HW4: Union-Find and MST

Due: Thursday, November 17, at 11:59 PM, via Canvas

You may work on your own or with one (1) partner.

For this assignment you will implement the union-find data structure with path compression and weighted union (WQUPC) as we saw in class. Unlike in HW3, the representation itself is not defined for you, so you'll have to define it. Then you will use your union-find data structure to implement Kruskal's minimum spanning tree (MST) algorithm.

This assignment depends on your graph implementation from HW2, which you will have to copy over.

In `unionfind.rkt`[1] I've supplied headers for the functions that you'll need to write, along with suggested helpers and some code to help with testing.

Background

In this assignment you will use a union-find structure to solve a particular problem: finding the MST of a graph. In this section we offer background on what an MST is and one algorithm for computing it.

Definitions

A graph is *connected* if there is a path from every vertex to every other; otherwise it comprises two or more *connected components*, each of which is a maximal connected subgraph. (A connected component is maximal in the sense that no additional vertices could be added and still have it be connected.)

A *spanning tree* of a connected graph G is a subgraph that includes all of G's vertices, but only enough edges for it to be connected and no more. Cycles would introduce redundant connectivity, so it's a tree. Note that the number of edges in a spanning tree is always one fewer than the number of vertices in the original graph.

[1] `http://goo.gl/ZiHSNd`

A *minimum spanning tree* for a connected graph is a spanning tree with minimum total weight. (There may be a tie.) We can interpret an MST as follows: If vertices represent sites of some kind, edges potential connections between them, and weights the costs of those edges, then an MST gives the lowest cost way to connect all the sites.

A graph that isn't connected has a minimum spanning tree for each of its connected components. This collection of MSTs is a *minimum spanning forest.*

Kruskal's algorithm

The result of Kruskal's algorithm is a graph with the same vertices as the input graph, but whose edges form a minimum spanning tree (or forest). The result graph starts with all of the vertices from the input graph and no edges. In other words, initially each vertex forms its own (degenerate) connected component.

The algorithm works by maintaining the set of connected components in the result (using a union-find data structure); it repeatedly adds an edge that connects two components, thus unifying them into one. In particular, to achieve minimality, it considers the edges in order from lightest weight to heaviest. For each edge, if its two vertices are already in the same connected component of the result graph, the edge is ignored; but if the edge would connect vertices that are in two different connected components then the edge is added to the resulting graph, thus joining the two components into one. When all edges have been considered then the result is a minimum spanning tree (or forest, as appropriate).

Your task

Part I: Union-Find

First you will need to define your representation, the `UnionFind` data type. Each `UnionFind` represents a "universe" with a fixed number of objects identified by natural numbers.

Then you will have to implement five functions:

```
create     : N -> UnionFind                ; O(n)
size       : UnionFind -> N                ; O(1)
union!     : UnionFind N N -> Void         ; amortized O(log* n)
find       : UnionFind N -> N              ; amortized O(log* n)
same-set?  : UnionFind N N -> Boolean      ; amortized O(log* n)
```

(Note: `N` *is the natural numbers and* $\log^*$ *is the iterated logarithm.)*

Calling `(create n)` returns a new `UnionFind` universe (defined by you) initialized to have `n` objects in disjoint singleton sets numbered 0 to `n` − 1. Given a universe `uf`, `(size uf)` returns the number of objects (not sets!) in the universe—that is, `size` will always return the number that was passed to `create` when that universe was initialized.

Functions `union!` and `find` implement the standard union-find operations: The function call `(union! uf n m)` unions the set containing `n` with the set containing `m`, if they are not already one and the same. `(find uf n)` returns the representative (root) object name for the set containing `n`. The `find` function must perform path compression, and because the `union!` function calls `find`, it (indirectly) performs path compression as well. The `union` function must set the parent of the root of the smaller set to be the root of the larger set. For convenience, `(same-set? uf n m)` returns whether objects `n` and `m` are in the same set according to union-find universe `uf`.

Part II: Kruskal's MST algorithm

Once you have a working union-find, you should implement Kruskal's algorithm as a function `kruskal-mst : WUGraph -> WUGraph`. Given any weighted, undirected graph `g`, `(kruskal-mst g)` returns a graph with the same vertices as `g` and edges forming a minimum spanning forest, using the algorithm as described above.

In order to represent the graph whose MST your are computing, you should use your graph implementation from HW2. There's no good way to import it, so you will have to copy and paste. (If you're working with a different partner now than you did for HW2 then you may use either your own `WUGraph` or theirs.)

In order to consider the edges in order by increasing weight, Kruskal's algorithm requires sorting the edges by weight. I used my HW3 solution to write

a heap sort (which works by adding all the things to sort to a heap and then removing them), but you may use any sorting algorithm you wish.

I've listed some helpers that you may find useful at the bottom of `unionfind.rkt`.

Deliverables

The provided file `unionfind.rkt`, containing

- a definition of your `UnionFind` data type,
- complete, working definitions of the five union-find operations specified above, and
- a working implementation of Kruskal's algorithm.

Thorough testing is strongly recommended but will not be graded.