Date: 22/10/2016

Michal Bochenek
Software Engineering year 3

Student ID: 40270585

Course ID: SET09117
Algorithms and Data Structures

Report

“Travelling Salesman Problem” Algorithm

Contents

1. Introduction:

This report is an overview about the results of a working algorithm designed for solving “travelling salesman problem”(TSP), it contains description of method taken in solving the problem, shows output results and conclusions taken from them and overall student performance.

TSP is a well-known issue where one needs to find an optimal path between cities aka points on the 2D graph by finding the shortest possible route between the point of origin (first city) and the last available unvisited location.

The problem is considered complex, the simplest solution would be analysing each possible route between points, which is reasonable for small data amounts as the complexity is (n-1)! possibilities but it grows potentially with the number of locations given to such a difficult calculation that it would take weeks or even more to calculate it on powerful machines using this method.

In order to solve this problem a number of intermediate solutions were designed to try to approximate the result, but it will nearly never produce an optimal output for large problems.

The chosen approach used to solve this algorithm was the nearest neighbour algorithm which was one of the earliest solutions given to this problem.

It works in steps as shown below:

1. Identify starting location, remove it from the list, assign it as a starting city.
2. Search and find the nearest neighbour to the starting city from the list.
3. Remove the found location and add it to visited list after starting city.
4. Replace current starting city with its nearest neighbour
5. Repeat from step 2, while there are positions on the initial list.
6. Produce result – the salesman's journey path in form of list.

Nearest neighbour algorithm, despite the fine performance, has a number of issues such as it sometimes can produce tour far from being optimal.
The easiest way to view the quality of the created journey is to compare initial steps chosen by the algorithm to the last few ones, if the routes are much longer or are "jumping" from one extreme to another then it is clearly something not right, probably more optimal path was omitted by the imperfection of the program design. As the list is getting shorter the lesser the choice of locations is given to the algorithm.
The complexity of the nearest neighbour algorithm is $O(n^2*2^n)$ which isn't very efficient (doesn't solve it polynomial time) but for the small and medium sized data it is still enough and much better than searching through all the possible solutions (O(n-1)!/2) in a reasonable time.["Wikipedia", *n.d.*]

2. Method taken:

To solve the problem a traditional methodology was chosen in which the problem solution was divided into steps such as:

1. Planning – based on assignment given the required tasks were identified.
2. Design and development – coding the problem using Java version 8 as a language and Eclipse, which was based on the pseudo-code given in assignment and research

of the existing solutions for the nearest neighbour algorithm.

3. Testing – the code was tested numerous times on 5 chosen data sets and additional few which were not included in the final results as they contained different data types. Student home computer was used for each test to repeat the initial conditions for each test. Each test run resulted with the same execution time for each data set thus there was no need to average the results.

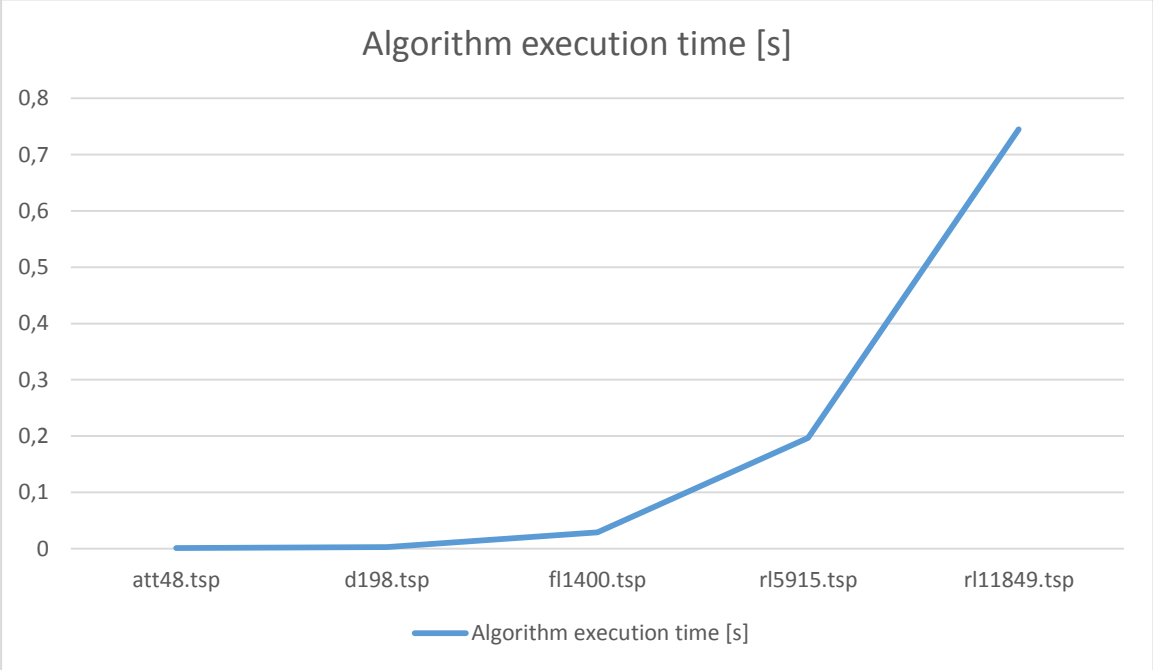
The instances (data sets) were selected from the existing data files provided by the lecturer, containing locations of the cities in form of coordinates (x,y) they were read by the program and checked for errors or wrong/missing data types i.e. Integers instead floating numbers were not accepted.
Five chosen data sets differ from each other by the order of complexity:

att48.tsp – 48 locations
d198.tsp – 198 locations
fl1400.tsp – 1400 locations
rl5915.tsp – 5915 locations
rl11849.tsp – 11849 locations

This is to ensure that the algorithm is not solving only the simplest sets and to obtain the difference in timings for small and medium data sets and discover possible errors described in paragraph 1.

3. Results:

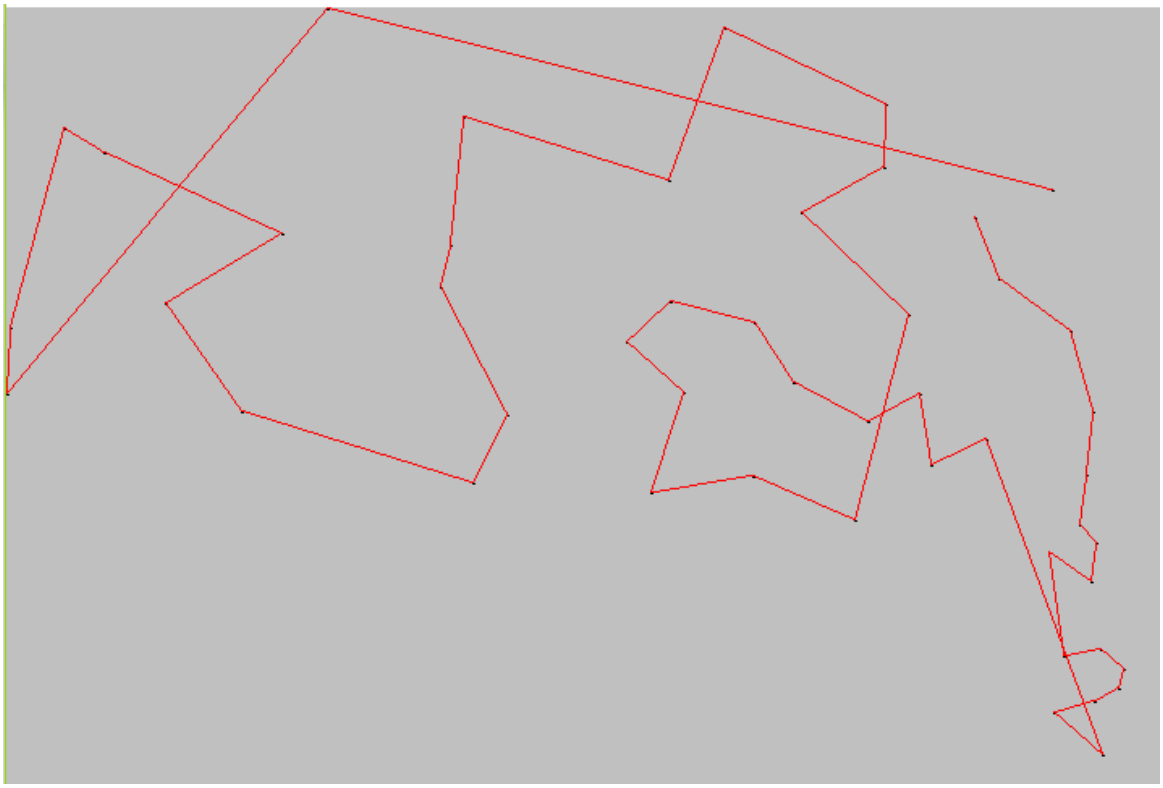
Timing graph:



Output data:

File name:	Execution time [s]:	Distance travelled [units]:
att48.tsp	0.001	39964.12
d198.tsp	0.003	14749.06
fl1400.tsp	0.029	25059.69
rl5915.tsp	0.197	692816.72
rl11849.tsp	0.745	1129930.15

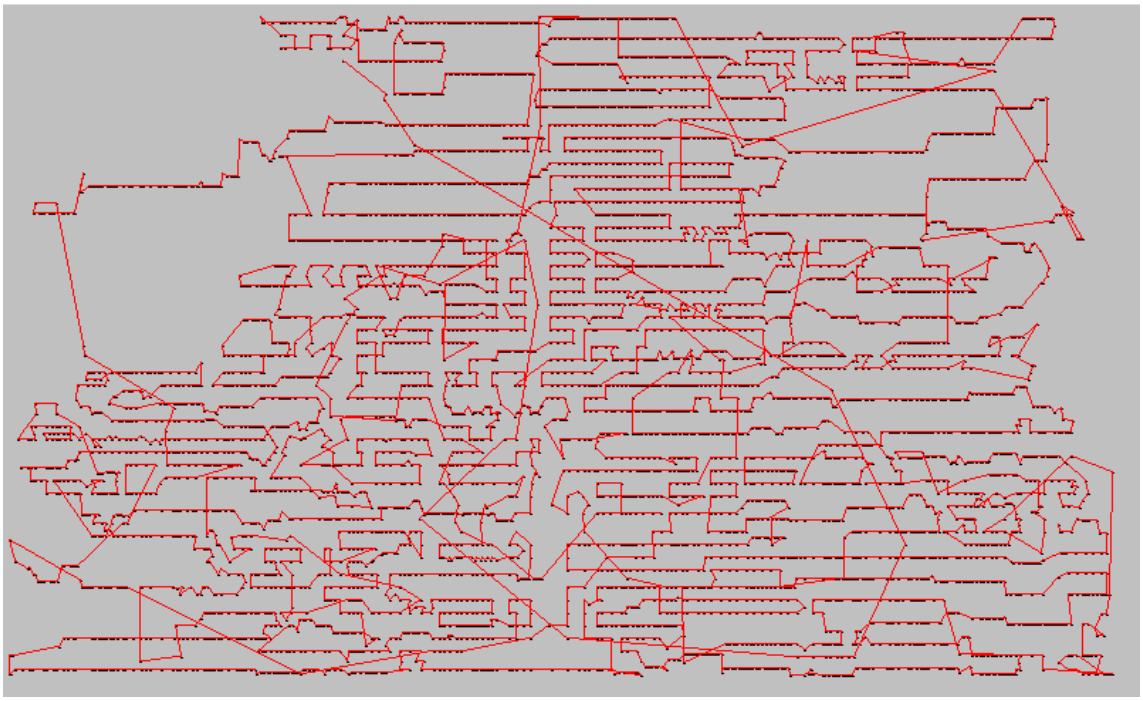
Graphical representation of the NN for att48.tsp:



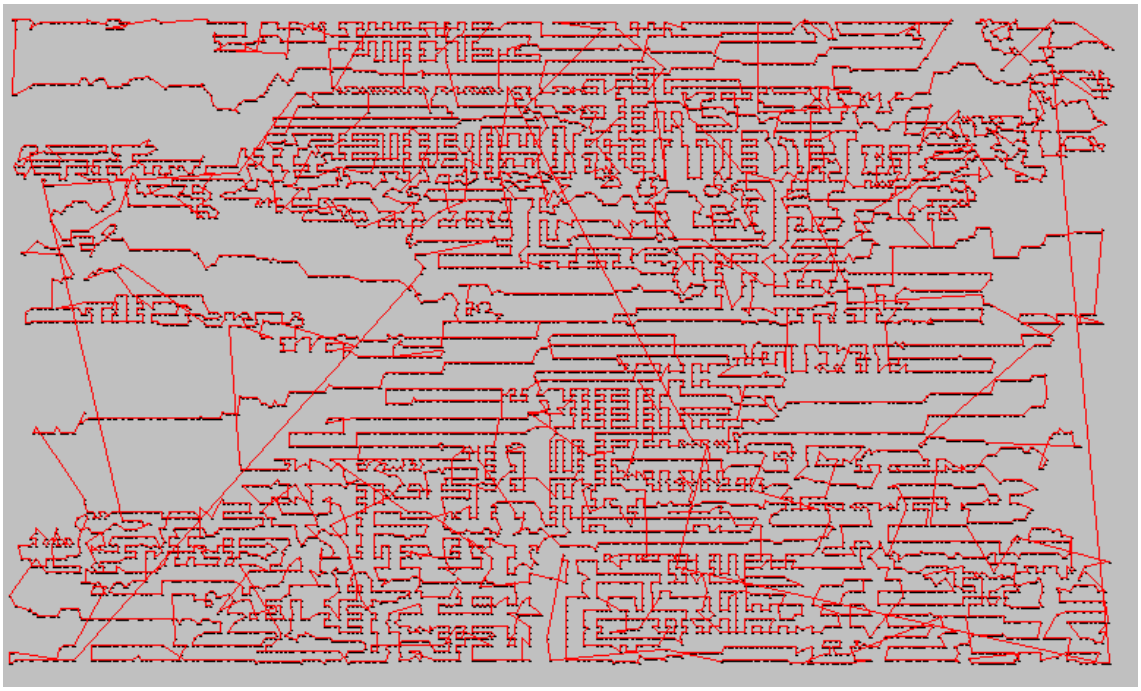
Graphical representation of the NN for d198.tsp:

Graphical representation of the NN for fl1400.tsp:

Graphical representation of the NN for rl5915.tsp:



Graphical representation of the NN for rl11849.tsp:



4. Conclusions:

The results obtained shown in the paragraph 3 prove exactly that the nearest neighbour algorithm has a potential to quickly and with a good accuracy solve the problem for small data sets as seen for file att48.tsp and d198.tsp.
No outstanding issues for these two output solutions can be easily found.
It is worth mentioning that for nearly 4 times more cities in latter set the algorithm execution time grows only from 0.001s to 0.003.
Data set for fl1400.tsp is hard to examine since the density of the locations is gathered around few centres but the solutions is given in reasonable time thus we can assume it is correct, compared with d198.tsp execution time is already nearly 9.7 times higher for over 7 times bigger data set.
First outstanding issue can be seen for rl5915.tsp where a few big “jumps” can be seen at the end of the path drawn by the algorithm, as mentioned before this usually is connected with omitting the optimal points by the algorithm, also for 4.225 times bigger data set it takes 6.8 more time than for the previous set.
The last data set - rl11849.tsp was chosen for its city amount, it also repeats the same issues found in previous result and execution time grows 3.8 times higher than rl5915.tsp used for just over 2 times bigger data.

The reliability of the results was ensured by keeping the testing environment in the same conditions for each attempt.
Java 8 Instant class was used to measure and record the execution time of the between the algorithm method start and stop.

The assignment was carried out without finding any overwhelming difficulties except for the time constraint for creating the solution code.
Although there are few fields in which it could be improved in future such as comparing the other algorithms using the same data sets, it was quickly discovered that there are better solutions such as k-nearest neighbour algorithm or more advanced methods. (En.wikipedia.org, 2016)

5. Appendix A – source code:

Main class – TSPmain.java:

```
package dataTSP;
//version 3 created on 21/10/2016
//student id: 40270585
//make sure that data files are placed inside the project root folder!

import java.awt.geom.Point2D;
import java.time.Duration;
import java.time.Instant;
import java.util.ArrayList;
import java.util.Scanner;

import javax.swing.JFrame;

public class TSPmain
{

	public static void main(String[] args)
	{
		//choose filename
		String filename = "";
		Scanner sc = new Scanner(System.in);
		System.out.println("This is algorithm solving travelling salesman
problem.");
		System.out.println("Please choose the file from the list below by
typing its name (name.tsp) and press enter:");
		System.out.println("Files are ordered descending, by the number of
cities present.");
		System.out.println("rl11849.tsp, rl5915.tsp, fl1400.tsp,  d198.tsp,
att48.tsp");
		filename = sc.next();
		sc.close();

		//use later to identify the graph scale
		int xmax=0;
		int ymax=0;
		int scale=0;
		//read choosen file and save city location details to arraylist
		ArrayList<Point2D> cityList = TSPSolution.loadTSPLib(filename);

		//find max value of x and y in the array list for later use in
setting up the scale of the graph
		for(int i=0; i<cityList.size(); i++)
		{
			if(cityList.get(i).getX()>xmax)
			{
				xmax=(int)cityList.get(i).getX();
			}
			if(cityList.get(i).getY()>ymax)
			{
				ymax=(int)cityList.get(i).getY();
			}
		}
```

```
            //an average of two furtherest values should produce reasonable
scale for the graph
            scale = xmax+ymax/2;

            if(!cityList.isEmpty())
            {
                  System.out.println("Cities loaded to array successfully,
proceeding...");
                  System.out.println("Graph scale set to: "+scale+"
proceeding...");
            }

            //create new TSPAlgorithm object and assign the loaded array
cityList to it
            TSPAlgorithm nearest_neighbour = new TSPAlgorithm(cityList);

            //measure method execution time from here:
            Instant start = Instant.now();
            //creates a new, sorted journey
            ArrayList<Point2D> journey = nearest_neighbour.TSPNN(cityList);
            //stop measurement
            Instant end = Instant.now();
            System.out.println("TSPAlgorithm duration time (0.000s):
"+Duration.between(start, end));

            //create new frame for displaying the graph
            JFrame frame = new JFrame();
            frame.setSize(1280, 960);
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            //this class can be fed with any solution in form of Point2D
arraylist to draw a journey graph
            TSPDrawGraph can1 = new TSPDrawGraph(journey, scale);
            frame.add(can1);
            frame.setVisible(true);
      }

}
```

Reading from file class – TSPSolution.java – provided by lecturer:

```
package dataTSP;
//version 1 created on 21/10/2016, using given code
//student id: 40270585
//make sure that data files are placed inside the project root folder!
//this class is reading points from a textfile and writing them to the arraylist

import java.awt.geom.Point2D;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;

public class TSPSolution
{
	public static ArrayList<Point2D> loadTSPLib(String fName)
	{
		//Load in a TSPLib instance. This example assumes that the Edge
weight type
		//is EUC_2D.
		//It will work for examples such as rl5915.tsp Other files such as
		//fri26.tsp .To use a different format, you will have to
		//modify the this code

		ArrayList<Point2D> result = new ArrayList<Point2D>();

		BufferedReader br = null;
		try
		{
			String currentLine;
			int dimension =0;//Hold the dimension of the problem
			boolean readingNodes = false;

			br = new BufferedReader(new FileReader(fName));

			while ((currentLine = br.readLine()) != null)
			{
				//Read the file until the end;
				if (currentLine.contains("EOF"))
				{
					//EOF should be the last line
					readingNodes = false;
					//Finished reading nodes
					if (result.size() != dimension)
					{
					//Check to see if the expected number of cities
have been loaded
					System.out.println("Error loading cities");
						System.exit(-1);
					}
				}

				if (readingNodes)
				{
					//If reading in the node data
					String[] tokens = currentLine.split(" ");
					//Split the line by spaces.
```

```
                                //tokens[0] is the city id and not needed in this
example
                                float x = Float.parseFloat(tokens[1].trim());
                                float y = Float.parseFloat(tokens[2].trim());
                                //Use Java's built in Point2D type to hold a city
                                Point2D city = new Point2D.Float(x,y);
                                //Add this city into the array list
                                result.add(city);
                        }

                        if (currentLine.contains("DIMENSION"))
                        {
                                //Note the expected problem dimension (number of
cities)
                                String[] tokens = currentLine.split(":");
                                dimension = Integer.parseInt(tokens[1].trim());
                        }

                        if (currentLine.contains("NODE_COORD_SECTION"))
                        {
                                //Node data follows this line
                                readingNodes = true;
                        }
                }
        }
        catch (IOException e)
        {
                e.printStackTrace();
        }
        finally
        {
                try
                {
                        if (br != null)br.close();
                }
                catch (IOException ex)
                {
                        ex.printStackTrace();
                }
        }

    return result;
    }

    //this is not solving the TSP! it just gives a total length of the routes
calculated without looking at the actual nearest 2 points on the list!
    public static double routeLength(ArrayList<Point2D> cities)
    {
        //Calculate the length of a TSP route held in an ArrayList as a set
of Points
        double result=0;//Holds the route length
        Point2D prev = cities.get(cities.size()-1);
        //Set the previous city to the last city in the ArrayList as we need
to measure the length of the entire loop
        for(Point2D city : cities)
        {
                //Go through each city in turn
                result += city.distance(prev);
                //get distance from the previous city
```

```
                prev = city;
                //current city will be the previous city next time
            }

            return result;
        }

}
```

Graph drawing class - TSPDrawGraph.java:

```
package dataTSP;
//version 5 created on 22/10/2016
//student id: 40270585
//this class is drawing cities and "roads" between them based on solution
arraylist

import java.awt.*;
import java.awt.geom.Point2D;
import java.util.ArrayList;

public class TSPDrawGraph extends Canvas
{
    private static final long serialVersionUID = 1L;
    private ArrayList <Point2D> locations = new ArrayList<>();
    private int res_scale;

    public TSPDrawGraph(ArrayList<Point2D> cityList, int scale)
    {
        locations.addAll(cityList);
        res_scale=scale;
    }

    //override canvas method
    public void paint(Graphics g)
    {
        g.setColor(Color.LIGHT_GRAY);
        //using default width and height of frame
        g.fillRect(0, 0, 1280, 960);

        //draw cities and roads between them
        paintCities(g, locations);
        paintRoads(g, locations);
    }

    //draw cities using locations
    private void paintCities(Graphics g, ArrayList<Point2D>locations)
    {
        g.setColor(Color.BLACK);
        //assume position 0 is a starting position
        //Point2D start = locations.get(0);
        //iterate through the list of cities and paint them on canvas
        for(Point2D i: locations)
        {
        //since pixels can't have floating points values, cast them to int
            int x = (int) ((i.getX()/res_scale)*getHeight());
            int y = (int) ((i.getY()/res_scale)*getHeight());
            g.fillOval(x, y, 2, 2);
        }
```

```
	}

	//draw roads
	private void paintRoads(Graphics g, ArrayList<Point2D>locations)
	{
		g.setColor(Color.RED);
		//iterate through the list of cities and paint the roads on canvas
		for(int r=0; r<locations.size()-1; r++)
		{
			int x1 = (int)
((locations.get(r).getX()/res_scale)*getHeight());
			int y1 = (int)
((locations.get(r).getY()/res_scale)*getHeight());
			int x2 = (int)
((locations.get(r+1).getX()/res_scale)*getHeight());
			int y2 = (int)
((locations.get(r+1).getY()/res_scale)*getHeight());
			//creates a line - road between 2 cities
			g.drawLine(x1, y1, x2, y2);
		}
	}
}
```

Nearest neighbour algorithm class - TSPAlgorithm.java:

```
package dataTSP;
//version 1 created on 22/10/2016
//student id: 40270585
//this class contains a solution algorithm for TSP
import java.util.ArrayList;
import java.awt.geom.Point2D;

public class TSPAlgorithm
{
	//initial arraylist containing all the cities to be visited
	private ArrayList <Point2D> cities = new ArrayList<>();
	//this will be the result after sorting it by the algorithm
	private ArrayList <Point2D> trip = new ArrayList<>();

	//create constructor to ensure that the city list is forwarded
	TSPAlgorithm(ArrayList<Point2D> citylist)
	{
		cities.addAll(citylist);
	}

	//nearest neighbour algorithm method
	public ArrayList<Point2D> TSPNN (ArrayList<Point2D> cities)
	{
		//starting point, can be upgraded later to prompt user to input the
starting city
		Point2D currentCity = cities.remove(0);
		//position of the closest city on the arraylist
		int index = 0;
		//create first shortest distance for the later use
		//make it very high thus it will always be replaced
		double min_dist = 999999999;
		double journey_length = 0;
```

```
            //do while array is not empty
while(cities.size()>0)
            {
                  trip.add(currentCity);

                  //search through the remaining cities for nearest neighbour
                  for(int i=0; i<cities.size(); i++)
                  {
                        //get distance between compared cities
                        double dist = (double)
currentCity.distance(cities.get(i));
                        //lesser than, will look for the first closest city
from the list
                        //may cause worthy considering issues if there are 2
closest cities
                        if(dist<min_dist)
                        {
                              min_dist=dist;
                              index = i;
                        }

                  }
                  //add closest city to the list
                  trip.add(cities.get(index));
                  //remove closest city from the cities array and add is as the
next city
                  currentCity=cities.remove(index);
                  journey_length = journey_length + min_dist;
                  //reset for loop values
                  index = 0;
                  min_dist= 999999999;

            }
            //returns a sorted array and measured total distance:
            System.out.println("Total distance travelled: "+journey_length);
            return trip;
      }
}
```

6. Appendix B – bibliography:

En.wikipedia.org. (2016). *K-nearest neighbors algorithm*. [online] Available at: https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm [Accessed 22 Oct. 2016].

En.wikipedia.org. (2016). *Nearest neighbour algorithm*. [online] Available at: https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm [Accessed 22 Oct. 2016].