Group 2
Travelling Salesperson 2D

Niv Adam, David Kaufmann, Casper Kristiansson, Nicole Wijkman

November 23, 2023

Kattis Submission: 12262948 (Score 30.03, Aiming for C)

1 Algorithms

In this section, we outline the different algorithmic approaches we tested. In general, we first used start heuristics to find a solid initial tour and then ran local optimization until the two seconds were over. We implemented various heuristics and local optimization techniques. We found that the Christofidis heuristic together with both 2-Opt and 3-Opt local optimization yields the best results. For many of these algorithms, it is helpful to interpret the problem as a graph. Each point corresponds to a vertex and between each pair of vertices, there is an edge with the distance between the points as its weight.

1.1 Start Heuristics

This section outlines the different start heuristics we implemented to find an initial tour.

1.1.1 Nearest Neighbor

The simplest heuristic we implemented is the nearest neighbour heuristic. It was outlined in the assignment on Kattis and is used as a comparison against student solutions. The idea is to start at a random point and select the closest other point as the next point on the tour. From that point on the nearest neighbor of the current point that is not yet part of the solution is selected as the next point.

1.1.2 Greedy

The greedy heuristic also follows a straightforward idea. It is often mentioned as a simple but effective heuristic [1]. The algorithm consistently incorporates the shortest edge that connects two previously unconnected components of the graph into the tour, ensuring that it does not increase any vertex's degree beyond two. This process is continued until all vertices belong to one connected component. To check whether an edge connects to distinct components, we check whether adding the edge creates a circle in the tour.

1.1.3 Christofidis

The heuristic proposed by Christofidis [2] is the most complex one we implemented. It starts by calculating a minimal spanning tree on the graph. To achieve this, we used the Kruskals algorithm [3]. It is very similar to the greedy heuristic explained in Section 1.1.2. The only difference is that it allows vertices to have a degree higher than two.

From this spanning tree which has minimal weight, all vertices with odd degrees are selected. We calculate a minimal weight perfect matching on these vertices. This is a set of edges such that every vertex has a connection to exactly one edge of the set. These edges are selected in a way that the weight of the entire matching is minimal. To calculate this matching, there is the blossom algorithm. But we did not find a fast implementation of that algorithm, therefore we used a package that implements an efficient calculation of a maximum weight matching. To make use of this, we transformed the weights of our edges. We calculated the new weights w_i' as

$$w_i' = \max w - w_i$$

This keeps all edges positive and keeps the same order. The only difference is that the previously longest edge is now the shortest.

Afterward, the edges of the matching are added to the minimal spanning tree. Since the matching adds one additional edge to each vertex with an odd degree in the resulting graph all vertices have an even degree. For such graphs where each vertex has an even degree, it is known that an Euler tour exists. An Euler tour visits each edge exactly once and returns to the start point. This tour can be used to calculate the resulting TSP tour. We simply calculate the Euler tour and discard every second appearance of vertices. This gives a tour containing each vertex exactly once. There are proven results that such a tour is not worse than 1.5 times the optimal tour [1].

1.2 Local Optimization

This section describes the different optimization methods we tried out. All of these are based on a tour that has previously been generated by one of the start heuristics outlined in 1.1.

1.2.1 2-Opt

We started with 2-Opt which is a simple, yet effective optimization method [1]. As long as computing time is still available, we iteratively choose two nodes in the tour and perform a so-called "swap". A swap reverses the sub-tour between the two chosen nodes and replaces the new tour with the old tour if an improvement can be seen. Under the condition that no inversion of any sub-tour results in an improvement, the procedure is stopped even before reaching the maximum computing time.

1.2.2 3-Opt

3-Opt is a further development of 2-Opt. We started by reading through its description in [1]. In our implementation, we iteratively choose any 3 edges, delete and reconnect them. Figure 1 nicely illustrates the different possibilities to reconnect. Similar to our 2-Opt approach, we only update the tour accordingly, if an improvement is measurable, i.e. a shorter tour results. We apply the same checks for computing time and improvement as described in the 2-Opt section. We used this approach to reach our maximum score.

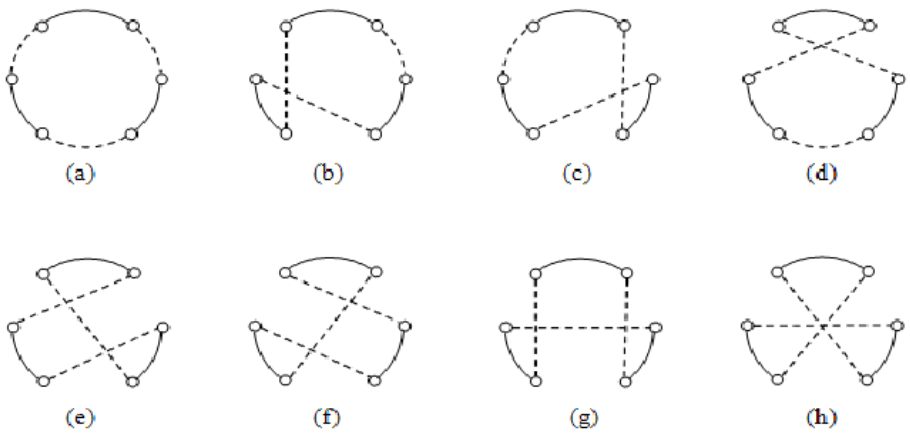

Figure 1: All possible 3-Opt reconnection cases [4]

1.2.3 Lin Kernighan

While exploring various algorithms for the Traveling Salesman Problem (TSP), we implemented and tested the Lin-Kernighan algorithm. To gain a deeper understanding of the algorithm, we studied the paper "An Effective Heuristic Algorithm for the Traveling-Salesman Problem" by S. Lin and B. W. Kernighan [5]. The paper presented the original Lin-Kernighan algorithm and how effective it is in generating near-optimal solutions for the TSP. We then attempted to implement and refine the Lin-Kernighan algorithm for the context of this particular project.

However, during our attempts, we encountered challenges in transitioning from 2-opt to 3-opt, as well as higher-order swaps within the Lin-Kernighan framework. The transition significantly increased the computation time. Despite our efforts to fine-tune the implementation to boost the overall performance, we could not get a score above 25 with this method. Due to the time limit of the first project deadline, we chose to not make further attempts using the Lin-Kernighan algorithm.

1.2.4 Simulated Annealing

Simulated Annealing is a local optimization technique based on 2-Opt. Other than in classical 2-Opt, in simulated annealing, moves that make the tour worse are allowed under certain circumstances. The rationale behind this is that it is very likely for classic 2-Opt to find a local optimum that is much worse than the actual global optimum. By allowing moves that make the tour worse, it is possible to escape such local optima. The Lin-Kernighan algorithm leverages a similar idea.

At the core of the simulated annealing approach is a temperature variable T. This variable determines the probability that a 2-Opt move making the tour worse is accepted. During the execution, the temperature is gradually lowered until it is very unlikely to accept moves that make the tour worse.

In our implementation, we start with a nearest neighbour tour and set the initial temperature to the length of this tour divided by the number of points (n). We then perform $100 \cdot n$ many 2-Opt moves per temperature and lower the temperature 50 times, by multiplying it with 0.95. For every 2-Opt move, we check if it is better than the current tour. If it is, we change the current tour to the result of the 2-Opt move. Otherwise, we calculate the difference between the current and new tour and change the current tour if a number randomly d

1.3 Firefly

The issue with the local optimization algorithms is that they focus on improving the solution within a small local region. Because of this they often converge quickly to a local optimum but have a hard time trying to escape it. The firefly algorithm [6] uses the behaviour of fireflies to find the best path in a TSP problem. Note, that this builds on the same idea as Simulated Annealing 1.2.4 and the Lin-Kernighan 1.2.3 Algorithm. One of the big benefits of the Firefly algorithm is that it's not a local optimization algorithm but a global optimization algorithm. The basics of the algorithm is that for a given problem, we assign a number of fireflies. Each firefly is attracted to other fireflies based on their brightness value. This means that the brighter a firefly is, the more fireflies will be attracted to it. Doing this allows for a solution to evolve towards a more optimal route.

The algorithm starts off by generating a set of random routes (number of fireflies). We then assign a brightness for each firefly, where the shorter path has a higher brightness. Because fireflies are attracted to the brighter ones, we can move a firefly (adjusting the path) by swapping cities in the route, adjusting the order, or performing other optimizations.

In order for this algorithm not to become a local optimization algorithm, the algorithm involves a lot of randomness. For example, when adjusting a route, the paths adjusted are based on a

random number generated. Even in some cases where the new route might have a worse distance, the algorithm can still choose that path in order to explore more possibilities. Because the fireflies will constantly go towards the best solution, an existing statement was defined, which in this case is simply when we reach just under 2 seconds of execution time.

When implementing this solution, we found that it didn't get the best score right away. Therefore, after declaring the different randomized initial routes, we applied 2-opt 1.2.1 for each route. This increased the performance of it by a lot. While this solution using both 2-opt with fireflies provided a good score, we believed that due to the time limit of 2 seconds, the randomization can sometimes make this algorithm perform worse. Therefore, we decided to not continue with the approach of using the Firefly global optimization algorithm.

2 Data Structures

2.1 Custom Randomization

A few of the algorithms we implemented and tried required randomization to find better solutions. This especially applies to the Firefly algorithm (see Section 1.3) which uses randomization to become a global optimization algorithm. The randomization implementation uses the basis Linear congruential generator (LCG) [7]. When calculating a new random number it uses the formula $X_{n+1} = (a \times X_n + c) \mod m$. The initial value of X_n is a seed value. Using this provides a good way to generate randomized numbers and shuffling arrays.

2.2 Graph

To model the problem we used a graph, where each city is represented as a vertex, and an edge connects every pair of vertices, with the distance between cities serving as the weight of the edge. In our implementation, the problem is represented as a complete graph where each city is a node connected to all other cities. The graph construct also contains a method that returns a sorted list of edges based on their lengths. This is for example used in the greedy heuristic, where we construct an initial tour based on the sorted edges from the graph. The strategy aimed at connecting cities by selecting the edges with the shortest lengths first.

2.3 Sparse Graph

The sparse graph data structure is designed to efficiently represent graphs with a low number of edges. It is implicitly represented through adjacency lists, where the direct neighbours of each node are stored in a list. The structure is created with the number of nodes specified, and the edges are added or removed based on the connections between nodes. This allows for efficient circle detection and neighbour look-up.

We mainly employed the sparse graph data structure to represent graphs characterized by a limited number of edges, optimizing operations such as circle detection and neighbour lookups. This sparse graph structure became the central data format for our tour construction, as seen in the iterative addition of the shortest edge in the Greedy heuristic.

3 Testing

To test our implementation outside Kattis, we built a number of test cases. We created some test graphs by explicitly listing their nodes and feeding them into our graph data structure. We mainly used this technique to cover edge cases, e.g. graphs with only one node. Furthermore, we built a method that generates random graphs based on a specified number of nodes. We leveraged the *rand* crate in order to generate random nodes. As those graphs are regenerated every time a test that calls them is executed, this is not a deterministic procedure. However, it helped us identify problems in graphs that we did not think of before.

4 Result

Some of the results are a bit odd and we believe it is worth briefly explaining our thoughts on them. First of all, it is interesting to note that the combination of 2- and 3-Opt yields better results together with Christofidis than just 3-Opt. In theory, 3-Opt should cover all possible 2-Opt moves. We believe that this is due to the fact that our 3-Opt implementation never tried all possibilities within two seconds. Therefore, it is beneficial to first do the simpler optimization on all vertices of the graph and then a few 3-Opt moves.

We also believe that it is possible to achieve much better results with simulated annealing. We found an issue in our implementation that we were not able to fix before the deadline. The algorithm requires to keep track of the distances of tours. To do this efficiently one can not calculate the distance from scratch each time, but rather should calculate the new distance based on the distance of the previous tour. However, we have some bug in the code that leads to negative distances during the execution. Observing that we know there is some issue with our distance calculation which we believe leads to the bad scores.

In implementing the Firefly algorithm without initial optimizations, we obtained relatively low scores. However, applying the 2-opt optimization to initial routes resulted in significant improvements across different fireflies. We also experimented with further optimization by implementing 3-opt, but this approach yielded poor scores due to time constraints. Running 3-Opt optimization for all different fireflies proved too time-consuming, leaving insufficient time for the actual Firefly algorithm to perform effectively.

In summary, our Christofidis implementation together with 2-Opt and 3-Opt optimization yielded the best result and a score of 30.03.

Algorithm	**Score**
Nearest Neighbour Hood	∼1.2
Greedy	5.97
Firefly	8.32
Firefly with 2-opt optimization	21.6
Firefly with 3-opt optimization	∼2.3
Lin Kernighan with 2-opt	23.2
Greedy with 2-opt	19.73
Christofidis with 3-opt	29.94
Christofidis with 2-opt & 3-opt	30.03
Simulated Annealing	3.06

Table 1: The total scoring of the different variations of algorithms we tested

References

[1] J. Håstad, *Notes for the course advanced algorithms*, Jan. 2000.

[2] E. Balas and N. Christofides, "A restricted lagrangean approach to the traveling salesman problem," *Mathematical Programming*, vol. 21, no. 1, pp. 19–46, 1981.

[3] J. B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings of the American Mathematical Society*, vol. 7, no. 1, pp. 48–50, 1956, ISSN: 0002-9939, 1088-6826. DOI: 10.1090/S0002-9939-1956-0078686-7.

[4] W. Tan, L. S. Lee, Z. Abdul Majid, and H.-V. Seow, "Ant colony optimization for capacitated vehicle routing problem," *Journal of Computer Science*, vol. 8, pp. 846–852, Jan. 2012.

[5] S. Lin and B. W. Kernighan, "An Effective Heuristic Algorithm for the Traveling-Salesman Problem," en, *Operations Research*, vol. 21, no. 2, pp. 498–516, 1973. [Online]. Available: http://www.jstor.org/stable/169020.

[6] S. N. Kumbharana and G. M. Pandey, "Solving travelling salesman problem using firefly algorithm," *International Journal for Research in science & advanced Technologies*, vol. 2, no. 2, pp. 53–57, 2013.

[7] S. Tezuka, "Linear congruential generators," in *Uniform Random Numbers: Theory and Practice*, Springer, 1995, pp. 57–82.