HW4: Graphs and MSTs

Due: Monday, December 7, at 11:59 PM, via Canvas

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

For this assignment you will implement an API for weighted, undirected graphs; then you will use this API, along with your solutions to previous homework assignments, to implement Kruskal's algorithm for computing a minimum spanning forest (detailed below).

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

Background

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 less than the number of vertices in the original graph.
- 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 miniumum 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: Graphs

First you will need to define your representation, the `WUGraph` data type. A `WUGraph` represents a weighted, unordered graph, where vertices are identified by consecutive natural numbers from 0, and weights are arbitrary numbers:

```
;; Vertex is N
;; Weight is Number
```

The API also uses a data type for weights that includes infinity:

```
;; A MaybeWeight is one of:
;; -- Weight
;; -- +inf.0
```

Your graph representation is up to you—you may use either adjacency lists or an adjacency matrix.

Once you've defined your graph representation, you will have to implement five functions for working with graphs:

```
make-graph   : N -> WUGraph
set-edge!    : WUGraph Vertex Vertex MaybeWeight -> Void
graph-size   : WUGraph -> N
get-edge     : WUGraph Vertex Vertex -> MaybeWeight
get-adjacent : WUGraph Vertex -> [List-of Vertex]
```

To construct a graph, we would start with `(make-graph n)`, which returns a new graph having `n` vertices and no edges. Then we add edges using `(set-edge! g u v w)`, which connects vertices `u` and `v` by an edge having weight `w`. The weight `w` may be an actual numeric weight or it may be `+inf.0`, which effectively removes the edge.

`(graph-size g)` returns the number of vertices in `g`, which is the same as the number originally passed to `make-graph` to create the graph. `(get-edge g u v)` returns the weight of the edge from `u` to `v`, which will be `+inf.0` if there is no such edge. Note that because the graph is undirected, `(get-edge g u v)` should always be the same as `(get-edge g v u)`.[1] Finally

[1] This probably means that `set-edge!` needs to maintain an invariant.

`(get-adjacent g v)` returns a list of all the vertices adjacent to `v` in graph `g`—note that an undirected graph does not distinguish predecessors from successors.

Part II: MSTs

Once you have a working graph API, 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 maintain and query the set of connected components, Kruskal's algorithm uses union-find. You should use your union-find data structure and operations from HW3. 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 HW3 then you may use either your own union-find 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 HW2 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 `mst.rkt`.

Deliverable

The provided file `mst.rkt` (`http://goo.gl/q3QfuL`), containing

1. a definition of your `WUGraph` data type,
2. complete, working definitions of the five graph API functions specified above, and
3. a working implementation of `kruskal-mst`.

Thorough testing is strongly recommended but will not be graded.