Nested Set Union

Daniel H. Larkin[1] and Robert E. Tarjan[1,2]

[1] Princeton University Department of Computer Science *
{dhlarkin,ret}@cs.princeton.edu
[2] MSR SVC

Abstract. We consider a version of the classic disjoint set union (union-find) problem in which there are two partitions of the elements, rather than just one, but restricted such that one partition is a refinement of the other. We call this the *nested set union problem*. This problem occurs in a new algorithm to find dominators in a flow graph. One can solve the problem by using two instances of a data structure for the classical problem, but it is natural to ask whether these instances can be combined. We show that the answer is yes: the nested problem can be solved by extending the classic solution to support two nested partitions, at the cost of at most a few bits of storage per element and a small constant overhead in running time. Our solution extends to handle any constant number of nested partitions.

Keywords: data structures, disjoint set union, union-find

1 Introduction

The *disjoint set union problem* is to maintain a partition under set union. More precisely, the problem is to maintain a collection of disjoint sets (the parts of the partition), each with a canonical element called its *root*, under an intermixed sequence of UNITE and FIND operations, defined as follows:

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

UNITE (x, y): If x and y are in the same set, return **false**; otherwise, unite the two sets containing x and y, choose a root for the new set, and return **true**.

Initially each set is a singleton whose root is its only element. During a UNITE, the implementation is free to choose any element in the set as the new root. The classic solution to this problem is to use a *compressed tree* [6]. Each set is represented by a rooted tree with a node for each element of the set; the tree root is the canonical element. Each node x has a pointer $x.p$ to its parent. The root points to itself. This representation makes implementing FIND and UNITE very simple. To execute FIND (x), follow parent pointers from x until reaching the root. The path of ancestors from x to the root is called the FIND *path*. To execute UNITE (x, y), first FIND the roots of the sets containing x and y. If they are the same, return **false**. Otherwise make one the parent of the other and return **true**.

* Research at Princeton University partially supported by NSF grant CCF-0832797.

This representation allows for a great deal of flexibility, and a rich body of literature explores the design space. The tree may be restructured freely during the traversal of a FIND path, and a variety of efficient methods for compacting each FIND path have been proposed. Furthermore, the root of a new set can be chosen arbitrarily, and several methods of making this choice, called *linking rules*, have been proposed. The most efficient algorithms combine a good form of compaction with a good linking rule.

Among the good compaction methods, arguably the simplest conceptually is *compression*, which replaces the parent of each node along the FIND path by the root [8]. Other good methods include *splitting* [13], *halving* [14], and *splicing* [2, 12].

Two simple linking rules are *naïve linking*, which when doing UNITE (x, y) chooses as the new root the root of the old tree containing x, and *linking by index* which requires that the elements be totally ordered and chooses as the new root the larger of the two old roots. Two more-efficient rules are *linking by size*, which chooses as the new root the root of the tree containing more nodes, and *linking by rank*, which maintains a non-negative integer rank for each node, initially 0, and chooses as the new root the old root of larger rank, increasing the rank of the new root by 1 if there is a tie. All these rules are deterministic. An efficient randomized rule is *randomized linking*, which totally orders the elements by choosing a fixed permutation uniformly at random and then does linking by index.

Tarjan and van Leeuwen [12] proved that compression, splitting, halving, or splicing in combination with linking by rank or compression, splitting, or halving in combination with linking by size takes $O(m\alpha(n, m/n))$ time to execute $m \geq n$ operations on sets containing a total of n elements, where α is a functional inverse of Ackermann's function. Goel et al. [7] recently obtained the same bound in expectation for randomized linking in combination with any of the four compaction methods. We will reference their paper extensively, and henceforth refer to it as GKLT. These results match the lower bound of Fredman and Saks [5] and thus are optimal to within a constant factor.

Some applications require the maintenance of more than one partition of the same collection of elements. An example is a recent algorithm of Fraczak et al. [4] for finding dominators in a flow graph, which needs to maintain two nested partitions of the node set. Of course one can maintain each partition using a separate data structure, but it is natural to ask whether one can save space by using a single compressed tree to represent both partitions, without sacrificing the inverse-Ackermann-function amortized time bound. We show that the answer is yes.

Let us define the problem precisely. For simplicity we consider the case of two nested partitions, but it is easy to extend our results to any fixed nesting depth. Given a set of n elements, we wish to maintain two nested partitions of the elements, the *coarse partition* and the *fine partition*, under intermixed FIND and UNITE operations on either partition. The only requirement is that the UNITE operations must maintain the nesting of the partitions: each fine UNITE must

combine sets contained in the same set of the coarse partition. Initially both partitions are the same, the partition into singletons.

Our solution to this problem represents both partitions by a single compressed forest, requiring one parent pointer and a few extra bits of information per node. Each coarse set is represented by a tree whose nodes are the elements of the set, with the root of the tree equal to the root of the set. We call such a tree a *coarse tree*. Each fine set is represented by a subtree whose nodes are the elements of the set, with the root of the subtree equal to the root of the set. We call such a subtree a *fine tree*; we call its root a *subroot*. The root of a coarse tree is also the root of a fine tree and hence also a subroot.

To complete the solution, we need algorithms for coarse and fine FIND and coarse and fine UNITE. We want to adapt algorithms for the one-level case, specifically path compaction methods and linking rules, to the two-level case. In doing this, we encounter two technical challenges, in the implementation of coarse FIND and that of fine UNITE. We cannot implement coarse FIND using any of the standard compaction methods because they can destroy the partition of the coarse tree into fine trees. To overcome this problem we do coarse FIND operations using two-level compression. First, we compress each part of the coarse FIND path that is within a single fine tree. This compresses the coarse FIND path into a path of subroots. Then we compress the path of subroots.

We call this method *segmented compression*. The idea of first compressing subpaths between subroots and then compressing the resulting path of subroots was used by Farrow in an unpublished algorithm for combining values along paths in rooted trees subject to arc additions. For a definition and some results on this problem see [3, 11]. In that application, there is no freedom in the way links are done. Farrow introduced a complicated definition of subroots and only managed to prove an $O(\log^* n)$ amortized time bound per operation. In our application, on the other hand, we can adapt standard linking rules, and we are able to obtain an inverse-Ackermann time bound per operation. Whether Farrow's algorithm has a $o(\log^* n)$ bound is an intriguing open problem.

The second challenge arises from the fine UNITE operations. If a fine UNITE is done on two fine sets whose subroots are children of the root of a coarse set, then making one of the subroots a child of the other uncompacts the coarse tree. This complicates the extension of standard linking rules to nested partitions, and it makes the analysis of the resulting algorithms much more than a straightforward extension of existing results.

We prove an asymptotically optimal bound for segmented compression used with an extended version of linking by rank. Our analysis is unconventional. Most analyses of disjoint set union structures use the "union forest" idea, which measures the effect of compactions on a forest built by the UNITE operations without any compaction. The fine UNITE operations prevent the use of this idea, requiring us to cope with non-fixed ranks and non-ancestral parent changes.

The remainder of our paper contains five sections. We begin in Section 2 by detailing our algorithms. In Section 3 we establish key properties of node ranks. In Section 4, we use these rank properties and adapt the analysis from GKLT

of standard compression to prove amortized bounds for segmented compression. We conclude in Section 5 with further discussion of addional results related to the design space, the extension to deeper nesting, and open problems.

2 Algorithms

In order to guarantee that the operations remain asymptotically efficient, we will adapt clever forms of linking and path compaction from other disjoint set union structures. In particular, we will adapt segmented compression and linking by rank. We call the root of each set in the coarse partition a *root* and one of a set in the fine partition a *subroot*. One simple property of our algorithms is that all roots are also subroots, but not vice-versa. Another is that fine sets are represented by contiguous subtrees.

For a node x, we denote its parent by $x.p$ and its rank by $x.r$. We also store a flag, $x.\gamma$ which is set to **true** if x is a subroot and **false** otherwise. We have one additional field for the purpose of analysis and which is not maintained by the algorithm. The *maximum rank* of a node, denoted $x.r_m$, is equal to the maximum value of $x.r$ over the sequence of operations.

Segmented compression was initially developed by Farrow to provide an efficient algorithm for path evaluation problems on unbalanced trees [3]. Farrow's work was quite complicated. There was no natural notion of a subroot in his application, so he used artificial functions to calculate subroot status. Furthermore, the path evaluation problem does not allow for the use of intelligent linking rules (and hence leads to unbalanced trees). His algorithm was designed to achieve an amortized $\mathrm{O}(\log^* n)$ bound per operation rather than the $\mathrm{O}(\log n)$ bound achieved by standard compression. So in truth, we are borrowing just the skeleton of his algorithm and will be providing a fully new analysis of it in this different setting.

Segmented compression can be viewed as a two-step process, though its implementation will be more direct. First, each non-subroot is made to point to its nearest ancestral subroot. Next, each subroot is made to point to the root of the tree. By default, segmented compression returns the roots of both the coarse and fine set containing the node to be found. All coarse FIND operations will be done using segmented compression, including those within UNITE operations. See Algorithms 1 and 2 for an implementation of FIND with segmented compression and Figure 1 for a visual example.

While one could use segmented compression for fine FIND operations, this is over-kill in a certain sense. We use a more local implementation for fine FIND operations. We perform standard compression within the fine subtree containing the node to be found and return the root of this subtree, which is the nearest ancestral subroot of the input node. We call this level-sensitive approach *local compression*. Local compression is compatible with the analysis throughout the paper, but we will not specifically mention it elsewhere for notational convenience, referring instead to just segmented compression.

Algorithm 1 Segmented Compression

```
procedure SegmentCompress(x)
    if x.p = x then
        return (x, x)
    (u, v) ← SegmentCompress(x.p)
    if x.γ = true then
        x.p ← u
        return (u, x)
    else
        x.p ← v
        return (u, v)        ▷ Segmented compression returns a (root, subroot) pair
```

Algorithm 2 Find with Segmented Compression

```
procedure Find(i, x)
    (u, v) ← SegmentCompress(x)
    if i = 0 then                                   ▷ Coarse partition
        return u
    else                                            ▷ Fine partition
        return v
```

We now describe our adapted form of linking by rank. Each node has an initial rank of 0. When performing a Unite at the coarse level, the algorithm works exactly like that of normal linking by rank. The root of each set is found using segmented compression. That of lesser rank is made the child of the other, making it the *loser* of the link. In case of a rank tie, the new parent is chosen arbitrarily and its rank is incremented. When performing a Unite at the fine level, the process is slightly different. The two subroots are found using compression. If one is already the parent of the other, all that needs to be done is to mark the child as a non-subroot. Otherwise, the one of lesser rank is made the child of the other. In case of a rank tie, the new parent is chosen arbitrarily and its rank incremented. If the rank of the winner is now equal to that of its parent, we need to make an adjustment. If the parent is the root, then the rank of the root is increased. Otherwise we set the parent pointer to the grandparent. This

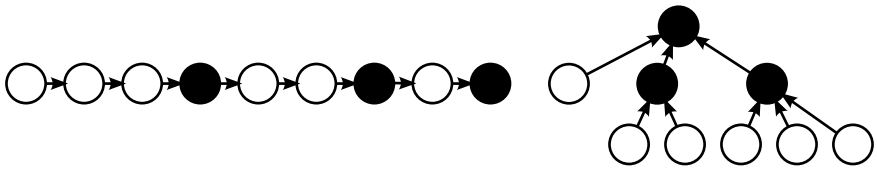

Fig. 1. The Find path on the left is segment compressed, resulting in the tree on the right. Subroots are shown in black, with non-subroots in white.

preserves strictly increasing ranks along a FIND path. In any such case, the loser of the link ceases to be a subroot. See Algorithm 3 for an implementation and Figure 2 for demonstration of different cases of subroot linking. It is also possible to allow limited rank ties. Instead of always ensuring that the rank of the root is *greater* than those of its children, one can instead simply ensure that it is *no less* than those of its children. This leads to a slightly better constant factor guarantee on the maximum rank, but the statement of the associated bound (Lemma 1) is a bit uglier.

Algorithm 3 Nested UNITE with Linking by Rank

```
procedure UNITE(i, x, y)
    u ← FIND(i, x)
    v ← FIND(i, y)
    if u = v then
        return false
    if u.r < v.r then
        u ↔ v
    v.p ← u
    if u.r = v.r then
        u.r ← u.r + 1
    if i = 1 then                          ▷ Check for and fix rank monotonicity
        v.γ = false
        if u = u.p then
            return true
        else if u.p = u.p.p and u.p.r = u.r then
            u.p.r ← u.r + 1
        else if u.p.r = u.r then
            u.p ← u.p.p
    return true
```

3 Properties of Node Ranks

We now establish a few key properties of node ranks with our modified algorithms. This lemma and the associated corollaries will allow us to adapt simple analyses of standard compression to work with segmented compression.

While it is almost trivial to prove that the sum of ranks is linear (each coarse UNITE can only contribute 1 to the sum, while a fine UNITE can contribute 2), we will need a somewhat stronger property. Let $R(k)$, $S(k)$, and $N(k)$ denote the maximum number of roots, subroots, and non-roots of rank k respectively (for the purpose of this analysis, "subroot" means a subroot which is not also a root).

Lemma 1. *Using linking by rank,* $R(k) \leq n \cdot (3/4)^k$ *and* $S(k) \leq 3/2 \cdot n \cdot (3/4)^k$.

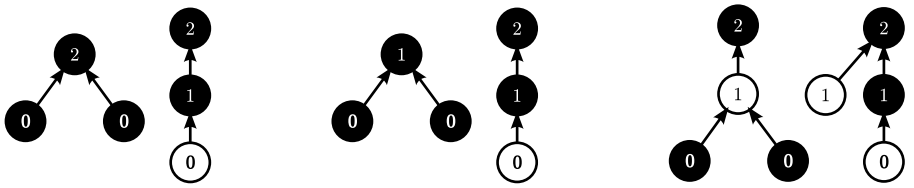

Fig. 2. Three different cases of subroot linking by rank. Subroots shown in black, with non-subroots shown in white. On the left, the two subroots are linked, and the rank of the winner increased. In the center, the rank increase causes the rank of the root to be equal to that of its child, so its rank is also increased. On the right, the rank increase causes the rank of the parent to be equal to that of its child, so the child's parent is set to the grandparent.

Proof. We examine different types of transitions a node can undergo, and demonstrate that credits can be transferred between the nodes involved in a way which requires no new credit to be introduced to the system. The transitions and associated node count adjustments are the following:

1. Two roots of ranks k and $j < k$ are linked. $R(j)$ decreases by 1 and $S(j)$ increases by 1.
2. Two subroots of ranks k and $j < k$ are linked. $S(j)$ decreases by 1 and $N(j)$ increases by 1.
3. Two roots of rank k are linked. $R(k)$ decreases by 2, $R(k+1)$ increases by 1, and $S(k)$ increases by 1.
4. Two subroots of rank k are linked underneath a root of rank at least $k+2$. $S(k)$ decreases by 2, $S(k+1)$ increases by 1, and $N(k)$ increases by 1.
5. Two subroots of rank k are linked underneath a root of rank $k+1$. $R(k+1)$ decreases by 1, $S(k)$ decreases by 2, $R(k+2)$ increases by 1, $S(k+1)$ increases by 1, and $N(k)$ increases by 1.

Each node must hold a minimum number of credits dependent on its type and rank. A root of rank k must hold at least $n_R(k)$ credits. Similarly a subroot and a nonroot of rank k must hold at least $n_S(k)$ and $n_N(k)$ credits respectively. We now examine the transitions and the requirements they impose on these these minimum values under the assumption that credits are preserved.

1. $n_R(k) \geq n_S(k)$
2. $n_S(k) \geq n_N(k)$
3. $2n_R(k) \geq n_R(k+1) + n_S(k)$
4. $2n_S(k) \geq n_S(k+1) + n_N(k)$
5. $n_R(k+1) + 2n_S(k) \geq n_R(k+2) + n_S(k+1) + n_N(k)$

These requirements leave some flexibility, but here is one simple and uniform way to satisfy them. We first set $n_N(k) = 0$ for all k, then we set $n_S(k+1) = {}^4\!/_3 n_S(k)$ and $n_R(k) = {}^3\!/_2 n_S(k)$. Finally we establish base values with $n_S(0) = 1$. This leads to a total credit of ${}^{3n}\!/_2$ in the system (n roots of initial rank 0).

One can easily verify that these transitions allow credits to be preserved while meeting the minimum requirements. We have that a root of rank k must have at least $n_R(0) \cdot (4/3)^k = 3/2 \cdot (4/3)^k$ credits. Therefore with a total of $3n/2$ credit in the system, at most $n \cdot (3/4)^k$ such nodes can exist. Adjusting for the value of $n_S(0)$, we also obtain a bound of at most $3n/2 \cdot (3/4)^k$ subroots of rank k.

Once credit leaves a root of rank k (either because it moves to a different node or because the node rank increases), it can never return to a root of rank k. This is also true of subroots and nonroots of a given rank. The manner in which credit is transferred imposes a partial order on node types. The lemma follows.

Corollary 1. *With linking by rank, the maximum node rank is* $O(\log n)$.

Corollary 2. *With linking by rank, the sum of ranks is* $O(n)$.

Corollary 3. *With linking by rank, the sum of maximum ranks is* $O(n)$.

4 Amortized Analysis

We will adapt the analysis of compression used in GKLT to count grandparent changes rather than parent changes. In particular we will use many of the same functional definitions and cite some building-block lemmas about them, but we will need to account for the non-fixed ranks and non-ancestral parent changes which may result from fine UNITE operations. Among other implications, this means we will not be able to use the standard "union-tree" approach, which assumes that all UNITE operations are performed first without compaction, and all compaction is performed on the final, full tree.

Ackermann's function [1, 10] is defined recursively on two non-negative integer variables:

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

A simple inductive argument shows that A is strictly increasing in both arguments, $A(k+1,j) \geq A(k,j+1)$, and $A(1,j) = j+2$. The function $A(k,j)$ increases very rapidly as k grows even a bit larger.

We will use the same definitions for the *inverse Ackermann function* $\alpha(r,d)$, the *index function* $b(k,r)$, and the *level function* $a(r,s)$ as found in GKLT. The variable $d = m/n$ is used for notational convenience. The inverse Ackermann function, defined for any non-negative integer r and non-negative real number d, is non-decreasing in r and non-increasing in d. The index function, defined for any non-negative integers k and r, is non-increasing in k and non-decreasing in r. Finally, the level function $a(r,s)$ is defined for any non-negative integers $r \leq s$.

$$\begin{aligned} \alpha(r,d) &= \min\{k>0 | A(k,\lfloor d \rfloor) > r\} \\ b(k,r) &= \min\{j \geq 0 | A(k,j) > r\} \\ a(r,s) &= \min(\{\alpha(r,d)+1\} \cup \{k \leq \alpha(r,d) \,|\, A(k,b(k,r)) > s\}) \end{aligned}$$

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

$$x.a = a\left(x.r, x.p.p.r\right)$$
$$x.b = b\left(x.a - 1, x.p.p.r\right) \text{ if } x.a > 0, x.b = 0 \text{ otherwise}$$

The following three lemmas about these definitions correspond to Lemmas 3.1, 3.2, and 3.3 in GKLT respectively, and the proofs are omitted.

Lemma 2. *If $r \leq s$, $a\left(r, s\right) = 0$ if and only if $r = s$.*

Lemma 3. *If $x.a \leq \alpha\left(x.r, d\right)$, then $x.b \leq \max\left\{x.r, 1\right\} \leq x.r + 1$.*

Lemma 4. *If $x.a = \alpha\left(x.r, d\right) + 1 = \alpha\left(x.p.p.r, d\right) + 1$, then $x.b \leq d$.*

With these preliminaries, we are ready to start building a framework to count grandparent changes. We start by defining a *count* for each node x.

$$x.c = x.a \times \left(x.r + 2\right) + x.b$$

The following two lemmas are the key to counting grandparent changes. Since ranks remain fixed during a FIND with the substitution of parents and grandparents the statements and proofs are equivalent to those of Lemmas 3.4 and 3.5 in GKLT (so, once more, the proofs are omitted).

Lemma 5. *During a* FIND*, for every node x, $x.c$ never decreases, and $x.c$ increases whenever $x.a$ or $x.b$ changes. If $x.a$ increases by k, $x.c$ increases by at least k.*

Lemma 6. *Let x and y be nodes such that $x.p.p.r \leq y.r$ and $x.a = y.a$ just before a* FIND *that sets $x.p.p$ to a node with rank at least $y.p.p.r$. Then the* FIND *increases $x.c$.*

Now we are ready to count grandparent changes. Let the potential of a node x be

$$\max\left\{0, \left(\alpha\left(x.r_m, d\right) + 1\right) \times \left(x.r_m + 2\right) + d + 1 - x.c\right\}.$$

Let the potential of a collection of trees be the sum of the potentials of their nodes, and let the amortized cost of an operation be the number of grandparent changes it makes plus the change in potential it causes (all but the last three nodes on the FIND path will change grandparents to the root).

Lemma 7. *The initial potential is $O\left(n\alpha\left(n, d\right) + m\right)$.*

Proof. By Corollary 3 the sum of maximum ranks is linear. By Corollary 1 the value of $\alpha\left(x.r_m, d\right)$ is $\mathrm{O}\left(\alpha\left(n, d\right)\right)$. Thus summing over the first product term, we get at most $\mathrm{O}\left(n\alpha\left(n, d\right)\right)$. An additive $\mathrm{O}\left(m + n\right)$ term comes from summing $d + 1$ over each node x.

Lemma 8. *The sequence of* UNITE *operations increases the potential by at most $O\left(n\alpha\left(n, d\right) + m\right)$.*

Proof. A UNITE can change two parent pointers and increase up to two ranks. This means that three nodes may have their counts decreased, contributing an increase to the potential.

We consider the potential increase contributed by the parent changes first. Since the new parent of the loser need not be an ancestor, it is possible to completely reset the level and index. This can happen only once per node though, as the node ceases to be a subroot. Thus if we sum over the maximum node counts for each node, then with Corollary 3 and Lemmas 3 and 4, we get

$$\begin{aligned}\sum_x x.c &= \sum_x (x.a \times (x.r + 2) + x.b) \\ &\leq \sum_x ((\alpha(n,d) + 1) \times (x.r_m + 2) + x.b) \\ &= (\alpha(n,d) + 1) \sum_x x.r_m + \sum_x x.b \\ &= \mathrm{O}(n\alpha(n,d) + m).\end{aligned}$$

It is also posible that the UNITE changes the parent of the winner, though in this case the rank of the grandparent can only increase. Thus it cannot increase the potential.

Now we consider the effect of rank increases on node counts. Each UNITE may increase the rank of a root, but when a root changes rank, its level and index remain fixed at 0, meaning there is no change in potential. We need only examine rank changes in non-root subroots. Each fine UNITE may increment the rank of one such node x. A given node may have its rank updated many times this way over the course of multiple UNITE operations; however, we can use Lemma 1 to bound the total change. The decrease in count due to a rank increase of x is at most $x.a \times (x.r + 2) + x.b$. By Lemma 1 there can be at most $3/2 \cdot n \cdot (3/4)^k$ subroots of rank k, and subsequently at most $3/2 \cdot n \cdot (3/4)^k$ UNITE operations increase a subroot's rank from $k-1$ to k. Since $x.r$ is $\mathrm{O}(\log n)$ by Corollary 1 for all nodes, we can sum the potential increased in this manner over all UNITE operations with relevant nodes indexed from 1 to $j < 3n$

$$\begin{aligned}\sum_{i=1}^{j} x_i.c &\leq (\alpha(n,d) + 1) \sum_{i=1}^{j} (x_i.r + 2) + \sum_{i=1}^{j} x_i.b \\ &\leq 3/2 \cdot n \cdot (\alpha(n,d) + 1) \sum_{k=0}^{\infty} (k+2) \cdot (3/4)^k + \sum_{i=1}^{j} x_i.b \\ &\leq 45n \cdot (\alpha(n,d) + 1) + \sum_{i=1}^{j} (x_i.r + 1) + \sum_{i=1}^{j} d \\ &\leq 45n \cdot (\alpha(n,d) + 1) + 3/2 \cdot n \cdot \sum_{k=0}^{\infty} (k+1) \cdot (3/4)^k + 3m \\ &= 45n \cdot (\alpha(n,d) + 1) + 36n + 3m\end{aligned}$$

to get a bound of $O(n\alpha(n,d)+m)$ with the help of Lemmas 3 and 4. Thus the total increase in potential due to UNITE operations is $O(n\alpha(n,d)+m)$.

Lemma 9. *The amortized cost a* FIND *using segmented compression is* $O(\alpha(n,d))$.

Proof. Consider any FIND path. Segmented 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 grandparent is changed by the segmented compression. For all such nodes, $x.a > 0$ by Lemma 2. If $x.a > 0$, $\alpha(x.r,d) = \alpha(x.p.p.r,d)$, and there is a node y after x on the path such that $x.p.p.r \leq y.r$ and $y.a = x.a$, it must be the case that the FIND increases $x.p.p.r$ to at least $y.p.p.r$, since it sets $x.p.p = v$. Thus, segment compressing the path reduces the potential of x by at least two by Lemmas 3, 4, and 6. Thus the segmented compression decreases the potential of x unless $\alpha(x.r,d) < \alpha(x.p.r,d)$ or x is either the last or second–to–last on its level with rank at least $x.r$. Since $\alpha(x.r,d) \leq \alpha(x.p.r,d)$ for every x, at most $2\alpha(v.r,d)$ nodes x have $\alpha(x.r,d) < \alpha(x.p.p.r,d)$. Since every node on the path has level at most $\alpha(v.r,d)+1$, at most $2\alpha(v.r,d)+2$ nodes are last or second–to–last on their level. The amortized cost of the FIND is thus at most $4\alpha(v.r,d)+2$. By Corollary 1 $v.r$ is $O(\log n)$. Subsequently the amortized cost of the FIND is $O(\alpha(n,d))$.

Theorem 1. *The total time to execute* $m \geq n$ *operations using linking by rank and segmented compression is* $O(m\alpha(n, m/n))$.

Proof. Each UNITE takes constant real time, uses two internal FIND operations, and may increase the potential. Thus, the theorem follows from Lemmas 7, 8, and 9.

5 Remarks

We have shown that the two-level nested set union problem can be solved in optimal time to within a constant factor while only using one pointer and very little additional space per element. We have provided a relatively simple deterministic solution. We will now proceed to address some natural questions about extensions to the problem and the design space around it, some of which remain open.

Additional results in the design space. It is natural to wonder whether other algorithms for disjoint set union may be adapted to the nested case. In our full paper [9], we answer this question in the affirmative. It is possible to adapt linking by size to work instead of linking by rank, though with slightly worse constants. In addition to segmented compression, we have developed segmented versions of splitting and halving, thereby offering one-pass compaction methods. Any of the three compaction methods can be used with either deterministic linking rule, and their standard (local) versions can be used for fine FIND operations, including those within fine UNITE operations. It has also been shown that randomized

linking works in conjunction with segmented compression. It is unclear whether it will work with the other compaction methods, including local compression. The simple permutation argument to bound rank ties no longer holds. Currently the proof relies critically on the fact that fine UNITE operations can only introduce a single new ancestor (i.e. both subroots are children of the root before linking).

Deeper nesting. As we do not yet have a concrete application for nesting deeper than two levels, we did not study this extension in full depth. Both the algorithm itself and the presentation of the analysis are more complicated, but all the core tools for such work are present in this paper. Rather than a single subroot bit, a small integer field would be kept to maintain the finest level at which a node is still a root. The natural generalization of segmented compression would make each node point to the nearest coarser node on the FIND path, and each recursive call would return a k-tuple rather than a pair. This solution increases the potential by a factor of k, adds an $\mathrm{O}(k)$ term to the amortized time for a FIND operation, and requires $\mathrm{O}(\log k)$ extra bits per node. This makes it optimal up to a constant factor when k is $\mathrm{O}(\alpha(n, m/n))$ and kn is $\mathrm{O}(m)$. It remains open whether a better solution could be had for larger k.

References

1. Wilhelm Ackermann. Zum hilbertschen aufbau der reellen zahlen. *Mathematische Annalen*, 99(1):118–133, 1928.
2. Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.
3. Rodney Farrow. Efficient on-line evaluation of functions defined on paths in trees. Technical Report 476-093-17, Rice University, 1977.
4. Loukas Fraczak, Wojciechand Georgiadis, Andrew Miller, and Robert E. Tarjan. Finding dominators via disjoint set union. *J. Discrete Algorithms*, 23:2–20, 2013.
5. Michael L. Fredman and Michael E. Saks. The cell probe complexity of dynamic data structures. In *Proc. 21st Annual ACM Symposium on Theory of Computing*, pages 345–354, 1989.
6. Bernard A. Galler and Michael J. Fisher. An improved equivalence algorithm. *Commun. ACM*, 7(5):301–303, 1964.
7. Ashish Goel, Sanjeev Khanna, Daniel H. Larkin, and Robert E. Tarjan. Disjoint set union with randomized linking. In *Proc. 25th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1005–1017, 2014.
8. Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms, Third Edition.* Addison-Wesley, 1997.
9. Daniel H. Larkin and Robert E. Tarjan. Nested set union. CoRR, 2014.
10. Rozsa Péter. *Rekursive funktionen.* Académiai Kiadó, 1951.
11. Robert E. Tarjan. Applications of path compression on balanced trees. *J. ACM*, 26(4):690–715, 1979.
12. Robert E. Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. *J. ACM*, 31(2):245–281, 1984.
13. Theodorus P. van der Wiede. *Datastructures: An Axiomatic Approach and the Use of Binomial Trees in Developing and Analyzing Algorithms.* Mathematisch Centrum, 1980.
14. Jan van Leeuwen and Theodorus P. van der Wiede. Alternative path compression techniques. Technical Report RUU-CS-77-3, Rijksuniversiteit Utrecht, 1977.