1 Disjoint-set data structure

1.1 Operations

Disjoint-set data structure enable us to efficiently perform operations such as placing elements into sets, querying whether two elements are in the same set, and merging two sets together. (Thus they’re very useful for simulating graph connectivity.) Must implement the following operations:

- `makeset(x)`: create a new set containing the single element `x`.
- `union(x, y)`: replace sets containing `x` and `y` by their union.
- `find(x)`: return name of set containing `x`.

We add for convenience the function `link(x, y)` where `x, y` are roots: `link` changes the parent pointer of one of the roots to be the other root. In particular, `union(x, y) = link(find(x), find(y))`, so the main problem is to make the `find` operations efficient.

1.2 Optimization Heuristics

We have two main methods of optimization for disjoint-set data structures:

- **Union by rank.** When performing a `union` operation, we prefer to merge the shallower tree into the deeper tree.
- **Path compression.** After performing a `find` operation, we can simply attach all the nodes touched directly onto the root of the tree.

Exercise 1 (1.0). When using neither union by rank nor path compression, what is the asymptotic runtime of `find(x)`? `union(x, y)`?

Solution

The worst possible case is when the elements get added into what resembles a linked list, for $O(n)$ cost of both finding and merging.

Exercise 2. Draw how the disjoint set data structure changes after each of the following operations using a particular heuristic:

(a) `union(x, y)` with union by rank.
(b) FIND\((x)\) with path compression.

![Diagram of a tree structure with nodes labeled a to g and an element x at the bottom.]

**Solution**

Exercise 3. When using the union by rank optimization *only*, what is the asymptotic runtime of the operation FIND\((x)\)?

**Solution**

When we use the union by rank heuristic, we have the property that the rank of a tree only increases when we union together two trees of the same rank. (Recall that rank = height). If you union a tree with rank 3 with one of rank 4, you get a tree of rank 4. Only when you union two trees of rank 4
can you get a tree of rank 5.

With this property, we can show that a tree of rank \( h \) has at least \( 2^h \) nodes. This can be proved using induction.

**Base Case:** When the rank of a tree is 0, there is always 1 node, which is indeed at least \( 2^0 \).

**Inductive Case:** Suppose it is true that every tree of rank \( h \) has at least \( 2^h \) nodes. If we had a tree of rank \( h + 1 \), we know that it must have come about through the union of two trees of rank \( h + 1 \) at some point in the history of this tree. By the inductive hypothesis, those two trees of rank \( h \) must have had \( \geq 2^h \) nodes each, and thus the number of nodes in this tree of rank \( h + 1 \) must be at least \( 2^h + 2^h = 2^{h+1} \).

Therefore, we have that the run-time of \texttt{find}(x) when using only the union by rank heuristic is \( O(\log n) \) because if there are \( n \) nodes, then the maximum rank of any tree is \( \log n \).

**Exercise 4.** Suppose that you are given an undirected friendship map for everyone at Harvard. (Vertices are students, and there is an edge between two students if they are friends.) Define a clique as a strongly connected component of this graph (note for those familiar with graph theory, this is not the typical definition of a clique on a graph).

(a) How could you (not using the union-find datastructure) determine the number of cliques?

(b) How could you solve this same problem using the union-find datastructure?

(c) How do the runtimes between the two approaches differ?

(d) What if now you are at Visitas and people are constantly making friends. You would like to know how many cliques there are at any given point in time with the knowledge that more edges are constantly being drawn on the graph. How do both approaches generalize?
What about if people are also allowed to have falling outs and unfriend each other? Is there any way to modify union-find to work in this situation as well.

Solution

(a) We can use BFS to find the number of connected components in the spirit of pset 2 question 2.

(b) We can make a set for every student, and then take unions for every edge that we see.

(c) Let $n$ be the number of students, and $m$ the number of friendships. BFS would take time $O(n + m)$ and union-find would take time $O(n + m\alpha(m))$ where $\alpha(m)$ is the inverse Ackermann function. These are essentially the same.

(d) We would have to re-run BFS after every edge that we add, adding a factor of $m$ to our runtime analysis. On the other hand, if we use union-find, we can keep on doing exactly what we were doing before, and thus the runtime stays the same.

(e) Both BFS as well as using union-find would have to re-compute after every edge deletion. To the extent that edge additions are more common, we may still prefer union-find, but there is no straightforward manipulation to be able to use our previously computed results incrementally when edges can be deleted.

2 Minimum Spanning Trees

2.1 Definition and Motivation

A spanning tree is a sub-graph with all the vertices of the initial graph that is also a tree (recall that a tree is characterized by being connected and acyclic). The minimum spanning tree of a (possible weighted) graph is a spanning tree whose sum of the weights on edges is minimal.

Exercise 5. True or false?

(a) All undirected graphs have a minimum spanning tree.

(b) All connected undirected graphs have a unique minimum spanning tree.

(c) All trees have a unique minimum spanning tree.
A minimum spanning tree is generally helpful when trying to connect $n$ objects of interest in an optimized way. For instance, if vertices represent cities and edges represent the cost of building a railroad between any two cities, then a MST gives the least possible cost of connecting all the cities. Analogous problems can be constructed for internet connectivity, electric grids, etc.

3 Exchange Property

In class, we saw the cut property, upon which the correctness of both Prim’s and Kruskal’s algorithms depend. We can also make an even stronger claim: the exchange property.

The exchange property states that if $T$ and $T'$ are spanning trees in $G(V,E)$, then given any $e' \in T' - T$, there exists an edge $e \in T - T'$ such that $(T - \{e\}) \cup \{e'\}$ is also a spanning tree.

Exercise 6. Prove the exchange property.

Solution

Adding $e'$ into $T$ results in a unique cycle. Some edge in this cycle must not be in $T'$ (else $T'$ wouldn’t be a tree). If we call this edge $e$, then deleting this edge restores a spanning tree (connected with $n - 1$ edges).

Exercise 7. Let $G$ be a graph. Let $G'$ be a directed graph whose vertices are spanning trees of $G$. Draw an edge from one vertex of $G'$ to another if the first vertex (spanning tree) can be transformed into the other by replacing one of its edges with another edge.

(a) Show that this is effectively an undirected graph.

(b) Is this graph connected?

(c) Suppose that $v$ is an MST. Prove that there exists a path in $G'$ from any $v'$ to $v$. 
Solution

(a) Swapping an edge in a spanning tree is a symmetric operation, so if there is an edge from $v$ to $v'$, there must exist an edge from $v'$ to $v$.

(b) That this graph is connected follows from the exchange property. Consider two spanning trees $v$ and $v'$. Suppose that $|T' - T| = n$. Then, by the exchange property we can find a $\hat{T}$ such that there is an edge between $T'$ and $\hat{T}$ and $|\hat{T} - T| = n - 1$. Repeating this logic results in a path between any two vertices in $G'$.

(c) This property follows immediately from the definition of connected. However, the result is worth highlighting as a special case of importance.