HW6: Union-Find and MST

Due: Wednesday, March 13, at 11:59 PM, via Canvas

You must work on your own for this assigment.

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 HW5, 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.

In `unionfind.rkt` I've supplied headers for the methods and function that you'll need to write, along with some code to help with testing.

This assignment depends on your graph implementation from HW4 and your binary heap implementation from HW5. You should place these in the same directory as `unionfind.rkt`, since it imports them. You may upload a zip file to Canvas with all three files, or upload your `unionfind.rkt` only and we will use our own versions of the other two when grading.

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.

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).

The union-find data structure

A union-find tracks a partition of some fixed number of objects, which are called its *universe*. A union-find is typically represented as either two vectors or a vector of pairs. In either case, it maps each object in the universe to two values: its *id* and its *weight*.

The ids of all the objects form a parent-oriented tree for each disjoint set, in the sense that every object's id is its parent object in some tree, except for the root of each tree, which is its own id. The weight of a root object gives the number of objects in that tree; the weights of non-root objects do not carry any useful information.

To start out, a union-find is initialized so that every object is its own id (and thus a root of a singleton tree) and every object has a weight of 1. Then the main operations are as follows:

- The *find* operation returns the representative of the given object's set, which is the root of its tree. We can find the root by repeatedly following each object to its id until we reach the root, which is its own id. Along the way back, *find* must perform path compression, in the sense that the id of every object it traverses must be updated to point directly to the root. (Alternatively, you can set the id of every object along the path to its grandparent object.)
- The *union* operation first finds the roots of the two given objects. (It must use your *find* operation to do this, to ensure path compression.) Then, it unions the two sets by making one root the id of the other. To maintain balance, it chooses the new root based on the weights of the two roots to be joined. If they are the same, then it doesn't matter, but if they are different, then the heavier root stays a root, and the lighter root has its id updated to point to the heavier one. Furthermore, the weight of the remaining root must be updated to include the weight of the lighter root that is now attached below it.

Your task

Part I: Union-Find

Your job is to complete the implementation of the `UnionFind` class. In particular:

1. Define the necessary field(s) in the `UnionFind` class.
2. Define the constructor (`__init__`) to initialize your fields.
3. Define the `len` method to return the number of objects.

4. Define the `find` method to return the representative of the given object's set.
5. Define the `union` method to union the two given objects' sets.

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

Methods `find` and `union` implement the standard union-find operations: The method call `uf.union(n, m)` unions the set containing `n` with the set containing `m`, if they are not already one and the same. `uf.find(n)` returns the representative (root) object name for the set containing `n`. The `find` method must perform path compression, and because the `union` method calls `find`, it (indirectly) performs path compression as well. The `union` method must set the parent of the root of the smaller set to be the root of the larger set, and must update the weight of the larger set.

For convenience, `uf.same_set?(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 must 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 consider the edges in order by increasing weight, Kruskal's algorithm requires sorting the edges by weight. I've provided a helper function `get_all_edges_increasing`, which takes a `WuGraph` and returns a vector of its edges in order of increasing weight. Your `kruskal_mst` function should use this function to get the vector of edges to iterate over.

Deliverables

The provided file `unionfind.rkt`, containing

- a complete definition of the `UnionFind` class with its fields, constructor, and methods,
- a working implementation of Kruskal's algorithm, and
- sufficient tests to cover all cases and be confident of your code's correctness.