Kruskal's/Disjoint Sets Worksheet

Kruskal's Algorithm

An algorithm that finds the **minimum spanning tree** of a graph. A spanning tree is a tree that connects all vertice of the graph. We define the weight of a spanning tree as the sum of the weights of all edges in the tree. A MST is a spanning tree of a graph that has the least weight.

The main idea of Kruskal's Algorithm is to select edges with the smallest weight to be in our tree, until a MST is formed.

Given a graph G, the algorithm works as follows:

1. Let T be an empty list of edges. This is where you will store the edges for your final MST.
2. Make a sorted list `SortedEdges` of all the edges in G, from smallest to greatest.
3. For each edge (`u->w`) in `SortedEdges`
   (a) If `u` and `w` are not already connected by the edges in T, then add the edge (`u->w`) to T.
   (b) If `u` and `v` are already connected by the edges in T, then continue.

We can see that the runtime of Kruskal's Algorithm is dependent on the sorting step in (2), and the check in (3a/b) to see if two vertices are already connected by T.

For a graph with E edges, Kruskal's actually runs in $\Theta(E \log E)$, the time it takes to sort the edges. In other words, we have a pretty fast scheme to do steps (3a/b). Let us see what data structure we use to do this:

Disjoint Sets

Disjoint sets gives us a fast way to define 'groups' of objects. If A, B, C are in the same set, we can think of them as being in the same group. To give this set a name, let's choose one of these elements as the *representative* of the group. If A is the representative of the set, then we can call the set of A, B, C as set A.

This is the intuition behind disjoint sets. We will represent sets as a tree-like structure. Each element is itself a node, that may point to another node. The root of any tree-set is the representative of the set. There are three operations we want our disjoint set to support: `MakeSet`, `Find`, and `Union`. We will now examine a particular implementation of disjoint sets: Weighted Quick-Union Trees

Weighted Quick Union Tree Operations

- `MakeSet(n)`

  1. Make n disjoint sets, numbered 0 to $n-1$

- `Find(n)`

  1. Find and return the root node of n
  2. **Path Compression** optimization:
     As you follow the parent pointers from n to the root, reassign the parent pointers of n and all other nodes you see along the way to be the root.
     Purpose: Prevent our tree from getting too tall.

- `Union(a, b)`
  Goal: merge the set of a with the set of b

  1. Let `root1 = find(a)`
  2. Let `root2 = find(b)`
  3. **Union by size** optimization:
     If `root2` has a smaller set size than `root1`, then set the parent pointer of `root2` to point to `root1`.
     Else, set the parent pointer of `root1` to point to `root2`.

Notice that both of the optimizations introduced have the goal of preventing the tree from getting too tall.

Runtime (including both optimizations):

- One call to `find`:
  Worst case $\Theta(\log(n))$
  Best case $\Theta(1)$

- One call to `Union`:
  Worst case $\Theta(\log(n))$
  Best case $\Theta(1)$

- For f calls to `find` and u calls to `union`:
  Essentially $\Theta(u + f * \alpha(f + u, u))$

Where α is the inverse Ackerman function, a function that grows veeeeeeeeeeeeerrrrrrrrryyyyyyyyyy slow.

With respect to Kruskal's Algorithm: Assume that when we first start the algorithm, all vertices are their own disjoint set. Suppose that whenever we add some edge (`u->w`) to T, we also `union(u,w)`. Thus, realize that in order to see if some u and w are already connected by the edges in T, we need only check to see if they are in the same set!

Practice Problems

1. **MST and Shortest Path questions**

   Answer the following questions regarding MSTs and shortest path algorithms for a **weighted, undirected graph**. If the question is T/F and the statement is true, provide an explanation. If the statement is false, provide a counterexample.

   (a) (T/F) If all edge weights are equal and positive, breadth-first search starting from node A will return the shortest path from a node A to a target node B.

   (b) (T/F) If all edges have distinct weights, the shortest path between any two vertices is unique.

   (c) (T/F) Adding a constant positive integer k to all edge weights will not affect any shortest path between vertices.

   (d) Draw a weighted graph (could be directed or undirected) where Dijkstra's would incorrectly give the shortest paths from some vertex.

   (e) (T/F) Adding a constant positive integer k to all edge weights will not affect any MST of the graph.

   (f) (T/F) Kruskal's algorithm works even when there are negative edge weights in the graph.

   (g) Extra for experts: Design an efficient algorithm for the following problem: Given a weighted, undirected, and connected graph G where the weights of every edge in G are all integers between 1 and 10, and a starting vertex s in G, find the shortest path from s to every other vertex in the graph.

   Your algorithm must run asymptotically faster than Dijkstra's

2. **Disjoint Sets Questions**

   (a) Draw the Weighted Quick Union object that results after the following four method calls:

   ```
   connect(1, 3)
   connect(0, 4)
   connect(0, 1)
   connect(0, 2)
   ```

   (b) In terms of runtime, what is the worst way to place the integers 1, 2, 3, 4, and 5 into the same set? Your answer should be in the form of a series of calls to the `connect` method.

   (c) Let us define the height of a quick union tree with a single node to be 0. What is the shortest and tallest height possible for a Quick Union object with 10 elements.

   (d) In general, what are the shortest and tallest heights possible for a Quick Union with k elements? What does this mean for the best and worst case runtimes for `isConnected` and `connect`?

   (e) What is the shortest and tallest height possible for a Weighted Quick Union with 10 elements? How about a Weighted Quick Union with N elements? What does this mean for the best and worst-case runtimes for `isConnected` and `connect`?

Solutions available in #7 and #8 of Guerilla Section 4