Introduction

Let $G = (V, E)$ be a connected, weighted, undirected graph. A spanning tree $T$ of $G$ is a graph that contains all the vertices of $G$, but only a subset of edges of $G$ such that $T$ is a tree.

A minimum spanning tree of $G$ is the spanning tree of a graph that has, out of all possible spanning trees, minimum edge weight total.

Union-Find

The disjoint sets (union find) data structure organizes a collection of elements into disjoint sets. Every element is in a set, and each set has a name, or a representative, an element in the set that the set is referred to.

Make-Set\((x)\) creates a new set with one member $x$ and with representative element $x$. $x$ cannot be in another set, as all sets are disjoint.

Union\((x, y)\) unisons the two sets containing $x$ and $y$. Let $S_x$ be the set that contains $x$, and $S_y$ be the set that contains $y$. If these two sets are not disjoint, then they must be the same set (because non-disjoint sets are not allowed), and thus no change is made. Otherwise, a new set $S$ is created with the elements in $S_x \cup S_y$, with any valid representative element. The original sets $S_x$ and $S_y$ are destroyed.

Find-Set\((x)\) returns a pointer to the representative element of the set containing $x$. If $x$ and $y$ belong to the same set, Find-Set\((x)\) will return the same element as Find-Set\((y)\).

There are two main ways to implement union-find: linked lists and forests. Using linked lists, each list corresponds to a set, the head of a list is the representative element of the set, and every element in the list points to the next element as well as the head of the list.

Using forests, we represent each set as a rooted tree. The root of the tree is the representative element, and each element points to its parent. The root’s parent is itself.

Problem. What is the running time of the above three operations if linked lists are used to represent sets? What if trees are used?

Solution. Using linked lists, Make-Set and Find-Set both run in $O(1)$ time. Union runs in $O(n)$ time, as in the worst case, every element needs to be updated to point to the head of the list.

Using a forest, Make-Set also runs in $O(1)$ time. Find-Set must traverse up the tree to find the representative element, and so takes worst case $O(n)$ time. Union requires us to find the set of two elements, and update the parent of one of the roots to the other, taking $O(n)$ time as well.

At first glance, it seems forests have suboptimal runtime. However, we can introduce two improvements that will make the runtime of using forests asymptotically better than using linked lists.

Union by rank We define the rank of a node to be the depth of the node - with the restriction that the rank does not decrease, even if the depth of the node decreases. Let $T_x$, $T_y$ be the two roots of the trees $x$ and $y$ belong to. If the rank of $T_x$ is smaller than $T_y$, then $T_x$ becomes a child of $T_y$, and vice versa. If they have the same rank, then we arbitrarily make one the child of the other, and the rank of the parent is incremented. This has the effect of always joining a shorter tree to a larger tree.

Path compression Whenever we call Find-Set\((x)\), we update $x$’s parent to be the root of the tree (the representative element). This has the effect of flattening a tree so that successive calls to Find-Set\((x)\) take constant time.
It can be proven that a series of $m$ \textsc{Make-Set}, \textsc{Union}, and \textsc{Find-Set} operations on $n$ elements takes worst case $O(m\alpha(n))$ time, where $\alpha(n)$ is the inverse Ackermann function. $\alpha(n)$ grows incredibly slowly, and in practice, never exceeds the value 4. Thus the amortized time taken for a single operation is almost constant!

The proof is incredibly complex, and is covered in section 21.4 of the textbook.

\section*{Generic MST Creation}

Suppose we want to create a minimum spanning tree algorithm. We might suppose that a good way to accomplish this would be to maintain an invariant that the set of edges we accrue during construction is always a subset of a valid MST. If at each iteration, we add an edge while maintaining this invariant, we will eventually arrive at a complete minimum spanning tree. We can represent the pseudocode as follows:

- Initialize an empty set $S$
- As long as $S$ is not yet a minimum spanning tree, repeat:
  - Find some edge $e$ such that $S \cup e$ is a subset of an MST.
  - $S = S \cup e$

While this doesn’t tell us much about the actual implementation, it is clear why the resulting edge set is correct. The issue comes when determining our edge $e$. We can use the \textit{cut property} to help determine $e$.

\textbf{Cut property} Let $S$ be a set of edges that belongs to some minimum spanning tree of a graph $G$. Given a cut $C$ of $G$ that respects $S$, let $e$ be an edge crossing the cut with minimum weight. Then, $S \cup e$ also belongs to some minimum spanning tree of $G$.

We will now look at two algorithms that use the cut property to determine MSTs.

\section*{Kruskal’s Algorithm}

Kruskal’s algorithm operates on the idea of combining connected components with edges of increasing weight. We begin with each vertex belonging to its own component. We then consider edges in order of increasing weight. If an edge does not create a cycle, we add it to our tree. Otherwise, we continue to the next edge.

We can see how this algorithm uses the cut property. At each iteration of the algorithm, we consider a certain edge $e = (u, v)$. If $u$ and $v$ belong to the same component already, then we don’t add $e$. Otherwise, we create a cut that respects the edges in the connected component containing $u$ and the connected component containing $v$. Group all other vertices by component and add the component to any one of the cut sets arbitrarily. $e$ must be the minimum weight edge crossing this cut (otherwise $u$ and $v$ would already be in the same component), and so $e$ belongs to the MST.

This algorithm lends itself well to the use of the disjoint sets data structure, where the sets represent connected components. We can use the \textsc{Find} operation to determine if two vertices belong to the same component, and the \textsc{Union} operation to merge two connected components. Thus, if we are given the edges of the graph in sorted order, Kruskal’s takes $O(|E|\alpha(|V|))$ due to $O(|E|)$ total \textsc{Union} and \textsc{Find} operations on $|V|$ items. If we must also sort the edges, then the running time increases to $O(|E| \log |V|)$.

\section*{Prim’s Algorithm}

Instead of connecting smaller components to create a spanning tree, Prim’s algorithm expands a single component until it becomes a tree. It first begins with a randomly chosen vertex to add to our subtree. Out of the edges that connect vertices not in the tree to vertices in the tree, we add the one with minimum weight. We continue until all the vertices are in the tree.

\footnote{https://en.wikipedia.org/wiki/Ackermann_function#Inverse}
Problem. How does Prim’s algorithm use the cut property to return a valid MST?

Solution. After adding an edge to the tree we have created a cut – one containing the vertices currently in the tree, and one containing all other vertices. Prim’s algorithm then selects the smallest weight edge crossing this cut, which, by the cut property, must be part of the MST.

Instead of iterating through all valid edges at each step, we can assign a value to a vertex denoting the minimum weight edge it takes to connect that vertex with the tree. At every iteration, we update these values accordingly. By using binary heaps to keep track of weights, Prim’s algorithm runs in \( O(|E| + |V| \log |V|) \) time.

Discussion Topics

- Is it guaranteed that a call to \texttt{Find(v)} will always return the same result throughout the algorithm? If not, is it possible to modify the algorithm such that it does?

  Solution. It is not guaranteed. Whenever a \texttt{Union} is performed, one of the subsets will always change its indicator value. It is not possible to prevent this for all vertices, as the nature of the methods requires indicators to change, but it is possible to modify the algorithm such that one particular indicator is maintained.

  We can add a condition around our set precedence to state that if the \texttt{Union} of two sets is performed, and one of the sets has our desired constant indicator, then force the other set to become a child of the first. Note that this will create inefficiencies, as we can no longer guarantee that the heights of our trees only increase in cases of equal height.

- Does Kruskal’s algorithm work on a graph with negative weights? What about Prim’s?

  Solution. Yes, both algorithms will select the appropriate edges in order.

Problems

Problem 1. Say we have some MST, \( T \), in a positively weighted graph \( G \). Construct a graph \( G' \) where for any weight \( w(e) \) for edge \( e \) in \( G \), we have weights \( (w(e))^2 \) in \( G' \). Does \( T \) still remain an MST in \( G' \)? Prove your answer. Now if \( G \) also had negative weights, would your answer change from the previous part? Prove your answer.

Solution. If \( G \) only has positive weights, then this claim holds. Proof by contradiction: assume the claim does not hold. Let us say we are using Kruskal’s algorithm (similar argument can be used for Prim’s). Consider the first edge where the algorithms running on \( G \) and \( G' \), diverged. Then we must have that the algorithm selected some edge \( e' \) within \( G' \) instead of \( e \) within \( G \). Then \( w(e) < w(e') \) but \( w(e)^2 > w(e')^2 \). Note that this is not possible for positive integers, and thus we have a contradiction. However, this claim is possible for negative integers! Thus, if \( G \) had negative weights, our answer would change to no, \( T \) is not necessarily an MST in \( G' \).

Problem 2. Let \( G \) be a graph with distinct edge weights. Does any valid MST contain the maximum weighted edge of \( G \)? Does every valid MST contain the minimum weighted edge of \( G \)?

Solution. If the maximum weighted edge of \( G \) is the only edge crossing some cut, then it must be included in some MST. Otherwise, the MST would not be connected.

Because the graph has distinct edge weights, by the cut property, all MSTs will contain the minimum weight edge.

Problem 3. Suppose we wish to find the maximum spanning tree - the spanning tree of maximum weight. Design an algorithm to find the maximum spanning tree in the same amount of time it takes to find the minimum spanning tree.
Solution. We can multiply all edge weights by -1, and then use Kruskal’s or Prim’s algorithm to find the minimum spanning tree of this augmented graph.

**Problem 4.** Imagine we have a graph $G$ where all edge weights are equal. Design an algorithm to efficiently find an MST of $G$. Analyze the running time.

*Solution.* Our optimal algorithm would be to run DFS and only keep track of the tree edges (so we don’t introduce any cycles). Notice at any step we can choose any edge, since the edge weights are all equal. The running time is then $O(E + V)$.

**Problem 5.** Suppose that we have found an MST $T$ of a graph $G$, but we are told afterwards that one of the edges $e$ in the graph $G$ has changed its weight. Is our tree $T$ still valid? If not, how can we modify $T$ to get a valid MST without rebuilding a new MST?

*Solution.* If the changed edge is in $T$, and has decreased in weight, then $T$ is still valid. If the changed edge is not in $T$, and has increased in weight, then $T$ is still valid.

If the changed edge is in $T$, and has increased in weight, then $T$ is no longer valid. Let us remove $e$ from $T$. We now have a disconnected graph as well as a cut. Let us select an edge crossing the cut of smallest weight and add it to $T$. Then, $T$ is valid.

If the changed edge is not in $T$, and has decreased in weight, then $T$ is no longer valid. Let us add $e$ into $T$. We no longer have a tree, and have a cycle $C$. Traverse the cycle to find the edge with maximum weight, and remove it from $T$. $T$ is once again a valid MST.