Travelling Salesman Problem

40173513 *
Edinburgh Napier University
Algorithms and Data Structures

November 18, 2016

1 Introduction

This report is going to compare 3 different variations of the Nearest Neighbour algorithm, which is a common method of solving the travelling salesman problem.

First algorithm:

The first algorithm is a vanilla nearest neighbour solution, this will be used as a benchmark for the other 2 algorithms. Nearest neighbour takes the first city in the data structure as the starting city and then iterates through the data structure to find the next city, which will have the lowest distance value than other cities. It then continues to do this until all the cities from the data structure have been added to the final sorted data structure. This method should be rather efficient in finding a short route length and it shouldn't take that much time to solve either, however the time taken to solve should increase depending on the number of cities that it has to run through.

Second algorithm(Nearest Neighbour(Limit)):

The second algorithm used in this experiment is nearest neighbour with a tiny limit placed on it that will reduce the number of iterations it has to do through the data structure which should improve the time taken to solve by a small amount. The limitation itself is an assumption based one, where it has been assumed that the distances between the cities wont reach less than 50 often so the first city that reaches a distance of lower than 50 will be chosen as the closest city and the iteration loop itself will be stopped. This limitation should also result in the route taken being different from the vanilla nearest neighbour and that may have various effects on the route length.

*e-mail:40173513@live.napier.ac.uk

Third algorithm(Nearest Neighbour(Limit + Sort)):

The third algorithm is the same as the second algorithm but with a little twist, the data structure containing all the cities has been sorted before going through the nearest neighbour algorithm. Sorting the data structure will require time to be spent on the actual sorting, which already means this algorithm should solve the problem slower than nearest neighbour, at least for the smaller amounts of cities, it may be faster for larger data sets. The sorting method used sorts the data by comparing two cities using Equation 1 and then taking the bigger value and putting it on top. The algorithm then follows in the footsteps of the second algorithm by implementing the limit on the iteration loop. The route length may be better in larger data sets.

$$\sqrt{coordinate.x^2 + coordinate.y^2} \tag{1}$$

The code for all of these algorithms can be found in Section 5.

2 Method

The experiment has been conducted by running each file containing the cities individually for each algorithm, so for example, Berlin52.tsp is ran first by using nearest neighbour and the results are collected then the same city file is ran using the second algorithm and the results of that are collected etc.

The program has been run as an instance for each of the experiments as to avoid unnecessary code being run at the same time which could hinder the time taken to solve values, creating inaccuracies. Each algorithm and city file combination is ran 10 times to get a wider scope of results and then the average is taken as the final value, this also improves the accuracy of the results. This experiment was carried out on the same computer, so the hardware and other factors will not affect the results.

Data sets used:

- berlin52.tsp (52 cities)
- d493.tsp (493 cities)
- rl1889.tsp (1889 cities)
- rl5915.tsp (5915 cities)
- rl11849.tsp (11849 cities)

The city files used have been chosen as they can show a clear relationship between the amount of cities used and the time taken to solve, where the time taken will increase as the amount of cities gets larger.

3 Results

The averaged results of the conducted experiment can be seen at Table 1.

	Nearest Neighbour		Nearest Neighbour (Limit)		Nearest Neighbour(Limit + Sort)	
No. Cities	Route Length	Time (ms)	Route Length	Time (ms)	Route Length	Time (ms)
52	8980.918279	0.6	9208.118155	0.6	9504.133066	1.7
493	43646.37396	7.2	44362.33228	6.5	46135.9899	9.3
1889	400684.6384	28.5	400684.6384	31.7	407181.4464	36.7
5915	707498.6308	166.8	696508.3193	137.3	701014.4814	137.2
11849	1139495.914	643.4	1126219.47	549.4	1123646.172	581

Table 1: Average results

This table(see Table 1) already shows that as the number of cities increase, the time does increase with it. The second algorithm is faster for majority of the data sets than the vanilla nearest neighbour and the route length also starts to become better on larger data sets. The third algorithm only improves its time and route length on the larger data sets.

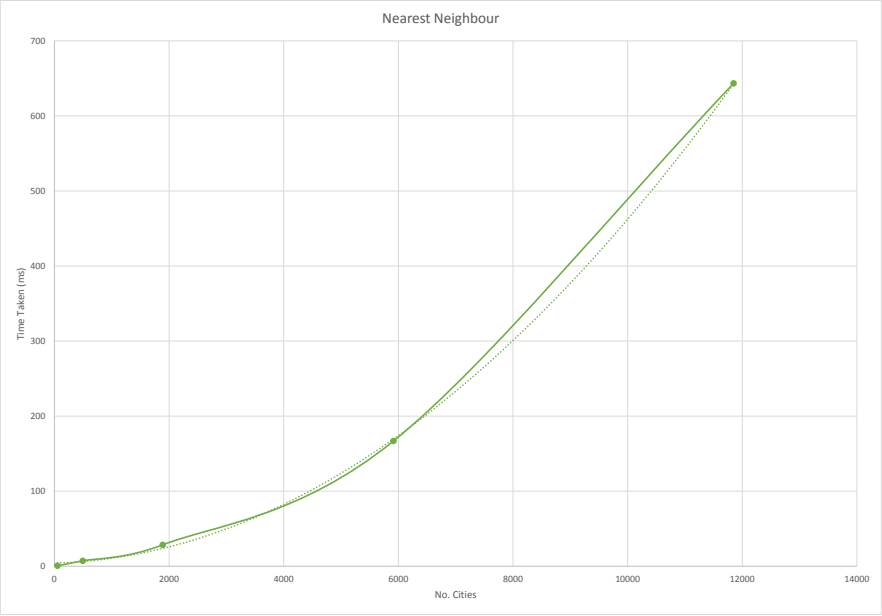

Figure 1: Graph showing Nearest Neighbours time to solve and number of cities relationship

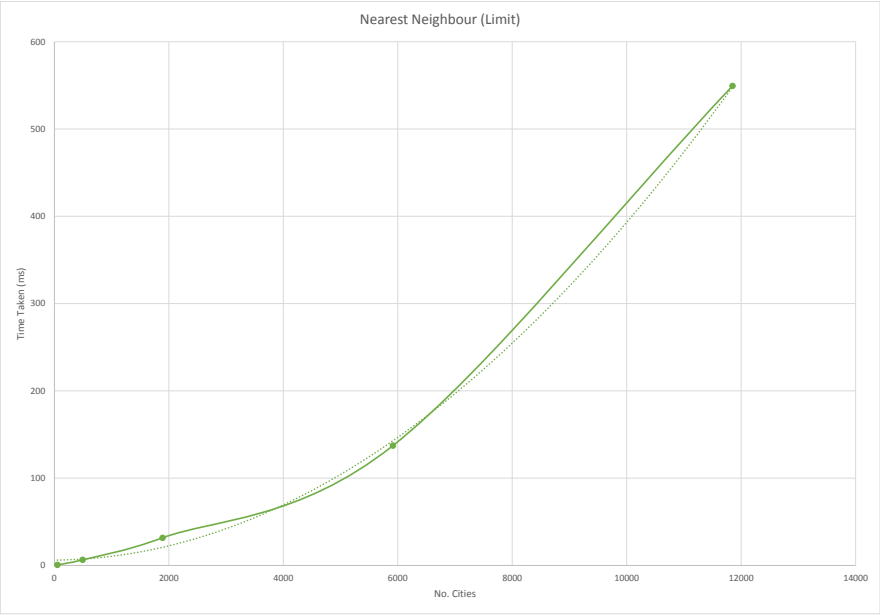

Figure 2: Graph showing Nearest Neighbours (Limit) time to solve and number of cities relationship

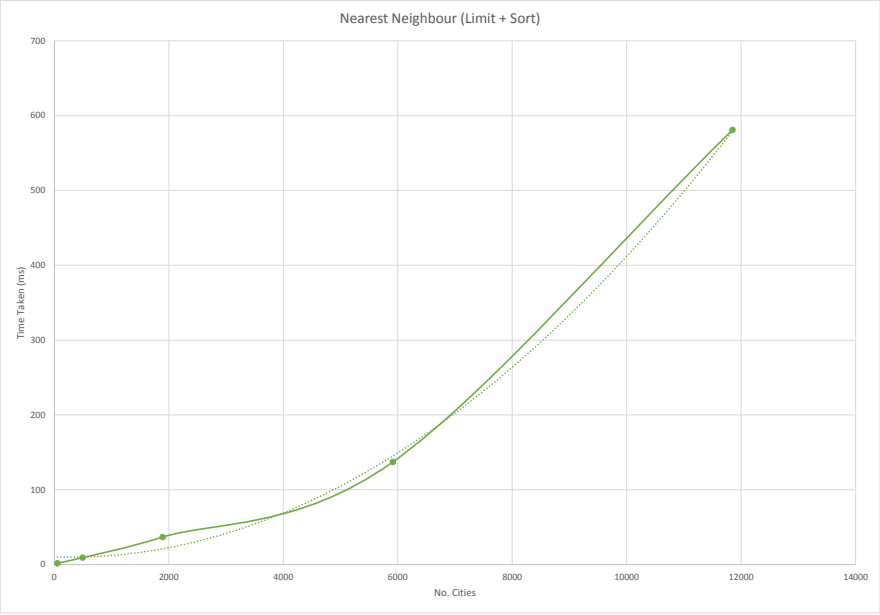

Figure 3: Graph showing Nearest Neighbours (Limit + Sort) time to solve and number of cities relationship

The graphs shown at Figure 1, Figure 2 and Figure 3 prove that there indeed is a relationship between the number of cities and the time taken to solve the problem. They also show that all algorithms tested here follow this rule.

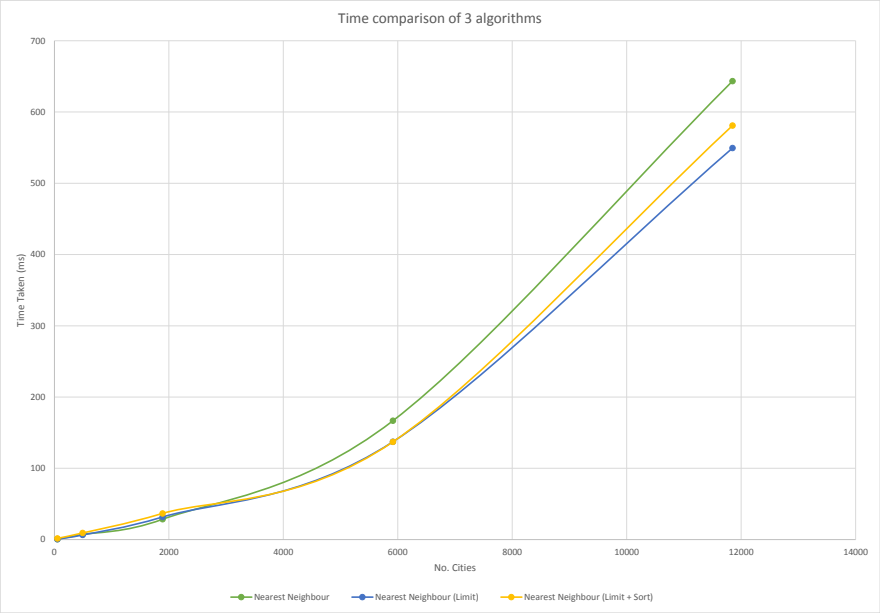

Figure 4: Graph comparing the time taken of all 3 algorithms

The graph shown at Figure 4 compares the 3 different algorithms in terms of time taken. As shown on this graph, the second algorithm(blue) is the best over all, second place is given to the third algorithm(yellow) (only after a certain number of cities is reached) and finally the vanilla nearest neighbour algorithm(green) takes the last place in terms of time taken to solve.

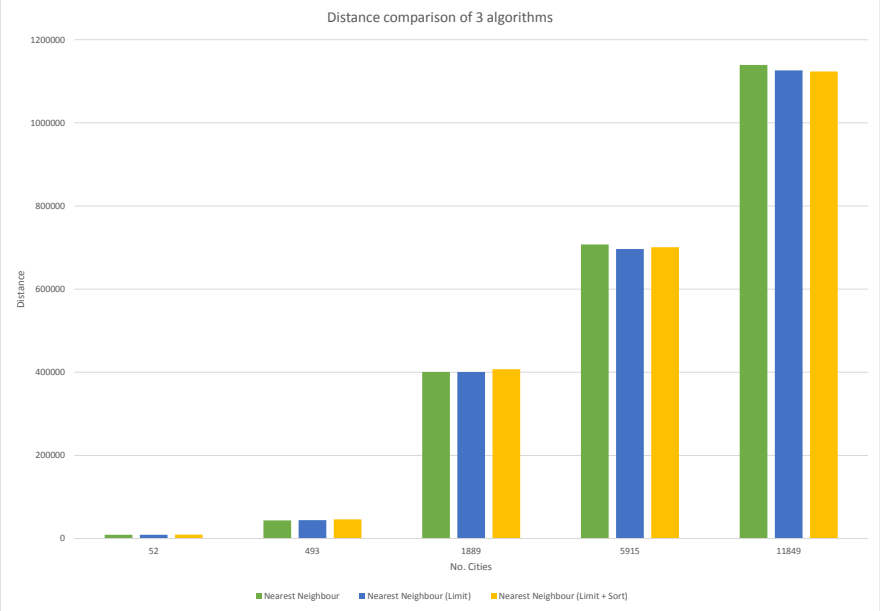

Figure 5: Bar chart comparing the route lengths of all 3 algorithms

The bar chart shown at Figure 5 compares the 3 different algorithm in terms of the route length. As shown on this bar chart, the first 2 data sets show that vanilla nearest neighbour(green) is superior, however at the 3rd data set the second algorithm(blue) produces the same route length and then is proven superior over the vanilla nearest neighbour algorithm. The third algorithm seems to become superior after the 3rd data set and becomes the best algorithm on the final data set, which tells us that it is the best algorithm to use for large data sets (e.g data sets containing over 12,000 locations).

The results gathered from the second algorithm and third algorithm seem valid as they do not stray too far from the vanilla nearest neighbour algorithm which is the benchmark. They are within a reasonable scope and have not been tampered with in any way during the run of the experiment, as shown in the source code section at the end.

4 Conclusions

The experiment was conducted on the same computer with minimal background applications running, this resulted in more accurate results. The program written to test the algorithms seems reliable as it produced the same results for the route length test, which is what has been expected, and it produced time results that did not vary from each other by a huge amount. As the results show, the vanilla nearest neighbour is a very good algorithm by itself, but the other 2 algorithms do have an edge over it on larger data sets. The second algorithm is faster in all data sets but one, and it produces a smaller route length at larger data sets, however it produces larger route lengths on smaller data sets so it loses its usability if you are looking for a better route finder than nearest neighbour on small data sets. The third algorithm runs slower than nearest neighbour and second algorithm since it has to sort the data set first, however as the data sets get larger, the time taken to sort is not as significant and proves more efficient than nearest neighbour. The route length produced by the third algorithm is larger on data sets smaller than 5,000 locations but it improves on data sets larger than that.

In conclusion, as shown by this experiment, the vanilla nearest neighbour algorithm should be used on small data sets, the second algorithm should be used on medium to large data sets to be of any value and finally the third algorithm should be used only on large data sets to be more valuable than the previous 2 algorithms.

5 Appendix

This section contains the source code for the program used in this experiment.

NearestNeighbour.java

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

public class NearestNeighbour {

   //Function to get the distance between 2 locations
   public double getDistance(Point2D currentCity, Point2D possibleCity) {

      return Point2D.distance(currentCity.getX(), currentCity.getY(),
          possibleCity.getX(), possibleCity.getY());
   }

    //The algorithm itself
   public ArrayList<Point2D> nearestNeighbour(ArrayList<Point2D> cities)
       {
```

```
      ArrayList<Point2D> result = new ArrayList<Point2D>();
      Point2D closest = null;
      //Do not remove the first city in list, will cause the algorithm
          to skip the final city in result
      Point2D currentCity = cities.get(0);

      while(cities.size() > 0) {

         result.add(currentCity);

         //Remove the current city here to avoid skipping the last city
             from final result
         cities.remove(currentCity);

         double distance = Double.POSITIVE_INFINITY;

         for(Point2D city : cities) {
            if(getDistance(currentCity, city) < distance) {
               closest = city;
               distance = getDistance(currentCity, city);
            }
         }

         currentCity = closest;

      }

      return result;
   }

}
```

Limit.java

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

public class Limit {

   //Function to get the distance between 2 locations
   public double getDistance(Point2D currentCity, Point2D possibleCity) {

      return Point2D.distance(currentCity.getX(), currentCity.getY(),
          possibleCity.getX(), possibleCity.getY());
   }

    //The algorithm itself
   public ArrayList<Point2D> algorithm(ArrayList<Point2D> cities){

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

```
      Point2D closest = null;

      //Do not remove the first city in list, will cause the algorithm
          to skip the final city in result
      Point2D currentCity = cities.get(0);

      while(cities.size() > 0) {

         result.add(currentCity);

         //Remove the current city here to avoid skipping the last city
             from final result
         cities.remove(currentCity);

         double distance = Double.POSITIVE_INFINITY;

         for(Point2D city : cities) {
            if(getDistance(currentCity, city) < distance) {
               closest = city;
               distance = getDistance(currentCity, city);
               //I'm putting a limit on the distance as assumption that
                   not many distances will be smaller than 50 so I can
                   save some time by breaking the loop when first value
                   lower than 50 is encountered
               if (distance < 50){
                  break;
               }
            }
         }

         currentCity = closest;

      }

      return result;
   }
}
```

SortLimit.java

```
import java.awt.geom.Point2D;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

public class SortLimit {

   //Function to get the distance between 2 locations
   public double getDistance(Point2D currentCity, Point2D possibleCity) {
```

```
   return Point2D.distance(currentCity.getX(), currentCity.getY(),
       possibleCity.getX(), possibleCity.getY());
}

 //The algorithm itself
public ArrayList<Point2D> algorithm(ArrayList<Point2D> cities){

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


   //Sorting the arrayList from biggest to lowest in both coordinates
   Collections.sort(cities, new Comparator<Point2D>(){

      public int compare(Point2D p1, Point2D p2) {

         double a = Math.sqrt(Math.pow(p1.getX(), 2) +
             Math.pow(p1.getY(),2));

         double b = Math.sqrt(Math.pow(p2.getX(), 2) +
             Math.pow(p2.getY(),2));

         if(a > b){
            return -1;
         }
         if(a < b){
            return 1;
         }

         return 0;
      }

   });

   //Do not remove the first city in list, will cause the algorithm
       to skip the final city in result
   Point2D currentCity = cities.get(0);

   while(cities.size() > 0) {

      result.add(currentCity);

      //Remove the current city here to avoid skipping the last city
          from final result
      cities.remove(currentCity);

      double distance = Double.POSITIVE_INFINITY;

      for(Point2D city : cities) {
         if(getDistance(currentCity, city) < distance) {
```

```
            closest = city;
            distance = getDistance(currentCity, city);
            //I'm putting a limit on the distance as assumption that
                not many distances will be smaller than 50 so I can
                save some time by breaking the loop when first value
                lower than 50 is encountered
            if (distance < 50){
               break;
            }
         }
      }

      currentCity = closest;

   }

   return result;
  }
}
```

FileLoader.java

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

public class FileLoader {

   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 arraylist
            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;
  }
}
```

Main.java

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


public class Main {

   //Function that calculates the route length
   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;
   }

   //Main method used for running the algorithms
   public static void main(String[] args) {

      //Create objects to call the algorithms
      NearestNeighbour neighbour = new NearestNeighbour();
      Limit limit = new Limit();
      SortLimit sort = new SortLimit();

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

      //All the city files here, uncomment the one you want to use

      //cities =
          FileLoader.loadTSPLib("src/Travelling_Salesman/berlin52.tsp");
      //cities =
          FileLoader.loadTSPLib("src/Travelling_Salesman/d493.tsp");
      //cities =
          FileLoader.loadTSPLib("src/Travelling_Salesman/rl1889.tsp");
      //cities =
          FileLoader.loadTSPLib("src/Travelling_Salesman/rl5915.tsp");
      //cities =
          FileLoader.loadTSPLib("src/Travelling_Salesman/rl11849.tsp");

      System.out.println("Before ArrayList size: " + cities.size());
```

```
        System.out.format("Before route length: %f%n",
            routeLength(cities));


        //Store the system time before running the algorithm
        final double startTime = System.currentTimeMillis();

        //Uncomment the algorithm you want to use

        //cities = neighbour.nearestNeighbour(cities);
        //cities = limit.algorithm(cities);
        //cities = sort.algorithm(cities);

        //Store the system time after running an algorithm
        final double endTime = System.currentTimeMillis();


        System.out.println("After ArrayList size: " + cities.size());
        System.out.format("After route length: %f%n", routeLength(cities));

        //Display the time taken to complete the algorithm
        System.out.format("Time to run algorithm: %fms%n", (endTime -
            startTime));

    }

}
```