Union Find and MST

Joshua Zhang, with contributions from Neeyanth Kopparapu

November 2022

1 Introduction

Union find, also known as disjoint set union, is a relatively simple yet useful algorithm for the Gold and Platinum divisions. Union find problems are usually tree or graph problems, but it can be applied to a variety of situations.

Given labeled sets containing distinct elements, union find consists of two functions:

- *find(a)* returns the label of the set that a belonged to.
- *union(a, b)* combines sets a and b, meaning all elements in set b now belong to set a, and set b effectively disappears.

2 Union Find

2.1 Naive Solutions

If we maintain the sets as lists, we can quickly come up with this naive solution:

- *find(a)* search all of the sets for the element, $O(N)$ time.
- *union(a, b)* combines two sets, $O(N)$ time.

Storing the set each element belongs to, we can achieve better performance:

- *find(a)* look up which set a belongs to, $O(1)$ time.
- *union(a, b)* change the stored table and combine sets, $O(N)$ time.

These all work, but usually we will to be more efficient than $O(N)$ in order to pass the test cases.

2.2 Pointer Representations

In order to optimize further, we need to think about the problem a bit differently. Let's represent the sets as a forest of trees. When we do *find*, we are looking for which tree a node belongs to, and when we do *union*, we are combining trees.

We will represent our trees as an array of pointers, so $array[i] =$ parent of i, where i is a node in the graph. If i has no parents, then $array[i] = -1$.

Under this representation, Union find is as follows:

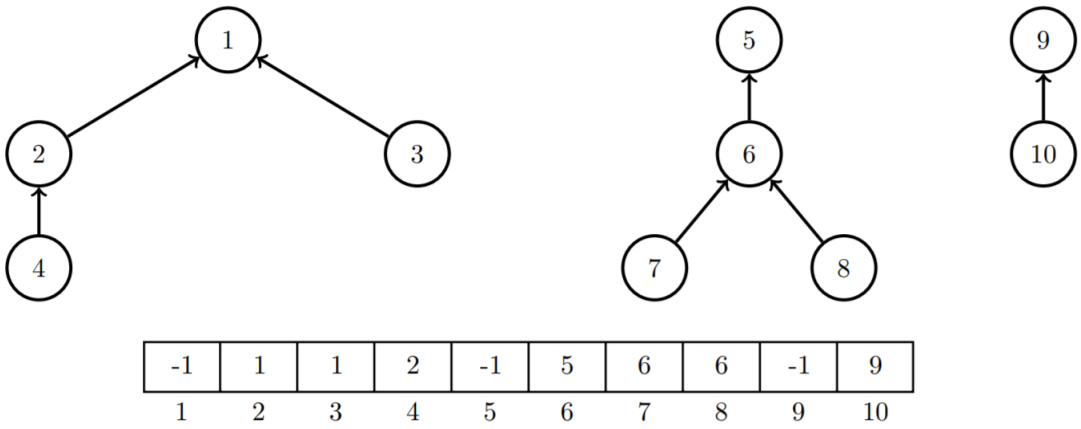


Figure 1: Example Trees and Pointer Representation

- *find(a)* find the pointer for a, traverse up the tree until we reach the root, return the root, which is the label for the tree. Since our tree can have any height, $O(N)$ time.
- *union(a, b)* connect node *find(b)* to *find(a)*, $O(N)$ time due to calling *find*.

2.3 Path Compression

If we traversing up a tree to get the the root during *(find)* multiple times, we are probably wasting time going up nodes we've already done. We can optimize our program by flattening the path. After we run *find(a)*, we can set the node that a points to to *find(a)*. If we build our trees from scratch using *find* and *union*, both will actually have $O(log(N))$ worst case performance.

2.4 Union By Rank

Another optimization we can do is restricting the height of our trees. We can do this by always adding onto the taller tree when *union* is called. Since path compression can mess with how tall our trees are, we should instead use rank. Rank starts off at the height of our initial tree. When two trees with the same rank are joined, the resulting rank is one higher. Otherwise, the rank is just the higher of the two. Doing union by rank guarantees that the height of our trees is no larger than $log(N)$, making *union* and *find* both $O(log(N))$. However, there are occasionally cases where we may want to keep track of the labels as we join trees. In these cases, we should not use union rank.

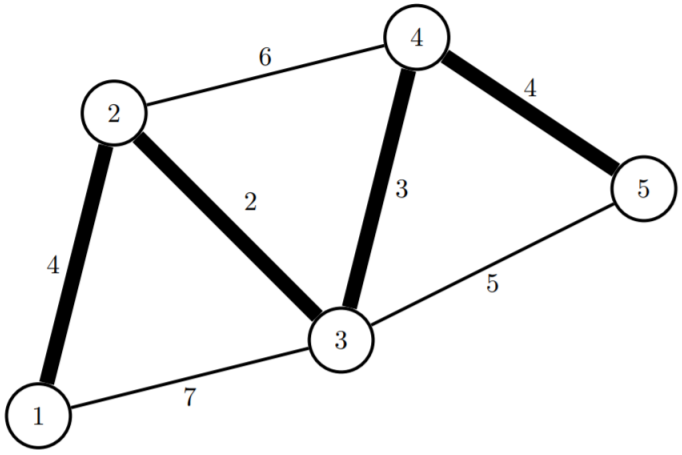
Using both optimizations turns the runtime of both operations to $O(\alpha(V))$, where $\alpha(V)$ is the inverse Ackerman function. For all intensive purposes $\alpha(V) \leq 5$, so this makes is basically constant. However, most of the time you can get away with using just one.

3 Minimum Spanning Tree

3.1 Introduction

The minimum spanning tree is a common application of union find, and also a topic in it's own right, appearing in both the Gold and Platinum divisions of USACO.

The problem is as follows: given a connected, weighted, and undirected graph, what is a way to connect all the nodes such that the sum of the weights of the edges used is as small as possible?



3.2 Kruskal's Algorithm

The idea of Kruskal's Algorithm is to sort the edges, and then go through them from smallest to biggest. Whenever we encounter an edge that would add a node to our tree, we use it to connect that edge to our tree. We skip any edges that would add a node already in our tree. The algorithm relies on union find to efficiently check whether or not nodes are in the tree. Each node starts off as its own set, and gradually gets added to the tree. If *find(a) = find(b)*, we know that they are both in the tree.

The complexity is thus $O(E \log E)$. There is also Prim's algorithm, which is conceptually similar but reverses the role of nodes and edges. We won't cover that, since it runs in $O(N^2)$ or $O(Elog(N))$ time, and very rarely is Kruskal's not a viable alternative.

4 Problems

- USACO 2014 March Contest, Silver Problem 1. Watering the Fields
- USACO 2011 December Contest, Gold Division Problem 2. Simplifying the Farm
- USACO 2014 January Contest, Gold Problem 3. Ski Course Rating
- USACO 2016 December Contest, Gold Problem 1. Moocast
- USACO 2020 January Contest, Platinum Division Problem 1. Cave Paintings