1 Introduction

Trees are one of the most significant classes of graphs. The importance of this widely used data structure is shown in its many applications, specifically in theoretical computer science.

A graph with no cycles is said to be acyclic. A tree is a connected, acyclic graph. The edges of a tree are called branches. Suppose we have a disjoint collection of trees, something known as a forest.

We want to know if two nodes are part of the same component. We would find this in the following way:

• \textit{find}(u): return the label/group of node \textit{u}
• \textit{union}(u, v): merge the components of \textit{u} and \textit{v}

We will go through some ways to optimize the performance of this and its applications.
2 Union Find

In a graph, keep pointers to parents of nodes (arbitrarily choose parents).

![Figure 3: Graphs](image)

We can then get \( \text{find} \) by tracing parents until we get to \(-1\), implying we hit the parent node of the component.

Doing the union is also simple. If we had to do \( \text{union}(3, 6) \), all we essentially have to do is set the parent of \( \text{find}(3) \) to \( \text{find}(6) \) (or vice versa).

However, we run into a problem - \( \text{find} \) starts to grow linearly. This is a problem in our runtime, since we are going for approximately constant complexity.

2.1 Optimizations

We are going to go back to the pointer representation, because that is what we will optimize. We can make two fixes:

- 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.

- The second is a bit more tricky. Assign the pointer associated with node to be \( \text{find}(\text{node}) \) at the end of the find operation. We can design \( \text{find}(\text{node}) \) to recursively call \( \text{find} \) on the pointer associated with node, so this fix sets pointers associated with nodes along the entire chain from node to \( \text{find}(\text{node}) \) to be \( \text{find}(\text{node}) \).

It turns out that these two optimizations turn the run-time of both operations to \( O(\alpha(V)) \), where \( \alpha(V) \) is the inverse Ackerman function, and for all intensive purposes \( \alpha(V) \leq 5 \), so this makes it approximately constant.

The pseudocode for the Union Find Algorithm is shown below.
3 Minimum Spanning Tree (MST)

Consider a connected, undirected, and weighted graph G. A spanning tree is a subgraph that is a tree including all the vertices of G. The minimum spanning tree (MST) is a spanning tree of G such that the sum of the edge weights is minimized. Finding the minimum spanning tree uses many of the same ideas discussed earlier. Note that if not all edge weights are distinct, then there may be multiple MSTs for a given graph.

```
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)
```

3 Minimum Spanning Tree (MST)

Consider a connected, undirected, and weighted graph G. A spanning tree is a subgraph that is a tree including all the vertices of G. The minimum spanning tree (MST) is a spanning tree of G such that the sum of the edge weights is minimized. Finding the minimum spanning tree uses many of the same ideas discussed earlier. Note that if not all edge weights are distinct, then there may be multiple MSTs for a given graph.
3.1 Kruskal’s Algorithm

Kruskal’s Algorithm finds the minimum spanning tree by greedily adding edges; for all edges not yet in the minimum spanning tree, we can repeatedly add the edge of minimum weight to the minimum spanning tree except when adding said edge forms a cycle (which violates the tree structure). In other words, it adds edges in increasing weight, skipping those whose addition would create a cycle.

- First, we need to sort the edges by ascending edge weight.
- Next, walk through the sorted edges and look at the two nodes the edge belongs to. If the nodes are already unified, we do not include this edge. Otherwise, we will include it and unify the nodes.
- The algorithm terminates when every edge has been processed or all the vertices have been unified. We can easily determine whether adding an edge will create a cycle in constant time by using Union Find.

Note that since the most expensive operation is sorting the edges, the computational complexity of Kruskal’s Algorithm is $O(E \log E)$.

To see why this will always work, assume that we are trying to add edge $e$, which connects vertices $u$ and $v$, to the MST. If the only path between $u$ and $v$ is through $e$, then adding $e$ cannot form a cycle, and by Kruskal’s Algorithm we will add $e$ to the MST. However, assume that another path from $u$ to $v$ exists. This path consists of a sequence of edges. By Kruskal’s Algorithm, any of these edges with weight less than $e$ are already in the MST. If all of these edges have weight less than the weight of $e$, then we skip over $e$ since adding it would create a cycle. Otherwise, note that $e$ has a lower weight than any of the edges in this path that are not yet in the MST, so adding $e$ is optimal.

The pseudocode for Kruskal’s Algorithm is shown below.

**Algorithm 2 Kruskal**

1: for all edges $u$, $v$ in sorted order do  
2: if $\text{FIND}(u) \neq \text{FIND}(v)$ then  
3: add $(u, v)$ to spanning tree  
4: end if  
5: $\text{UNION}(u, v)$  
6: end for

4 Prim’s Algorithm

Rather than greedily adding edges, Prim’s algorithm greedily adds vertices; on each iteration, we add the vertex that is closest to the current MST until all vertices have been added. The process of finding the closest vertex to the MST can be done efficiently using a priority queue in $O(\log N)$. After removing a vertex, we add all of its neighbors that are not yet in the MST to the priority queue and repeat. To begin the algorithm, we simply add any vertex to the priority queue. Note that Prim’s algorithm has complexity $O(E \log V)$ since in the worst case every edge will be checked and its corresponding vertex will be added to the priority queue. Alternatively, we may linearly search for the closest vertex instead of using a priority queue. Each linear pass runs in time $O(V)$, and this must be repeated $V$
times. Thus, this version of Prim’s algorithm has complexity $O(V^2)$.

To see why Prim’s algorithm works, consider a cut of the graph partitioning the graph into two sets of vertices A and B. Now consider the set of edges E connecting a vertex in A to a vertex in B. Note that at least one edge in E must be in the MST. This means that the edge in E with minimum weight must be in the MST. To prove Prim’s Algorithm, make A the set of vertices currently in the MST and B the set of all other vertices. Adding the vertex closest to the current MST is equivalent to adding the edge of minimum weight between A and B. Prim’s Algorithm follows by repeating this process.

The pseudocode for Prim’s Algorithm is shown below.

![Algorithm 2 Prim](image)

5 Problems

- USACO 2011 December Contest, Gold Division Problem 2. Simplifying the Farm
- USACO 2014 March Contest, Silver Division Problem 1. Watering the Fields
- USACO 2014 January Contest, Gold Division Problem 3. Ski Course Rating
- USACO 2015 February Contest, Silver Division Problem 3. Superbull
- USACO 2016 December Contest, Gold Division Problem 1. Moocast
- USACO 2016 February Contest, Platinum Division Problem 2. Fenced In