KTH Royal Institute of Technology
(Kungliga Tekniska Högskolan)

Advanced Algorithms: Project TSP

Kattis submission ID: **6432745**
Kattis score: **40.8252**

Frano Rajic, Ivan Stresec

Advanced Algorithms: DD2440 HT20-1
November 13, 2020

0. Contents

1. Travelling salesman problem

The travelling salesman problem (travelling salesperson problem; TSP) is one of the most famous NP-hard problems and is best defined by the question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?" [1].

1.1 Our TSP

For this project, we had to solve a 2D (Euclidean) version of the TSP as defined on the Kattis webpage. In short, each city is represented by two coordinates in a 2D plane. Our program receives the number of cities N and coordinates for each of those cities and must find the shortest Hamiltonian cycle it can (a cycle which visits each city only once, a tour) in under 2 seconds.

1.2 Programming Languages, Tools, and IDEs

For our solution, we have used C++ with the CMake tool for simple building and compiling. The rationale of using C++ was fairly simple: it offers higher efficiency compared to other dynamic languages while its standard library still contains most of the data structures and basic algorithms you need, removing the need to implement everything manually. We have also used Python with the Matplotlib library for some basic visualizations. For IDEs we used JetBrains' CLion and Microsoft's Visual Studio Code, and Git, with a GitHub repository, was used for version control and tracking changes.

2. Algorithms and implementation

To solve our problem, we mostly referred to the work by Johnson and McGeoch [2] which provides an excellent outline of the problem and algorithms designed to solve it. Altogether we have used 6 different algorithms mentioned in the work.

The algorithms used can be divided into two larger groups: tour construction algorithms and local optimization algorithms. Tour constructions algorithms build a starting tour, preferably as short as possible and then once we have a tour we use local optimization algorithms to improve the solution that the tour construction algorithm we used gave us.

In the following algorithms, and generally, we naturally look the TSP from a graph theory perspective. We think of cities as nodes (vertices) in a graph. In our case edges between these nodes are weighted by the Euclidean distance between the two nodes (cities) that an edge connects.

We should also mention that given a time restriction of 2 seconds, we created many time checks throughout our algorithms and stopped them if they took too long, i.e. longer than $1.98s$ or so. Most functions which implement those algorithms are given some point of time after which they must stop. Since most of the algorithms are iterative in nature this could be done quite simply by checking the elapsed time after each iteration.

2.1 Tour construction algorithms

For this project, we observed 3 different tour construction algorithms. Here we will outline the basics and shortly comment aspects of our implementation.

2.1.1 Greedy algorithm

Probably the most simple tour construction algorithm, the greedy algorithm simply picks the shortest edge it can, moving from some random starting city to the next until we create a Hamiltonian path. The only thing we have to be careful about is not to create a node with a degree larger than 2 or to create a cycle. In the end we just connect the two end-points of our path to create a cycle and we have our tour.

Implementations of this algorithm are straight-forward in any language so we don't discuss ours here.

2.1.2 Clarke-Wright saving's algorithm

The Clarke-Wright saving's algorithm is an algorithm originally designed to solve the vehicle routing problem, a problem very similar to the TSP.

The algorithm

To start off, we randomly pick a certain node. This node, or city, will be known as our hub. The idea is then to create $n-1$ pseudo-tours (cycles) in the form of (h, i, h), where h is our hub node, and i are indices of other cities. In terms of natural language, we create routes going from the hub to one city and back. Now, we want iteratively combine these pseudo-tours in such a way that we create a cycle with the shortest possible distance. For each pair of two cities, other than the hub, we calculate how much distance we would save if we connected the two pseudo-tours those towns participate in. Again, more intuitively, we want to calculate how much better would it be to, instead of returning to the hub after we visit one city, to visit another city after that first city, and only then return to the hub, i.e. replace pseudo-tours (h, i, h) and (h, j, h) with (h, i, j, h). This is also extended to pseudo-tours with more than 1 city, for instance, if we know we save some amount $s(i, j)$ by connecting cities i and j, we can connect these cities if we have two distinct pseudo-tours, one of which starts or ends with i, and one of which starts or ends with j. For example, we combine pseudo-tours $(h, i, ..., h)$ with $(h, ..., j, h)$ into $(h, ..., i, j, ..., h)$. It's worth noting that distances are symmetric so any cycle is still the same in any way if we reverse it, which is what we did to connect the two tour described in the last example. With these ideas, we can formulate the simple pseudo-code of the Clarke-Wright algorithm for the TSP [3]:

For a given set of n nodes do the following:

1. Select any node as a hub which we denote as node 0.
2. Compute savings $s(i, j) = c(0, i) + c(j, 0) - c(i, j)$ for all pairs of nodes, $c(x, y)$ is the distance (cost) between nodes x and y.
3. Sort the savings by decreasing values.
4. Create a set T of pseudo-tours $(0, i, 0)$ where $i = 1, 2, ...n - 1$.
5. Start from the top of the sorted savings and scan downwards, for each saving $s(i, j)$:
   (a) Check if there are two distinct pseudo-tours which end with i and j, if such tours exist link them at endpoints i and j, otherwise continue.
6. Return the last constructed tour.

Figure 2.1: Clarke-Wright saving's algorithm pseudo-code

The implementation

Considering our implementation, vis-à-vis the pseudo-code, two things are to be noted.

Firstly, savings were saved using tuples of size three, containing the two nodes and the savings amount. This way we could sort the savings by values like mentioned in step 3, and after sorting access the indices of two nodes each saving refers to for step 5.

Secondly, in step 5 we wanted to quickly access any pseudo-tours which start or end with some cities i and j without going through the whole set of pseudo-tours T. For this

purpose, we created a simple array of size $n-1$ containing pseudo-tours. If some pseudo-tour begins or end with a vertex of index i it can be accessed by indexing our array with i in $O(1)$. It can be easily observed that there is always at most one tour which starts or ends with some index i. Initialization of the array was easy as at the beginning each node i (excluding the hub) has only one pseudo-tour starting and ending with it: $(0, i, 0)$. Then, each time we link two pseudo-tours we remove those pseudo-tours from our array (we remove pseudo-tours at node indices which the two pseudo-tours start and begin with) and add the new pseudo-tour at indices of the nodes the new pseudo-tours starts and begins with. With this procedure, each iteration of step 5 has complexity $O(1)$.

The rest of the algorithm was straight-forward and was implemented in a simple fashion following the algorithm. Complexity is trivially $O(n^2 logn)$, the largest complexity being the sorting of the savings of which of there are n^2.

2.1.3 The Christofides approximation algorithm

The approximation algorithm of Christofides is a polynomial running time algorithm that has a worst-case ratio of just 3/2 assuming the triangle inequality.[4] A simple pseudo-code for this algorithm is in figure 2.2. An illustration is in given in figure 2.3

Suppose we are given an instance of TSP specified as a complete graph G with weights on its edges satisfying the triangle inequality. The Christofides approximation algorithm is as follows:

1. Construct a minimum spanning tree, M, for G.
2. Let W be the set of vertices of G that have odd degree in M and let H be the subgraph of G induced by the vertices in W . That is, H is the graph that has W as its vertices and all the edges from G that join such vertices. By the handshaking lemma, the number of vertices in W is even. Compute a minimum-cost perfect matching, P, in H.
3. Combine the graphs M and P to create a graph, G, but don't combine parallel edges into single edges. That is, if an edge e is in both M and P, then we create two copies of e in the combined graph, G.
4. Create an Eulerian circuit, C, in G, which visits each edge exactly once.
5. Convert C into a tour, T, by skipping over previously visited vertices.
6. Return the constructed tour.

Figure 2.2: The Christofides approximation algorithm pseudo-code [4]

The implementation

Construction of a minimum spanning tree was done by implementing Kruskal's algorithm as outlined in the pseudo-code [4]. For this purpose, we implemented the UnionFind data structure and also used the max heap structure from the C++ standard library.

For the perfect minimum-cost matching, we have used Kolmogorov's implementation [5] of the Blossom algorithm since the algorithm is very complex and the project instructions stated that an existing implementation can be used in this instance.

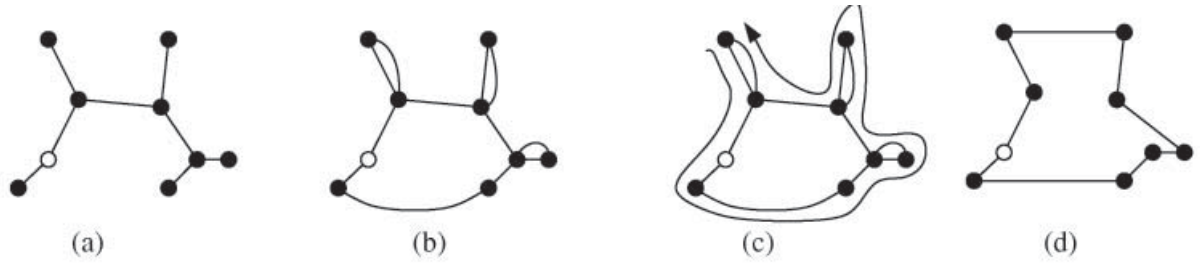

Figure 2.3: Illustrating the Christofides approximation algorithm: (a) a minimum spanning tree, M, for G; (b) a minimum-cost perfect matching P on the vertices in W (the vertices in W are shown solid and the edges in P are shown as curved arcs); (c) an Eulerian circuit, C, of G; (d) the approximate TSP tour, T [4].

An Eulerian circuit was created by traversing the graph using a stack data structure. A Hamiltonian path was constructed from the Eulerian circuit by simply skipping the vertices which occurred twice in the circuit.

2.2 Local optimization algorithms

2.2.1 2-opt

The algorithm

2-opt is a very simple, but also an efficient local optimization algorithm. The basic idea is as follows: we pick two edges in our tour and see if we could reduce the distance by removing them and reconnecting the vertices those edges originally combined. A very intuitive image of this algorithm can be seen in Figure 2.4.

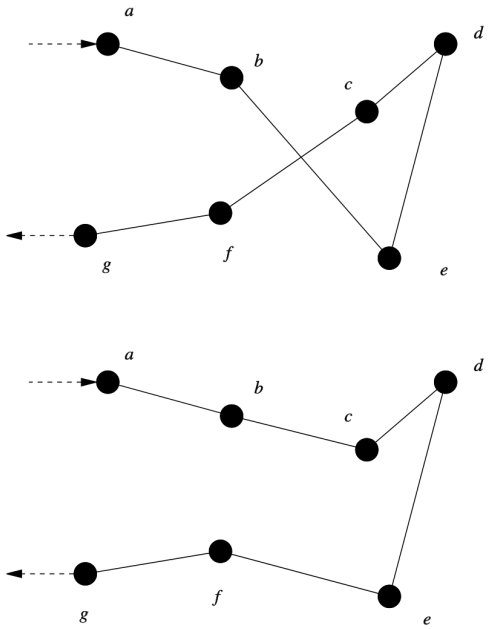

Figure 2.4: 2-opt procedure illustration [6]

The implementation

To implement 2-opt efficiently, we based our implementation on using k nearest neighbours, where k is a parameter that needs to be tuned. We took this nearest neighbour optimization idea from [2] and it boosted our original 2-opt implementation significantly. The nearest neighbours are calculated in a preprocessing step. For each of the n cities, k closest neighbours are found according to the minimum distance from the respective city. The indices of the closest cities are saved in an $n \times k$ array structure which is later used by 2-opt algorithm.

2-opt goes through all cities and looks where its closest neighbours are in the tour and then it tries to swap its two surrounding edges with the two surrounding edges of the closest neighbours. If the swap is valid and results in a decrease in the tour distance, then the tour is updated. If no swap can be made, the algorithm terminates.

2.2.2 3-opt

The algorithm

3-opt is an extension of the 2-opt optimization algorithm, where instead of 2 edges we pick 3 edges and determine if removing them in some way can provide a better solution. A simple illustration of the moves we can do with 3 edges is demonstrated in Figure 2.5. We can see that 3-opt moves are supersets of 2-opt moves, but also have moves that involve removing all 3 edges which aren't covered by 2-opt.

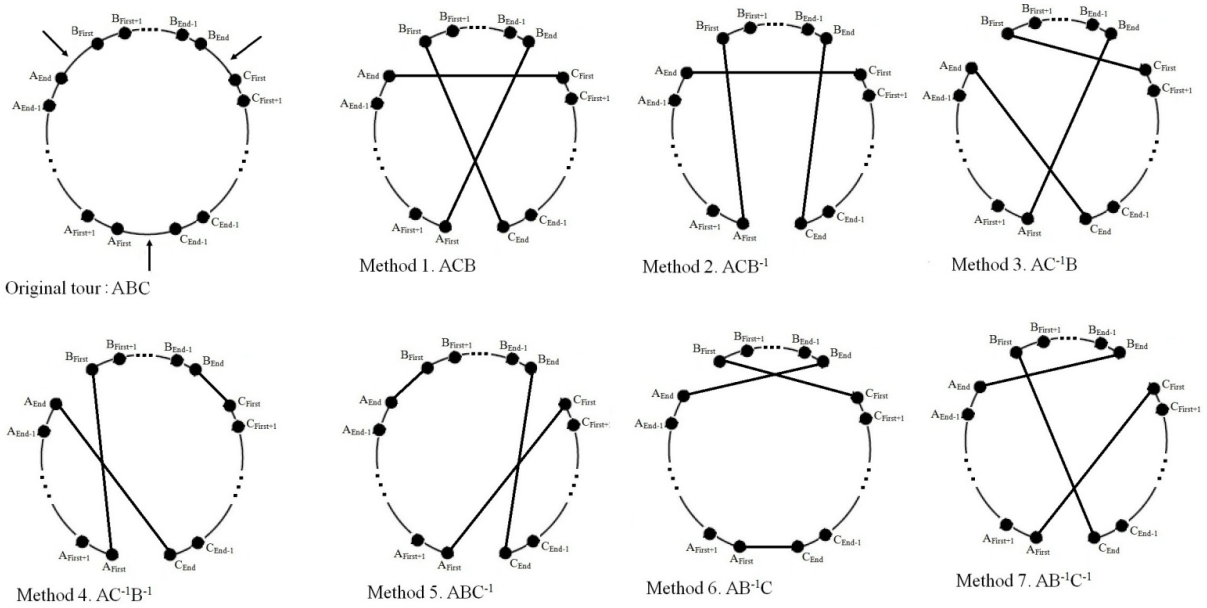

Figure 2.5: The seven possible 3-opt moves; the upper left graph is our starting tour, while the rest show possible relinking which can be done with the three chosen edges. [7]

The implementation

Here we went with a straight-forward implementation which simply went through all the possible combinations of 3 edges and checked which of the seven 3-opt moves would reduce

the distance. The first eligible move, a move which reduces the tour distance, that could be made was made. If no eligible moves were found we went to the next iteration.

2.2.3 Lin-Kernighan (k-opt)

The algorithm

The Lin-Kernighan heuristic (LKH; codename **Chokolino** in this report) is a generalization of 2-opt and 3-opt, which is why it is commonly referred to as k-opt. The algorithm is quite complex but has been proved to be very efficient in generating good solutions of the TSP.

A lot of work on the LKH has been done by K. Helsgaun, whose work we have used to learn more about the algorithm. The following is the pseudo-code of the basic algorithm which we have used [8]. It's important to note that x_i are edges we want to remove, y_i are edges we want to add, and G_n is the gain, the amount of distance by which we will have reduced the tour by removing some n edges x and adding some n edges y. Formally: $G_i = \sum_{i=1}^{n} g_i = \sum_{i=1}^{n} c(x_i) - c(y_i)$, where $c(x, y)$ is the weight of the edge, in this case the distance between two cities the edge connects:

1. Generate an initial tour T.
2. Let $i = 1$. Choose some node (city/town) t_1.
3. Choose an edge $x_1 = (t_1, t_2) \in T$.
4. Choose $y_1 = (t_2, t_3) \notin T$ such that $G_1 > 0$.
   If this is not possible go to Step 12.
5. Let $i = i + 1$.
6. Choose $x_i = (t_{2i-1}, t_{2i}) \in T$ such that:
   (a) if t_{2i} is joined to t_1, the resulting configuration is a tour, T',
   (b) $x_i \neq y_s$ for all $s < i$.
   If T' is a better tour than T, let T = T' and go to Step 2.
7. Choose $y_i = (t_{2i}, t_{2i+1}) \notin T$ such that:
   (a) $G_i > 0$,
   (b) $y_i \neq x_s$ for all $s \leq i$,
   (c) x_{i+1} exists.
8. If there is an untried alternative for y_2, let $i = 2$ and go to Step 7.
9. If there is an untried alternative for x_2, let $i = 2$ and go to Step 6.
10. If there is an untried alternative for y_1, let $i = 1$ and go to Step 4.
11. If there is an untried alternative for x_1, let $i = 1$ and go to Step 3.
12. If there is an untried alternative for t_1, then go to Step 2.
13. Stop (or go to Step 1)

Figure 2.6: The basic Lin-Kernighan algorithm

The pseudo-code is hard to understand at first, but the algorithm can be outlined as follows: we choose edges to add and remove until we find sets of edges to add and remove which can create a shorter tour. The whole process is done iteratively for all possible combinations.

The implementation

The pseudo-code is relatively complicated, and so was the implementation, especially in C++. We found a blog post by A. Mathéo [9] which outlines an idea of using recursion instead of goto statements which we see in the original pseudo-code (Figure 2.6). We borrowed this idea and implemented it ourselves. Below, in Algorithm 1 we wrote the idea's pseudo-code.

Even though complicated at first sight, what Algorithm 1 suggests is that the first 5 steps of the basic LKH pseudo-code shown in Figure 2.6 are made in one function main_loop, and steps from 6 and 7 are recursively implemented using functions choose_x and choose_y so as to cover the goto statements defined in steps 8 to 12.

Concerning our implementation, there are many details to note which can be seen in our code concerning the C++ structures and checks we had to make. One of the most interesting things is our implementation was creating a new tour given the set of edges to remove X and set of edges to add Y. The idea here was for each node to create tuples containing the nodes left and right of that node in the tour. Then, when removing edges X we noted missing nodes in our tuples as -1. To create a new tour, we went through the set of edges to add Y and filled the -1 spots using with respective node in the added edge. Finally we started at some random node and traversed the new tour though the created tuples, being cautious about possible cycles. The new tour could be returned in O(n) this way if it was valid, i.e. it did not have cycles.

Other than that some improvements were also used, such as using the k nearest neighbours for the choice of y edges, much like in 2-opt, as well as well as some other refinements mentioned in Helsgaun's work [8]. Tweaking some of these improvements and speeding up the code resulted in a performance increase.

The rest of the implementation and details can be seen in our code.

Algorithm 1: Recursive Lin-Kernighan

```
create an initial tour T;
do
    call main_loop(T);
while tour T has improved;

Function main_loop(T)
    declare X, Y; // structures which remember edges x_i and y_i
    for t_1 ∈ T do
        for t_2 next to t_1 in tour do
            x_1 = (t_1, t_2);
            X = new_set(x_1);
            for t_3 in cities do
                y_1 = (t_2, t_3);
                Y = new_set(y_1);
                G_1 = c(x_1) − c(y_1);
                if G_1 > 0 then
                    if choose_x(T, t_1, t_3, G_1, X, Y) then
                        return True;
                    end
                end
            end
        end
    end

Function choose_x(T, first, last, gain, add, remove)
    for t_{2i+1} in cities do
        x_i = (last, t_{2i}); G_i = gain + c(x_i); if t_{2i}! = t_1 and x_i ∉ Y then
            X.add(x_i);
            final_edge = (t_{2i}, t_1);
            new_X = X.add(final_edge);
            if construct_new_tour(T, new_X, Y) then
                return True;
            else
                return choose_y(T, t_1, t_{2i}, G_i, X, Y);
            end
        end
    end

Function choose_y(T, first, last, gain, add, remove)
    for t_{2i+1} in cities do
        y_i = (last, t_{2i+1});
        G_i = gain − c(y_i);
        if G_i > 0 and y_i ∉ remove then
            remove.add(y_i);
            return choose_x(T, first, t_{2i+1}, G_i, add, remove);
        end
    end
    return False;
```

3. Testing and results

To tested our implementation on different examples to get a feeling about the quality of performance by using three publicly available datasets (att48, berlin52 and a280) as well as a random city generator that we implemented to yield uniformly distributed numbers in $[-10^6, 10^6]$ for the x and y coordinates of each city. The big numbers were used on purpose to test the possible edge cases and notice possible overflows when trying different solutions and changing logic of algorithms. We used the random city generator most of the times to generate $N = 1000$ cities and to evaluate the competing algorithms on this test set, as it was the most challenging to solve among the used datasets in a time frame of 2 seconds.

To have a quantifiable measure of how a tweak or a parameter of implemented algorithms affected the overall performance, we always seeded the generator. The final assessment of performance was the submission on Kattis which we used to determine if our tests were representative and how close to the targeted score we were.

Results of *Greedy*, *Clarke Wright*, *Christofides* and *Christofides+Chokolino* on the dataset of randomly generated cities are shown in Figure 3.1. The greedy algorithm was very fast, but of course, provided awful solutions. Conversely, *Clarke Wright* and *Christofides+ Chokolino* gave much better results, but took a longer time. Clarke-Wright solutions were quite intuitive. By its own, Christofides' algorithm did not give very intuitive solutions, as it combines the results of the minimum spanning tree and minimum-weight perfect matching with not a lot of postprocessing in terms of defining a Hamiltonian path. However, the paths it would produce could be very well optimized and outperformed paths produced by the Clarke-Wright algorithm, even though they were initially better.

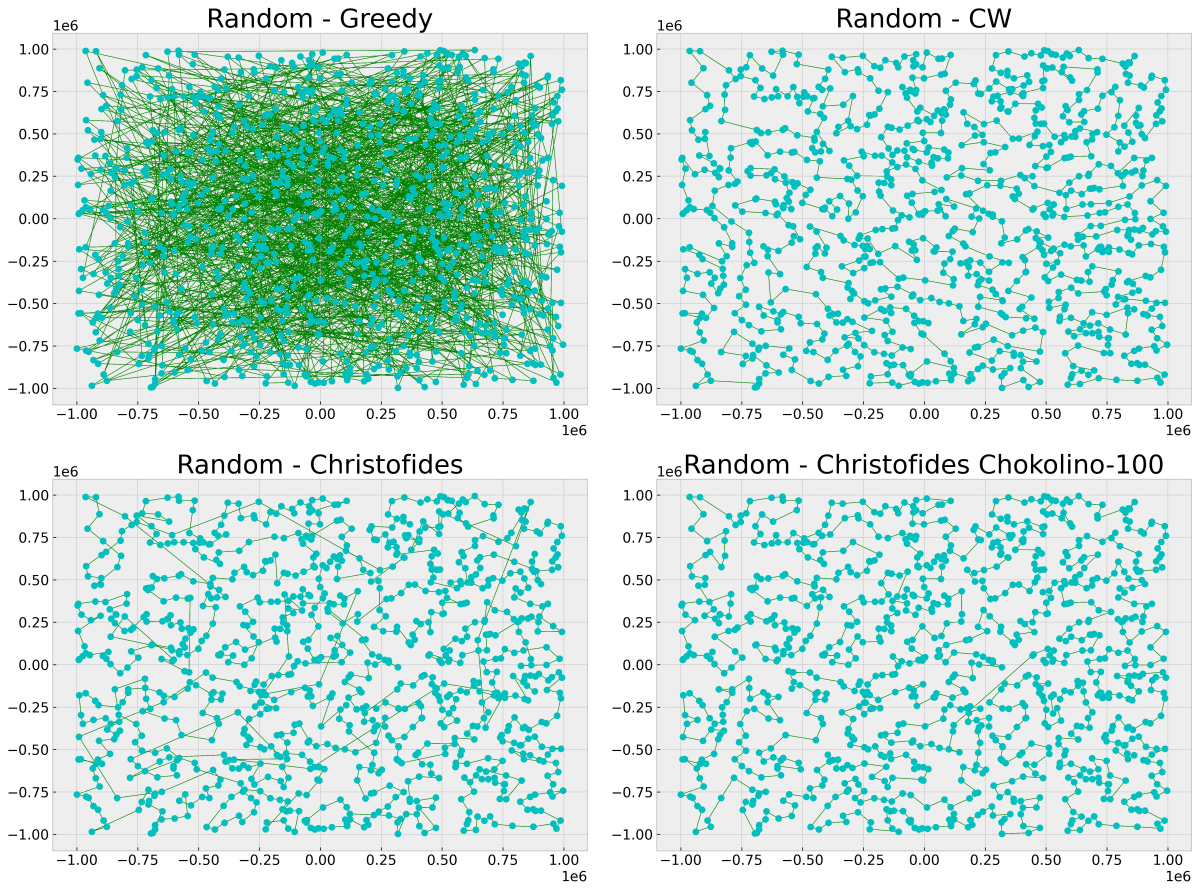

Figure 3.1: Results of different algorithms on 1000 random cities

Algorithm	Distance	Time elapsed
Greedy	1,031,578,702	0.03
CW	50,996,759	1.07
CW 2opt-100	49,659,772	1.30
CW 3opt	49,656,016	1.95
CW Chokolino-100	48,107,020	1.95
Christofides	61,212,251	0.33
Christofides 2opt-100	50,364,384	0.61
Christofides 3opt	50,777,627	2.00
Christofides Chokolino-100	47,442,184	1.95
Christofides Chokolino-2opt-100-3opt	47,353,370	1.95

Table 3.1: Random cities. Number 100 refers to number value of k used in k nearest neighbours.

Additionally, some of the resulting scores on the randomly generated cities are shown in the table 3.1. Generally, we can see that the Clarke-Wright saving's algorithm created

better initial solutions than the Christofides' algorithm, but testing showed that it's on the one hand much slower, especially on bigger examples with up to 1000 cities, and on the other hand harder to optimize, even if no time deadline is given. Optimizing Christofides' initial tour gave much better results.

Concerning local optimization methods, Chokolino performed very well and created very good solutions. Tweaking some parameters of our implementation made it run quite fast but limited the solution somewhat. If there was time we also ran the 2-opt and the 3-opt after Chokolino as it showed some improvement in the final results.

In figure 3.2, visualization of the resulting tours for the att48 problem are shown. att48 is about finding the shortest tour for 48 capital cities of the US. *Clarke Wright* and *Christofides+Chokolino* again give the most intuitive solutions. The corresponding performances are shown in table 3.2.

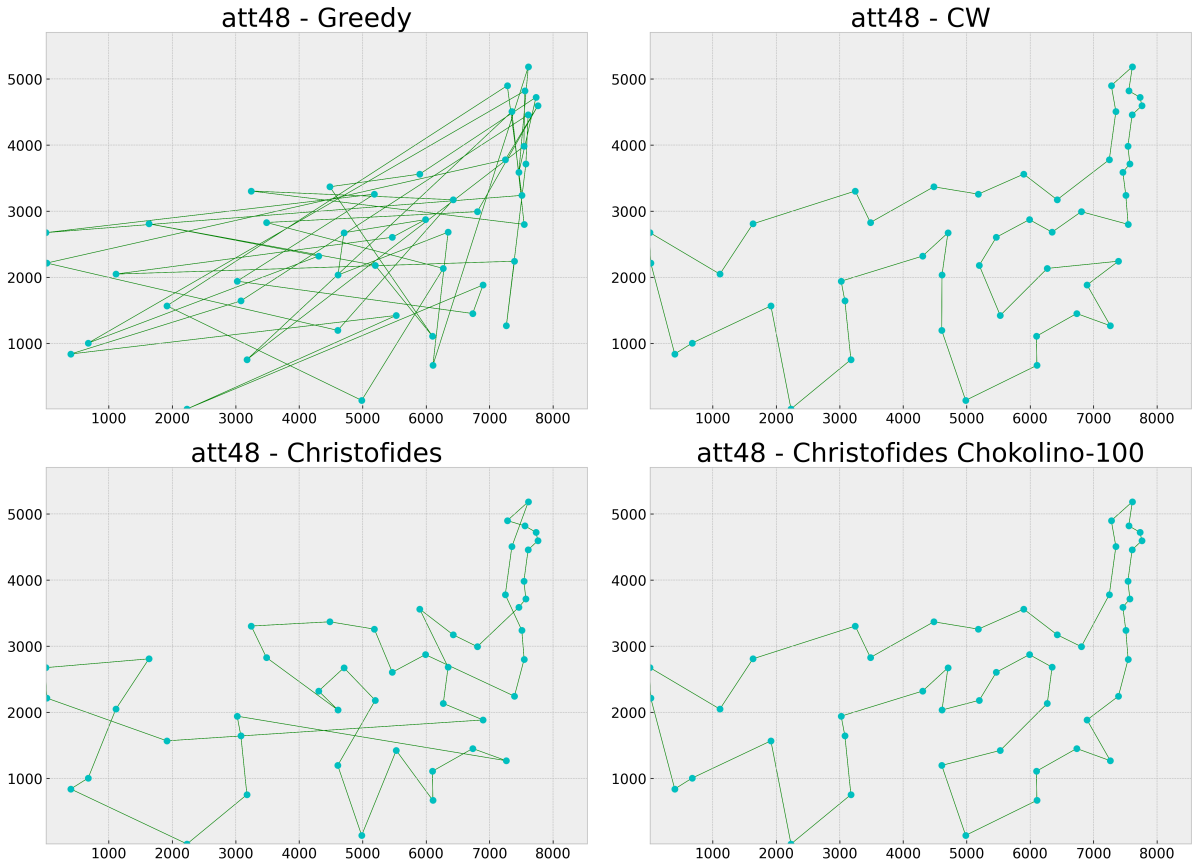

Figure 3.2: Results of different algorithms on 48 capitals of the US

Algorithm	Distance	Time elapsed
Greedy	154673	0.0002
CW	35015	0.0028
Christofides	44273	0.0016
Christofides 2opt-100	35477	0.0052
Christofides 3opt	34714	0.0133
Christofides Chokolino-100	33894	0.0700
Christofides Chokolino-2opt-3opt-100	33872	0.0626

Table 3.2: att48, capitals of the US, 48 cities. The best known solution is 10628.

Finally, in table 3.3 and 3.4 the results on the Berlin dataset and 280 drilling points problem are shown respectively. The value of the best known solution is also noted.

Algorithm	Distance	Time elapsed
Greedy	22,236	0.0001
CW	8,289	0.0023
Christofides	9,975	0.0021
Christofides 2opt-100	7,980	0.0053
Christofides 3opt	8,005	0.0137
Christofides Chokolino-100	7,777	0.1765
Christofides Chokolino-2opt-3opt-100	7,777	0.1619

Table 3.3: Berlin, 52 points. The best known solution is 7542.

Algorithm	Distance	Time elapsed
Greedy	2,811	0.0035
CW	2,940	0.0616
Christofides	3,991	0.0263
Christofides 2opt-100	2,751	0.0623
Christofides 3opt	2,739	0.8376
Christofides Chokolino-100	2,659	1.9504
Christofides Chokolino-2opt-3opt-100	2,653	1.9515

Table 3.4: a280, drilling problem. The best known solution is 2579.

The final Kattis score was the result of combining the Christofides construction algorithm with local optimization heuristics in the following order:

1. Chokolino (k=300)
2. 2-opt (k=300)
3. 3-opt

As already mentioned, the execution of the program would stop if any of these steps took too long, and the program would output the best solution up to that point.

4. Conclusion

In this project, we have tested 6 different algorithms which try to create approximate solutions for the TSP in terms of both tour construction and local optimization. More specifically, we tested greedy tour construction, the Clarke-Wright algorithm, the Christofides' algorithm, 2-opt, 3-opt, and the Lin-Kernighan (k-opt) algorithm. All of them performed pretty well, and we gained insight into some of their strengths and weaknesses considering the time of execution and the quality of the solution.

However, there is a lot of room for improvement, starting from improving the performance of our Christofides' algorithm implementation, as well as tuning and speeding up our implementations of the local optimization algorithms. It is also highly likely that we could exclude using our implementations of the 2-opt and 3-opt heuristics if we worked on our implementation of the Lin-Kernighan algorithm some more and made it more efficient, since it is conceptually a superset of the other two. There are also many randomized and nature-inspired algorithms which we haven't touched upon at all which could very well provide better solutions than our own.

On the whole, the project was interesting and we learned quite a lot about the TSP, as well as the methods and algorithms designed to solve it.

4. Bibliography

[1] Wikipedia contributors. *Travelling salesman problem*. URL: https://en.wikipedia.org/wiki/Travelling_salesman_problem (page 1).

[2] D. Johnson and L. A. McGeoch. "The Traveling Salesman Problem: A Case Study in Local Optimization". In: 2008 (pages 2, 6).

[3] Hoon Liong Ong and J.B. Moore. "Worst-case analysis of two travelling salesman heuristics". In: *Operations Research Letters* 2.6 (1984), pp. 273–277. ISSN: 0167-6377. DOI: https://doi.org/10.1016/0167-6377(84)90078-6. URL: http://www.sciencedirect.com/science/article/pii/0167637784900786 (page 3).

[4] Roberto Tamassia Michael T. Goodrich. *Algorithm design and applications*. first. Wiley, 2015 (pages 4, 5).

[5] Vladimir Kolmogorov. "Blossom V: a new implementation of a minimum cost perfect matching algorithm". In: *Mathematical Programming Computation* 1.1 (Apr. 2009), pp. 43–67. DOI: 10.1007/s12532-009-0002-8. URL: https://link.springer.com/article/10.1007%5C%2Fs12532-009-0002-8 (page 4).

[6] PierreSelim. *2opt procedure*. 2009. URL: https://commons.wikimedia.org/wiki/File:2-opt_wiki.svg (page 5).

[7] Hassan Ismkhan and Kamran Zamanifar. "Using ants as a genetic crossover operator in GLS to solve STSP". In: Dec. 2010. DOI: 10.1109/SOCPAR.2010.5686165 (page 6).

[8] Keld Helsgaun. "An effective implementation of the Lin–Kernighan traveling salesman heuristic". In: *European Journal of Operational Research* 126.1 (2000), pp. 106–130. ISSN: 0377-2217. DOI: https://doi.org/10.1016/S0377-2217(99)00284-2. URL: http://www.sciencedirect.com/science/article/pii/S0377221799002842 (pages 7, 8).

[9] Arthur Mathéo. *Lin-Kernighan in Python*. URL: https://arthur.maheo.net/implementing-lin-kernighan-in-python/ (page 8).