An Implementation and Analysis of the Union-Find Algorithm: CSCI 5454 Design and Analysis of Algorithms Project

Yi-Chen Kuo

December 12, 2016

1 Introduction

Union-find algorithm, is also called disjoint-set data structure, is a data structure that keep track of a set of elements partitioned into a number of subsets. It is designed to solve dynamic connectivity problem. In general, there are two main functions of this algorithm: Find operation and Union operation. Find operation used to check if two elements are connected or not and Union operation can be used to merge two groups into one component.

In 1975, Robert Tarjan is he first person to prove the upper bound of this algorithm by the inverse Ackermann function. In 2007, Sylvain Conchon and Jean-Christophe Filliâtre developed a persistent version of the disjoint-set forest data structure. [1]

There are two main algorithms of Union-find: Quick-find and Quick-union algorithm. In quick-find algorithm, find operation takes O(1) for checking whether two items are connected and union operation takes O(N) to union two items. In quick-union algorithm, both find and union function take O(N) to operate. In order to improve the efficiency of union-find, there are two improvements called weighted union-find and weighted union-find with path compressed. They both takes some steps to improve the algorithm. In weighted union-find, the running time of find operation and union operation are both O(logN). Furthermore, path compressed makes the running time would close to 1 by amortized analysis.

In this paper, it would present implementation of four algorithms: Quick-find, quick-union, weighted union-find and weighted union-find with path compressed algorithms. Implementations of these algorithms are presenting in Python.

2 The algorithms

Union-Find algorithms is used for solving dynamic connectivity problems. Supposed there are n elements in set S and there are several subsets S_1, S_2,…, S_i of S. The concept of union (S_i, S_j) is to union these two set into one component, and find(a, b) is to check if two element are in same set.

2.1 Quick-Find algorithm

For each element that are connected, we can consider that they are in same set. On contrast, if elements are not connected, they are in different sets. As a results, using an integer array indexed by object. Suppose there are N elements, then using 0 to N-1to represents these nodes. Initially, set up the element of array equal to its index, it means all the objects are independent at first. After implementing the algorithm, the elements of array would be changed depend on the calling of union operation. (The id of each object means the group they belong)

Node	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
id	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

Consider the pseudocode below.

```
def find(a,b): \\ check if objects a and b are connected

    if arr[a]==arr[b]:
        return True
    else:
        return False
def union(a,b): \\ merge set of a and b into same component.
    aid = arr[a]
    bid = arr[b]
    for i in range(0,len(arr)): \\ obtain elements that in the same group with a
        if arr[i]== aid:
            arr[i]= bid \\change it's element of array as same as group of b
    return arr
```

In find operation, just to check if their id is equal. If so, print "yes "means they are in same component. Otherwise, print "no". In union operation, go through whole array, looking for the entries whose id are equal to the id of first argument a, and set those to the id of the second argument b. Here is some example of how the algorithm works. Suppose union (0,1) first, change the id of node 0 to 1. And find (0,1) to check if they have same id. Because the id of o and 1 are both 1, they are in same set.

Node	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
id	**1**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

Find operation is efficient, it takes O(1) to check nodes are connected. However, union operation has to check whole array to check if the id of each node are equal with node a and change the id to node b. Therefore, it takes O(n) time to operate union function. The worst case is keep adding a

single element into a set. Suppose there are M union command on N objects, then it takes quadratic time.

2.2 Quick-Union algorithm

In quick-union algorithm, using tree concept to connect each node. Each element in array is going to contain a reference to its parent node in the tree.

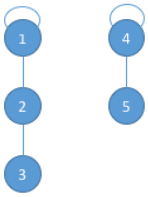
Initially, every node is the root of itself. Then use find operation to check whether the two items are in same tree. Furthermore, the process of union operation is to connect the roots of elements together. union operation only involves changing one entry (the final root of the tree) in the array. But, find operation requires a little more work, it should find the root of the node we concerned to check if these nodes are in same tree. The following pseudocode is implementation of quick-union algorithm.

```
def root(i): \\ chasing parent pointer until get the node where i is equal to id of i.
    a = i
    while(a != arr[a]):
        a = arr[a]
    return a
def find(a,b): \\ Check if a and b have the same root.
    if root(a) == root(b):
        return True
    else:
        return False
def union(a,b): \\ Set the id of a's root to the id of b's root.
     arr[root(a)] = root(b)
```

Implement root function to chase parent pointer until we get to the point where i is equal to id of i. As mentioned before, every node is root of themselves at first. So it means we find the final root. If it is not equal, we just move up 1 level in the tree continually until finding final root.

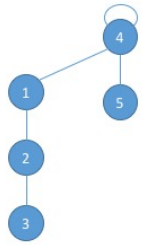
Then, the union operation is simply find the two root of both node and then set the root of the first one be the second one. In find operation, we just have to check if two nodes have same root.

Following are process of union operation of quick-union algorithm. There are two trees, {1,2,3} and {4,5}. The id of 3 points to node 2 which is parent of node 3 and id of 2 is node 1. Similarly, the root of node 5 is node 4. Besides, node 1 and node 4 are pointed to itself.



Node	0	1	2	3	4	5	6	7	8	9
id	0	1	1	2	4	4	6	7	8	9

Then, union (3,4). Use find-root function to find the final root of 3 which is node 1 and root of 5 is 4. Union these two tree by set the id of 1 to 4. 4 becomes the parent of 1. As a result, union operations just involves changing 1 entry in the array.



Node	0	1	2	3	4	5	6	7	8	9
id	0	4	1	2	4	4	6	7	8	9

In root operation, id[i] is parent of i, if the tree is long root of i is id[id[…id[i]]] until find the final root (time proportional to depth of i). Therefore, the find operation is O(N), because we have to find the root node in worst case and the tree depth can be very large. If the tree is tall, that means every node connect one by one, then the level of tree is n. Generally, find operation and union operation are O(N) in worst case, because it needs to do find-root operation to do union operation (to find the root of node, then union). [2]

2.3 Weighted Union-Find algorithm

In order to deal with situation of long tree, there is an improvement called weighted union-find algorithm to solve this problem.

The problem of quick-find algorithm is it might have long tree that cost more time to find the root. Therefore, the idea of weighted quick union is to implement the quick union algorithm, but take steps to avoid having tall tree.

If we have to union two tree together to avoid putting the large tree lower, we should keep track of the number of objects in each tree. Then link small tree below the large one.

In quick union, we connect trees to another according to the input of union, if we put tree p in first argument and tree q in second argument. Thus, it would connect tree p to q. so it might put the large tree lower which would increase the tall of tree. However, in weighted union-find algorithm, we always put the smaller tree lower. [3]

The following pseudocode is implementation of weighted union-find algorithm and the size of tree would be stored in an array called sz[].

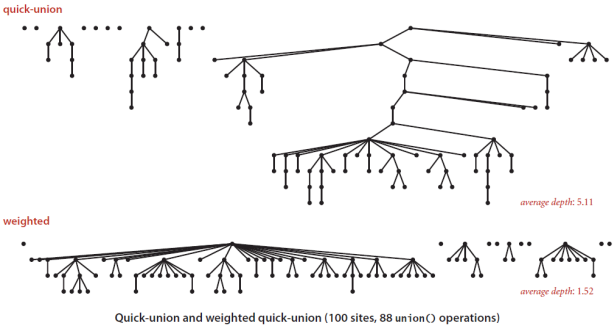
```
def root(i): \\ chasing parent pointer until get the node where i is equal to id of i.
    a = i
    while(arr[a] != a):
        a = arr[a]
    return a
def find(a,b): \\ Check if a and b have the same root.
    if root(a) == root(b):
        return True
    else:
        return False
def union(a,b): \\ link smaller tree into larger tree and update the size[] array.
    if sz[root(a)] > sz[root(b)]:
        sz[root(a)] += sz[root(b)]
        arr[root(b)] = root(a)
    else:
        sz[root(b)] += sz[root(a)]
        arr[root(a)] = root(b)
```

The find operation is identical to quick union. We use it to check if two nodes are in same tree by checking their root. The difference in weighted union-find algorithm is there is an extra array to remember the number of objects in the tree (called sz[] in implementation). And compare the size of trees to determine how to connect these two tree. Link the root of the smaller tree to the root of the larger tree in each case.

If we make b is the child of a, we have to increment the size of a by the size of b's tree. By updating the size[] array. We will know the size of tree that can be used in next comparison.

The following is the example to show the implementation of weighted.py for weighted quick union algorithm. Here, we have two tree as shown picture. The smaller tree that root is node 4 and larger tree's root is 6. Suppose we want to connect these trees. It should link the smaller one to another tree as shown in the right part of plot below.

The following is the example to show the effect of doing the weighted quick union. Where we always put the smaller tree lower. In the same set of union commands, the top picture is quick union which is a taller tree than the bottom one which all the node is within distance 4 from the root. [4]



Quick-union and weighted quick-union (100 sites, 88 union() operations)

Analyze the running time, it takes time proportional to how far down in the tree. And this guaranteed the depth of the node in the tree is at most the log(N). Here, we define rank r as the height of a tree. [5]

Lemma. Rank r has at least 2^r nodes.

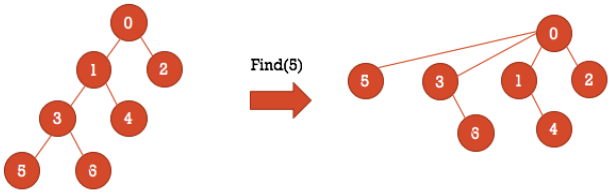
Proof. By induction on size of tree. Getting a tree with rank r, the tree is the union of two rank r-1 trees. By induction, a tree with rank r-1 has at least 2^{r-1} nodes, therefore the tree with rank r must have at least 2^r descendants. So, the height of trees is bounded by log(N).

By the lemma, the rank of tree is at most log(N). Therefore, in this algorithm both complexity of find and union are O(log N).

2.4 Weighted Union-Find algorithm with path compressed

The idea of weighted union-find algorithm with path compressed is setting all parents pointers to point to the final root on find operation. So, this algorithm improves weighted union-find algorithm.

When we ‘re trying to find the root of the tree containing node i, we will know all the nodes on the path from the node to the root. While we are doing that we can make each one of those node just point to the final root. The following is the example of how it works.



In the case, we are trying to find the root of 5. After we find the root, we make every node in path (that is 3 and 1) point to the root as well. That's going to be a constant extra cost, but it can flatten the tree by this method. This improvement is proved by Tarjan[6].

The Ackermann function is defined on m $\geq$ 0 and n $\geq$ 1. [8]

$$A(M,N)\begin{cases} N+1, if\ M=0 \\ A(M-1,1), if\ M>0\ and\ N=0 \\ A\big(M-1,A(M,N-1)\big), if\ \ M>0\ and\ N>0 \end{cases}$$

Theorem Using path compressed**,** M operation on N elements takes O($M\ \alpha$(M,N)) times.

Here defines the inverse Ackermann function α(M,N) by

$$\alpha(M,N)=\min\{i: A(i,\lfloor\tfrac{M}{N}\rfloor)\geq \log_2 N\}$$

Typically, $\alpha(M,N)\leq 4$ for most practical cases. Thus, $\alpha(M,N)$ is considered a constant. However, the prove is complex, so here introduce lg* which is simpler.

Theorem Using path compressed**,** M operation on N elements takes O(M lg* N) times.
lg*(N) is the number of times that need to apply logarithm to reduce N to be less equal than 1.

$$\lg^*N=\min\{k:\log\log\ldots\log N\ \leq\ 1\}$$

note: lg*N $\leq$ 5 for any N. Specifically, lg*(2^{65536}) = lg*($2^{2^{2^2}}$) = 5. 2^{65536} is about 20,000 digits number that would not be used practically. Because lg*N $\leq$ 5, we considered lg*N as O(1).
proof. There are two claims in following. [7]
Claim1. If node a is the parent of node b, then the rank(b) < rank(a).

Claim2. The size of tree with rank r is no more than $\frac{N}{2^r}$.

Suppose M is the total number of operations and N is the number of objects, and M $\geq N$. And there is a non-root vertex b. Define g(b) = lg*(rank(b)) and p(b) represents b's parent. Here, suppose we operate find operation on node c. This algorithm would find the path from node c to the final root and does path compression.

For each node in the path, the cost of operation is O(1):

Case1. If node b has parent and grandparent node, and g(b) = g(p(b)). Here charge 1 to node b.

Case2. Otherwise, charge 1 for find operation.

The amortized cost per find operation is at most O(lg*N). So, the total charge of M operation is bounded by O(M lg* N).

In general, any sequence of M union and find operations on N objects takes O(M lg* N) time. And the analysis can be improved to $M\,\alpha$(M, N), where α is the inverse Ackermann function which grows even slower than lg*N.

Because lg*N is never more than 5, that means the running time of weighted quick union with path compressed is going to be linear. The following pseudocode is implementation of weighted union-find with path compressed algorithm.

```
def root(i): \\ make every other node in path point to its grandparent.
        a = i
        while(arr[a] != a):
            arr[a] = arr[arr[a]]
            a = arr[a]
        return a
def find(a,b):
    if root(a) == root(b):
        arr[a]=arr[root(a)]
        return True
    else:
        return False
def union(a,b):
    if sz[root(a)] > sz[root(b)]:
        sz[root(a)] += sz[root(b)]
        arr[root(b)] = root(a)
    else:
        sz[root(b)] += sz[root(a)]
        arr[root(a)] = root(b)
```

3 Analysis

In this section, it would include the implementation of quick-find, quick-union, weighted union-find and path compression. Each implementation starts disconnected set and performed N unions randomly, and show the execute time of union operation and find operation for each algorithm.

Suppose there are N objects and N operations, the time of union operation of these algorithms are shown in Figure 1 and 2. The execute time is divided by N to normalized the data. Thus, the graph shows the time of single union operation.

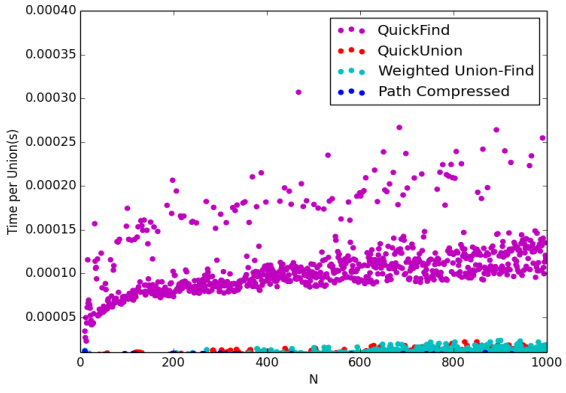


Figure 1: Union operation of Union-Find algorithms (Quick-find, quick-union, weighted union-find and path compressed

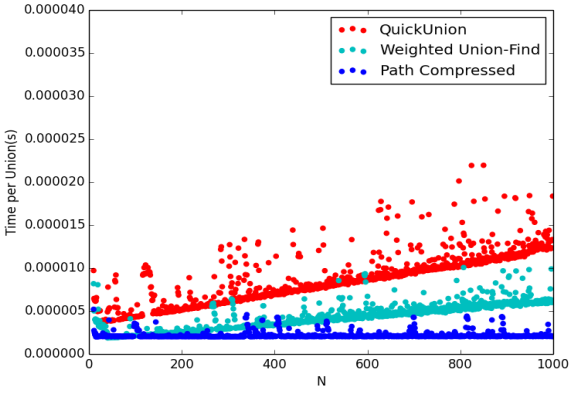


Figure 2: Union operation of Union-Find algorithms (Without quick-find algorithm)

In Figure 1, the result shows only quick-find algorithm has significant difference compared with other algorithms. Thus, in order to show the variance of quick-union, weighted union-find and path compression, Figure 2 presents the running time of these three algorithms.

As mention before, the union operation of quick-find is O(N). Therefore, comparing with other three algorithms, quick-find is not efficient. Take a closer look of comparison at Figure 2. Because weighted union-find avoid forming tall tree, which shorten the time for finding the final root. Therefore, weighted union-find improves quick-union which worst-case is O(N). Finally, path compression performs more efficient than others due to balance the tree in the process of tracking the root.

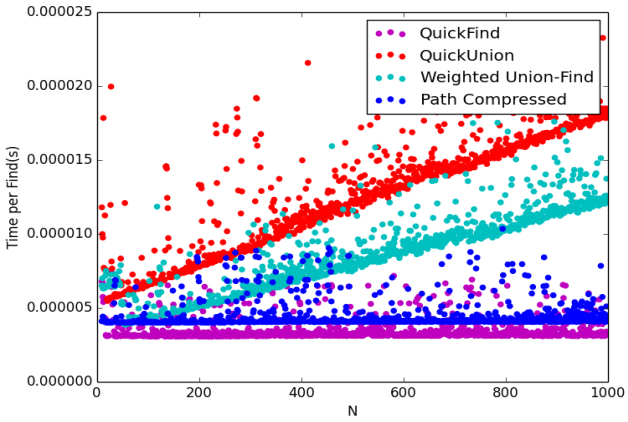


Figure 3: Find operation of Union-Find algorithms (Quick-find, quick-union, weighted union-find and path compressed

In Figure 3, the plot presents the execute time of find operation of each algorithm. It shows the find operation of quick-union is not efficient that takes O(N), which is in accordance with the performance of run time mentioned in previous chapter. Besides, weighted union-find have shorten path from node to root, the execute time of find operation is shorten as well. Finally, both quick-find and path compressed are efficient due to constant execute time.

4 Conclusion

In union-find algorithm, there are two main algorithms: quick-find, quick-union. Besides, there are two improvements to make this algorithm more efficient. And these algorithms involve two main operations: union and find, on a collection of disjoint-sets.

In general, quick find take constant time of find operation and it takes O(N) to get the entry which have same id with argument union operation. In quick union algorithm, because union operation need to do find when doing union. Therefore, both of them take O(N) time. For weighted union find algorithm, it flattens the tree, so the run time of find and union are both O(logN). Finally, the worst case of path compression is O(Mlg*N). Because lg*N is a constant, this algorithm is linear.

References

[1] History of Union-Find algorithm.

https://en.wikipedia.org/wiki/Disjoint-set_data_structure

[2] The overview of Union-find algorithm.

http://blog.csdn.net/dm_vincent/article/details/7655764

[3] Case study of union-find.

http://algs4.cs.princeton.edu/15uf/

[4] Weighted Quick Union & Quick Find Algorithm

https://www.leighhalliday.com/weighted-quick-union-find-algorithm-in-ruby

[5] Advanced Algorithm

https://www.cs.umd.edu/class/spring2011/cmsc651/lec05.pdf

[6] Princeton Algorithms Course

https://www.youtube.com/watch?v=SpqjfGTOriQ

[7] Analysis of Union-Find

https://people.eecs.berkeley.edu/~luca/cs170/notes/lecture12.pdf

[8] Ackermann Function

http://mathworld.wolfram.com/AckermannFunction.html