**Vladimir Milenković**

# The Travelling Salesman problem – Comparison of different approaches

Diploma in Computer Science

Trinity College

28th April 2020

# Proforma

| | |
|---|---|
| Name: | **Vladimir Milenković** |
| College: | **Trinity College** |
| Project Title: | **The Travelling Salesman problem – Comparison of different approaches** |
| Examination: | **Diploma in Computer Science, May 2020** |
| Word Count: | **TODO**[1] |
| Project Originator: | **Dr T. Sauerwald and V. Milenkovic** |
| Supervisor: | **Dr T. Sauerwald** |

## Original Aims of the Project

To code, compare and contrast different algorithms and approaches in solving the Travelling Salesman Problem. All the code done for this project is easy to use for anybody wishing to tackle this problem, and the overview of all the algorithms with the suggested usecase for each one is included as well.

## Work Completed

All work that was done during this project will be mentioned in the dissertation, the code will be submitted as well.

## Special Difficulties

Getting Lin-Kernighan algorithm to work significantly better than the other algorithms, reproducing the results mentioned on the original paper[12]. Also, the Blossom algorithm for minimum weight perfect matching is hard to get working.

[1]This word count was computed by `detex diss.tex | tr -cd '0-9A-Za-z \n' | wc -w`

## Declaration

I, Vladimir Milenkovic of Trinity College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed Vladimir Milenkovic

Date

# Contents

# List of Figures

# Acknowledgements

Add some acknowledgements here. Please, do add some acknowledgements here.

# Chapter 1

# Introduction

## 1.1 Motivation

The main aim of my CST Part II project is comparing the different approaches taken throughout the years for the Travelling Salesman problem - a very well-known optimisation problem. In that problem, we are given a weighted graph $G = (V, E, w)$, where $V$ denotes the set of vertices, $E$ denotes a set of edges, and $w : E \to \mathbb{R}^+$ is a weight function, mapping edges to weight that's assigned to them. We are interested in finding a minimum-weight Hamiltonian cycle (a cycle covering all the nodes) in this graph. More formally, if $n$ is the number of nodes in the graph (size of $V$), we are searching for a permutation of nodes $v_1, v_2, \ldots, v_n$ such that $w(v_1 \to v_2) + w(v_2 \to v_3) + \cdots + w(v_{n-1} \to v_n) + w(v_n \to v_1)$ is minimal possible. As we can see, a problem similar to this can arise in many real-world situations (a mailman having to visit a list of houses to deliver the mail, for example), so it is expected that a lot of effort has been put into solving this problem as efficiently as possible. This problem is known to be $\mathcal{NP}$-hard, basically meaning we cannot expert a polynomial time algorithm solving this problem. With that in mind, we see that we can compare the algorithms based on different metrices, including speed, space, approximation ratio, average closeness to the optimum and many more. This project will try to give a summary of some more or less successful attemps, and analyze what are the advangates and disadvantages of the algorithms involved.

## 1.2 History

First mathematical problems related to the Travelling Salesman problem date back to 1800s, where the problem similar to this one was treated by the Irish mathematician Sir W. R. Hamilton and by the British mathematician T. P. Kirkman. Since then, the problem was a topic for many famous mathematicians. An thorough overview of their early work can be found in the Graph Theory, 1736-1936 book written by N. Biggs, E. K. Lloyd and R. J. Wilson. The TSP, in the form we are tackling it today appears to be first studied in 1930s by Karl Menger in Vienna and Harvard. Throughout the years, the problem attracted much more attention because of the various ways to utilise its solution in many technology areas. A very good description of early studies concerning the TSP can be located in A. Schrijver's paper "On the history of combinatiorial optimisation (till 1960)""[16].

Nowadays, even though the NP-hardness of the TSP was proven more than half a century ago, the algorithms approximating this problem are getting better and better. The Concorde TSP solver (`http://www.math.uwaterloo.ca/tsp/concorde/index.html`), created by Applegate et al. in early 2000s, is the fastest TSP solver as of 2020, holding the record of fully solving the 85,900 nodes TSP instance[1] and solving all of the TSPLIB instances available. Some further optimal solutions to large TSP instances can be found on the following hyperlink: `http://www.math.uwaterloo.ca/tsp/optimal/index.html`.

## 1.3 Algorithms

In this section, I will list all the algorithm types that I have taken into consideration for this project. I have made this choice after analyzing already existings benchmarks about Travelling Salesman problem, trying to find the best tradeoff between complexity, running time, difficulty to reproduce and the result that the algorithm is producing. After having that finished, I have decided to split the algorithms in several classes, listed below:

- **Exact algorithms**
- **Approximation algorithms**
- **Improvement algorithms**
- **Heuristic algorithms**
- **Optimization algorithms**

# Chapter 2

# Preparation

In this chapter, I will describe, in full detail, all the algorithms and all the theoretical background needed to start the implementation part of the project, as well as finer details about some of the algorithms.

## 2.1 Computer characteristics

All the experiments in this project are run on my personal computer. Its characteristics are:

- CPU: Intel Pentium G2020 (2.9 GHz, dual-core)
- RAM: 16GB DDR3
- VGA: Nvidia GT 640
- HDD: 1TB

## 2.2 Programming language choice

Before starting any kind of implemenation, a backup tool has to be chosen. I have chosen a union of two things - a private GitHub repository and a backup on my external HDD. I had no problems with losing any data during the project implementation, but having multiple backup tools kept me safe throughout.

After considering multiple programming languages to be my language of implementation for this project, I chose C++. There are couple of reasons behind that decision: that is the language I am the most familiar with implementing competitive programming algorithms in it, and it also compares very solidly versus other options - which is the reason which makes it a primary competitive programming choice for most people. I am also very familiar with C++ Standard Template Library (more commonly known as STL), which helps me in writing optimised code faster, without the need to implement some commonly used data structures on my own. For easily keeping track of how long the program has been executing, I used the functions which can be found under the std::chrono namespace.

The only external library I will use in this project is Library for Efficient Modeling and Optimization in Networks (LEMON)[7] - an open source C++ library providing efficient implementations of some famous algorithms and data structures. I will use only the implementation of minumum weight perfect matching (Blossom algorithm), as a part of the implementation of the Christofides algorithm).

Concerning the evaluation chapter, I had to make some plots. In order to automatise the testing phase, I wrote and used some bash scripts. They will be included with all the code for this project.

Producing the plots is done in Python, using the matplotlib library.

## 2.3 Knowledge necessary

In order to understand and implement all the algorithms, I had to connect various areas of computer science, and in first two years of the Tripos a lot of stuff has been covered, but not all.

A brief overview of the Tripos subjects that do intersect with the theme of this project is given below:

- **Complexity theory** is definitely a must - all the knowledge needed to understand the complexity analysis section is covered in this Part IB course.

- **Algorithms** - this course is all about the TSP algorithms. The course has definitely helped me in writing the actual code. Also, my previous experience with coding has highly contributed to the speed of the actual implementation, as well as reducing the time necessary for finding flaws and fixing them fast.

- **Programming in C and C++** - as already mentioned, the whole project is implemented in C++ programming language, so this course helped me not to run in some of C++ cavities.

- **Artificial Intelligence** course has helped me to better understand the heuristic approaches in general, and some of the algorithms mentioned in this work are similar to some of the algorithms mentioned in the course - applied to different problems, however.

## 2.4 Complexity analysis

I have already mentioned the fact that the Travelling Salesman Problem is not expected to be solved in polynomial time. Now, we would like to see in which class does this problem belong to. In order to do that, we do not need the fully detailed definitions of them because they are not the main point of this dissertation - so I will give a brief and informal definition of complexity classes we take interest in here:

- $\mathcal{P}$ - the set of all problems which can be correctly solved in polynomial time.
- $\mathcal{NP}$ - the set of all problems whose *solutions* can be *verified* in polynomial time.
- **$\mathcal{NP}$-hard** - all the problems which can't be solved in polynomial time. This definition is very informal, but it suffices here.
- **$\mathcal{NP}$-complete** - problems which are both in $\mathcal{NP}$ and $\mathcal{NP}$-hard.

Now, the Travelling Salesman Problem, defined in the introduction is not $\mathcal{NP}$-complete. This might look a bit strange at first glance, but then, we remember that all $\mathcal{NP}$-complete problems are so-called decision problems - problems deciding whether something is true or not.

It is not hard to see that the TSP does belong to the $\mathcal{NP}$-hard class, using my previous $\mathcal{NP}$ complexity class definition. Also, for the formal proof, R. Karp showed in 1972.[11] that the Hamiltonian cycle problem is $\mathcal{NP}$-complete, which implies the $\mathcal{NP}$-hardness of the TSP. But, surprisingly?, it does not belong to $\mathcal{NP}$. In the verification problem, we would be given a graph and one of cycles from that graph, and we are to judge whether that exact cycle is the shortest one. Note that we can not decide that without actually solving the TSP and knowing what the optimal cycle's cost is, so this can not be done in polynomial time knowing the fact we can not solve TSP in that time. So, this definition of the TSP is not in $\mathcal{NP}$, making it not in $\mathcal{NP}$-complete as well.

The thing that is a bit strange in the above conclusion is the fact that most of the famous 'hard' problems are $\mathcal{NP}$-complete as well. But, all those problems are decision problems, unlike our first TSP definition (and pretty much the only sensible definition of this problem). We can artificially make TSP a decision problem, by formulating it as: given a graph $G$ and a number $p$, decide whether the cost of the shortest cycle in $G$ is less than or equal to the number $p$. This, rephrased TSP, is indeed in $\mathcal{NP}$-complete - the verification is straightforwardly done in polynomial time.

Conclusion to all this is that we definitely need and that it is actaully sensible to use some alternate problem solving algorithms than just the exact one - we need to apply some heuristics.

## 2.5 TSP types

The TSP, as defined in the beginning of the dissertation, can be defined on any kind of graph. However, a lot of real TSP instances do satisfy some additional constraints regarding edge weights. Some of these properties of TSP instances can help us in developing some algorithms which can profit from the fact that the weights are behaving according to some further conditions. In this section, I will introduce a couple of terms that will come in handy later.

In most of the algorithms, the graph distance matrix is going to be used as a way to represent the graph. A distance matrix is a $n \times n$ matrix $A$ representing the weights of edges of the graph, where $A_{i,j} = w(i \to j)$.

A TSP instance is called a **Symmetric TSP** instance if and only if the distance matrix is symmetric. So, in the case of a symmetric TSP, for each $u$ and $v$, $w(u \to v) = w(v \to u)$. Note that most of the non-artificially generated and in practice arising graphs are symmetric, making this a very natural restriction on edge weights, so I will focus exclusively on solving the symmetric TSP, if not explicitly said otherwise. Thus, I am able to write $w(a, b)$ instead of $w(a \to b)$.

**Metric TSP** is a subset of symmetric TSP where distance matrix obeys the *triangle inequality.* That means, for each triplet of vertices $a$, $b$, $c$ (we can assume they are pairwise connected so we don't have to deal with infinities), $w(a \to b) \leq w(a \to c) + w(c \to b)$. It doesn't make sense that the triangle inequality doesn't hold for something which is a road network or similar. Specially, if we can represent our nodes as a points in a Euclidean space and the distances are Euclidean distances between the corresponding points, that graph belongs to a special subset of Metric TSP called the **Euclidean TSP**[1].

[1]For Euclidean TSP, a PTAS[2] (polynomial-time approximation scheme) algorithm can be

## 2.6 Algorithm overview

The Table 2.1[2] listing the important information for all the algorithms in the project can be found here.

| Name | Type | Running time | Reference |
|---|---|---|---|
| Bruteforce | Exact | $\mathcal{O}(n!)$ | folklore |
| Held-Karp algorithm | Exact | $\mathcal{O}(n^2 2^n)$ | [3], [9] |
| Branch and bound algorithm | Exact | $\mathcal{O}(n!)$ | folklore |
| 2-approximation | Approximation | $\mathcal{O}(n^2)$ | TODO |
| Christofides | Approximation | $\mathcal{O}(n^3)$ | [4] |
| $k$-opt | Improvement | $O(n^k)^3$ | [8], [6] |
| Lin-Kernighan | Improvement | $O(n^{2.2})$ | [12] |
| Random algorithm | Heuristic | $\mathcal{O}(n)$ | folklore |
| Nearest neighbour heuristic | Heuristic | $\mathcal{O}(n^2)$ | folklore |
| Cheapest insertion heuristic | Heuristic | $\mathcal{O}(n^2 \log n)$ | [14] |
| Farthest insertion heuristic | Heuristic | $\mathcal{O}(n^2 \log n)$ | [14] |
| Nearest insertion heuristic | Heuristic | $\mathcal{O}(n^2 \log n)$ | [14] |
| Random insertion heuristic | Heuristic | $\mathcal{O}(n^2 \log n)$ | [14] |
| Convex-hull insertion heuristic | Heuristic | $\mathcal{O}(n^2 \log n)$ | folklore |
| Ant colony optimization | Optimisation | $\mathcal{O}(cnt\_ants \cdot n^2)$ | [5] |

Table 2.1: Algorithms overview

## 2.7 Algorithm types

### 2.7.1 Exact algorithms

In this section, we will see a few algorithms that solve the Travelling Salesman problem optimally - their output is guaranteed to be correct. Algorithms with that guaranteed (probably the most widely-used algorithm category) is called an exact algorithm. As we have shown in the previous section, there can be no polynomial time algorithm belonging to this category. Furthermore, an open problem is whether there exists an algorithm solving the TSP with running time of $\mathcal{O}^*(1.99999^n)$[17], where $n$ is the number of vertices of the graph. That doesn't

applied, which finds the solution that is within a factor of $1 + \frac{1}{c}$ in $\mathcal{O}(n(\log n)^{(\mathcal{O}(c\sqrt{d}))^{d-1}})$ for every fixed $c > 1$, where $d$ is the dimension of the Euclidean space.

[2]In the table, $n$ is the number of nodes in the graph

stop us in comparing different algorithms here, because we can clearly infer that one algorithm is better than the other in certain circumstances.

### 2.7.2 Approximation algorithms

An approximation algorithm does not compute the optimal solution, but we will have some guarantees about the value we obtain using these algorithms. If we don't require our solution to be fully optimal, but we just don't want to be too wrong, approximation algorithms can guarantee the solution we want really fast.

We should note here that it is not possible to polynomially approximate the TSP as good as we want. Determining how good can we approximate the problem is still a major open problem in the field of computer science. A real number $r$ is said the be an inapproximability bound for an optimization problem $\mathbb{P}$ if it is proved that a problem having to find a solution that is better than $r$ times the optimal solution for $\mathbb{P}$ is $\mathcal{NP}$-hard. A paper TODO CITAT released in 2013 improves the inapproximability bounds to 185/184 for the symmetric TSP, and 117/116 for the asymmetric TSP., though there is no proof that we can't improve the bounds further.

### 2.7.3 Improvement algorithms

The general idea here is that we start from any solution, and try to iteratively improve the solution until we converge or until we are satisfied with the cycle found. There are no guarantees of any kind for iterative improvement algorithms, but we will empirically validate later on that they do perform better than some algorithms with stronger mathematical background.

### 2.7.4 Heuristic algorithms

The TSP is a really famous problem. Having that in mind, it's sensible that a lot of heuristics, some better than the other, were tried in order to get close to the optimal solution. In this section, we will discuss some of the attempts tried, and later we will show how do they compare to each other.

### 2.7.5 Local search algorithms

A lot of effort has been put in the solving of the TSP. Thus, a lot of general ideas have been applied to it, with more or less success. Genetic algorithms, tabu search algorithms, ant colony optimization algorithms, simulated annealing are just the small subset of general heuristics that have been tested on the TSP. In my work, I have implemented the ant colony optimization algorithm.

## 2.8 Tested Data Sets

After having covered all the algorithms in the project, we come to a part where we need some data to run the algorithms on. I have decided to split all the data I will be testing on in two large classes - **synhetically generated data** and **real-world data**. Among each of those, there will be several different TSP classes we need to take care of, including symmetric/asymmetric TSP, metric TSP, euclidean/non-euclidean, etc. We also need to make some decision when making the synthetic data - the biggest focus of synthetic data generation will be the Euclidean TSP.

When speaking about real-world data, all the instances my implementation has been tested on is collected from the TSPLIB online library of TSP instances. There, we can find the optimal solution value for some TSP instances I will test the algorithms on, so we can see how good our solution compares to the optimal one even if I can't fully solve the TSP test.

All the data is in one of the TSPLIB formats availabe, and I have made all of my algorithms to run on distance matrix (apart from the convex hull heuristic, which uses a set of points). So, the code to convert all of different TSPLIB format into distance matrices is also included in the code repository attached to this project.

To convert the pairs of latitude and longitude in the distance, the *haversine formula* was implemented. The haversine formula is the special case of the *law of haversines*, a formula relating the sides and the angles of spherical triangles.

# Chapter 3

# Implementation

## 3.1 Exact algorithms

### 3.1.1 Bruteforce algorithm

The very first idea that would come up on our minds when taking a look at this problem is that we can try all the possible Halimtonian cycles. Number of Hamiltonian cycles in a complete graph is equal to the number of permutations of the set of vertices. If there are $n$ vertices in the graph, we have $n!$ permutations (and $n!$ candidate solutions). However, we don't need to consider cyclic rotations of same permutation, so $(n-1)!$ permutations are left to examine. Implementing this naively, we need $\mathcal{O}(n)$ time to calculate the cost for a given cycle, so the naive implementation of this algorithm results in $O(n \cdot (n-1)! = n!)$ complexity.

I am using the C++'s inbuilt function *bruteforce algorithm* for generating all the candidates for the solution (a permutation of all the nodes is equivalent to a cycle covering all the nodes), calculating the cost for each of them in $O(n)$, and taking the overall minimum.

### 3.1.2 Held-Karp algorithm

However, we can do a lot better than the *bruteforce algorithm.* We will try to use the dynamic programming approach - use the solution of already solved smaller subproblems in order to solve the bigger subproblem, and repeat this until we come to the whole problem - find the optimal cycle on the whole graph.

Let us have currently visited $k < n$ vertices, and let those vertices be $v_1$, $v_2$, $\ldots$, $v_m$. Note that for the continuation of the algorithm we don't actually need to keep track of the order in which we have traversed all the nodes but $v_m$ - the last node currently visited. So, instead of keeping track of $(v_1, v_2, \ldots, v_m)$, we can keep track of $(\{v_1, v_2, \ldots, v_m\}, v_m)$, meaning we can describe the state we are currently in as a pair of two things - a set of all the nodes we have visited and the last node we have visited. Note that this way of bookkeeping saves us a lot of space and time - the different states we can be in like this is $O(2^n * n)$, where the first part comes from all the possible subsets of nodes and the second part comes from all the possible ways to choose the second node. This is a large improvement compared to number of states if state were a list of nodes visited so far - that would add up to $O(n!)$.

Speaking about the algorithm, from one state we can transition to $O(n)$ new states, making the complexity of this algorithm $O(n^2 * 2^n)$. The space complexity is $O(n * 2^n)$. This complexity is still exponential, but it is much faster than previous algorithm. Just for the sake of clarity, if we want our algorithm to finish in couple of seconds on an average PC, we can run *bruteforce algorithm* for $n$ up to 12, where we can run our dynamic programming algorithm for $n$ up to 28 [1].

We can represent a set of nodes using a **bitmask**. The bitmask is a number in the interval $[0, 2^n)$ which represents a unique subset of the graph nodes. A subset corresponding to the bitmask $msk$ contains node $x$ if and only if $msk$ has $x$-th bit set. With this mapping, we mapped all the subsets to integers allowing us to have a dynamic programming matrix instead of having to keep track of sets of nodes in some different way.

The dynamic programming base case is

$$dp[2^i][i] = 0$$

and we can make the transition between states as

$$dp[S][p] = \min_{i \in S, i \neq p} dp[S - \{p\}][i] + cost(i \rightarrow p)^2$$

### 3.1.3 Branch and bound algorithm

Here, we come to the first algorithm which running time actually depends on the input data and not only the input size! From now on, all the algorithms involved will have their running time impacted by the actual input, apart from the approximation algorithms. Branch and bound algorithms are, in general, basically 'smart' bruteforce algorithms in which we don't explore a certain path if we know that there is no chance that we can find an optimal solution continuing along that path. Because this is, after all, a bruteforce algorithm, it's worst case

[1]These numbers are based on my implementation of algorithms ran on my PC

[2]by rpr. I mean representing

running time has $n!$ as one of the factors, but we hope and that will empirically be shown that this algorithm actually finishes much faster than the bruteforce algorithms in most cases. Also, which might be not completely intuitive, this algorithms is also better than the Held-Karp algorithm on average, being able to solve some TSP instances with $n = 40$ in couple of seconds.

First, what we want to do is to compute some lower bound of the cheapest cycle, and ideally we would like to compute that fast. For the sake of clarity, we can say that our graph is complete - if there is no edge between two vertices, we can assume that the edge exists, having value $\infty$. Now, we can take a look at $n \times n$ distance matrix $M$ of the graph, where $M_{i,j} = w(i \to j)$. Also, the assumption is that the graph doesn't have any self-loops, so $M_{i,i} = \infty$ for every $i$.

Let's consider one row of the matrix. If we have picked $v$-th row, that row is consisted of weights of all the outgoing edges from vertex $v$. In the optimal solution, at least one edge has to be a edge that is outgoing from $v$. So, if $m_i = \min(M_{i,1}, M_{i,2}, \ldots, M_{i,n})$, we are sure that the edge going out from $v$ is going to contribute to the solution by some value which is greater or equal to $m_i$. Hence, we can conclude that the cost of the optimal solution must be at least $min\_cost = \sum_{i=1}^{n} M_i$. After having that calculated, we can deduct $m_i$ from $M_{i,\cdot}$ and get $M'$. It's easy to see that $min\_cost + tsp\_cost(M') = tsp\_cost(M)$.

We can perform the same procedure on the colums of the new matrix (repeating the same argument for incoming edges of each vertex), and adding the cost we got here to the row-obtained cost. That value is definitely a lower bound for the solution, and hypothetically it can be equal to the optimal TSP cost, in case there is a cycle with value 0 in the newly formed distance matrix.

Finally, the idea of the branch and bound algorithm here is that we will discard all the states where $current\_cost + lower\_bound\_remaining(current\_state) \geq best\_solution\_so\_far$. This termination condition is not the strongest one we can find in polynomial time, but it works quite well. In evaluation chapter, we will see some numbers about number of explored states using this algorithm.

Concerning the evaluation part, the decision is to always open the node with the minimum current cost. A data structure that allows me to do so efficiently is a priority queue, which is implemented in the standard library. The data structure representing the current state of an algorithm is a triplet of current distance matrix state, current cost and the current path we have taken. I also defined a custom comparator which I can pass to the priority queue as one of the initialization arguments, trying to utilise everything C++ allows the programmer.

## 3.2 Approximation algorithms

### 3.2.1 2-approximation algorithm

If we try to think of a something in the structure which is impacted by all the nodes, have it's cost minimal and is easy to calculate, we remember that there is a subgraph of the graph called a **minimum spanning tree**. That is a subgraph of the graph which is a tree[3], with vertex set equal to the vertex set of the original graph, with minimal sum of the weights of all the edges.

Our 2-approximation algorithm runs as follows:

1. Pick a start vertex *start* - it doesn't matter how we choose this one.
2. Construct a MST with 1 as root, using, for example, Prim's algorithm.
3. List all the vertices according to preorder walk, and add *start* to the end.

This gaves us a cycle of all $n$ vertices.

We will first state and prove two lemmas here.

**Lemma 1.** *(***MST Lemma***) The cost of a MST, in a graph with non-negative weights, is less than or equal to the cost of a minimum cycle.*

*Proof.* This is pretty straightforward, because the cost of a TSP is less than or equal to a cost of the TSP without a single edge, which is a spanning tree as well, so it's cost cannot be less to a cost of a minimum spanning tree.

□

**Lemma 2.** *(***Shortcutting lemma***) The shortcutting (skipping some nodes in the cycle) does not increase the cycle length in a metric TSP problem.*

*Proof.* We can prove that removing one node from the path cannot increase the path length. If there are some three consecutive nodes $a$, $b$, and $c$, and we are to remove node $b$ from the cycle, the length of the path prior to the removal of $b$ is $w(a \to b) + w(b \to c)$, and after removal the cost is $w(a \to c)$. Because the TSP is metric, the triangle inequality holds, we know that for every triplet of nodes $a$, $b$, and $c$, $w(a \to c) \leq w(a \to b) + w(b \to c)$, so the latter cost cannot be greater than the previous cost. Hence, removal of one node cannot increase the cost, so the removal of any number of nodes cannot increase the cost as well.

□

[3]A tree graph is a connected acyclic graph. A tree with $n$ nodes has $n - 1$ edges.

**Theorem 1** (2-approx theorem)**.** *For the metric TSP, the cost of the cycle obtained above is at most 2 times more expensive that the optimal cycle.*

*Proof.* Let's consider a full preorder walk of the graph - basically a preorder walk, but we list the vertex again when we return to it. For example, in the tree on figure 1, the preorder walk with $start = 1$ would be $[1, 2, 3, 2, 4, 2, 1, 5, 1]$. In order to prove this, we will separately show three facts:

- The cost of a full preorder walk is at most two times larger than the cost of MST - every edge of the MST is visited at most twice.
- The cost of a cycle generated by this algorithm is not greater than the cost of a full preorder walk - this is easily shown using the Lemma 2 proven above.

Using these two facts, as well as Lemma 1 we conclude that the theorem 1 holds.

□

The complexity of this algorithms is the same as the complexity of computing the minimum spanning tree. The maximum node of edges a $n$-node graph can have is $\frac{n(n-1)}{2}$, so I will consider $|E| = \mathcal{O}(n^2)$ in further analysis.

The algorithm I chose for the minimum spanning tree problem is Kruskal's algorithm. Implemented using the disjoint-set union (DSUCITAT) data structure, which runs in $\mathcal{O}(|E|\alpha(n)) = \mathcal{O}(n^2\alpha(n))$, where $\alpha(n)$ is the inverse of the single-valued Ackermann function, a function that grows extremely slowly and is less than 5 for any reasonable $n$.

### 3.2.2 Christofides algorithm

This algorithm gives us a 1.5 approximation to the metric TSP. It was developed by Nicos Christofides in 1976[4], and as of 2020 it is the best approximation ratio that has been proven for the TSP on general metric spaces, although there are significantly better approximation algorithms for some special cases (e.g. Euclidean TSP).

The algorithm runs as follows:

- Let $G$ be the starting graph. Find a MST of $G$, call it $T$.
- Create a subraph of $G$, $O$, consisting of vertices with odd degree in $T$ and all the edges between them.

- Let $M$ be the minimum-weight perfect matching in $O$. Let $T'$ be the graph with all the nodes and all the edges from both $T$ and $M$.
- Construct any Eulerian cycle on $T'$, and reduce it to a Hamiltonian circuit by skipping repeated nodes. Return that Hamiltonian cycle.

**Theorem 2** (Christofides theorem). *For metric TSP, the cost of the cycle obtained above is at most 1.5 times larger than the cost of the optimal cycle.*

*Proof.* Let $S$ be the optimal solution to the TSP - a cycle with minimum cost. Now, we can remove all the nodes not in $O$ from this cycle, maintaning the relative order of all the others node - giving us $S'$. Using the Lemma 2, $cost(S') \leq cost(S)$. We now have a cycle with even number of nodes. We can show this using a famous handshaking lemma - the sum of degrees of all the nodes in the graph has to be even, because that sum is equal to doubled number of edges.

Let's enumerate the nodes of $S'$ going around the cycle, from 1 do $m$, where $m$ is the number of nodes in $O$. We can now split $S'$ into two sets of edges, $E_1$ and $E_2$, where $E_1$ contains all the nodes having first node (going counter-clockwise, for example) odd, and $E_2$ all the others. Those two sets are definitely two perfect matchings on the nodes of $O$, so $cost(E_1) \geq cost(M)$ and $cost(E_2) \geq cost(M)$ holds, where $M$ is the minimal perfect matching on the nodes of $O$. Also, we know that $cost(S) \geq cost(S') = cost(E_1) + cost(E_2)$, so by symmetry argument we can conclude that $\min(cost(E_1), cost(E_2)) \leq \frac{1}{2}cost(S)$. That implies $cost(M) \leq \frac{1}{2}cost(S)$. Cost of all the edges in the union of $M$ and $T$ is then at most $\frac{3}{2}cost(S)$, and the solution we are getting has to be at most $\frac{3}{2}cost(S)$, using the Lemma 2 one final time.

□

The same MST computation technique is used again - the Kruskal's algorithm. The big challenge we are facing is the computation of the minimum weight perfect matching. An algorithm solving this problem is the Blossom algorithms, developed by Jack Edmonds in 1965CITE. This algorithm runs in $\mathcal{O}(|E||V|^2) = \mathcal{O}(n^4)$ time complexity, and that is the computation part taking the most time. We can find the Euler tour using HierholzerCITE's algorithm in linear time, so the total complexity of Christofides algorithm is $\mathcal{O}(n^4)$.

A variation of this algorithm, with the minimum weight perfect matching approximated by greedy heuristic has been fully implemented by me. The LEMON library has been used to produce the minimum weight perfect matching to be used in the Christofides algorithm in this project.

## 3.3 Improvement algorithms

In each of these algorithms, we need to establish what we are going to use as a starting point - the initial cycle. The difference in the result produced when starting from totally random cycle and the output of some of the other algorithms implemented in this work is not significant - however, the running time of the randomised version (explained below) is greatly reduced when starting from a good results in the first place. I am using the output of nearest neighbour starting point as an entry point for all the algorithms in this section.

### 3.3.1 $k$-opt algorithm

The idea behind $k$-opt heuristic, for given $k$, is as follows: until convergence or until some other stopping condition, cut the cycle in $k$ chains, and try to rearrange these chains to obtain the least expensive cycle. Note that there is a significant tradeoff between running time for one iteration (which is $\mathcal{O}(\backslash\|!))$ and the improvement we can get (more small parts we cut it into, we have greater probability of being able to improve it). In a degenerate case, picking $k = n$ would yield a optimal solution after one iteration, but that iteration would take the same amount of time bruteforce does.

In practice, the $k$-opt algorithm is implemented in that form for $k = 2$ and $k = 3$[13]. For $k > 3$, some additional heuristics are used in order to prioritise some $k$-opt moves over another, because the computational complexity of trying all the moves is too high. This is why only 2-opt and 3-opt heuristics are implemented in this algorithm, and some additional algorithms to test would be a variations of $k$-opt for $k \geq 4$.

For both 2-opt and 3-opt versions, two different variations are implemented - a randomised and a deterministic one. In the randomised implementation, each iteration step consists of picking random indices at which we can cut our cycle, and then trying all the combinations to see whether we can improve the solution. Opposed to that, in the deterministic one, an iteration consists of trying all the possible cuts, making the complexity of one iteration for 2-opt and 3-opt $\mathcal{O}(n^2)$ and $\mathcal{O}(n^3)$ respectively. Later, we will see how these versions compare versus each other. A thing to node is that we can run the randomised version multiple times and take the best result obtained in any of the runs, while the deterministic version always produces the same result if the starting point is the same. Note that, if the procedure generating the initial cycle is not deterministic we can take the minimum of several runs in both versions of the algorithm.

### 3.3.2 Lin-Kernighan heuristic algorithm

The problem with the performance of k-opt is that we are setting the value of $k$ in advance. Lin-Kernighan algorithms manages to generalise of k-opt, at each iteration trying to pick the best value of $k$ to do the interchange. It is one of the most successful TSP heuristic today, and it is a heuristic giving the best results in my implementation.

The algorithm tries to find two sets $X$ (edges to remove) and $Y$ (edges to insert) such that they respect the following criteria: the exchange is sequential, forcing the latter node of $X_i$ to be the former node of $Y_i$, the running sum $G = \sum_i^{|X|} w(X_i) - w(Y_i)$ has to be positive (meaning each step of the algorithm improves the tour), and two sets have to be disjoint. The described procedure is running until we can find such sets to improve the solution. This project implements one of many variations of the Lin-Kernighan heuristic, first introduced by K. Helsgaun[10].

## 3.4 Heuristic algorithms

### 3.4.1 Random algorithm

This algorithm is not really something one would try in order to solve the TSP, but it is included in the project so we would get an idea of how good other algorithms really are. We will just take the best solution out of some predefined number of random cycles.

We can use the inbuilt random shuffle function (random_shuffle) in C++ to generate random solution, compute the weights of paths and take the minumum afterwards. The complexity is $\mathcal{O}(IT \times n)$, where $IT$ denotes the number of paths we will take the minimum of.

### 3.4.2 Nearest neighbour (greedy) - algorithm

The nearest neighbour greedy algorithm is probably the first one to come up to someone's mind when tackling this problem. Also, it doesn't require knowing anything about the whole graph when making the decision, only about the surroundings, so it is not strange that an actual salesman from the problem name would use it if necessary.

The idea is, being at a certain node, take the edge with minimum weight that takes us to some node which is not already visited. We constantly need to keep track of which nodes we have visited, take a look at all the edges incident to the node and take the minumum one. After the arrival to the last unvisited node, we have to return to the node from which we started - we don't know anything about the weight of this edge at this point. The implementation is pretty straightforward and it's complexity is $\mathcal{O}(n^2)$.

We can pick any node as a starting node, so we can take the minimum when starting from all the nodes, paying a factor of $n$ in the time complexity.

The additional memory complexity remains $\mathcal{O}(n)$.

**Theorem 3.** *If $a_c$ is the cost of the tour generated by this algorithm and $o_c$ is the cost of the optimal solution, $\frac{a_c}{o_c} \leq \frac{1}{2}(\lceil \log(n) + 1 \rceil)$.*

We won't prove this theorem here, but the proof can be found here[15].

### 3.4.3 Insertion heuristics

With all the insertion heuristics, the recipe is the same: starting from a single node with a self-loop (an one node cycle), we are searching for an optimal node and an optimal spot. Having found that, we insert the node at a proposed place, and keep going until we insert all the nodes, resulting in a solution. Also, starting from different nodes might lead to a different solution, so we can try all the possible starts. All the heuristic can be implemented in quadratic time apart from the cheapest insertion, where one extra logarithmic factor is a must.

For all these heuristics, we need to define one very useful function: the value of inserting a node $b$ between nodes $a$ and $c$. We will define that as

$$insert\_cost(a, b, c) = w(a \to b) + w(b \to c) - w(a \to c)^4$$

Insert code of three nodes is the value by which the cost of the cycle increases when adding the node.

We can calculate the insert cost in constant time, so there is no need of precalculating anything.

If our current cycle is $v_1$, $v_2$, $\ldots$, $v_l$, $v_1$, and we are to insert node $u$ in that cycle, we will always insert the node after node

$$i = \arg\max_{i=1}^{l} insert\_cost(v_i, u, v_{i+1})^5$$

We further define $min\_insert\_cost(T, u)$ as the minimal cost needed to insert $u$ at any position in $T$.

[4] $w(a \to a) = 0$ for every $a$

[5] In this expression, $l + 1 = 1$ (the addition wraps around)

In each of the algorithms, we start with a one-node cycle in the beginning, and then iteratively add all the other nodes. We can choose that node arbitratily, but what I do is calculate the solution for all the possible starting nodes and output the minimum amongst those solutions.

There are multiple variations here, and the ones I have implemented can be found below:

### Cheapest insertion heuristic

Here, we calculate the optimal insertion cost for all the nodes which are not currently in the cycle, and we do insert the node with minimal value on it's optimal place. We continue this process until all the nodes are a part of the cycle - then we can return that cycle as a solution. With some extensive usage of data structures (C++ standard library implementations are used), we can implement this in $\mathcal{O}(n^2 \log n)$ time.

### Nearest insertion heuristic

This heuristic is a bit different from the previous one - at each step of the iteration, we pick a node which is closest to the current tour. We define a distance between a tour and a node as the minimum distance between any of the tour's nodes and the node given. Then, we insert that node at it's optimal position.

We will first prove a lemma that will help us prove the approximation ratio of these two heuristics.

**Lemma 3.** *(***Insertion lemma***) Let $A$ and $B$ be two sets of nodes. Define $min\_edge(A, B) = \min\{w(a, b) | a \in A, b \in B\}$. In our insertion heuristic, denote by $T_i$ the tour we have after inserting exactly $i$ nodes, and by $v_{i+1}$ the node we are inserting as $(i + 1)$-th node. If we have an algorithm where*

$$min\_insert\_cost(T_i, v_{i+1}) \leq 2 \times min\_edge(T_i, V - T_i)^6$$

*holds for each $i \in \{1, \ldots, n - 1\}$, then*

$$cost(output) \leq 2 \times cost(MST)$$

*where output is the output of the algorithm satisfying the above criteria, and MST being the minimum spanning tree of the graph.*

[6] V is the set of all the nodes, $-$ sign is just the set difference