The Objective

Our goal today is to implement and contrast two strategies for constructing minimum spanning trees for connected, undirected graphs with edge weights.

A spanning tree for a connected, undirected graph $G$ is a graph with the same vertices as $G$ and a subset of the edges of $G$. The subset must be chosen so that the spanning tree is still connected (in the sense that there is a path using the chosen edges from any vertex to any other vertex) but is now acyclic (in the sense that there are no paths of positive length that begin and end at the same vertex). A minimum spanning tree is a spanning tree in which the sum of the edge weights is less than or equal to the sum of the edge weights of any spanning tree for the same graph.

Setup

In Eclipse, open a new project, but do not request a module-info.java file or create a named package within that project. We'll work today in the unnamed package for that project. Create a new class file, MinimumSpanningTrees.java, in the unnamed package. We'll implement the two strategies as separate, static methods within this class.

I've created a cut-down version of Weiss's DisjointSets class (lacking the main procedure, which he used for testing, and the initial package declaration). It’s available on MathLAN as /home/reseda/object-oriented-programming/resources/DisjointSets.java

Copy this file into your unnamed package. Similarly, you should bring in a copy of Weiss's Graph.java file from

/home/reseda/object-oriented-programming/resources/textbook-code/Graph.java

You’ll also need the java.util.PriorityQueue class and perhaps other classes from the Java standard library. You can use import directives to bring them in.

Kruskal’s Algorithm

Section 24.2.2 of our textbook describes Kruskal’s algorithm for constructing minimum spanning trees. The idea is to put all of the edges of the graph $G$ into a priority queue, with the smallest edge weights having the highest priority and to set up a disjoint-set structure for all of the vertices of $G$, with each vertex initially constituting a disjoint set of its own. Initially, the result of the procedure (which will eventually become a minimum spanning tree) will be a newly created graph to which the algorithm adds vertices and edges as it proceeds.

In the main loop of Kruskal’s algorithm, extract one edge at a time from the priority queue and determine whether or not its endpoints belong to different equivalence classes. If so, add the edge and its endpoints to the emerging minimum spanning tree and run the union method on the endpoints. A minimum spanning tree always contains exactly $|V| - 1$ edges, where $V$ is the number of vertices, so you can exit from the loop as soon as this condition is met, or equivalently when all of the vertices have been combined into a single equivalence class within the disjoint-set structure. If the priority queue becomes empty before this exit condition is reached, the graph $G$ was not connected to begin with. Your code may throw an exception if that happens.

1. Write a suitable header for the kruskal method within the MinimumSpanningTree class. It should take a Graph as argument and return a minimum spanning tree for that graph.
2. At the beginning of the body of the `kruskal` method, write the statements that initialize the data structures appropriately: a priority queue containing the edges of the graph (prioritized by edge weight), a disjoint-set structure containing the vertices of the graph, and a graph to hold the result. (You may find that you also need a map to correlate vertices of the graph with array indices in the disjoint-set structure.)

3. Write the main loop of the `kruskal` procedure.

4. Write a `main` method for the `MinimumSpanningTree` class in which you build a copy of the undirected, weighted graph shown in Figure 24.6(a) on page 899 of the textbook, use `kruskal` to compute a minimum spanning tree for it, and display the edges of the result.

**Prim’s Algorithm**

As mentioned at the end of Monday’s class, there is another widely used algorithm for building minimum spanning trees. In this approach, the idea is to start a minimum spanning tree with any single vertex of the graph $G$ and then to connect the other vertices, one by one, by repeatedly finding the least-weight edge that connects a vertex that is already in the emerging minimum spanning tree with one that is not yet in the tree.

To make it easier to find this least-weight edge, we can keep a priority queue of edges, once again prioritized by edge weight, so that the edges of least weight will emerge from the priority queue first. This time, however, instead of putting all of the edges in immediately, as part of the setup of the priority queue, we will add edges to the priority queue only as we encounter them on the adjacency lists of vertices that are in the emerging minimum spanning tree.

As a result, it will be an invariant of the priority queue that all of the edges that are currently in the queue will have at least one endpoint in the emerging minimum spanning tree. Whenever we extract an edge from the priority queue, then, we’ll need to check whether both endpoints are already in the tree. If so, discard that edge and select another; if only one of the endpoints is already in the tree, however, add the edge to the emerging minimum spanning tree and add the edges in the adjacency list of the vertex that was not previously in the tree to the priority queue.

Once again, we can stop whenever the emerging minimum spanning tree contains all the vertices of $G$, which will happen when we have added $|V| - 1$ edges to it. Again, if the priority queue becomes empty before this exit condition is met, then $G$ was not connected and the implementation can throw an exception.

5. Write the header for the `prim` method. It should specify the same argument and return types as `kruskal`.

6. At the beginning of the body of the `prim` method, write the statements that initialize the data structures appropriately: a graph to hold the result, initially containing a single arbitrarily chosen vertex $v$ of $G$, and a priority queue, initially containing the edges in $v$’s adjacency list.

7. Write the main loop of the `prim` procedure.

8. In the `main` method, use `prim` to compute a minimum spanning tree for the graph that you already constructed in an earlier exercise and display the edges of the result.

**Optional Exercise: Comparisons**

9. How would you measure and compare the number of operations required for the completion of each of these methods, as a function of the number of vertices and/or the number of edges of the graph? How could one add “instrumentation” to the `kruskal` and `prim` methods to collect relevant information during testing?