Union-Find with Constant Time Deletions

Stephen Alstrup[1], Inge Li Gørtz[1], Theis Rauhe[1],
Mikkel Thorup[2], and Uri Zwick[3]

[1] Department of Theoretical Computer Science, IT University of Copenhagen, Denmark. E-mail: {stephen,inge,theis}@itu.dk.
[2] AT&T Research Labs, USA. E-mail: mthorup@research.att.com.
[3] School of Computer Science, Tel Aviv University, Israel. E-mail: zwick@cs.tau.ac.il.

Abstract. A union-find data structure maintains a collection of disjoint sets under *makeset*, *union* and *find* operations. Kaplan, Shafrir and Tarjan [SODA 2002] designed data structures for an extension of the union-find problem in which elements of the sets maintained may be deleted. The cost of a *delete* operation in their implementations is the same as the cost of a *find* operation. They left open the question whether *delete* operations can be implemented more efficiently than *find* operations. We resolve this open problem by presenting a relatively simple modification of the classical union-find data structure that supports *delete*, as well as *makeset* and *union*, operations in *constant* time, while still supporting *find* operations in $O(\log n)$ worst-case time and $O(\alpha(n))$ amortized time, where n is the number of elements in the set returned by the *find* operation, and $\alpha(n)$ is a functional inverse of Ackermann's function.

1 Introduction

A union-find data structure maintains a collection of disjoint sets under the operations *makeset*, *union* and *find*. A *makeset* operation generates a singleton set. A *union* operation takes two sets and unites them, destroying the two original sets. A *find* operation takes an element and returns a reference to the set currently containing it. The union-find problem is one of the most fundamental data structure problems. It has many applications in a wide range of areas. For an extensive list of such applications, and for a wealth of information on the problem and many of its variants, see the survey of Galil and Italiano [7].

An extremely simple union-find data structure (attributed by Aho *et al.* [1] to McIlroy and Morris), which employs two simple heuristics, *union by rank* and *path compression*, was shown by Tarjan [12] (see also Tarjan and van Leeuwen [13]) to be amazingly efficient. It performs a sequence of M *find* operations and N *makeset* and *union* operations in $O(N+M\,\alpha(M,N))$ total time. Here $\alpha(\cdot,\cdot)$ is an extremely slowly growing functional inverse of Ackermann's function. In other words, the *amortized* cost of each *makeset* and *union* operation is $O(1)$, while the amortized cost of each *find* operation is $O(\alpha(M+N,N))$, only marginally more than a constant. Fredman and Saks [6] obtained a matching lower bound

in the cell probe model of computation, showing that this simple data structure is essentially optimal in the amortized setting.

The union by rank heuristics on its own implies that *find* operations take $O(\log n)$ worst-case time. Here n is the number of elements in the set returned by the *find* operation. All other operations take constant worst-case time. It is possible to trade a slower *union* for a faster *find*. Smid [11], building on a result of Blum [4], gives for any k a data structure that supports *union* operations in $O(k)$ time and *find* operations in $O(\log_k n)$ time. When $k = \log n / \log\log n$, both the *union* and *find* operation take $O(\log n / \log\log n)$ time. Fredman and Saks [6] (see also Ben-Amram and Galil [3]) again show that this tradeoff is optimal, establishing an interesting gap between the amortized and worst-case complexities of the union-find problem. Alstrup *et al.* [2] present union-find algorithms with simultaneously optimal amortized and worst-case bounds.

Local amortized bounds As noted by Kaplan *et al.* [8], the standard amortized bounds for *find* are global in terms of the total number N of elements ever created whereas the worst-case bounds are local in terms of the number n of elements in the current set we are finding. Obviously n may be much smaller than N. To state more local amortized bounds, we need a non-standard parameterization of the inverse Ackermann function. For integers $k \geq 0$ and $j \geq 1$, define an Ackermann function $A_k(j)$ as follows

$$A_k(j) = \begin{cases} j+1 & \text{if } k = 0, \\ A_{k-1}^{(j+1)}(j) & \text{if } k \geq 1. \end{cases}$$

Here $f^{(i)}(x)$ is the function f iterated i times on x. Now, define the inverse of the function $\bar{\alpha}(j, i)$, for integer $i, j \geq 0$, as

$$\bar{\alpha}(j, i) = \min\{k \geq 2 \mid A_k(j) > i\}.$$

(For a technical reason, $\bar{\alpha}(j, i)$ is defined to be at least 2 for every $i, j \geq 0$.) Relating to the standard definition of α, we have $\alpha(M, N) = \Theta(\bar{\alpha}(\lceil M/N \rceil, N))$. Kaplan *et al.* [8] present a refined analysis of the classical union-find data structure showing that the amortized cost of *find*(x) operation is only $O(\bar{\alpha}(\lceil (M + N)/N \rceil, n)$. Kaplan *et al.* [8] state their results equivalently in terms of a three parameter function that we will not define here. To get a purely local amortized cost for *find*, we note that $\bar{\alpha}(\lceil (M+N)/N \rceil, n) \leq \bar{\alpha}(1, n) = O(\alpha(n, n)) = O(\alpha(n))$.

Union-Find with Deletions In the traditional version of the union-find problem elements are created using *makeset* operations. Once created, however, elements are never destroyed. Kaplan *et al.* [8] consider a very natural extension of the union-find problem in which elements may be *deleted*. We refer to this problem as the *union-find with deletions* problem, or *union-find-delete* for short.

Using relatively straightforward ideas (see, e.g., [8]) it is possible to design a union-find-delete data structure that uses only $O(N^*)$ space, handles *makeset*, *union* and *delete* operations in $O(1)$ worst-case time, and *find* operations

in $O(\log N^*)$ worst-case time and $O(\alpha(N^*))$ amortized time, where N^* is the current number of elements in the whole data structure. The challenge in the design of union-find-delete data structures is to have the time of a *find*(x) operation depend on n, the size of the set currently containing x, and not on N^*, the total number of elements currently contained in all the sets.

Using an incremental background rebuilding technique for each set, Kaplan *et al.* [8] describe a way of converting any data structure for the classical union-find problem into a union-find-delete data structure. The time bounds for *makeset*, *find* and *union* operations change by only a constant factor, while the time needed for a *delete*(x) operation is the same as the time needed for a *find*(x) operation followed by a *union* operation with a singleton set. As a *union* operation is usually much cheaper than a *find* operation, Kaplan *et al.* [8] thus show that in both the amortized and the worst-case settings, a *delete* operation is not more expensive than a *find* operation. Combined with their refined amortized analysis of the classical union-find data structure, this provides, in particular, a union-find-delete data structure that implements *makeset* and *union* operations in $O(1)$ time, and *find*(x) and *delete*(x) operations in $O(\alpha(n))$ amortized time and $O(\log n)$ worst-case time. They leave open, however, the question whether *delete* operations can be implemented *faster* than *find* operations.

Our results We solve the major open problem raised by Kaplan *et al.* [8] and show that *delete* operations can be performed in *constant* worst-case time, while still keeping the $O(\bar{\alpha}(\lceil (M+N)/N \rceil, n)) = O(\alpha(n))$ amortized cost and the $O(\log n)$ worst-case cost of *find* operations, and the constant worst-case cost of *makeset* and *union* operations. We recall here that N is the total number of elements ever created, M is the total number of *find* operations performed, and n is the number of elements in the set returned by the *find* operation. The data structure that we present uses linear space and is a relatively simple modification of the classical union-find data structure. It is at least as simple as the data structures presented by Kaplan *et al.* [8].

As a by-product we also obtain a very concise potential-based proof of the $O(\bar{\alpha}(\lceil (M+N)/N \rceil, n))$ bound, first obtained by Kaplan *et al.* [8], on the amortized cost of a *find* operation in the classical setting. We believe that our potential-based analysis is much simpler than the one given by Kaplan *et al.* [8].

Our techniques Our new union-find-delete data structure, like most other union-find data structures, maintains the elements of each set in a rooted tree. As elements can now be deleted, not all the nodes in these trees will contain active elements. Nodes that contain elements are said to be *occupied*, while nodes that do not contain elements are said to be *vacant*. When an element is deleted, the node containing it becomes vacant. If proper measures are not taken, then a tree representing a set may contain too many vacant nodes. As a result, the space needed to store the tree, and the time needed to process a *find* operation may become too large. The new data structure uses a simple collection of local operations to *tidy up* a tree after each delete operation. This ensures that at

most half of the nodes in a tree are vacant. More importantly, the algorithm employs local constant-time *shortcut* operations in which the grandparent, or a more distant ancestor, of a node becomes its new parent. These operations, which may be viewed as a local constant-time variant of the path compression technique, keep the trees relatively shallow to allow fast *find* operations.

As with the simple standard union-find, the analysis is the most non-trivial part. The analysis of the new data structure uses two different potential functions. The first potential function is used to bound the *worst-case* cost of *find* operations. Both potential functions are needed to bound the *amortized* cost of *find* operations. The second potential function on its own can be used to obtain a simple derivation of the refined amortized bounds of Kaplan *et al.* [8] for union-find without deletions.

We end this section with a short discussion of the different techniques used to analyze union-find data structures. The first tight amortized analysis of the classical union-find data structure, by Tarjan [12] and Tarjan and van Leeuwen [13], uses *collections of partitions* and the so-called *accounting method*. The refined analysis of Kaplan *et al.* [8] is directly based on this method. A much more concise analysis of the union-find data structure based on potential functions can be found in Kozen [9] and Chapter 21 of Cormen *et al.* [5]. The amortized analysis of our new union-find-delete data structure is based on small but crucial modifications of the potential function used in this analysis. As a by product we get, as mentioned above, a simple proof of the amortized bounds of Kaplan *et al.* [8]. Seidel and Sharir [10] presented recently an intriguing top-down amortized analysis of the union-find data structure. Our analysis is no less concise, though perhaps less intuitive, and has the additional advantage of bounding the cost of an amortized operation in terms of the size of the set returned by the operation.

2 Preliminaries

The union-find and union-find-delete problems A classical union-find data structure supports the following operations:

- *make-set(x)*: Create a singleton set containing x.
- *union(A,B)*: Combine the sets A and B into a new set, destroying A and B.
- *find(x)*: Return an identifier of the set containing x.

The only requirement from the identifier, or name, returned by a *find* operation is that if two elements x and y are currently contained in the same set, then the calls *find*(x) and *find*(y) return the same identifier. Kaplan et al. [8] studied data structures that also support *delete* operations:

- *delete(x)*: Delete x from the set containing it.

A *delete* operation should not change the identifier attached to the set from which the element was deleted. It is important to note that a *delete* operation does not receive a reference to the set currently containing x. It only receives the element x itself. As mentioned, Kaplan et al. [8] essentially showed that *delete* operations are not more expensive than *find* operations.

Standard worst-case bounds for union-find We briefly review here the simple standard union-find data structure that supports *makeset* and *union* operations in constant time and *find* operations in $O(\log n)$ time, as it forms the basis of our new data structure for the union-find-delete problem.

The elements of each set A are maintained in a rooted tree $T = T_A$. The identifier of the set A is the root of T. Fixing some terminology, the *height* of a node $v \in T$, denoted by $h(v)$, is defined to be 0, if v is a leaf, and $\max\{h(w)|$ w is a child of $v\} + 1$, otherwise. Let root(T) denote the root of T. The height of a tree is the height of its root. For a node $v \in T$ let $p(v)$ denote the *parent* of v. A node $x \in T$ is an *ancestor* of a node $y \in T$ if x is on the path from y to the root of T—both y and the root included. A node $x \in T$ is a *descendant* of a node $y \in T$ if y is an ancestor of x.

Each node v has an assigned integer rank *rank*(v). An important invariant is that for the parent of a node always has a strictly higher rank than the node itself. The rank of a tree is defined to be the rank of the root of the tree.

We implement the operations as follows.

find(x): Follow parent pointers from x all the way to the root. Return the root as the identifier of the set.
make-set(x): Create a new node x. Let $p(x) \leftarrow x$, *rank*$(x) \leftarrow 0$.
union(A,B): Recall that A and B are root nodes. Assume w.l.o.g. that *rank*$(A) \geq$ *rank*(B). Make B a child of A. If *rank*(A)=*rank*(B), increase *rank*(A) by one.

Analysis Trivially, *makeset* and *union* operations take constant time. Since ranks are strictly increasing when following parent pointers, the time of a *find* operation applied to an element in a set A is proportional to *rank*(A). We prove, by induction, that *rank*$(A) \leq \log_2 |A|$, or equivalently, that

$$|A| \geq 2^{rank(A)} \ . \tag{1}$$

When A is just created with *make-set(x)*, it has rank 0 and $2^0 = 1$ elements. If C is the set created by *union(A,B)*, then $|C| = |A| + |B|$. If C has the same rank as A, or the same rank as B, we are trivially done. Otherwise, we have *rank*$(A) =$ *rank*$(B) = k$ and *rank*$(C) = k + 1$, and then $|C| = |A| + |B| \geq 2^k + 2^k = 2^{k+1}$. This completes the standard analysis of union-find with worst-case bounds.

3 Augmenting worst-case union-find with deletions

Each set in the data structure is again maintained in a rooted tree. In the standard union-find data structure, reviewed in Section 2, the nodes of each tree were identified with the elements of the set. In the new data structure, elements are attached to nodes, not identified with them. Some nodes in a tree are *occupied*, i.e., have an element attached to them, while others are *vacant*, i.e., have no element attached to them. An element can then be deleted by simply removing it from the node it was attached to. This node then becomes vacant.

The name of a set is taken to be its root node. As the name of a set is a node, and not an element, names do not change as a result of *delete* operations.

An obvious problem with this approach is that if we never remove vacant nodes from the trees, we may end up consuming non-linear space. To avoid this, we require our union-find trees to be *tidy*:

Definition 1. *A tree is said to be* tidy *if it satisfies the following properties:*

- *Every vacant non-root node has at least two children,*
- *Every leaf is occupied and has rank 0.*

It is easy to tidy up a tree. First, we remove vacant leaves. When a node becomes a leaf, its rank is reduced to 0. Next, if a vacant non-root node v has a single child w, we make the parent of v the parent of w and remove v. We call this *bypassing* v. The following Lemma is now obvious.

Lemma 1. *At most half of the nodes in a tidy tree may be vacant.*

Tidy trees thus use linear space. However, tidyness on it own does not yield a sublinear time bound on *find* operations. (Note, for example, that a path of occupied nodes is tidy.) Our next goal would be to make sure that the depth of a tree is logarithmic in the number of occupied nodes contained in it. Ideally, we would want all trees to be *reduced*:

Definition 2. *A tree is said to be* reduced *if it is either*

- *A tree composed of a single occupied node of rank 0, or*
- *A tree of height 1 with a root of rank 1 and occupied leaves of rank 0.*

Naturally, we will not manage to keep our trees reduced at all times. Reduced trees form, however, the base case for our analysis.

Keeping the trees shallow during deletions This section contains our main technical contribution. We show how to implement deletions so that for any set A,

$$|A| \geq (2/3)(6/5)^{rank(A)} \ . \tag{2}$$

Consequently, $rank(A) \leq \log_{6/5}(3|A|/2) = O(\log |A|+1)$. As the rank of a tree is always an upper bound on its height, we thus need to follow at most $O(\log |A|+1)$ parent pointers to get from any element of A to the root identifier.

The key idea is to associate the following value with each node v:

Definition 3. *The value $val(v)$ of a node v is defined as*

$$val(v) = \begin{cases} (5/3)^{rank(p(v))} & \text{if } v \text{ is occupied,} \\ (1/2)(5/3)^{rank(p(v))} & \text{if } v \text{ is vacant.} \end{cases}$$

Here, if v is a root, $p(v) = v$. The value of a set A is defined as the sum the values of all nodes in the tree T_A representing A: $VAL(A) = \sum_{v \in T_A} val(v)$.

The value 5/3 is chosen to satisfy Equation 2 and Lemma 2, 4, and 9 below. In fact, we could have chosen any constant value in $[(1+\sqrt{5})/2, 2)$. We are going to implement deletions in such a way that

$$VAL(A) \geq 2^{rank(A)} \ . \tag{3}$$

Since the tree representing a set A contains exactly $|A|$ occupied nodes, each of value at most $(5/3)^{rank(A)}$, and at most $|A|$ vacant nodes in T_A, each of value at most $(5/3)^{rank(A)}/2$, it will follow that

$$|A| \geq \frac{2^{rank(A)}}{(3/2)(5/3)^{rank(A)}} = (2/3)(6/5)^{rank(A)},$$

so (3) will imply (2).

The essential operation used to keep trees shallow is to *shortcut* from a node v, giving v a parent higher up over v in the tree. For example, path compression shortcuts from all nodes in a search path directly to the root. Since ranks are strictly increasing up through the tree, shortcutting from v increases the value of v by a factor of at least 5/3. This suggests that we can make up for the loss of a deleted node by a constant number of shortcuts from nearby nodes of similar rank. Before proceeding, let us check that reduced trees satisfy (3).

Lemma 2. *If the tree representing a set A is reduced then $VAL(A) \geq 2^{rank(A)}$.*

Proof. If A is of height 0, then $VAL(A) = (5/3)^0 = 1$ and $2^{rank(A)} = 1$. If A is of height 1, then $VAL(A) \geq (5/3)^1 + (1/2)(5/3)^1 = 5/2$ while $2^{rank(A)} = 2$. □

Let us for a moment assume that we have an implementation of *delete* that preserves, i.e., does not decrease, value, and let us check that the other operations preserve (3). A *makeset* operation creates a reduced tree, so (3) is satisfied by Lemma 2. Also, when we set $C := union(A, B)$, we get $VAL(C) \geq VAL(A) + VAL(B)$, and hence (3) follows just like (1).

Paying for a deletion via local rebuilding We now show how we can implement a delete operation in constant time, either without decreasing value of the set from which the element is deleted, or ending up with a reduced tree representing the set. Suppose we delete an element of A attached to a node u. As u becomes vacant, we immediately loose half its value. Before u was vacant the tree was tidy, but now we may have to tidy the tree. If u is not a leaf, the only required tidying up is to bypass u if it has a single child. If instead u was a leaf, we first delete u. If $p(u)$ is now a leaf, its rank is reduced to zero, but that in itself does not affect any value. If $p(u)$ is vacant and now has only one child, we bypass $p(u)$. This completes the tidying up.

Lemma 3. *Let v be the parent of the highest node affected by a* delete*, including tidying up. If rank$(v) = k$, then the maximal loss of value is at most $(9/10)(5/3)^k$.*

Proof. It is easy to see that the worst-case is when $v = p(p(u))$, where u is a deleted leaf and $p(u)$ is bypassed. Now u lost at most $(5/3)^{k-1}$ and $p(u)$ lost $(5/3)^k/2$, while the other child of $p(u)$ gained at least $((5/3)^k - (5/3)^{k-1})/2$ from the bypass. Adding up, the total loss is $(9/10)(5/3)^k$. □

Below we show how to regain the loss from a *delete* using a pointer to v from Lemma 3. To find nearby nodes to shortcut from, we maintain two doubly linked lists for each node v; namely $C(v)$ containing the children of v, and $G(v)$ containing the children of v that themselves have children. Thus, to find a grandchild of v, we take a child of a child in $G(v)$. Both lists are easily maintained as children are added and deleted: if a child u is added to v, it is added to $C(v)$. If u is the first child of v, we add v to $G(p(v))$. Finally, we add u to $G(v)$ if $C(u)$ is non-empty. Deleting a child is symmetric. Using these lists, we first prove

Lemma 4. *In a tidy tree, if node x has rank k and grandchildren, we can gain $\Omega((5/3)^k)$ value in $O(1)$ time.*

Proof. Using $G(x)$, find a child y of x that have children. If y is occupied, we can take any child z of y and shortcut to x. This increases the value of z by at least $(1/2)((5/3)^k - (5/3)^{k-1}) = (1/5)(5/3)^k$. We note that y may have rank much lower than $k-1$, but that would only increase our gain. If z is the last child of y, we remove y from $G(x)$. If, on the other hand, y is vacant, we have two cases. First note that since the tree is tidy, $|C(y)| \geq 2$. If $|C(y)| > 2$, we can just take any child z of y and shortcut to x as above. Otherwise $C(y) = \{z, z'\}$. If both z and z' are occupied, we shortcut both z and z' to x and remove y. This gives a gain of at least $2((5/3)^k - (5/3)^{k-1}) - (1/2)(5/3)^k = (3/10)(5/3)^k$. Otherwise, one of them, say z is vacant. Tidyness implies that z has at least two children. If more than two, any one of them can be shortcut to x gaining at least $(1/2)((5/3)^k - (5/3)^{k-2}) = (8/25)(5/3)^k$. If exactly two, then one of them is shortcut to y and the other to x while z is removed. The gain in value is at least $(1/2)((5/3)^k + 2(5/3)^{k-2}) = (7/50)(5/3)^k$. We note that all the above shortcuts preserves tidyness. □

The following lemma shows how we—using Lemma 4—can regain the value lost due to a deletion.

Lemma 5. *In a tidy tree with a pointer to a node v of rank k, we can increase the value by $t \cdot (5/3)^k$ or get to a reduced tree in $O(t)$ time.*

Proof. The proof is constructive. We set $x = v$ and repeat the following until either we have gained enough value, or reach the base case of a reduced tree:

1. While $G(x)$ is non-empty and there is more value to be gained, apply Lemma 4.
2. If x is not the root, set $x = p(x)$.

In case 1, we gain $\Omega((5/3)^k)$ per constant time iteration due to Lemma 4. We cannot get to case 2 twice in a row without getting to case 1, since $p(x) \in G(p(p(x)))$. Thus, in $O(t)$ time, we either gain $t \cdot (5/3)^k$ in value, or we end

with x the root but with no grand children, that is, a tree of height at most 1. If we are in the base case with a tree of height 0 or 1, we set $rank(x)$ to 0 or 1, respectively. □

Combining Lemmas 2, 3, and 5 with $t = O(1)$, we implement a deletion in constant time so that either we have no loss, meaning that (3) is preserved, or obtaining a reduced tree that satisfies (3) directly. Thus we have proved

Theorem 1. *In union-find with deletion we can implement each* makeset*,* union*, and* delete *in constant time, and each* find *in* $O(\log n)$ *time.*

4 Faster amortized bounds

We will now show that we can get much faster amortized bounds for *find*, yet preserve the previous worst-case bounds. All we have to do is to use *path compression* followed by tidying up operations. Path compression of a path from node $v \in T$ to node $u \in T$ makes every node on the path a child of u. When we perform a *find* from a node v, we compress the path to the root. Our analysis is new and much cleaner analysis than was previously known even without deletes.

Before going further, we note that path compression consists of shortcuts that increase value of the previous section, so intuitively, the path compression can only help the deletions. Below, we first present our new analysis without the deletions, and then we observe that deletions are only helpful.

Analysis We assign a potential $\phi(x)$ to each node x in the forest. To define the potential we need some extra functions. Define $Q = \lceil \frac{M+N}{N} \rceil$ and $\alpha'(n) = \bar{\alpha}(Q, n)$. Note that $Q \geq 2$ whenever $M > 0$. Our goal is to prove that the amortized cost of *find* is $O(\alpha'(n))$ where n is the cardinality of the set found. We also define $rank'(v) = rank(v) + Q$.

Definition 4. *For a non-root node x we define*

$$level(x) = \max\{k \geq 0 \mid A_k(rank'(x)) \leq rank'(p(x))\} ,$$

$$index(x) = \max\{i \geq 1 \mid A^{(i)}_{level(x)}(rank'(x)) \leq rank'(p(x))\} .$$

We have

$$0 \leq \text{level}(x) < \bar{\alpha}(rank'(x), rank'(p(x))) \leq \alpha'(rank'(p(x))) , \tag{4}$$

$$1 \leq \text{index}(x) \leq rank'(x) . \tag{5}$$

Definition 5. *The potential $\phi(x)$ of a node x is defined as*

$$\phi(x) = \begin{cases} \alpha'(rank'(x)) \cdot (rank'(x) + 1) & \text{if } x \text{ root,} \\ (\alpha'(rank'(x)) - level(x)) \cdot rank'(x) - index(x) + 1 & \\ \qquad \text{if } x \text{ not root and } \alpha'(rank'(x)) = \alpha'(rank'(p(x))), & \\ 0 & \text{otherwise.} \end{cases}$$

The potential $\Phi(x)$ of a set A is defined as the sum of the potentials of the nodes in the tree T_A representing the set A: $\Phi(A) = \sum_{x \in T_A} \phi(x)$.

At first sight the potential function looks very similar to the standard one from [5], but there are important differences. Using $\alpha(\mathit{rank}(x))$ instead of $\alpha(N)$ we get a potential function that is more locally sensitive. To get this change to work, we use the trick that the potential of a node is only positive if $\alpha'(\mathit{rank}'(x)) = \alpha'(\mathit{rank}'(p(x)))$.

From (4) and (5) it immediately follows that the potential of a node x with $\alpha'(\mathit{rank}'(x)) = \alpha'(\mathit{rank}'(p(x)))$ is strictly positive. We also note that the only potentials that can increase are those of roots. All other nodes keep their ranks while the ranks of their parents increase and that can only decrease the potential.

We will now analyze the change in potential due to the operations.

Lemma 6. *The cost of* makeset *is amortized as a constant per* makeset *plus a constant per* find*.*

Proof. When we create a new set A with rank 0, it gets potential $\bar{\alpha}(Q, Q)(Q + 1) = 2(Q + 1) = O((M + N)/N)$. Over N *makeset* operations, this adds up to a total increase of $O(M + N)$. □

Lemma 7. *The operation* union(A, B) *does not increase the potential.*

Proof. Suppose we make A the parent of B. If the rank of A is not increased, there is no node that increases potential, so assume that $\mathit{rank}'(A)$ is increased from k to $k + 1$. Then k was also the rank of B. If $\alpha'(k + 1) > \alpha'(k)$, then B gets zero potential along with any previous child of A. The potential of B is reduced by $\alpha'(k) \cdot (k + 1)$. On the other hand, the potential of A is increased by $(\alpha'(k)+1)\cdot(k+2)-\alpha'(k)\cdot(k+1) = \alpha'(k)+k+2$, which is less than $\alpha'(k)\cdot(k+1)$, as $k \geq 2$ and $\alpha'(k) \geq 2$. (Here we use the fact that $\bar{\alpha}(j, i) \geq 2$, for every $i, j \geq 0$.)

Finally, if $\alpha'(k + 1) = \alpha'(k)$, then the potential of A increases by $\alpha'(k)$ while the potential of B decreases by at least $\alpha'(k)$, since B was a root with potential $\alpha'(k) \cdot (k + 1)$ and now becomes a child with potential at most $\alpha'(k) \cdot k$. □

Lemma 8. *A path compression of length ℓ from a node v up to some node u decreases the potential by at least $\ell - (2 \cdot \alpha'(\mathit{rank}'(u)) + 1)$. In particular, the amortized cost is at most $O(\alpha'(\mathit{rank}'(u)))$.*

Proof. The potential of the root does not change due to the path compression. We will show that at least $\max\{0, l - (2 \cdot \alpha'(\mathit{rank}'(u)) + 2)\}$ nodes have their potential decreased by at least one.

There are at most $\alpha'(\mathit{rank}'(u))$ nodes x on the path that had $\alpha'(\mathit{rank}'(x)) < \alpha'(\mathit{rank}'(p(x)))$ before the operation. The potentials of these nodes do not change.

If node x had $\alpha'(\mathit{rank}'(x)) = \alpha'(\mathit{rank}'(p(x))) < \alpha'(\mathit{rank}'(u))$, then its potential drops to 0, and the decrease in x's potential is therefore at least one.

It remains to account for the nodes x with $\alpha'(\mathit{rank}'(x)) = \alpha'(\mathit{rank}(u))$. Let x be a node on the path such that x is followed somewhere on the path by a node $y \neq u$ with $\mathrm{level}(y) = \mathrm{level}(x) = k$. There can be at most $\alpha'(\mathit{rank}'(u)) + 1$ nodes

on the path that do not satisfy these constraints: The last node before u, u, and the last node on the path for each level, since $\text{level}(y) < \alpha'(\mathit{rank}'(u))$. Let x be a node that satisfies the conditions. We show that the potential of x decreases by at least one. Before the path compression we have $\mathit{rank}'(p(y)) \geq A_k(\mathit{rank}'(y)) \geq A_k(\mathit{rank}'(p(x))) \geq A_k(A_k^{(\text{index}(x))}(\mathit{rank}'(x))) = A_k^{(\text{index}(x)+1)}(\mathit{rank}'(x))$. After the path compression we have $\mathit{rank}'(p(x)) = \mathit{rank}'(p(y))$ and thus $\mathit{rank}'((p(x)) \geq A_k^{(\text{index}(x)+1)}(\mathit{rank}'(x))$, since $\mathit{rank}'(x)$ does not change and $\mathit{rank}'(p(y))$ does not decrease. This means that either $\text{index}(x)$ or $\text{level}(x)$ must increase by at least one. Thus $\phi(x)$ decreases by at least one. □

We conclude that the amortized cost of *find* in a set A is

$$O(\alpha'(\mathit{rank}'(A))) = O(\bar{\alpha}(Q, \mathit{rank}(A) + Q + c)) = O(\bar{\alpha}(Q, \mathit{rank}(A))).$$

The last step follows because $\bar{\alpha}$ is defined to be at least 2. Recall that $Q = \lceil \frac{M+N}{N} \rceil$ and that $\mathit{rank}(A) \leq \log_2 |A|$, so without deletions, this is the desired bound.

Deletion and path compression We now combine the path compression and amortized analysis with deletions. The potential used in the amortization is identical for vacant and occupied nodes. It is clear that deletions and tidying up can only decrease this potential, so they have no extra amortized cost. Likewise, a path compression can only increase value as it only performs shortcuts. However, after a path compression, there may be some cleaning to do if some vacant nodes go down to 0 or 1 children. We start the path compression from a tidy tree where each vacant node has at least two children, and the compression takes at most one child from each node on the path. Hence the only relevant tidying up is to bypass some of the nodes on the path. The tidying up takes time proportional to the length of the path, so the cost of a *find* is unchanged.

The tidying up does decreases value, but the loss turns out less than the gain from the compression.

Lemma 9. *Path compression followed by tidying up operations does not decrease the value of a tree.*

Proof. The path compression involves nodes $v_0, ..., v_\ell$ starting in some occupied node v_0 and ending in the root which has some rank k. After the compression, all nodes $v_0, .., v_{\ell-1}$ are children of the root v_ℓ. If node v_i is not bypassed when tidying up, its value gain is at least $((5/3)^{\mathit{rank}(v_\ell)} - (5/3)^{\mathit{rank}(v_{i+1})})/2$. If v_i is bypassed, then $0 < i < \ell$, and v_i is vacant, so the loss is $(5/3)^{\mathit{rank}(v_{i+1})}/2$. However, then v_i has a child w_i which gains at least $((5/3)^{\mathit{rank}(v_\ell)} - (5/3)^{\mathit{rank}(v_i)})/2$, so the total change is

$$((5/3)^{\mathit{rank}(v_\ell)} - (5/3)^{\mathit{rank}(v_{i+1})} - (5/3)^{\mathit{rank}(v_i)})/2$$

Since ranks are strictly increasing along a path, this change is positive for all but $i = \ell - 1$. On the other hand, the first node v_0 is always occupied, and has a gain of at least $(5/3)^{\mathit{rank}(v_\ell)} - (5/3)^{\mathit{rank}(v_1)}$, where $1 \leq \ell - 1$. We can use the value

gained by v_0 to pay for the value lost by bypassing both v_1 and v_{l-1}. There are two cases.

If both v_{l-1} and v_1 is bypassed we must have $l \geq 4$. Combining the changes in potential for the nodes v_0, v_1, and v_{l-1} we get,$(5/3)^{rank(l)} - (5/3)^{rank(v_1)} - (1/2)(5/3)^{rank(v_2)} - (1/2)(5/3)^{rank(v_{l-1})} > 0$.

If v_1 is not bypassed, we get that the total gain for v_0 and v_{l-1} is at least, $(5/3)^{rank(v_l)} - (5/3)^{rank(v_1)} - (1/2)(5/3)^{rank(v_l)}$, which is always positive. Thus the overall change in value is positive, or zero if the path has length 0 or 1 and no compression happens. □

Since our values and hence (3) are preserved, for any set A, we get $rank(A) = O(\log |A|)$. Thus our amortized cost of a *find* operation is $O(\bar{\alpha}(Q, O(\log |A|)) = \Theta(\bar{\alpha}(\lceil \frac{M+N}{N} \rceil, |A|)$. Summing up, we have proved

Theorem 2. *If we do a total of M* find *operations on a total of N* makeset *operations, then the operation times can be amortized as follows. We pay only a constant for each* makeset*,* union*, and* delete*, and for a* find *on an element in a set A, we pay $O(\bar{\alpha}(\lceil \frac{M+N}{N} \rceil, |A|)$. Meanwhile, the worst-case bounds of Theorem 1 are preserved.*

References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, 1974.
2. S. Alstrup, A. M. Ben-Amram, and T. Rauhe. Worst-case and amortised optimality in union-find. In *Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing (STOC'99)*, pages 499–506, May 1999.
3. A. M. Ben-Amram and Z. Galil. A generalization of a lower bound technique due to Fredman and Saks. *Algorithmica*, 30(1):34–66, 2001.
4. N. Blum. On the single-operation worst-case time complexity of the disjoint set union problem. *SIAM J. Comput.*, 15(4):1021–1024, 1986.
5. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.
6. M. Fredman and M. Saks. The cell probe complexity of dynamic data structures. In *Proceedings of the 21st Annual Symposium on Theory of Computing (STOC '89)*, pages 345–354, New York, May 1989. ACM Association for Computing Machinery.
7. Z. Galil and G. F. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Computing Surveys*, 23(3):319, Sept. 1991.
8. H. Kaplan, N. Shafrir, and R. E. Tarjan. Union-find with deletions. In *Proc. of the 13th ACM-SIAM Symp. On Discrete Mathematics (SODA)*, pages 19–28, 2002.
9. D. L. Kozen. *The Design and Analysis of Algorithms*. Springer, Berlin, 1992.
10. R. Seidel and M. Sharir. Top-down analysis of path compression. *SIAM J. Comput.*, 34(3):515–525, 2005.
11. M. Smid. A data structure for the union-find problem having good single-operation complexity. *ALCOM: Algorithms Review, Newsletter of the ESPRIT II Basic Research Actions Program Project no. 3075 (ALCOM)*, 1, 1990.
12. R. E. Tarjan. Efficiency of a good but not linear disjoint set union algorithm. *Journal of the ACM*, 22:215–225, 1975.
13. R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2):245–281, Apr. 1984.