Disjoint Set Union with Randomized Linking

Ashish Goel* Sanjeev Khanna† Daniel H. Larkin‡ Robert E. Tarjan§

Abstract

A classic result in the analysis of data structures is that path compression with linking by rank solves the disjoint set union problem in almost-constant amortized time per operation. Recent experiments suggest that in practice, a naïve linking method works just as well if not better than linking by rank, in spite of being theoretically inferior. How can this be? We prove that *randomized* linking is asymptotically as efficient as linking by rank. This result provides theory that matches the experiments, which implicitly do randomized linking as a result of the way the input instances are generated.

1 Disjoint Set Union via Compressed Trees

The *disjoint set union problem*, also called the *union-find problem*, is to maintain a collection of disjoint sets, each with a distinguished *root* element, under an intermixed sequence of the following two kinds of operations:

FIND (x): Return the root of the set containing element x.

UNITE (x, y): If elements x and y are in the same set, return **false**; otherwise, form the union of the sets containing x and y (destroying the old sets), choose a root for the new set, and return **true**.

Initially each set is a singleton, whose only element is its root. In each UNITE, the implementation is free to choose the root of the new set. Information associated with a set can be stored in its root.

The *compressed tree* solution to this problem [6] represents each set by a rooted tree whose nodes are the elements of the set and whose root is the root of the set. Each node x has a pointer $x.p$ to its parent; each root points to itself. To do FIND (x), follow parent pointers from x until reaching a node u pointing to itself. The path of ancestors from x to u is the FIND *path*. To do UNITE (x, y), first compute $u =$ FIND (x) and $v =$ FIND (y). If $u = v$, return **false**; otherwise, make one of u and v the parent of the other (thereby making the new parent the root of the new set), and return **true**.

By exploiting the flexibility inherent in this solution, one can significantly improve its efficiency. The implementation is free to restructure the trees, as long as it preserves the node set of each tree. One way to gain efficiency is to do *path compaction*: during a FIND, replace the parent of one or more nodes along the FIND path by a node farther along the path. The most drastic form of compaction is *compression*, attributed by Knuth [9] to Alan Tritter: during FIND (x), make the root of the tree containing x the parent of every node on the FIND path. Compression requires two passes over the FIND path, one to find the root, the other to update parents. One can do both passes bottom-up (from x to the root). Or, one can do the first pass bottom-up and the second top-down using recursion.

Alternatives to compression include two one-pass methods: *splitting* [18, 19], which for every ancestor y of x replaces the parent of y by its grandparent, and *halving* [18, 19], which for every other ancestor y of x replaces the parent of y by its grandparent. Both of these methods do less compaction than does compression. See the appendix for pseudocode implementing FIND with compression, splitting, and halving.

A second way to gain efficiency is to choose the root of each new tree carefully, using a *linking rule*. The baseline for comparison is *naïve linking*, which during UNITE (x, y) chooses FIND (x) as the new root. A rule with similar efficiency is *linking by index* [6], which requires that the elements be totally ordered and which chooses as the new root the larger of FIND (x) and FIND (y). Two better rules are *linking by size*, attributed by Knuth [9] to M. D. McIlroy, which chooses as the new root the root of the old tree having more nodes, breaking a tie arbitrarily; and *linking by rank* [17], which chooses as the new root the root of the old tree having greater height, ignoring any compaction and breaking a

*Department of Management Science and Engineering, Stanford University. Research supported in part by NSF grant 0904325 and the DARPA GRAPHS and XDATA programs. Email: ashishg@stanford.edu

†Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104. Email: sanjeev@cis.upenn.edu. Supported in part by National Science Foundation grants CCF-1116961 and IIS-0904314.

‡Princeton University Department of Computer Science. Email: dhlarkin@cs.princeton.edu.

§Princeton University Department of Computer Science and MSR SVC. Email: ret@cs.princeton.edu. Research of Daniel H. Larkin and Robert E. Tarjan at Princeton University partially supported by NSF grant CCF-0832797.

tie arbitrarily. The implementation of linking by size or rank requires storing the number of nodes or the height, respectively, in the root.

In some applications of disjoint set union, only the sets must be maintained, not additional information such as set names. A notable example is Kruskal's minimum spanning tree algorithm [11], which maintains the vertex sets of connected components using UNITE operations. In such applications there is an additional way to speed up each UNITE: interleave the two FIND operations, and stop when one of them finishes. One can alternate steps along the two paths, or use an ordering to determine the interleaving. Two methods that use the latter idea are *early linking by index*, analogous to linking by index, and *early linking by rank*, analogous to linking by rank, both described below. There is no similar way to do early linking by size: this would require increasing the size of the new root; but this root, which would be returned by the second FIND to finish, is in general not yet reached when the first FIND finishes. For the same reason, early linking by index or rank cannot be used if the roots store additional information about the sets.

Early linking by index requires that the elements be totally ordered. It does UNITE(x, y) by traversing the FIND paths from x and y concurrently, proceeding at each step from the node with the smaller parent. If it reaches two nodes with the same parent, it returns **false**; if it reaches a root, it makes this root a child of the parent of the current node on the other FIND path (thereby preserving increasing order along FIND paths and making the not-yet-found root the root of the new tree) and returns **true**.

Early linking by rank is like early linking by index except that it uses ranks to maintain a total order along each FIND path. This produces an extra case in the implementation: if the FIND operations reach two different roots of the same rank, one becomes the parent of the other, and the rank of the new parent increases by one.

One can combine any type of path compaction with any of the linking rules. In a UNITE that does early linking by index or rank, compaction proceeds only as far as the two FIND paths are traversed. Instead of compacting these paths individually, one can instead *splice* them together: when taking a step from a node on one FIND path, replace its parent by the parent of the current node on the other path. Splicing applies only to the pairs of FIND operations within UNITE operations; another kind of compaction must be done during any FIND that occurs outside of a UNITE. Dijkstra [4] attributes to M. Rem the method of early linking by index with splicing; an implementation appears in the appendix. Tarjan and van Leeuwen [17] invented the method of early linking by rank with splicing; see their paper for an implementation.

What is the theoretical efficiency of these methods? Tarjan [16] showed that with linking by size and compression, the worst-case total time for m operations on sets containing a total of n elements is $\mathrm{O}\left(m\alpha\left(\lg n, {}^{m}/_{n}\right)\right)$. Here lg is the base-two logarithm and α is a functional inverse of Ackermann's function defined in Section 3, which grows very slowly and is constant for all practical purposes. (In particular, $\alpha\left(n, {}^{m}/_{n}\right) = \alpha\left(\lg n, {}^{m}/_{n}\right) + \mathrm{O}\left(1\right)$.) Tarjan and van Leeuwen [17] extended Tarjan's work to show that the asymptotic efficiency depends only on the kind of linking done, not on the kind of compaction (as long as every FIND path is compacted): with naïve linking, linking by index, or early linking by index, the worst-case total time is $\mathrm{O}\left(m \log_{{}^{m}/_{n}+2} n\right)$; with linking by rank or size or early linking by rank, the total time is $\mathrm{O}\left(m\alpha\left(\lg n, {}^{m}/_{n}\right)\right)$. Fredman and Saks [5] showed that in the cell probe computation model any algorithm that solves the disjoint union problem takes $\Omega\left(m\alpha\left(\lg n, {}^{m}/_{n}\right)\right)$ total time. Thus all the methods that use linking by size or rank are asymptotically optimal.

Later work simplified and slightly improved the analysis that produces the inverse-Ackermann-function upper bound. Kozen [10] gave a simplified version of the Tarjan-van Leeuwen analysis for linking by rank with compression. Kaplan, Shafrir, and Tarjan [8] improved Kozen's analysis to give a local amortized bound, in which the time for FIND(x) depends only on the number of elements in the current set containing x, rather than on n. Alstrup, Gørtz, Rauhe, Thorup, and Zwick [3] simplified this analysis and made it into a potential-based argument. All these results bound the total length of FIND paths by counting parent changes using a bottom-up approach. Seidel and Sharir [13] used a top-down approach to obtain the same global bounds as Tarjan and van Leeuwen.

But what happens in practice? The most recent and most comprehensive experiments were done by Ali Patwary, Blair, and Manne [2]. They tested the algorithms described above, and others, on instances derived from running Kruskal's algorithm on several families of graphs. In this application, the only FIND operations are those within UNITE operations. A main conclusion of their study is that linking by index and early linking by index perform at least as well and often better than linking by rank or size and early linking by rank, in spite of the theoretical superiority of the latter methods.

We provide theoretical support for this conclusion. We analyze randomized linking and randomized early

linking, which are linking by index and early linking by index with the elements ordered uniformly at random. These are in effect the linking methods used in the experiments of Ali Patwary et al. on at least two of their three data sets, which contained graphs generated at random. The third data set contained "real-world" graphs, which may well have had their vertices given in random order. (Their paper is silent on this issue.) We show that with any kind of compaction, both randomized linking and early randomized linking take $O(m\alpha(\lg n, m/n))$ expected time. Since the lower bound of Fredman and Saks applies to randomized algorithms as well as deterministic ones, this bound is tight to within a constant factor. We conclude that in a setting in which the instances are independent of the identities of the nodes, the added complexity of maintaining ranks or sizes does not produce improved asymptotic efficiency.

The remainder of our paper contains five sections and an appendix. We begin in Section 2 by defining node ranks for randomized linking and randomized early linking and establishing properties of these ranks. In Section 3 we use the results of Section 2 in combination with a known analysis of linking by rank to analyze randomized linking and randomized early linking with compression. In section 4 we extend the analysis of Section 3 to splitting and halving. In Section 5 we analyze randomized early linking with splicing. We conclude in Section 6 with some brief remarks and open problems. The appendix contains implementations of the compaction methods we analyze.

2 Ranks for Randomized Linking

To analyze randomized linking and randomized early linking, we define a rank for each node and extend one of the known analyses of linking by rank. A natural approach is to let the rank of a node be its height in the forest built by the UNITE operations, ignoring compactions. This idea leads to a successful analysis of all the methods except randomized early linking with splicing. The problem with splicing is that heights are not necessarily monotonic in the node order, and a splice can make a smaller-height node the parent of a larger-height one. To avoid this problem, we use an alternative definition of rank that *is* monotonic in the node numbers. This allows us to successfully analyze all the methods.

Assume that the elements are permuted uniformly at random, numbered from 1 to n in permutation order, and identified by number. For each element x from 1 to n, let the *rank* $x.r$ of x be $\lfloor \lg n \rfloor - \lfloor \lg(n - x + 1) \rfloor$. Thus the rank of n is $\lfloor \lg n \rfloor$, the rank of $n-2$ and $n-1$ is $\lfloor \lg n \rfloor - 1$, the rank of $n-6$ through $n-3$ is $\lfloor \lg n \rfloor - 2$, and so on. We use these ranks in the analysis only: they do not affect the execution of the set operations, and they remain fixed throughout the execution. The ranks have three crucial properties, all immediate from the definition: (i) if $x < y$, then $x.r \le y.r$; (ii) the number of elements of rank k is at most $n/2^k$; and (iii) for any x, at least half the elements greater than x have ranks strictly greater than $x.r$.

To bound the time of a sequence of set operations, we count the number of parent changes caused by compaction: the total time of the operations is at most a constant times this number plus $O(1)$ per FIND. The known analyses of linking by rank rely on two facts: ranks strictly increase along each FIND path, and compression, splitting, and halving replace parents by ancestors. (Splicing has a more complicated effect that we study in Section 5.) With randomized linking or randomized early linking, property (i) implies that ranks increase along each FIND path, but not necessarily strictly. We need to bound the expected number of rank ties. Suppose the set operations are done using randomized linking or randomized early linking, but with no compaction. We shall prove that for every node x, the expected number of proper ancestors of x having the same rank as x after all the unites are done is at most two.

Let x be any node, and let σ be the sequence of UNITE operations that build the final set containing x, ignoring any UNITE operations that return **false**. Each successive UNITE in σ combines the set containing x with another set of arbitrary size. It is especially easy to analyze the ancestors of x produced by σ if each successive unite adds only a single element to the set containing x: either the new element becomes the root, and hence an ancestor of x, or the old root remains the root, and the new element becomes a non-ancestor of x. To handle the general case, we transform it into the case of adding one element at a time. To do this we reorder σ into a sequence $\sigma(x)$ of σ that depends on x but not on the random numbering, and that adds elements to the set containing x one-at-a-time. We analyze the ancestors of x produced by $\sigma(x)$ and relate them to those produced by σ: with randomized linking the ancestors of x produced by σ are a subset of those produced by $\sigma(x)$, and with randomized early linking the ancestors of x produced by σ are exactly those produced by $\sigma(x)$.

The sequence $\sigma(x)$ is σ reordered so that each successive UNITE adds a new element to the set containing x, with ties broken by order in σ. We develop a recursive characterization of $\sigma(x)$ below. Let $x_0 = x, x_1, x_2, \ldots$ be the successive vertices added by $\sigma(x)$ to the set containing x. We call x_j a *prefix maximum* if $x_i < x_j$ for

$i < j$.

LEMMA 2.1. *If $\sigma(x)$ is done using randomized linking or randomized early linking with no compaction, the ancestors of x are exactly the prefix maxima among the x_j.*

Proof. For any j, the root of the tree built by the first j UNITEoperations in $\sigma(x)$ is the maximum x_i such that $i \leq j$, which is a prefix maximum. Conversely, if x_j is a prefix maximum, it will be the root of the tree built by the first j UNITE operations in $\sigma(x)$.

Let UNITE(y, z) be the last UNITE in σ, and let σ' be σ with this UNITE deleted. Let S_1 containing y and S_2 containing z be the two sets built by σ', and let σ_1 and σ_2 be the subsequences of σ' that build S_1 and S_2, respectively.

LEMMA 2.2. *If x is in S_1, $\sigma(x)$ is $\sigma_1(x)$ followed by* UNITE(y, z) *followed by $\sigma_2(x)$; if u is in S_2, $\sigma(x)$ is $\sigma_2(x)$ followed by* UNITE(y, z) *followed by $\sigma_1(y)$.*

Proof. Assume x is in S_1; the argument is symmetric if x is in S_2. The UNITE operations in σ_1 have both inputs in S_1, and the UNITE operations in σ_2 have both inputs in S_2. In the construction of $\sigma(x)$ according to its definition, if some UNITE in σ_1 has not yet been selected, there will be an unselected UNITE in σ_1 that adds a new element to the set containing x. Such a UNITE will be selected before UNITE(y, z) by the tie-breaking rule. Once all the UNITE operations in σ_1 have been selected, UNITE(y, z) will be selected. Since no UNITE in σ_2 has a vertex in S_1 as an input, the UNITE operations in σ_2 will be selected in the order they occur in $\sigma_2(z)$.

LEMMA 2.3. *Let T and $T(x)$, respectively, be the trees built by executing σ and $\sigma(x)$ using randomized linking. Then every ancestor of x in T is an ancestor of x in $T(x)$.*

Proof. The proof is by induction on the length of σ using Lemma 2.2. Assume x is in S_1; the argument is symmetric if x is in S_2. Let T_1 and $T_1(x)$ be the trees built by σ_1 and $\sigma_1(x)$, respectively. Let v and w be the largest elements in S_1 and S_2, respectively. If $v > w$, the ancestors of x in T are exactly the same as in T_1, so the lemma holds. If $v < w$, the ancestors of x in T are w and those in T_1. In this case w is maximum in T, hence a prefix maximum in $\sigma(x)$, and hence an ancestor of x in $T(x)$ by Lemma 2.1, so the lemma holds in this case also. By the induction hypothesis every ancestor of x in T_1 is an ancestor of x in $T_1(x)$.

LEMMA 2.4. *Let T and $T(x)$, respectively, be the trees built by executing σ and $\sigma(x)$ using randomized early linking. Then the proper ancestors of x are the same in T and $T(x)$.*

Proof. The proof is by induction on the length of σ but is more complicated than the proof of Lemma 2.3. Assume x is in S_1; the argument is symmetric if x is in S_2. Let T_1, T_2, $T_1(x)$, and $T_2(z)$ be the trees built by σ_1, σ_2, $\sigma_1(x)$, and $\sigma_2(z)$, respectively. Let v be the largest element in S_1. The ancestors of x in T are exactly the ancestors of x in T_1, plus the ancestors of z in T_2 that are greater than v. By the induction hypothesis, these are exactly the ancestors of x in $T_1(x)$, plus the ancestors of z in $T_2(z)$ that are greater than v. Recall that $x_0, x_1, \ldots$ is the sequence of elements added to the set containing x by $\sigma(x)$. Let $z = x_k$. By Lemma 2.2, $x_k, x_{k+1}, \ldots$ is the sequence of elements added by $\sigma_2(z)$ as it builds $T_2(z)$. By Lemma 2.1, the ancestors of x in $T_1(x)$ are the prefix maxima in $x_0, x_1, \ldots, x_{k-1}$, and the ancestors of z in $T_2(z)$ are the prefix maxima in $x_k, x_{k+1}, \ldots$. Among the latter, those greater than v are exactly the ones that are also prefix maxima in $x_0, x_1, \ldots$. The lemma follows.

THEOREM 2.1. *With randomized linking or randomized early linking but no compaction, the expected number of proper ancestors of x of the same rank as x in the final forest is at most two.*

Proof. By Lemmas 2.3 and 2.4 it suffices to bound the expected number of ancestors of x having the same rank as x in the tree built by $\sigma(x)$. By Lemma 2.1 these are exactly the prefix maxima of $x_0, x_1, \ldots$ that are of the same rank as x, all of which precede the first prefix maximum (if any) of rank greater than that of x. Let $k > 0$ be such that if $i < k$, x_i has rank at most that of x, and let x_j be maximum among $x_0, x_1, \ldots, x_{k-1}$. Since the definition of $\sigma(x)$ is independent of the random numbering, every element greater than x_j is equally likely to be x_k. (This need not be true for elements less than x_k, but that is irrelevant to our argument.) Property (iii) implies that, given that x_k is a prefix maximum, the conditional probability that the rank of x_k is greater than that of x_j is at least $1/2$. Thus each successive prefix maximum has probability at least $1/2$ of having greater rank than x. It follows that the expected number of proper ancestors of x of the same rank as x is at most $\sum_{i=1}^{\infty} i/2^i = 2$. (Node x is an ancestor but not a proper ancestor of itself.)

We conclude this section by bounding the expected rank of the root of a tree as a function of the tree size. This allows us to bound the time of FIND(x) by

a function of the size of the current set containing x rather than by a function of n.

THEOREM 2.2. *Consider a tree of k nodes built by a sequence of* UNITE *operations using randomized linking or randomized early linking. The expected rank of the root is* $\mathrm{O}(\log k)$.

Proof. A crude argument suffices. The root is maximum among the k nodes of the tree. Property (ii) implies that for any $j > \lg k$, the probability that some node in the tree has rank greater than j is at most $k/2^j$. It follows that the expected rank of the root is at most $\sum_{j=\lceil \lg k \rceil}^{\infty} (j+1)k/2^j = \mathrm{O}(\log k)$.

3 Analysis of Compression

Now we are ready to extend an existing analysis of linking by rank to apply to randomized linking and randomized early linking. To obtain the tightest, most general results, we extend the analysis of Tarjan and van Leeuwen [17]: as far as we can tell, Kozen's analysis [10] and the later ones based on it [3, 8] are not flexible enough to handle splicing, and the top-down approach of Seidel and Sharir [13] does not give a local bound for any type of compaction.

Consider a sequence of m intermixed UNITE and FIND operations on sets containing a total of n elements, such that every element is an input to at least one operation. (Other elements can be deleted without affecting any of the operations.) For ease in stating time bounds we assume $n \geq 2$, which implies $m \geq n/2 \geq 1$. Let $d = m/n > 0$. Assume the operations are done using randomized linking or randomized early linking with some form of compaction.

For purposes of the analysis only, we assume as in Tarjan's original paper [14] that all links are done before all compactions. More precisely, all the UNITE operations are done first, in their original order but with no compaction, and then the compactions are done on the resulting trees. This does not affect the parent changes done by the compactions, and it slightly simplifies the analysis. We count the parent changes done by the compactions: if there are h nodes on a FIND path, a compression or split changes at least $h-2$ parents, a halving changes at least $\lfloor h/2 \rfloor - 1$, and a splice of two FIND paths having a total of h nodes changes at least $h-3$ parents. Thus the total time of the set operations is $\mathrm{O}(m)$ plus $\mathrm{O}(1)$ per parent change.

In our analysis we use the potential method of amortized analysis [14]. We assign to each state of the data structure a real-valued potential. We define the amortized cost of an operation to be its actual cost plus the change in potential it causes. Then the total cost of a sequence of operations is the sum of their amortized costs plus the initial potential minus the final potential. In all our uses of this technique, the potential is always non-negative, so the total cost of the operations is at most the sum of their amortized costs plus the initial potential. Also, the change in potential caused by a compaction is non-positive. Furthermore the potential of the data structure is the sum of the potentials of its nodes.

We define the node potentials using Ackermann's function and several auxiliary functions. We use a classical definition of this function: even though Seidel and Sharir [13] showed that one can use a more rapidly growing function, this changes only the additive constant in the inverse function.

Ackermann's function [1, 12] is a function of two non-negative integer variables defined recursively as follows:

$$A(0,j) = j+1$$
$$A(k,0) = A(k-1,1) \text{ if } k > 0$$
$$A(k,j) = A(k-1, A(k,j-1)) \text{ if } k > 0 \text{ and } j > 0$$

It is straightforward to prove by induction that A is strictly increasing in both arguments, $A(k+1,j) \geq A(k,j+1)$, and $A(1,j) = j+2$. (As k increases beyond 1, the function $A(k,j)$ rapidly increases: $A(2,j) > 2j$), $A(3,j) > 2^j, \ldots$)

The *inverse Ackermann function* α is defined for any non-negative integer r and non-negative real number d by

$$\alpha(r,d) = \min\{k > 0 | A(k, \lfloor d \rfloor) > r\}$$

This function is non-decreasing in the first argument and non-increasing in the second.

The *index function* $b(k,r)$ is defined for any non-negative integers k and r by

$$b(k,r) = \min\{j \geq 0 | A(k,j) > r\}$$

This function is non-increasing in the first argument and non-decreasing in the second. Note the similarity between α and b. We have defined α to be positive and extended its domain from integers to real values of d merely to simplify the statement of time bounds.

The *level function* $a(r,s)$ is defined for any non-negative integers $r \leq s$ by

$$a(r,s) = \min(\{\alpha(r,d)+1\} \cup \{k \leq \alpha(r,d) | A(k, b(k,r)) > s\})$$

LEMMA 3.1. *If $r \leq s$, $a(r,s) = 0$ if and only if $r = s$.*

Proof. Since $A(0,j) = j+1$, $b(0,r) = r$, which implies $A(0, b(0,r)) = A(0,r) = r+1$. Hence $a(r,s) = 0$ if $r = s$ and $a(r,s) > 0$ if $r < s$.

For each node x we define a *level* $x.a$, an *index* $x.b$, and a *count* $x.c$ as follows:

$$x.a = a(x.r, x.p.r)$$
$$x.b = b(x.a - 1, x.p.r) \text{ if } x.a > 0, x.b = 0 \text{ otherwise}$$
$$x.c = x.a \times (x.r + 2) + x.b$$

LEMMA 3.2. *If* $x.a \leq \alpha(x.r, d)$, *then* $x.b \leq \max\{x.r, 1\} \leq x.r + 1$.

Proof. If $x.a = 0$, $x.b = 0$, so the lemma holds. Suppose $x.a = k > 0$. Let $j = b(x.a, r)$. The definition of $x.a$ implies $A(k, j) > s$. If $j = 0$, then $A(k, 0) = A(k-1, 1) > s$. Hence $x.b = \min\{j \geq 0 | A(k-1, j) > s\} \leq 1$, so the lemma holds. If $j > 0$, then $A(k, j) = A(k-1, A(k, j-1)) > s$. Since $j = b(k, r), A(k, j-1) \leq r$, which implies $A(k-1, r) > s$. Hence $x.b \leq r$, so the lemma holds.

LEMMA 3.3. *If* $x.a = \alpha(x.r, d) + 1 = \alpha(x.p.r, d) + 1$, *then* $x.b \leq d$.

Proof. The definition of α implies $A(x.a - 1, d) = A(\alpha(x.p.r, d), d) > x.p.r$. Since $x.b = b(x.a - 1, x.p.r), x.b \leq d$.

LEMMA 3.4. *For every node* x, $x.a$ *and* $x.c$ *never decrease, and* $x.c$ *increases whenever* $x.a$ *or* $x.b$ *changes. If* $x.a$ *increases by* k, $x.c$ *increases by at least* k.

Proof. For fixed r, $a(r, s)$ is a non-decreasing function of s. Since $x.r$ never changes and $x.p.r$ never decreases, $x.a = a(x.r, x.p.r)$ never decreases. Since $x.p.r$ never decreases, while $x.a$ is constant $x.b$ never decreases, so a change in $x.b$ while $x.a$ is constant increases both $x.b$ and $x.c$. When $x.a$ increases by k, $x.b$ decreases by at most $x.r + 1$ by Lemma 3.2, so $x.c$ increases by at least $k(x.r + 2) - (x.r + 1) \geq k$.

The next lemma is the key to counting parent changes. It holds for all types of compaction.

LEMMA 3.5. *Let* x *and* y *be nodes such that* $x.p.r \leq y.r$ *and* $0 < x.a = y.a$ *just before a* FIND *that increases* $x.p$ *to at least* $y.p$. *Then the* FIND *increases* $x.c$.

Proof. Let $k = x.a = y.a$, $j = x.b$, and $j' = y.b$ just before the FIND. The definition of $x.a$ implies $A(k-1, j) = A(k-1, b(k-1, x.p.r)) \leq x.p.r$. The definition of b implies $y.r < A(k-1, b(k-1, y.r)) \leq A(k-1, b(k-1, y.p.r)) = A(k-1, j')$. Since $x.p.r \leq y.r$, $j < j'$. Since $j' = \min\{i \geq 0 | A(k-1, i) > y.p.r\}$, the FIND increases $b(k-1, x.p.r)$ to at least j', by increasing $x.p.r$ to at least $y.p.r$. It follows that either the FIND does not change $x.a$ but increases $x.b$, or it increases $x.a$. In either case $x.c$ increases by Lemma 3.4.

Now we are ready to count parent changes. Let the potential of a node x be the number of proper ancestors of x of the same rank as x plus $\max\{0, (\alpha(x.r, d) + 1) \times (x.r + 2) + d + 1 - x.c\}$. Let the potential of a collection of trees be the sum of the potentials of their nodes, and let the amortized cost of a FIND be the number of parent changes it makes plus the change in potential it causes.

LEMMA 3.6. *The expected initial potential is* $O(m)$.

Proof. By Theorem 2.1 the expected sum over all nodes x of the number of proper ancestors of x of the same rank as x is at most $2n$. Since $\alpha(r, d) \leq r$, the sum over all nodes of the rest of the potential is at most $n(d+1) + \sum_{r \geq 0} n(r+1)(r+2)/2^r = O(m)$.

LEMMA 3.7. *Suppose* FIND *operations are done with compression. Then the expected amortized cost of* FIND(x) *is* $O(\alpha(\log k, d))$, *where* k *is the number of elements in the current set containing* x, *or in the smaller of the two sets combined if the* FIND *is in a* UNITE *done using randomized eager linking that returns* **true**.

Proof. Consider any FIND path. Compression of the FIND path does not increase the potential of any node. Let v be the last node on the path, and let x be any node on the path whose parent is changed by the compression. If $x.a = 0$, compressing the path causes x to lose at least one proper ancestor of the same rank, thereby decreasing its potential. If $x.a > 0$, $\alpha(x.r, d) = \alpha(x.p.r, d)$, and there is a node y after x on the path such that $y.a = x.a$, compressing the path reduces the potential of x by at least one by Lemmas 3.2, 3.3, and 3.5. Thus the compression decreases the potential of x unless $\alpha(x.r, d) < \alpha(x.p.r, d)$ or x is last on its level. Since $\alpha(x.r, d) \leq \alpha(x.p.r, d)$ for every x, at most $\alpha(v.r, d)$ nodes x have $\alpha(x.r, d) < \alpha(x.p.r, d)$. Since every node on the path has level at most $\alpha(v.r, d) + 1$, at most $\alpha(v.r, d) + 2$ nodes are last on their level. The amortized cost of the FIND is thus at most $2\alpha(v.r, d) + 2$.

By Theorem 2.2, the expected value $\mathbf{E}[v.r]$ of $v.r$ is $O(\log k)$, where k is the number of elements in the set containing x, or in the smaller of the two sets combined if the FIND is in a UNITE done using randomized eager linking that returns **true**. If we extend $\alpha(r, d)$ for fixed d to a function over non-negative real numbers r by connecting successively larger values by straight lines, we obtain a piecewise linear concave function. By Jensen's inequality [7], the expected value $\mathbf{E}[\alpha(v.r, d)]$ of $\alpha(v.r, d)$ is at most $\alpha(\mathbf{E}[v.r], d) = O(\alpha(\log k, d))$.

THEOREM 3.1. *If* FIND *operations are done with compression, the expected total time of the operations is at*

most the sum over every FIND *of* $\mathrm{O}(\alpha(\log k, d))$*, where* k *is the number of elements in the current set containing the element found, or in the smaller of the two sets combined if the* FIND *is in a* UNITE *done using randomized eager linking that returns* **true**.

Proof. The theorem follows immediately from Lemmas 3.6 and 3.7.

4 Analysis of Splitting and Halving

The analysis of splitting is like that of compression but more elaborate. We need a lemma that complements Lemma 3.5.

LEMMA 4.1. *Let* x *and* y *be nodes such that* $x < y$ *and* $x.a < \min\{y.a, \alpha(x.r, d) + 1\}$ *just before a* FIND *that increases* $x.p$ *to at least* $y.p$*. Then the* FIND *increases* $x.a$ *to at least* $\min\{y.a, \alpha(x.r, d) + 1\}$*, and hence increases* $x.c$ *by at least* $\min\{y.a, \alpha(x.r, d) + 1\} - x.a$.

Proof. Let $k = y.a$. The definition of $y.a$ implies $A(k-1, b(k-1, x.r)) \leq A(k-1, b(k-1, y.r)) \leq y.p.r$. It follows that once $x.p$ is at least $y.p$, $x.a \geq \min\{y.a, \alpha(x.r, d) + 1\}$. By Lemma 3.4, $x.c$ increases by at least as much as $x.a$.

Let the potential of a node x be its number of proper ancestors of the same rank plus $2(\max\{0, (\alpha(x.r, d) + 1) \times (x.r + 2) + d + 1 - x.c\})$, let the potential of a collection of trees be the sum of their node potentials, and let the amortized cost of a FIND be the number of parent changes plus the change in potential. The expected total initial potential is $\mathrm{O}(m)$ by Lemma 3.6. A split cannot increase the potential of any node.

LEMMA 4.2. *Suppose* FIND *operations are done with splitting. Then the expected amortized cost of* FIND(x) *is* $\mathrm{O}(\alpha(\log k, d))$*, where* k *is the number of elements in the current set containing* x*, or in the smaller of the two sets combined if the* FIND *is in a* UNITE *done using randomized eager linking that returns* **true**.

Proof. Consider any FIND path. Let x be any node on the path other than the last two. Then the split changes the parent of x. If $x.a < \min\{x.p.a, \alpha(x.r, d) + 1\}$, splitting the path decreases the potential of x by at least $2(\min\{x.p.a, \alpha(x.r, d) + 1\} - x.a) \geq 1 + \min\{x.p.a, \alpha(x.r, d) + 1\} - x.a$ by Lemmas 3.2, 3.3, and 4.1. If $x.a = 0$, splitting the path decreases the potential by at least 1, since x loses a proper ancestor of the same rank. If $0 < x.a = x.p.a$ and $\alpha(x.r, d) = \alpha(x.p.r, d)$, splitting the path decreases the potential by at least $2 \geq 1$ by Lemmas 3.2, 3.3, and 3.5.

We claim that in all cases, including those in which the potential of x does not decrease, the potential of x decreases by at least

$$1 + x.p.a - x.a + 2(\alpha(x.r, d) - \alpha(x.p.r, d)) \tag{4.1}$$

Since $\alpha(x.r, d) \leq \alpha(x.p.r, d)$, this is true if $x.p.a < x.a$, because the value of (4.1) is non-positive. It is also true if $x.p.a = x.a$, since if $\alpha(x.r, d) = \alpha(x.p.r, d)$, the value of (4.1) is 1, and if $\alpha(x.r, d) < \alpha(x.p.r, d)$, the value of (4.1) is non-positive. If $x.a < x.p.a \leq \alpha(x.r, d) + 1$, the drop is at least $1 + x.p.a - x.a$, which is at least the value of (4.1). If $x.a < \alpha(x.r, d) + 1 < x.p.a$, the drop is at least $1 + \alpha(x.r, d) + 1 - x.a \geq 1 + x.p.a - x.a + \alpha(x.r, d) + 1 - x.p.a$. Since $x.p.a \leq \alpha(x.p.r, d) + 1$, the drop is at least $1 + x.p.a - x.a + \alpha(x.r, d) - \alpha(x.p.r, d)$, which is at least the value of (4.1). The last and most interesting case, which accounts for the factor of 2 in (4.1), is $x.a = \alpha(x.r, d) + 1 < x.p.a$. Since $x.p.a \leq \alpha(x.p.r, d) + 1$, $\alpha(x.p.r, d) - \alpha(x.r, d) \geq \max\{1, x.p.a - x.a\}$, so the value of (4.1) is non-positive.

Now we sum (4.1) over all nodes on the FIND path except the last two. Suppose there are h such nodes, with u the first and v the parent of the last. (Node v is next-to-last on the path.) The sum telescopes to

$$\begin{aligned} &h + v.a - u.a + 2(\alpha(u.r, d) - \alpha(v.r, d)) \\ &\quad \geq h + (v.a - \alpha(v.r, d) - 1) - (u.a - \alpha(u.r, d) - 1) + \\ &\qquad \alpha(u.r, d) - \alpha(v.r, d) \\ &\quad \geq h - 2\alpha(v.r, d) - 1 \end{aligned}$$

It follows that the amortized cost of the FIND is at most $2\alpha(v.r, d) + 1$. The rest of the proof is the same as the last paragraph of the proof of Lemma 3.7.

THEOREM 4.1. *If* FIND *operations are done with splitting, the expected total time of the operations is at most the sum over every* FIND *of* $\mathrm{O}(\alpha(\log k, d))$*, where* k *is the number of elements in the current set containing the element found, or in the smaller of the two sets combined if the* FIND *is in a* UNITE *done using randomized eager linking that returns* **true**.

Proof. The theorem follows immediately from Lemmas 3.6 and 4.2.

THEOREM 4.2. *If* FIND *operations are done with halving, the expected total time of the operations is at most the sum over every find of* $\mathrm{O}(\alpha(\log k, d))$*, where* k *is the number of elements in the current set containing the element found, or in the smaller of the two sets combined if the* FIND *is in a* UNITE *done using randomized eager linking that returns* **true**.

Proof. Instead of bounding the number of parent changes, we bound the number of grandparent changes: after halving a path, each node with a new parent except the last one also has a new grandparent. We redefine the level, index, and count of a node x to be $x.a' = a(x.r, x.p.p.r)$; $x.b' = b(x.a' - 1, x.p.p.r)$ if $x.a' > 0$, $x.b' = 0$ otherwise; and $x.c' = x.a' \times (x.r + 2) + x.b'$. We let the potential be the one defined in Section 3, with the new definitions of level, index, and count. We define the amortized cost of a FIND to be the number of grandparent changes plus the change in potential. Then Lemma 4.2 holds for halving by the same proof. The theorem follows immediately.

5 Analysis of Splicing

Splicing is unlike the other compaction methods in that the new parent of a node need not be one of its ancestors in the old tree. But the new grandparent is. Indeed, it is an old ancestor of the old grandparent. This implies that even though a node can acquire new ancestors, it loses at least as many as it gains. Our analysis of splicing uses this fact, along with all the ideas in Sections 3 and 4 and one additional property of levels.

In our analysis of splicing we use the following terminology. We denote by u and v the first nodes on the two paths to be spliced and by w the last node on both paths: the paths are from u to w and from v to w, and w is the nearest common ancestor of u and v in the tree before the splice. (Recall that we assume the UNITE operations are done first, without any compaction, and then compactions are done. Thus u and v are in the same tree when the splice occurs.) The *splice sequence* is the sequence formed by merging the two paths in increasing node order: the first node on the sequence is u or v and the last is w. We color the nodes on the path from u to w white and those on the path from v to w black; w gets both colors.

LEMMA 5.1. *Suppose* UNITE *operations are done using early linking by index with splicing. If x is a node on a spliced pair of paths, then its new grandparent was an ancestor of its old grandparent in the tree before the splice.*

Proof. Each node that changes parent gets a new parent of the opposite color. Hence any node that changes grandparent gets a grandparent of the same color. If node x changes grandparent, it also changes parent. Since its new parent is greater than its old parent, its new grandparent cannot be its old parent. Thus its new grandparent is an ancestor of its old grandparent in the tree before the splice.

COROLLARY 5.1. *If x is a node on a spliced pair of paths, the number of proper ancestors of x of the same rank as x does not increase as the result of a splice. If x and its grandparent have the same rank and x changes grandparent, its number of proper ancestors of the same rank strictly decreases.*

Proof. If the corollary holds for the old grandparent of x, then it holds for x by Lemma 5.1. The corollary follows by induction on the depth of x in the tree before the splice.

Even though a node that changes parent may not change grandparent, the set of nodes that change grandparent is dense in the splice sequence, as the following lemma shows.

LEMMA 5.2. *Among any six consecutive nodes in the splice sequence not including w, at least one has its grandparent among the six and changes grandparent as a result of the splice.*

Proof. If a node changes parent, its new parent is the first node of the opposite color following its old parent in the splice sequence. We prove the lemma by case analysis. If among the six nodes there are three nodes of the same color with the last two consecutive, then the first of these nodes satisfies the lemma, since its new parent follows its old grandparent in the splice sequence. This case applies if the last two nodes are the same color, say white, since then either there are three white nodes or the second, third, and fourth nodes are black. It also applies if the fourth and fifth nodes are the same color, say white: either there are at least three white nodes or the first, second, and third nodes are black. The only remaining possibility is that the last three nodes alternate in color, say white, black, white. In this case if the third node is white, it satisfies the lemma; and if it is black, the second node satisfies the lemma whether it is black or white.

The next lemma gives the additional property of levels that we need to analyze splicing.

LEMMA 5.3. *If $r \leq s \leq t$, $\max\{a(r,s), a(s,t)\} \geq a(r,t)$.*

Proof. Let $k = \max\{a(r,s), a(s,t)\}$. Since $a(r,t) \leq \alpha(r,d) + 1$, if $k \geq \alpha(r,d) + 1$ the lemma holds. Thus suppose $k \leq \alpha(r,d)$. By the definition of $a(r,s), A(k, b(k,r)) > s$. By the definition of $b(k,s)$, $b(k,s) \leq b(k,r)$. Since $\alpha(s,d) \geq \alpha(r,d)$, the definition of $a(s,t)$ gives $A(k, b(k,s)) > t$, which implies $A(k, b(k,r)) > t$. By the definition of $a(r,t)$, $a(r,t) \leq k$.

Our analysis of randomized early linking with splicing applies to early linking by rank with splicing as well. Indeed, the analysis becomes simpler because we do not need to consider nodes of level 0. Tarjan and van Leeuwen [17] claimed an inverse-Ackermann-function amortized bound for early linking by rank with splicing, but they left the proof as an exercise. We provide full details.

Let the potential of a node x be five times the number of proper ancestors of x of the same rank as x plus $10\left(\max\left\{0, \left(\alpha\left(x.r, d\right)+1\right)\times\left(x.r+2\right)+d+1-x.c\right\}\right)+$ $10\left(\max\left\{0, \left(\alpha\left(x.r, d\right)+1\right)\times\left(x.r+2\right)+d+1-x.c'\right\}\right)$. Here $x.c$ is as defined in Section 4 and $x.c'$ is defined as in Section 4. We call the second and third terms in the potential of x the *parent potential* and the *grandparent potential* of x, respectively. We use the parent potential to measure the effect of parent changes and grandparent potential to measure the effect of grandparent changes in cases where we cannot use the parent potential. Let the potential of a collection of trees be the sum of their node potentials, and let the amortized cost of a splice be the number of parent changes plus the change in potential. The expected total initial potential is $\mathrm{O}\left(m\right)$ by Lemma 3.6. By Lemma 5.1 and Corollary 5.1 a splice cannot increase the potential of any node.

LEMMA 5.4. *Suppose* UNITE *operations are done with randomized early linking and splicing. Then the expected amortized cost of a* UNITE *is* $\mathrm{O}\left(\alpha\left(\log k, d\right)\right)$*, where* k *is the number of elements in the set containing the found elements if the* UNITE *returns* **false***, or in the smaller of the two sets combined if the* UNITE *returns* **true***.*

Proof. Consider a splice of two paths, from u and from v to their nearest common ancestor w. Let its splice sequence be $x_1, \ldots, x_h = w$. Let the *pseudo-level* $x_i.a''$ of $x_i \neq w$ be $x_i.a'' = a\left(x_i.r, x_{i+1}.r\right)$. Since $x_{i+1} \leq x_i.p$, $x_i.a'' \leq x_i.a \leq x_i.a'$.

Our plan is to use the idea in the proof of Lemma 4.2: bound the potential drops of the nodes along a FIND path by amounts whose sum telescopes, sum the bounds, and use the sum to bound the amortized cost of the FIND. A splice sequence consists of two paths, not just one, however, and not all nodes on the spliced paths drop in potential by the needed amount. To analyze splicing, we identify a sufficiently dense subset of nodes whose potential does drop by the needed amount. We use the pseudo-levels to help identify these nodes and to produce a sum that telescopes.

We mark a subset of the nodes in the splice sequence, by first marking x_{h-1} and then proceeding backward along the sequence. Let x_i be the most recently marked node. Suppose x_i is white; proceed symmetrically if it is black. The next node to be marked depends on whether the pseudo-level of x_i is positive or 0. Suppose it is positive. If at most two nodes precede x_i, stop marking nodes. Otherwise, if there is a white node among the three nodes preceding x_i in the sequence, let y be the last such node (the child of x_i), and let x be the node of maximum pseudo-level among y and any black nodes between it and x_i. If the three nodes preceding x_i are all black, let y be the third node preceding x_i, and let x be the node of maximum pseudo-level in $\{y, y.p\}$. (Node $y.p$ is the node immediately after y in the splice sequence.) Then $y \leq x \leq y.p \leq x_i$. Mark x.

We bound the potential drop of y by an amount that depends on x, y and x_i. Suppose y is white. Then $y.p = x_i$. The splice increases $y.p$ to at least x_{i+1}. If $y.a = x_i.a''$ and $\alpha\left(y.r, d\right) = \alpha\left(x_i.r, d\right)$, the splice decreases the parent potential of y by at least $10 \geq 5$ by Lemmas 3.2, 3.3, and 3.5. If $y.a < \min\left\{x_i.a'', \alpha\left(y.r, d\right)+1\right\}$, the splice decreases the parent potential of y by at least $10\left(\min\left\{x_i.a'', \alpha\left(y.r, d\right)+1\right\}-y.a\right) \geq 5+5\left(\min\left\{x_i.a'', \alpha\left(y.r, d\right)+1\right\}-y.a\right)$ by Lemmas 3.2, 3.3, and 4.1. An argument like the one in the proof of Lemma 4.2 shows that the potential drop in all cases is at least $5+5\left(x_i.a''-y.a\right)+10\left(\alpha\left(y.r, d\right)-\alpha\left(x_i.r, d\right)\right)$. By Lemma 5.3, $y.a \leq x.a''$, so the drop in the parent potential of x is at least $5+5\left(x_i.a''-x.a''\right)+10\left(\alpha\left(y.r, d\right)-\alpha\left(x_i.r, d\right)\right) = 5+5\left(x_i.a''-x.a''\right)+10\left(\alpha\left(x.r, d\right)-\alpha\left(x_i.r, d\right)\right)+10\left(\alpha\left(y.r, d\right)-\alpha\left(x.r, d\right)\right)$.

The situation is similar if y is black, but the drop is in the grandparent potential of y. In this case the splice increases $y.p.p$ from less than x_i to at least x_{i+1}. If $y.a' = x_i.a''$ and $\alpha\left(y.r, d\right) = \alpha\left(x_i.r, d\right)$, the splice decreases the grandparent potential of y by at least $10 \geq 5$ by Lemmas 3.2, 3.3, and 3.5, since then $\alpha\left(y.r, d\right) = \alpha\left(y.p.p.r, d\right)$. If $y.a' < \min\left\{x_i.a'', \alpha\left(y.r, d\right)+1\right\}$, the splice decreases the grandparent potential of y by at least $10\left(\min\left\{x_i.a'', \alpha\left(y.r, d\right)+1\right\}-y.a'\right) \geq 5+5\left(\min\left\{x_i.a'', \alpha\left(y.r, d\right)+1\right\}-y.a'\right)$ by Lemmas 3.2, 3.3, and 4.1. By Lemma 5.3, $y.a' \leq x.a''$. By the argument in the previous paragraph, in all cases the drop in the grandparent potential of y is at least $5+5\left(x_i.a''-x.a''\right)+10\left(\alpha\left(x.r, d\right)-\alpha\left(x_i.r, d\right)\right)+10\left(\alpha\left(y.r, d\right)-\alpha\left(x.r, d\right)\right)$.

If the pseudo-level of x_i is 0, we choose the next node to mark in a different way. If x_i is preceded in the splice sequence by at most four nodes, we stop marking nodes. Otherwise, choose and mark a node x as follows. If any of the five nodes immediately preceding x_i in the splice sequence has positive pseudo-level, choose and mark any such node as x. If not, all these nodes as well as x_i have the same rank. Among these six nodes, choose and mark as x one whose grandparent is among the six and whose grandparent changes as a result of the splice. Such a node exists by Lemma 5.2;

by Corollary 5.1, the splice decreases the potential of x by at least 5.

The choice of x guarantees that, whether or not x has pseudo-level 0, its potential drops by at least $5 + 5(x_i.a'' - x.a'') + 10(\alpha(y.r, d) - \alpha(x_i.r, d))$, since this value is non-positive if $x.a'' > 0$.

At the end of the marking process, at least $h/5 - 1$ nodes are marked: the last marked node is followed by one unmarked node, the first marked node is preceded by at most four unmarked nodes, and there are at most four unmarked nodes between each pair of consecutive marked nodes. With each marked node except x_{h-1} we have identified a potential drop of at least 5, minus terms whose sum we shall show is $O(\alpha(w.r, d))$. It follows that the amortized cost of the splice is $O(\alpha(w.r, d))$. The lemma follows by the argument in the last paragraph of the proof of Lemma 3.7.

Before summing our bounds on the potential drops, we need to make sure that we are not double-counting. A node y of level 0 that contributes a potential drop changes from unmarked to marked, so it cannot contribute a second time. A node x that contributes a drop in parent potential changes parent from a node no greater than the most recently marked node to one greater than the newly marked node, so it cannot contribute a second time. A node that contributes a drop in grandparent potential changes grandparent from a node no greater than the most recently marked node to one greater than the newly marked node, so it cannot contribute another drop in grandparent potential, nor a drop in potential as a node of pseudo-level 0. It might, however, later contribute a drop in parent potential. But this is not a problem, since the parent and grandparent potentials of a node are counted separately.

Let the marked nodes in increasing order be $z_1, z_2, \ldots, z_{j+1}$; for $i \leq j$ let y_i be the node y chosen when z_i was marked. (If z_i has pseudo-level 0, $y_i = z_i$.) The potential drop caused by the splice is at least the sum over all $i \leq j$ of

$$\begin{aligned} &5 + 5(z_{i+1}.a'' - z_i.a'') \\ &+ 10(\alpha(z_i.r, d) - \alpha(z_{i+1}.r, d)) \\ &+ 10(\alpha(y_i.r, d) - \alpha(z_i.r, d)) \end{aligned} \tag{5.2}$$

If we sum (5.2) over all $i \leq j$, the first term sums to at least $h - 10$ and the second and third terms telescope to $5(z_{j+1}.a'' - z_1.a'') + 10(\alpha(z_1.r, d) - \alpha(z_{j+1}.r, d))$. An argument like that in the proof of Lemma 4.2 shows that $5(z_{j+1}.a'' - z_1.a'') + 10(\alpha(z_1.r, d) - \alpha(z_{j+1}.r, d)) \geq -10\alpha(w.r, d) - 5$.

It remains to estimate the sum of the last term. Each term in the sum is non-positive. Each z_i is distinct. Each y_i is either equal to z_i or precedes z_i in the splice sequence by one or two positions. Furthermore, if y_i precedes z_i by two positions, then $y_{i-1} = z_{i-1}$, either because the pseudo-level of z_i is 0 or because the two nodes following y_i in the splice sequence are the same color, which results in $y_{i-1} = z_{i-1}$ being the node immediately after y_i in the sequence. Thus if we drop each index i such that $y_i = z_i$ and re-index, $z_{i-1} \leq y_i$ for each $i > 1$. This implies $\alpha(z_{i-1}.r, d) \leq \alpha(y_i.r, d)$, which further implies that the sum of the third term over all indices is at least $10\alpha(w.r, d)$. We conclude that the amortized cost of the splice is $O(\alpha(w.r, d))$.

THEOREM 5.1. *If* UNITE *operations are done using randomized early linking with splicing, the expected total time of the* UNITE *operations is at most the sum over every* UNITE *of* $O(\alpha(\log k, d))$*, where k is the number of elements in the set containing the two found elements if the* UNITE *returns* **false***, or in the smaller of the two sets combined if the* UNITE *returns* **true***.*

Proof. The theorem follows immediately from Lemma 5.4.

6 Remarks

We have shown that randomized linking with compression, splitting or halving has an inverse-Ackermann-function bound on the expected amortized time per FIND, and that this is also true of randomized eager linking with compression, splitting, halving, or splicing. All the analyses use essentially the same potential function, so they are all compatible, which means that one can use either form of linking for any UNITE and any appropriate form of compaction for any FIND and the inverse-Ackermann function bound will still hold. The lower bound of Fredman and Saks [5] implies that all our bounds are tight to within a constant factor.

If all that one wants is a global bound of $O(\alpha(\lg n, d))$ on the expected amortized time per FIND, one can simplify the analysis. In particular, one can use the estimate $v.r \leq \lg n$ in the proof of Lemmas 3.7 and 4.2, and the estimate $w.r \leq \lg n$ in the proof of Lemma 5.4, thereby avoiding the use of Theorem 2.2. Furthermore, one can replace the definition of the level function by the simpler $a(r, s) = \max\{k \geq 0 | A(k, b(k, r)) > s\}$. With this definition of a, $x.a \leq \alpha(\lg n, d) + d + 1$. Furthermore if $k = \alpha(\lg n, d) + i$ for some $i > 0$, $x.b \leq d/2^{i-1}$. It follows that the number of times the index of a vertex of rank r can change without its level changing is $O(r\alpha(\lg n, d) + d + 1)$, and the number of times its level can change is at most $\alpha(\lg n, d) + d + 1$. This is enough to obtain the desired bound, and it considerably simplifies the analysis of splitting, halving, and splicing by eliminating the terms involving α in the estimated changes in node potentials.

One can obtain the results of Sections 3 and 4 with a simpler definition of levels and indices based on Kozen's work [10], namely $a(r,s) = \min\{k \geq 0 | A(k,r) > s\}$, $x.a = a(x.r, x.p.r)$, and $x.b = b(x.a, x.p.r)$. Lemma 5.4 does not hold for this definition of a, however, only the weaker inequality $\max\{a(r,s), a(s,t)\} + 1 \geq a(r,t)$ if $r \leq s \leq t$. We have not been able to prove Theorem 5.1 using these definitions; whether this is possible is an open problem.

In our view, the importance of our work is to show that in applications in which the instances are such that linking by index is effectively random linking, one does not need to use linking by rank or size to obtain inverse-Ackermann-function behavior. We do not advocate implementing randomized linking unless one can obtain it for free, such as when the elements are stored in a hash table, since linking by rank is simple to implement and has the same bounds. On the other hand, it is an interesting theoretical question to ask how much randomness is needed to obtain our results. We can show that a poly-logarithmic number of random bits suffice.

Randomized linking is equivalent to randomized linking by size: when linking the roots u and v of two trees containing $u.s$ and $v.s$ nodes, respectively, make u the parent of v with probability $u.s/(u.s+v.s)$. An even simpler form of randomized linking is coin-flip linking: when linking the roots u and v of two trees, make u the parent of v with probability $1/2$. We conjecture that coin-flip linking, unlike randomized linking, is *not* asymptotically optimal, and indeed is asymptotically no more efficient than naïve linking.

Support for this conjecture comes from considering the following bad example for naïve linking with path compression [17]. Let n be a power of 2. By doing $n/2 - 1$ UNITE operations, each of which combines two isomorphic trees, build a binomial tree of $n/2$ nodes. Now repeat the following two operations $n/2$ times: link the existing tree with a tree containing a single node, which becomes the new root, and then do a FIND on the deepest node in the tree. Each find takes $\Theta(\log n)$ time, for a total time of $\Theta(n \log n)$. A similar example for coin-flip linking consists of building a binomial tree B of $n/2$ nodes and then repeating the following three steps $n/4$ times: UNITE the existing tree with a new one-node tree; do this again; do a FIND on a node chosen at random from among the nodes in the original tree B. Each pair of UNITE operations on the average adds a new root to the current tree. As long as the depth of the current tree remains bounded by a polynomial of fixed degree in $\log n$, the expected length of each FIND path is $\Omega(\log n / \log\log n)$; if this remains true throughout the sequence, the expected total time is $\Omega(n \log n / \log\log n)$. We conjecture but so far have been unable to prove that with high probability the tree depth remains polylogarithmically bounded. Figure 1 shows the result of doing this sequence of operations for a range of values of n. The data strongly support the conjecture that the amortized time per find is much closer to $\log n$ than to $\alpha(n,0)$.

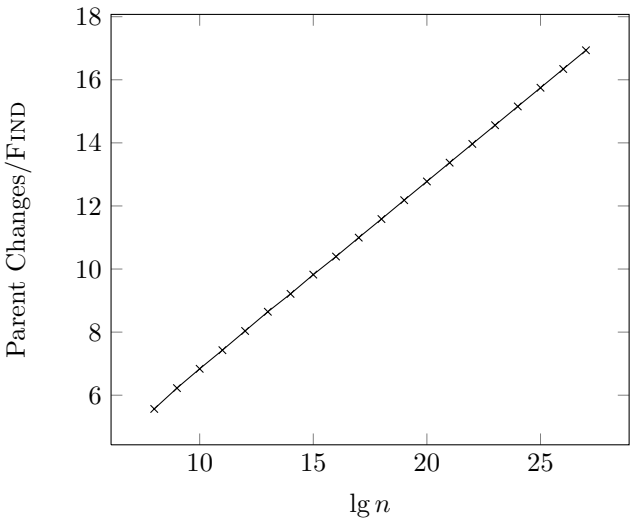

Figure 1: Average parent changes per find as a function of $\lg n$.

Another direction of work is to see if the idea of randomized linking can be extended to the problem of evaluating a function defined on paths in trees subject to links [15]. In this application, there is no choice in how to do the links. Nevertheless, we think randomness can help in this problem.

References

[1] W. Ackermann. Zum hilbertschen aufbau der reellen zahlen. *Mathematische Annalen*, 99(1):118133, 1928.

[2] Md. M. Ali Patwary, J. R. S. Blair, and F. Manne. Experiments on union-find algorithms for the disjoint-set data structure. In *Proc. 9th Annual International Symposium on Experimental Algorithms*, volume 6049 of LNCS, pp. 411-423. Springer, 2010.

[3] S. Alstrup, I. L. Gørtz, T. Rauhe, M. Thorup, and U. Zwick. Union-find with constant time deletions. In *Proc. 32nd Annual International Colloquium on Automata, Languages and Programming*, volume 3580 of LNCS, pp. 78-89. Springer-Verlag, 2005.

[4] E. W. Dijkstra. A Discipline of Programming. Prentice-Hall, pp. 161-167, 1976.

[5] M. L. Fredman and M. E. Saks. The cell probe complexity of dynamic data structures. In *Proc. 21st Annual ACM Symposium on Theory of Computing*, pp. 345-354, 1989.

[6] B. A. Galler and M. J. Fisher. *An improved equivalence algorithm*, *Commun. ACM*, 7(5):301-303, 1964.
[7] J. L. W. V. Jensen. Sur les fonctions convexes et les in égalit és entre les valeurs moyennes. *Acta Mathematica*, 30(1):175193, 1906.
[8] H. Kaplan, N. Shafrir, and R. E. Tarjan. Union-find with deletions. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 19-28, 2002.
[9] D. E. Knuth. The Art of Computer Programming, Volume 1: Fundamental Algorithms, Third Edition. Addison-Wesley, p. 576, 1997.
[10] D. C. Kozen. The design and analysis of algorithms. Springer-Verlag, 1992.
[11] J. B. Kruskal. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. *Proc. of the American Mathematical Society*, 7(1):48-50, 1956.
[12] R. Péter. Rekursive funktionen. Académiai Kiadó, 1951.
[13] R. Seidel and M. Sharir. Top-down analysis of path compression. *SIAM J. Computing*, 34(3):515-525, 2005.
[14] R. E. Tarjan. Amortized computational complexity. *SIAM J. on Algebraic Discrete Methods*, 6(2):306-318, 1985.
[15] R. E. Tarjan. Applications of path compression on balanced trees. *J. ACM*, 26(4):690-715, 1979.
[16] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215-225, 1975.
[17] R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *J. ACM*, 31(2):245-281, 1984.
[18] T. van der Weide. Datastructures: An Axiomatic Approach and the Use of Binomial Trees in Developing and Analyzing Algorithms. Mathematisch Centrum, 1980.
[19] J. van Leeuwen and T. van der Weide. Alternative path compression techniques. Technical Report RUU-CS-77-3, Rijksuniversiteit Utrecht, 1977.

7 Appendix: Compaction Pseudocode

Algorithms 1 and 2 are iterative and recursive implementations of find with compression, respectively. Algorithms 3 and 4 are iterative implementations of find with splitting and find with halving, respectively. Algorithm 5 implements early linking by index with splicing.

Algorithm 1 Iterative FIND with Compression

```
procedure FIND(x)
    u ← x.p
    v ← u
    while u.p ≠ u do
        u ← u.p
    while v ≠ u do
        x.p ← u
        x ← v
        v ← x.p
    return u
```

Algorithm 2 Recursive FIND with Compression

```
procedure FIND(x)
    if x.p.p ≠ x.p then x.p ← FIND (x.p)
    return x.p
```

Algorithm 3 Iterative FIND with Splitting

```
procedure FIND(x)
    u ← x.p
    while u.p ≠ u do
        x.p ← u.p
        x ← u
        u ← u.p
    return u
```

Algorithm 4 Iterative FIND with Halving

```
procedure FIND(x)
    while x.p.p ≠ x.p do
        x.p ← x.p.p
        x ← x.p
    return x.p
```

Algorithm 5 UNITE with Early Linking by Index and Splicing

```
procedure UNITE(x, y)
    u ← x
    v ← y
    while u.p ≠ v.p do
        if u.p > v.p then
            u ↔ v
        w ← u.p
        u.p ← v.p
        if v = w then
            return true
        else
            u ← w
    return false
```