Algorithm Engineering
WS 22/23

Jakob Sanowski
Fiona Riesterer

Report Assignment 04

Abstract

We implement and compare two algorithms, namely, the Nearest Neighbor heuristic and the 2-OPT heuristic to approximate the Traveling Salesman Problem. We first explain the algorithms and then describe how we implemented them. We tested the algorithms on graphs that represent countries and their cities. The results show that combining Nearest Neighbor and 2-OPT give the shortest tours.

1 Introduction

The Traveling Salesman Problem is well-known question. To find an optimal tour is not easy so it is suitable to use approximation algorithms like Nearest Neighbor and 2-OPT. We compared these two algorithms.

The remainder of this work structured as follows. In Section 2 we explain the problem and the two algorithms. We then explain our implementations in Section 3. The experiments are described in Section 4 and are then discussed and interpreted in 5.

2 Preliminaries

Given a complete Graph $G = (V, E)$, we try to find a tour, that visits each vertex exactly once and has a minimal length. The problem is known as the Traveling Salesman Problem. Since it isn't easy to find the optimal tour, there are heuristic that try to come as close to the optimum as possible. One is the Nearest Neighbor Heuristic (see 2.1 and the other is the 2-OPT Heuristic (see 2.2).

2.1 Nearest Neighbor Heuristic

Given a starting vertex v, the algorithm finds the nearest vertex u and connects them with an edge. It then proceeds to find the nearest neighbor of u out of the remaining vertices, adds an edge and so on. If there are no vertices left, then an edge between the last and the first vertex is added. The resulting tour length is the sum of all edge lengths.

2.2 2-OPT Heuristic

Given any tour T of a graph $G = (V, E)$. Find two edges, that, if swapped, yield to a better tour T' than T. Find those two edges, that would improve the tour the most and swap them. Do this until nothing can be improved anymore.

If you swap two edges $e_1 = (u, v)$ and $e_2 = (w, x)$, you would get two new edges $e_1\prime = (u, w), e_2\prime = (v, x)$. So you also have to change the direction of the path between v and w.

3 Algorithm & Implementation

This section provides information about the actually used algorithms and their respective implementations.

3.1 Implementation Details

We used Java 18 to implement the algorithms. As distance measure between two points a, b that have x- and y-coordinates, we used the euclidean distance which is defined as follows: $\sqrt{(a_x - b_x)^2 + (a_y - b_y)^2}$

3.1.1 Nearest Neighbor Heuristic

In order to get a complexity of $\mathcal{O}(1)$ for removing objects we use a HashMap for storing the nodes. Starting with a randomly selected node we first remove the selected node and then linearly search for the closest node in the remaining nodes and add an edge from our selected node to the found node. The fond node becomes the new selected node. This process is repeated until there is no node left. We then simply add an edge from the last found node to our starting node to complete the round tour. 1 shows the pseudo-code for this algorithm. This has a complexity of $\mathcal{O}(n^2)$.

Algorithm 1: Nearest Neighbor Heuristic

```
Input: set of nodes $V$
Output: tour $T$
nearestNeighborHeuristic($V$) begin
    $start, v \leftarrow v \in V$
    while $V$ not empty do
        $v\prime \leftarrow$ nearestNeighbor($v, V$)
        $T \leftarrow T \cap \{v, v\prime\}$
        $v \leftarrow v\prime$
    $T \leftarrow T \cap \{v, start\}$
    return $T$
nearestNeighbour($v$,$V$) begin
    remove($v, V$)
    for every $v\prime \in V$ do
        if $v\prime$ closer to $v$ than $min$ then
            $min \leftarrow v$
    return $min$
```

In order to reduce the number of nodes we need to compare with we tried to use a grid structure for storing the nodes. The idea was to divide the space over which the nodes are distributed into smaller equally sized subspaces in order to group clusters of nodes together. To find the nearest neighbor of a node n we first search in the subspace the node is located in and then move outwards searching the surrounding subspaces in a greedy fashion. The node found with the greedy search is then used to limit the search space to all subspaces which could contain a node closer to n than the found one. We then search for the closest node to n in this limited space. This approach though has the major drawback that the complexity for finding the closest neighbour is dependent on the distance to the closest neighbour because we need to verify every subspace possibly containing a closer node is empty. Unfortunately our implementation does not always find the closest neighbour due to a bug we could not identify. We still included it in our experiments because it only happened in very rare cases and if could still showcase possible run time improvements.

3.1.2 2-OPT Heuistic

We used an Arraylist to store our edges. In order to find edges that can be swapped, we simply compared each edges individually. We then swapped the first two edges that would improve the tour, even if it might not be the best improvement. After swapping the edges, we have to change the direction of the path in between those edges. We always change the direction of the path that is shortest. Depending on which path you use, for example if you want to swap two edges $e_1 = (u, v), e_2 = (w, x)$ and $|p_1| = | < v, \dots, w > | \leq | < x, \cdots, u > | = |p_2|$, we change the direction of p_1 which results in the swapped edges $e_1\prime = (u, w), e_2\prime = (v, x)$. If p_2 was larger, then the resulting swapped edges would be $e_1\prime = (w, u), e_2\prime = (x, v)$. It is important to note, that this does not change the distance of the two edges. The pseudocode for the 2-OPT heuristic is shown here 2.
To get a better understanding on how the algorithm works, you can also take a look at the figures 5 to 17 where the initial and resulting tours are drawn.

Algorithm 2: 2-OPT Heuristic

```
Input: any tour $T$
Output: improved tour $T\prime$
twoOpt(Edgelist $T$) begin
    while there are improvements possible do
        for every edge $e \in T$ do
            for every edge $v \in T; v \neq e$ do
                if swap $v, e$ yields to shorter
                  tour then
                    swap($v, e$)
```

Swapping two edges and changing the direction of an edge has a constant running time, e.g. is in $O(1)$,

because you only have to change the two vertices in each edge and maybe swap their positions. Since we are using an Arrayllist, this can be done in $O(1)$.
Changing the path direction between the two swapped edges, can't be done in constant time. In our implementation we could limit the number of edges in the path between the two swapped edges to at most $\frac{m}{2}$, since we always swap the edges, s.t. the shortest path has to change the direction. Changing the direction of the path then has running time $O(\frac{m}{4})$, because we always start from the head and tail of the path, change the direction of these edges and then swap their position, doing at most $\frac{m}{4}$ swapping operations.

Searching for an edge pair that can be swapped has a running time in $O(m^2)$ (where m is the number of edges) since we have to compare each of the edges. In the worst case we have to perform a swapping operation for each edge pair, which would result in a running time of $m^2 \cdot \frac{m}{4} \in O(m^3)$. We think that there can be found a better bound for the running time of the edgeswap steps, for example by limiting the number of total edgeswaps that will be performed depending on the number of edges, but we couldn't come up with a suitable solution.

4 Experimental Evaluation

In this section, the experimental setup is described and the results are presented.

4.1 Data and Hardware

The running times of the 2-OPT heuristic and the combination of the 2-OPT heuristic and the Nearest Neighbor heuristic were measured on an AMD Ryzen 7 with a base clock of 1.8 GHz and a boost clock of up to 4.3 GHz. The running times for the Nearest Neighbor heuristic were measured on an Intel i7 6700K with a clock speed of 4.2 GHz. Both algorithm were tested on all 25 inputs found at https://www.math.uwaterloo.ca/tsp/world/countries.html

4.2 Results

We ran three experiments.

First was measuring the running times of the two different algorithms. As you can obtain from figure 3 and figure 2 the running times of 2-OPT are slower than for Nearest Neighbor. Also, using the Nearest Neighbor tour as initial tour for 2-OPT did not significantly lead to a better running time of the algorithm (fig. 2).

Figure 3 shows the results of the nearest neighbour algorithms. It can be seen that both algorithms produce very similar results in the resulting tour length though the grid-algorithms is slightly longer than the quadratic-algorithm on most instances. On most instances the quadratic-algorithm was significantly faster, though on the larges instances the grid-algorithm performed better. Interestingly the grid-algorithms running time seems to correlate with the length of the resulting tour.

Second, we compared the number of edge swapping operations and the running time when starting 2-OPT from an arbitrary tour and starting 2-OPT on the tour found by Nearest Neighbor.

Third was comparing the resulting tour lengths to the Bound found at https://www.math.uwaterloo.ca/tsp/world/summary.html and computing the ratio. The results are shown in figure 1.We also provided pictures of three of the tours that show the different tours when starting 2-OPT from an arbitrary tour and starting with the tour found by Nearest Neighbour (figures 5 to 17).

5 Discussion and Conclusion

Comparing the two nearest neighbour algorithms we could observe a correlation between the running time and the resulting tour lenght for the grid-algorihtm. This is due to the complexity of finding the nearest neighbour is dependent on the distance to it. Instance 'wi29' in figure 3 shows this very well because it has very few nodes which are far appart from each other, resulting in a very long run time for the grid-algorithm.

For the running times we can obtain that 2-OPT is

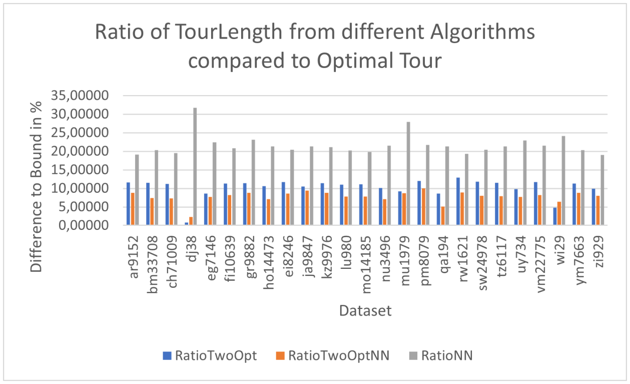

Figure 1: Ratio of resulting Tourlength when using 2-OPT, Nearest Neighbor or both combined

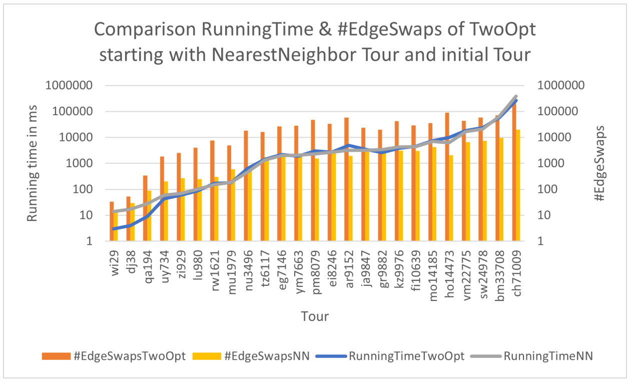

Figure 2: Running time of 2-Opt Heuristic & number of edge swaps with different starting tours

slower than Nearest Neighbor but is getting way better results. The difference in running time is due to the expensive edge swap operations that 2-OPT does to get a better tour. The results are even improving when starting 2-OPT from the tour found by Nearest Neighbor. But even if the number of edge swaps is less than for an arbitrary tour, there is not much improvement on the running time. That's because the time needed to find a better starting tour (here Nearest Neighbor) is comparable to the time needed to perform the extra edge swaps. Since it leads to better results and has nearly the same running time, computing a better starting tour is definitely better than starting from an arbitrary tour. And even if Nearest Neighbor is much faster than 2-OPT, 2-OPT should be chosen over Nearest Neighbor since it computes way better results.

To sum up, we can say that 2-OPT produces way better results than Nearest Neighbor, but has worse running time. And since the combination of both algorithms did not increase the overall running time, this is the best solution to find a short tour in a reasonable time span.

6 References

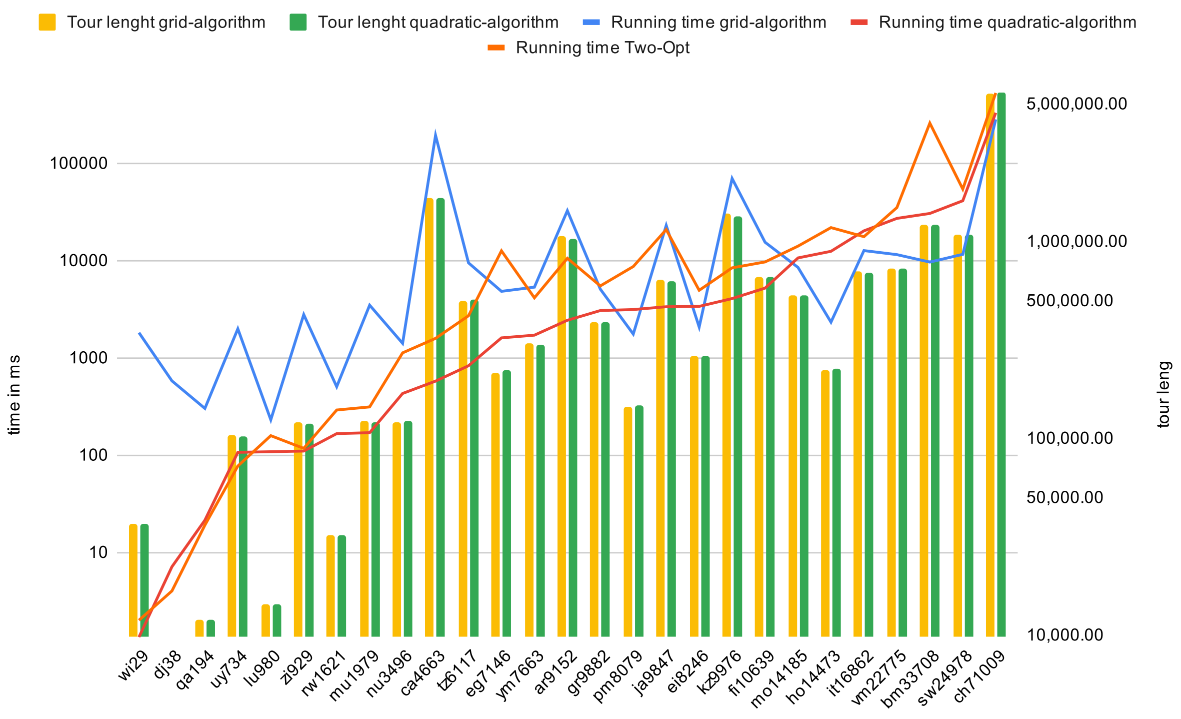

Figure 3: Nearest Neighbor running time and resulting tour length

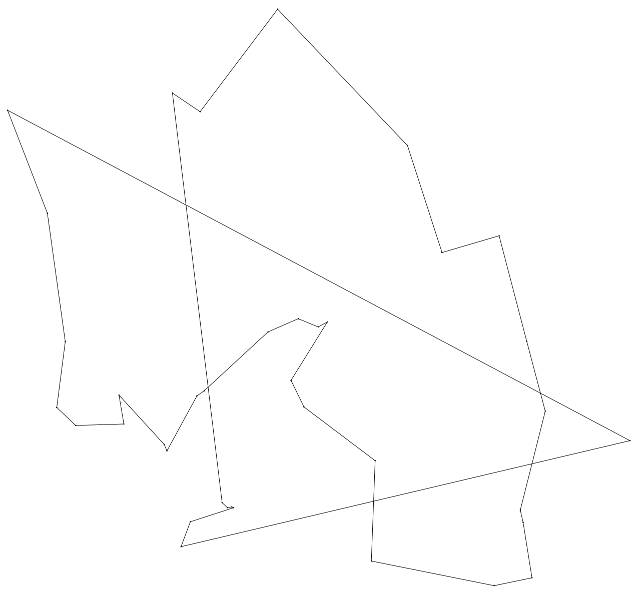

Figure 4: Nearest Neighbor Tour Djibouti

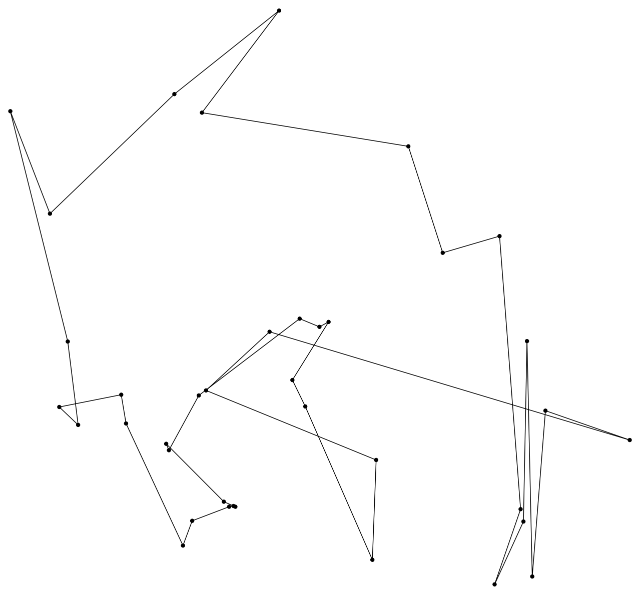

Figure 6: Half Tour Djibouti

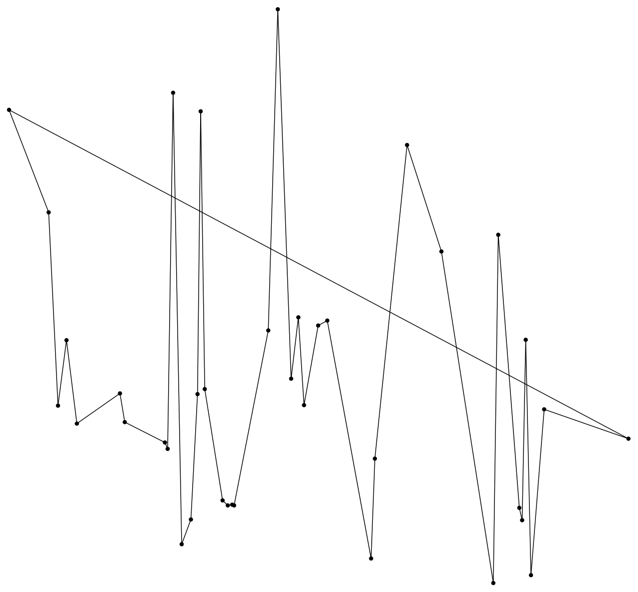

Figure 5: Initial Tour Djibouti

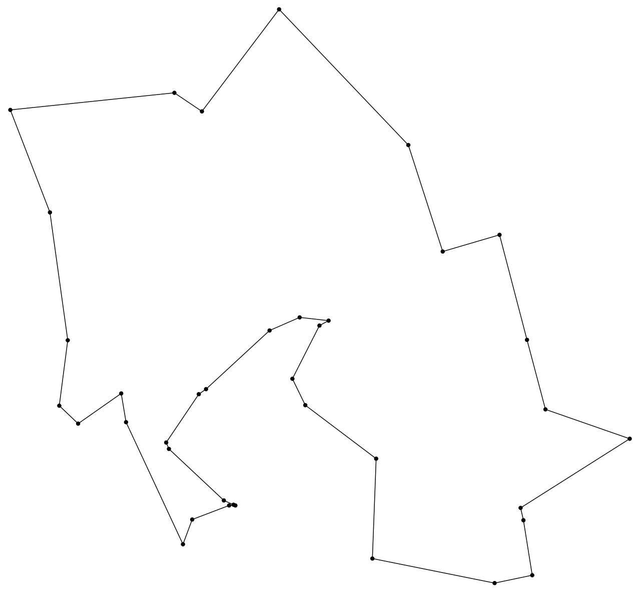

Figure 7: Resulting Tour Djibouti

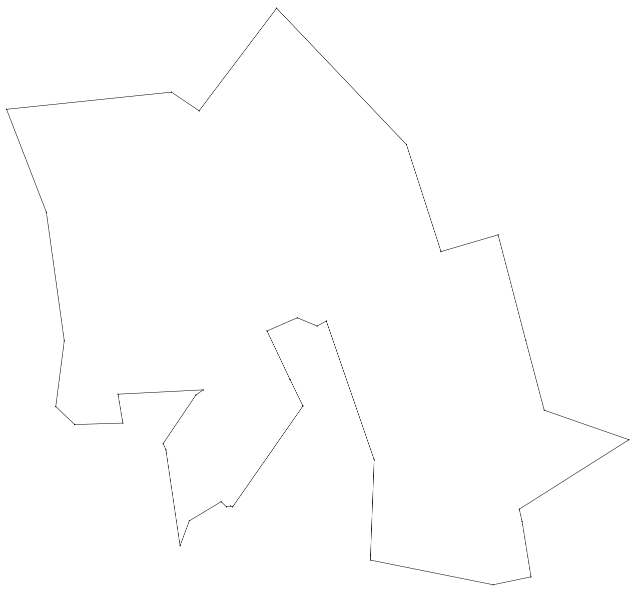
Figure 8: Resulting Tour with Nearest Neighbour Djibouti

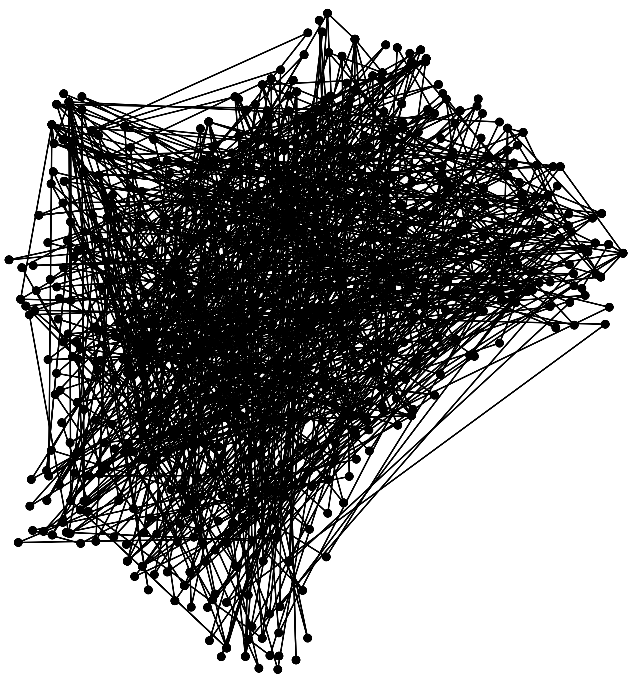
Figure 10: Initial Tour Luxembourg

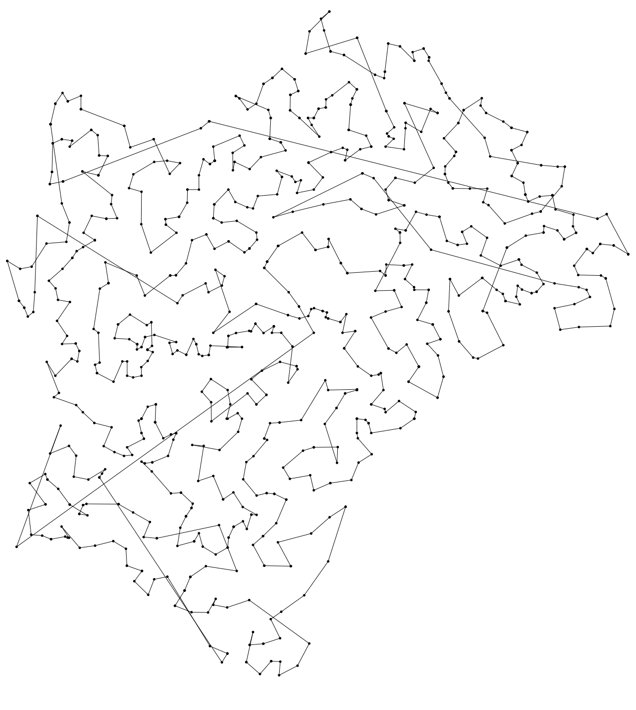
Figure 9: Nearest Neighbor Tour Luxembourg

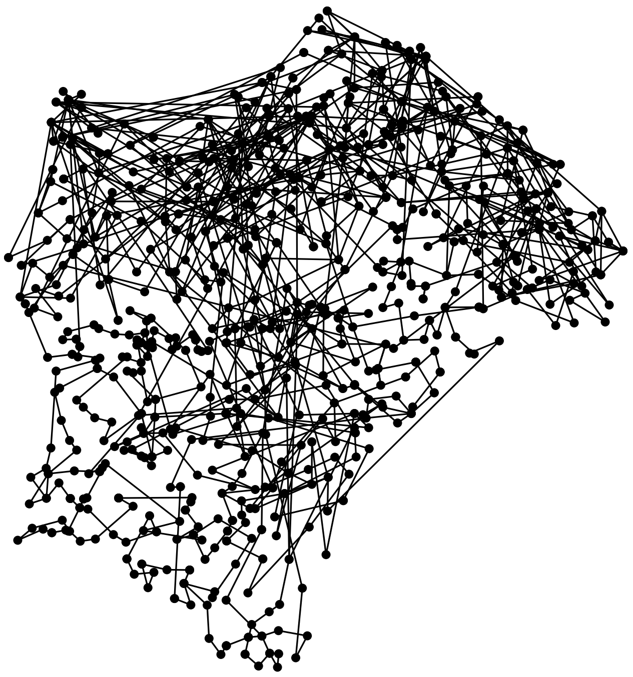
Figure 11: Half Tour Luxembourg

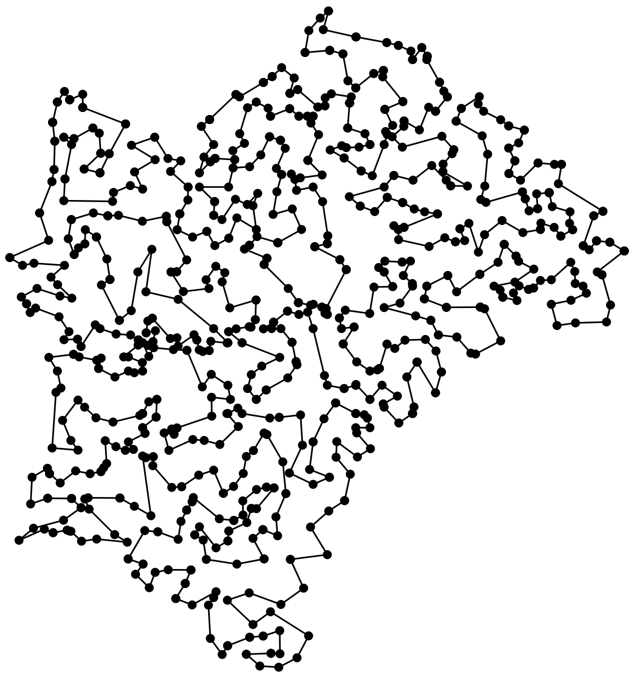
Figure 12: Resulting Tour Luxembourg

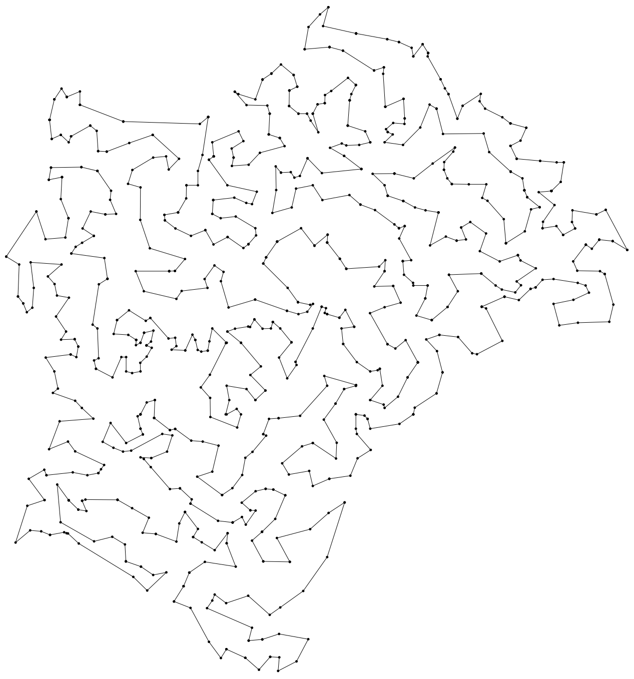

Figure 13: Resulting Tour with Nearest Neighbour Luxembourg

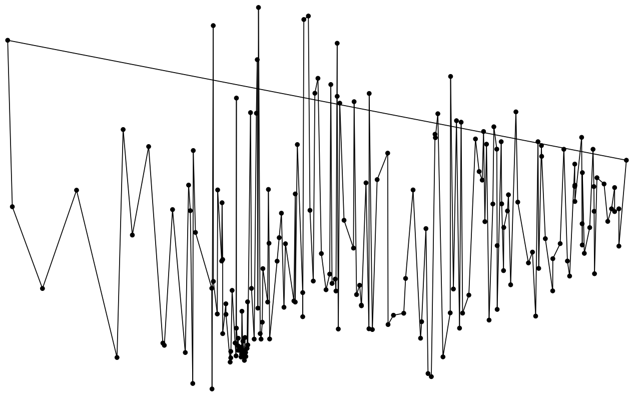

Figure 15: Initial Tour Qatar

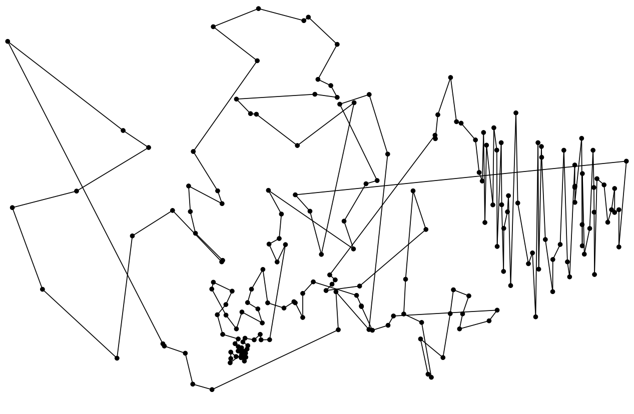

Figure 16: Half Tour Qatar

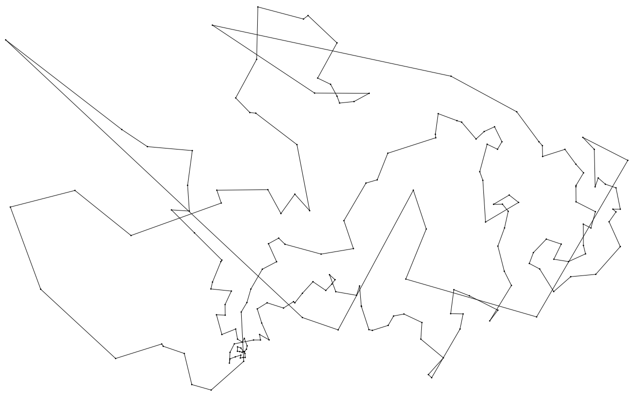

Figure 14: Nearest Neighbor Tour Qatar

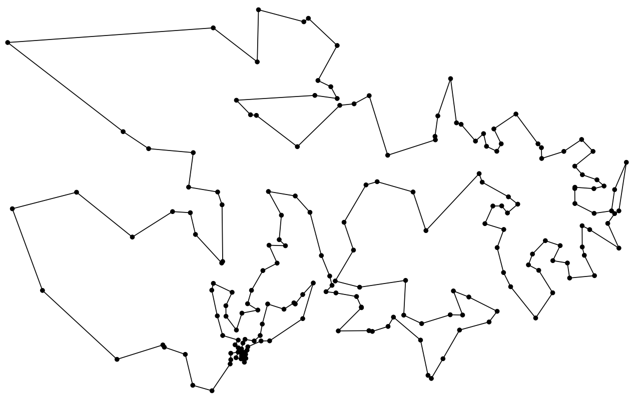

Figure 17: Resulting Tour Qatar

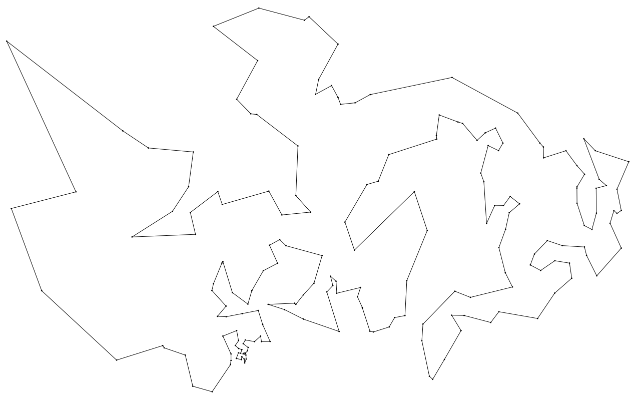

Figure 18: Resulting Tour with Nearest Neighbour Qatar