Assignment 5

This fifth and last assignment focuses on the two algorithms for finding a minimum spanning tree for a graph. You will be implementing Prim's and Kruskal's algorithms.

Updating your repository

Most of the hard work was done in the last lab. All you need to do is fetch and merge the changes I made to the upstream repository.

Before we start anything, make sure that you have NO uncommitted changes. You must either commit what you have, if it is ready to commit, or stash the changes, allowing you to retrieve them later.

1. First make sure that your local master is the active branch. It should have a little checkbox next to it.
2. We should make sure to bring in the newest version of the upstream remote. Right-click on the "upstream" remote in the "Remotes" section on the left, and choose "fetch upstream".
3. Now right-click on the upstream/master branch within the "Remotes" section to bring up its context menu. One of the options will say "Merge upstream/master onto master". Click it.
4. You should now be ready to go. Whenever you push your changes to your origin, these commits from the upstream will go along with them. If you like, you can go ahead and push now.

The assignment

Introductory stuff

In this assignment there are two files you will be looking at: `Spanning.java`, inside `src/main/java/utils` is the main file that you will have to edit. You may also need to look into the `WeightedGraph.java` file. It contains an implementation for a weighted graph. This is similar to the simple graph you have used before. You will still want to take a look at the functions provided in that file.

As in the previous assignment, you will find numerous TODO comments throughout the file. They mark the locations where you need to add something to the implementations. Some times it will be one or two lines, some times it may be more. If you use Eclipse, you can see blue rectangles on the right side of the code indicating the location of all TODOs, and you can click on them to go there. You can keep track of your progress that way, so make sure to remove the TODO from the methods you have already implemented.

A standard printout of a run of the various methods can be obtained by running the main method of the Spanning class (so running the Spanning class as an application). To do that, you will need to first build using the corresponding gradle task, and then run the runSpanning gradle task from the run group of tasks (you may need to refresh the gradle project and possibly close and reopen the gradle tasks to make it appear).

Alternatively, you can go to your terminal (or Eclipse's console), go to the main project folder and run (you may need to run the build gradle task before this works):

```
java -cp build/libs/algorithms-assignments.jar utils.Spanning
```

If you like to write more tests, you can create test files under /src/test/java. Ask for help if you are not sure how to start those tests. You can use the :test Gradle task to run the tests.

You should make a new *commit* in GitKraken after completing each self-contained part, typically after each method you complete. Do not try to commit everything at once.

When you are done with the assignment, in order to "submit" it, simply **push** your commits via GitKraken. These will now be posted on your account in GitLab and I will be able to follow them from there.

The structure of the files

There are two file of concern in this assignment. Spanning.java and WeightedGraph.java.

WeightedGraph.java contains a simple implementation of a weighted graph. You should look at the file to make sure you are familiar with how to add edges to your trees. In particular you will want to familiarize yourself with the following methods in WeightedGraph:

- getEdge and hasEdge.
- addEdge (two forms).
- order.
- outgoingEdges.

There is also an Edge class within WeightedGraph that also has some useful methods:

- First off, the two vertices of an edge are stored in the variables v1 and v2 respectively, and they are always ordered in numerical order. So the vertex connecting 2 and 3 always have v1=2 and v2=3 regardless of whether you look at it as going from 3 to 2 or from 2 to 3 when you traverse it.
- isAdjacentTo returns whether this edge has the provided vertex as one of its endpoints.

- getOtherVertex returns the other vertex in the edge than the one provided. For instance if the edge is connecting 2 to 3, then getOtherVertex(2)=3 and getOtherVertex(3)=2.

The Spanning class is the class where you will implement the two different algorithms. There are two inner classes within Spanning, namely Prim and Kruskal.

- Both are set up so that when the constructor is called, it automatically also runs the algorithm and stores the result in a public tree variable.
- Both use priority queues to hold a list of edges for processing. Note that priority queues in Java are "minimum" priority queues, so the value at the head of the queue is the one with the smallest weight, as we want.
- Prim implements Prim's algorithm.
  - It uses a queue priority queue to keep track of the edges that are candidates for the next step.
  - While the algorithm as described in books is meant to store the candidate next vertices in the queue, and to update them if a shorter edge to them is found, our algorithm instead stores the edges and does not need to worry about doing any updating. But the queue may end up containing two edges both pointing to the same vertex, and only the shortest of those edges will get used.
  - What this means is that when you get the next edge out of the queue, you must first check that this edge does actually lead to a new vertex, and if it doesn't then simply move on to the next edge.
  - For this reason there is also a boolean array visited that keeps track of whether the vertices have been visited already or not (i.e. whether they have become part of the tree).
- Kruskal implements Kruskal's algorithm.
  - It uses a unionFind array to hold the union-find structure and update it accordingly.
  - It also uses a priority queue to hold the edges by their weight.

The actual tasks

Now on to the tasks you need to complete. Make sure you have read the previous section for an understanding of the structure of the classes you will be working with.

1. You should start with Prim's algorithm. The constructor is provided for you, and a general skeleton as well. The methods you will need to fill in are as follows:

- processVertex is a method that "processes" a vertex. What this means is that it marks that vertex as visited, and then loops through the outgoing edges from this vertex.For each of those edges, it looks at what the "other" vertex is (a method in Edge will help with that), and if that other vertex is not visited yet then it adds that edge to the queue.
- findUnvisitedVertex is a method that given an edge where one of the vertices is visited and the other is not, returns the one that is not. This is being used when an edge is looked at, to determine which vertex is next.
- Finally, run is the main method. It starts by processing the vertex 1, then loops through the queue until the queue is empty. At each iteration, it looks at the next edge in the queue. If both vertices on that edge are visited, then it simply skips that edge. If not, then one of the two vertices is visited and the other is not. So we must add this edge to our result tree, and we must find and process the vertex on that edge that is not yet visited.

2. Next up is Kruskal's algorithm. The constructor and main skeleton is again provided for you. The methods you will need to fill in are as follows:

   - initializeUnionFind sets up initial values for the unionFind array. You just need to go through and make each vertex "point" to itself.
   - findRoot finds the "root" of the component in the union-find structure that a given vertex corresponds to. For this you will need to start with the vertex and traverse the unionFind pointers until you find a vertex that points to itself. This is the root, and you must make the initial vertex point directly to the root before returning.
   - areConnected is a function provided for you. It uses findRoot to determine if two vertices belong to the same unionFind component.
   - union joins the two components that its argument vertices belong to, by locating the roots for both vertices using findRoot and then making one root point to the other (you will need to make a choice, don't change both). Your answer will look a lot like areConnected.
   - addAllEdgesToQueue is another initialization method, that adds all the graph's edges into a queue. You simply need to loop over all vertices, and for each vertex loop over the outgoing edges, and insert those edges into the queue. BUT you must make sure not to add edges twice, as your loop will visit every edge twice. A simple way to ensure that is to only add the edge if its v1 vertex matches the vertex that you are currently looping over.
   - Finally, run puts it all together. It starts by initializing the union find structure and the edge list (these steps are part of the provided skeleton), and then there is a loop over the queue structure as long as that structure is nonempty. You will need to process the next edge in the queue, and see if the vertices in it are already connected in the unionFind structure. If they are not, then you must add this edge to your tree and also "union" the vertices in the unionFind.

And we're done!