Union-Find

Justin Zhang
Daniel Wisdom

17 September 2017

1 Union Find

Given an array a of length n, with elements $1, 2, 3...n$, each representing a "group," union find consists of two operations:

- *union(p, q):* Connect two groups with elements at indices p and q; change all elements in group q to group p.
- *find(p):* Return the group of the element at index p.

Let's formalize this concept...

2 Connectivity

2.1 Connected components

In an undirected graph, we say that two vertices are *connected* if you can reach one from the other by traversing a series of edges. Then, a connected component is a subgraph such that any two vertices in the component are connected.

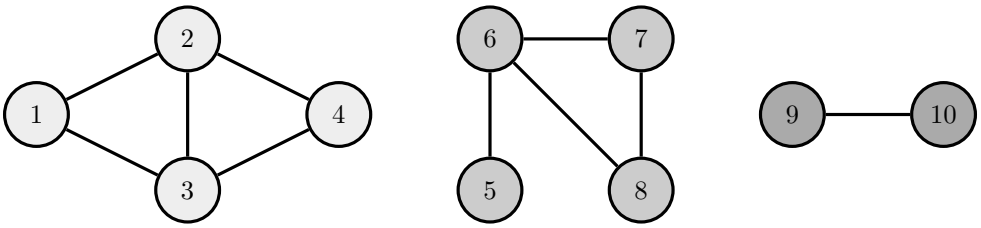

Figure 1: This diagram illustrates a graph with three connected components. *Credit: Samuel Hsiang*

Determining connectivity can be done with flood fill (DFS/BFS). Union find is concerned with the following operations (equivalent to those listed in Section 1):

- *find(p):* Determine which connected component a vertex p is in. This is easily accomplished with flood fill.
- *union(p, q):* Connect two vertices p and q in the graph.

3 Union-find (Disjoint-set)

The goal of the *union-find data structure* is to solve these two queries efficiently. Naive solutions might be to:

- Represent the problem with an solely an array (Section 4). This performs *find* in $O(1)$ (index lookup) and *union* in $O(N)$ (iterate through the entire array to change the elements).
- Represent the problem with a graph or graph-based array (Section 5). This performs both *find* in $O(N)$ (traverse the tree upward to find the root) and *union* in $O(N)$ (change the root pointer, which requires *find*).

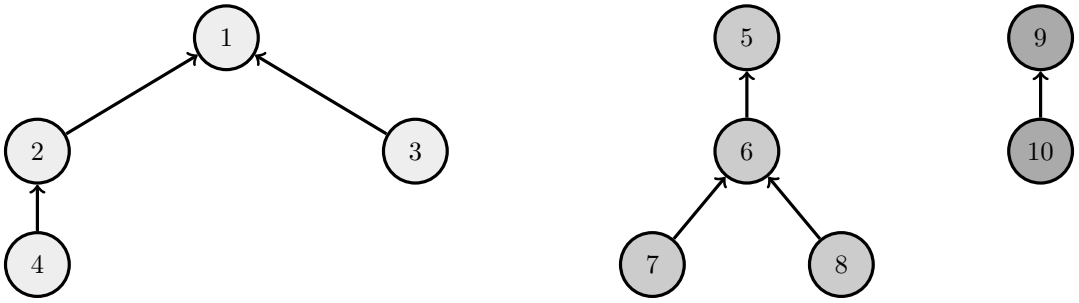

Figure 2: Representation of union-find as a forest. *Credit: Samuel Hsiang*

If we number our nodes sequentially, we can easily represent these pointers in an array instead. Instead of a null parent pointer, root nodes will point to themselves.

1	1	1	2	5	5	6	6	9	9
1	2	3	4	5	6	7	8	9	10

Figure 3: Representation of union-find as an array. *Credit: Samuel Hsiang*

4 Quick-find

One way to implement *Union* and *Find* is to have every node directly point to the root node of the component.

- *find(p):* Return the value at index p of the array, the index of the connected component's root.
- *union(p, q):* Search the array. For every element that contains the value q, replace it with p.

Now we have an algorithm that runs in $O(1)$ time for *find*, but $O(N)$ time for *union*. However, this is the same time complexity that we would get by simply performing a flood-fill every time we ran *union*. We can do better.

5 Quick-union

With quick-find, we update every element in one connected component whenever we perform a union, which requires us to search the entire array. Instead, we can take a lazier approach and only update the pointer of the root element. This takes advantage of the interpretation of union-find as a tree.

- *find(p):* Follow the parent pointer of p until we reach the representative element.
- *union(p, q):* Change the parent pointer of $find(q)$ to point to $find(p)$.

Because *find* needs to traverse the tree until it reaches a root element, its worst-case complexity is $O(N)$, proportional to the height of the tree. And because *union* requires us to call *find*, its complexity is also $O(N)$. This seems worse than Quick-find! But we can significantly improve the complexity of *find* by limiting the depth of the tree.

5.1 Weighting

The worst-case scenario with quick-union is a very large tree that takes a long time to traverse. However, this is easy to avoid. Whenever we perform *union*, if we keep track of the size of each tree, we can always join a smaller tree to the root of a larger tree, rather than the other way around.

It turns out that this optimization limits the maximum depth of any tree to $\log N$. This means that the cost of both *find* and *union* are now limited to $O(\log N)$, which is already a significant improvement.

5.2 Path compression

Intuitively, flattening the tree would make the find operation faster by shortening the number of pointers we need to traverse. So another optimization we can make is every time we perform $find(p)$, to change p and all of its parents to point to its root node. This allows us to avoid traversing the same path more than once.

Combining weighting with path compression brings down the cost of *find* and *union* to amortized $O(\alpha(N))$, where α represents the extremely slowly-growing *inverse Ackermann function*. For practical purposes, $\alpha(N) < 5$. In fact, this is asymptotically optimal: union-find in constant time is impossible.

6 Pseudocode

This is a sample implementation of weighted quick-union with path compression. *Credit: Samuel Hsiang*

Algorithm 1 Union-Find

```
function FIND(v)
    if v is the root then
        return v
    parent(v) ← FIND(parent(v))
    return parent(v)
function UNION(u, v)
    uRoot ← FIND(u)
    vRoot ← FIND(v)
    if uRoot = vRoot then
        return
    if size(uRoot) < size(vRoot) then
        parent(uRoot) ← vRoot
        size(vRoot) ← size(uRoot) + size(vRoot)
    else
        parent(vRoot) ← uRoot
        size(uRoot) ← size(uRoot) + size(vRoot)
```