Joseph Ville
CSCI 351-01
4/14/16

Report – Programming Project 3

1. Each of my two analysis programs take in, at the command line, the name of a file in Graph File Format. The graph file represents a map of the ancient Egyptian empire, with the vertices being the cities, and the edges being the roads.

   `Cities` constructs a graph from the input data, which is simply a list of coordinates for the vertices of the graph. It then computes the minimum spanning tree of the graph, and outputs to the console the total distance of the minimum spanning tree. It also writes a new file containing the same list of coordinates from the input, as well as a list of the edges in the graph.

   `BetweennessCentrality` also constructs a graph from the input and finds the minimum spanning tree. Next, it computes the betweenness centrality of each vertex in the minimum spanning tree. The program then prints the rank, vertex number, and betweenness centrality of the top-40- ranked vertices with respect to betweenness centrality. The program also prints these same values of the empire's capital city, the city at vertex 0.

2. `Cities` uses the Parallel Java 2 library, so the classpath must be set accordingly before the code can be compiled. Once it is compiled, the following two commands will run their respective programs:

   `java Cities <fileName>`
   where `<fileName>` is the name of a file in Graph File Format.

   `java BetweennessCentrality <fileName>`
   where `<fileName>` is the name of a file in Graph File Format.

3.
```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.math.BigDecimal;
import java.math.MathContext;
import java.text.DecimalFormat;

import edu.rit.util.PriorityQueue;

/**
 * Process a graph file with the locations of cities and construct a
 * minimum spanning tree from that graph
 * @author Joseph Ville
 *
 */
```

```
public class Cities
{
	private static int V; // number of vertices
	private static int E; // number of edges
	private static Vertex[] vertices; // array of vertices
	private static double totalDistance; // total distance of the MST
	private static boolean eLine; // whether there's an edge line in the graph file

	/**
	 * Default constructor
	 */
	public Cities()
	{
	}

	/**
	 * Main method for this program
	 * @param args
	 */
	public static void main(String[] args)
	{
		if(args.length < 1)
		{
			usage();
		}

		Cities cities = new Cities();
		cities.readFile(args[0]);

		// start at vertex 0, because it is the center city of the empire
		int startingVertex = 0;

		cities.minSpanningTree(startingVertex);
		totalDistance = 0.0;
		E = V - 1;
		String city = args[0].substring(0, args[0].indexOf('-'));
		cities.writeFile(city + "-roads.txt");

		// totalDistance is calculated in writeFile()
		System.out.println("Total distance = " + new
DecimalFormat("0.000").format(totalDistance));
	}

	/**
	 * Compute the minimum spanning tree of the graph
	 * @param startingVertex - the starting vertex
	 * @return the vertices, as a minimum spanning tree
	 */
```

```
	public Vertex[] minSpanningTree(int startingVertex)
	{
		PriorityQueue<Vertex> minPQ = new PriorityQueue<Vertex>();

		for(int i = 0; i < vertices.length; i++)
		{
			vertices[i].setPredecessor(Integer.MIN_VALUE);
			vertices[i].setDistance(Double.POSITIVE_INFINITY);
		}
		vertices[startingVertex].setDistance(0);

		for(int i = 0; i < vertices.length; i++)
		{
			minPQ.add(vertices[i]);
		}

		while(!minPQ.isEmpty())
		{
			Vertex v = minPQ.remove();

			// the original graph is completely connected, so every other
			// vertex is adjacent to v
			for(Vertex w : vertices)
			{
				if(v.getIndex() != w.getIndex())
				{
					double vwDist = euclideanDistance(v.getX(), v.getY(),
w.getX(), w.getY());
					if(w.enqueued() && (vwDist < w.getDistance()))
					{
						w.setPredecessor(v.getIndex());
						w.setDistance(vwDist);
						w.increasePriority();
					}
				}
			}
		}
		return vertices;
	}

	/**
	 * Write to a file, first the list of vertices, each with it's (x, y) coordinates
	 * @param outFile - the name of the file to write to
	 */
	public void writeFile(String outFile)
	{
		MathContext mc = new MathContext(6);
```

```
		try
		{
			BufferedWriter bw = new BufferedWriter(new FileWriter(outFile));
			bw.write("g " + V + " " + E + "\n");
			for(int i = 1; i < V; i++)
			{
				bw.write("d " + i + " " + new BigDecimal(vertices[i].getX(), mc) +
" " + new BigDecimal(vertices[i].getY(), mc) + "\n");
			}

			for(int i = 1; i < V; i++)
			{
				bw.write("e " + i + " " + vertices[i].getPredecessor() + "\n");
				totalDistance += vertices[i].getDistance();
			}
			bw.close();
		}
		catch (IOException e)
		{
			System.err.println("There was an error writing to the file.");
			e.printStackTrace();
		}
	}

	/**
	 * Reads and processes a file in the Graph File Format
	 * @param fileName - the file to process
	 * @return vertices - an array of vertices for the graph
	 * @throws Exception
	 */
	public Vertex[] readFile(String fileName)
	{
		String line = "";

		try
		{
			BufferedReader buff = new BufferedReader(new FileReader(fileName));

			boolean gLine = false;
			eLine = false;

			while((line = buff.readLine()) != null)
			{
				String[] lineArr;

				// ignore any blank line
				if(!line.equals(null) && !line.isEmpty())
				{
```

```
                                        lineArr = line.split(" ");

                                        /* Required. Edges of the graph.
                                        First field - source vertex #, 0 <= an int <= V-1
                                        Second field - destination vertex #, 0 <= an int <= V-1
                                        Third field - edge weight, a floating pt #. If 3rd field
omitted,
                                                assume to be 1 by default.
                                        */
                                        if(lineArr[0].equals("e"))
                                        {
                                                eLine = true;
                                                int index = Integer.parseInt(lineArr[1]);
                                                int neighbor = Integer.parseInt(lineArr[2]);

                                                initializeIfNull(index, V);
                                                initializeIfNull(neighbor, V);
                                                vertices[index].addNeighbor(neighbor);
                                                vertices[neighbor].addNeighbor(index);
                                        }
                                        /* Required. Occurs once, at beginning of file
                                        The parameters of the graph.
                                        First field is # of vertices V, an int >= 0
                                        Second field is # of edges E, an int >= 0
                                        */
                                        else if(lineArr[0].equals("g"))
                                        {
                                                gLine = true;
                                                int numVertices = Integer.parseInt(lineArr[1]);
// V and E are guaranteed to be >= 0
                                                int numEdges = Integer.parseInt(lineArr[2]);


                                                /* ok to do this here, because this will be
executed before
                                                any of the other if statements on this or any

other
                                                pass of the while loop */
                                                V = numVertices;
                                                E = numEdges;
                                                vertices = new Vertex[V];
                                        }
                                        // ignore
                                        else if(lineArr[0].equals("v"))
                                        {
                                                continue;
                                        }
                                        // coordinates of the vertex
                                        else if(lineArr[0].equals("d"))
```

```
                                        {
                                                int index = Integer.parseInt(lineArr[1]);
                                                double x = Double.parseDouble(lineArr[2]);
                                                double y = Double.parseDouble(lineArr[3]);

                                                initializeIfNull(index, V);
                                                vertices[index].setX(x);
                                                vertices[index].setY(y);
                                                vertices[index].setIndex(index);
                                        }
                                        // ignore
                                        else if(lineArr[0].equals("c"))
                                        {
                                                continue;
                                        }
                                }// end if
                        }// end while

                        buff.close();

                        // make sure the file contained the required lines
                        if(gLine == false)
                        {
                                System.err.println("The file is in an invalid format.");
                                System.exit(0);
                        }
                }
                catch(NumberFormatException nfe)
                {
                        System.err.println("A number in the file had invalid format");
                        nfe.printStackTrace();
                }
                catch(Exception ex)
                {
                        System.err.println("There was an error reading the file");
                        ex.printStackTrace();
                }
                return vertices;
        }// end readFile()

        /**
         * Compute the Euclidean distance between two points, given the (x, y)
         * coordinates of each.
         * @param x1
         * @param y1
         * @param x2
         * @param y2
         * @return the distance
```

```
	 */
	public static double euclideanDistance(double x1, double y1, double x2, double y2)
	{
		return Math.sqrt(Math.pow((x1 - x2), 2) + Math.pow((y1 - y2), 2));
	}

	/**
	 * Check if a vertex is null, and if it is, initialize it.
	 * @param index - the vertex to check
	 * @param V - the number of vertices
	 */
	private static void initializeIfNull(int index, int V)
	{
		if(vertices[index] == null)
		{
			vertices[index] = new Vertex(index, V);
		}
	}

	/**
	 * @return true - if the file contains edge lines
	 *		false - if it does not
	 */
	public boolean fileHasEdges()
	{
		return eLine;
	}

	/**
	 * Print a usage message and exit
	 */
	public static void usage()
	{
		System.err.println("Usage: java Cities <fileName>"
				+ "<fileName> = name of a file in graph file format");
		System.exit(0);
	}
}


import java.math.BigDecimal;
import java.math.MathContext;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.LinkedList;
import java.util.AbstractMap.SimpleEntry;
import java.util.List;
```

```
/**
 * Calculate the betweenness centrality of a graph,
 * given the graph in Graph File Format
 * @author Joseph Ville
 *
 */
public class BetweennessCentrality
{
	private static int V; // number of vertices

	/**
	 * The main method for this program
	 * @param args
	 */
	public static void main(String[] args)
	{
		if(args.length != 1)
		{
			usage();
		}
		Vertex[] vertices;
		int capital; // the rank of the capital city
		Cities cities = new Cities();

		cities.readFile(args[0]);
		vertices = cities.minSpanningTree(0);
		V = vertices.length;
		populateNeighbors(vertices);
		List<SimpleEntry<Integer, Double>> bc = betweennessCent(vertices);
		MathContext mc = new MathContext(6); // to format numbers to 6 sig digs
		capital = -1; // should never be -1 one when trying to access it, and we want to
know if it is.

		System.out.println("Rank\tVertex\tBetweenness");
		for(int i = 0; i < V; i++)
		{
			if(i < 40)
			{
				System.out.println((i+1) + "\t" + bc.get(i).getKey() + "\t" + new
BigDecimal(bc.get(i).getValue(), mc));
			}
			if(bc.get(i).getKey() == 0)
			{
				capital = i;
			}
		}
		System.out.println("\nCapital city:");
```

```
                System.out.println((capital+1) + "\t" + bc.get(capital).getKey() + "\t" + new
BigDecimal(bc.get(capital).getValue(), mc));
        }

        /**
         * Populate the neighbor field for each vertex if there are no
         * "e" lines in the graph file
         * @param vertices
         */
        public static void populateNeighbors(Vertex[] vertices)
        {
                for(int i = 1; i < vertices.length; i++)
                {
                        vertices[i].addNeighbor(vertices[i].getPredecessor());
                        vertices[vertices[i].getPredecessor()].addNeighbor(i);
                }
        }

        /**
         * Compute the betweenness centrality of every vertex
         * @param vertices
         * @return a List<SimpleEntry<Integer, Double>> that maps the vertex number (the
index) to its betweenness
         */
        public static List<SimpleEntry<Integer, Double>> betweennessCent(Vertex[] vertices)
        {
                List<SimpleEntry<Integer, Double>> bc = new ArrayList<SimpleEntry<Integer,
Double>>();
                int countTotal = 0;
                for(int i = 0; i < V; i++)
                {
                        for(int j = i; j < V; j++)
                        {
                                // if they are not the same vertex
                                if(i != j)
                                {
                                        bfs(vertices, i, j);
                                        countTotal++;
                                }
                        }
                }
                for(int i = 0; i < V; i++)
                {
                        vertices[i].divideNumPaths(countTotal);
                        bc.add(new SimpleEntry<Integer, Double>(i,
vertices[i].getNumPaths()));
                }
```

```
		// sort the list according to the values (in this case, the betweenness)
		Collections.sort(bc, new Comparator<SimpleEntry<Integer, Double>>(){

			/**
			 * Compare the values of two of the entries of the comparator
			 * @param arg0
			 * @param arg1
			 * @return a negative integer if arg0 <  arg1
			 *                     zero                     if arg0 == arg1
			 *                     a positive integer if arg0 >  arg1
			 */
			@Override
			public int compare(SimpleEntry<Integer, Double> arg0,
SimpleEntry<Integer, Double> arg1)
			{
				return arg1.getValue().compareTo(arg0.getValue());
			}

		});
		return bc;
	}

	/**
	 * Perform a breadth-first search on an array of vertices
	 * @param vertices - array of vertices
	 * @param start - starting vertex
	 * @param dest - destination vertex
	 */
	public static void bfs(Vertex[] vertices, int start, int dest)
	{
		int[] parent = new int[vertices.length];
		boolean[] seen = new boolean[vertices.length];
		LinkedList<Integer> queue;

		queue = new LinkedList<Integer>();
		seen[start] = true;
		queue.add(start);

		int current = Integer.MIN_VALUE;
		while(!queue.isEmpty())
		{
			current = queue.poll();
			ArrayList<Integer> neighbors = vertices[current].getNeighbors();
			for(int n = 0; n < neighbors.size(); n++)
			{
				if(!seen[neighbors.get(n)])
				{
					seen[neighbors.get(n)] = true;
```

```
                                        queue.add(neighbors.get(n));
                                        parent[neighbors.get(n)] = current;
                                }
                        }
                }

                // backtrack to update number of paths
                int y = dest;
                while(y != start)
                {
                        if(parent[y] != start)
                        {
                                vertices[parent[y]].incrementNumPaths();
                        }
                        y = parent[y];
                }
        }

        /**
         * Print a usage message and exit
         */
        public static void usage()
        {
                System.err.println("Usage: java BetweennessCentrality <fileName>\n"
                                + "<fileName> = a file in Graph File Format");
                System.exit(0);
        }
}


import java.util.ArrayList;
import edu.rit.util.PriorityQueue.Item;

/**
 * A class to represent a vertex object
 * @author Joseph Ville
 */
public class Vertex extends Item
{
        private ArrayList<Integer> neighbors; // neighbors of the current vertex
        private double x, y; // the coordinates for this vertex
        private int index; // the index of this vertex
        private double distance; // the shortest distance from this to any other vertex in the
graph
        private int predecessor; // predecessor to this vertex

        // number of min length paths where this vertex appears. Will become the centrality
        private double numPaths;
```

```
/**
 * Construct an object of this class
 * @param index
 */
public Vertex(int index, int V)
{
	neighbors = new ArrayList<Integer>();
	this.x = 0;
	this.y = 0;
	numPaths = 0.0;
}

/**
 * Construct an object of this class
 * @param index
 * @param distance
 */
public Vertex(int index, double distance)
{
	neighbors = new ArrayList<Integer>();
	this.x = 0;
	this.y = 0;
	this.distance = distance;
}

/**
 * @return true - if the distance of this item is less than
 *                                        the distance of the item passed in
 *                      false - otherwise
 */
@Override
public boolean comesBefore(Item arg0)
{
	if(this.distance < ((Vertex)arg0).distance)
	{
		return true;
	}
	return false;
}

/**
 * Find the degree of this vertex. Will only work after
 * neighbors list has finished populating.
 * @return the degree of this vertex
 */
public int degree()
{
```

```
		return neighbors.size();
}

/**
 * Get the list of vertices that are adjacent to the current vertex
 * @return the list of neighbors
 */
public ArrayList<Integer> getNeighbors()
{
		return neighbors;
}

/**
 * Adds a vertex number to the end of this vertex's neighbor list
 * @param n - the neighbor to be added
 */
public void addNeighbor(int n)
{
		this.neighbors.add(n);
}

/**
 * @return the y
 */
public double getY()
{
		return y;
}

/**
 * @param y the y to set
 */
public void setY(double y)
{
		this.y = y;
}

/**
 * @return the x
 */
public double getX()
{
		return x;
}

/**
 * @param x the x to set
 */
```

```
public void setX(double x)
{
	this.x = x;
}

/**
 * @return the index
 */
public int getIndex()
{
	return index;
}

/**
 * @param index the index to set
 */
public void setIndex(int index)
{
	this.index = index;
}

/**
 * @return the distance
 */
public double getDistance()
{
	return distance;
}

/**
 * @param distance the distance to set
 */
public void setDistance(double distance)
{
	this.distance = distance;
}

/**
 * @return the predecessor
 */
public int getPredecessor()
{
	return predecessor;
}

/**
 * @param predecessor the predecessor to set
 */
```

```
	public void setPredecessor(int predecessor)
	{
		this.predecessor = predecessor;
	}

	/**
	 * Increment the numPaths variable
	 */
	public void incrementNumPaths()
	{
		numPaths++;
	}

	/**
	 * @return the number of paths for this vertex
	 */
	public double getNumPaths()
	{
		return numPaths;
	}

	/**
	 * divide the number of paths variable by the given divisor
	 * and store it back in the same variable
	 * @param divisor
	 */
	public void divideNumPaths(int divisor)
	{
		numPaths /= divisor;
	}
}// end class Vertex
```

4. These results were generated using the following command line:

```
java Cities egypt-cities.txt
```

Results

```
g 200 199
d 1 0.374516 11.8515
d 2 -4.19318 -10.1176
d 3 -15.9191 -2.80382
d 4 6.56390 -2.66122
d 5 4.40599 -4.48003
d 6 -7.93670 15.8426
d 7 -10.5291 -1.45591
d 8 -1.84183 1.41522
d 9 -2.07661 -0.996556
d 10 -1.26193 -0.389156
d 11 -19.1375 -1.30398
d 12 -7.88723 -3.79586
```

d 13 -5.71363 12.1560
d 14 -4.12140 1.89049
d 15 -0.310026 4.84681
d 16 0.372065 10.4306
d 17 4.95673 -4.78604
d 18 10.0259 -4.27128
d 19 -5.20532 -2.25130
d 20 16.7302 2.19480
d 21 16.1915 -7.49015
d 22 8.78035 16.9592
d 23 -2.48731 2.74142
d 24 -1.88856 2.28969
d 25 7.02632 -5.50792
d 26 11.1158 -8.40500
d 27 -5.20020 -3.05843
d 28 10.2620 -12.3961
d 29 -7.84394 8.40742
d 30 -13.3078 -12.0350
d 31 10.7061 7.85033
d 32 12.0237 -8.46813
d 33 4.68870 10.3691
d 34 5.88120 -11.5094
d 35 -6.79134 -1.94784
d 36 -1.03871 1.34506
d 37 14.3652 -11.2637
d 38 -11.3120 3.24297
d 39 -14.7397 7.65741
d 40 10.6034 9.10640
d 41 0.288905 2.59327
d 42 13.1986 11.0576
d 43 -5.03105 -10.6151
d 44 2.08053 -0.709623
d 45 11.1081 -12.1008
d 46 -9.62969 6.61208
d 47 -8.11588 14.9199
d 48 6.12416 -6.64726
d 49 2.04420 3.05175
d 50 6.62231 -12.7654
d 51 -18.1448 -7.73453
d 52 7.10874 9.88424
d 53 -4.98767 -9.95043
d 54 2.52237 -6.32603
d 55 4.94612 5.97579
d 56 10.2053 -5.33742
d 57 2.54400 -1.93802
d 58 17.8182 -0.0858070
d 59 1.77975 10.1789
d 60 -12.4406 6.10013
d 61 -14.0285 13.0744
d 62 1.25402 -4.32587
d 63 -2.13447 3.59990
d 64 7.44207 5.44760

d 65 1.57810 2.56362
d 66 -1.30629 -5.71431
d 67 -0.284930 11.1226
d 68 6.73032 -1.33189
d 69 13.8373 14.3964
d 70 -5.20615 14.0361
d 71 3.71100 2.93542
d 72 -1.57614 -0.904914
d 73 -1.85063 -0.797475
d 74 4.31528 4.57171
d 75 -17.5724 6.76661
d 76 11.2213 6.43399
d 77 -3.12578 -1.25554
d 78 -1.18423 -0.839497
d 79 1.01957 0.145833
d 80 14.0095 3.58280
d 81 4.14253 12.6557
d 82 0.916972 0.610410
d 83 2.31860 11.3358
d 84 -3.34227 -7.47182
d 85 -4.18910 -11.7352
d 86 5.81203 11.7787
d 87 -1.88569 11.3618
d 88 12.6706 -10.1302
d 89 0.701146 -10.1129
d 90 -9.31903 -4.00654
d 91 -1.45574 7.10204
d 92 2.54218 1.22418
d 93 -2.63311 8.69288
d 94 -13.9180 2.55105
d 95 7.90191 12.0468
d 96 -15.9231 -9.93038
d 97 -0.506278 2.04269
d 98 2.05319 -12.0404
d 99 -1.23767 -5.07120
d 100 -9.14798 -10.6501
d 101 14.4618 -7.75749
d 102 17.3866 -2.60072
d 103 3.27979 -2.20611
d 104 2.62175 6.81040
d 105 8.65539 11.0432
d 106 -18.5279 2.60341
d 107 -0.496561 19.9859
d 108 13.0399 1.47389
d 109 9.09669 3.34045
d 110 -1.26872 0.342563
d 111 7.71581 7.50644
d 112 -11.1645 9.42646
d 113 16.4059 -2.13230
d 114 10.9545 10.3073
d 115 8.58500 -13.8326
d 116 4.69043 5.45680

d 117 6.61391 -8.83125
d 118 6.86351 -0.0141145
d 119 -0.586094 14.5984
d 120 7.43752 -3.28053
d 121 -5.08337 1.48833
d 122 -15.5554 4.11732
d 123 4.29406 1.28505
d 124 -13.3208 -13.9853
d 125 1.63197 -2.60365
d 126 1.03019 -7.57558
d 127 -9.82508 -6.62832
d 128 0.249301 -12.2895
d 129 3.33742 4.19354
d 130 6.88700 18.4488
d 131 -12.9876 2.49946
d 132 -5.80827 2.27374
d 133 -16.9364 -1.80101
d 134 11.4924 -12.5771
d 135 8.40390 -2.85103
d 136 -2.44695 -2.16922
d 137 -2.45669 -18.4102
d 138 -3.53462 -10.1812
d 139 -0.464344 3.59867
d 140 -7.17332 4.75287
d 141 1.75211 3.49084
d 142 5.62832 2.56412
d 143 -15.3703 11.7326
d 144 6.01202 -16.9944
d 145 -4.61095 1.39760
d 146 8.50867 -15.7433
d 147 -3.97144 -10.7368
d 148 -4.36141 -10.7994
d 149 -0.0624049 4.58278
d 150 2.85016 2.13093
d 151 8.97852 3.69585
d 152 9.31525 -2.25507
d 153 6.96846 -13.1173
d 154 1.26783 1.30894
d 155 1.70990 0.444944
d 156 -12.7361 7.82844
d 157 9.17678 -10.9941
d 158 -0.301406 1.47213
d 159 -2.23335 1.82041
d 160 -8.92361 -17.5122
d 161 6.31726 -13.3258
d 162 5.69578 10.7648
d 163 0.0982780 -3.23040
d 164 0.806370 14.7430
d 165 18.0659 -2.63652
d 166 -16.1293 2.33408
d 167 -15.4295 3.73886
d 168 6.23165 -2.53043

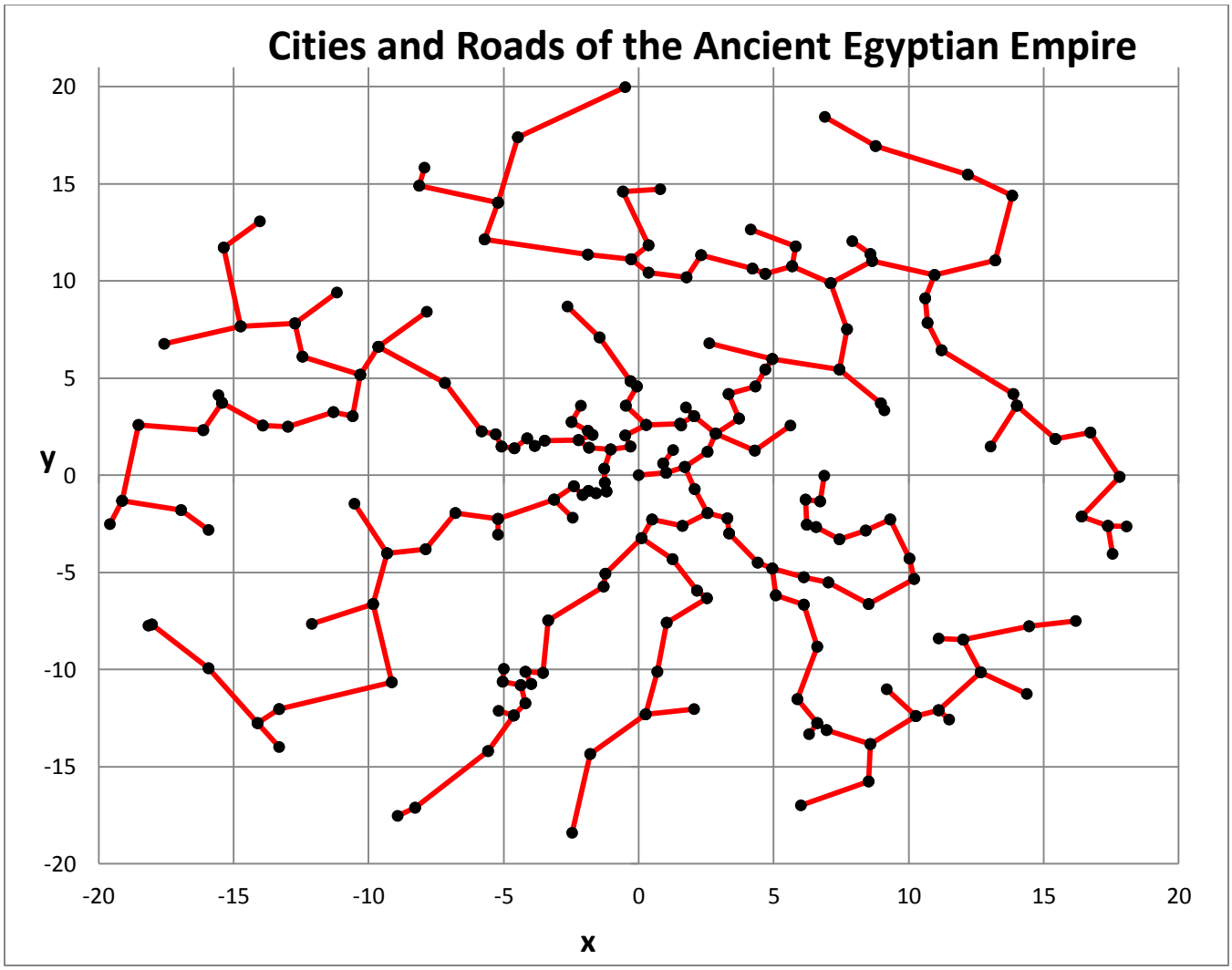
```
d 169 -3.85068 1.51167
d 170 -5.18591 -12.1136
d 171 1.54197 2.65836
d 172 -5.29642 2.12011
d 173 -19.5887 -2.51334
d 174 3.35115 -2.98725
d 175 0.501583 -2.27078
d 176 6.18760 -1.25214
d 177 12.1946 15.4791
d 178 -1.80288 -14.3547
d 179 -4.62979 -12.3565
d 180 -14.1181 -12.7439
d 181 -1.71185 2.07373
d 182 -10.5999 3.04321
d 183 5.08140 -6.18191
d 184 6.12812 -5.24011
d 185 -5.56531 -14.1923
d 186 8.52456 -6.63629
d 187 -10.3100 5.18920
d 188 -2.40383 -0.545177
d 189 4.21659 10.6390
d 190 -8.28797 -17.1005
d 191 17.5488 -4.04490
d 192 13.8686 4.18230
d 193 -3.48147 1.78585
d 194 2.17103 -5.92323
d 195 15.4320 1.87333
d 196 -12.1026 -7.63616
d 197 -4.46946 17.4123
d 198 8.57445 11.3847
d 199 -18.0236 -7.66248
e 1 67
e 2 138
e 3 133
e 4 120
e 5 174
e 6 47
e 7 90
e 8 36
e 9 73
e 10 110
e 11 106
e 12 35
e 13 87
e 14 169
e 15 149
e 16 59
e 17 5
e 18 56
e 19 77
e 20 195
e 21 101
```

e 22 177
e 23 24
e 24 181
e 25 184
e 26 32
e 27 19
e 28 115
e 29 46
e 30 100
e 31 40
e 32 88
e 33 162
e 34 117
e 35 19
e 36 158
e 37 88
e 38 182
e 39 156
e 40 114
e 41 171
e 42 114
e 43 148
e 44 155
e 45 28
e 46 140
e 47 70
e 48 183
e 49 150
e 50 34
e 51 199
e 52 111
e 53 43
e 54 194
e 55 116
e 56 186
e 57 44
e 58 20
e 59 83
e 60 187
e 61 143
e 62 163
e 63 23
e 64 55
e 65 171
e 66 99
e 67 16
e 68 176
e 69 42
e 70 13
e 71 150
e 72 78
e 73 72

e 74 129
e 75 39
e 76 31
e 77 188
e 78 10
e 79 0
e 80 192
e 81 86
e 82 79
e 83 189
e 84 66
e 85 148
e 86 162
e 87 67
e 88 45
e 89 126
e 90 12
e 91 15
e 92 155
e 93 91
e 94 131
e 95 198
e 96 180
e 97 41
e 98 128
e 99 163
e 100 127
e 101 32
e 102 113
e 103 57
e 104 55
e 105 52
e 106 166
e 107 197
e 108 80
e 109 151
e 110 36
e 111 64
e 112 156
e 113 58
e 114 105
e 115 153
e 116 74
e 117 48
e 118 68
e 119 1
e 120 135
e 121 145
e 122 167
e 123 150
e 124 180
e 125 57

e 126 54
e 127 90
e 128 89
e 129 71
e 130 22
e 131 38
e 132 172
e 133 11
e 134 45
e 135 152
e 136 77
e 137 178
e 138 84
e 139 41
e 140 132
e 141 49
e 142 123
e 143 39
e 144 146
e 145 14
e 146 115
e 147 2
e 148 147
e 149 139
e 150 92
e 151 64
e 152 18
e 153 50
e 154 82
e 155 79
e 156 60
e 157 28
e 158 97
e 159 8
e 160 190
e 161 50
e 162 52
e 163 175
e 164 119
e 165 102
e 166 167
e 167 94
e 168 4
e 169 193
e 170 179
e 171 49
e 172 121
e 173 11
e 174 103
e 175 125
e 176 168
e 177 69

e 178 128
e 179 85
e 180 30
e 181 159
e 182 187
e 183 17
e 184 17
e 185 179
e 186 25
e 187 46
e 188 9
e 189 33
e 190 185
e 191 102
e 192 76
e 193 159
e 194 62
e 195 80
e 196 127
e 197 70
e 198 105
e 199 96



5. The total distance of all the roads in the Egyptian empire is 314.058.
Command line used: `java Cities egypt-cities.txt`

6. Command line used: `java BetweennessCentrality egypt-cities.txt`

Rank	Vertex	Betweenness
1	150	0.662814
2	57	0.484372
3	155	0.467035
4	41	0.463467
5	49	0.463065
6	36	0.462513
7	171	0.457236
8	92	0.456683
9	44	0.437688
10	97	0.426834
11	158	0.423015
12	71	0.379296

13	129	0.374372
14	74	0.369347
15	116	0.364221
16	55	0.361307
17	64	0.352563
18	52	0.346533
19	111	0.331357
20	103	0.294874
21	8	0.288442
22	174	0.288442
23	17	0.288342
24	159	0.287940
25	5	0.281910
26	193	0.247739
27	169	0.240603
28	14	0.233367
29	125	0.226030
30	145	0.226030
31	121	0.218593
32	175	0.218593
33	163	0.217839
34	172	0.211055
35	110	0.203417
36	132	0.203417
37	10	0.195678
38	140	0.195678
39	105	0.189749
40	46	0.188844

7. Alexandria's betweenness centrality is 0, and it's rank is 157. Based on this calculation, I conclude that Pharaoh will most definitely have me decapitated.

8. In completing this project, I have gained a better understanding of betweenness centrality. I learned about it in class, but did not fully understand it. Using it in such a practical application for this project helped me learn about it more fully.

   I also found it interesting that Alexandria was so far from being the most central city. This makes me wonder why Alexandria was even the capital, with it being over three-quarters of the way down the list.