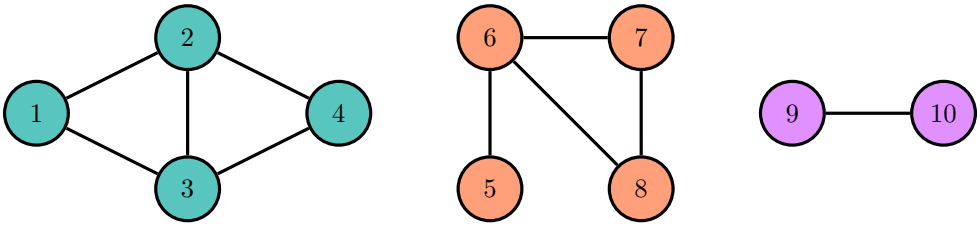
Union-Find and Kruskal

Samuel Hsiang

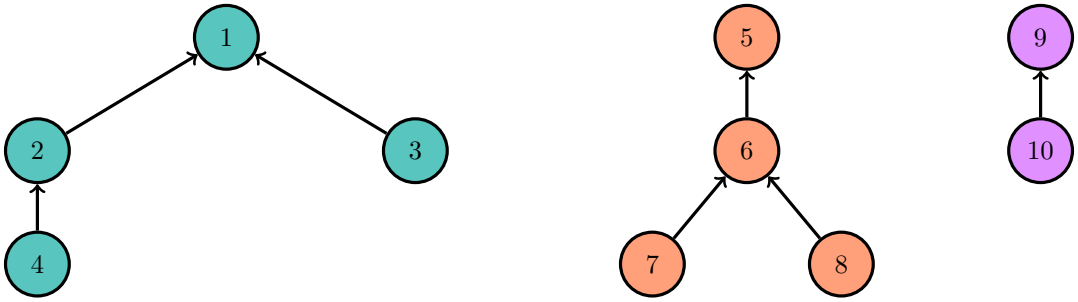
March 11, 2016

1 Union-Find (Disjoint Set Union)

A *connected component* of an undirected graph is a subgraph such that, for any two vertices in the component, there exists a path from one to the other. The diagram illustrates three connected components of a graph.



We maintain a pointer to another vertex it's connected to, forming a *forest*, or collection of trees. To check whether two elements are in the same component, simply trace the tree up to the root by jumping up each pointer.



The idea of a pointer can easily be stored within an array.

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

We want to support two operations: $find(v)$, which returns the root of the tree containing v, and $union(u, v)$, which merges the components containing u and v. This second operation is easy given the first; simply set the pointer of $find(u)$ to be $find(v)$.

But a problem quickly arises – the $find$ operation threatens to become linear. There are two simple things we can do to optimize this.

The first is to always add the shorter tree to the taller tree, as we want to minimize the maximum height. An easy heuristic for the height of the tree is simply the number of elements in that tree. We can keep track of the size of the tree with a second array. This heuristic is obviously not perfect, as a larger tree can be shorter than a smaller tree, but it turns out with our second optimization that this problem doesn't matter.

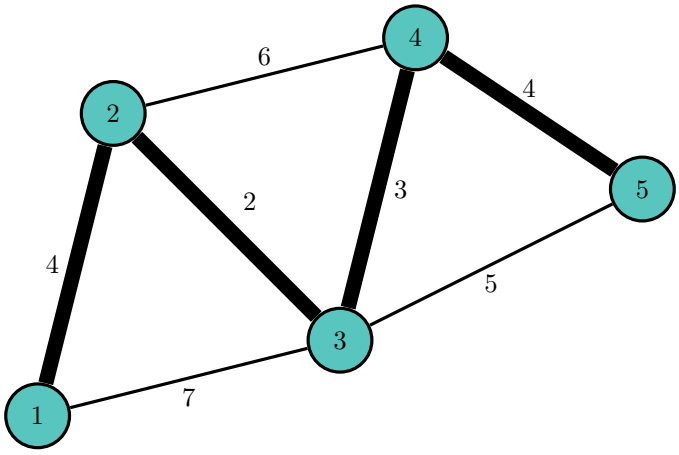
The second fix is to simply assign the pointer associated with v to be $find(v)$ at the end of the $find$ operation. We can design $find(v)$ to recursively call $find$ on the pointer associated with v, so this fix sets pointers associated with nodes along the entire chain from v to $find(v)$ to be $find(v)$. These two optimizations combined make the *union* and $find$ operations $O(\alpha(V))$, where $\alpha(n)$ is the inverse Ackermann function, and for all practical values of n, $\alpha(n) < 5$.

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)
```

2 Kruskal and Minimum Spanning Trees

Consider a connected, undirected graph. A *spanning tree* is a subgraph that is a tree and contains every vertex in the original graph. A *minimum spanning tree* is a spanning tree such that the sum of the edge weights of the tree is minimized. Finding the minimum spanning tree uses many of the same ideas discussed earlier.



Kruskal's algorithm greedily adds edges. It iterates over all the edges, sorted by weight. We need to watch out for adding a cycle, breaking the tree structure, which means we need to keep track of each vertex's connected component. If an edge connects two vertices from the same connected component, we don't want to add it to our tree. However, we have a union-find algorithm that works perfectly for this.

Algorithm 2 Kruskal

```
for all edges (u, v) in sorted order do
    if FIND(u) ≠ FIND(v) then
        add (u, v) to spanning tree
        UNION(u, v)
```

This algorithm requires a sort of the edges and thus has complexity $O(E \log E) = O(E \log V)$.