Project 5: Computing a Minimal Spanning Tree using Kruskal's and Prim's Algorithms.

Due: 05/12/2021

Overview

For this project, you will implement **Prim's and Kruskal's algorithms** for computing the minimum spanning tree of a graph. Suppose a new railway company would like to have train stations in all major US cities, and their transport network should "*connect*" all cities. To get *some initial bounds on the cost* of the routes, the company could find a minimum spanning tree of the graph where the nodes are cities and the edges are railroads. The *cost* of each edge would be equal to the cost of running trains between the two cities (when the company computes this cost, they should take many factors into account including distance, fees for using the existing railroads owned by other companies, or cost of building a new railroad, how popular this route would be etc.). In this project, the costs are proportional to the distance between cities.

This project has a graphical user interface (GUI) that will display the graph of cities on the "map"[1] (the "map" is simply an image), and the minimal spanning tree:

Figure 1: The "map" of the USA shows "nodes" (major cities) and edges connecting them, as well as the edges in the minimal spanning tree (shown in blue).

The input file USA.txt has been provided in the **input** folder, but your program should be general and take the input txt file as a **command line argument** to MSTDriver (see package `algo`). Read the graph data from the file and store it in the Graph object.

---

[1] The image of the USA map and the input file are courtesy of Prof. Julie Zelenski.

The GUI for this project has been provided to you in the GUIApp class, you do not need to modify it. Once the user clicks the Prim's button in the GUI window, the program will call computeMST method in the PrimAlgorithm and display the edges of the MST tree in blue. When the user clicks the Kruskal's, button, the GUIApp would call computeMST() method in the KruskalAlgorithm class and show the corresponding MST. Reset button will clear the MST.

You are required to use the provided starter code.

Input File Format

Your program should take the input txt file as a **command line argument to MSTDriver**. You are given a file called "USA.txt" that contains the graph data in the format below. The keyword NODES is on the first line, followed by the number of the nodes on the second line, followed by the lines describing each node (one node per line). Each node is defined by the name of the city, and the x and y coordinates of the city *in the image*. Please note that the x and y coordinates are given so that we can display the nodes on the map; they are **not** used in Prim's and Kruskal's algorithms.
The arcs (edges) are listed after the nodes, starting with the ARCS keyword. Each edge has the origin and the destination city, as well as the cost ("weight") associated with this route between the two cities.

```
NODES
<num nodes>
<name of the city> <x-coordinate> <y-coordinate>
…
ARCS
<name of the city> <name of the city> <cost>
…
```

Note that if two cities city1 and city2 are connected via an edge, this edge will show up in the input text file only **once**. When you create an adjacency list of this graph, make sure to *add the edge going in the opposite direction as well.*

Graph Representation

See classes in the package called `graph`. Class `Graph` should store:
- An array of nodes `CityNode[] nodes;`
The index of each node in this array is the node's id. The nodes are added to the array (and hence are assigned id-s) in the order in which you read them from the text file. Each node (`CityNode`) in your graph should store: the name of the city and the location of the node on the map.

- An adjacency list `Edge[] adjacencyList`
The adjacency list stores a linked list of outgoing edges for each vertex (this is array we earlier called "graph" in code examples). Each `Edge` stores the id of the source vertex, the id of the destination version, the cost of the edge and the reference to the next Edge in the linked list for the given source vertex. Note that you are **not** allowed to use class LinkedList from the Collections framework in this project.

- The total number of edges (`numEdges`) in the graph.

- A HashMap that maps each city name to the corresponding node id.
Insert keys and values into this HashMap when you read the nodes from the file, and use the HashMap to find the node id given the name of the city, when you read the edges from the text file.

In order for the provided `GUIApp` class to be able to display nodes and edges of the graph, class Graph has the following methods (do not modify them):

- `public Point[] getNodes()` - Returns an array of locations of each node.
- `public String[] getCities()` - Returns an array of city names
- `public Point[][] getEdges()` - Returns edges as a 2D array of points. For each edge, we store an array of two Points, v1 and v2: v1 is the source vertex for this edge, v2 is the destination vertex.

These methods have been provided to you, but will work only after you fill in code in the constructor of class Graph and fill out the `nodes` array and the `adjacency list`.

Prim's Algorithm

See classes in the package "algo". Class MSTAlgorithm is an abstract class that is the parent of classes PrimAlgorithm and KruskalAlgorithm.

The constructor of this class takes the graph and the source vertex as parameters.

The computeMST method should start with the source vertex and build the MST (minimal spanning tree) using Prim's algorithm. You need to create a table, where for each node id you would store:

- The current estimate of the **cost** of adding this vertex to the MST tree
- The "parent" node for each node (the previous node on the path).

Given the source vertex, Prim's algorithm proceeds as following:

```
Initialize the table
Repeat numVertices times:
   v  = findMinimumUknownVertex()
   Mark v as known
   for each neighbor u of v:
      if (u is unknown)
        if table[u].cost > cost of edge from v to u {
                 table[u].cost  = cost of edge from v to u
                  table[u].path = v
         }
```

Prim's algorithm needs to repeatedly find an "unknown" node with the smallest cost. A **PriorityQueue** (using MinHeap) is the most efficient way to implement this, and using it is required for this project (you are required to implement a priority queue using a min heap,

from scratch).

While the priority queue is not empty, you would remove the vertex with the smallest "cost", and update the costs of its neighbors – **also updating the costs in the priority queue** (you would need to write a reduceKey() method in the MinHeap class).

Once the user clicks the Prim's button in the GUI window, the GUI would call computeMST() method in Prim's and display the edges of the MST in blue.

Priority Queue

You need to implement **your own Priority Queue** for this project (you are **not** allowed to use a built in Priority Queue). The priority queue needs to be implemented using a MinHeap. Please fill in code in class `MinHeap` in the package called "priorityQueue". The `MinHeap` class should have the following methods:

- `void insert(int nodeId, int priority)` - Inserts node id with the given priority into the priority queue. **Priority** is the cost of the edge.
- `int removeMin()` - Removes the vertex with the smallest priority from the queue, and returns it.
- `void reduceKey(int nodeId, int newPriority)` Reduces the priority of the given vertex in the priority queue to newPriority, rearranging the queue (minHeap) as necessary.
  To implement this method efficiently, you will need to keep track of where each element is in the priority queue (min heap). The simplest way to do it is to store an **array of positions in the priority queue for each node id (see below).**

So in your `MinHeap` class, you need to store:
- the actual **heap** array, where each element has a node id and a cost.
- array of **positions** (where an index is a node id, and the value is its index in the heap array)
- current size of the heap.

Disjoint Sets and Kruskal's Algorithm

Kruskal's algorithm computes MST using the Disjoint Sets data structure. The constructor of this class takes only the graph as a parameter. Before you implement Kruskal's, understand the code in class DisjointSets in the *sets* package. You do not need to modify the DisjointSets class.

Once you finished and tested Disjoint Sets, you can implement Kruskal's. Kruskal's alogrithm proceeds as follows (this code will be in method `computeMST`):

Create n sets (where n is the number of vertices) by placing each vertex in its own set
Sort edges based on the cost (in increasing order)
For each edge e = (v1, v2) in the list
    Compute the sets that v1 and v2 belong to.
    If v1 and v2 belong to the same set:
        *Do **not** add this edge to MST.*
    Otherwise:
        Add e to the minimum spanning tree

Merge sets that v1 and v2 belong to

You can use the *find* method in DisjointSets class to check which set a particular vertex belongs to. You can use the *union* method in DisjointSets class to merge the two sets. You may use ArrayList and Collections.sort to sort the edges, but make sure class Edge implements `Comparable` and edges are compared based on the cost.

Submission

Submit your solution to your github repository by the deadline. You are required to have a minimum of 10 meaningful commits made over the course of at least 4 days.

Please note that your code should follow Java style guidelines (see StyleGuidelines.pdf posted for project 1). The instructor may invite students for a code review for this project.

Testing

You are responsible for testing your project. Hopefully, GUI makes testing a bit easier for this project. Test helper classes like Graph and MinHeap separately before combining them into the final project.