Compressing Trees with a Sledgehammer

Daniel H. Larkin

A Dissertation

Presented to the Faculty

of Princeton University

in Candidacy for the Degree

of Doctor of Philosophy

Recommended for Acceptance

by the Department of

Computer Science

Advisor: Robert E. Tarjan

January, 2016

Abstract

A variety of problems can be solved more efficiently with the use of *compressed trees*. The compressed tree method is flexible and can be applied in different ways to solutions for many problems including disjoint set union, nested set union, and path evaluation.

The central contribution of this thesis is a unified framework for the analysis of the compressed tree method and a meta-theorem that proves an inverse-Ackermann-type bound for a class of algorithms. The meta-theorem requires only that the compaction method be well-structured and that the other operations maintain a forest structure that admits a well-behaved rank function. These are the weakest known conditions of any analysis to-date that proves such a bound.

We use the framework to give a clean, compact alternative analysis of all but one of the known efficient algorithms for disjoint set union; we provide an explicit proof of optimality for the remaining known optimal method, splicing by rank, which was left as an exercise by Tarjan and van Leeuwen [24]; we prove that a new algorithm for disjoint set union called *randomized linking* is optimal, giving theoretical backing for some recent experimental results; and finally we proceed to explore the nested set union problem, giving an optimal algorithm for any constant number of partitions as well as a few alternatives for the restricted case of two partitions.

Acknowledgements

. . . and at once I knew, I was not magnificent.

JUSTIN VERNON, *"Holocene"*

First and foremost, I would like to thank my advisor, Bob Tarjan, without whom this thesis would not have been possible. His guidance allowed me to mature greatly as a researcher, and his collaborative efforts helped me accomplish much of the work contained in this thesis. I give further thanks to Jeff Erickson for helping me find my niche, for guiding me into a fantastic graduate program, and for his continued mentorship throughout the intervening years. Thank you as well to my other committee members—Bob Sedgewick, Sanjeev Arora, and Zeev Dvir—for their time, feedback, and consideration of this thesis. Additional thanks is owed to my other coauthors: Ashish Goel and Sanjeev Khanna, whose joint work appears in part here; Sid Sen, whom I had the pleasure to work with quite closely early on in my research career; as well as Shiri Chechik, Liam Roditty, Grant Schoenebeck, and Virginia Vassilevska Wiliams.

On a more personal level, I owe a great deal to my family, my friends, and my lovely wife–to–be for supporting me, not just during my Ph.D. but also in the twenty-two preceding years that led me to this path. I thank you all for putting up with me at my worst and celebrating with me at my best, as cliché as that may sound. Seriously, I love you guys.

Finally, I would like to acknowledge that this thesis would not exist if not for the hundreds of gallons of coffee that have fueled my waking hours.

Contents

List of Algorithms

List of Figures

Chapter 1

Introduction

1.1 Efficiency via Compressed Trees

A variety of problems can be solved more efficiently with the use of *compressed trees*. The compressed tree method is flexible and can be applied in different ways to solutions for many problems. The common theme, what makes this a cohesive method, is the similarity of the effects of the techniques used—all start with some input forest and in some way partially flatten the trees as computation proceeds in order to reduce future work. We call such techniques *compaction methods*, and highlight compaction as the central component. Any other operations performed are considered auxiliary.

We will provide a formal definition of the compressed tree method in Section 1.2, but first we will define and discuss three concrete problems. These are instances that highlight the algorithmic and analytical techniques central to the method and demonstrate the considerable flexibility that it offers.

1.1.1 Disjoint Set Union

The most well-studied problem that can utilize the compressed tree method is that of *disjoint set union*, sometimes called *union-find*. In this setting, the goal is to maintain a partition on n elements subject to an online sequence of $m \geq n$ operations. Each set has a distinguished element called the *root*, which serves as a canonical representative of the set. The implementation is free to choose the root element. The operations allowed are defined as follows.

- FIND(x) — Find and 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, choose a new root for the combined set, and return **true**.

Initially each set is a singleton, making its only element its root. The implementation is free to choose the new root from among the resulting set's elements in each UNITE that combines two sets.

Early work on disjoint set union can be viewed as the historical origin of the compressed tree method—as early as 1964, Galler and Fischer proposed a tree-based solution to the problem [10]. Each element is equated with a node in a forest. Each node u has a pointer u.PARENT to its parent, and each set is represented by a tree. The root of a tree is the root of the corresponding set, and its parent is itself. To implement FIND(x), one follows parent pointers from x until reaching a node u such that $u = u$.PARENT, indicating a root. The collection of nodes traversed during a FIND is called the FIND *path*, and the final node is called the *head*. To perform UNITE(x, y), one calculates $u =$ FIND(x) and $v =$ FIND(y), and if $u \neq v$, arbitrarily sets either u.PARENT $\leftarrow v$ or v.PARENT $\leftarrow u$.

This solution still leaves some flexibility, and one can exploit this to make the implementation faster. The structure of a given tree is entirely arbitrary so long as the node set remains the same. Therefore, the implementation is free to restructure the trees as desired.

One way to gain efficiency is to use a *linking rule* to choose the new root during a UNITE that combines two sets. The simplest linking rule, used for comparison of worst-case efficiency, is *naïve linking*, which for UNITE(x, y) always choses FIND(x) as the new root. Another rule with similar worst-case efficiency is *linking by index* [10], which requires the nodes to be totally ordered and chooses as the new root the larger of FIND(x) and FIND(y). Two better rules are *linking by size* and *linking by rank*. Knuth attributes linking by size to McIlroy [14], and it chooses as the new root the old root whose tree contains more nodes, breaking a tie arbitrarily. Linking by rank, proposed by Tarjan and van Leeuwen [24], chooses as the new root the old root whose tree would be taller absent the effect of compaction. Implementing linking by size or linking by rank requires storing the size or height of each tree respectively.

Another way to gain efficiency is to compact the FIND path as it is traversed. The most drastic form of path compaction is *compression*, attributed by Knuth to Tritter [14]. Compression changes the parent of each node on the FIND path to be the head of the path. This requires two passes over the path—one to find the head of the path and one to change all the parents. Alternatives to compaction include *splitting*, which replaces the parent of each node on the FIND path with its grandparent, and *halving*, which does the same for every other node on the path [25, 26]. Both these methods, proposed by van Leeuwen and van der Wiede, require only one pass over the FIND path and do considerably less flattening than compression.

In some settings when performing UNITE(x, y), one can perform FIND(x) and FIND(y) simultaneously to further accelerate the process. One can take steps along the two paths concurrently, using some ordering on the nodes to determine which pointer to advance at each step. This process terminates once one root is found, and this root is linked underneath the current node on the other path, hence leading to the moniker *early linking*. It is important to note that since the second root is not found, this technique will not work in any setting

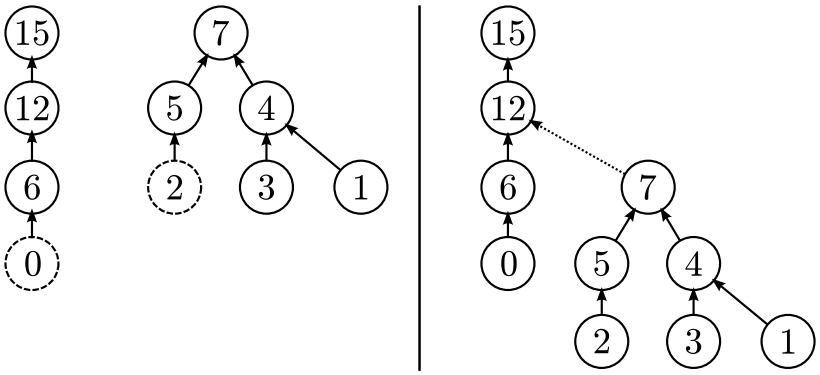

Figure 1.1: Consider executing UNITE$(0, 2)$. The pointers are advanced to the next node on each path choosing the node with lesser index. Once the first root, 7, is found, it is linked to the parent of the current node on the other path, 12.

that requires updating the information stored with the root of the newly formed set. For this reason, we cannot adapt this technique for use with linking by size, which requires that the size of the smaller root be added to that of the larger. We can, however, consider *early linking by index*, which uses indices to order the nodes, and *early linking by rank*, which uses ranks. In either case, we need not update information stored in the new root. See Figure 1.1 for an example of UNITE with early linking by index.

Any of the compaction methods discussed so far can be combined with any of the linking rules. Furthermore, instead of compacting the two paths separately when using early linking, it is possible to interleave their compaction. Prior to taking a step along one path, *splicing* sets the parent of the current node on that path to be the parent of the current node on the other path. The method of early linking by index with splicing is attributed by Dijkstra to Rem [6], while early linking by rank with splicing was proposed by Tarjan and van Leeuwen [24]. Splicing can only be used on pairs of paths, so one of the other methods must be used for individual FIND operations.

Thus far we have only discussed these methods algorithmically and made vague claims about their relative efficiency. All bounds stated here are for a sequence of $m \geq n$ operations, which is a reasonable assumption given that not all nodes are even accessed if $m < n$. Tarjan

was the first to prove an inverse-Ackermann type bound for the problem, specifically showing that linking by size with compression takes $O(m\alpha(n, m/n))$ time [21]. (See Section 2.2 for a definition of α.) In the same paper, he proved a matching lower bound, but only for this specific algorithm. In a later paper, he proved a matching lower bound for a much broader class of algorithms [23]. Tarjan and van Leeuwen proved in a later paper that in fact several algorithms have the same upper bound: any of compression, splitting, and halving in conjunction with linking by size or linking by rank, as well as splicing with early linking by rank. They also proved that several algorithms do not admit an inverse-Ackermann upper bound but instead have logarithmic worst-case performance. For instance, using any of the aforementioned compaction methods with naïve linking, linking by index, or early linking by index results in logarithmic performance. See their paper for more details on these results [24].

Other work has mostly focused on simplifying and tightening these bounds. Fredman and Saks gave an information theoretic lower bound that applies to virtually any algorithm for the problem, including randomized methods [9]. This tells us that not only do the analyses of Tarjan and van Leeuwen give optimal bounds for the algorithms considered, the algorithms are in fact optimal for the problem itself. Kozen gave a simplified version of Tarjan and van Leeuwen's analysis for linking by rank with compression [15]. Kaplan et al. improved Kozen's analysis to give a local bound sensitive to the individual set sizes involved in each operation [13]. Alstrup et al. simplified this result and converted it to a potential-based argument [3], an idea that has carried through to the analysis in Chapter 2. Each of these analyses, as well as the one presented in Chapter 2, counts parent changes in a bottom-up fashion, considering changes that happen to individual nodes rather than examining the larger structure of the tree. Siedel and Sharir offered a top-down analysis of linking by rank with path compression that attains a global inverse-Ackermann bound but does not offer local guarantees [20].

Patwary et al. conducted extensive experiments with dozens of implementations in application to Kruskal's minimum spanning tree algorithm [16, 18]. The inputs to Kruskal's algorithm were from three classes of graphs—two classes that were generated randomly and one class of "real-world" graphs (which may very well have the vertices specified in some random order). The study found that on these inputs linking by index with one-pass or interleaved methods, such as splitting and splicing respectively, typically performed best. This runs counter to the intuition that linking by rank and linking by size are better than linking by index based on worst-case performance. Wondering why linking by index works better in this setting, we asked: *does randomization help linking by index in a provable manner?*

Yes, it does.

We propose a new linking rule called *randomized linking* based on linking by index. It has expected efficiency similar to linking by rank and linking by size. We also propose a variant called *early randomized linking* based on early linking by index. Specifically, we apply a uniformly random permutation to the nodes and then perform linking by index or early linking by index using the total ordering given by the permutation. This is in effect the algorithm used in the experiments of Patwary et al. By exploiting a relaxation of the traditional analysis which we develop in Chapter 2, we prove in Section 3.2.3 that these linking rules attain the optimal bound in expectation.

In Chapter 3 we will provide more detailed descriptions of each of these optimal methods, as well as offer proofs of the optimal upper bounds using the results of Chapter 2.

1.1.2 Path Evaluation

The *path evaluation problem* abstracts a variety of problems on graphs, including disjoint set union. Solutions can be used to compute nearest common ancestors off-line, verify and construct minimum spanning trees, do interval analysis on a flowgraph, find the dominators of a flowgraph, and build the component tree of a weighted tree [5, 22].

Let $(S, \oplus)$ be a semigroup with associative operation $\oplus$. The path evaluation problem begins with a forest of n singleton nodes each labeled with an element of S. A solution must then process a sequence of $m \geq n$ intermixed operations of the following types.

- EVAL(x) — Return the product of all labels on the path from x to the root of its tree.
- UPDATE(x, ℓ) — Given a root x with label x.LABEL, set x.LABEL $\leftarrow \ell \oplus x$.LABEL.
- LINK(x, y) — Given two roots x and y, set x.PARENT $\leftarrow y$.

Note that LINK specifies which node becomes the new root, and thus precludes the use of linking rules. This can lead to exactly the type of unbalanced trees that give worst-case examples for compaction methods such as compression, splitting, and halving. Thus, these algorithms can only guarantee logarithmic performance in this setting.

Although certain classes of input leave the trees suitably balanced to guarantee an optimal bound [22], the best known bound for the general problem is $O(m \log^* n)$, due to Farrow [7]. He proposed two related algorithms. *Segmented compression*, which attains the aforementioned bound, partitions the tree into good and bad nodes and handles the two differently. Subpaths of bad nodes are labeled with contiguous sets of integers, and then complicated bit manipulation of the labels is used to designate a subset of these bad nodes as subroots. The subroots act as barriers to compression, such that each non-subroot has its parent changed to the nearest ancestral subroot, and then each subroot has its parent changed to the root. Farrow very carefully defined these functions on the labels in order to impose an artificial structure on the trees that allows him to adapt the $\log^*$ disjoint set union analysis of Hopcroft and Ullman [12]. Farrow claimed that his second technique, *stratified compression*, attains the optimal inverse-Ackermann bound. Unfortunately, this claim has not been verified. His paper contains several bugs, and it is not clear whether these prevent his techniques from working. In any case, stratified compression applies the ideas of segmentation recursively, baking the inverse-Ackermann function into the algorithm itself.

Both these algorithms are very complicated, in no small part because they were engineered to fit the analysis. The explicit decomposition into good and bad subpaths further complicates the picture. In Chapter 2, we discuss some proposed techniques that may help transfer this decomposition from the algorithm to the analysis, thereby allowing for more natural algorithms. We hope to encourage the development of simpler algorithms that match or improve upon the $O(m \log^* n)$ bound.

1.1.3 Nested Set Union

The *nested set union* problem is a variation on disjoint set union. Here, the goal is to maintain k partitions subject to the constraint that the $(i+1)^{th}$ partition is a refinement of the i^{th}. The operations are the same as in the disjoint set union problem, but are augmented with an extra parameter that specifies the partition on which to operate.

- FIND(i, x) — Find and return the root of the set containing x in partition i.
- UNITE(i, x, y) — If $i > 1$ and x and y are in different sets in partition $i-1$, return **error**; if x and y are in the same set in partition i, return **false**; otherwise unite the two sets in partition i, choose a new root for the combined set, and return **true**.

Note that the restriction specifying that x and y must be in the same set in partition $i-1$ in order to combine the two sets preserves the nesting property. Also note that since for $k = 1$ this is the same problem as previously defined, nested set union is a generalization of disjoint set union. As such, the inverse-Ackermann lower bound of Fredman and Saks still applies.

The problem arises out of an algorithm for finding dominators in flow graphs due to Fraczak et al. [8]. Their algorithm uses two instances of the disjoint set union problem that happen to obey the nesting property specified. This nested problem can of course be solved by using k separate structures that solve the original problem. It is natural to wonder whether

one can save space and store all the partitions using a single structure with one pointer per node without sacrificing the inverse-Ackermann bound on the running time. In Chapter 4, we provide an algorithm that does so for any constant number of nested partitions, as well as some alternatives that handle the restricted case of two partitions.

Our solutions are all based on subpartitioning the trees in the forest. Each set in the first, coarsest partition is represented by a single tree. Each set in the second-coarsest partition is represented by dividing each tree into contiguous subtrees. The root of each such subtree is called a *subroot* and remains eligible for linking in the second partition. To represent progressively finer partitions we further subdivide each tree into finer subtrees.

We need to account for these changes when we adapt our algorithms. The idea of segmented compaction, borrowed from Farrow [7], preserves the partition by preventing parent changes from changing the node sets of fine subtrees. Modifications to linking by rank and linking by size allow us to prove that subroot linking preserves certain important invariants about node ranks and set sizes.

One important thing to consider is that the traditional analyses for disjoint set union algorithms all use an offline fiction—the "union forest" approach— in which we suppose all the UNITE operations are performed first without any compactions, then all the compactions are carried out on the resulting forest. The nodes accessed during these offline compactions are in exactly the same sequence as they would be in the the original sequence. Most significantly, this allows the analyses to define very simple, fixed rank functions based on height in the tree. Unfortunately, the union-forest approach does not appear to generalize easily to algorithms for the nested problem. This results in a need to cope with changing ranks.

1.2 The Compressed Tree Method

Sometimes it is entirely appropriate to kill a fly with a sledgehammer!

Maj. I. L. Holdredge

The design and analysis of all these methods have been largely separate. Proposed methods have tended to arise out of experimentation and intuition without explicit guiding design principles. Moreover, each combination of linking rule and compaction method historically has been analyzed separately, requiring to some extent the reinvention of various techniques in order to tailor them to the specific application.

The central portion of this thesis seeks to address these issues with a unified analysis—a meta-theorem that applies to virtually all known optimal algorithms utilizing the compressed tree method. There are two main points to take away from this result. First, virtually all the known optimal algorithms share certain key properties that allow them to share a core analysis. Second, this seems to suggest that when developing new algorithms either for increased practical efficiency or to address related applications, these properties may be desirable. While it is not known whether these properties are necessary to obtain the optimal bound, and in fact it seems unlikely that they provide an exact characterization, they are the weakest sufficient conditions of any analysis to-date.

Before we can state the meta-theorem, it is necessary to define the compressed tree method more precisely. As in all the problems defined in the previous section, the goal is to maintain a rooted forest of n nodes, initially all singletons, under a sequence of $m \geq n$ compactions intermixed with any number of other auxiliary operations. Compact($\langle u_1, u_2, \ldots, u_h \rangle$) takes an input sequence and changes the parents of some of the input nodes subject to the following constraints:

- for each $i < j$, either u_i is a proper descendent of u_j or neither u_i nor u_j has the other

as an ancestor;

- the parent changes do not introduce any cycles;
- for each node which has its parent changed, the new parent is contained in the input sequence;
- and for each such node, if the old parent is contained in the input sequence, the new parent is greater in input order.

Since we make few explicit assumptions about the behavior of the algorithms, we do not give an explicit running time, but rather a bound on the number of nodes *processed*, or equivalently, the combined length of all compacted sequences. Luckily, most algorithms which utilize the compressed tree method spend only constant time on each node processed.

The auxiliary operations are free to restructure the forest arbitrarily so long as no cycles are introduced. To analyze any specific algorithm, we will require the definition of a rank function over the nodes—each node u is assigned a non-negative integer rank $u.\textsc{rank}$ that may vary with time. We use this function to gauge the effects of the auxiliary operations on the structure of the forest and the way they interact with compaction. It is important to note that this function is merely analytical and the algorithm need have no knowledge of it.

The only auxiliary operation we will consider explicitly is $\textsc{Link}(u, v)$, which sets $u.\textsc{parent} \leftarrow v$. We do not require that u and v both be roots, but only that u is not an ancestor of v; if u has a parent prior to the link, it is detached from its old parent by the link. Linking is a primitive used extensively in algorithms for the disjoint and nested set union problems, as well as path evaluation. It can also be used to build an arbitrary initial forest configuration. For these reasons, we will prove some useful lemmas about linking in Section 2.4.

With the method defined and the notion of an analytical rank function introduced, we can now discuss our meta-theorem more meaningfully. The bound incorporates an *added*

potential term ϕ_+ that accounts for the effects of auxiliary operations. The theorem can be stated as follows.

Theorem 2.3.2 (Sledgehammer Theorem). *Using well-structured compaction in the presence of weakly well-behaved ranks, a sequence of m compactions processes $O(m\alpha(n, d) + \phi_+)$ nodes, where $d = m/n$.*

Note that if the added potential does not dominate the other term in the bound, as we will prove is the case for typical uses of linking, this simplifies to the optimal $O(m\alpha(n, m/n))$.

As we will see in Chapter 2, the properties required of a compaction method to be "well-structured" and of a rank function to be "well-behaved" are reasonably broad. They apply to all but one of the known optimal algorithms, and there is plenty of flexibility to develop compliant algorithms in the future.

These properties will be quite straightforward to prove in many cases. In all the cases we consider, and most we can envision, the user need not understand the intricacies of inverse-Ackermann type functions to apply the Sledgehammer Theorem (2.3.2). Certainly, some will find this a boon.

Notes

This thesis draws heavily on the author's previous work, specifically "Disjoint Set Union with Randomized Linking," joint with Ashish Goel, Sanjeev Khanna, and Robert Tarjan, which appeared in SODA 2014 [11]; and "Nested Set Union," joint with Robert Tarjan, which appeared in ESA 2014 [17]. The analysis in Chapter 2 is largely based on those appearing in both these papers. Chapter 3 provides an alternative analysis of and builds on the work in the former, while Chapter 4 does the same with regard to the latter.

This research was partially supported by NSF grants CCF-0832797 and CCF-1420112.

Chapter 2

The Sledgehammer Theorem

As explained in Chapter 1, this chapter deals with a meta-theorem that applies to many algorithms which use the compressed tree method. We will use it extensively throughout Chapters 3 and 4, and it can potentially be used to analyze other algorithms in the future. Because of this broadness of applicability, we have taken to affectionately calling it the "Sledgehammer Theorem."

This chapter proceeds in five sections. In Section 2.1 we provide a high-level overview and some intuition for our approach, defining the sufficient conditions we will use to prove our primary theorem. Section 2.2 holds some mathematical preliminaries, notational definitions, and some key building-block lemmas. In Section 2.3 we prove our primary theorem. Section 2.4 discusses the use of the theorem and proves two useful lemmas about linking. We conclude the chapter in Section 2.5 with a discussion about future work on extending the theorem to handle path evaluation and some additional remarks.

2.1 How Little Is Sufficient?

A sledgehammer breaks glass but forges steel.

LEON TROTSKY

Virtually all the analyses of compaction on balanced trees make use of two key properties. The first is the ability to define an appropriate *rank* function on the nodes. The second relates specifically to the local guarantees made by the compaction method. We will provide generalized, relaxed forms of both these properties and utilize them for our bounds. While it is not clear, and in fact seems unlikely, that these conditions are necessary for an algorithm to perform optimally, they are the weakest sufficient conditions used by any known analysis to-date. Before we get to the full general forms, we will consider some of the specific versions used in other analyses to inform our choice and provide some intuition.

2.1.1 Well-Behaved Ranks

Nearly all the extant analyses of compaction use rank functions defined over the nodes in order to analyze progress made by compaction. There are two key properties of these rank functions used: ranks are non-decreasing along a path, and the sum of ranks is linear. In some settings the ranks are strictly increasing along a path, while others allow a constant number of nodes on a path to share the same rank.

Linking by rank and linking by size can both be analyzed by defining the rank $u.\textsc{rank}$ of a node u to be its height in the tree formed by the UNITE sequence without the use of compaction. This clearly implies strictly increasing ranks along paths, and because the trees are balanced, the rank sum will indeed be linear. Randomized linking requires a different approach. The nodes are distinctly labeled with a uniformly random permutation of the numbers from 1 to n, and $u.\textsc{rank} = \lfloor \lg n \rfloor - \lfloor \lg(n - u - 1) \rfloor$, where lg is the base-2

logarithm. The randomized linking algorithm ensures non-decreasing ranks along paths, and the definition of ranks implies that the rank sum is linear. The expected number of ancestors of a given node with the same rank is constant, as we will prove in Section 3.2.3.

We now discuss a non-negative integer rank function defined over all nodes. The value of $u.\text{RANK}$ is allowed to vary through time, but must remained fixed during each compaction. We consider the notion of *rank ties*, or proper ancestors v of u such that $u.\text{RANK} = v.\text{RANK}$. At a given point, the current number of rank ties for a node u is denoted $u.\text{TIES}$. We also want to consider the total number of ties throughout the sequence, or the *accumulated rank ties*, denoted $u.\text{TIES}_{acc}$. To define this formally, let $u.\text{TIES}_{set}$ be defined as the set of all pairs (v, i) such that v is a proper ancestor of u with $u.\text{RANK} = v.\text{RANK}$ at the beginning of the i^{th} compaction, but not at the end of the $(i-1)^{th}$ compaction. Then the number of accumulated rank ties is equal to the cardinality of this set, or $u.\text{TIES}_{acc} = |u.\text{TIES}_{set}|$. One can think of this quantity as a counter which does not decrement—each time u gains an ancestor of equal rank, $u.\text{TIES}_{acc}$ is incremented. A given node v can contribute multiple times to $u.\text{TIES}_{acc}$ if it is an ancestor of u with equal rank, ceases to be so, then becomes such again. Finally, let $u.\text{RANK}_{max}$ denote the maximum rank attained by u throughout the operation sequence. Consider then the following properties.

Property 1. *The rank* $u.\text{RANK}$ *of each node* u *is non-decreasing in time.*

Property 2. *For each compacted sequence* $\langle u_1, u_2, \ldots, u_h \rangle$ *and all* $1 \leq i < h$, $u_i.\text{RANK} \leq u_{i+1}.\text{RANK}$.

Property 3. *The total number of rank ties,* $\sum_{u \in N} u.\text{TIES}_{acc}$, *is* $O(n\alpha(n, d))$.

Property 4. *The sum of maximum ranks,* $\sum_{u \in N} u.\text{RANK}_{max}$, *is* $O(n)$.

Property 5. *For some constants* $f > 1$ *and* c, *and for all* $k > 0$, $|\{u \in N | u.\text{RANK}_{max} = k\}| \leq cn/f^k$.

Note that Property 5 implies Property 4, and thus is stronger. A rank function that satisfies Properties 1, 2, 3, and 5 is *strongly well-behaved*, while one that satisfies Properties 1, 2, 3, and 4 is *weakly well-behaved*. Our proof of the Sledgehammer Theorem (2.3.2) will only require the rank function to be weakly well-behaved.

Proving the stronger property may be useful in attaining other bounds. For instance, it can be used to bound the increase in node potentials due to rank increases for non-fixed rank functions and to bound the increase in node potentials caused by multiple rounds of linking. Both of these are addressed in Section 2.4. Weakly well-behaved ranks will generally suffice for fixed rank functions used for the disjoint set union problem.

Also note that we can relax Property 3 above to hold in expectation rather than the worst case. This will simply convert the worst-case amortized bound in the Sledgehammer Theorem (2.3.2) to an expected amortized bound. This will be vital for our analyses of randomized linking in Section 3.2.3 and nested randomized linking in Section 4.2.3.

The fixed rank functions discussed above for linking by rank, linking by size, and randomized linking are all strongly well-behaved. We will formally prove that each of these methods admits a strongly well-behaved rank function in Section 3.2.

2.1.2 Well-Structured Compaction

Consider three methods of path compaction: compression, splitting, and halving. Put simply, compression replaces the parent of each node on the path with the head of the path, splitting replaces the parent of each node with its grandparent, and halving replaces the parent of every other node with its grandparent.

Let $u.\textsc{parent}^t$ denote the t^{th} ancestor of u, and let us examine what local guarantees these compaction methods make. Splitting is the easiest case to state: for all but the last two nodes on the path, $u.\textsc{parent}$ is replaced by $u.\textsc{parent}^2$. Halving does the same, but only for every other node. Compression provides a less local guarantee, but in some sense

that is all right. If we take an arbitrary constant t, then for all but the last t nodes on the path, compression replaces $u.\text{PARENT}$ by some ancestor y of $u.\text{PARENT}^t$.

In effect, the key property of these methods is that compaction identifies a sufficiently dense collection of nodes on the path and replaces the parent of each by a node at least some constant distance further up the path, or more specifically replaces the t^{th} ancestor with at least the t^{th} ancestor of its successor in the collection for some constant $t > 0$. This leads us to our general properties imposed on compaction.

Let $\langle u_1, u_2, \ldots, u_h \rangle$ be a sequence subject to compaction, and let g, q, and t be positive integer constants. Then there exists a sequence of indices $i_1 < i_2 < \ldots < i_{h'}$ such that the following properties hold in the presence of weakly well-behaved ranks.

Property 6. *For all* $1 \leq j < h'$, $i_j + g \geq i_{j+1}$, *and* $gh' + q \geq h$.

Property 7. *For all* $1 \leq j < h'$, $u_{i_j}.\text{PARENT}^t.\text{RANK} \leq u_{i_{j+1}}.\text{RANK}$.

Property 8. *For all* $1 \leq j < h'$, *the new value of* $u_{i_j}.\text{PARENT}^t.\text{RANK}$ *is at least the old value of* $u_{i_{j+1}}.\text{PARENT}^t.\text{RANK}$.

We say that a compaction method that satisfies Properties 6, 7, and 8 is *well-structured*. Compression and splitting do so with $g = t = 1$ and $q = 2$, and halving does so with $g = t = 2$ and $q = 3$. Thus all three methods are well-structured. We will prove this formally in Section 3.1.

2.2 Gathering Basic Materials

2.2.1 Ackermann's Function and its Ilk

Ackermann's function [1, 19] is defined recursively on two non-negative integer variables 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 show that A is strictly increasing in both arguments, $A(k + 1, j) \geq A(k, j + 1)$, and $A(1, j) = j + 2$. A increases very, very quickly with k: $A(2, j) > 2j$, $A(3, j) > 2^j$, and so on. For an explicit example of how large the values of the function can be, even for small input, $A(4, 2) = 2^{65536} - 3$.

Because Ackermann's function is defined on two variables, it does not necessarily have a uniquely defined inverse function. We will define two different inverses that take both a primary input and a secondary value with which to fix one input to Ackermann's function.

What we will call the *inverse Ackermann function*, α, is essentially an inverse in the first parameter of A. It is defined for any non-negative integer r and non-negative real number d by

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

Note that α is non-decreasing in the first argument and non-increasing in the second. Unless otherwise specified, we will assume $d = m/n$.

We also define an *index function*, INDEX, which is used as an inverse in the second parameter of A. It is defined for any non-negative integers k and r as

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

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

Finally, it will be useful in our analysis to define a function which in a sense measures the distance between two inputs using Ackermann's function as a scale. The *level function*, LEVEL, is defined for two non-negative integers r and s as

$$\text{LEVEL}(r, s) = \min(\{\alpha(r, d) + 1\} \cup \{k \leq \alpha(r, d) | A(k, \text{INDEX}(k, r)) > s\}).$$

Taking the minimum with $\alpha(r,d)+1$ in effect caps the value of $\textsc{Level}(r,s)$ in order to make our analysis more convenient. One key property of this definition is that the value of $\textsc{Level}$ is strictly positive if $r \leq s$.

Lemma 2.2.1. *If $r \leq s$, $\textsc{Level}(r,s) = 0$ if and only if $r = s$.*

Proof. Since $A(0,j) = j+1$, $\textsc{Index}(0,r) = r$, which implies $A(0,\textsc{Index}(0,r)) = A(0,r) = r+1$. Hence $\textsc{Level}(r,s) = 0$ if $r = s$ and $\textsc{Level}(r,s) > 0$ if $r < s$. □

2.2.2 Ackermann's Function as a Ruler

The goal now is to establish a concrete link between the functions defined and the way nodes make progress. As compaction proceeds, a node's rank remains fixed, but the rank of its t^{th} ancestor may increase due to parent changes. In this sense, we can measure a node's progress by the gap between the ranks. With that in mind, we define for each good node u a *level* $u.\textsc{level}$, an *index* $u.\textsc{index}$, and a *count* $u.\textsc{count}$ as follows:

$$u.\textsc{level} = \textsc{Level}\big(u.\textsc{rank}, u.\textsc{parent}^t.\textsc{rank}\big),$$

$$u.\textsc{index} = \textsc{Index}\big(u.\textsc{level}-1, u.\textsc{parent}^t.\textsc{rank}\big) \text{ if } u.\textsc{level} > 0,$$

$$u.\textsc{index} = 0 \text{ otherwise, and}$$

$$u.\textsc{count} = u.\textsc{level} \times (u.\textsc{rank}+2) + u.\textsc{index}.$$

The level and index act respectively as primary and secondary measures of progress, while the count is simply a linear combination of the two. The definition of the level function implies a bound of $\alpha(u.\textsc{rank},d)+1$ for $u.\textsc{level}$. We will prove below that the index remains suitably bounded; that compaction can only increase a node's level; that as long as a node's level remains fixed, compaction can only increase its index; and that whenever compaction changes either the node's level or index, its count increases.

Lemma 2.2.2. *If $u.\textsc{level} \leq \alpha(u.\textsc{rank},d)$, then $u.\textsc{index} \leq \max\{u.\textsc{rank},1\} \leq u.\textsc{rank}+1$.*

Proof. If $u.\textsc{level} = 0$, $u.\textsc{index} = 0$, so the lemma holds. Suppose $u.\textsc{level} = k > 0$. Let $j = \textsc{Index}(k, r)$. The definition of $u.\textsc{level}$ implies $A(k, j) > u.\textsc{parent}^t.\textsc{rank}$. If $j = 0$, then $A(k, 0) = A(k-1, 1) > u.\textsc{parent}^t.\textsc{rank}$. Hence

$$u.\textsc{index} = \min\{j \geq 0 | A(k-1, j) > u.\textsc{parent}^t.\textsc{rank}\} \leq 1,$$

so the lemma holds. If $j > 0$, then

$$A(k, j) = A(k-1, A(k, j-1)) > u.\textsc{parent}^t.\textsc{rank}.$$

Since $j = \textsc{Index}(k, r)$, $A(k, j-1) \leq u.\textsc{rank}$, which implies $A(k-1, u.\textsc{rank}) > u.\textsc{parent}^t.\textsc{rank}$. Hence $u.\textsc{index} \leq u.\textsc{rank}$, so the lemma holds. □

Lemma 2.2.3. *If* $u.\textsc{level} = \alpha(u.\textsc{rank}, d) + 1 = \alpha(u.\textsc{parent}^t.\textsc{rank}, d) + 1$, *then* $u.\textsc{index} \leq d$.

Proof. The definition of α implies

$$A(u.\textsc{level} - 1, d) = A\big(\alpha\big(u.\textsc{parent}^t.\textsc{rank}, d\big), d\big) > u.\textsc{parent}^t.\textsc{rank}.$$

Since $u.\textsc{index} = \textsc{Index}(u.\textsc{level} - 1, u.\textsc{parent}^t.\textsc{rank})$, $u.\textsc{index} \leq d$. □

Lemma 2.2.4. *For every node* u, $u.\textsc{level}$ *and* $u.\textsc{count}$ *never decrease due to compaction, and* $u.\textsc{count}$ *increases whenever* $u.\textsc{level}$ *or* $u.\textsc{index}$ *changes as a result of compaction. If* $u.\textsc{level}$ *increases by* k, $u.\textsc{count}$ *increases by at least* k.

Proof. For fixed r, $\textsc{Level}(r, s)$ is a non-decreasing function of s. Since $u.\textsc{rank}$ does not change and $u.\textsc{parent}^t.\textsc{rank}$ never decreases as a result of compaction, $u.\textsc{level} = \textsc{Level}(u.\textsc{rank}, u.\textsc{parent}^t.\textsc{rank})$ never decreases. Since $u.\textsc{parent}^t.\textsc{rank}$ never decreases, while $u.\textsc{level}$ is constant $u.\textsc{index}$ never decreases, so a change in $u.\textsc{index}$ while $u.\textsc{level}$ is constant increases both $u.\textsc{index}$ and $u.\textsc{count}$. When $u.\textsc{level}$ increases by k, $u.\textsc{index}$ decreases by at most $u.\textsc{rank} + 1$ by Lemma 2.2.2, so $u.\textsc{count}$ increases by at least $k(u.\textsc{rank} + 2) - (u.\textsc{rank} + 1) \geq k$. □

Finally, in order for these values to be useful, we must show that under reasonable conditions, compaction does in fact increase the node counts. This will occur under the condition that a node's successor in the chosen subsequence has the same or greater level.

Lemma 2.2.5. *Let u and v be nodes such that $u.\text{PARENT}^t.\text{RANK} \leq v.\text{RANK}$ and $0 < u.\text{LEVEL} = v.\text{LEVEL}$ just before a compaction that increases $u.\text{PARENT}^t.\text{RANK}$ to at least $v.\text{PARENT}^t.\text{RANK}$. Then the compaction increases $u.\text{COUNT}$.*

Proof. Let

$$\begin{aligned} k &= u.\text{LEVEL} = v.\text{LEVEL}, \\ j &= u.\text{INDEX} = \text{INDEX}\big(k-1, u.\text{PARENT}^t.\text{RANK}\big), \\ j' &= v.\text{INDEX} = \text{INDEX}\big(k-1, v.\text{PARENT}^t.\text{RANK}\big), \text{ and} \\ j'' &= \text{INDEX}(k-1, v.\text{RANK}) \end{aligned}$$

just before the compaction. The definition of INDEX implies that since

$$u.\text{PARENT}^t.\text{RANK} \leq v.\text{RANK} \leq v.\text{PARENT}^t.\text{RANK}, \quad j \leq j'' \leq j'.$$

The definition of $v.\text{LEVEL}$ implies

$$\begin{aligned} v.\text{RANK} < A(k-1, \text{INDEX}(k-1, j'')) &\leq y.\text{PARENT}^t.\text{RANK} \\ &< A(k-1, \text{INDEX}(k-1, j')). \end{aligned}$$

Thus $j < j'$. Since $j' = \min\{i \geq 0 | A(k-1, i) > v.\text{PARENT}^t.\text{RANK}\}$, the compaction increases $\text{INDEX}(k-1, u.\text{PARENT}^t.\text{RANK})$ to at least j', by increasing $u.\text{PARENT}^t.\text{RANK}$ to at least $v.\text{PARENT}^t.\text{RANK}$. It follows that either the compaction does not change $u.\text{LEVEL}$ but increases $u.\text{INDEX}$, or it increases $u.\text{LEVEL}$. In either case $u.\text{COUNT}$ increases by Lemma 2.2.4. □

Lemma 2.2.6. *Let u and v be nodes such that*

$$u.\text{RANK} \leq v.\text{RANK} \text{ and } u.\text{LEVEL} < \min\{v.\text{LEVEL}, \alpha(u.\text{RANK}, d) + 1\}$$

just before a compaction that increases $u.\text{PARENT}^t.\text{RANK}$ *to at least* $v.\text{PARENT}^t.\text{RANK}$*. Then the compaction increases* $u.\text{LEVEL}$ *to at least* $\min\{v.\text{LEVEL}, \alpha(u.\text{RANK}, d) + 1\}$*, and hence increases* $u.\text{COUNT}$ *by at least* $\min\{v.\text{LEVEL}, \alpha(u.\text{RANK}, d) + 1\} - u.\text{LEVEL}$*.*

Proof. Let $k = v.\text{LEVEL}$. The definition of $v.\text{LEVEL}$ implies

$$\begin{aligned} A(k-1, \text{INDEX}(k-1, u.\text{RANK})) &\leq A(k-1, \text{INDEX}(k-1, v.\text{RANK})) \\ &\leq v.\text{PARENT}^t.\text{RANK}. \end{aligned}$$

It follows that once $u.\text{PARENT}^t$ is at least $v.\text{PARENT}^t$,

$$u.\text{LEVEL} \geq \min\{v.\text{LEVEL}, \alpha(u.\text{RANK}, d) + 1\}.$$

By Lemma 2.2.4, $u.\text{COUNT}$ increases by at least as much as $u.\text{LEVEL}$. □

2.3 Crafting a Sledgehammer

For each node u, we want to define a potential $u.\text{POTENTIAL}$ such that $u.\text{POTENTIAL}$ decreases whenever $u.\text{COUNT}$ increases or $u.\text{TIES}$ decreases, and we want it to decrease by enough in each of these cases to pay for roughly g nodes in the sequence. The latter condition of course can be accompished by scaling by g, while the former can be accomplished by summing $u.\text{TIES}$ and the difference between the maximum and current value of $u.\text{COUNT}$. We define the potential of u as

$$u.\text{POTENTIAL} = 2g(u.\text{TIES} + u.\text{COUNT}_{max} - u.\text{COUNT}), \text{ where}$$

$$u.\text{COUNT}_{max} = (\alpha(u.\text{RANK}_{max}, d) + 1) \times (u.\text{RANK}_{max} + 2) + d + 1.$$

Then by the definition of a and Lemmas 2.2.2 and 2.2.3 the potential of u is always nonnegative. The potential of the forest is the sum of the node potentials. Compaction does not increase the potential of any node; however, depending on the setting, other operations interleaved with the compactions may do so. We let ϕ_+ denote the *added potential*, or total increase in potential due to these other operations.

We proceed by bounding the initial potential of the forest. Recall that the forest begins as a collection of singletons.

Lemma 2.3.1. *In the presence of weakly well-behaved ranks, the initial potential of the forest is* $O(n\alpha(n,d)+m)$ *where* $d = m/n$.

Proof. Since the maximum value of a node u's potential is

$$2g(u.\text{TIES}_{acc} + u.\text{COUNT}_{max}),$$

u can contribute at most this much to the initial potential ϕ_0. Summing over all nodes and taking into account Properties 3 and 4 we obtain

$$\begin{aligned}
\phi_0 &\leq 2g \sum_{u \in G} (u.\text{TIES}_{acc} + u.\text{COUNT}_{max}) \\
&\leq 2g \sum_{u \in G} (u.\text{TIES}_{acc} + (\alpha(u.\text{RANK}_{max}, d) + 1) \times (u.\text{RANK}_{max} + 2)) + d + 1 \\
&\leq 2g \left(\left(\sum_{u \in G} u.\text{TIES}_{acc} \right) + (\alpha(n,d)+1) \left(\sum_{u \in G} u.\text{RANK}_{max} + 2 \right) + \left(\sum_{u \in G} d + 1 \right) \right) \\
&\leq \text{O}(n\alpha(n,d)+m)
\end{aligned}$$

□

We define the amortized cost of a compaction to be the number of nodes processed plus the change in potential. Thus bounding the added potential and the amortized cost of a compaction gives us a bound on the total number of nodes processed.

Theorem 2.3.2 (**Sledgehammer Theorem**). *Using well-structured compaction in the presence of weakly well-behaved ranks, a sequence of* m *compactions processes* $O(m\alpha(n,d)+\phi_+)$ *nodes, where* $d = m/n$.

Proof. Consider any sequence subject to compaction. Well-structured compaction guarantees a dense subsequence of nodes that satisfies the properties we need to show that their

potential decreases sufficiently. We observe that $gh' \geq h - q$, and will prove that the potential of the nodes on the path drops by at least $g(h' - 1) - 2\alpha(n, d)$, thus the amortized cost of a compaction is $O(\alpha(n, d))$.

Let $\langle u_{i_1}, u_{i_2}, \ldots, u_{i'_h} \rangle$ be the subsequence of nodes specified by Property 6. Let u be any one of these nodes besides the last and let v be its successor in this subsequence. Then by Properties 7 and 8 compaction changes $u.\text{PARENT}^t.\text{RANK}$ from at most $v.\text{RANK}$ to at least $v.\text{PARENT}^t.\text{RANK}$. If $u.\text{LEVEL} < \min\{v.\text{LEVEL}, \alpha(u.\text{RANK}, d) + 1\}$, compacting the sequence decreases the potential of u by at least

$$\begin{aligned} &2g(\min\{v.\text{LEVEL}, \alpha(u.\text{RANK}, d) + 1\} - u.\text{LEVEL}) \\ &\quad \geq g(1 + \min\{v.\text{LEVEL}, \alpha(u.\text{RANK}, d) + 1\} - u.\text{LEVEL}) \end{aligned}$$

by Lemmas 2.2.2, 2.2.3, and 2.2.6. If $u.\text{LEVEL} = 0$, then

$$u.\text{RANK} = u.\text{PARENT}.\text{RANK} = \ldots = u.\text{PARENT}^t.\text{RANK} \leq v.\text{PARENT}^t.\text{RANK},$$

and compaction decreases the potential by at least g, since u loses at least one proper ancestor of the same rank. If $0 < u.\text{LEVEL} = v.\text{LEVEL}$ and $\alpha(u.\text{RANK}, d) = \alpha(v.\text{RANK}, d)$, compaction decreases the potential by at least $2g \geq g$ by Lemmas 2.2.2, 2.2.3, and 2.2.5.

We claim that in all cases, the potential of u decreases by at least

$$g(1 + v.\text{LEVEL} - u.\text{LEVEL} + 2(\alpha(u.\text{RANK}, d) - \alpha(v.\text{RANK}, d))). \tag{2.1}$$

This will hold true even in the case that u's potential does not change, which will coincide with a non-positive value for Equation 2.1 (recall that compaction does not increase the potential of any node).

Since $\alpha(u.\text{RANK}, d) \leq \alpha(v.\text{RANK}, d)$, this is true if $v.\text{LEVEL} < u.\text{LEVEL}$, because the value of Equation 2.1 is non-positive. It is also true if $v.\text{LEVEL} = u.\text{LEVEL}$, since if $\alpha(u.\text{RANK}, d) = \alpha(v.\text{RANK}, d)$, the value of Equation 2.1 is g, and if $\alpha(u.\text{RANK}, d) < \alpha(v.\text{RANK}, d)$, the value of Equation 2.1 is non-positive. If $u.\text{LEVEL} < v.\text{LEVEL} \leq \alpha(u.\text{RANK}, d) + 1$, the

drop is at least $g(1 + v.\textsc{level} - u.\textsc{level})$, which is at least the value of Equation 2.1. If $u.\textsc{level} < \alpha(u.\textsc{rank}, d) + 1 < v.\textsc{level}$, the drop is at least

$$\begin{aligned} &g(1 + \alpha(u.\textsc{rank}, d) + 1 - u.\textsc{level}) \\ &\quad \geq g(1 + v.\textsc{level} - u.\textsc{level} + \alpha(u.\textsc{rank}, d) + 1 - v.\textsc{level}). \end{aligned}$$

Since $v.\textsc{level} \leq \alpha(v.\textsc{rank}, d) + 1$, the drop is at least

$$g(1 + v.\textsc{level} - u.\textsc{level} + \alpha(u.\textsc{rank}, d) - \alpha(v.\textsc{rank}, d)),$$

which is at least the value of Equation 2.1. The last and most interesting case, which accounts for the factor of 2 in Equation 2.1, is $u.\textsc{level} = \alpha(u.\textsc{rank}, d) + 1 < v.\textsc{level}$. Since $v.\textsc{level} \leq \alpha(v.\textsc{rank}, d) + 1$,

$$\alpha(v.\textsc{rank}, d) - \alpha(u.\textsc{rank}, d) \geq \max\{1, v.\textsc{level} - u.\textsc{level}\},$$

so the value of Equation 2.1 is non-positive.

Now we sum Equation 2.1 over the chosen subsequence $\langle u_1, \ldots u_{h'-1}\rangle$. The sum telescopes to

$$\begin{aligned} &g(h' - 1 + u_{h'}.\textsc{level} - u_1.\textsc{level} + 2(\alpha(u_1.\textsc{rank}, d) - \alpha(u_{h'}.\textsc{rank}, d))) \\ \geq &g(h' - 1) + g((u_{h'}.\textsc{level} - \alpha(u_{h'}.\textsc{rank}, d) - 1) \\ &\qquad - (u_1.\textsc{level} - \alpha(u_1.\textsc{rank}, d) - 1) \\ &\qquad + \alpha(u_1.\textsc{rank}, d) - \alpha(u_{h'}.\textsc{rank}, d)) \\ \geq &g(h' - 2\alpha(u_{h'}.\textsc{rank}, d) - 2) \end{aligned}$$

It follows that the amortized cost of the compaction is at most

$$h - g(h' - 2\alpha(u_{h'}.\textsc{rank}, d) - 2) \leq 2g\alpha(u'_h.\textsc{rank}, d) + 2g + q.$$

Thus the amortized cost of a compaction is $O(\alpha(n, d))$. The total cost of compaction of m sequences is at most m times this amount, plus the initial potential which is $O(n\alpha(n, d) + m)$ by Lemma 2.3.1, plus the total potential added. The theorem holds. □

2.4 Working on the Railroad

Then the section foreman said, "Hey, hammer-swinger! I see you brought your own hammer, boy, but what all can them muscles do?"

JOHNNY CASH, *"The Legend of John Henry's Hammer"*

In order to use the Sledgehammer Theorem (2.3.2), one must of course show that the chosen compaction method is well-structured and define a weakly well-behaved rank function. Chapters 3 and 4 contain several proofs of this sort. One must demonstrate also that the added potential does not dominate the amortized cost of compaction. In order to do so, it suffices to bound the total decrease in node counts $u.\textsc{count}$ caused by auxiliary operations. There are two possible sources of potential increase—rank increases and parent changes. Since it may not be clear how to go about bounding these changes, we provide three lemmas below. These may serve as a template for other such proofs, and should be useful in their own right.

We first consider the auxiliary operation of linking, and in particular, consider *rounds* of linking. A round of linking can be thought of similarly to a tournament. Each node may participate in many links within a single round—as long as it continues to win these links. Once a node loses a link, it is no longer eligible for linking in that round. All the linking that occurs within algorithms for the disjoint set union problem can be viewed as a single round of linking. If we were to then allow non-roots to be linked, as in the case of nested set union, we need to consider multiple rounds of linking. In fact, the structure of the linking for the k-level nested set union problem can be viewed precisely as k rounds of linking.

The first of these lemmas applies to the case of a single round of linking, such that the rank of a node remains fixed after losing a link, which applies to most algorithms for the disjoint set union problem. It also provides the necessary freedom to assume that the forest

begins in a state other than a collection of singletons—a single round of linking can build an abritrary forest.

Lemma 2.4.1. *If the forest undergoes a single round of linking in the presence of weakly well-behaved ranks with no other structural changes besides compaction, and once a node loses a link its rank remains fixed until all linking is completed, then* $\phi_+ \leq 0$.

Proof. The forest begins as a collection of singletons, such that each node u's parent is itself, and thus $u.\textsc{level} = u.\textsc{index} = u.\textsc{count} = 0$. When a node wins a link, its parent does not change. Its rank may increase, but this does not change its level or index, and thus its potential does not change. Therefore, we need only consider the effects of losing a link. Once u loses a link, its rank remains fixed, so there is no need to consider rank increases. The only source of added potential is the parent change from losing a link. Since before losing a link, $u.\textsc{count} = 0$ for each node u, the count can only increase, thereby decreasing potential, so the lemma holds. □

Our second lemma covers the potential added by parent changes due to multiple rounds of linking. Our proof requires only weakly well-behaved ranks. Among other possible applications, this will give us a bound for our algorithms for nested set union.

Lemma 2.4.2. *If the forest undergoes* k *rounds of linking in the presence of weakly well-behaved ranks, the parent changes increase the total potential by at most* $O(kn\alpha(n, d) + km')$ *where* $d = m'/n$.

Proof. Winning a link does not cause the parent of a node to change. Losing a link can cause a node's parent to change, possibly to a node with lesser rank than the previous parent, and therefore may reset a node's count and increase its potential. Importantly, this can happen once per node per round of linking. In other words, each round of linking can reset the potential of the forest, albeit in a staggered fashion. Thus, k rounds of linking

can can increase the potential by at most k times the value in the proof of Lemma 2.3.1, or $O(kn\alpha(n,d) + km')$. □

Our third lemma bounds the potential increase caused by rank increases for a strongly well-behaved rank function. Note that this does not specify which auxiliary operations are used, and thus applies to any algorithm admitting strongly well-behaved ranks.

Lemma 2.4.3. *With a strongly well-behaved rank function, rank increases can increase the potential by at most $O(n\alpha(n,d) + m')$, where $d = m'/n$.*

Proof. Recall Property 5 specifies that at most n/f^k nodes have maximum rank k, and that $u.\text{COUNT} = u.\text{LEVEL} \times (u.\text{RANK} + 2) + u.\text{INDEX}$. By definition,

$$u.\text{LEVEL} \leq \alpha(u.\text{RANK}, d) + 1 \leq \alpha(n,d) + 1$$

and by Lemmas 2.2.2 and 2.2.3 $u.\text{INDEX} \leq u.\text{RANK} + d + 1$. Increasing the rank of u from j to $j+1$ can result in drop of at most

$$u.\text{LEVEL} \times (j+2) + u.\text{INDEX} \leq \alpha(n,d) \times (j+2) + j + d + 1.$$

Summing over all good nodes u and rank increases from 0 to $u.\text{RANK}_{max} - 1$, we get

$$\begin{aligned}
\phi_+ &\leq \sum_{u \in G} \sum_{j=0}^{u.\text{RANK}_{max}-1} \alpha(n,d) \times (j+2) + j + d + 1 \\
&\leq \sum_{k=0}^{\infty} \frac{n}{f^k} \sum_{j=0}^{k-1} \alpha(n,d) \times (j+2) + j + d + 1 \\
&= \sum_{k=0}^{\infty} \frac{n}{f^k} \left(\left(\sum_{j=0}^{k-1} \alpha(n,d) \times (j+2) \right) + \left(\sum_{j=0}^{k-1} j + d + 1 \right) \right) \\
&= n\alpha(n,d) \left(\sum_{k=0}^{\infty} \frac{1}{f^k} \left(\sum_{j=0}^{k-1} j + 2 \right) \right) + n \left(\sum_{k=0}^{\infty} \frac{1}{f^k} \left(\sum_{j=0}^{k-1} j + d + 1 \right) \right).
\end{aligned}$$

It is simple to verify (for instance through standard computer algebra software) that we can simplify the double summations above to obtain

$$\begin{aligned}\phi_+ &\leq n\alpha(n,d)\left(\frac{f(2f-1)}{(f-1)^3}\right) + n\left(\frac{df^2-df+f^2}{(f-1)^3}\right)\\ &\leq (f(2f-1))n\alpha(n,d) + n\big(df^2-df+f^2\big)\\ &\leq (f(2f-1))n\alpha(n,d) + m'f^2 + nf^2\\ &= \mathrm{O}(n\alpha(n,d)+m').\end{aligned}$$

□

2.5 Remarks

We have provided an inverse-Ackerman type bound that applies to all compaction methods satisfying a relatively weak property that holds for virtually all known efficient compaction methods. In proving this bound, we defined a framework to analyze various algorithms that interleave other operations with compactions. For instance, this bound offers a way to analyze the disjoint set union problem without using the union-forest approach, and also applies to the nested set union problem.

There is still quite a bit of work to be done to prove an optimal bound for compaction in unbalanced trees. To that end we propose extending the Sledgehammer Theorem (2.3.2) to incorporate an analytical partition of the node set into *good* and *bad* nodes. In effect, this is an attempt to move the partitioning technique Farrow used in his algorithms into the analysis. The hope is that simpler, uniform algorithms could then be used to tackle the path evaluation problem and other compressed tree applications on unbalanced trees.

Suppose that we have an unbalanced tree, so that no well-behaved rank function exists. If we were to allow certain nodes to be specified as bad and relax some of the rank properties in Section 2.1 to apply only to the good nodes, then we could perhaps make some progress. More specifically, suppose that we were to allow bad nodes to have greater rank than their

parents, require that the sum over good nodes of accumulated rank ties be small, rather than over all nodes, and require that the ranks of only good nodes decay exponentially. Then we could allow for controlled violation of rank monotonicity, and for many bad nodes to share the same rank and to have many bad ancestors with the same rank. These relaxed properties would apply to many sane rank functions defined over unbalanced trees.

Moreover, once a node previously designated as bad satisfied certain properties, like having an ancestor with strictly greater rank, we could allow it to be transferred into the good set in the partition.

In this way, an extended Sledgehammer Theorem would apply to the behavior of the compaction method on the good nodes over the sequence of compactions. The behavior with respect to the bad nodes would need to be analyzed separately.

If such a relaxation were made, it would remain to adapt the statement and proof of the Sledgehammer Theorem (2.3.2) to the changes. One obvious approach would be to bound the number of good nodes processed in terms of the number of bad nodes processed. The most straightforward way to handle this would be to consider contiguous subsequences of good nodes, the number of which is bounded by the number of compactions plus the number of bad nodes processed. This may leave an extra inverse-Ackermann term in the bound, so a more careful adaptation may be required.

There are still many questions regarding such an extension, a few of which follow.

First, and most importantly, is this the right extension? We are not certain whether this captures the difference in the behavior of compaction methods between balanced and unbalanced trees.

Second, do we need a stronger notion of what it means for a compaction method to be well-structured with regard to unbalanced trees? Avoiding an extra inverse-Ackermann term may require that the compaction method convert the path to a forest of constant or inverse-Ackermann depth.

Last, is there a good way to handle the analysis of compaction on the bad nodes? We do not know if there are some underlying properties of compaction that will lead to good behavior on bad nodes as there is for the good nodes. It may be that any efficient algorithm will be in some way unique and need a great deal of individual attention.

Despite these remaining questions and noted apparent shortcomings, there is good news—we have a powerful tool with which to succinctly analyze many algorithms.

Chapter 3

Disjoint Set Union

Why slap them on the wrist with a feather when you can belt them over the head with a sledgehammer?

Katharine Hepburn

As detailed in Chapter 1, disjoint set union is the quintessential compressed tree application. The literature includes several methods for path compaction as well as several algorithms for linking. These can be used in dozens of combinations, and many papers have been written studying these combinations both theoretically and practically.

The analysis in this chapter is modular. Compaction methods and linking rules can be analyzed separately with the use of our tools from Chapter 2. To drive home this point, we present the following corollary, which follows directly from the Sledgehammer Theorem (2.3.2) and Lemmas 2.4.1 and 2.4.3.

Corollary 3.0.1. *If* Find *operations are performed with a well-structured compaction method and* Unite *operations are performed with a linking rule that admits a strongly well-behaved rank function, a sequence of m intermixed operations takes $O(m\alpha(n, m/n))$ time.*

Once a given compaction method is proved to be well-structured, it follows that it performs optimally in conjunction with a linking rule that admits a strongly well-behaved rank function. Similarly, as soon as a strongly well-behaved rank function is proved for a linking rule, it follows that it provides an optimal algorithm when used in conjunction with any well-structured compaction method.

We will proceed to show that compression, splitting, and halving are well-structured, and that linking by rank, early linking by rank, linking by size, randomized linking, and early randomized linking all admit strongly well-behaved rank functions. Additionally, we will prove that splicing performs optimally when used with a linking rule that admits a strongly well-behaved rank function (though we do not know whether splicing is well-structured). One interesting point about this type of proof is compatibility: if two compaction methods satisfy Properties 6, 7, and 8 for the same choice of constants g, q, and t, then they can be used interchangeably for any given FIND. In proving that each method is well-structured, we will choose the most natural constants for the individual method; however, it is entirely possible to show that a fixed choice of constants works for all of them simultaneously. For example, compression, splitting, and halving all work with $g = t = 2$ and $q = 4$.

3.1 Compaction Methods

3.1.1 Compression

Compression replaces the parent of each node on the FIND path with the head of the path. It requires two passes over the path—one to find the head of the path and one to set the new parent of each node. See Figure 3.1 for an example and Algorithms 3.1.1 and 3.1.2 for recursive and iterative implementations of FIND with compression respectively.

In order to show that compression is well-structured, we need to prove that there exist constants g, q and t such that Properties 6, 7, and 8 hold. Since each node on the FIND

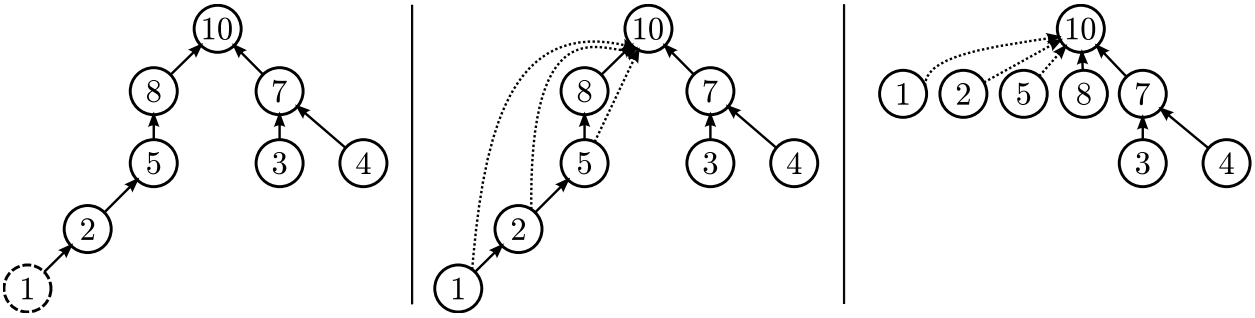

Figure 3.1: The effects of path compression beginning at the node labeled 1.

Algorithm 3.1.1 Recursive FIND with Compression

```
1: function FIND(x)
2:     if x.PARENT^2 ≠ x.PARENT then
3:         x.PARENT ← FIND(x.PARENT)
4:     return x.PARENT
```

Algorithm 3.1.2 Iterative FIND with Compression

```
1: function FIND(x)
2:     u ← x.PARENT
3:     v ← u
4:     while u.PARENT ≠ u do
5:         u ← u.PARENT
6:     while v ≠ u do
7:         x.PARENT ← u
8:         x ← v
9:         v ← x.PARENT
10:    return u
```

path has its parent set to the head of the path, that means each node except the last 2 will have its parent changed to a node with rank at least that of its grandparent, so we can chose $t = g = 1$ and $q = 2$.

Lemma 3.1.1. *For $t = g = 1$ and $q = 2$, compression satisfies Properties 6, 7, and 8 in the presence of weakly well-behaved ranks. Thus compression is well-structured.*

Proof. Given an input sequence $\langle u_1, u_2, \ldots, u_h \rangle$, we select as our chosen subsequence $\langle u_1, u_2, \ldots, u_{h-2} \rangle$. Then $h' = h - 2$, implying that $gh' + q = 1(h-2) + 2 = h$ and thus that Property 6 is satisfied. Since for each $1 \leq j < h'$ the successor of u_{i_j} in the sequence is its parent and given our choice of $t = 1$, it is clear that $u_{i_j}.\text{PARENT}^1.\text{RANK} = u_{i_{j+1}}.\text{RANK}$, and thus Property 7 is satisfied. Finally, compression changes the parent of each of the nodes in the chosen subsequence to be the head of the path, so for $1 \leq j < h'$, $u_{i_j}.\text{PARENT}^1.\text{RANK}$ gets set to $u_h.\text{RANK}$. Property 2 implies that with weakly well-behaved ranks, this value must be at least $u_{i_j}.\text{PARENT}^2.\text{RANK} = u_{i_{j+1}}.\text{PARENT}^1.\text{RANK}$, and thus Property 8 holds. Therefore, the lemma holds. □

3.1.2 Splitting

Splitting replaces the parent of each node on the FIND path with its grandparent. This method requires only one pass over the path. See Figure 3.2 for an example and Algorithm 3.1.3 for an iterative implementation.

Algorithm 3.1.3 FIND with Splitting

```
function FIND(x)
    u ← x.PARENT
    while u.PARENT ≠ u do
        x.PARENT ← u.PARENT
        x ← u
        u ← u.PARENT
    return u
```

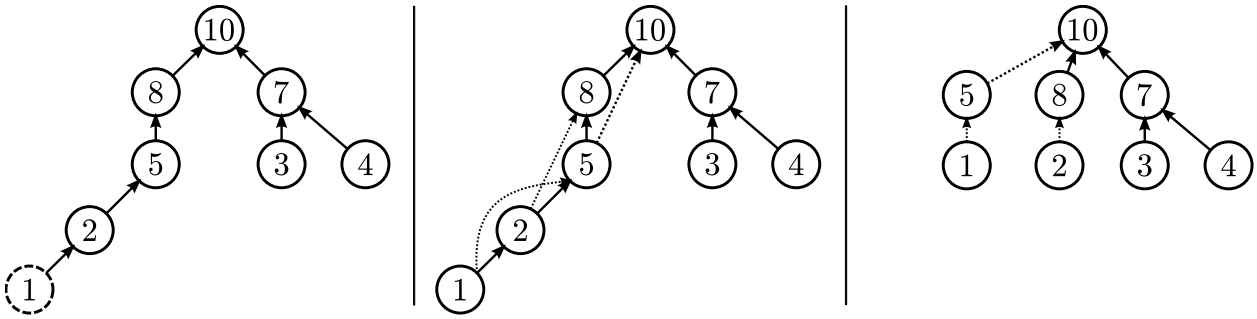

Figure 3.2: The effects of path splitting beginning at the node labeled 1.

The proof that splitting is well-structured is remarkably similar to that for compression. Since each node on the FIND path has its parent changed to its grandparent, that means each node except the last 2 will have its parent changed, so we can chose $t = g = 1$ and $q = 2$, just as for compression. The proof is almost identical.

Lemma 3.1.2. *For $t = g = 1$ and $q = 2$, splitting satisfies Properties 6, 7, and 8 in the presence of weakly well-behaved ranks. Thus splitting is well-structured.*

Proof. Given an input sequence $\langle u_1, u_2, \ldots, u_h \rangle$, we select as our chosen subsequence $\langle u_1, u_2, \ldots, u_{h-2} \rangle$. Then $h' = h - 2$, implying that $gh' + q = 1(h-2) + 2 = h$ and thus that Property 6 is satisfied. Since for each $1 \leq j < h'$ the successor of u_{i_j} in the sequence is its parent and given our choice of $t = 1$, it is clear that $u_{i_j}.\text{PARENT}^1.\text{RANK} = u_{i_{j+1}}.\text{RANK}$, and thus Property 7 is satisfied. Finally, splitting changes the parent of each of the nodes in the chosen subsequence to be its grandparent, so for $1 \leq j < h'$, $u_{i_j}.\text{PARENT}^1.\text{RANK}$ gets set to $u_{i_j}.\text{PARENT}^2.\text{RANK} = u_{i_{j+1}}.\text{PARENT}^1.\text{RANK}$, and thus Property 8 holds. Therefore, the lemma holds. □

3.1.3 Halving

Halving replaces the parent of every other node on the FIND path with its grandparent. This method requires only one pass over the path. See Figure 3.2 for an example and

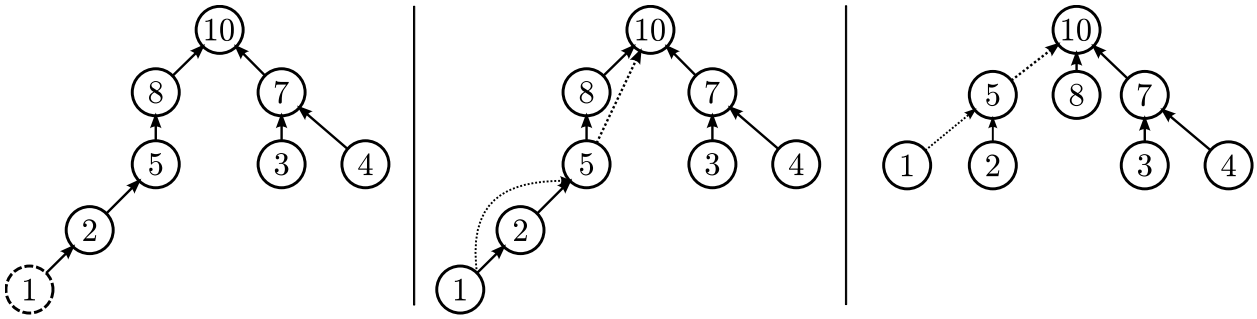

Figure 3.3: The effects of path halving beginning at the node labeled 1.

Algorithm 3.1.4 for an iterative implementation.

Algorithm 3.1.4 FIND with Halving

```
1: function FIND(x)
2:     while x.PARENT^2 ≠ x.PARENT do
3:         x.PARENT ← x.PARENT^2
4:         x ← x.PARENT
5:     return x.PARENT
```

Note that halving is very similar to splitting algorithmically. Unfortunately, since only the parent of every other node on the path changes, it is a bit harder to see how it is well-structured. Luckily, we can exploit the uniform structure of these changes. Rather than looking at parent intervals, we consider grandparent intervals. With the exception of a few nodes near the head of the path, each node u that has its parent changed as a result of halving will also have its grandparent changed to be its grandparent's grandparent—$u.\text{PARENT}^2$ will be changed to $u.\text{PARENT}^4$.

Now, to examine exactly which nodes along the path have their parents changed, we will need to consider the parity of h. If h is even, then the last three nodes will not change parents; if h is odd, then the last two nodes will not change parents. Since $(h \mod 2)$ is 0 if h is even and 1 if h is odd, the subsequence of nodes whose parents change can be written as $\langle u_1, u_3, u_5, \ldots, u_{h_0} \rangle$ where $h_0 = h - 3 + (h \mod 2)$.

Lemma 3.1.3. *For $t = g = 2$ and $q = 3$, halving satisfies Properties 6, 7, and 8 in the presence of weakly well-behaved ranks. Thus halving is well-structured.*

Proof. Given an input sequence $\langle u_1, u_2, \ldots, u_h \rangle$, we select as our chosen subsequence $\langle u_1, u_3, u_5, \ldots, u_{h_0} \rangle$ where $h_0 = h-3+(h \mod 2)$. Note that $h_0 \geq h-3$, and thus $2h' \geq h-3$. This implies that $gh' + q \geq h - 3 + 3 = h$ and thus that Property 6 is satisfied. Since for each $1 \leq j < h'$ the successor of u_{i_j} in the sequence is its grandparent and given our choice of $t = 2$, it is clear that $u_{i_j}.\text{PARENT}^2.\text{RANK} = u_{i_{j+1}}.\text{RANK}$, and thus Property 7 is satisfied. Finally, halving changes the parent of each of the nodes in the chosen subsequence to be its grandparent. Then for all but the last node in the chosen subsequence, the new grandparent is the old grandparent's grandparent. More precisely, for $1 \leq j < h'$, $u_{i_j}.\text{PARENT}^2.\text{RANK}$ gets set to $u_{i_j}.\text{PARENT}^4.\text{RANK} = u_{i_{j+1}}.\text{PARENT}^2.\text{RANK}$, and thus Property 8 holds. Therefore, the lemma holds. □

3.1.4 Splicing

Splicing is unlike the other compaction methods in that the local guarantees it makes about ancestor changes are much less rigid. It is not clear whether splicing is well-structured. Our analysis in this section makes use of many of the tools developed in Chapter 2, but does not apparently lend itself to the direct use of the Sledgehammer Theorem (2.3.2). Instead, we will show that if UNITE operations are performed with splicing and a linking rule that admits a strongly well-behaved rank function, then the amortized time per UNITE is $O(\alpha(n, d))$.

Splicing is used on pairs of FIND paths within UNITE operations. Both paths are traversed simultaneously. Given the current node on each path, the one with the greater parent is determined using the ordering on the nodes; in case of a tie in the ordering, one of the parents is selected arbitrarily. The other node, say u, is made to point to this parent, and the pointer on u's path advances to u's old parent. This process continues until both of the current nodes share a parent. See Figure 3.4 for a demonstration and Algorithm 3.1.5 for an

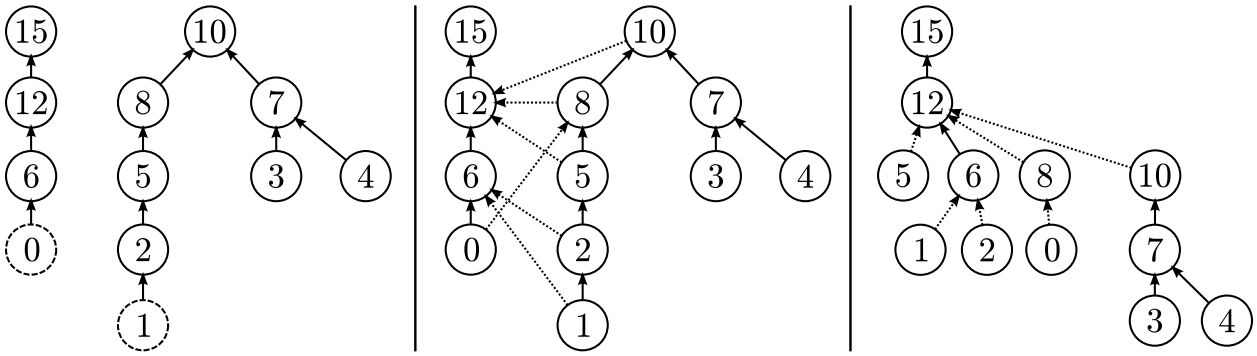

Figure 3.4: The effects of splicing by index beginning at the nodes labeled 0 and 1. The splice sequence is $\langle 0, 1, 2, 5, 6, 8, 10, 12 \rangle$.

implementation of early linking by index with splicing.

Algorithm 3.1.5 UNITE with Early Linking by Index and Splicing

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

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 Chapter 2 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 visited by the splice. 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 white and those on the path from v black; w gets both colors.

Lemma 3.1.4. *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 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 before the splice. □

Corollary 3.1.5. *If x is a node on a spliced pair of paths, x.*TIES *does not increase as the result of a splice. If x and its grandparent have the same rank and x changes grandparent, x.*TIES *strictly decreases.*

Proof. If the corollary holds for the old grandparent of x, then it holds for x by Lemma 3.1.4. The corollary follows by induction on the depth of x 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 3.1.6. *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. (See Figure 3.5.) 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

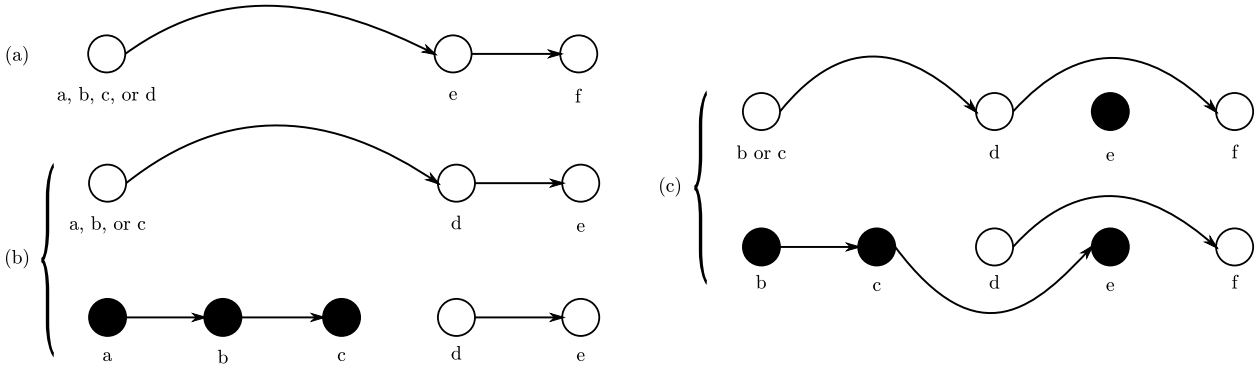

Figure 3.5: Cases in the proof of Lemma 3.1.6. Nodes a, b, c, d, e, and f are consecutive nodes in the splice sequence. Arrows denote parent pointers.

(a) Nodes e and f are the same color, say white, and one of a, b, c, d is also white. The last white node among a, b, c, d changes grandparent.

(b) Nodes d and e are the same color, say white. If a, b, or c is white, the last white one among them changes grandparent. If a, b, and c are all black, a changes grandparent.

(c) Nodes d and f are the same color, say white, and e is the opposite color. If b or c is white, the last one changes grandparent. If b and c are black, b changes grandparent.

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 3.1.7. *If* $r \leq s \leq t$, $\max\{\textsc{Level}(r,s), \textsc{Level}(s,t)\} \geq \textsc{Level}(r,t)$.

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

Our analysis of splicing applies to any compatible linking rule that admits strongly well-behaved ranks. We will use $x.\textsc{level}$, $x.\textsc{index}$, $x.\textsc{count}$, and $x.\textsc{count}_{max}$ as defined in Chapter 2 for $t = 1$, while $x.\textsc{level}'$, $x.\textsc{index}'$, $x.\textsc{count}'$, and $x.\textsc{count}'_{max}$ will be defined in the same manner for $t = 2$. Let the potential of a node x be redefined as

$$x.\textsc{potential} = 5x.\textsc{ties} + 10(x.\textsc{count}_{max} - x.\textsc{count}) + 10(x.\textsc{count}'_{max} - x.\textsc{count}').$$

Due to the similarity of the potential definition in Chapter 2, Lemma 2.4.3 still holds and covers the potential increases due to linking.

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. Note that this potential function is largely compatible with that defined in Chapter 2, and that most of the lemmas proved about the original potential function still hold. Importantly, Lemmas 2.4.1 and 2.4.3 tell us that the initial and added potential is $O(n\alpha(n,d)+m)$. By Lemma 3.1.4 and Corollary 3.1.5 a splice cannot increase the potential of any node.

Lemma 3.1.8. *If* UNITE *operations are performed with splicing and a linking rule which admits strongly well-behaved ranks, the amortized cost of a* UNITE *is* $O(\alpha(n,d))$.

Proof. Consider a splice of two paths, from u and from v to the final node w. Let its splice sequence be $x_1,\ldots,x_h = w$. Let the *pseudo-level* $x_i.\text{LEVEL}''$ of $x_i \neq w$ be $x_i.\text{LEVEL}'' = \text{LEVEL}(x_i.\text{RANK}, x_{i+1}.\text{RANK})$. Since $x_{i+1} \leq x_i.\text{PARENT}$, $x_i.\text{LEVEL}'' \leq x_i.\text{LEVEL} \leq x_i.\text{LEVEL}'$.

Our plan is to use the idea in the proof of the Sledgehammer Theorem (2.3.2): 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. We identify a sufficiently dense subset of nodes whose potential drops 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. (See Figure 3.6.) 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,

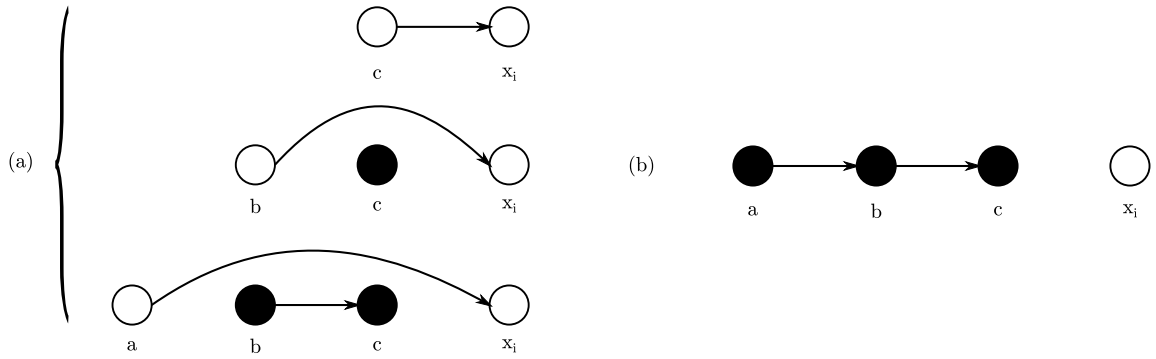

Figure 3.6: Cases in the proof of Lemma 3.1.8. Nodes a, b, c, and x_i are consecutive nodes in the splice sequence.

(a) One of a, b, c is white. Node y is the last of these that is white, namely c in (a.1), b in (a.2), and a in (a.3). Node x is c in (a.1), the node in $\{b, c\}$ of maximum pseudo-level in (a.2), and the node in $\{a, b, c\}$ of maximum pseudo-level in (a.3).

(b) All of a, b, c are black. Node y is a, node x is the node in $\{a, b\}$ of maximum pseudo-level.

and let x be the node of maximum pseudo-level in $\{y, y.\textsc{parent}\}$. (Node $y.\textsc{parent}$ is the node immediately after y in the splice sequence.) Then $y \le x \le y.\textsc{parent} \le x_i$. 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 $y.\textsc{parent} = x_i$. The splice increases $y.\textsc{parent}$ to at least x_{i+1}. If $y.\textsc{level} = x_i.\textsc{level}''$ and $\alpha(y.\textsc{rank}, d) = \alpha(x_i.\textsc{rank}, d)$, the splice decreases the parent potential of y by at least 10 $\ge$ 5 by Lemmas 2.2.2, 2.2.3, and 2.2.5. If $y.\textsc{level} < \min\{x_i.\textsc{level}'', \alpha(y.\textsc{rank}, d) + 1\}$, the splice decreases the parent potential of y by at least

$$10(\min\{x_i.\textsc{level}'', \alpha(y.\textsc{rank}, d) + 1\} - y.\textsc{level})$$

$$\ge 5 + 5(\min\{x_i.\textsc{level}'', \alpha(y.\textsc{rank}, d) + 1\} - y.\textsc{level})$$

by Lemmas 2.2.2, 2.2.3, and 2.2.6. An argument like the one in the proof of the Sledgehammer Theorem (2.3.2) shows that the potential drop in all cases is at least

$$5 + 5(x_i.\textsc{level}'' - y.\textsc{level}) + 10(\alpha(y.\textsc{rank}, d) - \alpha(x_i.\textsc{rank}, d)).$$

By Lemma 3.1.7, $y.\textsc{level} \leq x.\textsc{level}''$, so the drop in the parent potential of x is at least

$$\begin{aligned}
&5 + 5(x_i.\textsc{level}'' - x.\textsc{level}'') + 10(\alpha(y.\textsc{rank}, d) - \alpha(x_i.\textsc{rank}, d)) \\
&\quad = 5 + 5(x_i.\textsc{level}'' - x.\textsc{level}'') + 10(\alpha(x.\textsc{rank}, d) - \alpha(x_i.\textsc{rank}, d)) \\
&\qquad + 10(\alpha(y.\textsc{rank}, d) - \alpha(x.\textsc{rank}, d)).
\end{aligned}$$

The situation is similar if y is black, but the drop is in the grandparent potential of y. In this case the splice increases $y.\textsc{parent}^2$ from less than x_i to at least x_{i+1}. If $y.\textsc{level}' = x_i.\textsc{level}''$ and $\alpha(y.\textsc{rank}, d) = \alpha(x_i.\textsc{rank}, d)$, the splice decreases the grandparent potential of y by at least $10 \geq 5$ by Lemmas 2.2.2, 2.2.3, and 2.2.5, since then $\alpha(y.\textsc{rank}, d) = \alpha(y.\textsc{parent}^2.\textsc{rank}, d)$. If $y.\textsc{level}' < \min\{x_i.\textsc{level}'', \alpha(y.\textsc{rank}, d) + 1\}$, the splice decreases the grandparent potential of y by at least

$$\begin{aligned}
&10(\min\{x_i.\textsc{level}'', \alpha(y.\textsc{rank}, d) + 1\} - y.\textsc{level}') \\
&\quad \geq 5 + 5(\min\{x_i.\textsc{level}'', \alpha(y.\textsc{rank}, d) + 1\} - y.\textsc{level}')
\end{aligned}$$

by Lemmas 2.2.2, 2.2.3, and 2.2.6. By Lemma 3.1.7, $y.\textsc{level}' \leq x.\textsc{level}''$. By the argument in the previous paragraph, in all cases the drop in the grandparent potential of y is at least

$$\begin{aligned}
5 + 5(x_i.\textsc{level}'' - x.\textsc{level}'') + 10(\alpha(x.\textsc{rank}, d) - \alpha(x_i.\textsc{rank}, d)) \\
+ 10(\alpha(y.\textsc{rank}, d) - \alpha(x.\textsc{rank}, d)).
\end{aligned}$$

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 3.1.6; by Corollary 3.1.5, 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

$$5 + 5(x_i.\text{LEVEL}'' - x.\text{LEVEL}'') + 10(\alpha(y.\text{RANK}, d) - \alpha(x_i.\text{RANK}, d)),$$

since this value is non-positive if $x.\text{LEVEL}'' > 0$.

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 $O(\alpha(w.\text{RANK}, d))$. It follows that the amortized cost of the splice is $O(\alpha(w.\text{RANK}, d))$.

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

$$\begin{aligned} &5 + 5(z_{i+1}.\text{LEVEL}'' - z_i.\text{LEVEL}'') \\ &\quad + 10(\alpha(z_i.\text{RANK}, d) - \alpha(z_{i+1}.\text{RANK}, d)) \\ &\quad + 10(\alpha(y_i.\text{RANK}, d) - \alpha(z_i.\text{RANK}, d)) \end{aligned} \tag{3.1}$$

If we sum (3.1) over all $i \leq j$, the first term sums to at least $h - 10$ and the second and third terms telescope to $5(z_{j+1}.\text{LEVEL}'' - z_1.\text{LEVEL}'') + 10(\alpha(z_1.\text{RANK}, d) - \alpha(z_{j+1}.\text{RANK}, d))$. An argument like that in the proof of the Sledgehammer Theorem (2.3.2) shows that

$$\begin{aligned} &5(z_{j+1}.\text{LEVEL}'' - z_1.\text{LEVEL}'') + 10(\alpha(z_1.\text{RANK}, d) - \alpha(z_{j+1}.\text{RANK}, d)) \\ &\quad \geq -10\alpha(w.\text{RANK}, d) - 5. \end{aligned}$$

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 $\alpha(z_{i-1}.\text{RANK}, d) \leq \alpha(y_i.\text{RANK}, d)$, which further implies that the sum of the last term over all indices is at least $-10\alpha(w.\text{RANK}, d)$. We conclude that the amortized cost of the splice is $O(\alpha(w.\text{RANK}, d))$. □

3.2 Linking Rules

3.2.1 Linking by Rank

Linking by rank maintains the rank $u.\text{RANK}$ of a node, initially 0, that serves as an upper bound on its height in the forest. When linking two roots, the winner is the node with greater rank; in case of a tie, the winner is chosen arbitrarily and its rank incremented.

Not counting the effects of compaction, this guarantees that the height of the forest does not increase unnecessarily. The related method of early linking by rank terminates the internal FIND operations when one of them locates a root, linking this root underneath the current node in the other tree. In effect, this limits the amount of work done if one tree is considerably larger than the other. As a side effect, the implementation of these internal FIND operations must be integrated into the UNITE implementation. See Algorithm 3.2.1 for an implementation of linking by rank and Algorithm 3.2.2 for an implementation of early linking by rank with splitting. Implementations of early linking by rank with compression or halving are similar, but slightly more complicated, while early linking by rank with splicing is similar to Algorithm 3.1.5.

Algorithm 3.2.1 UNITE with Linking by Rank

```
function UNITE(x, y)
    u ← FIND(x)
    v ← FIND(y)
    if u = v then
        return false
    if u.RANK < v.RANK then
        u ↔ v
    v.PARENT ← u
    if u.RANK = v.RANK then
        u.RANK ← u.RANK + 1
    return true
```

While not historically the first good linking rule, linking by rank is arguably the simplest deterministic method that achieves the optimal bound. Its direct use of ranks in the algorithm makes it easy to translate into the analysis and requires only $O(\lg \lg n)$ bits of storage per node, compared to $O(\lg n)$ bits for linking by size.

In order to prove that linking by rank and early linking by rank admit a strongly well-behaved rank function, we first need a candidate rank function. In this case, it is simple enough to use the ranks maintained by the algorithms themselves. We will then need to prove Properties 1, 2, 3, and 5. The first three are simple observations of the algorithms,

Algorithm 3.2.2 UNITE with Early Linking by Rank and Splitting

```
function UNITE(x, y)
    u ← x
    v ← y
    w ← u
    z ← v
    while u.PARENT ≠ v.PARENT do
        if u.PARENT.RANK > v.PARENT.RANK then
            u ↔ v
            w ↔ z
        if u ≠ u.PARENT then
            w.PARENT ← u.PARENT
            w ← u
            u ← u.PARENT
        else
            if u.RANK = v.RANK then
                if v = v.PARENT then
                    v.RANK ← v.RANK + 1
                else
                    v ← v.PARENT
            u.PARENT ← v
            z.PARENT ← v
            return true
    return false
```

while the fourth can be proved using straightforward induction.

Lemma 3.2.1. *The ranks maintained by linking by rank and early linking by rank satisfy Properties 1, 2, 3, and 5. Thus it is a strongly well-behaved rank function.*

Proof. The only changes that linking by rank and early linking by rank make to a node's rank are increments, thus satisfying Property 1. Whenever a rank tie is introduced by linking, the winner's rank is increased, preserving strictly increasing ranks along paths in the forest. Since each compacted sequence is a path, Property 2 is satisfied. Similarly, there are no rank ties which remain after linking is completed, satisfying Property 3.

To prove that at most $n/2^k$ nodes u have $u.\text{RANK}_{max} = k$, we will use an inductive argument. For the base case of $k = 0$, it is true of course that at most $n/2^0 = n$ nodes have maximum rank 0. Now suppose that the statement holds for k. In order to produce a node with rank $k + 1$, the algorithms must link two roots of rank k. This process results in one root of rank $k + 1$ and one non-root of rank k, thereby eliminating two nodes from the pool of candidates for such links. Thus this can happen at most $n/2 \cdot 2^k = n/2^{k+1}$ times, and the statement holds for $k + 1$. Thus, the rank function satisfies Property 5 as well. The lemma holds. □

Now that we have proof of a strongly well-behaved rank function for linking by rank and early linking by rank, we can state two theorems of optimality, the truth of which follows directly from Corollary 3.0.1 and Lemmas 3.1.1, 3.1.2, 3.1.3, 3.1.8, and 3.2.1.

Theorem 3.2.2. *If* FIND *operations are performed with any combination of compression, splitting, or halving, and* UNITE *operations are performed with linking by rank, then a sequence of m intermixed operations takes $O(m\alpha(n, m/n))$ time.*

Theorem 3.2.3. *If* FIND *operations are performed with any combination of compression, splitting, or halving, and* UNITE *operations are performed with early linking by rank with*

compression, splitting, halving, or splicing, then a sequence of m intermixed operations takes $O(m\alpha(n, m/n))$ time.

3.2.2 Linking by Size

Linking by size maintains the size of each set. This is stored in the root u of each set using $\mathrm{O}(\lg n)$ bits and denoted $u.\textsc{size}$, initially equal to 1. When linking two roots, the winner is the one with more elements in its set; in case of a tie, the winner is chosen arbitrarily. The size of the winner is updated by adding the size of the loser. This maintains an important balance condition: each node u's subtree contains at most half the nodes in its parents subtree, or, equivalently, $u.\textsc{parent.size} \geq 2u.\textsc{size}$. See Algorithm 3.2.3 for an implementation.

Algorithm 3.2.3 UNITE with Linking by Size

```
function UNITE(x, y)
    u ← FIND(x)
    v ← FIND(y)
    if u = v then
        return false
    if u.SIZE < v.SIZE then
        u ↔ v
    v.PARENT ← u
    u.SIZE ← u.SIZE + v.SIZE
    return true
```

In order to prove that linking by size admits a strongly well-behaved rank function, we first need a candidate rank function. We can exploit the balance condition: let $u.\textsc{rank} = \lfloor \lg u.\textsc{size} \rfloor$. Using this definition we will be able to prove all the necessary properties.

Lemma 3.2.4. *The ranks defined for linking by size satisfy Properties 1, 2, 3, and 5. Thus it is a strongly well-behaved rank function.*

Proof. The only changes that linking by size makes to a node's rank are through increasing the size field. Since $\lfloor \lg u.\textsc{size} \rfloor$ is a monotonically increasing function, this satisfies Prop-

erty 1. The balance property $u.\text{PARENT.SIZE} \geq 2u.\text{SIZE}$ implies that

$$\begin{aligned} u.\text{PARENT.RANK} &= \lfloor \lg u.\text{PARENT.SIZE} \rfloor \\ &\geq \lfloor \lg(2u.\text{SIZE}) \rfloor = \lfloor \lg u.\text{SIZE} + 1 \rfloor > \lfloor \lg u.\text{SIZE} \rfloor = u.\text{RANK}. \end{aligned}$$

This implies strictly increasing ranks along paths. Since each compacted sequence is a path, Property 2 is satisfied. Similarly, there are no rank ties which remain after linking is completed, satisfying Property 3.

The algorithm ensures that for a root u, the number of nodes in u's tree is exactly $u.\text{SIZE}$. Note that since $u.\text{SIZE}$ does not change after u loses a link, its rank can only change while it is a root. Furthermore, $u.\text{RANK} = k$ implies that $u.\text{SIZE} \geq 2^k$. Then for a node u to have $u.\text{RANK}_{max} = k$, its tree must have contained at least 2^k nodes and strictly less than 2^{k+1} nodes when it lost a link. Let the set of nodes in u's tree prior to losing the link be denoted S_u. Then S_u must be disjoint from S_v for another node v such that $v.\text{RANK}_{max} = k$. For this this not to be the case, u must be contained in S_v, implying that v at one point won a link with one of u's ancestors, adding all nodes in S_u to S_v. This implies that $v.\text{SIZE} \geq 2u.\text{SIZE}$ prior to winning this link, meaning $v.\text{RANK}_{max} \geq k+1$. Thus for all nodes u with $u.\text{RANK}_{max} = k$, the sets S_u are disjoint, so at most $n/2^k$ nodes can have maximum rank k. Therefore the rank function satisfies Property 5 as well, and the lemma holds. □

Now that we have proof of a well-behaved rank function for linking by size, we can state a theorem of optimality, the truth of which follows directly from Corollary 3.0.1 and Lemmas 3.1.1, 3.1.2, 3.1.3, and 3.2.4.

Theorem 3.2.5. *If* FIND *operations are performed with any combination of compression, splitting, or halving, and* UNITE *operations are performed with linking by size, then a sequence of m intermixed operations takes $O(m\alpha(n, m/n))$ time.*

3.2.3 Randomized Linking

Randomized linking consists of labeling the nodes with a uniformly random permutation of the numbers from 1 to n and then performing linking by index on the labeling. Linking by index simply chooses as the winner the root with greater label. Randomized early linking is the same, except that it uses early linking by index instead of linking by index. Early linking by index is analagous to early linking by rank. See Algorithm 3.2.4 for an implementation of linking by index and Algorithm 3.1.5 for an implementation of early linking by index with splicing.

Algorithm 3.2.4 UNITE with Linking by Index

```
function UNITE(x, y)
    u ← FIND(x)
    v ← FIND(y)
    if u = v then
        return false
    if u < v then
        u ↔ v
    v.PARENT ← u
    return true
```

For each node u labeled from 1 to n, let $u.\text{RANK} = \lfloor \lg n \rfloor - \lfloor \lg(n - u + 1) \rfloor$. 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. Among other implications, this rank function implies the following properties.

Property 9. *If $u < v$, then $u.\text{RANK} \leq v.\text{RANK}$.*

Property 10. *For any element u, at least half the elements greater than u have rank strictly greater than $u.\text{RANK}$.*

Property 11. *The number of elements of rank k is at most $n/2^k$.*

Most of the properties we need to demonstrate about this rank function to show it is strongly well-behaved in expectation are immediate from the definition—the exception being

that the number of rank ties is limited. Nodes along a path have strictly increasing labels, but we can have rank ties. We will show that the expected number of proper ancestors of each node u with rank $u.\text{RANK}$ is bounded by a small constant with high probability. This is a tricky argument to make directly, so we will use a resequencing technique to make it easier.

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

The sequence $\text{SEQ}(u)$ is SEQ reordered so that each successive UNITE adds a new element to the set containing u, with ties broken by order in SEQ. We develop a recursive characterization of $\text{SEQ}(u)$ below. Let $u_0 = u, u_1, u_2, \ldots$ be the successive vertices added by $\text{SEQ}(u)$ to the set containing u. We call u_j a *prefix maximum* if $u_i < u_j$ for all $i < j$.

Note that all the following proofs study the effect of linking without compaction. Compaction can only help us by removing ancestors, so this is a safe assumption to make. All the lemmas about the reordered sequence apply deterministically to linking by index. The random permutation does not matter until we consider the expected number of prefix maxima

in Lemma 3.2.10.

Lemma 3.2.6. *If* $\textsc{Seq}(u)$ *is performed using linking by index or early linking by index with no compaction, the ancestors of* u *are exactly the prefix maxima among the* u_j*.*

Proof. For any j, the root of the tree built by the first j $\textsc{Unite}$ operations in $\textsc{Seq}(u)$ is the maximum u_i such that $i \leq j$, which is a prefix maximum. Conversely, if u_j is a prefix maximum, it will be the root of the tree built by the first j $\textsc{Unite}$ operations in $\textsc{Seq}(u)$. □

Let $\textsc{Unite}(v, w)$ be the last $\textsc{Unite}$ in $\textsc{Seq}$, and let $\textsc{Seq}'$ be $\textsc{Seq}$ with this $\textsc{Unite}$ deleted. Let S_1 containing v and S_2 containing w be the two sets built by $\textsc{Seq}'$, and let $\textsc{Seq}_1$ and $\textsc{Seq}_2$ be the subsequences of $\textsc{Seq}'$ that build S_1 and S_2, respectively.

Lemma 3.2.7. *If* u *is in* S_1*,* $\textsc{Seq}(u)$ *is* $\textsc{Seq}_1(u)$ *followed by* $\textsc{Unite}(v, w)$ *followed by* $\textsc{Seq}_2(w)$*; if* u *is in* S_2*,* $\textsc{Seq}(u)$ *is* $\textsc{Seq}_2(u)$ *followed by* $\textsc{Unite}(v, w)$ *followed by* $\textsc{Seq}_1(v)$*.*

Proof. Assume u is in S_1; the argument is symmetric if u is in S_2. The $\textsc{Unite}$ operations in $\textsc{Seq}_1$ have both inputs in S_1, and the $\textsc{Unite}$ operations in $\textsc{Seq}_2$ have both inputs in S_2. In the construction of $\textsc{Seq}(u)$ according to its definition, if some $\textsc{Unite}$ in $\textsc{Seq}_1$ has not yet been selected, there will be an unselected $\textsc{Unite}$ in $\textsc{Seq}_1$ that adds a new element to the set containing u. Such a $\textsc{Unite}$ will be selected before $\textsc{Unite}(v, w)$ by the tie-breaking rule. Once all the $\textsc{Unite}$ operations in $\textsc{Seq}_1$ have been selected, $\textsc{Unite}(v, w)$ will be selected. Since no $\textsc{Unite}$ in $\textsc{Seq}_2$ has a vertex in S_1 as an input, the $\textsc{Unite}$ operations in $\textsc{Seq}_2$ will be selected in the order they occur in $\textsc{Seq}_2(w)$. □

Lemma 3.2.8. *Let* $\textsc{Tree}$ *and* $\textsc{Tree}(u)$*, respectively, be the trees built by executing* $\textsc{Seq}$ *and* $\textsc{Seq}(u)$ *using linking by index. Then every ancestor of* u *in* $\textsc{Tree}$ *is an ancestor of* x *in* $\textsc{Tree}(u)$*.*

Proof. The proof is by induction on the length of $\textsc{Seq}$ using Lemma 3.2.7. Assume u is in S_1; the argument is symmetric if u is in S_2. Let $\textsc{Tree}_1$ and $\textsc{Tree}_1(u)$ be the trees built

by $\textsc{Seq}_1$ and $\textsc{Seq}_1(u)$, respectively. Let x and y be the largest elements in S_1 and S_2, respectively. If $x > y$, the ancestors of u in $\textsc{Tree}$ are exactly the same as in $\textsc{Tree}_1$, so the lemma holds. If $x < y$, the ancestors of u in $\textsc{Tree}$ are y and those in $\textsc{Tree}_1$. In this case y is maximum in $\textsc{Tree}$, hence a prefix maximum in $\textsc{Seq}(u)$, and hence an ancestor of u in $\textsc{Tree}(u)$ by Lemma 3.2.6, so the lemma holds in this case also. By the induction hypothesis every ancestor of u in $\textsc{Tree}_1$ is an ancestor of u in $\textsc{Tree}_1(x)$. □

Lemma 3.2.9. *Let* $\textsc{Tree}$ *and* $\textsc{Tree}(u)$*, respectively, be the trees built by executing* $\textsc{Seq}$ *and* $\textsc{Seq}(u)$ *using early linking by index. Then the proper ancestors of* u *are the same in* $\textsc{Tree}$ *and* $\textsc{Tree}(u)$*.*

Proof. The proof is by induction on the length of $\textsc{Seq}$ but is somewhat more complicated than the proof of Lemma 3.2.8. Assume u is in S_1; the argument is symmetric if u is in S_2. Let $\textsc{Tree}_1$, $\textsc{Tree}_2$, $\textsc{Tree}_1(u)$, and $\textsc{Tree}_2(w)$ be the trees built by $\textsc{Seq}_1$, $\textsc{Seq}_2$, $\textsc{Seq}_1(u)$, and $\textsc{Seq}_2(w)$, respectively. Let x be the largest element in S_1. The ancestors of u in $\textsc{Tree}$ are exactly the ancestors of u in $\textsc{Tree}_1$, plus the ancestors of w in $\textsc{Tree}_2$ that are greater than x. By the induction hypothesis, these are exactly the ancestors of u in $\textsc{Tree}_1(u)$, plus the ancestors of w in $\textsc{Tree}_2(w)$ that are greater than x. Recall that $u_0, u_1, \ldots$ is the sequence of elements added to the set containing u by $\textsc{Seq}(u)$. Let $w = u_k$. By Lemma 3.2.7, $u_k, u_{k+1}, \ldots$ is the sequence of elements added by $\textsc{Seq}_2(w)$ as it builds $\textsc{Tree}_2(w)$. By Lemma 3.2.6, the ancestors of u in $\textsc{Tree}_1(u)$ are the prefix maxima in $u_0, u_1, \ldots, u_{k-1}$, and the ancestors of w in $\textsc{Tree}_2(w)$ are the prefix maxima in $u_k, u_{k+1}, \ldots$. Among the latter, those greater than x are exactly the ones that are also prefix maxima in $u_0, u_1, \ldots$. The lemma follows. □

Lemma 3.2.10. *With randomized linking or randomized early linking but no compaction, the expected number of proper ancestors of* u *of the same rank as* u *in the final forest is at most two.*

Proof. By Lemmas 3.2.8 and 3.2.9 it suffices to bound the expected number of ancestors of

x having the same rank as u in the tree built by SEQ(u). By Lemma 3.2.6 these are exactly the prefix maxima of $u_0, u_1, \ldots$ that are of the same rank as u, all of which precede the first prefix maximum (if any) of rank greater than that of u. Let $k > 0$ be such that if $i < k$, u_i has rank at most that of u, and let u_j be maximum among $u_0, u_1, \ldots, u_{k-1}$. Since the definition of SEQ(u) is independent of the random numbering, every element greater than u_j is equally likely to be u_k. (This need not be true for elements less than u_k, but that is irrelevant to our argument.) Property 10 implies that, given that u_k is a prefix maximum, the conditional probability that the rank of u_k is greater than that of u_j is at least $1/2$. Thus each successive prefix maximum has probability at least $1/2$ of having greater rank than u. It follows that the expected number of proper ancestors of u of the same rank as u is at most $\sum_{i=1}^{\infty} i/2^i = 2$. (Node u is an ancestor but not a proper ancestor of itself.) □

Lemma 3.2.11. *The ranks defined for randomized linking and randomized early linking satisfy Properties 1, 2, and 5. They also satisfy Property 3 in expectation. Thus in expectation it is a strongly well-behaved rank function.*

Proof. Property 1 is immediate from the definition of fixed ranks. Property 2 follows from the fact that the algorithm ensures strictly increasing labels along paths and Property 9. Properties 3 and 5 follow directly from Lemma 3.2.10 and Property 11 respectively. □

Now that we have proof of a well-behaved rank function for randomized linking and randomized early linking, we can state two theorems of expected optimality, the truth of which follow directly from Corollary 3.0.1 and Lemmas 3.1.1, 3.1.2, 3.1.3, 3.1.8, and 3.2.11.

Theorem 3.2.12. *If* FIND *operations are performed with any combination of compression, splitting, or halving, and* UNITE *operations are performed with randomized linking, then a sequence of m intermixed operations takes $O(m\alpha(n, m/n))$ time in expectation.*

Theorem 3.2.13. *If* FIND *operations are performed with any combination of compression, splitting, or halving, and* UNITE *operations are performed with randomized early linking with*

compression, splitting, halving, or splicing, then a sequence of m intermixed operations takes $O(m\alpha(n, m/n))$ time in expectation.

Reducing the Required Randomness

Up until this point, we have assumed that the nodes are labeled with a uniformly random permutation, which of course requires $\Omega(n \log n)$ random bits; however, this is not strictly necessary. We will now show that in fact only $\mathrm{O}\big(\log^2 n\big)$ truly random bits suffice. We will use the notion of almost k-wise independence to establish this result. Specifically, we will assign almost $\lceil \log n \rceil$-wise independent random ranks to nodes in the range 1 to $\lceil \log n \rceil$, and use these to generate a pseudorandom permutation of the nodes with which to perform linking by index. We will ensure that these random ranks satisfy Property 9 and closely approximate Properties 10 and 11. We now describe the implementation in detail.

Let S_n be some collection of n-bit strings in $\{0,1\}^n$. We say that S_n is a *p-wise independent* space if when we sample a string $X = x_1 x_2 \dots x_n$ uniformly at random from S_n, then for any $q \in [p]$, a string $\alpha \in \{0,1\}^q$, and positions $1 \le j_1 < j_2 < \ldots < j_q \le n$, we have

$$\mathbf{P}\big[x_{i,j_1} x_{i,j_2} \dots x_{i,j_q} = \alpha\big] = 2^{-q}.$$

Similarly, we say that S_n is *p-wise independent space with an ϵ-bias* if when we sample a string $X = x_1 x_2 \dots x_n$ uniformly at random from S_n, then for any $q \in [p]$, a string $\alpha \in \{0,1\}^q$, and positions $1 \le j_1 < j_2 < \ldots < j_q \le n$, we have

$$2^{-q} - \epsilon \le \mathbf{P}\big[x_{i,j_1} x_{i,j_2} \dots x_{i,j_q} = \alpha\big] \le 2^{-q} + \epsilon.$$

We will rely on the following result of Alon et al. [2]: one can generate n p-wise independent random bits with an ϵ-bias using only $\mathrm{O}\big(\log\big(\frac{p \log n}{\epsilon}\big)\big)$ random bits.

We now assign random ranks to the n nodes as follows. We begin by initializing two linked lists. One will be called the *rank-assigned* list, while the other is called the *working* list. First, add all nodes to the working list. We will proceed in $p = \lceil \log n \rceil$ rounds. Let n_i be

the length of the working list at the beginning of the i^{th} round, and let v_i be the position of node v in the working list at the beginning of the i^{th} round. In each round we draw a sample X_i from a p-wise independent space S_{n_i} with a $(1/n^2)$-bias where $X_i = x_{i,1}, x_{i,2}, \ldots, x_{i,n_i}$—in effect we generate a bit for each node in the working list. By the above-mentioned result of Alon et al., each X_i can be generated using $\mathrm{O}(\log n)$ random bits; thus this process uses $\mathrm{O}\big(\log^2 n\big)$ random bits in total.

In round i, a node v is assigned rank i and moved from the working list to the end of the rank-assigned list if its corresponding bit x_{i,v_i} is 1. The remaining nodes are left in the working list. If v is still in the working list at the end of round p, or equivalently $x_{i,v_i} = 0$ for all $i \in [p]$, then define $v.\textsc{rank} = p$.

After all rounds are completed, traverse the rank-assigned list from the beginning and assign the nodes labels from 1 to n based on their order in the list. Thus since the nodes were moved to the end of the rank-assigned list in order of increasing ranks, the ranks satisfy Property 9.

By definition of ϵ-bias spaces, we know that for any $1 \leq i < p$,

$$\begin{aligned}\mathbf{P}[v.\textsc{rank} = i] &= \mathbf{P}\big[x_{1,v_1} = 0, x_{2,v_2} = 0, \ldots, x_{i-1,v_{i-1}} = 0, x_{i,v_i} = 1\big] \\ &\leq \left(\frac{1}{2} + \epsilon\right)^i \\ &\leq 2^{-i} + \frac{1}{n},\end{aligned}$$

for $\epsilon = 1/n^2$. Similarly, $\mathbf{P}[v.\textsc{rank} = p] \leq 2^{-p+1} + 1/n$.

Let $\gamma(i)$ be the random variable that denotes the number of nodes with rank $i \in [p]$. It follows then

$$\mathbf{E}[\gamma(i)] \leq \frac{2n}{2^i} + 1.$$

Note that this is quite close to Property 11, and in fact satisfies the more important Property 5. Note also that this entire procedure then takes linear time in expectation since the number of bits generated is proportional to the sum of ranks.

We now establish that the ranks above are sufficiently random to prove that Lemma 3.2.10 holds, or more accurately, that the expected number of proper ancestors of v with the same rank as v is O(1). Given the sequence $\textsc{Seq}(v)$, let A denote the subsequence of nodes in $\textsc{Seq}(v)$ that have rank at least $k = v.\textsc{rank}$, excluding v itself. Consider any subsequence $\tau = \langle x_{j,u_1}, x_{j,u_2}, \ldots x_{j,u_q} \rangle$ of $q \leq p-1$ bits from any X_j such that $v_j \notin \{u_1, u_2, \ldots, u_q\}$. Let α be an arbitrary string of q bits. Conditioned on $v.\textsc{rank}$ (which fixes at most one bit from X_j), the probability that $\tau = \alpha$ is at most

$$\frac{2^{-(q+1)} + \epsilon}{1/2 - \epsilon} \leq 2^{-q} + 4\epsilon = 2^{-q} + 4/n^2.$$

If $v.\textsc{rank} = p$, then for any $w \neq v$,

$$\mathbf{P}[w.\textsc{rank} = p] \leq \left(1/2 + 4/n^2\right)^{-p+1} \leq 2^{-p+1} + \mathrm{o}(1/n) = \mathrm{O}(1/n);$$

hence $\mathbf{E}[A.\textsc{size}] = \mathrm{O}(1)$. Otherwise, $v.\textsc{rank} < p$, and for any element $w \in A$, its rank $w.\textsc{rank} \geq v.\textsc{rank}$ only if $x_{k,w_k} = 1$. Thus the expected size of the longest prefix of A with $x_{k,w_k} = 1$ for all w in the prefix is at most

$$\sum_{i=1}^{p-1} i\left(\frac{1}{2^i} + \frac{4}{n^2}\right) + n\left(\frac{1}{2^{p-1}} + \frac{4}{n^2}\right) = \mathrm{O}(1).$$

The second term in the expression above corresponds to the case when the size of the prefix is at least $p-1$; in this case we cannot assume any independence for subsequent nodes in $\textsc{Seq}(v)$, and we bound it by the maximum possible size of set A. Thus Lemma 3.2.10 holds as before.

3.3 Remarks

In this chapter we have provided an alternate, compact analysis of all the linking rules and all but one of the compaction methods known to-date to give an optimal solution for disjoint set union using the tools developed in Chapter 2.

Additionally, we showed that the remaining compaction method, splicing, performs optimally when combined with a linking rule that admits strongly well-behaved ranks. This proof provides explicit evidence that splicing by rank is optimal, which was left as an exercise by Tarjan and van Leeuwen [24]. The analysis, though not strictly in our framework from Chapter 2, uses many of the same tools and techniques. It remains an open question whether splicing is well-structured by the definition given, and if not, whether the definition can be relaxed so it applies to splicing while still giving sufficient structure to prove the Sledgehammer Theorem (2.3.2).

We also proposed a new linking rule, randomized linking, which we have proved admits a strongly well-behaved rank function in expectation and which has been shown to perform very well in experiments. One point worth noting is that we do not advocate the direct implementation of randomized linking. Rather, we have demonstrated that in settings where the randomness can be attained for free, for instance when the input is sufficiently randomized or where element names are already hashed, it is safe to use linking by index or early linking by index. We demonstrated that $\mathrm{O}\big(\log^2 n\big)$ truly random bits suffice if used carefully, but it is not clear that this bound is tight.

It is worth noting that there is a natural analogy between randomized linking and the treap data structure of Aragon and Seidel [4]. In their case they implicitly use a uniformly random permutation to structure a binary search tree in heap order, and in order to reduce the number of random bits required, they generate successive bits of their priorities as needed for comparisons. It may be that a strategy such as this could be used instead of our $\mathrm{O}\big(\log^2 n\big)$ analysis to provide an arguably simpler algorithm using $\mathrm{O}(n)$ random bits.

Our analyses illustrate the commonalities between the algorithms in question and give a blueprint for analyzing possible future algorithms.

Coin-Flip Linking

Randomized linking is equivalent to randomly linking by size: when linking the roots u and v of two trees containing $u.\textsc{size}$ and $v.\textsc{size}$ nodes, respectively, make u the parent of v with probability $u.\textsc{size}/(u.\textsc{size} + v.\textsc{size})$. An even simpler randomized form of 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 solutions using coin-flip linking, unlike those using randomized linking, are *not* asymptotically optimal, and indeed are asymptotically no more efficient than those using naïve linking.

Support for this conjecture comes from considering the following bad example for naïve linking with path compression [24]. 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 3.7 shows the experimental result of executing 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 $\alpha(n, d)$.

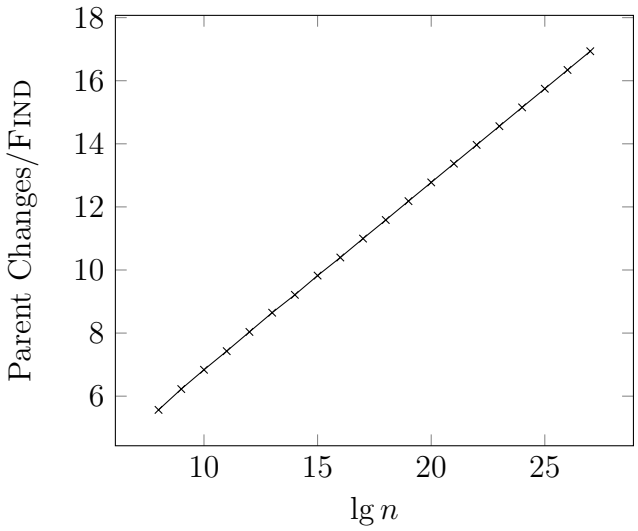

Figure 3.7: Average parent changes per FIND as a function of $\lg n$ in coin-flip linking experiments.

Chapter 4

Nested Set Union

You could have a steam train if you'd just lay down your tracks.

PETER GABRIEL, *"Sledgehammer"*

As introduced in Chapter 1, nested set union is a generalization of disjoint set union in which k nested partitions are maintained rather than just one. The algorithms presented here are generalizations of those presented in Chapter 3.

We present two compaction methods: we prove that *segmented compression* is well-structured for an arbitrary constant k, while *segmented splitting* is well-structured for $k = 2$. Similarly, we present three linking rules: we prove *nested linking by rank* admits strongly well-behaved ranks for arbitrary constant k, *nested linking by size* admits strongly well-behaved ranks for $k = 2$, and *randomized nested linking* admits strongly well-behaved ranks for $k = 2$ given certain assumptions about the compaction method (which happen to hold for segmented compression).

Each of these algorithms respects the nested partition. Each tree represents a set in the first partition, and each tree is divided into contiguous subtrees representing finer sets. In order to keep track of this nesting algorithmically, we store with each node u the coarsest

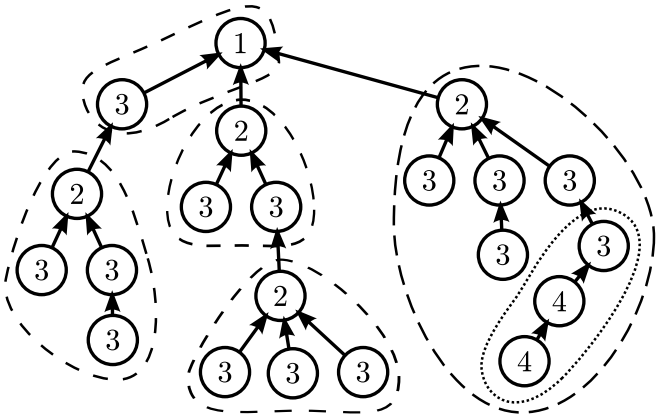

Figure 4.1: An example of tree partitioning for nested set union algorithms. The number displayed in each node u is the value of u.SUBROOT—the coarsest partition in which it is still a root. The entire tree is a set in the first partition, the groups of nodes with dashed outlines are sets in the second partition, and the group with dotted outline is a set in the third partition (all other sets in the third partition are individual nodes).

partition in which it is still a root, u.SUBROOT, initially 1. Then the root v of the set in the i^{th} parition containing u is the nearest ancestor v of u such that v.SUBROOT $\leq i$ (note that this may be u itself). See Figure 4.1 for a visual example of the partitioning scheme.

In order to ensure that the operations respect the nested partition, two nodes u and v can only be linked in partition $i > 1$ if they are in the same set in partition $i-1$. Suppose u loses such a link. It ceases to be a root in its current partition, so u.SUBROOT is incremented. Its subtree is removed from its current location in the tree and reattached underneath v. Since u and v were already in the same set in partition $i-1$, this reattachment is safe.

As compaction proceeds, our algorithms ensure that any parent changes respect the nesting as well—no parent changes will change the node sets. More specifically, we guarantee that if u.PARENT gets set to u.PARENT^i, then for all $1 \leq j < i$,

$$u.\text{SUBROOT} \leq u.\text{PARENT}^j.\text{SUBROOT}.$$

To give us a slightly shorter presentation of some results later on, we present the following

corollary, which follows directly from the Sledgehammer Theorem (2.3.2) and Lemmas 2.4.2 and 2.4.3.

Corollary 4.0.1. *If* FIND *operations are performed with a well-structured compaction method and* UNITE *operations are performed with a linking rule that admits a strongly well-behaved rank function, a sequence of m intermixed operations takes $O(m\alpha(n, m/n))$ time.*

4.1 Compaction Methods

4.1.1 Segmented Compression

Segmented compression carries out as drastic compaction as possible while still respecting the nested partition. Each node u on the FIND path has its parent set to its nearest coarser ancestor v—the first node past u on the path such that $u.\text{SUBROOT} > v.\text{SUBROOT}$. Note that for $i > 1$, it isn't strictly necessary to run the compression all the way to the root. Instead, the FIND may return as soon as a root in the i^{th} partition is found. We will call this variant *local compression*. Both of these variants can be carried out by the same recursive function with a simple change of parameter. Such an implementation is shown in Algorithm 4.1.1, which makes use of a simple stack to keep track of coarse ancestors, and its effects are demonstrated in Figure 4.2. See Algorithms 4.1.2 and 4.1.3 for the instantiations necessary for FIND with segmented and local compression respectively.

By setting each node's parent to be its nearest coarser ancestor, segmented and local compression collapse the FIND path to a tree of depth at most k. This means that for each node u on the path, $u.\text{PARENT}^k$ is at least the head of the path. For all but the last $2k$ nodes on the path, their new k^{th} ancestor has rank at least that of their old $2k^{th}$ ancestor. Thus we can pick $g = t = k$ and $q = 2k$ to show that segmented compression is well-structured.

Lemma 4.1.1. *For $g = t = k$ and $q = 2k$, segmented compression satisfies Properties 6,*

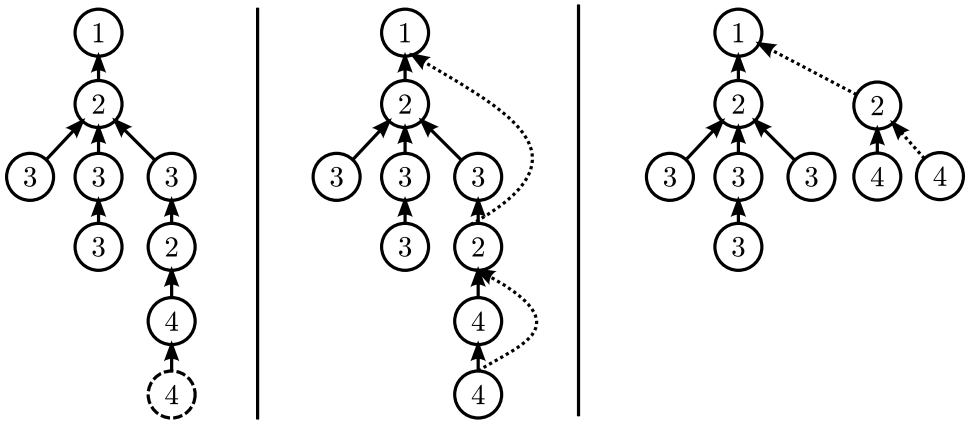

Figure 4.2: The value shown in each node u is the value of u.SUBROOT. The figure demonstrates the effects of segmented-compression beginning at the node outlined in red.

Algorithm 4.1.1 Segmented Compression

```
procedure SegmentCompress(S, i, x)
    if x.subroot ≤ i then
        S ← Push(S, x)
        return S
    S ← SegmentCompress(S, i, x.parent)
    u ← Peek(S)
    while u.subroot ≥ x.subroot and NotEmpty(S) do
        S ← Pop(S)
        u ← Peek(S)
    x.parent ← u
    S ← Push(S, x)
    return S
```

Algorithm 4.1.2 Find with Segmented Compression

```
procedure Find(i, x)
    S ← SegmentCompress(NewStack(), 1, x)
    u ← Peek(S)
    while u.subroot > i do
        S ← Pop(S)
        u ← Peek(S)
    return u
```

Algorithm 4.1.3 FIND with Local Compression

```
procedure FIND(i, x)
    S ← SEGMENTCOMPRESS(NEWSTACK(), i, x)
    u ← PEEK(S)
    while u.SUBROOT > i do
        S ← POP(S)
        u ← PEEK(S)
    return u
```

7, and 8 in the presence of weakly well-behaved ranks. Thus segmented compression is well-structured.

Proof. Given an input sequence $\langle u_1, u_2, \ldots, u_h \rangle$, we select as our chosen subsequence $\langle u_1, u_{k+1}, u_{2k+1}, \ldots, u_{h'} \rangle$. Then $kh' \geq h - 2k$ and thus that Property 6 is satisfied. Since for each $1 \leq j < h'$ the successor of u_{i_j} in the sequence is $u_{i_j}.\text{PARENT}^k$ and given our choice of $t = k$, it is clear that $u_{i_j}.\text{PARENT}^k.\text{RANK} = u_{i_{j+1}}.\text{RANK}$, and thus Property 7 is satisfied. Finally, segmented compression changes the k^{th} ancestor of each node in the sequence to one with rank at least that of the head of the path, so for $1 \leq j < h'$, $u_{i_j}.\text{PARENT}^k.\text{RANK}$ gets set to some $r \geq u_h.\text{RANK}$. Property 2 implies that with weakly well-behaved ranks, this value must be at least $u_{i_j}.\text{PARENT}^{2k}.\text{RANK} = u_{i_{j+1}}.\text{PARENT}^k.\text{RANK}$, and thus Property 8 holds. Therefore, the lemma holds. □

4.1.2 Segmented Splitting

As in the case of disjoint set union, it may be that a one-pass method would perform better in practice. To this end, we propose *segmented splitting* for the restricted case of two partitions. To prove that this method is well-structured, we will have to make a slight modification—considering intervals of three nodes at a time rather than two—which makes the algorithm slightly more complicated than standard splitting. It is possible that similar modifications could be developed for the general case of k partitions, but they would likely be rather involved.

As in the case of segmented compression, it is possible to restrict the compaction to only the relevant subtree for operations in the second partition. We use *local splitting* as a primitive in our fully segmented algorithm, splitting each contiguous subpath of nodes which are not roots in either partition, handling the subroot, and then proceeding upward along the path.

Local splitting replaces the parent of each node u with $u.\text{PARENT}^3$, with two exceptions: if $u.\text{PARENT}$ is a subroot, but not u, u's parent does not change; if $u.\text{PARENT}^2$ is a subroot, but not u or $u.\text{PARENT}$, this node becomes u's parent. See Algorithm 4.1.4 for an implementation of local splitting and Algorithm 4.1.5 for an implementation of the path segmentation.

Algorithm 4.1.4 Local Splitting

```
procedure SPLIT(x)
    if x.SUBROOT ≤ 2 then
        return x
    u ← x.PARENT
    if u.SUBROOT ≤ 2 then
        return u
    v ← u.PARENT
    while v.SUBROOT = 3 do
        x.PARENT ← v.PARENT
        x ← u
        u ← v
        v ← v.PARENT
    x.PARENT ← v
    return v
```

Algorithm 4.1.5 Segmented Splitting

```
procedure SEGMENTSPLIT(x)
    v ← SPLIT(x)
    u ← v
    while u.PARENT ≠ u do
        x ← u.PARENT
        u.PARENT ← x.PARENT^2
        u ← SPLIT(x)
    return (u, v)
```

The key property we will use to show segmented splitting is well-structured is that all but the last few nodes get a new grandparent that is at least the old great-great-grandparent. To understand why, consider the cases for a node u which is not among the last 6 on the path. If u is a subroot, its new parent is $u.\text{PARENT}^3$, and thus its new grandparent is at least $u.\text{PARENT}^4$. This is similarly the case if none of u, $u.\text{PARENT}$, and $u.\text{PARENT}^2$ is a subroot. If $u.\text{PARENT}^2$ is a subroot, then it becomes u's new parent, and its new grandparent is $u.\text{PARENT}^5$. If $u.\text{PARENT}$ is a subroot, then u's parent does not change; however, its new grandparent will be $u.\text{PARENT}^4$ since $u.\text{PARENT}$ will change parents.

Lemma 4.1.2. *For $g = t = 2$ and $q = 6$, segmented splitting satisfies Properties 6, 7, and 8 in the presence of weakly well-behaved ranks. Thus segmented splitting is well-structured.*

Proof. Given an input sequence $\langle u_1, u_2, \ldots, u_h \rangle$, we select as our chosen subsequence $\langle u_1, u_3, u_5, \ldots, u_{h_0} \rangle$ where $h_0 = h-6+(h \mod 2)$. Note that $h_0 \geq h-6$, and thus $2h' \geq h-6$. This implies that $gh' + q \geq h - 6 + 6 = h$ and thus that Property 6 is satisfied. Since for each $1 \leq j < h'$ the successor of u_{i_j} in the sequence is its grandparent and given our choice of $t = 2$, it is clear that $u_{i_j}.\text{PARENT}^2.\text{RANK} = u_{i_{j+1}}.\text{RANK}$, and thus Property 7 is satisfied. Finally, segmented splitting guarantees that for all but the last node in the chosen subsequence, the new grandparent is at least the old grandparent's grandparent. More precisely, for $1 \leq j < h'$, $u_{i_j}.\text{PARENT}^2.\text{RANK}$ gets set to $r \geq u_{i_j}.\text{PARENT}^4.\text{RANK} = u_{i_{j+1}}.\text{PARENT}^2.\text{RANK}$, and thus Property 8 holds. Therefore, the lemma holds. □

4.2 Linking Rules

4.2.1 Nested Linking by Rank

Like linking by rank, *nested linking by rank* maintains the rank $u.\text{RANK}$ of each node, initially 0. When linking two nodes, the node of greater rank is the winner; in the case

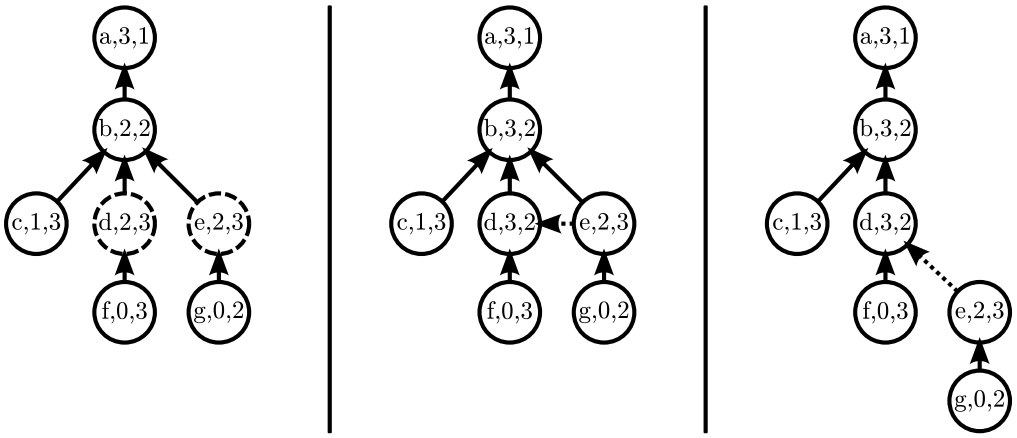

Figure 4.3: The triple of values shown in each node u is of the form $u, u.\text{RANK}, u.\text{SUBROOT}$. The figure demonstrates the effects of linking nodes d and e: d wins the link, so $e.\text{PARENT} \leftarrow d$, $d.\text{RANK}$ and $e.\text{SUBROOT}$ are incremented, and since $b.\text{RANK} < d.\text{RANK}$, $b.\text{RANK}$ is also incremented. The propagation of rank updates ceases upon discovering that $a.\text{RANK} = b.\text{RANK}$ and $a.\text{SUBROOT} < b.\text{SUBROOT}$.

of a tie, the winner is chosen arbitrarily and its rank incremented. In either case, the loser v has $v.\text{SUBROOT}$ incremented. If the rank of the winner u was incremented and it is not a root, then further updates may propagate up the tree. Unlike linking by rank, nested linking by rank does allow limited rank ties, but only in the case of nodes w such that $w.\text{SUBROOT} > w.\text{PARENT.SUBROOT}$. We can use two tricks then which preserve the nested partition, maintain non-decreasing ranks, and keep the number of rank ties limited. If $u.\text{RANK} > u.\text{PARENT.RANK}$, we increment $u.\text{PARENT.RANK}$ and advance up the path, setting $u \leftarrow u.\text{PARENT}$. (See Figure 4.3.) If $u.\text{RANK} = u.\text{PARENT.RANK}$ and $u.\text{SUBROOT} \leq u.\text{PARENT.SUBROOT}$, we set $u.\text{PARENT} \leftarrow u.\text{PARENT}^2$ and terminate. (See Figure 4.4.) Thus each node u maintains $u.\text{RANK} \leq u.\text{PARENT.RANK}$ and $u.\text{TIES} \leq k$. See Algorithm 4.2.1 for an implementation.

Proving that the ranks maintained by the algorithm are strongly well-behaved will be considerably more involved than it was for the non-nested algorithm. We will have separate proofs first that the ranks decay exponentially and that there are limited rank ties. The

Algorithm 4.2.1 UNITE with Nested Linking by Rank

```
function UNITE(i, x, y)
    if i > 1 then
        u ← FIND(i − 1, x)
        v ← FIND(i − 1, y)
        if u ≠ v then
            return error
    u ← FIND(i, x)
    v ← FIND(i, y)
    if u = v then
        return false
    if u.RANK < v.RANK then
        u ↔ v
    v.PARENT ← u
    v.SUBROOT ← i + 1
    if u.RANK = v.RANK then
        u.RANK ← u.RANK + 1
        while (u.RANK > u.PARENT.RANK) or
                (u.RANK = u.PARENT.RANK and u.SUBROOT ≤ u.PARENT.SUBROOT)
        do
            if u = u.PARENT then
                return true
            else if u.RANK > u.PARENT.RANK then
                u.PARENT.RANK ← u.PARENT.RANK + 1
                u ← u.PARENT
            else
                u.PARENT ← u.PARENT^2
    return true
```

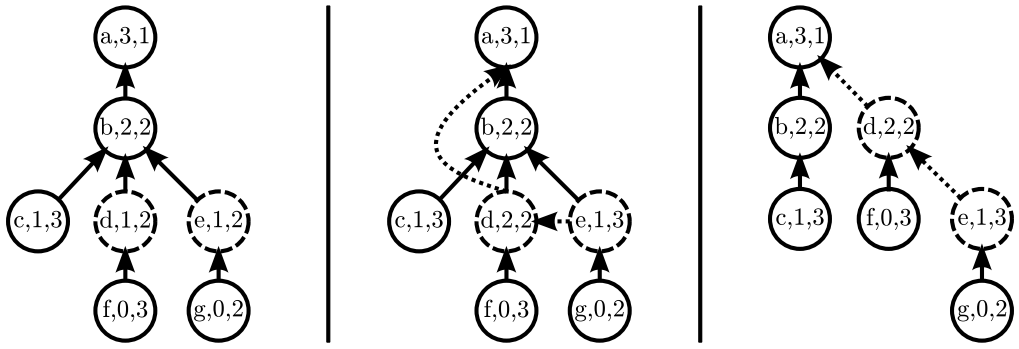

Figure 4.4: The triple of values shown in each node u is of the form $u, u.\text{RANK}, u.\text{SUBROOT}$. The figure demonstrates the effects of linking nodes d and e: d wins the link, so $e.\text{PARENT} \leftarrow d$, $d.\text{RANK}$ and $e.\text{SUBROOT}$ are incremented, and since $b.\text{SUBROOT} \geq d.\text{SUBROOT}$, it is safe to shorcut and set $d.\text{PARENT} \leftarrow b$. This maintains the invariant that $u.\text{RANK} = u.\text{PARENT.RANK}$ only if $u.\text{PARENT.SUBROOT} < u.\text{SUBROOT}$.

former constitutes the bulk of this subsection, and uses a credit-based argument. Unfortunately, we do not know of a more concise or elegant proof, but at least it is simple in principle.

Lemma 4.2.1. *Using nested linking by rank, at most cn/f^x nodes u have $u.\text{RANK}_{max} = x$ where $c = k + k/(k+1) - k(k/(k+1))^k$ and $f = k^k + 1/k^k$.*

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 that requires no new credit to be introduced to the system. Let $\text{ROOTS}_i(x)$ denote the number of roots of rank x in partition i. The transitions and associated node count adjustments are the following:

1. Two roots in partition i of ranks x and $y < x$ are linked. $\text{ROOTS}_i(y)$ decreases by 1 and $\text{ROOTS}_{i+1}(y)$ increases by 1.

2. Two roots in the first partition of rank x are linked. $\text{ROOTS}_1(x)$ decreases by 2, $\text{ROOTS}_1(x+1)$ increases by 1, and $\text{ROOTS}_2(x)$ increases by 1.

3. Two roots in partition $i > 1$ of rank x are linked underneath a node of rank at least

$x+1$. $\textsc{Roots}_i(x)$ decreases by 2, $\textsc{Roots}_i(x+1)$ increases by 1, and $\textsc{Roots}_{i+1}(x)$ increases by 1.

4. Two roots in partition $i > 1$ of rank x are linked underneath a node of rank x. Let $i > i_1 > \ldots > i_j \geq 1$ be the values of $u.\textsc{subroot}$ for the successive ancestors u of the winning root that have their ranks increased. Then $\textsc{Roots}_i(x)$ decreases by 2, $\textsc{Roots}_i(x+1)$ increases by 1, and $\textsc{Roots}_{i+1}(x)$ increases by 1, and for each $1 \leq \ell \leq j$, $\textsc{Roots}_{i_\ell}(x)$ decreases by 1 and $\textsc{Roots}_{i_\ell}(x+1)$ increases by 1.

Each node must hold a number of credits dependent on its type and rank. A root in partition i of rank x must hold at least $\textsc{Credits}_i(x)$ credits. We now examine the requirements each transition imposes on these these values under the assumption that the amount of credit in the system does not increase.

1. $\textsc{Credits}_i(x) \geq \textsc{Credits}_{i+1}(x)$
2. $2\textsc{Credits}_1(x) \geq \textsc{Credits}_1(x+1) + \textsc{Credits}_2(x)$
3. $2\textsc{Credits}_i(x) \geq \textsc{Credits}_i(x+1) + \textsc{Credits}_{i+1}(x)$
4. $2\textsc{Credits}_i(x) + \sum_{\ell=1}^{j} \textsc{Credits}_{i_\ell}(x) \geq \textsc{Credits}_i(x+1) + \textsc{Credits}_{i+1}(x) + \sum_{\ell=1}^{j} \textsc{Credits}_{i_\ell}(x+1)$

These requirements leave some flexibility, but here is one simple and uniform way to satisfy them. We first set $\textsc{Credits}_{k+1}(0) = 1$, then we set $\textsc{Credits}_{i-1}(x) = a \cdot \textsc{Credits}_i(x)$ and $\textsc{Credits}_i(x+1) = b \cdot \textsc{Credits}_i(x)$. As a general form then, $\textsc{Credits}_i(x) = a^{k+1-i}b^x$. This leads to a total credit of $a^k \cdot n$ in the system since all nodes begin as roots of rank 0 in the first partition.

To show that the transitions do not require an increase in total credit, we must find constants a and b such that the associated inequalities hold. To show that the number of nodes of rank x decreases exponentially with x, we will require $b > 1$. The first transition

implies simply that $a \geq 1$ while the second and third imply that $2a \geq ab + 1$. The last constraint on the other hand is somewhat tricky. The worst case will be that $i_1 = i - 1, i_2 = i - 2, \ldots, i_j = 1$. To see that this is true, it suffices to observe that

$$\begin{aligned} & \textsc{Credits}_i(x+1) - \textsc{Credits}_i(x) \\ = \; & a^{k+1-i}b^x(b-1) \\ \geq \; & a^{k+1-j}b^x(b-1) \\ = \; & \textsc{Credits}_j(x+1) - \textsc{Credits}_j(x) \end{aligned}$$

for $i \leq j$, which is certainly true for $a \geq 1$. Then we must prove that for our choices,

$$\begin{aligned} & 2\textsc{Credits}_i(x) + \sum_{\ell=1}^{i-1} \textsc{Credits}_{i-\ell}(x) \\ & \geq \textsc{Credits}_i(x+1) + \textsc{Credits}_{i+1}(x) + \sum_{\ell=1}^{i-1} \textsc{Credits}_{i-\ell}(x+1). \end{aligned}$$

This can be expressed as follows.

$$\begin{aligned} 2a^{k+1-i}b^x + \sum_{\ell=1}^{i-1} a^{k+1-i+\ell}b^x &\geq a^{k+1-i}b^{x+1} + a^{k-i}b^x + \sum_{\ell=1}^{i-1} a^{k+1-i+\ell}b^{x+1} \\ 2a + \sum_{\ell=1}^{i-1} a^{\ell+1} &\geq 1 + \sum_{\ell=0}^{i-1} a^{\ell+1}b \\ 2a(a-1) - a\big(a - a^i\big) &\geq ab\big(a^i - 1\big) + a - 1 \end{aligned}$$

We posit that chosing $a = k + 1/k$ and $b = k^k + 1/k^k$ satisfies all the necessary inequalities. Trivially, $a > 1$ and $b > 1$ for $k \geq 2$. To check the second inequality, observe that for

$k \geq 2$, $1/k^{k-1} \leq 1/2$ and $1/k^k \leq 1/4$ and therefore

$$\begin{aligned}
2\left(\frac{k+1}{k}\right) &\geq \left(\frac{k+1}{k}\right)\left(\frac{k^k+1}{k^k}\right)+1 \\
2k+2 &\geq (k+1)\left(1+\frac{1}{k^k}\right)+k \\
k+2 &\geq (k+1)+\frac{k+1}{k^k} \\
1 &\geq \frac{1}{k^{k-1}}+\frac{1}{k^k} \\
1 > \frac{3}{4} &= \frac{1}{2}+\frac{1}{4} \geq \frac{1}{k^{k-1}}+\frac{1}{k^k}.
\end{aligned}$$

Finally, the last inequality can be simplified considerably. It would take quite a bit of space to show all the intermediate steps, but they can easily be verified (for instance through standard computer algebra software).

$$\begin{aligned}
2a(a-1)-a\big(a-a^i\big) &\geq ab\big(a^i-1\big)+a-1 \\
k^k &\geq k(k+1)\left(\left(\frac{k+1}{k}\right)^i-1\right) \\
k^{k-1} &\geq (k+1)\left(\left(\frac{k+1}{k}\right)^i-1\right) \\
k^{k-1} &\geq \frac{(k+1)^{i+1}}{k^i}-(k+1) \\
1+\frac{k+1}{k^{k-1}} &\geq \frac{(k+1)^{i+1}}{k^{k+i-1}}
\end{aligned}$$

To help prove this inequality, we make two relaxations:

$$1+\frac{k+1}{k^{k-1}} \geq 1 \text{ and } \left(\frac{k+1}{k}\right)^3\left(\frac{k+1}{k^2}\right)^{k-2} \geq \frac{(k+1)^{i+1}}{k^{k+i-1}}.$$

Then for $k \geq 4$,

$$\left(\frac{k+1}{k}\right)^3\left(\frac{k+1}{k^2}\right)^{k-2} \leq \left(\frac{5}{4}\right)^3\left(\frac{5}{16}\right)^2 = \frac{3125}{16384} < 1$$

so the inequality holds. For the case of $k = 3$,

$$\left(\frac{k+1}{k}\right)^3\left(\frac{k+1}{k^2}\right)^{k-2} \leq \left(\frac{4}{3}\right)^3\left(\frac{4}{9}\right) = \frac{256}{243} < \frac{13}{9} = 1+\frac{k+1}{k^{k-1}}$$

so the inequality again holds.

Unfortunately, the inequality *does not* hold for $k = 2$; however, we can handle this case fairly easily by slightly altering the definition of $\textsc{Credits}_i(x)$. For $i \leq k = 2$, we will still define $\textsc{Credits}_i(x) = a^{k+1-i}b^x$, but we will set $\textsc{Credits}_3(x) = 0$ for all x. Since nodes u with $u.\textsc{subroot} = k + 1$ are no longer eligible for linking, they do not need to hold any credits. This is actually a safe change to make for all the inequalities—it only possibly frees up more credit to be transferred to nodes that gain credit during a transition. It also holds for $k > 2$, but would have made the math above messier. We need only show then that the fourth transition does not increase the total amount of credit for $k = 2$ with this change. The only possible case for this transition is when $i = 2$. The corresponding inequality is

$$\begin{aligned} 2ab^x + a^2b^x &\geq ab^{x+1} + a^2b^{x+1} \\ 2 + a &\geq b + ab \\ 2 + \frac{3}{2} &\geq \frac{5}{4} + \left(\frac{3}{2}\right)\left(\frac{5}{4}\right) \\ \frac{7}{2} &> \frac{25}{8} \end{aligned}$$

so the transition does not increase the total amount of credit for $k = 2$.

We now have that any node u such that $u.\textsc{subroot} = k+1$ holds 0 credits and attained its maximum rank while it was still a root in the k^{th} partition. Furthermore, any root in the i^{th} partition with rank x must hold $a^{k+1-i}b^x$ credits. Note that whenever credits are transferred away from a node u, they are transferred to a node v such that $v.\textsc{rank} > u.\textsc{rank}$ or $v.\textsc{subroot} < u.\textsc{subroot}$. Then credit held by a root u in the i^{th} partition of rank x can never be held by another such root v. Therefore, at most $(k/k+1)^{k+1-i}\left(\frac{k^k}{k^k+1}\right)^x \cdot n$ nodes u can ever have $u.\textsc{subroot} = i$ and $u.\textsc{rank} = x$. Note that there can then be at most $(k/k+1)\left(\frac{k^k}{k^k+1}\right)^x \cdot n$ nodes u such that $u.\textsc{subroot} = k+1$ and $u.\textsc{rank} = x$.

Then summing over these values, there can be at most

$$\left(\frac{k}{k+1}\right)\left(\frac{k^k}{k^k+1}\right)^x \cdot n + \sum_{i=1}^{k}\left(\frac{k}{k+1}\right)^{k+1-i}\left(\frac{k^k}{k^k+1}\right)^x \cdot n$$
$$=\left(k + \frac{k}{k+1} - k\left(\frac{k}{k+1}\right)^k\right)\left(\frac{k^k}{k^k+1}\right)^x \cdot n$$

nodes u with $u.\text{RANK}_{max} = x$. □

Lemma 4.2.2. *Using nested linking by rank, the total number of rank ties,* $\sum_{u \in N} u.\text{TIES}_{acc}$ *is* $O(n)$.

Proof. The initial forest consists of singletons, so initially there are 0 rank ties. Path compaction will not introduce any new rank ties, meaning only linking can contribute to the sum. Let us then consider the ways that linking could increase $u.\text{TIES}$. When u loses a link, it is possible that it gains new ancestors; however, its new parent is guaranteed by the algorithm to have greater rank. When u wins a link, it does not gain any new ancestors, but its rank can increase, causing a rank tie. In the case that increasing u's rank causes $u.\text{RANK} = u.\text{PARENT.RANK}$ such that $u.\text{SUBROOT} > u.\text{PARENT.SUBROOT}$, the algorithm stops. This potentially adds more than 1 to $u.\text{TIES}$. Luckily, u can only gain up to k ancestors of equal rank at a given time—the algorithm only allows $u.\text{RANK} = u.\text{PARENT.RANK}$ if $u.\text{SUBROOT} > u.\text{PARENT.SUBROOT}$, and for all v, $v.\text{SUBROOT} \leq k+1$. Thus $u.\text{TIES}$ can increase by at most k with each rank increase, so $u.\text{TIES}_{acc} \leq k u.\text{RANK}_{max}$. Summing over all nodes we obtain $k \sum_{u \in N} u.\text{RANK}_{max} = \text{O}(n)$ by Lemma 4.2.1. □

Lemma 4.2.3. *The ranks maintained by nested linking by rank satisfy Properties 1, 2, 3, and 5. Thus it is a strongly well-behaved rank function.*

Proof. The only changes the algorithm makes to the ranks is to increment them, thus ensuring Property 1 holds. Whenever increasing a node's rank, the algorithm guarantees that it is no greater than its parent's rank. Since all compacted sequences are paths, Property 2 holds. Properties 3 and 5 follow directly from Lemmas 4.2.1 and 4.2.2 respectively. □

Now that we have proof of a well-behaved rank function for nested linking by rank, we can state two theorems of optimality, the truth of which follows directly from Corollary 4.0.1 and Lemmas 4.1.1, 4.1.2, and 4.2.3.

Theorem 4.2.4. *For arbitrary constant k, if* FIND *operations are performed with segmented compression, and* UNITE *operations are performed with nested linking by rank, then a sequence of m intermixed operations takes $O(m\alpha(n, m/n))$ time.*

Theorem 4.2.5. *For $k = 2$, if* FIND *operations are performed with segmented compression or segmented splitting, and* UNITE *operations are performed with nested linking by rank, then a sequence of m intermixed operations takes $O(m\alpha(n, m/n))$ time.*

4.2.2 Nested Linking by Size

There is not necessarily anything inherently unique about nested linking by rank—the nested problem still leaves plenty of flexibility for alternate solutions. It is simple enough to adapt linking by size for the restricted case of two partitions (though extending it to many partitions may prove much more complicated).

As in the case of linking by size, *nested linking by size* maintains for each node u a size field u.SIZE. This serves as an upper bound on but may not be precisely equal to the number of nodes in u's subtree. When linking two nodes, the node with greater size wins, and the loser's size is added to the winner's. In order to maintain the invariant that u.PARENT.SIZE $\geq 2u$.SIZE for all nodes, we need to make a further adjustment for the case of subroot linking. Consider a subroot u which has just won a link and had its size increased. If u.PARENT is not a root in the first partition, then we can shortcut and set u.PARENT to be u.PARENT^2 to preserve the invariant while maintaining the partitions. In the case that u.PARENT is a root in the first partition, we cannot shortcut without destroying the partition. Instead, we add *twice* the loser's size to u.PARENT.SIZE. This preserves

the invariant, and does not cause further issues since u.PARENT does not have any other ancestors. See Algorithm 4.2.2 for an implementation.

Algorithm 4.2.2 UNITE with Nested Linking by Size

```
function UNITE(i, x, y)
    if i > 1 then
        u ← FIND(i − 1, x)
        v ← FIND(i − 1, y)
        if u ≠ v then
            return error
    u ← FIND(i, x)
    v ← FIND(i, y)
    if u = v then
        return false
    if u.SIZE < v.SIZE then
        u ↔ v
    v.PARENT ← u
    v.SUBROOT ← i + 1
    u.SIZE ← u.SIZE + v.SIZE
    if u.SUBROOT > 1 then
        if u.PARENT = u.PARENT^2 then
            u.PARENT.SIZE ← u.PARENT.SIZE + 2v.SIZE
        else
            u.PARENT ← u.PARENT^2
    return true
```

We will use the same rank function as for linking by size: let $u.\text{RANK} = \lfloor \lg u.\text{SIZE} \rfloor$. Since the algorithm guarantees that $u.\text{PARENT.SIZE} \geq 2u.\text{SIZE}$, it also implies $u.\text{PARENT.RANK} > u.\text{RANK}$, which gives us no rank ties and non-decreasing ranks along paths. Proving that the ranks decay exponentially is a bit more complicated, so we will use a credit-based argument similar to that of nested linking by rank. Luckily, the argument is simpler since we need only address two partitions.

Lemma 4.2.6. *Using nested linking by size, at most* $11/2 \cdot n \cdot (3/4)^x$ *nodes* u *have* $u.\text{RANK}_{max} = x$.

Proof. We require a root with size x to hold $\text{POTENTIAL}_1(x) = x^b$ credits, a subroot with size

y to hold $\textsc{Potential}_2(y) = a \cdot y^b$ credits, and a non-subroot to hold zero credits. Initially each node holds one credit, so the total credit in the system is n. We show that each UNITE either preserves or decreases the credit in the system. The key constraints imposed by the different types of UNITE operations are

$$\textsc{Potential}_1(x) + \textsc{Potential}_1(y) \geq \textsc{Potential}_2(x) + \textsc{Potential}_1(x+y)$$

if $x \leq y$, and if $x \leq y \leq z$,

$$\begin{aligned} &\textsc{Potential}_2(x) + \textsc{Potential}_2(y) + \textsc{Potential}_1(z) \\ &\quad \geq \textsc{Potential}_2(x+y) + \textsc{Potential}_1(2x+z). \end{aligned}$$

A very useful approximation will be $(x+y)^b \leq \big(2^b - 1\big)x^b + y^b$ for $x \leq y$ (this is tight for $x = y$). Substituting for the definitions of our potential functions in the first constraint, we find

$$\begin{aligned} x^b + y^b &\geq ax^b + \big(2^b - 1\big)x^b + y^b \geq ax^b + (x+y)^b \\ x^b &\geq ax^b + \big(2^b - 1\big)x^b \\ 2 - 2^b &\geq a \end{aligned}$$

Now working from the second constraint, we get the following.

$$\begin{aligned} ax^b + ay^b + z^b &\geq a\big(2^b - 1\big)x^b + ay^b + \big(2^b - 1\big)(2x)^b + z^b \geq a(x+y)^b + (2x+z)^b \\ ax^b &\geq a\big(2^b - 1\big)x^b + \big(2^b - 1\big)(2x)^b \\ a &\geq a\big(2^b - 1\big) + \big(2^b - 1\big)2^b \\ a\big(2 - 2^b\big) &\geq 2^{2b} - 2^b \\ a &\geq \frac{2^{2b} - 2^b}{2 - 2^b} \end{aligned}$$

Note that these two inequalities are satisfied for $b = \lg(4/3)$ and $a = 4/9$.

Now suppose a root u has rank x. Then $u.\textsc{size} \geq 2^x$, which implies that it holds at least $2^{xb} = (4/3)^x$ credits. Thus at most $n \cdot (3/4)^x$ roots may attain rank x. Similarly a subroot of rank x must hold at least $4/9 \cdot (4/3)^x$ credits, so there may be at most $9/4 \cdot n \cdot (3/4)^x$

subroots that attain rank x. A node's rank does not change after it ceases to be a subroot, so there can similarly be at most $9/4 \cdot n \cdot (3/4)^x$ non-subroots that attain rank x. As in the case of nested linking by rank, the credit transfer rules impose a partial order on node types, where credit is transferred only to greater types. The lemma holds. □

Lemma 4.2.7. *The ranks defined for nested linking by size satisfy Properties 1, 2, 3, and 5. Thus it is a strongly well-behaved rank function.*

Proof. The only changes that linking by size makes to a node's rank are through increasing the size field. Since $\lfloor \lg u.\text{SIZE} \rfloor$ is a monotonically increasing function, this satisfies Property 1. The balance property $u.\text{PARENT.SIZE} \geq 2u.\text{SIZE}$ implies that

$$\begin{aligned} u.\text{PARENT.RANK} &= \lfloor \lg u.\text{PARENT.SIZE} \rfloor \\ &\geq \lfloor \lg(2u.\text{SIZE}) \rfloor = \lfloor \lg u.\text{SIZE} + 1 \rfloor \\ &> \lfloor \lg u.\text{SIZE} \rfloor = u.\text{RANK}. \end{aligned}$$

This implies strictly increasing ranks along paths. Since each compacted sequence is a path, Property 2 is satisfied. Similarly, there are no rank ties, satisfying Property 3. Property 5 follows directly from Lemma 4.2.6. The lemma holds. □

Now that we have proof of a well-behaved rank function for nested linking by size, we can state a theorem of optimality, the truth of which follows directly from Corollary 4.0.1 and Lemmas 4.1.1, 4.1.2, and 4.2.7.

Theorem 4.2.8. *For $k = 2$, if* FIND *operations are performed with segmented compression or segmented splitting, and* UNITE *operations are performed with nested linking by size, then a sequence of m intermixed operations takes $O(m\alpha(n, m/n))$ time.*

4.2.3 Randomized Nested Linking

To provide another alternative, we also show how to adapt randomized linking to the restricted case of two partitions. Very little change is required. *Randomized nested linking*

consists of labeling the nodes with a uniformly random permutation and then performing *nested linking by index* with the labels. When linking two roots, the one with greater label is chosen as the winner, and the loser u has $u.\textsc{subroot}$ incremented. See Algorithm 4.2.3 for an implementation of nested linking by index.

One important caveat is that our analysis requires that subroot linking occurs directly below the root, a property that segmented splitting does not have. Therefore our proof of optimality only holds for the use of randomized nested linking in conjunction with segmented compression.

Algorithm 4.2.3 UNITE with Nested Linking by Index

```
function UNITE(i, x, y)
    if i > 1 then
        u ← FIND(i − 1, x)
        v ← FIND(i − 1, y)
        if u ≠ v then
            return error
    u ← FIND(i, x)
    v ← FIND(i, y)
    if u = v then
        return false
    if u < v then
        u ↔ v
    v.PARENT ← u
    v.SUBROOT ← i + 1
    return true
```

To prove that randomized nested linking admits well-behaved ranks, we will use the same rank function as for randomized linking: let $u.\textsc{rank} = \lfloor \lg n \rfloor - \lfloor \lg(n - u + 1) \rfloor$. The approach to the proof is modeled after that presented in Section 3.2.3.

Let u be any node, and let SEQ be the sequence of UNITE operations which build the final sets containing u, ignoring any UNITE operations that return **false** or **error**. Let CSEQ be the subsequence that builds the final set in the first partition containing u, and let FSEQ be that which builds the final set in the second partition containing u. As in the previous

argument, we will reorder these sequences into CSeq(u) and FSeq(u) so that each adds a single element at a time to the sets containing u in the first and second partitions respectively.

Let $u_0 = u, u_1, u_2, \ldots$ be the successive vertices added by CSeq(u) to the set in the first partition containing u, and the vertices added by FSeq(u) to the set in the second partition containing u be $v_0 = u, v_1, v_2, \ldots$. We call u_j (respectively v_j) a *prefix maximum* if $u_i < u_j$ (respectively $v_i < v_j$) for all $i < j$. Lemma 4.2.9, which follows, is essentially equivalent to Lemma 3.2.6, so we omit the proof. We prove an equivalent statement for FSeq(u), Lemma 4.2.10. Lemmas 4.2.11 and 4.2.12 below are equivalent to Lemma 3.2.7, so the proofs are likewise omitted.

Lemma 4.2.9. *If* CSeq(u) *is performed in isolation using randomized nested linking and segmented compression, the ancestors of u are exactly the prefix maxima among the u_j.*

Lemma 4.2.10. *If* FSeq(u) *is performed after* CSeq(u) *using randomized linking and segmented compression, the new ancestors of u introduced by* FSeq(u) *are exactly the prefix maxima among the v_j.*

Proof. Before two sets are linked, their respective subtrees are segment compressed. Segmented compression cannot introduce any new ancestors. Following compression the loser of the link is a child of the root, and the winner is either another child of the root or the root itself. If it is the root, then the Unite does not introduce any new ancestors above u. Otherwise, the only possible new ancestor added is the root of the other subtree, say w. By the properties of randomized nested linking if w wins the link, then w is greater than any element in the resulting set containing u, and by the properties of FSeq(u), it must be a prefix maximum.

Conversely, suppose v_j is a prefix maximum. Then when v_j is linked to the set containing u, it is the root of its own set in the second partition by the properties of FSeq(u). Since it is a prefix maximum, it is greater than the root of the set containing u and becomes the new root of the set, and thus an ancestor of u. □

Lemma 4.2.11. *Let* $\textsc{Unite}(0, y, z)$ *be the last* Unite *in* CSeq*, and let* $\textsc{CSeq}'$ *be* CSeq *with this* Unite *deleted. Let* S_1 *containing* y *and* S_2 *containing* z *be the two sets built by* $\textsc{CSeq}'$*, and let* $\textsc{CSeq}_1$ *and* $\textsc{CSeq}_2$ *be the subsequences of* $\textsc{CSeq}'$ *that build* S_1 *and* S_2*, respectively. If* u *is in* S_1*,* $\textsc{CSeq}(u)$ *is* $\textsc{CSeq}_1(u)$ *followed by* $\textsc{Unite}(0, y, z)$ *followed by* $\textsc{CSeq}_2(z)$*; if* u *is in* S_2*,* $\textsc{CSeq}(u)$ *is* $\textsc{CSeq}_2(u)$ *followed by* $\textsc{Unite}(0, y, z)$ *followed by* $\textsc{CSeq}_1(y)$*.*

Lemma 4.2.12. *Let* $\textsc{Unite}(1, y, z)$ *be the last* Unite *in* FSeq*, and let* $\textsc{FSeq}'$ *be* FSeq *with this* Unite *deleted. Let* S_1 *containing* y *and* S_2 *containing* z *be the two sets built by* $\textsc{FSeq}'$*, and let* $\textsc{FSeq}_1$ *and* $\textsc{FSeq}_2$ *be the subsequences of* $\textsc{FSeq}'$ *that build* S_1 *and* S_2*, respectively. If* u *is in* S_1*,* $\textsc{FSeq}(x)$ *is* $\textsc{FSeq}_1(u)$ *followed by* $\textsc{Unite}(1, y, z)$ *followed by* $\textsc{FSeq}_2(z)$*; if* u *is in* S_2*,* $\textsc{FSeq}(u)$ *is* $\textsc{FSeq}_2(u)$ *followed by* $\textsc{Unite}(1, y, z)$ *followed by* $\textsc{FSeq}_1(y)$*.*

We now have the machinery necessary to prove that the ancestors of u in Seq are a subset of those produced by $\textsc{CSeq}(u)$ followed by $\textsc{FSeq}(u)$.

Lemma 4.2.13. *Every ancestor of* u *throughout the sequence of operations* Seq *is at some point an ancestor of* u *in the sequence* $\textsc{CSeq}(u)$ *followed by* $\textsc{FSeq}(u)$*.*

Proof. The proof is by induction on the length of Seq using Lemmas 4.2.11 and 4.2.12. Let $\textsc{Unite}(i, y, z)$ be the final Unite in Seq, and $\textsc{Seq}'$ be the sequence with this Unite deleted. We proceed in cases by the value of i.

First suppose $i = 1$, and the final Unite occurs between two sets in the first partition. Let S_1 containing y and S_2 containing z be the two coarse sets built by $\textsc{Seq}'$ that are joined by the Unite, and let $\textsc{Seq}_1$ and $\textsc{Seq}_2$ be the subsequences of $\textsc{Seq}'$ that build S_1 and S_2, respectively. Assume u is in S_1; the argument is symmetric if u is in S_2. Let v and w be the largest elements in S_1 and S_2, respectively. If $v > w$, then no node in S_2 becomes an ancestor of u in either sequence, so the lemma holds. If $v < w$, w becomes an ancestor of u under Seq, but no other node in S_2 does. In this case w is maximum in $S_1 \cup S_2$, hence

a prefix maximum in CSeq(u) and therefore an ancestor of u in the reordered sequence by Lemma 3.2.6, so the lemma holds in this case also.

Now suppose $i = 2$, and the final Unite occurs between two sets in the second partition. Let S_1 containing y and S_2 containing z be the two fine sets built by Seq$'$ that are joined by the Unite, and let Seq$_1$ and Seq$_2$ be the subsequences of Seq$'$ that build S_1 and S_2, respectively. Assume u is in S_1; the argument is symmetric if u is in S_2. Let v and w be the largest elements in S_1 and S_2, respectively. If $v > w$, then no node in S_2 becomes an ancestor of u in either sequence, so the lemma holds. Now, if $v < w$, the subtree containing u is linked underneath w, so the ancestor set could conceivably change considerably; however, with segmented compression, w is at this point either the root of the entire tree or a child of said root. Thus, since u is in the the same tree, all proper ancestors of w are already shared by u. Then w becomes an ancestor of x under Seq, but no other node in S_2 does. In this case w is maximum in $S_1 \cup S_2$, hence a prefix maximum in FSeq(u) and therefore an ancestor of u in the reordered sequence by Lemma 4.2.10, so the lemma holds in this case also.

By the induction hypothesis every ancestor of u in Seq is an ancestor of u in CSeq(u) followed by FSeq(u). □

Lemma 4.2.14. *With randomized nested linking and segmented compression, the expected value of u.ties$_{acc}$ is at most four.*

Proof. By Lemmas 4.2.9, 4.2.10, and 4.2.13, it suffices to bound the expected number of prefix maxima following u of rank equal to u.rank. By the same argument as in the proof of Lemma 3.2.10, each sequence CSeq(u) and FSeq(u) is expected to produce at most two such ancestors, thus the lemma holds. □

Lemma 4.2.15. *The ranks defined for nested randomized linking satisfy Properties 1, 2, and 5. They also satisfy Property 3 in expectation. Thus in expectation it is a strongly well-behaved rank function.*

Proof. Property 1 is immediate from the definition of fixed ranks. Property 2 follows from the fact that the algorithm ensures strictly increasing labels along paths and Property 9. Properties 3 and 5 follow directly from Lemma 4.2.14 and Property 11 respectively. □

Now that we have proof of a well-behaved rank function for nested randomized linking, we can state a theorem of optimality, the truth of which follows directly from Corollary 4.0.1 and Lemmas 4.1.1 and 4.2.15.

Theorem 4.2.16. *For $k = 2$, if* FIND *operations are performed with segmented compression and* UNITE *operations are performed with nested randomized linking, then a sequence of m intermixed operations takes $O(m\alpha(n, m/n))$ time.*

4.3 Remarks

We have proved that segmented compression and nested linking by rank can be used to solve the problem in optimal time up to constant factors for any constant number of partitions. Additionally, we have given a one-pass compaction method and two alternative linking rules for the case of two partitions. It is entirely possible that segmented splitting and nested linking by size could scale up to handle any constant number of partitions, but the implementation of the former and the analysis of both would likely become increasingly complicated.

We allowed limited rank ties in our algorithm for nested linking by rank. We do not know how to make the analysis work without them. It remains an open problem whether the rank ties are strictly necessary, or if a similar algorithm could be developed without them.

We have no reason to believe that randomized nested linking is incompatible with compaction methods other than segmented compression or that it cannot handle more than two partitions; however, our analysis techniques seem insufficient for proving that the number of rank ties remains suitably bounded in these settings. As in the case of disjoint set union,

we do not advocate direct implementation of randomized nested linking, but rather have shown that it is safe to use nested linking by index when the input is already sufficiently randomized.

One important thing to note is that using any of these algorithms to solve the nested problem presents a constant-factor time-space trade-off in comparison with storing each partition separately. In the worst case, our time bounds are roughly a factor k greater than those for the corresponding algorithms for a single partition; however, they are guaranteed to save roughly a factor k in space. In practice, it seems unlikely that the worst-case time trade-off would be seen, and additional benefits may be seen in caching behavior due to the reduced data footprint.

Chapter 5

Conclusion

The realization that he was utterly powerless was like the blow of a sledgehammer, yet it was curiously calming as well.

Milan Kundera, *The Unbearable Lightness of Being*

To bring this work to a close, we observe that even though it is a technique that has been studied for over five decades, there is still much we do not know about the compressed tree method. The Sledgehammer Theorem (2.3.2) is certainly not the final word on the matter.

It does not seem to apply to *every* optimal algorithm using the compressed tree method, just a lot of them. This seems to stem from the fact that our definitions of well-structured compaction and well-behaved rank functions do not encapsulate the necessary conditions for optimal behavior, only some sufficient ones. *Are there general conditions that are both necessary and sufficient for optimal behavior?*

Our analysis of randomized linking shows that linking by index does not just behave as well or better than linking by rank and size in a randomized setting by chance, it does so provably with high probability. We offered experimental evidence that the simpler randomized method of coin-flip linking probably has near-logarithmic expected performance on

some inputs, but we have not been able to prove it. *Does flipping a coin not work?*

In proposing solutions to the nested set union problem, we have given a theoretical time-space trade-off. This has some intellectual value, but we do not know if it has practical merit. To our knowledge, no experiments have been done on the problem, and we only know of one application for the problem. *Can this trade-off help real-world algorithms?*

We have not made any measurable, direct progress on the path evaluation problem. It remains that the best known bound is super-inverse-Ackermann, and that the only proposed solution with claimed inverse-Ackermann complexity has potentially serious issues that have prevented proper assessment. *Can our work in Chapter 2 help make progress on this problem in the future?*

Despite these open questions, we clearly have a powerful tool at our disposal.

We have a sledgehammer.
Let us keep watch for nails.

References

[1] Wilhelm Ackermann. Zum hilbertschen aufbau der reellen zahlen. *Mathematische Annalen*, 99(1):118–133, 1928.

[2] Noga Alon, Oded Goldreich, Johan Håstad, and René Peralta. Simple construction of almost k-wise independent random variables. *Random Struct. Algorithms*, 3(3):289–304, 1992.

[3] Stephen Alstrup, Inge Li Gørtz, Theis Rauhe, Mikkel Thorup, and Uri Zwick. Union-find with constant time deletions. In *Proc. 32nd Annual International Colloquium on Automata, Languages and Programming*, pages 78–89, 2005.

[4] Cecilia R. Aragon and Raimund G. Seidel. Randomized search trees. In *Proc. 30th Annual Symposium on Fundamentals of Computer Science*, pages 540–545, 1989.

[5] Adam L. Buchsbaum, Loukas Georgiadis, Haim Kaplan, Anne Rogers, Robert E. Tarjan, and Jeffery Westbrook. Linear-time algorithms for dominators and other path-evaluation problems. *SIAM J. Computing*, 38(4):1533–1573, 2008.

[6] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[7] Rodney Farrow. Efficient on-line evaluation of functions defined on paths in trees. Technical Report 476-093-17, Rice University, 1977.

[8] Loukas Fraczak, Wojciechand Georgiadis, Andrew Miller, and Robert E. Tarjan. Finding dominators via disjoint set union. *J. Discrete Algorithms*, 23:2–20, 2013.

[9] 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.

[10] Bernard A. Galler and Michael J. Fisher. An improved equivalence algorithm. *Commun. ACM*, 7(5):301–303, 1964.

[11] 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.

[12] John E. Hopcroft and Jeffrey D. Ullman. Set merging algorithms. *SIAM J. Computing*, 2(4):294–303, 1973.

[13] Haim Kaplan, Nira Shafrir, and Robert E. Tarjan. Union-find with deletions. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 19–28, 2002.

[14] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms, Third Edition*. Addison-Wesley, 1997.

[15] Dexter C. Kozen. *The Design and Analysis of Algorithms*. Springer-Verlag, 1992.

[16] Joseph 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.

[17] Daniel H. Larkin and Robert E. Tarjan. Nested set union. In *Proc. 22nd Annual European Symposium on Algorithms*, pages 618–629, 2014.

[18] Md. Mostafa Ali Patwary. Experiments on union-find algorithms for the disjoint-set data structure. In *Proc. 9th Annual International Symposium on Experimental Algorithms*, pages 411–423, 2010.

[19] Rozsa Péter. *Rekursive funktionen.* Académiai Kiadó, 1951.

[20] Raimund Seidel and Micha Sharir. Top-down analysis of path compression. *SIAM J. Computing*, 34(3):515–525, 2005.

[21] Robert E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(7):215–225, 1975.

[22] Robert E. Tarjan. Applications of path compression on balanced trees. *J. ACM*, 26(4):690–715, 1979.

[23] Robert E. Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *J. Computer and Systems Science*, 18(2):110–127, 1979.

[24] Robert E. Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. *J. ACM*, 31(2):245–281, 1984.

[25] Theodorus P. van der Wiede. *Datastructures: An Axiomatic Approach and the Use of Binomial Trees in Developing and Analyzing Algorithms.* Mathematisch Centrum, 1980.

[26] Jan van Leeuwen and Theodorus P. van der Wiede. Alternative path compression techniques. Technical Report RUU-CS-77-3, Rijksuniversiteit Utrecht, 1977.

By the end of my Ph.D., I could swing a sledgehammer.

Jocelyn Bell Burnell