LBNL-57527

A Simpler Proof Of The Average Case Complexity Of Union-Find With Path Compression

Kesheng Wu and Ekow Otoo

Lawrence Berkeley National Laboratory, Berkeley, CA, USA

{KWu, EJOtoo}@lbl.gov

April 28, 2005

Abstract

We present a modified union-find algorithm that represent the data in an array rather than the commonly used pointer-based data structures, and a simpler proof that the average case complexity of the union-find algorithm is linear.

Keywords and Phrases: *union-find algorithms, analysis of algorithms, average case complexity.*

1 Introduction

The disjoint set union (also known as union-find) problem has many applications and is well studied [1, 3]. The data structure used in most of the successful algorithms for union-find, is a rooted trees [3]. There are two principal strategies for improving the union-find algorithms; path compression and weighted union [1]. In this paper, we discuss a simple array-based implementation with the path compression strategy. We prove that the average expected time complexity is linear in the number of union and find operations. There are two well known proofs of this linearity; the first based on the spanning tree model [6] and, the second based on the assumption that each tree is equally likely to participate in the next union operation [2]. The first model is considered more realistic than the second, but the proof of the second is much simpler. In this paper, we present a proof based on the average length of find paths. The proof is as simple as the second, and its assumption is as realistic as the first.

Given n objects, union-find involves three operations *union*, *find* and *make* for building and maintaining the objects in sets. Starting with n objects, the three operations can be defined as the following three functions. See Figure 1 for an illustration.

- *make*(x): generate a new set containing only x.

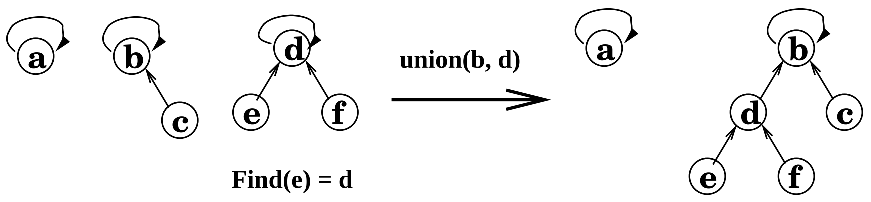

Figure 1: An illustration of union-find with six objects named ’a’ through ’f’.

letter labels	a	b	c	d	e	f
integer labels	0	1	2	3	4	5
content of the array						
before union(b, d)	**0**	**1**	**1**	**3**	**3**	**3**
after union(b, d)	**0**	**1**	**1**	**1**	**3**	**3**

Figure 2: An array representation of the rooted trees of Figure 1.

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

The complexity union-find algorithm is typically measured by the total time required to perform $n - 1$ union operations mixed with m find operations. Using the path compression strategy alone, there are known scenarios where the cost of $n - 1$ union operations grows super-linearly [4]. However, these cases are so rare that the average cost is in fact linear [2, 6].

2 An array based implementation

The equivalence information in the union-find problem is usually stored in rooted trees. Each object is represented as a node in the trees and each node has a link to its parent. In our array-based implementation the parent links are stored in an array as in [2]. It is easy to verify that any labeling of the tree can be mapped to some integer labels and therefore allow the parent links to be stored in an array as shown in Figure 2.

Our implementation uses two basic functions; *find* and *set* as outlined below, where the pseudo-codes are presented with C++-like syntax. The parent links are stored in an array `P`.

Algorithm 1: Algorithm *find*

```
/* Finds the root label of node i                                   */
Input: An array P and a Node label i
unsigned find(const std::vector<unsigned>& P, const unsigned i)
begin
    unsigned root = i;
    while P[root] != root do
        root = P[root];
    return root
end
```

Algorithm 2: Algorithm *set*

```
/* Set all parents of node k to point to a new root node.             */
Input: An array P, a Node label k and the root
void set(std::vector<unsigned>& P, unsigned k, unsigned root)
begin
    unsigned i = k;
    while P[i] != i do
        unsigned j = P[i];
        P[i] = root;
        i = j;
    P[i] = root;
end
```

The actual find operation we use will be referred to as *findCompress*, which basically calls *find* followed by *set*. We implement the *union* function as follows.

Algorithm 3: Algorithm *Union*

```
/* Combine the two trees containing respectively nodes i and j,        */
/* and return the root of the new tree.                                */
Input: An array P, and two arbitrary nodes i and j.
unsigned union(std::vector<unsigned>& P, unsigned i, unsigned j)
begin
    unsigned root = find(P, i);
    if i != j then
        unsigned tmp = find(P, j);
        if root > tmp then
            root = tmp;
        set(P, j, root);
    set(P, i, root);
    return root;
end
```

When nodes i and j are not the same, the cost of this *union* function is equivalent to two *find* operations with compression plus a few additional steps. One feature of this function is that at the expense of two *findCompress* on the trees before the union operation, we achieve the same effect as if the operations were performed on the united tree.

3 Cost analysis

Since our implementation only uses path compression, the cost analyses by Yao [6], and also by Doyle and Rivest [2] should apply. Both analyses give the average cost of $n-1$ union operations, although typical measures of the union-find problem is in terms of a mixture of $n-1$ union operations and m find operations. In this section, we consider the more general cost measure. In our analysis, we consider the cost of a *union* operation to be exactly two *findCompress* operations. Since we never assign a zero cost to any *findCompress* operation, this should not affect the asymptotic complexity of the algorithms. This assumption simplifies our analysis and allows us to only consider the average cost of *findCompress* operations. We define the cost of a *findCompress* operation to be the the number of nodes on a find path, which is also referred to as path length.

To compute the average cost of *findCompress* operations of any tree with t nodes, we first evaluate the total cost of t *findCompress* operations starting from every node of the tree. Note that every find path must contain a starting node and the root node. There is one path starting with the root. Altogether, there are $2t-1$ nodes at the beginning and the end of t paths. Next, we count the number of other nodes along the paths. First we determine the number of paths that a node participates in.

Lemma 1 *Let d_i be the number of children of a node i before any* find *operation. Among the t* find *operations, node i appears exactly d_i times where it is neither at the starting node nor the root.*

Proof. If node i appears in a find path and it is neither the starting node nor the root, one of it children must be also on the same find path. Each time this happens, the path compression strategy will remove the child of i and make it a child of the root. Since node i initially has d_i children, it can appear in find paths at most d_i times. For any child of node i, the first time the child appears in a find path, node i must also be on the same find path. Among the t *find* operations, all children of node i must be involved, and, node i must appear in at least d_i find paths. Therefore, node i appears exactly d_i times. ∎

For simplicity, let the root node be node 0 and let the remaining nodes be numbered from 1 to $t-1$.

Theorem 2 *For a tree with t nodes, the total cost of performing one* find *operation on every node is $2t-1+\sum_{i=1}^{t-1} d_i$.*

Proof. According to Lemma 1, each node i appears d_i times as an internal node of the find paths, except the root node 0. Therefore the total number of internal nodes in all paths is $\sum_{i=1}^{t-1} d_i$. Including the start nodes and the root, the total number of nodes appearing in t find paths is $2t-1+\sum_{i=1}^{t-1} d_i$. ∎

A slight overestimation of the total cost is $2t-1+\sum_{i=0}^{t-1} d_i$. Since a tree with t nodes has $t-1$ edges, $\sum_{i=0}^{t-1} d_i = t-1$. The total cost of t *findCompress* operations is no more than $3t-2<3t$, and the average cost of a find is no more than 3, i.e., $\mathcal{O}(1)$.

The trees of union-find data structure forms a forest. To start with, there are only trivial edges that point back to each node itself. After each union operation, the forest gains one nontrivial edge. The sum of d_i across all trees also increases by one. After u *union* operations, the sum of d_i is u, i.e., $\sum d_i = u$. The number of trees is $n-u$. For each tree, there is one find path containing only the root node. Thus the total cost of n *findCompress* operations is no more than $2n-(n-u)+u=n+2u$, which is less than $3n$. In fact, since the root of each tree is counted separately, the contribution from $\sum d_i$ to the total cost of find operations is actually less than u. This reduces the maximum cost. Overall, the total number of nodes on all find paths is between n and $n+2u$. The average cost of a find operation is never more than 3. After a *findCompress* operation is performed on each node, any subsequent find operation will involve a starting node and the root node. The cost of this operation is no more than 2. This leads to the following corollary.

Corollary 3 *After any number of union operations, the average cost of a find operation is* $\mathcal{O}(1)$.

Since the average cost of a *findCompress* operations is $\mathcal{O}(1)$ after arbitrary number of union operations. The average cost of $(n-1)$ union operations mixed with m *findCompress* operations is equivalent to $(m+2n-2)$ *findCompress* operations. Therefore, the total cost is $\mathcal{O}(m+n)$.

4 Summary

In this paper, we have presented a version of union-find algorithms with the path-compression strategy and given a simpler proof that its average expected run time is bounded by a linear function of the number of union and find operations. This complexity analysis indicates that our algorithm is competitive with those employing more complicated optimization strategies. In separate tests, we have demonstrated that using our array based algorithm require less memory and less time than the commonly used union-find algorithms [5]. Even though we only use a simple optimization strategy for union-find operations, because our algorithm is more compact and has more regular memory access patterns, it actually requires less time.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison - Wesley, Reading, Mass., 1974.

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

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

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

[5] Kesheng Wu, Ekow Otoo, and Arie Shoshani. Optimizing connected component labeling algorithms. In *Proceedings of SPIE Medical Imaging Conference 2005, San Diego, CA*, 2005. A draft appeared as LBNL report LBNL-56864.

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