Union Find

Shikun Wang (Jason) and Xinran Gu

December 8, 2017

Abstract

In this project, a Union-Find data structure is implemented with both python and Java, including a slow version, a path compression version, and a union by rank version, with pseudo code from Cornell's note.

1 Introduction to Union-Find

Union and Find are two commonly used operations in the Disjoint set data structure. This data structure is able to represent the partitions of a set of nodes, and check whether two nodes are connected in a efficient way. In particular, union function connect two nodes together, joining two nodes' parents and setting them to the leader of that particular partition. And find function return the leader of the partition that a node belongs to. In this project, we are using array of size n to implement the data structure, disjoint set with size n and its two commonly used operations, union and find. And we also tried to improve the implementation by using the path compression find and union by rank methods.

2 Summary of Implementation

The ith elements of array of size n, uf, is the index of parent of node ith. So, initially, each element is equal to the its index in the array since every node is its own parent.

2.1 Slow Version

During the slow union(x, y) operation, we set the parent of y points to the parent of x by changing the value of y's parent to be x's parent. Since we can use find operation (talk about next) to find the parent of a element, the pesudo Code is as following:

$$uf[Find(y)] = Find(x)$$

And Find function is recursively calling itself to trace tree up until the root, which is its parent. And the pseudo code is as following:

```
Find(x);
if uf[x] = x then
    return x
end
else
    return Find(uf[x])
end
```

2.2 Path compression

However, the find operation is running time is not good. It is linear, tracing up the tree to root, so the depth of the tree matters. The path compression version solves this problem and makes improvement. Upon each call, it sets the node's parent as its current parents' parent. Therefore, it reduces the time to traverse each tree to get the parent next time, since the tree becomes shallower.

The pseudo code is shown below:

```
Find(x);
if uf[x] != uf[uf[x]] then
  |  uf[x] = Find(uf[x])
end
return uf[x]
```

It can be proved that the running time would be $O(k \log n)$, where k is the number of Union and Find operations.

2.3 Union by Rank

The union by rank version adds an extra array to store the rank of each node, which is initially set as 1. The rank of x is the length of tree rooted from x. When calling the Union function, it chooses to update the parent of node with lower rank to the node with higher rank. This prevents any tree from exceeding log(n), which guarantees a better running time.

3 Summary of experiments

3.1 Python Implementation

The following table is a comparison between function within my implementation

Algorithms	n=1000	n=10000	n=100000
Slow Union + Slow Find*	15.34 ms	n=9000: 961.38ms	–
Slow Union + PC Find	6.72ms	74.72ms	862.5ms
Union By Rank + Slow Find	7.025ms	68.89ms	711.25ms
Union By Rank + PC Find	6.10ms	68.61ms	719.8ms

*The slow union+slow find implementation failed with 10000 and 100000 because the maximum recursion depth exceeded.
As shown above, the path compression and union by rank improved the running time a lot.

3.2 Java Implementation

The following table is a comparison between functions within my implementation

Algorithms	n=1000	n=10000	n=100000
Slow Union + Slow Find	2.0 ms	11.0ms	3536.0ms
Slow Union + PC Find	2.0ms	7.0ms	26.0ms
Union By Rank + Slow Find	3.0ms	6.0ms	30.ms
Union By Rank + PC Find	2.0ms	7.0ms	27.ms

As you can see, the path compression algorithm actually improves the performance a lot when n is large. And also union by rank improves the performance when n is large. But when n is small, there is not much difference.

And the following is comparison with implementation on Algorithm 4th textbook.

Algorithms	n=1000	n=10000	n=1000000
My Union Find	2.0 ms	7.0ms	305.0ms
Princeton's Union Find*	3.0ms	7.0ms	323.0ms

The performance of our implementation is pretty similar to the one on Algorithm 4th textbook by Princeton University (https://algs4.cs.princeton.edu/15uf/UF.java.html).