Union Find

Ryan Jian

December 13, 2013

1 Introduction

Union find is used to implement disjoint set data structures. For our purposes, the data structure will support the following three operations:

1. $MakeSet(x)$: Create a new set containing the element x. Not matter how you do it, this is trivial.
2. $Union(a, b)$: Merge sets a and b.
3. $Find(x)$: Find the set that x belongs to.

Each set is defined by a representative element so that each operation deals only in terms of elements. $Find(x)$ returns the representative of the set x belongs to, and $Union(a, b)$ takes the representative elements of a and b as its arguments.

2 Quick-Find

Each element in the set will have a pointer to the set's representative element.

1. $Union(a, b)$: $O(N)$. Change all the pointers in a so that they point to b.
2. $Find(x)$: $O(1)$. Return the element that x points to.

One heuristic we can apply is to make $Union()$ change the pointers in the set with fewer elements. Surprisingly, this simple modification lowers the time complexity of $Union()$ to $O(N \log(N))$ for any number of calls to it when there are N elements.

3 Quick-Union

We want to improve the runtime of $Union()$, so we will try a different approach. For each element we will maintain a pointer to another element in the same set that is not necessarily the reprensentative element. This effectively constructs a linked list for each set where the head of each linked list denotes the corresponding set.

1. $Union(a, b)$: $O(1)$. Append the list of one set to the list of the other set.
2. $Find(x)$: $O(N)$. Start at x and repeatedly follow the pointer of the current element until we reach an element that points to itself, or the head of the linked list.

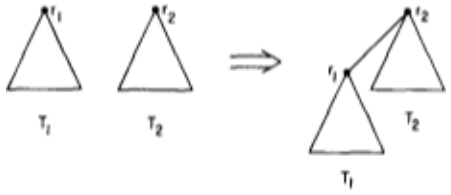
4 Disjoint-set Forests

All that Quick-Union has done is switch the time complexities for the two operations. We can reduce the time complexity of $Find()$ by building upon the linked list idea from Quick-Union and representing each set as a tree. To do this, we slightly alter $Union()$, by changing the pointer of one set to point to the other set. Because each set is a tree, this makes the tree of one set become the child of the other. $Find()$ can be kept the same, although it now stops at the root of a tree instead of the head of a linked list.

Keep in mind that this does not change the runtime of our Quick-Union data structure at all because in the worst case, the tree can be unbalanced and degenerate into a linked list.

4.1 Union By Rank

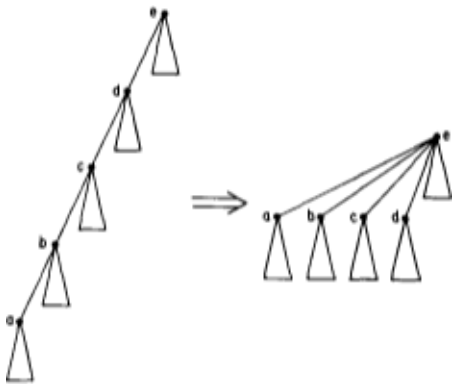
We can apply heuristic similar to what we used for Quick-Find. We change $Union()$ such that we minimize the depth, which we call the *rank*, for reasons that will become clear later. To do this, we always make the tree with the lower rank become the child of the other.



```
Union(a, b):
    a = Find(a), b = Find(v)
    if rank[a] < rank[b]:
        swap(a, b)
    parent[b] = a
    rank[a] = max(rank[a], rank[b] + 1)
```

4.2 Path Compression

We can observe that $Find()$ becomes faster the flatter the tree is, because fewer levels in the tree need to be traversed to reach the root. With this idea, we can use another heuristic: we relink every node in the upward traversal of the tree to the root. This significantly improves the runtime for future operations.



```
Find(x):
    if parent[x] != x:
        parent[x] = Find(parent[x])
    return parent[x]
```

4.3 Time Complexity

It will turn out that we can use both union by rank and path compression to improve the runtime even further. As a side note when we do this the rank of a node will no longer equal the depth of the tree rooted there (hence why we call it the rank in the first place and not the depth). Regardless, the rank is still a useful quantity, and combining the two heuristics improves the runtime of $Find()$ to amortized $O(\alpha(N))$, where $\alpha(N)$ is the inverse Ackermann function (The proof of this is very complicated, so we won't include it here). The Ackermann function grows extremely quickly meaning that its inverse grows extremely slowly, so slowly that for all practical values of n, $\alpha(N)$ is less than or equal to 5. Furthermore, it is possible to show that this is asymptotically optimal and that any pointer-based structure must have use at least $O(\alpha(N))$ amortized time for $Find()$.

5 Problems

1. Kruskal's algorithm for the minimum spanning tree of a graph.
2. (USACO OPEN08, nabor) Farmer John has N ($1 \leq N \leq$ 100,000) cows who group themselves into "Cow Neighborhoods". Each cow is at a unique rectilinear coordinate, on a pasture whose X and Y coordinates are in the range 1..1,000,000,000. Two cows are neighbors if at least one of two criteria is met: (1) If the cows are no further than some integer Manhattan distance C ($1 \leq C \leq$ 1,000,000,000) apart. (2) If cow A and B are both neighbors of cow Z, then cow A is a neighbor of cow B. Given the locations of the cows and the distance C, determine the number of neighborhoods and the number of cows in the largest neighborhood.
3. (SPOJ INVENT) Given tree with N ($1 \leq N \leq$ 15,000) vertices, find the minimum possible weight of a complete graph (a graph where every pair of vertices is connected) such that the given tree is its unique minimum spanning tree.
4. (SPOJ CPAIR) Given a sequence A of N ($1 \leq N \leq 100,000$) non-negative integers, Answer Q ($1 \leq Q \leq 100,000$) queries, where each query consists of three integers, v, a, b. The answer is the number of pairs of integers i and j such that $1 \leq i \leq j \leq N$, $a \leq j - i + 1 \leq b$, and $A[k] >= v$ for every integer k between i, and j, inclusive.