Edinburgh Napier UNIVERSITY

Coursework report

Algorithms and Data structures (SET09117)

Ivan ROGER (40285021)

I. Introduction

I.1. Abstract

This coursework was all about the Traveling Salesman problem (TSP). This problem consists of trying to find a good if not the best route to go through all cities. The number of possible solutions for a problem increases exponentially with the number of cities. And it soon becomes impossible to check each solution. So we try to find a good answer. This problem can be represented as a Hamiltonian circuit in graph.

The algorithms and the interface produced have been implemented in Java. The datasets are imported from files of this website: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp/. The interfaces are made using the native Swing library.

You can find all the code for this coursework on the github repository : https://github.com/Ivan-Roger/NAP09117_CW

II. Method

II.1. Experiments

At first I order to really understand and make sure I didn't miss anything I spent a lot of time trying to sort very small datasets (4-6) on paper. Then I implemented the pseudo-code for the nearest neighbour algorithm. Using the debugger I watched every step of the algorithm in order to get a good vision and understanding of what was done.

When implementing the other algorithms I again spent some time on paper to make a described plan of each step. When I encountered error it was also very helpful to reproduce the code on paper to see where was the mistake.

At first I wanted to write an algorithm of my own but couldn't figure out where to start.

So I decided to look for possible improvements that could be done to existing algorithms.

To improve the Nearest Neighbour algorithm I realised that sometime the closest node was really far so I had an idea of improving the result by trying to split the graph into several smaller clusters of nodes where I would run a Nearest Neighbour algorithm and then I would put every cluster together in a single circuit.

Unfortunately I didn't find out how to separate the dataset into multiple clusters. I wasn't able to find a condition defining when to consider that a point was out of the applicable area.

Another problem I decided to try solving was the fact that often on the default dataset a lot of vertices are crossing each others. So I thought that by solving this problem I would find a possible solution. I ended up with the Intersection Solving algorithm described later.

When I analysed the way I conceived my intersection solving algorithm was really similar to a Two Opt algorithm so I did implement this one.

I spent a lot of time on trying to find ways to make the Two Opt algorithm quicker. I spotted the fact that at every test we are calculating the length of the entire route which takes a lot of time. So I have been looking for an alternative and thought that by calculating only the distance of what was altered it would be quicker. Unfortunately I didn't manage to find a good enough solution and only found quicker ways to realise the Two Opt algorithm but in an infinite loop. It could have proven useful because by stopping it any time we would still get an improved result but I decided to go back on the default Two Opt algorithm because it was ensuring a finite running time.

II.2. Dataset selection

I selected multiple data sets for my tests. I started with a small one to be able to follow easily what the program was doing. So that if I did an error in the code it was easier to see it.

My first dataset was composed of 48 cities (cf. Illustration 1).

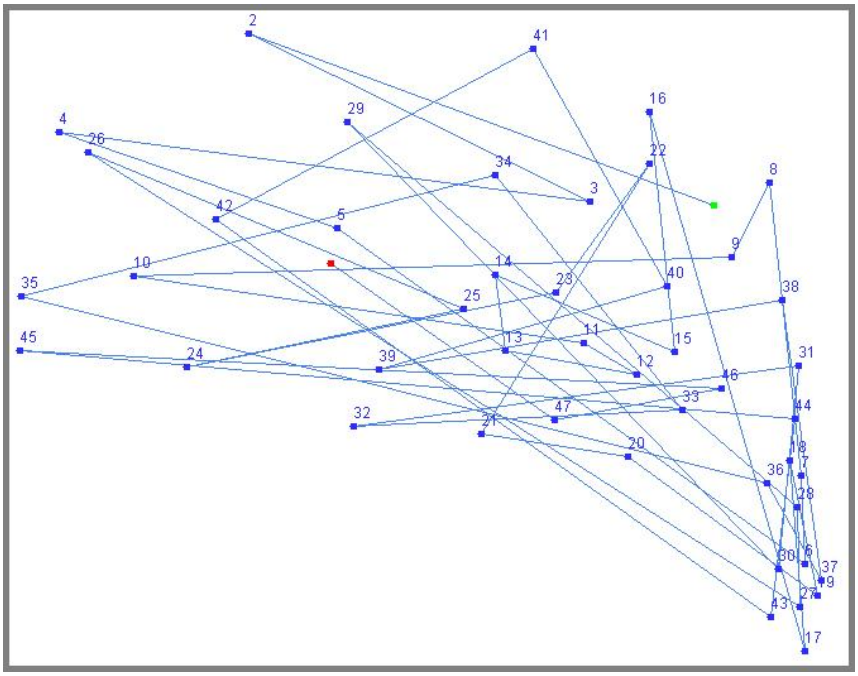

Illustration 1: Small dataset (att48)

I also used bigger dataset afterwards to test the speed of my algorithms. To do so I used problems with gradually increasing number of cities, I began with 120 and 152, and finally worked with 4461 cities.

For some algorithms the way cities are placed can also influence on the result. I have remarked this with the file *a280.tsp* where points are all aligned or with the file *fnl4461.tsp* where the initial circuit goes in vertical vertices only.

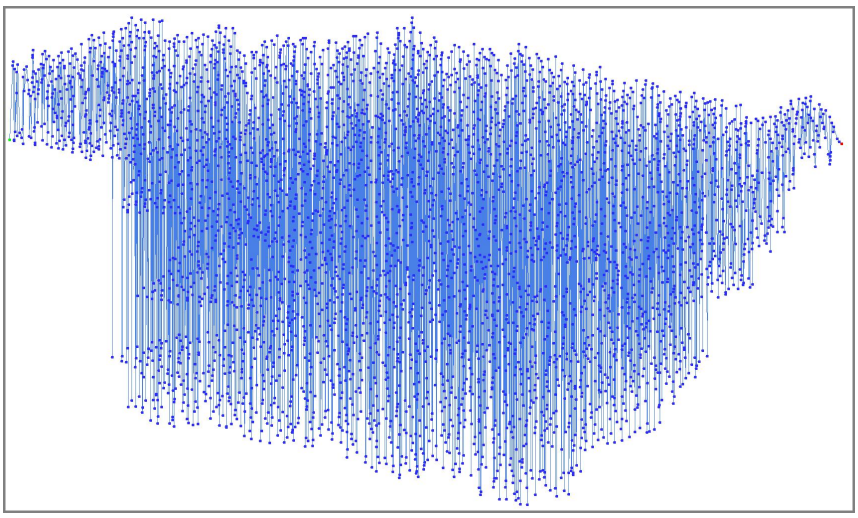

Illustration 2: Raw dataset fnl4461

III. Results

I implemented and worked on three types of algorithms.

III.1. Nearest Neighbour

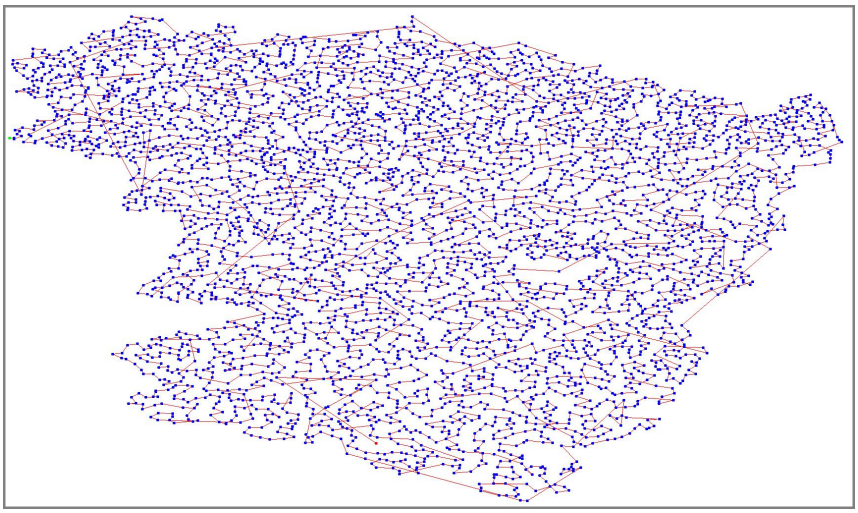

Illustration 3: Nearest Neighbour on fnl4461

The first one is the nearest neighbour. The pseudo-code was already given so I had to implement it in Java.

This algorithm starts on a node and looks through all the other nodes which one is the closest. When the closest is found. Start again from this node. The list of node searched contains all the nodes that haven't yet been selected.

The nearest neighbour returns a really good a solution and is relatively quick.

This solution is valid because all nodes are visited and the whole forms a Hamiltonian circuit.

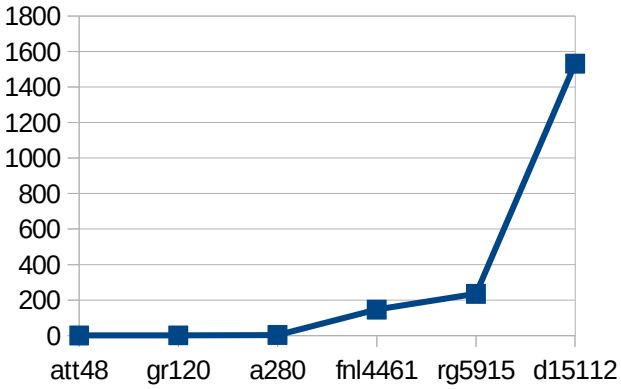

Illustration 4: Time to complete

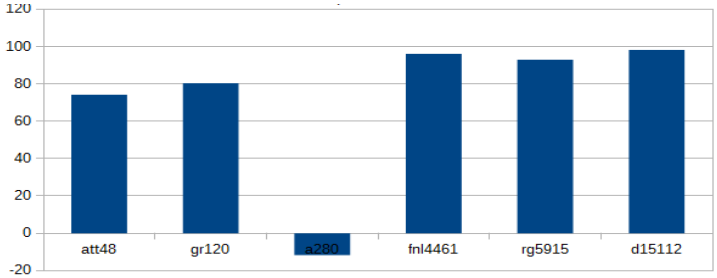

Illustration 5: Improvement in percent

As we can see on the fourth illustration the cost is of the shape of a square.

As for the improvement it's almost always between 70% and 100% except for the dataset *a280* where the result is worse than the original route.

This algorithm can be really useful because it has a predicted running time. Given the size of the dataset we know it will always take the same time to solve the problem. And will always produce the exact same output. The result does not depend on the route of the input dataset.

III.2. Intersection solving

The second algorithm I used tests if each vertex of the circuit crosses another one. If it does is solves it by swapping the starting node of the two vertices. Unfortunately depending on the shape of the circuit this algorithm can end up in an infinite loop.

The solution given is not really good and but does what it meant to do. There are no crossings.

The time taken to run is highly dependant of the initial route.

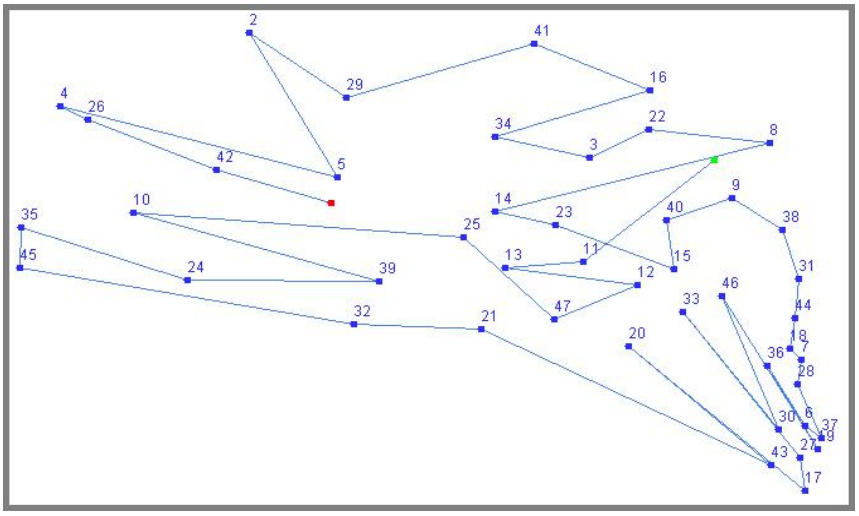

Illustration 6: Solve Intersect on att48

Dataset	Time taken (in milliseconds)
att48	9399.71
berlin52	186.41
gr120	3071.86
pr152	3.04
a280	24615.99

As we can see the problem with 48 cities took nine seconds to run and the problem with 152 cities only took three milliseconds. This algorithm can be useful after a Nearest Neighbour and before a Two Opt.

The result is a valid solution because it always returns a route going through all cities once and can form a Hamiltonian circuit.

In the worst case scenario this algorithm can take an infinite amount of time. Some datasets do not have a solution without crossing.

On another hand this algorithm can be stopped at any time and will still return a valid result.

Most of the time the behaviour of this algorithm is as follows: first the result will seem to only go worse as it was but in the end all the lines will form in a sort of a star with all vertices going in a shape similar to a circle.

III.3. Two Opt

The last solution I implemented is the Two Opt algorithm.

This one takes a vertex and swaps it's starting node with the one of another vertex. If the length of the new route is shorter then it keeps it, else it discards the swap and tries with next vertex. This algorithm keep running until no improvements can be found.

This algorithm produces quite good results but is long to run. The time it takes is dependant on the size of the dataset but also relies on the input route.

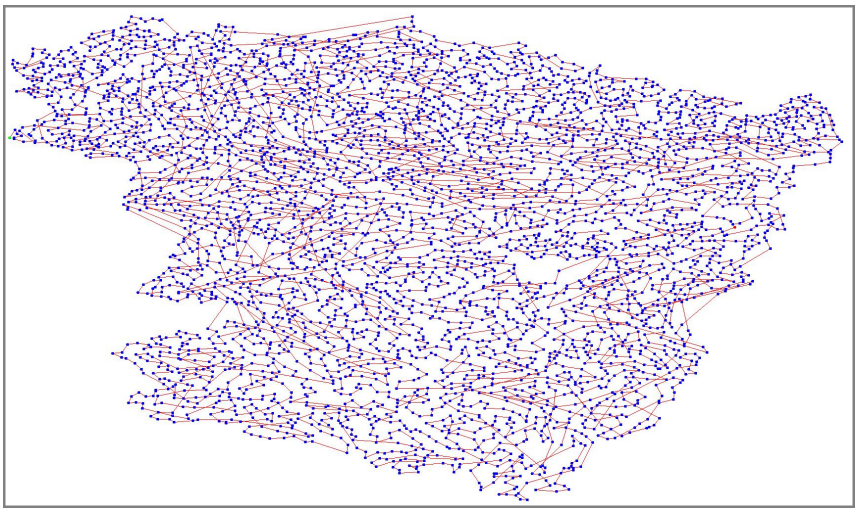

Illustration 7: Two Opt after Nearest Neighbour on fnl4461

This algorithm will take a long time to run on a raw dataset but will prove powerful after a nearest neighbour. This algorithm is a good solution to improve a result given by another.

If no improvements can be made the running time will be really small.

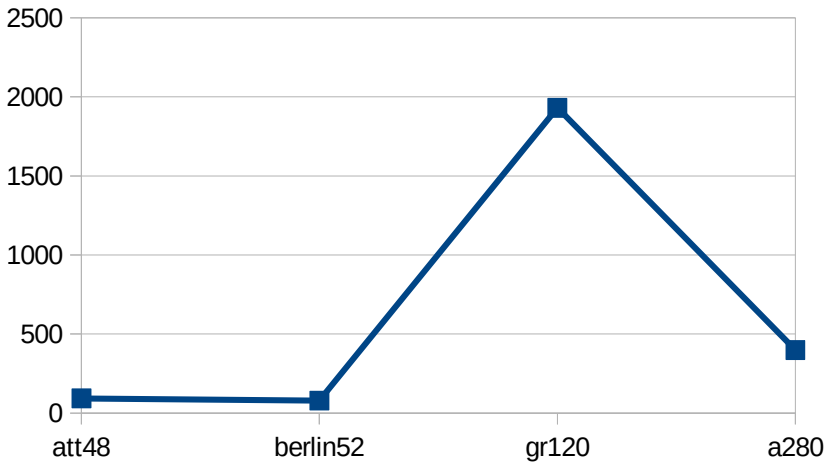

Illustration 8: Time (in ms)

The chart doesn’t look like anything because the dataset influence the run time.

In the worst case scenario this algorithm can take a factorial shape.

IV. Conclusions

IV.1. Summary

In this coursework I worked on how to implement a standard algorithm from it’s pseudo-code version into a real language, how to improve an existing algorithm by determining what are the down parts of the results returned.

I also learned that most of the improvements ideas are way more complicated to put together than we could expect. Sometimes we don’t think of all the aspects of the problem but when we try to implement it we quickly realize that we are missing some knowledge or techniques.

IV.2. Performance analysis

I am, in a general overview, quite happy with the results of this coursework. I think I managed my time pretty well. I probably should have spent less time on the graphical interface but it has proven really useful when trying to figure out why some algorithm didn’t produce the expected result and helped me find out what was going wrong.

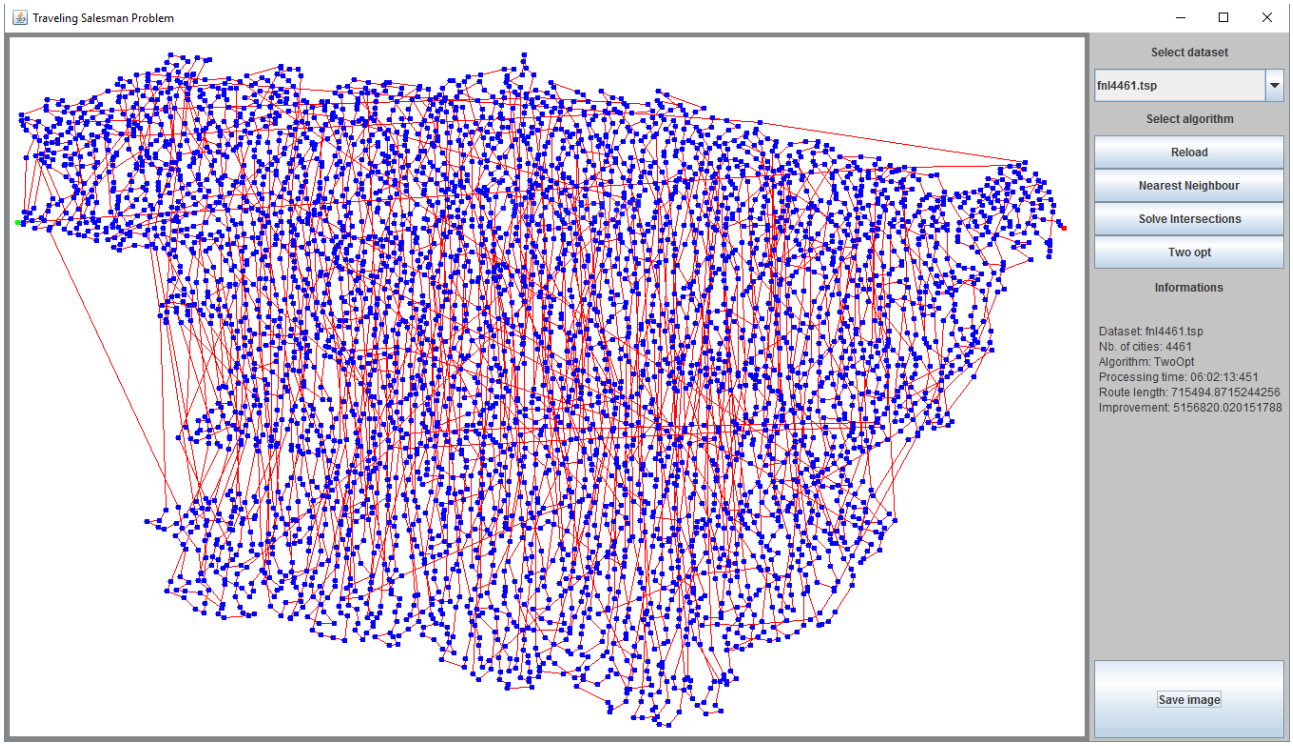

Illustration 9: Graphical Interface

I have acquired a good knowledge of the notion of time bound treatment systems. I am now more conscious of the fact that some problem may not be solved in an acceptable time and this with even smaller datasets that expected.

I spent a good amount of time running some algorithms and sometimes for more than 15 hours to finally decide to stop them because I figured out it wasn't going to end soon enough.

I have gained interest on how to analyse and compute the operations that take a lot of time to run and how to try avoiding them.

I am now convinced that code optimisation is a really large and complex field of study.

V. References

Datasets: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp/

General documentation:

- https://en.wikipedia.org/wiki/Travelling_salesman_problem
- https://plus.maths.org/content/travelling-salesman
- http://www.math.uwaterloo.ca/tsp/

Two Opt: https://en.wikipedia.org/wiki/2-opt

Java Swing documentation:

Stack overflow: http://stackoverflow.com/

VI. Appendix

VI.1. Source: Nearest Neighbour

```
ArrayList<City> res = new ArrayList<>();

City currentCity = cities.get(0);
cities.remove(0);
res.add(currentCity);

while (!cities.isEmpty()) {
        // Because we are forced to give an initial value
        double distance = Float.MAX_VALUE;
        City closest = currentCity;
        for (City p : cities) {
                if (currentCity.distance(p) < distance) {
                        closest = p;
                        distance = currentCity.distance(p);
                }
        }
        cities.remove(closest);
        res.add(closest);
        currentCity = closest;
}

return res;
```

VI.2. Source: Solve Intersections

```
ArrayList<City> res = (ArrayList<City>) cities.clone();

boolean intersectFound;
do {
        intersectFound = false;
        // Loop through all cities (except the first)
        for (int i=1; i<res.size()-1; i++) {
                City startA = res.get(i);
                City endA = res.get(i+1);
                // Create a line to represent the edge
                Line2D lineA = new Line2D.Float(startA, endA);

                // Loop through all cities
                for (int j=1; j<res.size()-1; j++) {
                        // if it's the previous, same, or next skip.
                        if (Math.abs(i-j)<2) continue;

                        City startB = res.get(j);
                        City endB = res.get(j+1);
                        // Create a line to represent the edge
                        Line2D lineB = new Line2D.Float(startB, endB);

                        // If the two edge intersects
                        if (lineA.intersectsLine(lineB)) {
                                // Update the loop flag
                                intersectFound = true;

                                // Swap the nodes
                                res = TwoOpt.swapOpt(res, i, j);

                                // Update data for the next loop
                                startA = res.get(i);
```

```
                                endA = res.get(i+1);
                                lineA = new Line2D.Float(startA, endA);
                        }
                }
        }
} while (intersectFound);

return res;
```

VI.3. Source: Two Opt

```
ArrayList<City> res = (ArrayList<City>) cities.clone();

// Variable to store the best route length found.
double best_dist = TSPLib.routeLength(res);
// Flag to continue looping when we made an improvement
boolean improvementMade;
do {
        improvementMade = false;
        // Loop through each cities
        for (int i=1; i<res.size()-1; i++) {
                // Loop through cities following the current one
                for (int j=i+1; j<res.size()-1; j++) {
                        // Get the new route and it's length
                        ArrayList<City> new_route = swapOpt(res, i, j);
                        double new_dist = TSPLib.routeLength(new_route);

                        // Test if it's an improvement
                        if (new_dist<best_dist) {
                                // Then keep it, update variables
                                res = new_route;
                                best_dist = new_dist;
                                improvementMade = true;
                        }
                }
        }
} while (improvementMade);

return res;
```

VI.4. Function: TwoOpt.swapOpt(in, i, j)

```
ArrayList<City> out = (ArrayList<City>) in.clone();

City tmp = out.get(i);
out.set(i, out.get(j));
out.set(j, tmp);

return out;
```