Project 3: GPS

Maryfrances Umeora	Ethan Yang
mumeora@u.rochester.edu	eyang13@u.rochester.edu
TA Name: Linan Li	TA Name: Bartlomiej Jezierski

Intro

When given longitudes and latitudes of intersections and roads, we are able to construct a visual map and use Dijkstra's algorithm to find the shortest path from one intersection to another.
Note to Grading TA: If you don't read anything in this pdf, at least scroll to the very bottom for an explanation of what we did for extra credit. Thank you. Work was done on a need to do basis. No major distribution or rime or reason was used.

How to Run

There's only one java file that contains all of our code, called `StreetMap.java`.
When on command prompt, you can use any variation of

```
java StreetMap map.txt [--show ] [--directions startIntersection endIntersection]
```

Important: Don't forget to include the "--" that is before show and directions.

Flow of Events

1) First, the program reads the commands. It reads each part of the command by turning the line into a String array. It determines if the array contains the String "--show" or "--directions" and appropriately sets the booleans *show* and *directions* to true.
2) We then create a Scanner of the text file specified and reads line by line. If the first character is "i", we create a vertex with the name, longitude, and latitude. At the same time, we keep track of the minimum and maximum longitude and latitude to use scaling the map. If the first character is "r", we create an edge with the name, vertices, the distance which is determined by the haversine() method, and adding the edges to the vertices LinkedList of edges. Continue until we have read the entirety of the file.
3) After this, if the user is asking for directions, we perform Dijkstra's algorithm. First, we get the vertex from the HashMap, set its distance to 0, and enqueue it into the Priority Queue. From here, we just perform Dijkstra's algorithm. Our boolean *foundpath* keeps track of if the algorithm actually found a path, if not, we inform the user and set the boolean *directions* back to false to ensure that when we draw the map we do not try to draw the route. If we did find a path, then we create a String to represent what we will print for the path. We start at the end destination and go backwards to the source. As long

as *tempV* is not source, we add the name of *tempV* to the String, and at the end, we print the path on console.

4) To display the map, we have the canvas that only shows if the boolean *show* is true. In the paintComponent() method we have a for loop that goes through each vertex in the HashMap. For each vertex, we calculate the scaled x and y coordinates using an equation that we had to figure out. $x = (Longitude - minimum\ Longitude)/range\ of\ Longitudes * Width$ and $y = Height - (Latitude - minimum\ Latitude)/range\ of\ latitude * Height$. Of course, we set the vertex's x and y coordinates for easier access later. Next, we go through the edges that come from this vertex, and get the corresponding vertex at the end of the edge. We then use the same equation, and draw a line between these two vertices using the calculated x and y. After we draw all of the edges in the map, we change the color to red so if we draw the route, it'll be easily distinguishable. The if statement checks if the boolean *directions* is still true, and if it is, then we go backwards from the target vertex, similar to how we printed the path before. While tempV is not the start of the path, we draw a line using the coordinates of tempV and the coordinates of tempV.prev, the previous vertex on this route. Once tempV is the source vertex, then the drawing is complete.
5) Once the map is drawn, there is additional functionality. There are 2 dropBoxes and one button at the bottom of the panel. These allow the user to change the route calculated. The itemStateChange() keeps track of the String chosen in the dropBoxes and sets the source or target vertex based on their new value. The actionPerformed() method activated when the button is pressed, and reinitialize the vertices and recalculate the route from these new source and target. Calls the findRoute() and repaint() to get the new path and redraw the map.

The Code

We use a total of 4 classes included in the main StreetMap class.

Vertex Class

Lines 39-56

This class creates vertex objects that have the properties name, previous vertex, X & Y coordinates, Longitude and Latitude, a list of edges, distance, and boolean *visited*. Distance is initialized as Double.MAX_VALUE to represent infinity.

Edge Class

Lines 61-73

This class creates edges that have a name, two vertices that it connects, and a weight value.

VertexComparator Class

Lines 77-83

This class implements Comparator to allow us to compare two vertices based on their distance values.

Canvas Class

Lines 88-175

This class is where the actual drawing occurs. The paintComponent() method goes through all of the vertices stored in the HashMap (graph) and calculates their x and y coordinates using the formula, saving the x and y coordinates as the vertex's data. We then go through each of the edges from this vertex, and draw a line from this vertex to the other end vertex of the edge. After drawing all of these edges, we check if *directions* == true. If it is true, then we move backwards from the target vertex. We get the x and y coordinates of the current *tempV* and the vertex that is previous to it. Then draw a line using these x and y. Set t*empV = prevV* and continue this cycle until *tempV == source vertex*. This draws the route between the desired destinations.

The class implements MouseMotionListener in order to be able to do what we did for extra credit, which you can find the details to below.

Distance() and haversine() Methods

Lines 181-207

These methods are used to calculate distance between two vertices based on their longitudes and latitudes. Takes in 2 latitudes and longitudes to calculate the distance between them. Returns a double as distance. As per the teacher's permission, we borrowed the basis of the code from this link:
https://github.com/jasonwinn/haversine/blob/master/Haversine.java?fbclid=IwAR1aVZ2zUNFp2Up5E2Rq7gyyV3J4YyY8x_FTMaoDuBAwNdPNFjFCEY4xy0g.

findRoute() Method

Lines 213-299

This method is used to calculate the path from the source vertex to the end. This basically is Dijkstra algorithm. No parameters.

itemStateChanged() Method

Lines 304-315

This method is used to keep track of if the user changes the menu in the dropBoxes in the panel. If there is a change, then we set the source or target to the appropriate vertex. The leftmost box changes the source while the right one changes the target. No parameters.

actionPerformed() Method

Lines 320-331

This method is used to keep track of if the user presses the button. If the button is pressed, we reset all of the vertices' distance back to Double.MAX_VALUE, previous vertex back to null, and lastly, their visited boolean back to false. Then, we recall findRoute() and repaint() on the canvas to redraw the map with the new route. No parameters.

The Main Method

Lines 337-442

The main method is mainly utility based.
Lines 342-352 reads the commands and determines if the user is asking to show the map, and if they want to find directions.
Lines 355-402 reads the input text file. It creates a vertex or an edge with the appropriate data.
Lines 405-408 happens if the user wants to find a route. It calls the findRoute() method.
Lines 411-440 is where the map is drawn. We also create JComboBoxes and a JButton that allows the user to create a new route after the canvas is created.

Expected Runtimes

The program should not take longer than a minute to complete.
Drawing the Map - O(|V|+|E|)
Finding Shortest Path - O(|V|*log|V|)

Extra Credit

1) Rendering Hints

You might notice that drawing with Java Graphics tends to create a rather pixelated image. Thus, before drawing, we set the Rendering Hints of the graphics object in order to create a smoother image.

2) Interactive Map Part.1

You may notice that if you hover your mouse over the map we created, you will be spammed in the console with information of where you currently are. We used MouseMotionListener to get the intersections within 5 pixels away from the location of the mouse and report that to you.

2) Interactive Map Part.2

At the bottom of the canvas is also 2 DropBoxes and 1 button. These allow you to change the route. The leftmost dropBox changes the starting point, and the rightmost changes the destination. The button is what activates the new route.

3) Detailed README file

This speaks for itself.