Union-Find

Kevin Geng

30 September 2016

1 Connectivity

1.1 Connected components

To start our discussion, let's consider the idea of connected components. In an undirected graph[1], 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.

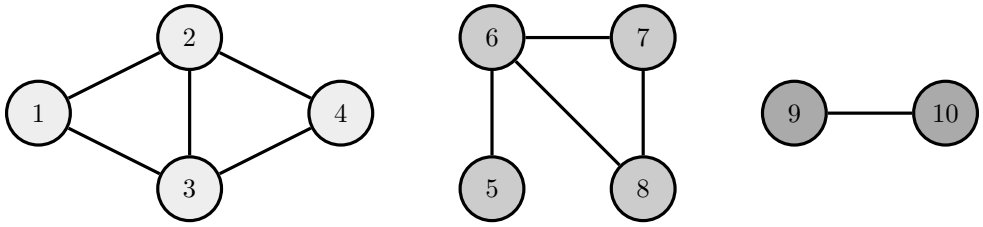

Figure 1: This diagram illustrates a graph with three connected components. *Credit: Samuel Hsiang*

To determine the connected components in a graph, we can simply perform a flood-fill on the graph. Specifically, for each of the vertices in the graph, we can determine what other vertices are connected to it using DFS or BFS, then mark them as belonging to the same component with an identifier unique to the component. This runs in $O(V + E)$ time.

1.2 Dynamic connectivity

The *dynamic connectivity* problem is an extension of the problem of connectivity. We want to be able to add edges to a graph, but also determine the connected components of that graph at any time. However, for the purposes of this lecture, we will not consider removing edges.

Alternatively, we can consider the problem in terms of two operations that we would like to perform on an undirected graph:

- *find(p):* Determine which connected component a vertex p is in. Note that we can easily determine whether two vertices are connected by checking whether they are in the same connected component.
- *union(p, q):* Connect two vertices p and q in the graph.

We can also think of this problem in terms of elements in subsets, instead of vertices in components. Though we need to know what vertices in a connected component, we do not need to know how those vertices are connected, because they are all connected to each other.[2] This frees us from the constraints of the data structures that we use for graphs, such as adjacency lists.

[1] We will consider connectivity in directed graphs later this year.

[2] This is because connectivity is an *equivalence relationship*: it is reflexive, symmetric, and transitive.

2 Union-find (Disjoint-set)

The goal of the *union-find data structure*, or the *disjoint-set data structure* is to efficiently solve the dynamic connectivity problem. Instead of using a graph representation, for each node, we can maintain a parent pointer to one other node that it's connected to. Eventually, following these pointers will lead to a root node that does not point to anything. Since all nodes in the component point to it, it is the *representative element* of the component. This forms a collection of trees, also known as a *forest*.

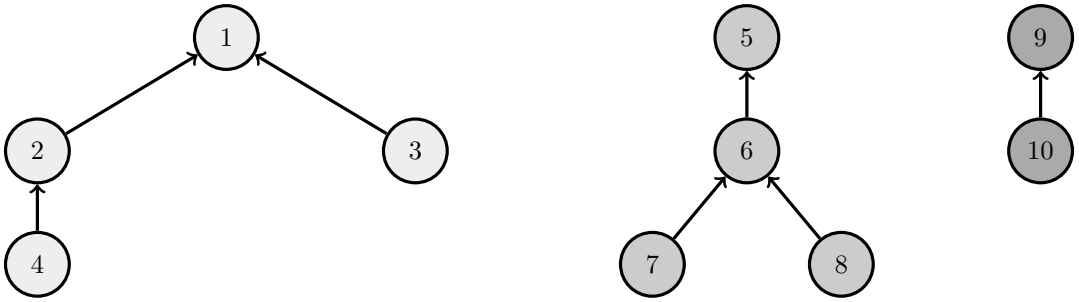

Figure 2: Representation of union-find as a forest. *Credit: Samuel Hsiang*

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*

3 Quick-find

One way to implement *Union* and *Find* is to have every node directly point to the root node of the component. In fact, this is the same data structure that we used to solve the connected components problem.

- *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.

4 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.

4.1 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 lgN. This means that the cost of both $find()$ and *union*() are now limited to $O(logN)$, which is already a significant improvement.

4.2 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.

5 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)
```