\sigma(b')^e \geq \beta^e \sigma(b)^e \geq (c - 2)\sigma(b)^e > |p(x) - p(y)|$. Since x descends from b' , so must y . $\square$

Before implementing the high-level algorithm, we note that computing nca s using the compressed tree (or using other auxiliary trees that we shall see) requires more than just the nca node. Fix any tree and consider nodes x, y . Let $a = nca(x, y)$. For $z = x, y$, let a_z be the ancestor of z immediately preceding a ; if $a = z$ then take $a_z = a$. Define $ca(x, y)$, the *characteristic ancestors* of x and y , as the ordered triplet (a, a_x, a_y) . Our nca algorithms actually compute ca .

We will use the following data structure. Every vertex x stores an *ancestor table*, $ancestor_x[0..[\log_\beta cn^e]]$, where

$$ancestor_x[i] \text{ is the last ancestor } b \text{ of } x \text{ that has } (c - 2)\sigma(b)^e < \beta^i.$$

If no such ancestor exists (i.e., $(c - 2)\sigma(x)^e \geq \beta^i$), then the entry is ϵ .

Figure 2 gives the core of the algorithm for characteristic ancestors. As in the high-level nca_C algorithm, it computes the node b and converts b to the desired nearest common ancestor a . It also computes the characteristic ancestor a_x , using an auxiliary node b_x . We omit the computation of a_y , which is symmetric to a_x . We write ca_C to refer to the entire algorithm that uses Figure 2 to compute the entire tuple (a, a_x, a_y) . C continues to refer to an arbitrary tree that has a fat preorder.

LEMMA 2.2. *The ca_C algorithm is correct and uses $O(1)$ time.*

PROOF. We start by showing that the first two lines of Figure 2 make

$$w = \text{the first ancestor of } x \text{ that has } (c - 2)\sigma(w)^e \geq \beta^i. \quad (3)$$

The assignment to i makes $\beta^i \leq |p(x) - p(y)|$. Thus, the inequality of Equation (3) is weaker than the definition of b in nca_C , so w exists. Consider two cases for w , first $v \neq \epsilon$. The definition of $ancestor_x[i]$ shows that v is the last ancestor of x with $(c - 2)\sigma(v)^e < \beta^i$. Thus, $w = \pi(v)$ satisfies Equation (3). Next, suppose $v = \epsilon$. The definition of $ancestor_x[i]$ shows $w = x$ satisfies Equation (3).

```

 $i \leftarrow \lfloor \log_{\beta} |p(x) - p(y)| \rfloor; \quad v \leftarrow \text{ancestor}_x[i]$ 
 $w \leftarrow \text{if } v \neq \epsilon \text{ then } \pi(v) \text{ else } x$ 
if  $(c - 2)\sigma(w)^e > |p(x) - p(y)|$  then
     $b \leftarrow w; \quad b_x \leftarrow \text{if } v \neq \epsilon \text{ then } v \text{ else } x$ 
else
     $b \leftarrow \pi(w); \quad b_x \leftarrow w$ 
if  $b$  is an ancestor of  $y$  then
     $a \leftarrow b; \quad a_x \leftarrow b_x$ 
else
     $a \leftarrow \pi(b); \quad a_x \leftarrow b$ 

```

Fig. 2. The abbreviated ca_C algorithm.

As mentioned, Equation (3) implies that b is an ancestor of w (possibly $b = w$). We next show that b is either w or its parent. Clearly, this holds if $w = \rho(C)$. Suppose $w \neq \rho(C)$. For $w' = \pi(w)$, Equation (2) and $e \geq 1$ show $(c - 2)\sigma(w')^e \geq (c - 2)\beta^e \sigma(w)^e \geq \beta^{i+1}$. Since $\beta^{i+1} > |p(x) - p(y)|$, w' satisfies the inequality (of the nca_C algorithm) that defines b . So b is either w or w' .

This implies the test on $\sigma(w)$ in Figure 2 correctly sets b to w or $\pi(w)$. The test on b shows a is set correctly (according to the nca_C algorithm). It remains to analyze a_x . Examining the code shows the following:

When $a = \pi(b)$, a_x is set correctly to b , so assume $a = b$ and $a_x = b_x$.

When $b = \pi(w)$, a_x is set correctly to w , so assume $b = w$.

If $v \neq \epsilon$, a_x is set correctly, since $b = w = \pi(v)$ and $b_x = v$. So assume $v = \epsilon$.

The code sets $w = x$ and $b_x = x$. So the algorithm now has $a = b = w = x$ and $a_x = x$. Clearly this is correct.

The time bound of $O(1)$ easily follows. (The value i is computed as in Appendix A.) $\square$

Now let C be a tree on n nodes satisfying Equation (2) for $\sigma = s = s_C$. (An example is the compressed tree, with $\beta = 2$.) We show that a fat preordering of C exists and can be constructed in $O(n)$ time. We use a recursive numbering procedure. It traverses C top-down. When visiting a node u , u will have already been assigned an interval $[\bar{p}(u), \bar{q}(u))$ with $\bar{q}(u) - \bar{p}(u) = cs(u)^e$. Initially assign $\rho(C)$ the interval $[0, cn^e)$. Each child of u will get the leftmost possible interval (i.e., the intervals of u 's children will form a partition of an interval beginning at $\bar{p}(u) + 1$). To visit u , execute the following procedure:

Assign $p(u) \leftarrow \bar{p}(u) + s(u)^e$ and $q(u) \leftarrow \bar{q}(u) - s(u)^e$. Then assign intervals to the children of u , starting at $p(u) + 1$, as follows:
 For each child v of u , assign the next interval in $[p(u), q(u))$ of length $cs(v)^e$ to v , and then visit v .

LEMMA 2.3. *The numbering algorithm gives a valid fat preordering with $\sigma = s_C$ when C is a tree satisfying Equation (2) for $\sigma = s_C$. The time is $O(n)$ for $n = |V(C)|$.*

PROOF. It is clear that the algorithm achieves properties (i) – (iii) of fat preorder, and it runs in $O(n)$ time. We must show that the intervals assigned by u all fit into the interval $[\bar{p}(u), \bar{q}(u))$ given to u . For u a leaf this holds, since the interval's size is $c \geq 3$. Assume u is an interior node and let U denote the set of children of u . Starting with the relation $s(u) = 1 + \sum_{v \in U} s(v)$, multiply by $s(u)^{e-1}$ and use Equation (2) (and its consequence $s(u) \geq \beta$) to obtain $s(u)^e \geq \beta^{e-1} + \beta^{e-1} \sum_{v \in U} s(v)^e$. This implies $cs(u)^e / \beta^{e-1} \geq 1 + \sum_{v \in U} cs(v)^e$. The right-hand side is the total size of intervals assigned in $[p(u), q(u))$ (the term 1 accounts for the number $p(u)$). Since $[p(u), q(u))$ has length $(c - 2)s(u)^e$, it suffices to have $c - 2 \geq c / \beta^{e-1}$. This is equivalent to the left inequality of Equation (1). $\square$

After numbering the tree in fat preorder, we construct the ancestor tables: For every node x we find successive entries of $ancestor_x[i]$ by traversing the path from x to $\rho(C)$. The time and space for this is $O(n \log n)$ and dominates the entire algorithm.

The last important step in the nca algorithm is a procedure (due to [13]) that computes characteristic ancestors in T from those in the compressed tree $C(T)$. To state it, let $C = C(T, \mathcal{P})$ for an arbitrary set of paths $\mathcal{P}$. Suppose the characteristic ancestors $ca_C(x, y) = (c, c_x, c_y)$ are known and we seek the characteristic ancestors $ca_T(x, y) = (a, a_x, a_y)$.

Let P be the path of $\mathcal{P}$ with apex c . The definition of C implies that $a = nca_T(x, y)$ is the first common ancestor of x and y on P . For $z \in \{x, y\}$, let b_z denote the first ancestor of c_z on P (i.e., b_z is c_z or $\pi_T(c_z)$). Then a is the shallower vertex of b_x and b_y .

Next, we show how to find a_x (the same procedure applies to a_y). Consider three cases.

CASE $a \neq b_x$: This makes the predecessor a^- of a on P an ancestor of b_x (possibly $a^- = b_x$). Thus, a^- is an ancestor of x . Clearly, this makes $a_x = a^-$.

CASE $a = b_x \neq c_x$: This makes $b_x = \pi_T(c_x)$. Combining equations gives $a = b_x = \pi_T(c_x)$. So, by definition, $a_x = c_x$.

CASE $a = b_x = c_x$: We will show

$$c_x = x.$$

Combining this with the case definition gives $a = x$. So the definition of ca_T shows $a_x = x$. (This also makes $a_x = c_x$, as in the previous case.)

If c_x is a leaf of C , the displayed equation follows since c_x is an ancestor of x . If c_x is not a leaf of C , then c_x is the unique vertex of P that is nonleaf (i.e., $c_x = c$). The definition of ca_C shows $x = c$. Combining gives $c_x = c = x$.

Putting these pieces together gives our algorithm for nca queries on a static tree. Let us summarize the algorithm. A preprocessing step computes the compressed tree $C = C(T)$. It is numbered in fat preorder ($\beta = 2$). In addition, the order of nodes in each heavy path is recorded (so we can find a^- in the first case described earlier). The ancestor tables for C are constructed. The query algorithm computes characteristic ancestors by finding $ca_C(x, y)$ and using it to find $ca_T(x, y)$.

LEMMA 2.4. *A tree T with n nodes can be preprocessed using $O(n \log n)$ time and space so that ca queries can be answered in $O(1)$ time.*

Note that the preprocessing time and the space are both $O(n)$ except for the resources needed to compute and store the ancestor tables.

Incremental Trees

We extend these ideas to trees that grow by *add_leaf* operations. It is then easy to complete the incremental tree data structure by incorporating *add_root* operations.

We start by presenting the high-level strategy. Then we give the data structure and detailed algorithm, and, finally, we prove that it works correctly.

We use a dynamic version D of the compressed tree, maintaining a fat preordering and computing cas as before. In more detail, D is maintained to be $C(T, \mathcal{P})$ for a time-varying collection of paths $\mathcal{P}$ that always partitions $V(T)$. *add_leaf* makes the new leaf a singleton path of $\mathcal{P}$. The algorithm to maintain D is based on this operation: Let v be an apex of D (i.e., a shallowest vertex on some path of $\mathcal{P}$). Thus, $V(D_v) = V(T_v)$. To *recompress* v means to replace D_v in D by $C(T_v)$. As usual, $C(T_v)$ is defined using the heavy paths of T_v , and the recompression changes $\mathcal{P}$ accordingly. Each node of T_v gets *reorganized* in this recompression.

Recompressing v updates the fat preordering of D as follows. Let s be the size function on the recompressed subtree $D_v = C(T_v)$. The fat preordering will take σ to be s . The other parameters of the ordering are specified later. If $v = \rho(D)$, the recompression uses the (static) fat preordering

algorithm to assign new numbers to the nodes of D_v in the interval $[0, cs(v)^e)$. If $v \neq \rho(D)$, let u be the parent $u = \pi_D(v)$. Let $\bar{Q}(u)$ be the currently largest value $\bar{q}(z)$ for a child z of u in D . The *expansion interval* for u is $[\bar{Q}(u), q(u))$. Clearly, no numbers have been assigned in this interval. Use the fat preordering algorithm to assign new numbers to the nodes of T_v in the interval $[\bar{Q}(u), \bar{Q}(u) + cs(v)^e)$. This updates $\bar{Q}(u)$ to $\bar{q}(v)$, decreasing the size of u 's expansion interval by $cs(v)^e$. The old interval for v , $[\bar{p}(v), \bar{q}(v))$, will no longer be used; in effect, it is discarded.

The last part of the high-level description is based on a parameter α , $3/2 > \alpha > 1$. For any node u let $s(u)$ denote its current size in D and let $\sigma(u)$ denote the σ -value for u in the current fat preordering (i.e., u 's interval $[\bar{p}(u), \bar{q}(u))$ has size $c\sigma(u)^e$). $s(u)$ equals $\sigma(u)$ plus the number of descendants that u has gained since its reorganization. D is maintained to always have

$$s(u) < \alpha\sigma(u)$$

for every node u .

We turn to the data structure. In D the data structure maintains the values of $\bar{p}, p, q, \bar{q}, \bar{Q}, s$, and σ for every vertex u . It also marks the apexes. The D tree is represented by parent pointers. The tree T is represented by children lists (i.e., each node has a list of its children). Also, each path of $\mathcal{P}$ is recorded (for the *ca* algorithm).

Now we give the detailed algorithms. The *ca* algorithm is the same as the static case. *add_leaf*(x, y) proceeds as follows:

Add y to the list of children of x in T . Make y a singleton path of $\mathcal{P}$ by marking it an apex and setting $\pi_D(y)$ to x if x is an apex; else $\pi_D(x)$. Increase $s(a)$ by 1 for each ancestor a of y in D . Let v be the last ancestor of y that now has $s(v) \geq \alpha\sigma(v)$. (This condition holds for $v = y$ by convention.) Recompress v . (Start by computing the values $s_T(z)$ for $z \in T_v$.) Then construct the new ancestor table for each node of T_v .

Note that if $v = y$ in this algorithm, recompressing v does not change D but assigns v its fat preorder values.

Finally, we give the parameters for the fat preorder. As we shall show, they are selected so the preceding strategy can be carried out; in particular, expansion intervals are large enough. Starting with the preceding parameter $\alpha \in (1, 3/2)$, we will use

$$\beta = \frac{2}{2\alpha - 1}$$

as the constant of Equation (2) and, in addition to Equation (1), we require

$$c \frac{(\alpha - 1/2)^e + 1/2^e}{1 - 1/\alpha^e} \leq c - 2. \quad (4)$$

Notice $\alpha \in (1, 3/2)$ implies the fraction of the left-hand side approaches 0 as $e \rightarrow \infty$, so this inequality can always be achieved. For example, take $\alpha = 6/5$, $\beta = 10/7$, $e = 4$, $c = 5$. (Then Equation (1) amounts to $1.1 \leq 3 \leq 4.1$ and Equation (4) amounts to $2.93 \leq 3$.) The fat preorder must satisfy the defining properties (i) – (iii) for these parameters.

To show the algorithm is correct, we must first ensure that expansion intervals are large enough. More precisely, consider an apex u of D that has just been reorganized. The algorithm adds $< (\alpha - 1)\sigma(u)$ descendants of u in D before recompressing D_u . The additions may cause various children v of u to be recompressed and thus assigned new intervals in $[p(u), q(u))$. We must show that the total length of all intervals ever assigned to children of u is $< q(u) - p(u) = (c - 2)\sigma(u)^e$. Here, strict inequality accounts for the fact that the integer $p(u)$ is assigned to u . Also note that the “total length” includes both the original intervals assigned when u is reorganized and the new intervals.

We will use the following notation. We continue to let $\sigma(u)$ denote its value when u is reorganized. Let v be a child of u in D . If v is an apex, it may be recompressed some number of times, say ℓ times. Let $\sigma_i(v)$, $i = 0, \dots, \ell$ be the various values of $\sigma(v)$. For example, $\sigma_0(v)$ is the value of $\sigma(v)$ when u is reorganized. For $i \geq 1$,

$$\sigma_i(v) \geq \alpha \sigma_{i-1}(v). \quad (5)$$

(If v is not an apex, then $\ell = 0$ and $\sigma_0(v) = 1$.)

LEMMA 2.5. *From when u is reorganized until its next reorganization, the total length of all intervals ever assigned to children of u is $< q(u) - p(u)$.*

PROOF. For any child v of u , Equation (5) implies $\sigma_\ell(v) \geq \alpha^i \sigma_{\ell-i}(v)$ for every $i = 0, \dots, \ell$. Thus, the total size of all intervals ever assigned to v is strictly less than

$$c\sigma_\ell(v)^e (1 + (1/\alpha)^e + (1/\alpha)^{2e} + \dots) = \frac{c\sigma_\ell(v)^e}{1 - 1/\alpha^e}.$$

Obviously, this holds for children with $\ell = 0$, too. So the total size of intervals ever assigned to children of u is strictly less than

$$S = \frac{c \sum_{v \in U} \sigma_\ell(v)^e}{1 - 1/\alpha^e}.$$

We can assume u has at least two children when it is initially reorganized. (If u starts with ≤ 1 child, u gets reorganized as soon as it gains a child since $\alpha\sigma(u) < (3/2)\sigma(u) \leq \sigma(u) + 1$.) Right after u was reorganized, every child v had $\sigma_0(v) = s(v) \leq s(u)/2 = \sigma(u)/2$ descendants in D . u gets $< (\alpha - 1)\sigma(u)$ new descendants before reorganization. The sum of all $\sigma_\ell(v)$ values is less than $\alpha\sigma(u)$. Since $e > 1$, simple calculus shows S is maximized when u starts with exactly two children, each with $\leq \sigma(u)/2$ children, and every new node descends from the same initial child. (For any initial configuration, S is maximized when all new nodes descend from the child that starts with the greatest value $\sigma_0(v)$. This value in turn is maximized when there are only two descendants and each starts with $\leq \sigma(u)/2$ descendants.) Thus, the maximum value of S is

$$c\sigma(u)^e \frac{(\alpha - 1/2)^e + 1/2^e}{1 - 1/\alpha^e}.$$

The left inequality of Equation (4) implies $S \leq (c - 2)\sigma(u)^e$ as desired. $\square$

Now we complete the correctness proof.

LEMMA 2.6. *The add_leaf and ca algorithms are correct.*

PROOF. We start by verifying Equation (2); that is, at any time when v is a child of u in D , the current σ function satisfies $\sigma(u) \geq \beta\sigma(v)$. Immediately after u was last reorganized we have $\sigma(u) = s(u) \geq 2s(v) = 2\sigma(v)$. This inequality holds even if v has not been added by *add_leaf* since we take $\sigma(v) = 0$. After the reorganization, $\sigma(u)$ does not change, and u gets less than $(\alpha - 1)\sigma(u)$ new descendants. Thus, at any time $s(v) \leq \sigma(u)/2 + (\alpha - 1)\sigma(u) = (\alpha - 1/2)\sigma(u)$. We always have $\sigma(v) \leq s(v)$. Thus, $\sigma(v)/(\alpha - 1/2) \leq \sigma(u)$ (i.e., $\beta\sigma(v) \leq \sigma(u)$).

The defining properties (i)–(iii) of fat preorder numbers hold since recompression uses the static preorder numbering algorithm. (Since n is nondecreasing, the current σ always has values $\leq n$.) The rest of the data structure consists of the *ancestor* tables and the orderings of the paths of $\mathcal{P}$. If $x \in T_v$, *ancestor_x* is constructed in entirety in the recompression. If $x \notin T_v$ the recompression does not change *ancestor_x*. This table remains valid since no ancestor b of x is in T_v , so $\sigma(b)$ does not change. Similarly, a path of $\mathcal{P}$ with an apex not in T_v is vertex-disjoint from T_v . So it does not change when v is recompressed, and the data structure representing it remains valid.

The *ca* algorithm works correctly since the data structure is correct. $\square$

LEMMA 2.7. *The nearest common ancestors problem with `add_leaf` and `ca` operations can be solved in $O(m + n \log^2 n)$ time and $O(n \log n)$ space.*

PROOF. A `ca` operation uses $O(1)$ time, so the `cas` use $O(m)$ time total.

In `add_leaf`(x, y), examining each ancestor of y uses $O(\log n)$ time. Recompressing a node v uses $O(s(v))$ time for all processing except constructing the new ancestor tables, which uses $O(s(v) \log n)$ time. Hence, the time for an `add_leaf` operation is $O(s(v) \log n)$. The algorithm's recompression strategy implies $s(v) < 1 + \alpha\sigma(v) \leq (1 + \alpha)\sigma(v)$. So the time is

$$O(\sigma(v) \log n).$$

When v is recompressed, $\geq (\alpha - 1)\sigma(v)$ descendants y' of v have been added since the last reorganization of v . Charge the above time $O(\sigma(v) \log n)$ to these new descendants y' , at the rate of $O(\log n)$ per node. Since $\alpha > 1$, this accounts for the time recompressing v .

A given node y' is charged at most once from a node v that was an ancestor of y' when `add_leaf` added y' . (In proof, recompressing v reorganizes every new ancestor w of y' . So y' will not be charged from w . In other words, after y' is charged from v , it is only charged from proper ancestors of v .) Thus, Equation (2) implies any y' is charged $\leq \log_{\beta} cn^e$ times total. So the total time charged to y' is $O(\log^2 n)$. The time bound follows.

For the space bound, note that the ancestor tables use space $O(n \log n)$. The remaining space (as specified in the data structure for D) is linear. $\square$

The lemma does not require n (the number of `add_leaf` operations) to be known in advance. (This is the case in most of our applications of this algorithm, although not in the implementation of Edmonds' algorithm.) The timing analysis still applies verbatim. The use of ancestor tables necessitates a storage management system to achieve the lemma's space bound. We use a standard doubling strategy. Since we use this several times, the following lemma gives a precise statement. For completeness, the lemma is proved in Appendix B, a simple application of Cormen et al. [5, Sec. 17.4.1].

Consider a collection of arrays $A_i[1..n_i]$ that grow by adding entries. Each such operation enlarges A_i with a new entry (i.e., n_i increases by 1 and the contents of old entries in A_i do not change). We implement this operation by allocating space for all arrays A_i sequentially within a larger array S . When the current version of A_i in S becomes full, we allocate a new version at the end of S with twice the size. We will also allow an operation that creates a new array A_i .

LEMMA 2.8. *A collection of arrays $A_i[1..n_i]$, $i = 1, \dots, k$ that grows by adding new array entries and creating new arrays can be maintained within an array $S[1..4n]$, for $n = \sum_i n_i$. The extra time is $O(n)$.*

Returning to Lemma 2.7, when the final value of n is unknown, note that the space usage consists of ancestor tables and single values associated with each vertex. (The children lists for T are stored as vertex values: v points to its first child, and each child of v points to the next child. The paths of $\mathcal{P}$ are also stored as vertex values $\text{pred}(v)$: a vertex v on a heavy path has $\text{pred}(v)$ equal to the predecessor of v on the heavy path.) The single values are updated in `add_leaf` operations and recompressions. Lemma 2.8 is used to manage all the space (including single vertex values). We conclude that Lemma 2.7 holds in entirety when n is not known in advance.

Now we extend these algorithms to allow `add_root` operations in addition to `add_leaf`. We show that the general incremental tree problem reduces to `add_leaf` and `ca` operations. First extend the characteristic ancestor operation. For an arbitrary node r of T , let $\text{nca}(x, y; r)$ denote the nearest common ancestor of x and y when T is rerooted at vertex r . Define $\text{ca}(x, y; r)$ similarly. All other terminology is unchanged (e.g., $\text{ca}(x, y)$ denotes the characteristic ancestors in T with its original

root and similarly for the term “ancestor.”) The following lemma shows that we can compute $ca(x, y; r)$ just using the ca functions on T .

LEMMA 2.9. (i) Any 3 nodes in a tree x, y, z have $|\{nca(x, y), nca(x, z), nca(y, z)\}| \leq 2$.

(ii) $ca(x, y; z) = \begin{cases} ca(x, y) & \text{if } nca(x, z) = nca(y, z), \\ (a, \pi(a), a_y) & \text{if } nca(x, z) = nca(x, y) \neq nca(y, z) \text{ and } ca(y, z) = (a, a_y, a_z). \end{cases}$

REMARK: Part (i) and symmetry of x and y show that part (ii) gives a complete definition of $ca(x, y; z)$.

PROOF. (i) Let a be the shallowest of the three nodes $nca(x, y)$, $nca(x, z)$, $nca(y, z)$. So wlog $a = nca(x, y)$, and let $ca(x, y) = (a, a_x, a_y)$. If $a \neq nca(x, z)$ then $nca(x, z)$ is an ancestor of x deeper than a . So $a_x \neq a$, and z descends from a_x . Thus, the path from y to z goes through a and a_x , so $a = nca(y, z)$.

(ii) Let $ca(x, z) = (b, b_x, b_z)$.

Suppose $nca(x, z) = nca(y, z)$. $b = nca(x, z)$ is an ancestor of y . If $b_x \neq b$ and b_x is an ancestor of y , then $nca(y, z)$ descends from b_x , contradicting $b = nca(y, z)$. Thus, $b = nca(x, y)$ and $b = nca(x, y; z)$. Clearly, $ca(x, y; z) = ca(x, y)$.

Next suppose $nca(x, z) = nca(x, y) \neq nca(y, z)$. The equality implies b is an ancestor of y , and the inequality implies $b \neq b_z$ and b_z is an ancestor of y . Thus, $nca(x, y; z) = nca(y, z)$. Let $ca(y, z) = (a, a_y, a_z)$. Then $ca(x, y; z) = (a, \pi(a), a_y)$. $\square$

It is now a simple matter to implement *add_root* operations in terms of *add_leaf*: We never change the children lists of the data structure representing T . Instead, we maintain a pointer ϱ that gives the current root of T as defined by *add_root* operations. The operation *add_root*(y) is implemented as

$$\text{add_leaf}(\varrho, y); \varrho \leftarrow y. \quad (6)$$

The algorithm for $ca(x, y)$ is $ca(x, y; \varrho)$.

We close this section by noting that *add_root* can be implemented directly, without the general reduction. The main observation is that *add_root* changes the compressed tree in a simple way: Let T' be the result of performing *add_root*(y) on T , a tree with root x . If $|V(T)| > 1$, then y plus the heavy path with apex x in T forms a heavy path in T' . Thus, $C(T')$ can be constructed from $C(T)$ by changing the name of the root from x to y and giving the root a new child named x . (This works when $|V(T)| = 1$, too.) This transformation is easily implemented in our data structure.

COROLLARY 2.10. *The nearest common ancestors problem with add_leaf, add_root, and ca operations can be solved in $O(m + n \log^2 n)$ time and $O(n \log n)$ space.*

As before, the corollary does not require that n be known in advance.

3 NCAS FOR EDMONDS' ALGORITHM

This section gives a simple algorithm to find the *ncas* needed in Edmonds' matching algorithm. Each *nca* operation uses $O(1)$ time, and the total time for all *add_leaf*'s is $O(n \log n)$. The extra space is $O(n)$. So this completes our efficient implementation of the weighted matching algorithm. (Readers interested only in matching needn't go beyond this section.)

This section also introduces the multilevel approach, wherein the incremental tree algorithm is used on a number of trees derived from the given tree, each at a given “level.” We use three versions of the approach. The simplest is for Edmonds' algorithm, and the two more elaborate versions are presented in the next two sections.

Finally, the bitstring data structure presented in this section is applied in Appendix C to show a simple implementation of the incremental-tree set merging algorithm of Gabow and Tarjan [11].

Fig. 3. (a) Data structure for $ncas$ in Edmonds' algorithm. $\mu = \lfloor \log n \rfloor$. y is a 1-node, $\hat{z}$ is nonfull. (b) Generalization to L levels, $T_L = T$, $L \geq \ell > 1$.

The idea is to reduce the number of tree nodes by contracting small subtrees. The terms *vertex* and *the incremental tree* T refer to the objects of the given problem (e.g., an operation $add_leaf(x, y)$ makes vertex x the parent of vertex y in T). We use two trees, illustrated in Figure 3(a). T_2 is the incremental tree T , enhanced with its data structure. T_1 is a smaller version of T , derived by contractions and deletions.

This indexing is a special case of our second multilevel structure, illustrated in Figure 3(b): A tree T is represented on $L > 1$ levels, with $T = T_L$; and, for level $\ell \in [2..L]$, tree $T_{\ell-1}$ is a minor of T_ℓ . Edmonds' algorithm uses $L = 2$.

For Edmonds' algorithm, define $\mu = \lfloor \log n \rfloor$. The algorithm maintains a partition of the vertices of $T = T_2$ into subtrees of $\leq \mu$ vertices called *2-subtrees*. A 2-subtree containing exactly μ vertices is *full*. T_1 is formed from $T = T_2$ by discarding the nonfull 2-subtrees and contracting the full ones. A node of T_1 (i.e., a contracted 2-subtree) is called a *1-node*. (Figure 3(b) illustrates the general case.)

We use this additional notation, illustrated in Figure 3: For any vertex x , $\hat{x}$ denotes the 2-subtree containing x . As an example, 2-subtrees are created and maintained in $add_leaf(x, y)$ as follows:

If $\hat{x}$ is a full 2-subtree, then y is made the root of a new 2-subtree. Otherwise, y is added to $\hat{x}$. If this makes $\hat{x}$ full, then a 1-node is created for $\hat{x}$. It is added to T_1 either as the root if $\hat{x} = T$ or with parent the 1-node containing $\pi(\rho(\hat{x}))$. (Note that this description guarantees that T_1 is a tree and not a forest.)

```

if  $\widehat{x} \neq \widehat{y}$  then
  /* set  $x$  and  $y$  so  $\widehat{x} = \widehat{y}$  and  $nca(x, y)$  is unchanged */
  if  $\widehat{x}$  is nonfull then  $x \leftarrow \pi(\rho(\widehat{x}))$ ; if  $\widehat{y}$  is nonfull then  $y \leftarrow \pi(\rho(\widehat{y}))$ 
   $(a, a_x, a_y) \leftarrow ca(\overrightarrow{x}, \overrightarrow{y})$ 
  if  $a \neq a_x$  then  $x \leftarrow \pi(\overleftarrow{a_x})$ ; if  $a \neq a_y$  then  $y \leftarrow \pi(\overleftarrow{a_y})$ 
return  $nca(x, y)$ 

```

Fig. 4. Finding $ncas$ in Edmonds' algorithm.

When $\widehat{x}$ is full, $\overrightarrow{x}$ denotes the 1-node containing x .⁴ If x is a 1-node (i.e., the contraction of a 2-subtree S), $\overleftarrow{x}$ denotes the root vertex of S ($\overleftarrow{x}$ is a vertex of $T_2 = T$). Also note that we write functions of nodes like $\pi(x)$ and $ca(x, y)$, relying on context (i.e., the identity of arguments x and y) to indicate which tree T_i is being used.

T_1 is processed using the incremental-tree nca algorithm of Lemma 2.7. Clearly, there are $O(n/\log n)$ 1-nodes, so the time spent on T_1 is $O(m + n \log n)$ and the space is $O(n)$.

T uses a simple data structure: Each root x of a 2-subtree is marked as such and stores the size of its tree $|V(\widehat{x})|$. If $\widehat{x}$ is full, then x has a pointer to its 1-node $y = \overrightarrow{x}$; also, y has a pointer to $x = \overleftarrow{y}$. Each nonroot has a pointer to its 2-subtree root. Each node x of T has a parent pointer as well as child pointers; the children of x that belong to $\widehat{x}$ all occur before the children not in $\widehat{x}$.

Figure 4 gives the algorithm for $nca(x, y)$. Note that the ca operation takes place in T_1 and the nca operation is in a 2-subtree, in T_2 .

We complete the data structure by showing how to process add_leaf and nca operations in 2-subtrees. We do this by maintaining a representation of ancestors as bitstrings in trees that grow by add_leaf operations, assuming their size remains $\leq \log n$. Edmonds' algorithm uses this data structure on every 2-subtree. The details of the data structure are as follows.

Let T be a tree that grows by add_leaf operations, of size at most $|V(T)| \leq \log n$. The nodes of T are numbered sequentially as they get added, starting at 1. The number of node x is stored as its "identifier" $id[x] \in [1..|V(T)|]$. T also has an array $v[1..|V(T)|]$ that translates identifiers to their corresponding node (i.e., $v[i]$ specifies the vertex of T whose identifier is i).

Each vertex $x \in T$ has a RAM word $anc[x]$ that stores a string of $\leq \log n$ bits. The i th bit of $anc[x]$ (corresponding to 2^i) is 1 if and only if node number i is an ancestor of x . So, for example, bits 1 and $id[x]$ are always 1. The key property is that reading the bits most-significant-first gives the ancestors of x in their proper order (i.e., decreasing depth).

For add_leaf , we maintain a value s as the current size of T . $add_leaf(x, y)$ is implemented as

$$\pi(y) \leftarrow x; id[y], s \leftarrow s + 1; v[s] \leftarrow y; anc[y] \leftarrow anc[x] + 2^s.$$

We precompute a table that gives most-significant bits. Specifically, for any bitstring $b \neq 0$ of $\log n$ bits, $msb[b]$ is the index of the most significant bit of b . The operation $nca(x, y)$ is implemented as

$$v[msb[anc[x] \wedge anc[y]]].$$

It is easy to see that add_leaf and nca both use $O(1)$ time. To use this data structure in Edmonds' algorithm, we keep space usage linear by using the doubling strategy of Lemma 2.8 on the collection of v arrays of all 2-subtrees.

Using the results of Gabow [9], which leaves the incremental-tree nca problem as the last detail of Edmonds' algorithm, we get the following:

⁴The forward arrow notation corresponds to the arrows in Figure 3.

THEOREM 3.1. *A search of Edmonds' algorithm can be implemented in time $O(m + n \log n)$ and space $O(m)$.*

The bitstring data structure is used in the next section, where we need the complete ca vector and not just nca . To accomplish this, we precompute another table for least-significant bit. Specifically, for any bitstring $b \neq 0$ of $\log n$ bits, $lsb[b]$ is the index of the least significant bit of b . Recalling $ca(x, y) = (nca(x, y), a_x, a_y)$, node a_x is found by

$$a_x \leftarrow \text{if } nca(x, y) = x \text{ then } x \text{ else } v[lsb(anc[x] \wedge \neg anc[y])],$$

and similarly for a_y .

4 MULTILEVEL INCREMENTAL-TREE ALGORITHMS

This section begins by giving the details of the multilevel approach illustrated in Figure 3(b). Then it presents a 3-level algorithm to solve the incremental-tree nca problem in time $O(1)$ for nca queries, total time $O(n)$ for add_leaf and add_root operations, and space $O(n)$. That algorithm is used in the next section to construct our most general nca algorithm. It uses the multilevel structure presented in this section, with an unbounded number of levels and some changes that we note.

The Framework

This section gives the high-level organization of an incremental nca algorithm with an arbitrary number of levels. We will only need three levels in the next section, but the number of levels is unbounded in Section 5.

The terms *vertex* and *the incremental tree* T refer to the objects of the given problem (e.g., an operation $add_leaf(x, y)$ makes vertex x the parent of vertex y in T). A multilevel algorithm works on a number of levels designated $\ell = L, L - 1, \dots, 1$.

The incremental tree T is represented by a tree T_ℓ on every level (a small tree T may have T_ℓ empty for levels ℓ less than some threshold). T_L is T . Every other T_ℓ is a smaller tree derived from $T_{\ell+1}$ by deletions and contractions. Each T_ℓ is composed of ℓ -nodes (called *nodes* if the level is clear). The algorithm maintains a partition of the nodes of T_ℓ into subtrees called ℓ -subtrees. Each level is provided with given algorithms that solve the incremental problem in any ℓ -subtree; the multilevel algorithm described in this section sews these given algorithms together to solve the incremental problem on the given tree T .

Every level ℓ has an integral size parameter μ_ℓ .⁵ Every ℓ -subtree S contains $\leq \mu_\ell$ ℓ -nodes. S is *full* if equality holds. $\mu_1 = n + 1$, so T_1 is always nonfull (if it exists). For $L \geq \ell > 1$, $T_{\ell-1}$ is formed from T_ℓ by discarding every nonfull ℓ -subtree and contracting the full ones. The fact that any T_ℓ is a tree (i.e., not a forest) follows from this invariant: For every level ℓ , the nonfull ℓ -subtrees of T_ℓ are at its frontier (i.e., any node x in a nonfull ℓ -subtree S has all its T_ℓ -children in S).

Efficiency in a multilevel algorithm is achieved using the shrinkage of the tree from level to level. Specifically, an ℓ -node with $\ell < L$ contains $\prod_{i=\ell+1}^L \mu_i$ vertices of T . So the number of ℓ -nodes is

$$|V(T_\ell)| \leq n / \prod_{i=\ell+1}^L \mu_i. \quad (7)$$

We use this additional notation: Let x be an ℓ -node, $L \geq \ell \geq 1$. $\widehat{x}$ denotes the ℓ -subtree containing x . If $\widehat{x}$ is full (in particular $\ell > 1$), then $\vec{x}$ denotes the $(\ell - 1)$ -node that is the contraction of $\widehat{x}$. If $\ell < L$, then x is the contraction of an $(\ell + 1)$ -subtree S , and $\overleftarrow{x}$ denotes the root node of S . As before, we write functions of nodes like $\pi(x)$, relying on context (i.e., the identity of argument x) to indicate which tree T_ℓ is referenced.

⁵In Section 5, these size parameters are replaced by a notion of “stage.”


```

 $(a, a_x, a_y) \leftarrow c(\vec{x}, \vec{y}, \ell - 1)$ 
if  $a_x \neq a$  then  $x \leftarrow \pi(\overleftarrow{a_x})$ ; if  $a_y \neq a$  then  $y \leftarrow \pi(\overleftarrow{a_y})$ 
 $(b, b_x, b_y) \leftarrow \widehat{c}(x, y, \ell)$ 
if  $b_x = b$  and  $a_x \neq a$  then  $b_x \leftarrow \overleftarrow{a_x}$ ; if  $b_y = b$  and  $a_y \neq a$  then  $b_y \leftarrow \overleftarrow{a_y}$ 
return  $(b, b_x, b_y)$ 

```

Fig. 5. Procedure for $c(x, y, \ell)$ when $\widehat{x}$ and $\widehat{y}$ are full.

Each T_ℓ uses this data structure: Let x be an ℓ -node. If x is the root of its ℓ -subtree, it stores the subtree size $|V(\widehat{x})|$ ($1 \leq |V(\widehat{x})| \leq \mu_\ell$). It also has pointers to $\overleftarrow{x}$ if $\ell < L$ and $\overrightarrow{x}$ if $\ell > 1$. A nonroot x has a pointer to its ℓ -subtree root. Every x has a parent pointer.

The main routines for incremental trees are

$$a(x, y, \ell), \widehat{a}(x, y, \ell), c(x, y, \ell), \widehat{c}(x, y, \ell).$$

$a(x, y, \ell)$ is a recursive routine that performs the *add_leaf* operation for ℓ -node x and new ℓ -node y . It may call $a(\vec{x}, \vec{y}, \ell - 1)$. For vertices x, y in the given graph, the operation *add_leaf* (x, y) is performed by $a(x, y, L)$. The a routine makes use of $\widehat{a}$ to grow ℓ -subtrees. Specifically for $\widehat{a}(x, y, \ell)$, x is an ℓ -node and y a new ℓ -node that is made a child of x in the ℓ -subtree $\widehat{x}$.

The ca operation is organized similarly. It uses the recursive routine $c(x, y, \ell)$, which returns the characteristic ancestors of ℓ -nodes x and y , possibly invoking $c(\vec{x}, \vec{y}, \ell - 1)$. For vertices x, y in the given graph, the operation *nca* (x, y) is performed by $c(x, y, L)$. The c routine makes use of $\widehat{c}$, which returns the characteristic ancestors of ℓ -nodes x and y that belong to the same ℓ -subtree $\widehat{x} = \widehat{y}$.

We will extend these operations later to allow *add_root*. Also looking ahead, in Section 5, the *nca* routine will use c and $\widehat{c}$ with some obvious modifications. The *link* routine will use the same overall structure as a and $\widehat{a}$ for *add_leaf*.

Now we describe the two recursive algorithms starting with $a(x, y, \ell)$:

CASE $\widehat{x}$ is full: Make x the parent of node y , and make y a singleton ℓ -subtree.

CASE $\widehat{x}$ is nonfull: Execute $\widehat{a}(x, y, \ell)$. If $\widehat{x}$ is still not full, we are done; but suppose $\widehat{x}$ has become full. Create a new $(\ell - 1)$ -node z . Make z the node $\overrightarrow{x}$. Now there are two subcases:

SUBCASE $\widehat{x} = T_\ell$: Make z the unique $(\ell - 1)$ -node, as well as a singleton $(\ell - 1)$ -subtree.

SUBCASE $\widehat{x} \neq T_\ell$: $w = \pi(\rho(\widehat{x}))$ is in a full ℓ -subtree. Execute $a(\overrightarrow{w}, z, \ell - 1)$ to add z as a new $(\ell - 1)$ -leaf.

The *add_leaf* algorithm preserves the defining properties of T_ℓ trees and so is correct. The total time for all *add_leaf* operations is dominated by the time used by $\widehat{a}$ to build all the ℓ -subtrees, $L \geq \ell \geq 1$. (This includes the time to create a new singleton subtree.)

We turn to the c routine. The high-level strategy is simple: Use $c(\vec{x}, \vec{y}, \ell - 1)$ to find the ℓ -subtree S containing *nca* (x, y), and then use the $\widehat{c}$ routine in S to find the desired level ℓ characteristic ancestors (c, c_x, c_y) . There are various special cases, depending on whether or not $\widehat{x}$ and $\widehat{y}$ are full, whether or not S contains c_x or c_y , etc. The details of $c(x, y, \ell)$ are as follows.

CASE $\widehat{x}$ and $\widehat{y}$ are both full: Figure 5 gives pseudocode for this case. It handles special cases such as $x = y$ or $\widehat{x} = \widehat{y}$.

CASE *One or both of $\widehat{x}, \widehat{y}$ is nonfull*: If $\widehat{x} = \widehat{y}$, we use $\widehat{c}(x, y, \ell)$ directly. (This includes the special case where there are no $(\ell - 1)$ -nodes.) Assuming $\widehat{x} \neq \widehat{y}$, when $\widehat{x}$ is nonfull we replace x by $\pi(\rho(\widehat{x}))$, and similarly for y . We then execute the code of Figure 5. If the returned b_x is the replacement for x , we change b_x to $\rho(\widehat{x})$, and similarly for y .

The analysis of this algorithm is similar to *add_leaf*: Correctness follows from the defining properties of T_ℓ trees. The time for an operation $nca(x, y)$ is dominated by the time used by the routines $\widehat{c}(x, y, \ell)$, $L \geq \ell \geq 1$.

We extend the routines to allow *add_root*, similar to the extension for Corollary 2.10, as follows. *add_root* is still implemented by Equation (6), where now ϱ is a pointer to a node in the tree T_L . The routine for $nca(x, y)$, instead of immediately calling $c(x, y, L)$, is modified to use Lemma 2.9(ii) as before. Specifically, it calls $c(x, y, L)$, $c(x, \varrho, L)$, and $c(y, \varrho, L)$, and chooses $nca(x, y)$ according to the lemma.

Linear-Time Incremental Trees

Take $L = 3$ levels with

$$\mu_3 = \mu_2 = \lceil \log n \rceil, \mu_1 = n + 1. \quad (8)$$

Level 1 uses the incremental-tree algorithm of Section 2. It uses $O(m + n)$ time and $O(n)$ space. This follows since Equation (7) shows there are $\leq \frac{n}{\log^2 n}$ 1-nodes. Thus, Corollary 2.10 shows the time on level 1 is $O(m + \frac{n \log^2 n}{\log^2 n}) = O(m + n)$. The space in level 1 is $O(\frac{n \log n}{\log^2 n}) = O(n)$.

Levels 3 and 2 both use the bitstring data structure of Section 3. For level 2, we compute the complete characteristic ancestor vector $ca(x, y)$, as described in Section 3. The analysis of that section shows that levels 3 and 2 use $O(m + n)$ time. The space is kept linear, $O(n)$, by using the doubling strategy of Lemma 2.8 for the v tables of nonfull ℓ -subtrees.

This completes the three-level incremental tree algorithm.

THEOREM 4.1. *The incremental-tree nearest common ancestors problem with *add_leaf*, *add_root*, and *ca* operations can be solved in $O(m + n)$ time and $O(n)$ space.*

As in Corollary 2.10, the theorem does not require n (the number of *add_leaf* and *add_root* operations) to be known in advance. To achieve this, first consider Equation (8) defining the μ_i . One approach is to update these values every time n doubles. Instead, we will simply interpret n in Equation (8) to be N , the maximum integer that can be represented in the RAM. So $\mu_3 = \mu_2 = \log N$ is the number of bits in a RAM word. The timing estimates are unchanged. For instance, the time in level 1 is $O(m + \frac{n \log^2 n}{\log^2 N}) = O(m + n)$. The space in level 1 is $O(\frac{n \log n}{\log^2 N}) = O(n)$. The space for all three levels is maintained in one array S , using Lemma 2.8.

5 LINK OPERATIONS

This section extends the multilevel data structure to solve our most general dynamic *nca* problem. The algorithm processes m *nca* and *link* operations on a set of n nodes in time $O(m\alpha(m, n) + n)$ and linear space $O(n)$.

The multilevel structure shares many details with that of the previous section: The levels $\ell = L, \dots, 1$ and the notions of ℓ -tree, ℓ -node, and ℓ -subtree are all unchanged. A difference is that a tree T_ℓ at level $\ell > 1$ gives its level $\ell - 1$ counterpart $T_{\ell-1}$ by contracting *every* ℓ -subtree (i.e., no subtrees are deleted). The notations $\widehat{x}$, $\vec{x}$, and $\overleftarrow{x}$ are defined without change.

nca operations are implemented using the c and $\widehat{c}$ routines, as in last section. *link* operations are implemented using a recursive routine l similar to a of last section. The analog of $\widehat{a}$ for link is folded into l (i.e., there is no $\widehat{l}$). It is convenient to use an extra argument for l : We write $l(r, x, y, \ell)$

where r is the root of the ℓ -tree containing x . Call a tree built up by *link* operations a *link tree*. The operation $\text{link}(x, y)$ is performed by $l(\rho, x, y, L)$ for ρ the root of the link tree containing x .

Example Algorithms

For motivation, we start by sketching the two simplest versions of our algorithm.

Algorithm 1 Every link tree is represented as an incremental tree. The *link* operation uses *add_leaf* and *add_root* operations to transfer the nodes of the smaller tree into the larger (for trees of equal size, break the tie arbitrarily). It then discards the incremental tree of the smaller tree. The number of node transfers is $O(n \log n)$. So Theorem 4.1 shows the total time is $O(m + n \log n)$.

The analysis of Algorithm 1 is based on what we will call the “stage” of the link tree: A tree in stage σ has between 2^σ and $2^{\sigma+1}$ vertices. We view the analysis as charging a vertex $O(1)$ to advance from one stage to the next. (This accounts for the total time spent on *add_leaf* and *add_root* operations since Theorem 4.1 shows the time spent on a tree that ultimately grows to n_s nodes is $O(n_s)$; i.e., the time is proportional to the number of node transfers.) Our more efficient algorithms maintain explicit stages, and these stages will require faster growth in the tree size.

Algorithm 2 Algorithm 1 can be improved using a two-level strategy similar to previous ones. Level 2 classifies each tree as stage 1 or 2: A 2-tree is in stage 1 if it has $< \log n$ nodes and stage 2 if it has $\geq \log n$ nodes.

A stage 2 2-tree is partitioned into 2-subtrees, each of which contains $\geq \log n$ nodes. Each 2-subtree is represented as an incremental tree using the data structure of Theorem 4.1. Contracting all these 2-subtrees gives its corresponding 1-tree.

A stage 1 2-tree is also a 1-tree. For consistent terminology in stage 1, view each 2-node as a 2-subtree.

Level 1 uses Algorithm 1 on all 1-trees.

The l routine works as follows on level 2: It sets $\pi(y) \leftarrow x$. Then, letting X and Y denote the 2-trees containing x and y , respectively, it executes the case that applies:

CASE *Both trees are in stage 2*: Link the level-1 trees using Algorithm 1.

CASE *Only one tree is stage 2*: If X is stage 2, transfer the nodes of Y to $\hat{x}$, using *add_leaf* operations. Then discard the data structures for Y on levels 1 and 2.

If Y is stage 2, do the same, using appropriate *add_root* operations in the transfer of X to $\hat{y}$.

CASE *Both trees are stage 1*: If the combined trees contain $\geq \log n$ nodes, initialize the 2-tree as a new stage 2 tree, with one 2-subtree consisting of all nodes of X and Y . Discard the data structures for X and Y on both levels.

Otherwise, link the level 1 trees using Algorithm 1.

The total time is dominated by the time spent for all incremental trees on both levels 1 and 2. On level 2, a 2-subtree that grows to contain n_i nodes (as in the last two cases) uses time $O(n_i)$ for all *add_leaf* and *add_root* operations. So, all 2-subtrees use total time $O(n)$.

Consider level 1. The 1-trees for stage 2 2-trees collectively contain $\leq n / \log n$ nodes. Each node is transferred by Algorithm 1 at most $\log n$ times. So the total time is $O(n)$. The stage 1 2-trees collectively contain n nodes. Each is transferred $\leq \log \log n$ times by Algorithm 1. So the total time is $O(n \log \log n)$. This term strictly dominates the algorithm’s time bound.

Clearly, we can improve this algorithm by adding another stage for 2-trees with $\leq \log \log n$ nodes. The time becomes $O(n \log^{(3)} n)$. Continuing in this fashion, we can achieve time $O(n \log^* n)$.⁶ Let us sketch this algorithm. (The detailed version of the algorithm is the case $\ell = 2$

⁶ $\log^{(i)} n$ and $\log^* n$ are defined as in Cormen et al [5].

of algorithm $\mathcal{A}_\ell$ presented later.) It is convenient to switch notation from small functions like $\log$ to large ones like exponentiation. Recall the superexponentiation function, defined by $2 \uparrow 1 = 2$, $2 \uparrow (s + 1) = 2^{(2 \uparrow s)}$.

In Algorithm 2, level 2 has $\log^* n$ stages. A stage σ 2-tree has between $2 \uparrow \sigma$ and $2 \uparrow (\sigma + 1)$ nodes. It is partitioned into 2-subtrees, each of which contains $\geq 2 \uparrow \sigma$ nodes. The remaining properties of 2-trees are essentially the same as in the previous algorithm.

The l routine uses new criteria to determine the cases but is otherwise unchanged. In more detail, let X be in stage $\sigma(X)$ and similarly for $\sigma(Y)$, and let $\sigma = \max\{\sigma(X), \sigma(Y)\}$.

If the combined trees contain $\geq 2 \uparrow (\sigma + 1)$ nodes, a new stage $\sigma + 1$ tree is initialized (as in the last case just given).

Otherwise, if $\sigma(X) \neq \sigma(Y)$, the nodes of the smaller 2-tree are transferred to the larger (as in the preceding middle case).

Otherwise, ($\sigma(X) = \sigma(Y)$) and Algorithm 1 links the images of the two 2-trees (as in the first and last cases).

The time for all *link* operations is $O(n \log^* n)$. This holds because the time on each stage is $O(n)$. Let us sketch a proof. Consider level 2. As before, a 2-subtree that grows to contain n_i nodes uses time $O(n_i)$ for all *add_leaf* and *add_root* operations. This gives $O(n)$ time total for each stage on level 2. There are $\log^* n$ stages, so the total time is $O(n \log^* n)$.

As for level 1, a 1-node is a contracted 2-subtree. A fixed stage σ of level 2 contains a total of $\leq n/(2 \uparrow \sigma)$ 2-subtrees. Thus, over the entire algorithm, stage σ has $\leq n/(2 \uparrow \sigma)$ 1-nodes x . After being transferred $2 \uparrow \sigma$ times by Algorithm 1, x 's 1-tree has grown to $\geq 2^{2 \uparrow \sigma} = 2 \uparrow (\sigma + 1)$ 1-nodes. So the 2-tree containing $\overleftarrow{x}$ has advanced to stage $\sigma + 1$. So $O(2 \uparrow \sigma)$ time total is spent on x in level 1. Thus, the time for Algorithm 1 to process all stage σ 1-nodes is $O(\frac{n}{2 \uparrow \sigma} \cdot 2 \uparrow \sigma) = O(n)$. Again, there are $\log^* n$ stages, so the total time on level 1 is $O(n \log^* n)$.

We conclude that Algorithm 2 uses total time $O(m + n \log^* n)$.

The General Algorithm

The construction can be repeated using Algorithm 2 to get an even faster Algorithm 3, and so on. We now present the complete algorithm.

Define Ackermann's function $A_i(j)$ for $i, j \geq 1$ by

$$\begin{aligned} A_1(j) &= 2^j, \text{ for } j \geq 1; \\ A_i(1) &= 2, \text{ for } i \geq 2; \\ A_i(j) &= A_{i-1}(A_i(j-1)), \text{ for } i, j \geq 2. \end{aligned}$$

Define two inverse functions,

$$\begin{aligned} a_i(n) &= \min\{j : A_i(j) \geq n\}; \\ \alpha(m, n) &= \min\{i : A_i(4 \lceil m/n \rceil) \geq n\}, \text{ for } m, n \geq 1. \end{aligned}$$

These definitions differ slightly from those of Tarjan [20], but this does not change asymptotic estimates. The most significant difference is that our function $A_i(1)$ is constant compared to a rapidly growing function in Tarjan [20]. This makes for a more convenient treatment of the base case in our algorithms. We use some very weak properties of Ackermann's function, including these inequalities, which are proved in Appendix D:

$$A_i(j+1) \geq 2A_i(j), \text{ for } i, j \geq 1; \tag{9}$$

$$A_{i+1}(j) \geq A_i(2j), \text{ for } i \geq 1, j \geq 4; \tag{10}$$

$$\alpha(m', n') \geq \alpha(m, n) - 1, \text{ for } m' \leq 2m, n' \geq n. \tag{11}$$

A preprocessing step tabulates the relevant values of Ackermann's function. We use the values $A_i(j)$ that are $\leq n$ for $i \leq \alpha(m, n)$. Define an array $ackermann[1.. \log n, 1.. \log n]$: If $A_i(j) \leq n$, then $ackermann[i, j] = A_i(j)$; else $ackermann[i, j] = \epsilon$. This table also allows us to find $\alpha(m, n)$, which is $\leq \log n$ (since Equation (10) shows $A_{\log n}(4) \geq A_1(2^{1+\log n})$). The table is initialized, and $\alpha(m, n)$ is found, in time $O(\log^2 n)$. The table allows any desired value of Ackermann's function to be found in $O(1)$ time.

We use the linear-time incremental tree data structure of Theorem 4.1. Call a tree that is represented by this data structure an *incremental tree*. The preprocessing step computes all the tables for this algorithm in time $O(n)$.

The approach is similar to that of Gabow [7] for a list-splitting problem. We construct a family of algorithms $\mathcal{A}_\ell$, $\ell \geq 1$. $\mathcal{A}_\ell$ is a multilevel algorithm based on the function A_ℓ . It calls $\mathcal{A}_{\ell-1}$ if $\ell > 1$. $\mathcal{A}_\ell$ runs in time $O(m\ell + na_\ell(n))$.

Algorithm $\mathcal{A}_\ell$ works on level ℓ . The terms ℓ -node and ℓ -tree refer to the objects manipulated by $\mathcal{A}_\ell$. Every link tree corresponds to an L -tree with the same nodes and edges. Every level ℓ has $a_\ell(n)$ stages σ , $\sigma = 0, \dots, a_\ell(n) - 1$. Each ℓ -tree T belongs to a unique stage σ defined as follows:

CASE $|V(T)| < 4$: T is in stage $\sigma = 0$. Stage 0 uses a trivial algorithm so an invocation of c or ℓ uses time $O(1)$.

CASE $|V(T)| \geq 4$: T is in the stage $\sigma \geq 1$ satisfying $|V(T)| \in [2A_\ell(\sigma), 2A_\ell(\sigma + 1))$. (This is possible since $A_\ell(1) = 2$.) An ℓ -subtree in stage σ is a subtree that has $\geq 2A_\ell(\sigma)$ nodes. The nodes of T are partitioned into ℓ -subtrees. If $\ell > 1$, then T , with each ℓ -subtree contracted, is represented on level $\ell - 1$.

Note that the contracted tree on level $\ell - 1$ may be a trivial tree in stage 0 (of level $\ell - 1$). Also, if $\ell = 1$, there is no need to store the contracted tree since T has only one ℓ -subtree. This follows, since an ℓ -subtree has $\geq 2A_1(\sigma) = 2^{\sigma+1}$ nodes and $|V(T)| < 2A_1(\sigma + 1) = 2^{\sigma+2}$ nodes.

Algorithm $\mathcal{A}_\ell$ uses the following data structure. Each ℓ -tree and ℓ -subtree is represented by its root. An ℓ -tree T is stored using parent pointers and children lists. If r is the root of T , then $s(r)$ equals the size of T (the number of its ℓ -nodes). For any node x , $\sigma(x)$ equals the stage of x 's ℓ -tree; if $\sigma(x) > 0$, then $\hat{x}$ points to the ℓ -subtree containing x . Each ℓ -subtree is represented as an incremental tree. Recall (from Theorem 4.1) that it has a root pointer ϱ that is updated by *add_root* operations.

We turn to the *link* and *nca* operations. Initially, every node is a singleton link tree, in stage 0 of level L . Recall that the operation *link*(x, y) is processed by invoking the recursive algorithm $l(r, x, y, \ell)$ with arguments $\ell = L$ and r equal to the root of the link tree containing x . r is found by a simple recursive algorithm: If $\rho(\hat{x})$ is the root of its ℓ -tree, then it is r . Otherwise, recursively compute r as the root of the $(\ell - 1)$ -tree containing $\vec{x}$ and set $r \leftarrow \vec{r}$.

The algorithm for $l(r, x, y, \ell)$ is as follows. Let X and Y denote the ℓ -trees with root r and y , respectively, on entry to l .

Combine Step: Combine X and Y into a new ℓ -tree T_ℓ by setting $\pi(y) \leftarrow x$ and adding y to the child list of x .

The rest of the algorithm determines the stage of T_ℓ and its decomposition into ℓ -subtrees. Start by increasing $s(r)$ by $s(y)$. Let $\sigma = \max\{\sigma(x), \sigma(y)\}$. Execute the first of the following cases that applies and then return.

CASE 1 $s(r) \geq 2A_\ell(\sigma + 1)$: Make T_ℓ a new stage $\sigma + 1$ ℓ -tree consisting of one ℓ -subtree, as follows: Initialize a new incremental tree $\hat{r}$. Traverse T_ℓ top down; when visiting a node v , do an *add_leaf* operation to add v to $\hat{r}$. Discard the data structures for X and Y on all levels $\leq \ell$. If $\ell > 1$, then create an $(\ell - 1)$ -tree in stage 0 for T_ℓ , consisting of one node.

Fig. 6. Examples for *link*. $L = 3$. (a) Trees for *link*(x, y): (a.1) 3-trees X and Y . (a.2) 3-subtrees for X and Y . X has one 3-subtree. (a.3) 2-trees for X and Y . (b) T_3 formed for *link*(x, y), and link tree Z . (b.1)–(b.3) 3-trees, 3-subtrees, and 2-trees, as before. (c) T_3 formed for *link*(z, r). (c.1)–(c.3) 3-tree, 3-subtrees, 2-tree. (c.4) 2-subtree contains entire 2-tree.

CASE 2 $\sigma(x) > \sigma(y)$: Traverse Y top down, doing *add_leaf* operations to add each node of Y to the incremental tree $\hat{x}$. Discard the data structures for Y on all levels $\leq \ell$.

CASE 3 $\sigma(x) < \sigma(y)$: Traverse the path from x to r in X , doing *add_root* operations to add each node to the incremental tree $\hat{y}$. Then traverse X top down, doing *add_leaf* operations to add the other nodes to $\hat{y}$. Discard the data structures for X on all levels $\leq \ell$ and set $\sigma(r) \leftarrow \sigma(y)$.

CASE 4 $\sigma(x) = \sigma(y)$: If $\sigma > 0$, then do $l(\vec{r}, \vec{x}, \vec{y}, \ell - 1)$. If $\sigma = 0$, then combine X and Y using a trivial algorithm.

Figure 6 illustrates two *link* operations. *link*(x, y) starts with the trees of Figure 6(a) and executes Case 3 in level 3. The resulting link tree *link*(x, y) has root r , which is the nonroot node ρ in the incremental tree of Figure 6(b.2). *link*(z, r) starts with the trees of Figure 6(b) and executes Case 4 in level 3. Then it executes Case 1 for the two stage 0 trees in level 2. The resulting 2-tree (Figure 6(c.3)) is in stage 1 since it has $5 \geq 4$ nodes. All 2-nodes are in one 2-subtree, as in Figure 6(c.4). Figure 6(b.2) and (c.2) illustrate that, in general, any incremental tree may have its root pointer ρ pointing to a node at arbitrary depth.

LEMMA 5.1. *The algorithm for link(x, y) preserves all the defining properties of the data structure.*

PROOF. We sketch the argument giving only the most interesting details. Assume the book-keeping fields $\sigma(v)$ and $\widehat{v}$ are updated when node v is added to a new incremental tree.

Consider a link tree T . For every level $\ell \in [1..L]$, T_ℓ denotes the corresponding tree as defined by the data structure's parent and child pointers. T is represented correctly by T_L , by a simple induction using the Combine Step. Furthermore, for every level $\ell \in [2..L]$, $T_{\ell-1}$ is formed from T_ℓ by contracting every ℓ -subtree. (Notice that in Case 2 $T_{\ell-1}$ does not change since Y is absorbed into $\widehat{x}$. Similarly for Case 3 and $\widehat{y}$.) This implies that the vertex y is the root of its tree T_L , and $\vec{y}$ and all lower images are roots of their ℓ -trees (see especially Figure 6(b.2)). This justifies the argument $\vec{y}$ in the recursive call of Case 4.

Now consider the four cases that determine the stage and the ℓ -subtrees.

CASE 1: The new ℓ -tree belongs in stage $\sigma + 1$ because $s(r) < 4A_\ell(\sigma + 1) \leq 2A_\ell(\sigma + 2)$ by Equation (9).

CASES 2–3: In Case 2, the incremental tree $\widehat{x}$ exists, since x is in a positive stage. Similarly, in Case 3, $\widehat{y}$ exists.

CASE 4: The new ℓ -tree T_ℓ (formed in the Combine Step) is correctly partitioned into ℓ -subtrees, since X and Y were. Also, Case 4 always has $\ell > 1$. (So level $\ell - 1$ actually exists.) This is because if $\ell = 1$ and $\sigma(r) = \sigma(y) = \sigma$, then Case 1 applies since $s(r) \geq 2(2A_1(\sigma)) = 2^{\sigma+2} = 2A_1(\sigma + 1)$.

If $\sigma = 0$, then $s(r) < 4$; so, having updated T_ℓ to the combined tree, we are done. $\square$

The algorithm for $ca(x, y)$ is trivial in universe zero. In positive universes, it is the multi-level algorithm $c(x, y, \ell)$ of Section 4. The first case of the c algorithm is always used (since every ℓ -subtree is contracted to an $(\ell - 1)$ -node). It executes the code of Figure 5.

LEMMA 5.2. *Algorithm $\mathcal{A}_\ell$ executes a sequence of m ca and link operations on a set of n nodes in $O(m\ell + na_\ell(n))$ time and $O(na_\ell(n))$ space.*

PROOF. First consider the time. A ca query uses $O(\ell)$ time in a positive stage, since $O(1)$ time is spent on each of ℓ levels of recursion. The time is $O(1)$ in universe zero.

The time for *links* is estimated as follows. Charge each *link* operation $O(\ell)$ time to account for the initial computation of the root r plus the ℓ levels of recursion and associated processing in routine l (e.g., Case 4 and the Combine Step). So far, all charges are included in the term $O(m\ell)$ of the lemma.

For the rest of the time, call an *add_leaf* or *add_root* operation an *add* operation and define

$$\eta = \text{the total number of } add \text{ operations.}$$

(η includes all *add* operations done in recursive calls.) The rest of the time for l is proportional to η . Here, we are using Theorem 4.1, which shows that an incremental tree that grows to contain n_i nodes uses time $O(n_i)$ for all *add_leaf* and *add_root* operations. (Also note that discarding data structures in Cases 1–3 is just a no-op.) For the time bound of the lemma, it suffices to show $\eta = O(na_\ell(n))$. In fact, we will show by induction on ℓ that

$$\eta \leq 2na_\ell(n). \quad (12)$$

First consider the *add* operations in Cases 1–3 of level ℓ (i.e., we exclude the operations that result from a recursive call made in Case 4 from level ℓ). Each such *add* is done for a node previously in a lower stage of level ℓ . So, at most one *add* is done for each node in each stage. This gives $\leq na_\ell(n)$ *adds* total. (In particular, this establishes the base case of the induction, $\ell = 1$.)

To bound the number of *adds* in all levels $< \ell$, fix a stage $\sigma > 0$ of level ℓ . We will show there are $\leq n$ *adds* total in recursive calls made from stage σ of level ℓ . So, the $a_\ell(n)$ stages contribute

a total of $\leq na_\ell(n)$ adds in levels $< \ell$. Adding together the two bounds gives Equation (12) and completes the induction.

First note an approach that does *not* work. The inductive assumption holds for $\mathcal{A}_{\ell-1}$. So we could estimate the total number of $(\ell - 1)$ -nodes, say $n_{\ell-1}$, and use the inductive bound $2n_{\ell-1}a_{\ell-1}(n_{\ell-1})$. But this overestimates the number of adds, since $\mathcal{A}_\ell$ discards the entire data structure for an ℓ -tree as soon as it moves to a higher stage (i.e., when it has size $\geq 2A_\ell(\sigma + 1)$ in Case 1, or earlier in Cases 2 and 3). So, instead, our approach is to count the number of adds for each maximal ℓ -tree M_i of stage σ .

Let M_i have n_i vertices and s_i ℓ -subtrees. Then

$$s_i \leq n_i/2A_\ell(\sigma) \leq A_\ell(\sigma + 1)/A_\ell(\sigma), \quad (13)$$

where the first inequality uses the lower bound $2A_\ell(\sigma)$ on the size of an ℓ -subtree and the second inequality uses the upper bound $2A_\ell(\sigma + 1)$ on the size of an ℓ -tree. The $(\ell - 1)$ -tree for M_i has s_i nodes. The inductive assumption shows the number of adds to form this $(\ell - 1)$ -tree is $\leq 2s_i a_{\ell-1}(s_i)$.

Using the second inequality of Equation (13) gives

$$a_{\ell-1}(s_i) \leq a_{\ell-1}(A_\ell(\sigma + 1)/A_\ell(\sigma)) \leq a_{\ell-1}(A_\ell(\sigma + 1)) = a_{\ell-1}(A_{\ell-1}(A_\ell(\sigma))) = A_\ell(\sigma).$$

Using this and the first inequality of Equation (13) shows that the total number of adds for all $(\ell - 1)$ -trees of stage σ is at most

$$\sum_i 2s_i a_{\ell-1}(s_i) \leq 2A_\ell(\sigma) \sum_i s_i \leq 2A_\ell(\sigma) \sum_i n_i/2A_\ell(\sigma) = \sum_i n_i \leq n.$$

This bound of n adds per stage implies $\leq na_\ell(n)$ recursive adds total. This completes the induction.

Now consider the space. There are initially n nodes on level L . Additional nodes are only created in Case 1. The number of these nodes is obviously bounded by the number of *add leaf* operations and so is $\leq \eta$. Theorem 4.1 shows the space used for incremental trees is proportional to η . (As usual, all space is allocated from one global array S using Lemma 2.8.) So Equation (12) implies the desired space bound. $\square$

The remaining issue is how to choose the number of levels ℓ . Consider the usual case where bounds on m and n are known when the algorithm begins. Take $\ell = \alpha(m, n)$. By definition, $a_{\alpha(m, n)}(n) \leq 4\lceil m/n \rceil \leq m/n + 4$. So the lemma implies the following:

THEOREM 5.3. *A sequence of $\leq m$ nca and link operations on a universe of $\leq n$ nodes can be processed in time $O(m\alpha(m, n) + n)$ and space $O(m + n)$.*

Appendix E extends this result to settings where m and n are not known in advance.

The multilevel method we have used can be applied to achieve the same time and space bounds for several other problems. As mentioned earlier, Gabow [7] applies it to solve the list splitting problem that arises in *expand* steps of Edmonds' algorithm. The technique was rediscovered by Han La Poutré [14]: He presents a multilevel algorithm for the set merging problem (this application is noted in Gabow [7, p. 99]). Also, a result similar to Corollary E.1 was independently arrived at [J.A. La Poutré, personal communication]. Other applications include the static cocycle problem introduced in Gabow and Stallmann [10], both for graphic matroids and the job scheduling matroid; the former is useful for various problems involving spanning trees.

APPENDICES

A COMPUTING LOGARITHMS

We show how to compute $\lfloor \log_\beta r \rfloor$ for a given integer $r \in [1..cn^e]$ in time $O(1)$. Here, $\beta = a/b$ is a fixed rational number for positive integers $a > b$, c and e are fixed integers, $c \geq 1$, $e > 1$.

Let $k = \lfloor \log_\beta n \rfloor$. We precompute these values:

- k, a^k, b^k .
- a table $\ell[1..n]$ with $\ell[r] = \lfloor \log_\beta r \rfloor$ for $r \in [1..n]$.

We show the precomputation time is $O(n)$.

The following code precomputes the ℓ table:

```

a' = 1; b' = 1; k = -1
for r = 1 to n do
  while r ≥ a'/b' do
    a' = aa'; b' = bb'; k = k + 1
  ℓ[r] = k

```

On exit, k is the desired value $\lfloor \log_\beta n \rfloor$, and the desired values a^k and b^k are given by a'/a and b'/b , respectively. It is clear that the time is $O(n)$.

Now we give the algorithm to compute $\lfloor \log_\beta r \rfloor$ for a given integer $r \in [1..cn^e]$. Let h be the unique integer satisfying

$$\beta^{hk} \leq r < \beta^{(h+1)k}.$$

So,

$$\log_\beta r = hk + \log_\beta(r/\beta^{hk}). \quad (14)$$

Taking floors gives the desired value.

We find h by testing successive values β^{hk} . The desired h is at most $e + (e + \log_\beta c)/k = O(1)$. This follows for $r \leq cn^e$ since $c = \beta^{\log_\beta c}$ and $n < \beta^{k+1}$ implies $n^e < \beta^{ke+e}$. Using the values a^k, b^k , the time is $O(e) = O(1)$.

The desired floor of the logarithmic term in Equation (14) is $\lfloor \log_\beta \lfloor r/\beta^{hk} \rfloor \rfloor$. This corresponds to an entry in the ℓ table since $r/\beta^{hk} < \beta^k \leq n$. The desired entry is found as $\ell[r\beta^{hk}/a^{hk}]$ (since division is truncating). Again, the time is $O(1)$.

B PROOF OF SPACE ALLOCATION LEMMA 2.8

PROOF. We maintain the invariant that each current array A_i has size $2s_i$ in S when $n_i \in [s_i + 1..2s_i]$, and, furthermore, the total amount of space in S used for versions of A_i is $\leq 4s_i$. (Thus, at every point in time, every array A_i has used $\leq 4s_i < 4n_i$ entries of S ; i.e., all allocated storage is within $S[1..4n]$.)

When a new A_i is created, we set $s_i = 1$, allocate 2 cells, and set $n_i = 2$.

When a new entry is added to A_i and $n_i = 2s_i$, we allocate a new copy of A_i of size $4s_i$ at the end of S , copying the contents of $A_i[1..n_i]$ into it. Setting $s'_i = 2s_i$ and $n_i = 2s_i + 1$ the new A_i has size $4s_i = 2s'_i$ with $n_i = s'_i + 1$. A_i has used $\leq 4s_i + 4s_i = 4s'_i$ entries of S . So the invariant is preserved.

The time for the operation (copying n_i entries) is $O(n_i)$ and can be charged to the $s_i = n_i/2$ elements added to A_i since the last allocation. $\square$

C INCREMENTAL-TREE SET MERGING

Incremental-tree set merging, introduced by Gabow and Tarjan [11], is a special case of the general union-find problem. It achieves a slight asymptotic improvement of the best-known general union-find algorithm, reducing time $O(m\alpha(m, n))$ for $m \geq n$ finds in a universe of n elements to linear

time $O(m)$. It has a variety of applications [11]. It is more general than its special case, static-tree set merging. The implementation of this section simplifies the tables that must be precomputed. Specifically, Gabow and Tarjan [11] store forests in single memory cells. The precomputation step checks that bitstrings correspond to forests and performs a depth-first search of such forests. Our precomputed tables are for several natural bit operations, specifically most-significant-bit, powers of two, and logical *and*.

The incremental-tree set merging data structure maintains a collection of disjoint sets, called *i-sets*, specified by these operations:

add_leaf (x, y) - make a new node y the child of node x , and make $\{y\}$ a singleton *i-set*;
union(x) - merge the *i-set* containing x with the *i-set* containing its parent $\pi(x)$;
find(x) - return the root of the *i-set* containing x .

T denotes the tree that is grown by *add_leaf* operations. We allow the operation *add_leaf*(ϵ, x) to initialize T to consist of the root x . Our algorithm is similar to Section 3, and we use its terminology. As in Figure 3(a), $T = T_2$, and a full 2-subtree gets added to T_1 as a 1-node. We use the following data structure.

Each 2-subtree uses a bitstring data structure based on Section 3 and specified as follows. *id*, *v*, and *anc* are defined and maintained as in Section 3. In addition, the 2-subtree has a bitstring *roots* of $\lfloor \log n \rfloor$ bits. The i th bit of *roots* is 1 if and only if node number i is the root of its *i-set*. In other words, the i th bit is 0 if and only if *union*($v[i]$) has been performed. The *roots* bitstring is initialized to $2^{\lfloor \log n \rfloor} - 1$ when the 2-subtree is created. (This makes the 0th bit of *roots* equal to 1, but this is irrelevant.)

The operation *union*(x) is implemented as

$$roots[\widehat{x}] \leftarrow roots[\widehat{x}] - 2^{id[x]}.$$

Nothing is done in T_1 even if $x = \rho(\widehat{x})$: Unions in T_1 are done in a lazy fashion, as explained later. For any vertex x , define

$$\beta(x) = anc[x] \wedge roots[\widehat{x}].$$

If $\beta(x) \neq 0$, it is easy to see $find(x) = v[msb[\beta(x)]]$. Here, v denotes the v table for $\widehat{x}$.

T_1 is processed using a set-merging data structure on the 1-nodes. A simple choice is the “relabel-the-smaller-half” algorithm, where a *union* operation uses $O(\log n)$ time and *find* is $O(1)$ [5]. For clarity, call the operations in T_1 *union*₁ and *find*₁. We will manipulate T_1 -sets like *i-sets* (i.e., *union*₁ will always merge a T_1 -set rooted at x into the T_1 -set containing $\pi(x)$).

We will maintain T_1 -sets to preserve this invariant:

- (I) For arbitrary 1-nodes x and $y = find_1(x)$, $\widehat{x}$ and $\widehat{y}$ are in the same *i-set*. Furthermore, $y \neq x$ implies $\widehat{y}$ is not an *i-set* root.

For a vertex x , the operation *find*(x) is implemented as follows.

```

if  $\widehat{x}$  is nonfull and  $\beta(x) = 0$  then  $x \leftarrow \pi(\rho(\widehat{x}))$ 
if  $\beta(x) = 0$  then
    loop /*  $x$  is always a vertex of  $T$ ,  $y$  is always a 1-node */
         $y \leftarrow find_1(\widehat{x})$  /*  $\beta(x) = 0$  */
         $x \leftarrow \pi(\widehat{y})$ 
        if  $\beta(x) \neq 0$  then break
        union1( $y, \widehat{x}$ )
    return  $v[msb[\beta(x)]]$ 

```

It is easy to see invariant (I) is maintained, and the *find* algorithm is correct. (Note that $\overline{y}$ is never an *i*-set root. This follows from invariant (I) when $\overline{x} \neq \text{find}_1(\overline{x})$ and from the condition $\beta(x) = 0$ when $\overline{x} = \text{find}_1(\overline{x})$.)

Next, we show the time for m *find* operations and n *unions* is $O(m + n)$.

A *union* operation is done in a 2-subtree data structure and so takes $O(1)$ time. At most $n/\lfloor \log n \rfloor$ *union*₁ operations are done in executions of *find*. (Each *union*₁ is nontrivial, i.e., it starts with y and $\overline{x}$ in different T_1 -sets. This follows from the preceding *find*₁ operation and redefinition of x .) So *union*₁ operations use total time $O((n/\log n) \log n) = O(n)$.

We turn to *find* operations. It is easy to see that, ignoring time for *union*₁s, a *find* uses $O(1)$ time plus the time for all executions of *find*₁. The total number of executions of *find*₁ is $\leq m + n/\lfloor \log n \rfloor$. (The m term accounts for the first execution of *find*₁ in all *find*s. The remaining executions of *find*₁ all follow a *union*₁, so they number $\leq n/\lfloor \log n \rfloor$ as earlier.) This implies that *find*s use total time $O(m + n)$ as desired.

D SIMPLE INEQUALITIES FOR ACKERMANN'S FUNCTION

PROOF OF EQUATION (9), $A_i(j + 1) \geq 2A_i(j)$ FOR $i, j \geq 1$:

First note the trivial inequality $2^i \geq 2i$ for $i \geq 1$. Also, for every $i \geq 1$, $A_i(2) = 4$.

Next, we show $A_i(j) \geq 2^j$ by induction on i , with the inductive step inducing on j . The base case $j = 2$ of the inductive step is $A_i(2) = 4 = 2^2$. For the inductive step

$$A_i(j) = A_{i-1}(A_i(j-1)) \geq 2^{A_i(j-1)} \geq 2^{2^{j-1}} \geq 2 \cdot 2^{j-1} = 2^j.$$

Now, Equation (9) itself follows from the first $\geq$ relation displayed earlier and $2^{A_i(j-1)} \geq 2A_i(j-1)$. $\square$

PROOF OF (10), $A_{i+1}(j) \geq A_i(2j)$ FOR $i \geq 1, j \geq 4$:

Note that Equation (10) needn't hold for $j < 4$: Recalling that A_2 is superexponentiation $A_2(j) = 2 \uparrow j$, $A_2(3) = 16 < 64 = A_1(6)$. However, Equation (10) holds for $i = 1, j = 4$: $A_2(4) = 2^{16} > 2^8 = A_1(8)$. In general, for $i \geq 1$,

$$A_{i+1}(4) = A_i(A_{i+1}(3)) = A_i(A_i(A_{i+1}(2))) = A_i(A_i(4)).$$

Also, A_i is an increasing function by Equation (9).

We prove Equation (10) by induction on i . For the base case, $A_{i+1}(4) = A_i(A_i(4)) \geq A_i(2^4) > A_i(8)$. For the inductive step, $A_{i+1}(j + 1) = A_i(A_{i+1}(j))$. The argument to A_i is $A_{i+1}(j) > A_i(2j) \geq 2^{2j} \geq 2(2j) \geq 2j + 2$. $\square$

PROOF OF EQUATION (11), $\alpha(m', n') \geq \alpha(m, n) - 1$ FOR $m' \leq 2m, n' \geq n$:

Let $i = \alpha(m, n)$. We wish to show $\alpha(m', n') \geq i - 1$ (i.e., $A_{i-2}(4\lceil \frac{m'}{n'} \rceil) < n'$). This follows since

$$A_{i-2}\left(4\left\lceil \frac{m'}{n'} \right\rceil\right) \leq A_{i-2}\left(8\left\lceil \frac{m}{n} \right\rceil\right) \leq A_{i-1}\left(4\left\lceil \frac{m}{n} \right\rceil\right) < n \leq n'.$$

The first inequality uses $\lceil m'/n' \rceil \leq \lceil 2m/n \rceil \leq 2\lceil m/n \rceil$. The second uses Equation (10), which applies since $4\lceil \frac{m}{n} \rceil \geq 4$. (A slightly more involved calculation shows $\alpha(m', n') \leq \alpha(m, n) + 2$ when $m' \geq m, n' \leq 2n$, but we do not use this fact.) $\square$

E NCAS WITH LINKS AND MAKE_NODES

This appendix extends the dynamic *nca* algorithm of Section 5 to settings where m and n are not known in advance. More precisely, in addition to *nca* and *link* operations, we allow the operation *make_node*(x), which creates a new node x in a singleton tree. We prove the exact analog of Theorem 5.3 in the new setting (Corollary E.1).

It is convenient to assume that new nodes x are not counted in n until they are involved in a *link* operation. (Since $\alpha(m, n)$ is increasing with n , this can only make the desired time bound of Theorem 5.3 stronger.) Since a *link* increases n by ≤ 2 , we always have

$$m \geq n/2.$$

Our basic technique is a doubling strategy. But two difficulties must be overcome: First, $\alpha(m, n)$ is decreasing with m . So, the time bound for a *ca* operation can decrease as the algorithm progresses. Second, the term $n\alpha_\ell(n)$ in the bound of Lemma 5.2 does not change in a predictable way.

We begin by describing our new procedure. It uses algorithm $\mathcal{A}_\ell$ where ℓ is repeatedly modified. In precise terms, the sequence of operations is divided into *periods*. The parameters n and m denote their values at the start of a period. The period processes *nca* and *link* operations using algorithm $\mathcal{A}_\ell$ where $\ell = \alpha(m, n)$. (Note that the first period begins with the execution of the first *link*; i.e., $m = 1, n = 2, \ell = 1$.) The period continues as long as the value of α remains in $\{\ell - 1, \ell\}$. In other words, we declare a new period whenever n' (the current number of *links*), m' (the current number of *links* and *ncas*), and $\ell' = \alpha(m', n')$, have $\ell' > \ell$ or $\ell' < \ell - 1$. The last period ends at the conclusion of the algorithm (α need not have changed). In precise terms, the algorithm is as follows.

Before executing the current *nca/link* operation, update $n', m',$ and ℓ' to include that operation. If $\ell' \in \{\ell - 1, \ell\}$, then execute the operation. Otherwise, do the following:

Set $n \leftarrow n', m \leftarrow m', \ell \leftarrow \ell'$. Reorganize the entire data structure to use algorithm $\mathcal{A}_\ell$. Do this by making each current link tree T an incremental tree and placing it in the appropriate stage for $\mathcal{A}_\ell$. If $\ell > 1$, add a corresponding node in stage 0 of level $\ell - 1$. Finally, execute the current *nca/link* operation.

This procedure clearly handles *links* and *ncas* correctly. The time to start a new period is $O(n)$ for the new value of n . (This includes the time to compute a new *ackermann* table, find ℓ , compute new incremental tree tables, and find the stage for each incremental tree. All tables are computed for the value $2n$; e.g., *ackermann* stores all values $A_i(j) \leq 2n$.)

Now we prove that the procedure achieves our goal. Note that the resource bounds of the following corollary are essentially the same as Theorem 5.3.

COROLLARY E.1. *A sequence of *nca* and *link* operations can be processed in time $O(m\alpha(m, n))$ and space $O(m)$. Here, m is the number of *ncas* and *links*, n is the number of *links*, and neither is known in advance.*

PROOF. The bulk of the argument establishes the time bound. We will charge processing time to the counts n and m . To do this, define a *unit* of time to be enough to pay for any constant amount of computing in the algorithm's time bound.

As in the algorithm, the analysis uses $n, m,$ and ℓ to denote the values when the first *nca/link* operation of the period is executed. In addition, we use $\bar{n}$ and $\bar{m}$ to denote the counts up to and including the last operation of the period. (So $\alpha(\bar{n}, \bar{m}) \in \{\ell - 1, \ell\}$.) We also use n', m', ℓ' to denote those values for the first operation of the next period. (So $m' = \bar{m} + 1$. This holds for the last period by convention. We also take $n' = \bar{n}$ for the last period.) Note that $m' - m$ is the number of operations in the period.

The proof of the time bound consists of these three claims.

CLAIM 1. *The time for any period is at most $(m' - m)\ell + 12\bar{m}$ units.*

CLAIM 2. *The total time from the start of the algorithm to the end of any period but the last is at most $(12\ell' + 50)\bar{m}$ units. (As earlier, $\ell' = \alpha(m', n')$.)*

CLAIM 3. *When the last period ends, the total time for the entire algorithm is at most $(12\ell + 62)\bar{m}$ units. The time bound of the corollary holds.*

In Claim 1, all parameters (e.g., ℓ) are defined for the period under consideration. In Claim 3, all parameters are defined for the last period.

PROOF OF CLAIM 1. Lemma 5.2 shows the time for a period is $O((m' - m)\ell + \bar{n}a_\ell(\bar{n}))$. So we need only show $\bar{n}a_\ell(\bar{n}) \leq 12\bar{m}$.

First observe

$$a_\ell(\bar{n}) \leq 4 \left\lceil \frac{\bar{m}}{\bar{n}} \right\rceil. \quad (15)$$

This is equivalent to $A_\ell(4\lceil \frac{\bar{m}}{\bar{n}} \rceil) \geq \bar{n}$. This inequality holds by definition if $\alpha(\bar{m}, \bar{n}) = \ell$. The other possibility is $\alpha(\bar{m}, \bar{n}) = \ell - 1$. But this also implies the same inequality since we have $A_\ell(4\lceil \frac{\bar{m}}{\bar{n}} \rceil) \geq A_{\ell-1}(4\lceil \frac{\bar{m}}{\bar{n}} \rceil) \geq \bar{n}$.

The right-hand side of Equation (15) is $< 4(\bar{m}/\bar{n} + 1)$. Thus, $\bar{n}a_\ell(\bar{n}) \leq 4(\bar{m} + \bar{n})$. Using $\bar{m} \geq \bar{n}/2$, the last quantity is bounded by $4\bar{m} + 8\bar{m} = 12\bar{m}$. $\diamond$

We prove Claims 2 and 3 by charging each *nca/link* at most γ time units, where γ is $12\ell' + 50$ in Claim 2 and $12\ell + 62$ in Claim 3.

CLAIM 2 IMPLIES CLAIM 3. Claim 2 shows that the total time from the start of the algorithm to the beginning of the last period is accounted for by a charge of $\gamma = 12\ell + 50$, where $\ell = \alpha(m, n)$ is the value used in the data structure of the last period. (This holds *a fortiori* if the last period is actually the first period.)

Account for the time in the last period in two steps. First account for the term $(m' - m)\ell$ in Claim 1 by charging each *nca/link* of the last period ℓ units. Since $\ell < \gamma$, every *nca/link* is charged $\leq \gamma$ units. Next account for the term $12\bar{m}$ in Claim 1 by increasing γ by 12, so the new charge is $12\ell + 62$ units. This gives the first part of Claim 3.

The final value of α is $\geq \ell - 1$. So, using the first part of Claim 3 and changing $\bar{m}$ to the parameter m of the corollary, the time bound for the entire algorithm is $O(m(\alpha(m, n) + 1)) = O(m\alpha(m, n))$. Claim 3 is now completely proved. $\diamond$

PROOF OF CLAIM 2. Assume Claim 2 holds at the end of the previous period. Now switch to the notation of the current period. (The value ℓ' in Claim 2 becomes the parameter of the current period, ℓ .) So, each operation preceding the current period is charged $\gamma = 12\ell + 50$ units. Let ℓ' now denote the value of α after the current period ends. We wish to show the total time is accounted for by charging every operation $\gamma' = 12\ell' + 50$ units. Consider the two possibilities for ℓ' .

CASE $\ell' \geq \ell + 1$: Account for the first term of Claim 1 by charging each *nca/link* of the current period ℓ units. Certainly, $\ell < 12\ell + 50$. So now every *nca/link* is charged $\leq \gamma$ units. Account for the second term by increasing γ to $\gamma + 12 = 12(\ell + 1) + 50 \leq 12\ell' + 50$. This gives Claim 2 for the current period. (This case applies when the current period is the first period, since $\ell = 1$.)

CASE $\ell' \leq \ell - 2$: If $m' \leq 2m$, then Equation (11) implies $\ell' = \alpha(m', n') \geq \alpha(m, n) - 1 = \ell - 1$. So

$$m' > 2m.$$

Equation (11) also implies $\ell' = \alpha(m', n') \geq \alpha(\bar{m}, \bar{n}) - 1$ since $m' = \bar{m} + 1 \leq 2\bar{m}$. Using $\alpha(\bar{m}, \bar{n}) \geq \ell - 1$ gives $\ell' = \alpha(m', n') \geq \ell - 2$. With the inequality assumed for the current case, we get

$$\ell' = \ell - 2.$$

Since $m'/2 > m$, we have $m' - m > m'/2$. So, we can account for the second term of Claim 1 by charging each *nca/link* of this period 24 units ($24(m'/2) = 12m' > 12\bar{m}$). The charge for the first term of Claim 1 is ℓ . So, the total charge to each new time period is $\ell + 24 = (\ell' + 2) + 24 = \ell' + 26$.

Each of the first $m - 1$ operations is currently charged $\gamma = 12\ell + 50 = 12(\ell' + 2) + 50$. Transfer 24 units to an operation of the current period. (Permissible since $m' - m > m > m - 1$.) The first $m - 1$ operations are now each charged $\gamma' = 12\ell' + 50$. The operations of the current period are each charged $\leq (\ell' + 26) + 24 < 12\ell' + 50 = \gamma'$. So, γ' accounts for all the time so far. $\diamond$

Finally, consider the space. Lemma 5.2 shows the space for each period is $O(\bar{n}a_\ell(\bar{n}))$. The argument of Claim 1 shows that any period has $\bar{n}a_\ell(\bar{n}) = O(\bar{m})$. Since $\bar{m}$ is at most the final value of m , and the space for a period is always reused, the space bound follows. $\square$

ACKNOWLEDGMENTS

The author thanks several anonymous referees for their careful reading and many suggestions.

REFERENCES

- [1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. 1976. On finding lowest common ancestors in trees. *SIAM J. Comput.* 5 (1976), 115–132.
- [2] M. A. Bender and M. Farach-Colton. 2000. The LCA problem revisited. *Latin American Theoretical Informatics (LATIN) 2000* (2000), 88–94.
- [3] O. Berkman and U. Vishkin. 1993. Recursive star-tree parallel data structure. *SIAM J. Comput.* 22, 2 (1993), 221–242.
- [4] R. Cole and R. Hariharan. 2005. Dynamic LCA queries on trees. *SIAM J. Comput.* 34, 4 (2005), 894–923.
- [5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. 2001. *Introduction to Algorithms* (2nd ed.). McGraw-Hill, New York.
- [6] J. Edmonds. 1965. Maximum matching and a polyhedron with 0,1-vertices. *J. Res. Nat. Bur. Standards* 69B (1965), 125–130.
- [7] H. N. Gabow. 1985. A scaling algorithm for weighted matching on general graphs. In *Proc. 26th Annual Symp. on Found. of Comp. Sci.* IEEE Computer Society, Long Beach, CA, 90–100.
- [8] H. N. Gabow. 1990. Data structures for weighted matching and nearest common ancestors with linking. In *Proc. 1st Annual ACM-SIAM Symp. on Disc. Algorithms*. SIAM, Philadelphia, PA, 434–443.
- [9] H. N. Gabow. 2016. Data structures for weighted matching and extensions to b -matching and f -factors. (2016). Available at <https://arxiv.org/abs/1611.07541>.
- [10] H. N. Gabow and M. Stallmann. 1985. Efficient algorithms for graphic matroid intersection and parity. In *Automata, Languages and Programming: 12th Colloquium; Lecture Notes in Computer Science 194*, W. Brauer (Ed.). Springer-Verlag, 210–220.
- [11] H. N. Gabow and R. E. Tarjan. 1985. A linear-time algorithm for a special case of disjoint set union. *J. Comp. and System Sci.* 30, 2 (1985), 209–221.
- [12] D. Gusfield. 1999. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, New York.
- [13] D. Harel and R. E. Tarjan. 1984. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.* 13, 2 (1984), 338–355.
- [14] J. A. La Poutré. 1990. New techniques for the union-find problem. In *Proc. 1st Annual ACM-SIAM Symp. on Disc. Algorithms*. SIAM, Philadelphia, PA, 54–63.
- [15] B. Schieber and U. Vishkin. 1988. On finding lowest common ancestors: Simplification and parallelization. *SIAM J. Comput.* 17, 6 (1988), 1253–1262.
- [16] A. Schrijver. 2003. *Combinatorial Optimization: Polyhedra and Efficiency*. Springer, New York.
- [17] D. D. Sleator and R. E. Tarjan. 1983. A data structure for dynamic trees. *J. Comp. and System Sci.* 26 (1983), 362–391.
- [18] B. Smyth. 2003. *Computing Patterns in Strings*. Pearson Education Limited, Harlow, England.
- [19] R. E. Tarjan. 1979. Applications of path compression on balanced trees. *J. ACM* 26, 4 (1979), 690–715.
- [20] R. E. Tarjan. 1983. *Data Structures and Network Algorithms*. SIAM, Philadelphia, PA.

Received December 2014; accepted May 2017