The Union-Find Problem Is Linear

Hantao Zhang*
Computer Science Department
The University of Iowa
Iowa City, IA 52242
hzhang@cs.uiowa.edu

Abstract

The *union-find problem*, also known as the *disjoint set union problem*, involves two operations, *union* and *find*, on a collection of disjoint sets. The worst case time complexity of a sequence of n such operations was known to be near linear. We prove that its worst case time complexity is linear. The new proof technique is based on amortized complexity analysis and transformation. Our proof is substantially simpler than that of the near-linear time complexity analysis appearing in many textbooks.

key words: the union-find problem, amortized complexity, analysis of algorithms

1 Introduction

The *union-find problem*, also known as the *disjoint set union problem*, is well-studied in data structures and algorithms [1]. If we regard initially n objects as n singleton sets, the union-find problem involves the following two operations.

- *find*(x): find the representative of the set containing object x.
- *union*(x, y): union the two disjoint sets containing objects x and y, respectively.

The complexity of the union-find problem is typically measured by the total time required to perform a sequence of m mixed unions and finds. In general, we cannot have more than $n-1$ union operations concerning n objects. If the number of union operations is less than $n-1$, then we will have more than one set at the end of the operations, and in this case, we can divide the operations into groups according to the final sets and analyze the complexity of each group. Without loss of generality, we assume that there are exactly $n-1$ unions and at least one find operation, hence $n \leq m$. Based on the work of Robert Tarjan [7, 8, 9], it is widely believed that the worst-case time complexity of m union-find operations grows super-linearly (in terms of m), but linear in practice. This result was enforced by Michael Fredman and Michael Saks' STOC'89 paper [3] and was referred by the well-known wikipedia [10]. In this paper, we correct a mistake of 30 years by proving that the worst-case time complexity of m union and find operations is linear, both in theory and in practice.

2 Disjoint-set forests

The best known implementation of the union-find problem is to use rooted trees [6], with each node containing one object and each tree representing one set. Moreover, the representative of each set is the root node. To carry out the *find*(x) operation, we locate the tree node containing x;

*Partially supported by the National Science Foundation under Grant CCR-0604205.

then we follow the parent pointers to the root of the tree. To carry out the *union*(x, y) operation, we make the root of the tree containing x as son of the root of the tree containing y.

Based on the tree structure, there are two principal strategies for improving the union-find algorithms: path compression and weighted union [1]. That is, in the *find*(x) operation, we compress the path from x to the root as follows: Making all vertices reached during the *find*(x) operation sons of the root of the tree. The idea of path compression is first used by McIlroy and Morris [6] in an algorithm for finding spanning trees. Our result depends solely on the idea of path compression. Weighted union, including both union by rank or union by height, may improve the performance of the algorithms, but it does not affect the linear time complexity proof. For simplicity, we ignore the presentation of weighted union in this paper.

By the abuse of notation, let x denote both an object and the tree node containing that object. Let $p[x]$ be the parent of node x in a tree; if x is a root, then $p[x] = x$. Initially, we have $p[x] = x$ for every object x. The execution of the union and find operations are described by the following pseudocode [1].

```
find(x)
1 if (x != p[x] and p[x] != p[p[x]])
2    then p[x] := find(p[x])
3 return p[x]

union(x, y)
 Pre: find(x) != find(y)
1 link(find(x), find(y))

link(u, v)
 Pre: u != v, p[u] = u and p[v] = v,
1 p[u] := v
```

The `find` procedure returns the root of the tree containing x and at the same time compress the path from x to the root. That is, each call of `find`(x) returns $p[x]$ in line 3. If x is the root or its parent is the root, then line 2 is not executed and $p[x]$ is returned at line 3. This is the case in which the recursion bottoms out. Otherwise, line 2 is executed, and the recursive call with parameter $p[x]$ returns the root which is also returned at line 3. At the same time, in line 2, $p[x]$ is updated with the root, thus completing path compression.

3 Complexity analysis

The complexity of `find`(x) is the same as the distance from x to the root. The complexity of `union`(x, y) is dominated by those of `find`(x) and `find`(y), as `link`(u, v) takes only constant time. For our purpose, we assume that the cost of `link`(u, v) is one and the real cost of `find`(x) is either 1 if x is a root node or the distance from x to the root.

We will prove that a sequence of m mixed union and find operations takes $O(m)$ time. Let S_0 be a sequence of m operations, $a_1, a_2, ..., a_m$, where a_i is either `union`(x, y) or $y :=$ `find`(x). By the assumption, $n - 1$ of these m operations are unions. We will first transform S_0 into S_1 by replacing each `union`(x, y) in S_0 by the following three operations:

$$u :=\texttt{find}(x), v :=\texttt{find}(y), \texttt{link}(u, v)$$

Let the resulting sequence be S_1. Obviously, the real costs of S_0 and S_1 are the same and S_1 has $m + 2n - 2$ operations ($n - 1$ links and $m + n - 1$ finds).

Before we go on, given a sequence S of operations, let us the *switch* of a node x, $s(x)$, in S is an integer between 1 and $|S|$. Initially, $s(x) = |S_1|$ in S_1; its value may be decreased in later sequences. The *find cost* of a node x with respect to an integer j is computed as follows: If $j \leq s(x)$, the *find cost* of x is 1; otherwise, it is 2. The *adjusted cost* of the `find`(x) operation in a sequence S of

operations is computed as follows: Suppose $\texttt{find}(x)$ is the j^{th} operation in S. The *adjusted cost* of $\texttt{find}(x)$ is the sum of all find costs of nodes (except the root node) on the find path with respect to j. It is easy to see that the adjusted cost of the $\texttt{find}(x)$ operation is no less than its real cost but no bigger than the twice of its real cost. The *adjusted cost* of a link operation is always one, the same as the real cost. Formally, let $C(S)$ be the adjusted cost of a squence S, $C(a, j)$ be the adjusted cost of an operation a with respect to j, and $c(x, j)$ be the find cost of an object x with respect to j, then

$$\begin{aligned} C(S) &= \textstyle\sum_{j=1}^{p} C(a_j, j) \text{ where } S = \langle a_1, a_2, ..., a_p \rangle, \\ C(\texttt{find}(x), j) &= \max\{1, \textstyle\sum_{i=1}^{q-1} c(x_i, j)\}, \text{where the find path of } x \text{ is } \langle x_1, x_2, ..., x_q \rangle \\ C(\texttt{link}(x, y), j) &= 1 \\ c(x, j) &= \textbf{if } (s(x) \leq j) \textbf{ then } 1 \textbf{ else } 2 \end{aligned}$$

Obviously, $C(S)$ is an upper bound of the real cost of all operations in S. In the remaining of the paper, by "cost" we mean "adjusted cost".

By assumption, there are $n-1$ link operations in S_1. If $\texttt{link}(x, r)$ is the last link operation, then r is the root of the tree formed by the $n-1$ link operations. Moreover, for every node x other than r, there is exactly one $\texttt{link}(x, w)$ in S_1 for some w. No matter how the $\texttt{find}$ operations are performed in S_1, r is always the root of the resulting tree. Let us call r the ultimate root. The switch value of a node may be decreased when this node is added as a child of r.

Next, we will obtain a series of sequences, S_1, S_2, ..., S_k, such that $C(S_i) \leq C(S_{i+1})$ for every $1 \leq i < k$. Later, we will show that $C(S_k)$ is $O(m)$. Hence the real cost of S_0 is $O(m)$ because it is the same as $C(S_1)$ and $C(S_1) \leq C(S_2) \leq \cdots \leq C(S_k) = O(m)$.

Starting from S_1, for the rest of sequences $S_2, ..., S_k$, it involves the following transformation: For $1 \leq i < k$, if S_i contains

$$y := \texttt{find}(x), \texttt{link}(u, v)$$

as two consecutive operations, then S_{i+1} is the same as S_i, except that the above two operations are swapped, i.e., the two operations are replaced by the following:

$$\texttt{link}(u, v), y := \texttt{find}(x)$$

For simplicity, we also require that $\texttt{find}(x)$ must be the last find operation before any link operation in S_i.

The soundness of the above swapping is evident because the precondition of $\texttt{link}$ does not change during the swap because $\texttt{find}(x)$ does not change the root status of any node, that is, a root remains a root and a non-root remains a non-root. Moreover, this kind of swaps cannot go on forever. Once every link operation appears before every find operation in a sequence S_i, we arrive at S_k. Using the idea of bubble sort, we may show that $k = O(m^2)$.

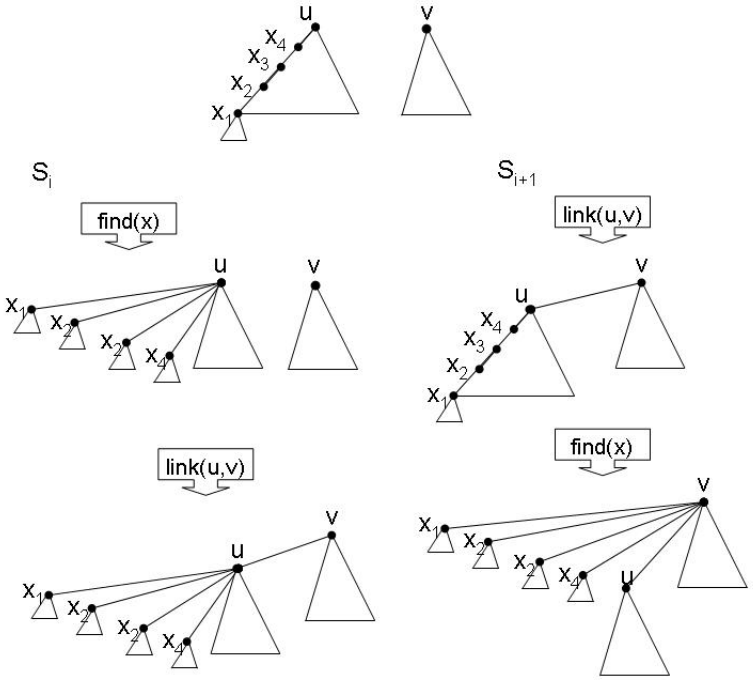
Suppose the swapped $\texttt{find}(x)$ operation is the j^{th} operation in S_i, and $x_1, x_2, ..., x_d$ are the nodes added to the ultimate root r by this find operation, then the switch values of those nodes in S_{i+1} is set to j, that is, $s(x_1) = s(x_2) = \cdots = s(x_d) = j$; the switch values of other nodes are the same as in S_i. In other words, $c(x_y, j)$ is 1 in C_i but 2 in C_{i+1} for $y = 1, ..., d$. This idea is illustrated by the following example.

Example 1 Let $l(x, y)$ denote $\texttt{link}(x, y)$ and $f(x)$ (with subscripts) denote $\texttt{find}(x)$. Consider the following sequences of operations by the transformation described above.

		$s(a)$	$s(b)$	$s(c)$	$s(r)$	$C(S_i)$
S_1 :	$l(a,b), l(b,c), f_1(a), f_2(a), f_3(b), l(c,r)$	6	6	6	6	7
S_2 :	$l(a,b), l(b,c), f_1(a), f_2(a), l(c,r), f_3(b)$	6	5	6	6	9
S_3 :	$l(a,b), l(b,c), f_1(a), l(c,r), f_2(a), f_3(b)$	4	5	6	6	11
S_4 :	$l(a,b), l(b,c), l(c,r), f_1(a), f_2(a), f_3(b)$	3	3	6	6	12

The ultimate root is r and the cost of S_1 is $C(S_1) = 7$, which is also the real cost. $f_3(b)$ is the swapped find operation in the transformation from S_1 to S_2. Since $f_3(b)$ brings b to r, $s(b) = 5$

Figure 1: Illustration of the impact of switching $\texttt{link}(u, v)$ and $\texttt{find}(x)$ on the trees. It can be seen that if $v = r$, the ultimate root, then the number of children added to r by $\texttt{find}(x)$ in S_{i+1} is d ($d = 4$ in this figure). The cost of *find*(x) will be $d+1$ more in S_{i+1} than in S_i because the switch values of these nodes are set to be the position of $\texttt{find}(x)$ in S_{i+1}. The extra cost will suffice to cover the difference of later find operations in S_i and S_{i+1}. So $C(S_i) \leq C(S_{i+1})$.



(the fifth operation in S_1) in S_2 and $C(S_2) = 9$. $f_2(a)$ is the swapped find operation in the transformation from S_2 to S_3. Since it brings a to r, $s(a) = 4$ in S_3 and $C(S_3) = 11$. Finally, $f_1(a)$ is the last swapped find operation which brings both a and b to r, so $s(a) = s(b) = 3$ in S_4 and $C(S_4) = 12$. []

The following technical lemma is crucial for the main result of this paper.

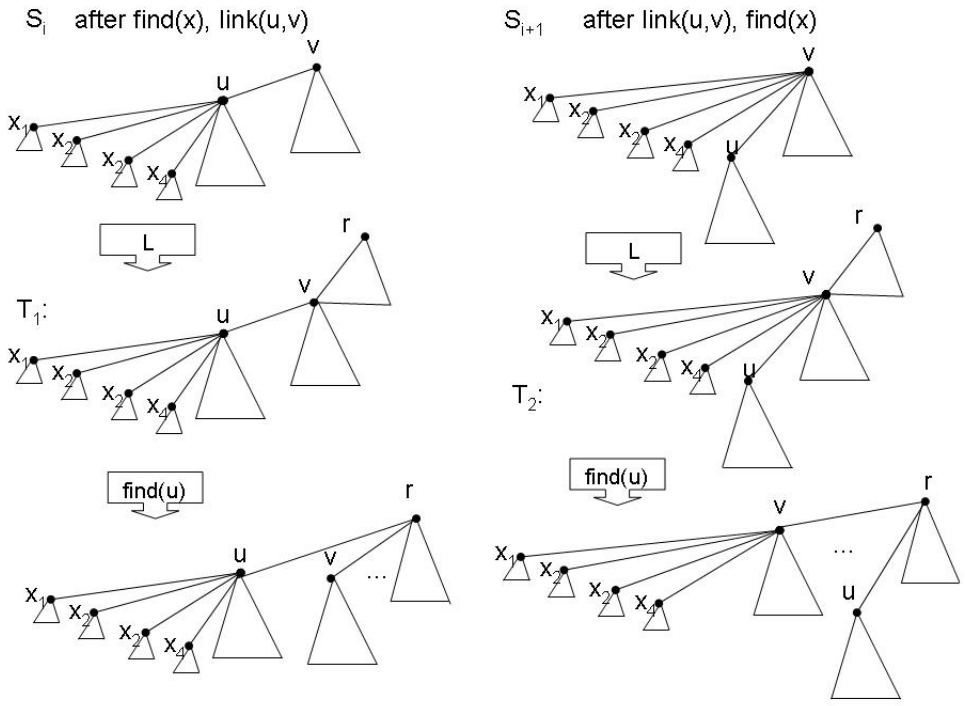
Lemma 1 *For $1 \leq i < k$, suppose $\langle \texttt{link}(u, v), y := \texttt{find}(x) \rangle$ are the two swapped operations in S_{i+1}, then $C(S_i) \leq C(S_{i+1})$.*

Proof Let the ultimate root be r. If x does not appear in the tree rooted by u, then the swap has no impact at all on any operation following them in the sequence. If $x = u$, then the cost of $\texttt{find}(u)$ is the same in S_i and S_{i+1}. In both cases, the swap has no impact at all on any operation following them, hence $C(S_i) \leq C(S_{i+1})$.

Now let us consider the remaining situation that x is in the tree rooted by u and $x \neq u$. Let the path from x to u be $\langle x_1, x_2, ..., x_d, x_{d+1} \rangle$, where $x = x_1$, $u = x_{d+1}$, $d \geq 1$, and $p[x_i] = x_{i+1}$ for $1 \leq i \leq d$. In both S_i and S_{i+1}, the cost of $\texttt{link}(u, v)$ is always one. The length of the find path of $\texttt{find}(x)$ is d in S_i but $d+1$ in S_{i+1}. Since $x_1, x_2, ..., x_d$ cannot be children of r (the ultimate root) in S_i, the cost of $\texttt{find}(x)$ is d in S_i, and $p[x_i] = u$ for all $1 \leq i \leq d$ after its completion. On the other hand, in S_{i+1}, the cost of $\texttt{find}(x)$ is either $d+1$ or $2d+1$, and $p[x_i] = v$ for all $1 \leq i \leq d+1$. Figure 1 illustrates the same tree which is changed by $y := \texttt{find}(x), \texttt{link}(u, v)$ in S_i (on the left) and by $\texttt{link}(u, v), y := \texttt{find}(x)$ in S_{i+1} (on the right).

There are two cases to consider.

Figure 2: Illustration of the case 2.1. It can be seen that after completing L and $\texttt{find}(u)$, all the `find` operations will have the same complexity.



Case 1: $v = r$**.**

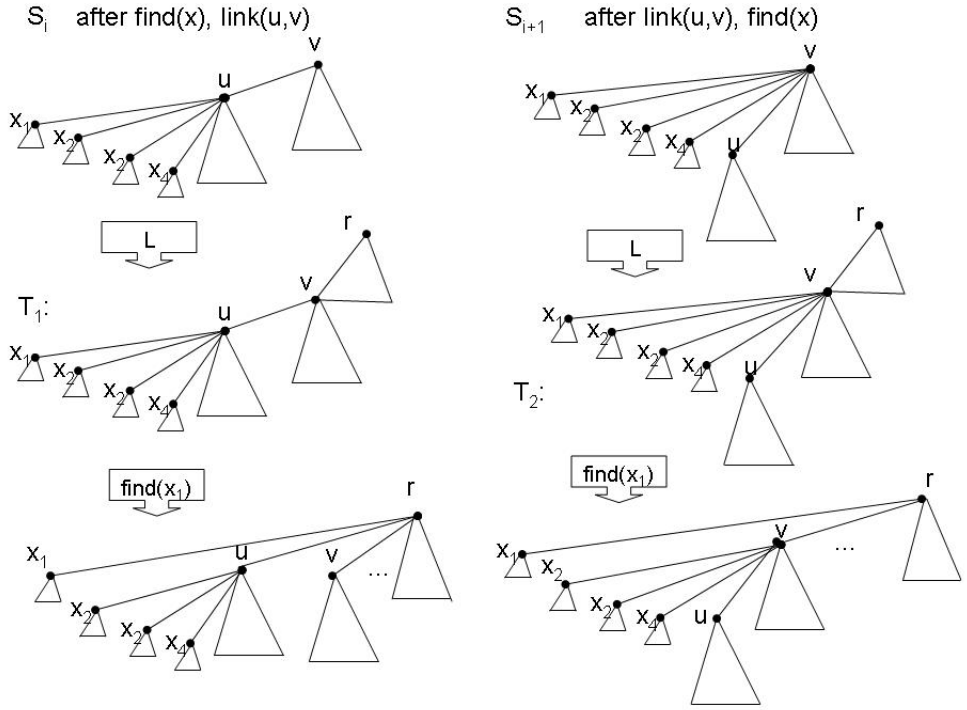
In this case, the swapped $\texttt{find}(x)$ operation brings the nodes on the find path of x to the root r in S_{i+1}. If there exist find operations after the swapped find opertion in S_i for these nodes, say $\texttt{find}(x_i)$ (or any node in the subtree rooted by x_i), such operations in S_i will cost one more than the same operation in S_{i+1}. To avoid the cost of S_i being greater than the cost of S_{i+1}, for S_{i+1}, we decrease the switch values of these nodes on the find path to increase the cost of $\texttt{find}(x)$. That is, we set $s(x_1) = s(x_2) = \cdots = s(x_d) = j$, where j is the position of $\texttt{find}(x)$ in S_i. The cost of $\texttt{find}(x)$ in S_{i+1} is $2d+1$, which compensates possible smaller costs of later find operations in S_{i+1} for x_1, ..., x_d (or any node in the subtrees rooted by them). As a result, we have $C(S_i) \leq C(S_{i+1})$ (Figure 1). Note that this is the only place when we decrease the switch values (thus increase the find cost) of nodes and these nodes are the children added to the ultimate root by the swapped find opertion. By assumption, this find operation is the last one before any link operation in S_i. So the `find` operations appearing after $\texttt{find}(x)$ in S_i will not appear in any future swapping. As a result, the switch value of any node cannot be increased.

Case 2: $v \neq r$.

Since $\texttt{find}(x)$ is the last find operation before any link operation, S_i can be written as $S'\langle a, a'\rangle LF_i$, and S_{i+1} as $S'\langle a', a\rangle LF_i$, where a is $y := \texttt{find}(x)$, a' is $\texttt{link}(u, v)$, L is a sequence of link operations, F_i is a sequence of find operations, and S' is a sequence of mixed operations. Suppose the trees after completing $S'\langle a, a'\rangle L$ in S_i and $S'\langle a', a\rangle L$ in S_{i+1} are T_1 and T_2, respectively (see Figure 2). For the `find(y)` operation in F_i, if y is not in the subtree rooted by u in T_1, it will have the same cost and effect in both T_1 and T_2.

If $\texttt{find}(y)$ is the first operation in F_i, where y is in the subtree rooted by u in T_1, there are two cases to consider:

Figure 3: Illustration of the case 2.2. It can be seen that after completing L, `find`(x_1) and `find`(u), all the find operations will have the same complexity.



- Case 2.1: y is not in the subtrees rooted by x_i in T_1.
- Case 2.2: y is in one of the subtrees rooted by x_i in T_1.

For case 2.1, suppose the sum of find costs for the nodes in the path from v to r is d' and the distance from y to u is d'', then the cost of `find`(y) is $d' + d'' + 1$ in both S_i and S_{i+1}. After completing `find`(y), the distance of x_i to r is two in both S_i and S_{i+1} for any $1 \leq i \leq d$. Figure 2 illustrates the case when $y = u$ ($d'' = 0, d = 4$). So every find operation of F_i will have the same cost in both S_i and S_{i+1}.

By a similar reasoning, we can also show that $C(S_i) \leq C(S_{i+1})$ in the case 2.2 (see Figure 3). Table 1 summarizes the costs of find operations in the proof of this lemma. []

Lemma 2 *If every find operation appears before any link operation in S_k, then $C(S_k)$ is $O(m)$.*

Proof. At first, note that S_k can be written as LF_k, where L is a sequence of $n-1$ link operations and F_k is a sequence of $m' = m+n-1$ find operations. Since $C(L) = n-1$, we prove $C_L(F_k) = O(m')$, where $C_L(F_k)$ denotes the cost of F_k after performing L on the initial singleton sets; this task is similar to the exercise problem 22.3-4 in [1] and we borrow a proof from [11].

Let the tree formed by L be T, which has n nodes and $n-1$ edges. Let $d(x)$ be the number of children that x has in T. For any i^{th} operation of F_k, say `find`(x), if $x = r$ (the ultimate root), then its cost is 1. If $x \neq r$, let the find path for this operation be $\langle x, y_1, ..., y_{e_i}, r\rangle$. The cost of this find operation is no more than $2e_i + 2$. For any node y other than r, the number of occurrences of y as y_j in the above find path is at most $d(y)$ because if y appears in one find path of F_k, let y' be the child of y in this path, then y will lose y' after path compression, so $\langle y', y\rangle$ cannot appear more than once in the find paths of F_k.

Table 1: Summary of the costs of the involved find operations in S_i and S_{i+1} in the proof of Lemma 1. The $\texttt{link}(u, v)$ operation always has cost 1. The find path of find(x) is $\langle x = x_1, x_2, ..., x_d, v\rangle$. In the case 1, v is the ultimate root r. In both cases (2.1) and (2.2), v is not r, and d' is the cost of $\texttt{find}(v)$.

	S_i	S_{i+1}
$\texttt{find}(x)$	d	$2d+1$
d $\texttt{find}(x_i)$	$3d$	$2d$
$\texttt{find}(u)$	1	1
Total	$4d+1$	$4d+2$

case 1

	S_i	S_{i+1}
$\texttt{find}(x)$	d	$d+1$
$\texttt{find}(u)$	$d'+1$	$d'+1$
d $\texttt{find}(x_i)$	$2d$	$2d$
Total	$3d+d'+1$	$3d+d'+2$

case 2.1

	S_i	S_{i+1}
$\texttt{find}(x)$	d	$d+1$
$\texttt{find}(x_1)$	$d'+2$	$d'+1$
$\texttt{find}(u)$	1	2
d $\texttt{find}(x_i)$	$2d$	$2d$
Total	$3d+d'+3$	$3d+d'+4$

case 2.2

Hence, assuming $e_r = 0$,

$$C_L(F_k) \leq \sum_{i=1}^{m'}(2e_i+2) = 2m' + 2\sum_{i=1}^{m'} e_i \leq 2m' + 2 \sum_{\text{node } x \neq r} d(x) = 2m' + 2(n-1-d(r)) < 2(m'+n).$$

So $C(S_k) = C(L) + C_L(F_k) < n + 2(m'+n) < 2m + 4n$. Because $m \geq n$, $C(S_k)$ is indeed $O(m)$. []

Using the idea stated in the beginning of this section and the above two lemmas, we can easily prove the following theorem.

Theorem 1 *The complexity of a sequence of m mixed union and find operations is $O(m)$.*

4 What went wrong

Our result is contradictory to the public belief that the union-find problem is not linear. In his 1975 paper [7], Robert Tarjan proved that the complexity of m operations on n objects, $m \geq n$, is between $k_1 m\alpha(m, n)$ and $k_2 m\alpha(m, n)$ for some positive constants k_1 and k_2, where $\alpha(m, n)$ is related to a functional inverse of Ackermann's function and is very slow-growing. In 1979, Tarjan published a different non-linear proof of the same result, along with other results [8]. In [9], Tarjan and van Leeuwen analyzed the asymptotic worst-case running time of a number of variants of the union-find problem, using the result in [8]. On the other hand, several researchers, including Andrew Yao [12], Jon Doyle and Ronald Rivest [2], and more recently, Wu and Otoo [11], proved that the expected time of the union-find problem is linear, which is consistent with our result.

In the following, we will examine some potential pitfalls in the proofs of [8]; the reader is recommended to have [8] at hand.

Let $B(i, j)$ be the function given in [8], where for $i, j \geq 1$,

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

Theorem 4.3 of [8]. For any $k, s \geq 1$, let T be a complete binary tree of depth $d > B(k,s)$. Let $\{v_i \mid 1 \leq i \leq s2^{B(k,s)}\}$ be a set of pairwise unrelated vertices in T, each of depth strictly greater than $B(k,s)$, such that exactly s vertices in $\{v_i\}$ occur in each subtree of T rooted at a vertex of depth $B(k,s)$. Then for $n = 2^{h+1} - 1$ and $m = s2^{B(k,s)}$ there is a set-union problem which

1. the union tree is T;
2. the set of finds is $\{\text{find}(v_i) \mid 1 \leq i \leq m\}$;
3. the answer to each find is a vertex of depth strictly less than $B(k,s)$; and
4. ... ...

In [8], the proof of this theorem consists of an induction proof based on the definition of $B(k,s)$. That is, the first equation of $B(k,s)$ provides the base case; the second equation provides the inductive proof of case $B(i,1)$ with case $B(i-1,2)$ being the hypothesis; the last equation provides an inductive proof of the general case with two induction hypotheses on cases $B(i,j-1)$ and $B(i-1,2^{B(i,j-1)})$, respectively.

The first pitfall is on the definition of depth d in the theorem. The theorem does not specify whether it is "for all $d > B(k,s)$" or "there exists $d > B(k,s)$". For the base case proof, we need the latter, i.e., "there exists $d > B(k,s)$". However, in the proof of the inductive cases, the former was used in [8] for the induction hypotheses.

The second pitfall is on the use of m which is a function of k and s. In the proofs of the inductive cases in [8], m is assumed to be a constant. That is, the same m is used for both the proving case and its hypotheses.

We also found a pitfall in the proof of an important corollary of the above theorem.

Corollary 4.1 of [8]. Let $k, s \geq 1$. Let T be a complete binary tree of depth $B(k,s)$. Then there is a set-union problem whose union tree is T, which contains $m = s2^{B(k,s)}$ finds, and which requires at least $(k-1)m$ links for its solution.

In [8], the proof goes as follows: Choose $l \geq 1$ such that $2^l \geq s$. Let T' be a complete binary tree formed by replacing each leaf of T by a complete binary tree of height l. ...

Here is the pitfall of this proof: Suppose T has n vertices. Then T may have $n/2$ leaves. A complete binary tree of height l has more than s vertices. So T' may have more than $n' = (1+s)n/2$ nodes. A linear function of n' is not a linear function n. Hence, the complexity result on T' certainly does not apply to T.

The lower bound result of [8] uses the above corollary and its validity is certainly in doubt.

Another influential paper on the non-linear lower bound of the union-find problem was presented by Fredman and Saks at the Twenty-First Annual ACM Symposium on Theory of Computing (STOC'89) [3]. Theorem 5 of [3] states the non-linear result. The first line of the proof of Theorem 5 says "We consider the case of n Find's and $n-1$ Union's, beginning with n singleton sets". Then in the middle of the same paragraph, it says "the Union-Find sequences we consider do not create sets of size larger than $\sqrt{(n)}$, ..." Apparently, this assumption is wrong since $n-1$ unions will create the set of size n. The rest proof is based on this assumption.

5 Conclusion

We have presented a new proof which shows that the complexity of the union-find problem is linear. This result is better than what we believed in the past thirty years. While our result does not provide a new algorithm for the union-find problem, the proof of our result is substantially simpler than the old near-linear proof [1] and will likely save the time for those who study such algorithms. Since the union-find problem has many applications [1, 10], our result also improves the complexity of any algorithm using the union-find data structure.

Acknowledgment

The author wishes to thank Kasturi Varadarajan, Steve Bruell, and Krzysztof Templin for helpful comments on an early version of this paper.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to Algorithms. McGraw-Hill, 2nd Ed. 2001.

[2] Jon Doyle, Ronald L. Rivest. Linear expected time of a simple union-find algorithm. *Inf. Process. Lett.*, 5(5):146-148, 1976.

[3] Michael L. Fredman and Michael E. Saks. The cell probe complexity of dynamic data structures. In Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing, pages 345–354, Seattle, WA, May 1989.

[4] Zvi Galil, Giuseppe F. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Comput. Surv.* 23(3) 319–344 (1991)

[5] B.A. Galler and M.J. Fisher. An improved equivalence algorithm. Commun. ACM 7(5), 301-303 (1964)

[6] J.E. Hopcroft and J.D. Ullman. Set-merging algorithms. *SIAM j. Comput.* 2 (1973) 294–303

[7] Robert E. Tarjan, Efficiency of a good but not linear set union algorithms. *J. ACM*, 22(2): 215-225 (1975)

[8] Robert E. Tarjan, A class of algorithms which require nonlinear time to maintain disjoint sets. *J. Computer and System Sciences*, 18(2): 110-127 (1979)

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

[10] Wikipedia: Disjoint-set data structure. `http://en.wikipedia.org/wiki/Disjoint-set_data_structure`, 2008.

[11] K. Wu, E. Otoo, A simpler proof of the average case complexity of union-find with path compression. Lawrence Berkeley National Laboratory, Technical Report, LBNL-57527, 2005

[12] Andrew C. Yao. On the expected performance of path compression algorithms. *SIAM J. Comput.* 14(1) 129–133 (1985)