Union Find

George Tang

21 September 2018

1 Connectivity

1.1 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.

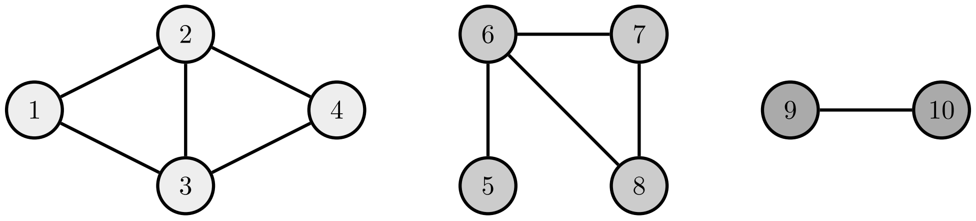

Figure 1: This diagram illustrates a graph with three connected components. *Credit: Samuel Hsiang*

Determining which components are connected can be done with DFS/BFS in $O(V+E)$ time. Pick a vertex that has not been visited, mark it with unique identifier and as visited, and recur on all connected neighbors, marking them with the same identifier. Continue this process until all vertices have been visited.

1.2 Dynamic Connectivity

Dynamic connectivity is an extension of connectivity. We want to be able to add edges and merge components, but also determine if two vertices are in the same component at any time. For the purpose of this lecture, we will not consider removing edges.

Specifically, we want to support the following operations.

- $find(v)$: Determine which component v is in.
- $union(u, v)$: Connect vertices u and v.

2 Union Find (Disjoint Set Union)

The goal of the *union find data structure*, aka *disjoint set data structure* is to solve these queries efficiently. We discuss two approaches:

- *Quick Find*: Represent the problem with an array. Every index represents a vertex and holds the component id. This performs *find* in $O(1)$ (index lookup) and *union* in $O(N)$ (iterate through the entire array to change the elements). This is equivalent to running flood fill for every union operation.
- *Quick Union*: 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 points to itself. 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.

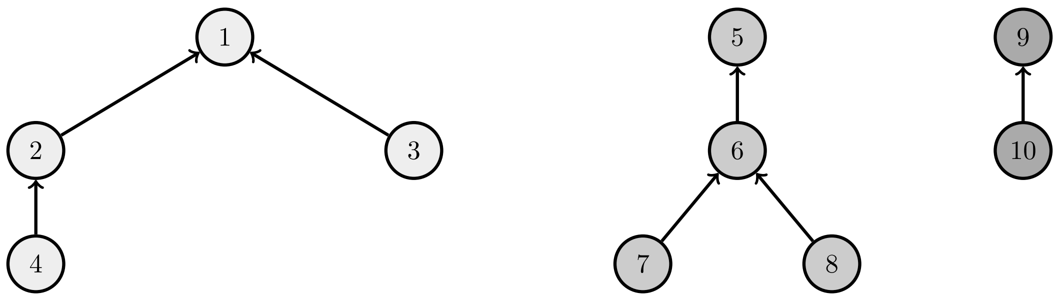

Figure 2: Representation of union-find as a forest. *Credit: Samuel Hsiang*

3 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, *quick union* takes a lazier approach and only updates the pointer of the root element. This takes advantage of the interpretation of union-find as a tree.

- *find(v):* Follow the parent pointer of v until we reach the root and return root id.
- *union(u, v):* Change the parent pointer of $find(u)$ to point to $find(v)$.

If we number our nodes sequentially, we can represent the pointers as an array.

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

Figure 3: Pointer Array. *Credit: Samuel Hsiang*

Because *find* traverses 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 is worse than Quick-find! But we can significantly improve *find* by limiting the depth of the tree.

3.1 Weighting

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. This optimization limits the maximum depth of any tree to $\log N$. This means that the cost of both *find* and *union* are now $O(\log N)$.

3.2 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 $find(v)$, to change v 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.

4 Pseudocode

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

Algorithm 1 Union-Find

```
function FIND($v$)
    if $parent(v)$ is v then
        return $v$
    $parent(v) \leftarrow$ FIND($parent(v)$)
    return $parent(v)$
function UNION($u$, $v$)
    $uRoot \leftarrow$ FIND($u$)
    $vRoot \leftarrow$ FIND($v$)
    if $uRoot = vRoot$ then
        return
    if $size(uRoot) < size(vRoot)$ then
        $parent(uRoot) \leftarrow vRoot$
        $size(vRoot) \leftarrow size(uRoot) + size(vRoot)$
    else
        $parent(vRoot) \leftarrow uRoot$
        $size(uRoot) \leftarrow size(uRoot) + size(vRoot)$
```

5 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.

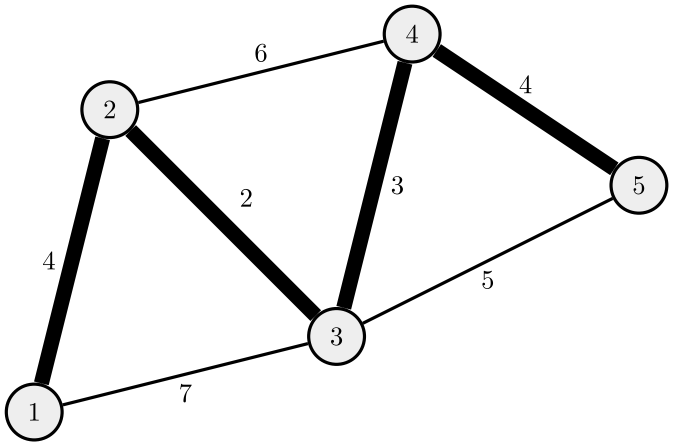

Figure 4: Figure 4: MST. *Credit: Samuel Hsiang*

5.1 Kruskal's Algorithm

We begin with every node being its own disjoint set. We construct a MST by greedily adding edges from lowest to highest weight. However, a tree is defined as having no cycles (cannot leave a node from an edge and enter through a different edge), so if two nodes are already in the same component, we disregard that edge. The MST of n nodes always contains exactly $n - 1$ edges, so we can stop looking through more edges after adding $n - 1$ edges to the MST. Conveniently, union find works perfectly for this. This algorithm requires sorting the edges and thus has complexity $O(ElogE + E\alpha(V))$.

Algorithm 2 Kruskal

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

5.2 Example

Moocast (USACO Gold, December 2016)

Farmer John's Cows ($1 \leq N \leq 1000$) each have a walkie talkie. They will spend X dollars such that each walkie talkie has transmission radius $\sqrt{N}$. Each cow can communicate another as the cows can relay their message through some sequence of transmissions. Determine the minimum cost such that every cow can communicate with another.

5.3 Example 2

Superbull (USACO Silver, February 2015)

Given $N \leq 2000$ teams and the "excitement" caused by a match between every pair of teams, find the most excitement possible in a tournament. The loser is eliminated after every game, but you get to pick who loses.

6 Monotonic Queries

You are given a graph and a set of chronological queries (by time, value, etc) that ask about the state of the components at that given state.

All we have to do maintain the state of the components using union find over the chronology. In fact, this is exactly how we build MSTs (state is weight of edge, query is do we have one component given we follow the rules for MSTs?).

6.1 Offline Queries

What if they are not in order? If all you need to do is read the queries and answer them within the time limit, it is an easy extension to monotonic queries. Store the initial order of the queries, sort the queries, and answer them like monotonic queries. Each answer is mapped to its respective query. Then, loop through the initial order, extracting the respective stored answers.

6.2 Example

Mootube (USACO Gold, January 2018)

Farmer John created Mootube, where every two connected videos have a relevance value. He wants to filter content using this metric. Specifically, given a value k and the video the user is currently watching, connected videos are suggested to a user if their relevance is greater than k. Moreover, this process is recursive, meaning if a third video not directly connected to the first is relevant to some chain of relevant videos, it is also relevant.

Given $1 \leq Q \leq 100000$ queries, each with two values, v, k, representing the current video and metric used, how many relevant videos are there?