Vladimir Milenković

The Travelling Salesman problem - Comparison of different approaches

Diploma in Computer Science

Trinity College

26th March 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 one of the most famous problems ever to exist - 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. Also, the

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

Blossom algorithm for minimum weight perfect matching is hard to get working.

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

My project tries to tackle one of the most famous problems in the history of Computer science - the Travelling Salesman 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 interesting in finding a minimum-weight cycle in this graph. 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 that there is no polynomial time algorithm solving this problem. With that in mind, we are forced to release some requirements of our algorithm: not demanding that we get a fully correct solution, only sufficiently close to the solution, and/or not demanding to finish in polynomial time, but sufficient fast. 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 Algorithms

In this section, I will list all the algorithms that I have taken into consideration for this project. I have made this choice after analyzing already existings benchhmarks 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**

1.2.1 Exact algorithms

Exact algorithms are those algorithms which produce the correct solution every time they are run. Knowing the $\mathcal{NP}$-hardness of the TSP, we know that there is no algorithm of this kind running in polynomial time. I have considered 3 algorithms here:

- *Brute-force algorithm*
- *Held-Karp (dynamic programming) algorithm*
- *Branch-and-Bound algorithm*

1.2.2 Approximation algorithms

Approximation algorithms are ones that can guarantee some bound concerning the output of the algorithm. For example, the *2-approximation algorithm* guarantees that the cost of the cycle it returned is at most two times larger than the cost of the optimal cycle. They usually run in polynomial time, and all the algorithms I've included in the project do belong in that category.

- *2-approximation algorithm*
- *Christofides-greedy algorithm*
- *Christofides algorithm*

1.2.3 Improvement algorithms

Improvement algorithms are those algorithm which consecutively try to improve an existing solution. Starting with some solution (greedy, for example), they try to alter it in a way that a better solution is produced. Running this algorithm for some time will eventually make the algorithm converge, getting a solution which we take as a final one. In practice, combining improvement techniques with some good strategies for the initial solution produce the best results, and a lot of best solutions on well-known TSP instances are obtained using improvement methods.

- *2-opt algorithm*
- *3-opt algorithm*
- *Lin-Kernighan algorithm*

1.2.4 Heuristic algorithms

Heuristic algorithms are algorithms which usually have polynomial running time, usually do not have a guaranteed bound of error, but in general they produce results that are close to the optimal solution. There are a lot of different heuristic for tackling this problem, so I have selected a few that did produce some solid results. Also, these algorithms should not be too hard to code or to understand, in general, and it is surprising how good results can they achieve:

- *Random algorithm*
- *Nearest neighbour algorithm (greedy)*
- *Insertion heuristic - cheapest, farthest, nearest, random*
- *Convex-hull heuristic algorithm*

1.2.5 Optimization algorithms

In **optimization algorithms**, we are starting with random solutions, computing the results those solutions are producing and then adapting towards solutions which are more preferable (have less cost in this example). Running this iteratively, we converge to a solution which can be pretty close to the correct one, no guarantees shown. There are also quite some algorithms which perform the process outlined above, here are some:

- *Ant-colony optimization algorithm*
- *Simulated annealing algorithm*

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 some lemmas and proofs about working performance of the procedures described.

2.1 Complexity analysis

I have already mentioned the fact that the Travelling Salesman Problem can not 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 in $\mathcal{NP}$-hard, using my previous $\mathcal{NP}$ complexity class definition. Also, for the formal proof, Karp TODO showed in 1972. 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.

2.2 TSP types

The TSP, as defined in the beginning of the dissertation, can be solved on any kind of graph. However, a lot of real TSP instances do obey some further criteria on the 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 somehow. In this section, I will introduce a couple of terms that will come in handy later.

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 graphs are symmetric, making this a very natural restriction on edge weights, so in most of the cases we will focus on this type of TSP.

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), $w(a \to$

$b) <= w(a \to c) + w(c \to b)$. 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].

2.3 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. As we have shown in the previous section, there can be no polynomial time algorithm belonging to this category. Furthermore, there is no algorithm with running time $\mathcal{O}(1.99999^n)$, where n is the number of vertices of the graph. That doesn't stop us in comparing different algorithms here, because we can clearly infer that one algorithm is better than the other in certain circumstances.

2.3.1 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 cycles. Number of cycles in a 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). 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 $O(n * n!)$ complexity.

2.3.2 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, ..., 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, ..., v_m), we can keep track of ($\{v_1, v_2, \ldots, v_m\}$, v_m), meaning we can describe the state we

[1]For Euclidean TSP, a PTAS (polynomial-time approximation scheme) algorithm can be applied, which finds the solution that is within a factor of $1 + \frac{1}{c}$ in $\mathcal{O}(n(\log n)^{(\mathcal{O}(c\sqrt{d}))^{d-1}})$, where d is the dimension of the Euclidean space.

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 $O(n^2 * 2^n)$. 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 [2].

2.3.3 Branch and bound algorithm

Here, we come to the first algorithm which running time actually depends on the input data. From now on, TODO almost all the algorithms involved will have their correctness and running time impacted by the actual input. 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 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 ∞. 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

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

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 $current\_cost + tsp\_approximation(current\_state) >= best\_solution\_so\_far$. This termination condition is not the strongest one we can find in polynomial time, but it works quite good. In evaluation chapter, we will see some numbers about number of explored states using this algorithm.

2.4 Approximation algorithms

We will not produce the optimal solutions here, 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.

2.4.1 2-approximation algorithm

If we try to think of a something in the graph which is impacted by all the nodes, have it's cost minimal and is easy to calculate, we remember that there is something 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 one lemma here.

Lemma 1. *The cost of a MST 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.

□

Theorem 1 (2-approx theorem)**.** *For 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 triangle inequality property[4] of the TSP, because two adjacent edges are constantly being replaced by one connecting non-adjacent ends when transitioning from a full preorder walk to the cycle above.

Using these two facts, as well as MST lemma TODO we conclude that the theorem 1 holds.

□

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

[4] This is the part that requires the TSP to be metric

2.4.2 Christofides algorithm

This algorithm gives us a 1.5 approximation to the metric TSP. It was developed by Nicos Christofides in 1976., and as of 2019 it is the best approximation ratio that has been proven for the TSP on general metric spaces, although there are some better approximation 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 PM 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 PM.
- 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. Christofides proof here.

□

2.5 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.

2.5.1 k-opt algorithm

The idea behind k-opt heuristic, for given k, is as follows: until convergence or until some other stopping point, 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 $\mathcal{O}(\|!)$) 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.

It is empirically validated that picking $k >= 4$ is worse than picking $k <= 3$. In my implementation, I have coded 2-opt and 3-opt separately TODO, and shown that 3-opt performs better than 2-opt.

2.5.2 Lin-Kernighan heuristic algorithm

The problem with k-opt is that we are setting the value of k in advance. Lin-Kernighan algorithms is basically a generalization 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 procedure runs as follows:

LKH here

2.6 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.

2.6.1 Random algorithm

This algorithm isn't 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.

2.6.2 NN (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.

When in a certain node, the algorithms chooses the edge with minimum cost leading to some, so far unvisited, node and traverses to there. After visiting all

the nodes, it returns to the node where the path has started. It is not hard to see that the solution depends on the choice of the starting node, hence we can start from all the nodes and take the minimum solution which improves the solution with only a factor of n.

This algorithm can go horribly wrong, and we can construct a case where the solution obtained this way is arbitrarily times longer than the optimal solution. Still, it performs surprisingly fine on average.

2.6.3 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.

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

Cheapest insertion heuristic

Farthest insertion heuristic

Nearest insertion heuristic

Random insertion heuristic

Convex-hull insertion heuristic

2.7 Optimisation algorithms

2.7.1 Ant colony optimisation algorithm

Chapter 3

Implementation

Chapter 4

Evaluation

Chapter 5

Conclusion

Bibliography

[1] S.W. Moore. How to prepare a dissertation in latex, 1995.

Appendix A

Project Proposal

Vladimir Milenković
Trinity College
vm370@cam.ac.uk

Individual project

Computer Science Tripos, Part II

The Travelling Salesman problem - Comparison of different approaches

---

18 October 2019

Project Originators:
Dr Thomas Sauerwald
Vladimir Milenković

Project Supervisor:
Dr Thomas Sauerwald

Director of Studies:
Prof Frank Stajano
Dr Sean Holden

Project Overseers:
Dr Rafal Mantiuk
Prof Andrew Pitts

Introduction and Description of the Work

The Travelling Salesman problem is one of the most famous NP-hard problems. It has been the topic of many researches, and many famous mathematicians and computer scientists tried to approach it. This problem is occurring frequently even in real life - for example, we are willing to visit multiple cities and we know what the flight cost between each pair of cities is, can we make an cost-optimal route visiting all of them? Also, a good solution to this problem would have several applications: in planning, logistics, microchip manufacturing, etc..

The mathematical formulation of the problem follows: given an undirected weighted graph, find the Hamiltonian cycle with minimum weight, where we define the weight of a cycle as a sum of weights of all the edges involved.

As this problem has been and still is one of the most major 'unsolved' problems in Computer Science, there has been many different tries to find a solution which is good by some metric. Some of the approaches that were taken with more or less success are:

- Exact algorithms: the most straight forward brute-force approach can be done in $\mathcal{O}(n!)$ by simply trying all the possible permutations. There is a dynamic programming approach in $\mathcal{O}(2^n n^2)$, done by Held–Karp. There is no known algorithm which solves the TSP in $\mathcal{O}\big((2-\varepsilon)^n\big)$.
- Approximation algorithms: these are the algorithms that don't search for the optimal solution, but they find the solution that is guaranteed to be, for example, at most 2 times worse that the optimal one. Approximation algorithms nowadays can usually find the solution in 2-3% to the optimal solution within reasonable time. FPTAS (Fully Polynomial-Time Approximation Scheme) is one of the algorithms that I will implement.
- Heuristic algorithms: In this type of algorithms, there are no guarantees about the solution we're going to find whatsoever, we are trying to follow some 'sensible' heuristics in order to make our solution as good as possible.
- Genetic algorithms: which are the randomized algorithms in which we are simulating some processes observed in natural evolution.
- Neural networks: There are many papers about trying to approach the TSP using neural networks. There has been some success, mostly using

recurrent neural networks, but right now they are not outperforming other algorithms.

In this project, I will be implementing atleast 2 algorihms per each category stated above and comparing them to each other on different types of graphs. I will mainly focus on comparing their solution correctness, and comparing their running time. My language of choice of implementing the algorithm is C++, due to my already substantial experience with it. Also, one of the reasons is that C++ is performance efficient, thus my programs will be running a bit faster compared to some other options I could've taken. Also, in the neural network part of the project, I will probably be using Python, because of the existence of various machine learning libraries, such as TensorFlow/Keras.

Relevant courses

Concerning the courses I have already and will be taking at my Computer Science Tripos, here are the ones that my project will interfere with:

Algorithms Main part of this project is going to be actually coding the algorithms solving the problem. Both my previous algorithm experience and knowledges I've gotten during my Part IA Algorithms course will come in handy.

Artificial Intelligence I/II Since I'll be coding some heuristic algorithms, as well as doing some machine learning, things I've learned are going to be influential for the project.

Programming in C/C++ My main choice of language for this project is going to be C++.

Complexity Theory The problem this whole project is about is NP-hard, so a proof of that is going to be in the project.

Project Structure

This project intends to produce benchmarks and results about various algorithms and their behaviour on substantially different inputs.

The execution of the project can be split into 5 main objectives. In the project timetable given below, I will write down the times at which I'm planning to finish each of them. Objectives, in chronological order, can be found below:

- **Preparation** – this part of the project mainly consists of getting more familiar with the topic. My plan, in order to successfully finish this objective, is to read books and papers about algorithms solving the TSP and thinking about ways to implement them, trying to pick the most concise and the most effective implementation. To confirm that I've successfully finished this chapter, I'm planning to be able to prove the correctness, the complexity and to be completely familiar with every algorithm I'm going to implement in the project.

- **Implementation** – this part of the project consists of actually implementing the algorithms, as well as the generators of the data I'm going to test on and all the other necessary code in order to be able to efficiently evaluate the algorithms I've coded. By generators of the data, I mean both collecting the already existing data from the web (e.g. flight durations, road lengths, etc..), as well as generating my own random data with some tunable parameters (e.g. graph density).

- **Evaluation** – this objective consists of evaluating the algorithms on different examples, plotting the graphs of different characteristics I'm going to measure. For example: running time, space complexity, deviation from the optimal solution (if known), impact of varying algorithm parameters (if suitable). Here, I'm planning to use some of the well-known TSP datasets available online, in order to test my implementation versus already existing ones. One of the datasets I'm planning to use is the one available at University of Waterloo TSP page.

- **Extensions** – this objective consists of me trying to give my own optimizations of certain algorithms on different types of graphs, in order to get a better performance. One more thing I could add to this project is to try to solve it using linear programming, and to benchmark that approach versus already existing ones. I have several ideas about observing how do heuristic approaches behave when altering some parameters in the actual heuristic being used and plotting the outcomes. I'm planning to start doing Extensions and trying to achieve something new after completely finishing the core project.

- **Dissertation** – this part of the project consists of collecting all the code involved in the project, as well as writing the dissertation. In the dissertation, I would like to cleanly show how all of the above objectives have been accomplished in a chronological order.

Success Criteria

I will consider my project as a successfully completed one after achieving all of the above objectives apart from **Extensions** one. Several checkpoints I should accomplish at that point are:

- Well written and easily readable source code for all the algorithms should be provided at the end of the project, as well as the empirical validation that all the algorithms performed close to as one could predict based on the already existing knowledge.
- Scripts needed to compile and benchmark the project.
- Plots about algorithms performance, after being evaluated multiple times on various test samples, both generated and downloaded.
- The dissertation, in which my process of accomplishing all the objectives should be clearly explained, as well as the conclusions I've achieved after being able to evaluate different approaches and some directions for future work on this topic.

I hope that **Extensions** objective will also be achieved before the project submission deadline, however it is not the core part of the project and shouldn't be essential for its success.

Resources

I'm planning to use my own laptop (Dell XPS 9570, 16 GB RAM, Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz, running GNU/Linux). I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure. My contingency plans are listed below:

- I will be using GitHub as the projects main backup tool, creating a private repository and storing all the data necessary for the project, including the dissertation, in that repository.

- I will also backup all the data at my PC on daily basis, on my own external HDD.

I'm pretty sure that no other resources are required in order to complete my project.

Project timetable

My high level milestones to be achieved are: getting prepared and studying for the project until the end of November, actually implementing the project until the end of January, finishing the evaluation part by the end of February, and writing the main part of the dissertation by the end of March. If everything works as expected, I'm planning to to my Extensions objective in April, trying further to improve the project. I expect the submission by the end of the April, so I don't get too close to the deadline, which is a situation in which nobody would like to be.

Concerning a bit more detailed timetable, I will divide the time from today until the end of the project in 15 two-weeks periods, and I will list the small objectives I will aim to achieve in each of them.

Part 1: *18 October – 31 November*

→ Writing and refining the project proposal.

- **DEADLINE:** Project proposal (25 October)

→ Creating the GitHub repository and setuping the software and hardware needed to achieve my contingency plans.

Part 2: *1 November – 14 November*

→ Getting more familiar with heuristic algorithms in general.

→ Reading the books and the papers about existing TSP algorithms and deeply studying the subject.

Part 3: *15 November – 28 November*

→ Efficient implementation and testing of exact TSP algorithm.

→ Coding approximation algorithms (Christofides' algorithm being one of them).

Part 4: *29 November – 12 December*

→ Testing the approximation algorithms

→ Starting to work on different heuristic approaches.

Part 5: *13 December – 26 December*

→ Further working on heuristic algorithm. I want to make sure that all the parameters of the heuristic are going to be easily tunable, in order for Extensions to be done without much problem.

Part 6: *27 December – 8 January*

→ Testing heuristic algorithms.

→ Reading more papers about machine learning approach to TSP, here focusing mainly on the implementation aspects.

Part 7: *9 January – 22 January*

→ Implementing some of the papers with the idea from above.

→ Testing and being able to produce the same results as in the paper.

Part 8: *23 January – 4 February*

- **DEADLINE:** Progress report submission (31 January)

→ Buffer slot to finish any unfinished work from before.

→ Writing the project progress report and getting ready for the progress report presentation.

Part 9: *5 February – 18 February*

→ Writing the generators for the data I'm going to test my various algorithms on.

→ Writing the code that's going to link my algorithms with the generators, allowing easy testing and evaluation part.

Part 10: *19 February – 4 March*

→ At this point, if everything above is done correctly, the process of evaluating and producing the plots shouldn't be too big of a deal. Generating suitable plots for the dissertation.

→ Finishing if any of previous objectives isn't already achieved.

Part 11: *5 March – 18 March*

→ Starting to work on the project dissertation, writing the introduction chapter.

Part 12: *19 March – 1 April*

→ First draft of the dissertation.

Part 13: *2 April – 15 April*

→ Further refining the dissertation, listening to the comments from both my supervisor and DOSes.

→ Start working on Extensions topics.

Part 14: *16 April – 29 April*

→ Finishing my dissertation and making it ready for submission.

→ Further working on Extensions, and incorporating it in the dissertation if successful.

Milestone: Final version of the dissertation ready for submission.

Part 15: *30 April – 8 May*

→ Dissertation submission.

- **DEADLINE:** Dissertation (8 May)