Vladimir Milenković

The Travelling Salesman problem - Comparison of different approaches

Diploma in Computer Science

Trinity College

20th 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[9]. 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 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)[5] - 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, 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.

2.5 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[1] (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	$\mathcal{O}(n^2 2^n)$	[2], [7]
Branch and bound algorithm	Exact	$\mathcal{O}(n!)$	folklore
2-approximation	Approximation	$\mathcal{O}(n^2)$	TODO
Christofides	Approximation	$\mathcal{O}(n^3)$	[3]
k-opt	Improvement	$O(n^k)$[3]	[6], [4]
Lin-Kernighan	Improvement	$O(n^{2.2})$	[9]
Random algorithm	Heuristic	$\mathcal{O}(n)$	folklore
Nearest neighbour heuristic	Heuristic	$\mathcal{O}(n^2)$	folklore
Cheapest insertion heuristic	Heuristic	$\mathcal{O}(n^2 \log n)$	[10]
Farthest insertion heuristic	Heuristic	$\mathcal{O}(n^2 \log n)$	[10]
Nearest insertion heuristic	Heuristic	$\mathcal{O}(n^2 \log n)$	[10]
Random insertion heuristic	Heuristic	$\mathcal{O}(n^2 \log n)$	[10]
Convex-hull insertion heuristic	Heuristic	$\mathcal{O}(n^2 \log n)$	TODO
Ant colony optimization	Optimisation	$\mathcal{O}(cnt\_ants \cdot n^2)$	TODO

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 $\mathcal{O}^*(1.99999^n)$[11], 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

$$dp[msk \text{ rpr. } S][p] = \min_{i \in S, i \neq p} dp[msk \oplus 2^p \text{ rpr. } S - \{p\}][i] + cost(i \to p)^2$$

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 ∞. 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 $M_{i,.}$ 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 MST lemma 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[3], 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 shortcutting lemma, $cost(S') \leq cost(S)$. 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.

It is empirically validated that picking $k \geq 4$ is worse than picking $k \leq 3$, the so I have implemented 2-opt and 3-opt versions of k-opt algorithm.

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[8].

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.

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, ..., 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.

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:

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

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

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.

Proof. □

Theorem 4. *Both cheapest insertion heuristic and nearest insertion heuristic on a metric TSP provide a 2-approximation to the TSP.*

Proof. We will show this using insertion lemma, by showing that both cheapest and nearest insertion heuristics obey the criteria which is necessary for lemma to be applied.

Let T be the current tour created. Let the minimum weight edge among all the edges stretching between T and $V - T$ be $a \leftrightarrow c$, where $a \in T$. Let b be any

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

neighbour of a in T. We can easily show that $insert\_cost(a, b, c) \leq 2 \times w(a, c)$ using the triangle inequality.

Now, in cheapest insertion heuristic the insert cost has to be at most $insert\_cost(a, b, c)$, so the lemma criterium holds. We can easily conclude the same thing for nearest insertion heuristic as well.

With this shown, we can apply the previously mentioned lemma to prove the theorem. Remember the MST lemma.

□

Farthest insertion heuristic

In this heuristic, we are doing something which might sound a bit strange, however, it can perform better than some of the above on some TSP instances. Instead of choosing the closest node to the tour, we are choosing the farthest one, inserting it first. This makes sense because we are interested in optimally positioning the 'far-away' nodes in the beginning - finding a better placement to outlier nodes contribute to the solution more than the optimal placement of nodes which are next to each other. This is a handwavy explanation, but we will see its good performance in the evaluation chapter. We can prove that the ratio between this solution and the optimal one is less than the constant times the logarithmic function with number of nodes as it's argument. This heuristic gives a logarithmic approximation, meaning that the ratio of the solution obtained and the optimal one is less than the constant times the logarithm of n.

Random insertion heuristic

Instead of choosing the node using some heuristic - we can choose the next node to insert arbitrarily. After choosing it, we insert it to it's optimal place in the tour. This improves the time needed for the algorithm to finish, and actually doesn't decrease the performance a lot. This gives the logarithmic approximation as well.

Convex-hull insertion heuristic

When dealing with **euclidean TSP**, we have some further facts to utilise.

Lemma 4. *In Euclidean TSP, the optimal cycle does not intersect itself.*

Proof. Let $a \to c$ and $d \to b$ (with a and b being in same connected component after removing those two edges, WLOG) be the edges in the optimal cycle intersecting each other. By the triangle inequality, which holds from the fact that the TSP is euclidean, we can easily show that removing thoes edges and adding

$a \to d$ and $c \to b$ improves the solution, making the assumption that the cycle is optimal false. Hence, the optimal cycle cannot intersect itself.

□

Then, we can conclude that all the points on the convex hull of all the points are going to be in same order both in TSP optimal solution as well as in the convex hull. We can create the convex hull using the Graham scanCITE algorithm in $\mathcal{O}(n \log n)$ time, and then use any of the previous heuristics to add all the other nodes. The cheapest insertion heuristic was chosen here.

3.5 Local search algorithms

3.5.1 Ant colony optimisation algorithm

In general, the **ant colony optimization** algorithm (**ACO**) is a method of solving various graph problems, most of them about finding a path that is good evaluated on some metric. The TSP is no expection - we can easily fit its formulation into the general framework for the ACO.

This algorithm is inspired by the behaviour of real ants and their pheromone-based communication. An ant, initially walking randomly, will mark the paths that are better than the other by leaving some amount of pheromone on its path. Future ants will be guided by the pheromone smell so more ants will be attracted to good paths, though there is still a posibility of going in all the directions so all the paths will be investigated. This converges at some point, giving us a solution that can be pretty close to the optimal one.

The algorithm resembles the ant behaviour, leaving 'pheromone' on all the edges, where the edge distance is one of the factors in choosing the next edge. Releasing a large number of ants will lead us to the convergence. Implementation-wise, we can have a matrix with dimension $n \times n$ where we can store the concentration on pheromone on every edge. When an ant is present in some node, it chooses the edge to take among all the unvisited nodes respecting the probability mass distribution, which is a function of the length of the path and the pheromone concentration. We define some hyperparameters in order to weigh the strength of pheromone versus the path length.

There are several iterations of the algorithm - in one, we make a number of ant cycles and calculate the weights of their cycles. Then, we change the pheromone matrix based on all the cycles obtained in this iteration, and move on. We don't update after every ant finishes it's trajectory in order to stabilize the algorithm, making it less prune to failing if some bad paths get chosen in

the beginning. We also need to define a starting point, and we can either limit the number of iterations/ants to run or we can stop when the length of the cycle stops decreasing.

Chapter 4

Evaluation

4.1 A priori knowledge

Prior to starting the evaluation part, I surely had some thoughts about which algorithm is going to outperform all the others, speaking about some fixed data set. In the later sections of this chapter, the actaul results tell us whether the expectations turned out to be right. Below is an overview of several facts we will test onwards.

- Focusing on the exact algorithms, the dynamic programming approach does always perform better than the bruteforce approach. However, even knowing the fact that the worst case complexity of branch-and-bound algorithms is the same as the bruteforce approach, I expect it to outperform, time-wise, the dynamic programming algorithm, on both generated and real-world data.

- I expect the randomized implementation of k-opt heuristic to outperform the deterministic one, because the wider set of cycles gets its chance to be a stage of the improvement. However, there is no easy way to estimate the running time of the randomized k-opt before it converges, so that will be further tested in this chapter.

- The algorithms of Christofides (both with greedy matching and with optimal one) are expected to outperform the 2-approximation algorithm on average and I would be really surprised if that doesn't show true, even though that doesn't have to be the case. Also, it is expected that the Christofides implementation with calculating the minimum weight perfect matching outperforms its greedy heuristic approximation.

- I am expecting the Convex hull heuristic to find cycles significantly better than the other heuristic on the TSP instances with Euclidean metric - the convex hull heuristic uses the fact that the points in the optimal solution have the same order as in the convex hull, thus starting the greedy insertion from a much better starting point. Also, I expect its performance to raise as the percentage of the points in the set belong to its convex hull.

- I don't expect other insertion heuristic(s) to vary a lot. The farthest insertion heuristic is a bit favourised even though it does the insertion 'in the strangest' way, but a good starting point will be established. I think, even though this is not tested, that a hybrid algorithm starting with farthest insertions and continuing with cheapest/nearest insertion could outperform all of the insertion heuristics implemented here, though the improvement wouldn't be any large.

- Finally, starting from a solid starting point (nearest neighbour algorithm, in this case), the Lin-Kernighan heuristic will obtain the best results out of all the non-exact algorithms implemented in this project. Nowadays, the best results are obtained by procedures having the variation of this algorithm - chained LKHCITE as the starting point for the further improvement. The solver that managed to successfully solve all the instances in the TSPLIB archive, the ConcordeCITE solver, is no exception. It uses a smarter version of branch-and-bound algorithms described in this dissertation after nicely bounding the solution using the abovementioned heuristic.

4.2 Data sets

After collecting the data from the TSPLIB as well as generating my own data, there are several categories each of the tests belongs to. The plots, starting from the following section will be clearly marked to which category do they belong, and the main goal of the project is to see which algorithms gets advantage of a test belonging to some category.

First, all the data has been split in two disjoint sets - real data, collected from the TSPLIB, and synthetic data, generated by me. Data generation techniques are explained in this writeup, as well.

I will focus primarily on the symmetric TSP in this work, as the asymmetric TSP is not commonly seen and much less effort has been put in developing heuristic for solving it. Nonetheless, some of the algorithms do perform equally

good when placed in the asymmetric world, and that will be mentioned where suitable.

The test cases will be split in metric/non-metric group as well, however more attention will be paid to metric TSP because that is the scenario that is seen more often. Also, a subset of metric TSP, Euclidean TSP, will be separated, knowing that we expect some algorithms to run better on it.

Also, the cases will also be split in several size categories, by the number of the nodes of the graph. I will consider all the graphs to be complete ones, because we can turn any non-complete to a complete ones by adding a infinite-weighted edge between each two nodes not originally connected.

4.3 Evaluation procedure

All the evaluation procedures have been run on my machine, previously described hereTODO.

Unless specified, all the numbers in any of the tables are average of exactly 5 runs of the algorithms. Also, some thresholds on the running time and memory consumption have to be established - the verdict of the execution will be described as Time Limit Exceeded (**TLE**) if the execution takes more than 5 minutes. Also, if the memory the program request the amount of memory unable to run on my computer previously described, the verdict will be characterised as Memory Limit Exceeded (**MLE**).

4.4 CustomTODO comparisons

4.4.1 Exact algorithm comparison

Algorithm	Average time complexity	Worst-case time complexity	Memo
Bruteforce	$\mathcal{O}(n!)$	$\mathcal{O}(n!)$	
Held-Karp algorithm	$\mathcal{O}(n^2 2^n)$	$\mathcal{O}(n^2 2^n)$	
Branch and bound algorithm	$\mathcal{O}(n!)$[1]	$\mathcal{O}(n!)$	

Table 4.1: Exact algorithms

All of these algorithms do find the optimal solution to the TSP when they finish, so we will compare their time complexities in this subsection. These algorithms are not affected by the fact whether the TSP instance is real/artificial

or whether is symmetric or not, hence I will compare them on a set of randomly generated symmetric testcases.

The results can be found in following figure[3]:

Algorithm	sym5	sym8	sym10	sym12	sym13	sym16	sym20	sym22	sym25
Bruteforce	1ms	12ms	1s	12s	157s	**TLE**	**TLE**	**TLE**	**TLE**
Held-Karp	1ms	1ms	1ms	7ms	14ms	139ms	3s	17s	**MLE**
Branch and bound	1ms	5ms	7ms	35ms	91ms	308ms	48s	75s	**MLE**

Table 4.2: Exact algorithm comparison

A modification that has been tried is to initially bound the solution of branch and bound with some good approximation. Christofides and 3-opt were tried, but the performance gains for the biggest testcases this algorithm manages to solve were negligible, so that hybrid is omitted.

As we can see, the branch and bound algorithm never managed to outperform the dynamic programming approach on totally randomly generated test cases. However, there definitely are some examples of instances where the branch and bound clearly wins against the Held-Karp algorithm. An example of that are test cases bb_suitable50 and bb_suitable100 (both have exactly one cycle which is many times better than all the other cycles we can make). The branch and bound algorithm manages to find the correct solution in less than a second even for a 100 node graph, which is clearly undoable via any other exact algorithm.

4.4.2 k-opt implementation comparison

The algorithms here perform the same type of improvement step - given the cutting spots in the cycle, the procedures searches for the best way to rearrange the parts. Two variations we will consider here is going through all the pairs/triples for 2-opt/3-opt respectively in some order, or randomly choosing a suitable triplet on every iteration. We will compare the result obtained as well as the running time until it stops further improving.

The starting point for all of these algorithms is a nearest neighbour solution starting from a random node - just so we add some extra non-determinism everywhere.

For this comparison, I have chosen 4 TSPLIB instances (burma14, att48, ch130, a280), as well as 4 randomly generated symmetric instances (sym20,

[3]sym$n$ stands for a symmetric TSP instance with n nodes

Algorithm	Number of iterations	Memory complexity
2-opt random	n^2 without further improvement	$\mathcal{O}(n)$
2-opt deterministic	until no further improvement (at least n^2)	$\mathcal{O}(n)$
3-opt random	n^3 without further improvement	$\mathcal{O}(n)$
3-opt deterministic	until no further improvement (at least n^3)	$\mathcal{O}(n)$

Table 4.3: k-opt algorithms

sym50, sym100, sym300). The results can be found below (the table entries are ordered pairs of average solution, and the running time):

Algorithm	burma14	att48	ch130	a280
2opt-full	(3432.4, 1ms)	(11039.4, 1ms)	(6660, 3ms)	(2866.2, 12ms)
2opt-random	(3672, 1ms)	(11576.4, 1ms)	(6942.8, 14ms)	(2946.4, 63ms)
3opt-full	(3397.2, 1ms)	(10832.4, 110ms)	(6359.8, 1.9s)	(2708.8, 19s)
3opt-random	(3376.6, 1ms)	(10797, 252ms)	(6316, 5.7ms)	(2776.6, 59s)
optimal solution	3323	10628	6110	2579

Table 4.4: k-opt comparison - real

Algorithm	sym20	sym50	sym100	sym300
2opt-full	(2535, 1ms)	(2627.6, 1ms)	(2974.8, 2ms)	(3113.4, 20ms)
2opt-random	(3029.6, 1ms)	(2795.8, 1ms)	(3297.2, 6ms)	(3598, 59ms)
3opt-full	(2440.4, 6ms)	(2227.8, 96ms)	(2394.6, 900ms)	(2223.4, 63s)
3opt-random	(2462.6, 21ms)	(2198.8, 324ms)	(2312.4, 2.4s)	(2192, 122s)
optimal solution	3323	unknown ($\leq$ 2102)	unknown ($\leq$ 2246)	unknown ($\leq$ 214

Table 4.5: k-opt comparison - artificial

First thing we notice is that there is no visible difference between real and synthetically generated test cases when evaluating k-opt heuristic.

This has shown that 3-opt clearly outperforms 2-opt, no matter what variation of each we take into consideration. Examining the difference between the variations, we can notice that 2-opt deterministic works quite better than its randomised counterpart. However, if we would have taken the minimum value obtained in multiple runs, on a lot of occasions the randomised versions would win, but the variance is just too high. A good example of that are results of 2-opt randomised on sym20 test case: $(2334, 3177, 2531, 3447, 2480)$, where the minimum result (2480) is better than all the results obtained using the deterministic

version. Taking a look at the running time, 3-opt full and 3-opt random difference in accuracy is not extremely large while the running time gap is substantial in favour of full version, so there is a tradeoff between those two.

Taking these results into account, we will continue the comparison with 2-opt full as it is clearly dominant, and we will proceed with both version of 3-opt for the further evaluation.

4.4.3 Approximation algorithms comparison

We will see how our approximation algorithms compare versus each other, and how do the compare versus the optimal solution. Because we would like to have the optimal solution calculated for all instances, I am using TSPLIB instances with optimal solution attached to them.

The algorithms we are considering here are:

Algorithm	Approximation ratio	Time complexity	Memory complexity
2-approximation	2	$\mathcal{O}(n^2\alpha(n))$	$\mathcal{O}(n^2)$
Christofides-greedy	2^4	$\mathcal{O}(n^2\alpha(n))$	$\mathcal{O}(n^2)$
Christofides	$\frac{3}{2}$	$\mathcal{O}(n^3 \log(n))^5$	$\mathcal{O}(n^2)$

Table 4.6: Approximation algorithms

For evaluation, a set of 10 TSPLIB instances is used: burma14, gr21, berlin52, gr120, ch130, d198, pr226, a280, fl417, att532. All of these algorithms are deterministic, so only one run is enough. Also, they are all sufficiently fast and do not require a lot of memory, so the correctness is the metric in focus. Results are shown:

Algorithm	Approximation ratio
2-approximation	34.7%
Christofides-greedy	13.8%
Christofides	7.4%

Table 4.7: Approximation comparison

4.4.4 Insertion heuristics

First, the overview of all insertion heuristics implemented in this work is given. Note that this table gives the time complexity for starting from one starting node,

but for the sake of comparison the best solution of n runs will be included as a separate entry.

Algorithm	Time complexity	Approximation ratio
Random insertion heuristic	$\mathcal{O}(n^2)$	$\mathcal{O}(\log n)$
Cheapest insertion heuristic	$\mathcal{O}(n^2 \log n)$	2
Nearest insertion heuristic	$\mathcal{O}(n^2)$	2
Farthest insertion heuristic	$\mathcal{O}(n^2)$	$\mathcal{O}(\log n)$
Convex-hull cheapest insertion	$\mathcal{O}(n^2 \log n)$	2

Table 4.8: Approximation algorithms

All of these algorithm do not require some extensive memory usage and the time complexities are almost identical, so the closeness of the solution produced versus the optimal solution will be the point of focus once more. The instances used are 10 TSPLIB instances, same ones used as in approximate algorithms evaluation. We can also find the comparison between the heuristic which has shown the best results in the first evaluation versus the convex hull heuristic - but only on Euclidean TSP instances.

Algorithm	Approximation ratio
Random insertion heuristic[6]	8.8%
Random insertion heuristic (n runs)	3.4%
Cheapest insertion heuristic	15.6%
Cheapest insertion heuristic (n runs)	10.8%
Nearest insertion heuristic	19%
Nearest insertion heuristic (n runs)	14.8%
Farthest insertion heuristic	22.8%
Farthest insertion heuristic (n runs)	15.6%

Table 4.9: Insertion heuristics comparison

As we can see from the table, the random heuristic outperforms all the other, probably because it gets to explore much more solutions than the others. The random insertion heuristic ran n times produces the results that were not worse than 10% in any of the given tests, which is an outstanding result obtained by something that simple.

The following table depicts the comparison of random heuristic ran multiple times with or without previously calculating the convex hull. It will be tested on a set of 5 Euclidean TSP instances: berlin52, ch130, pr226, a280 and gr666.

Algorithm	Approximation ratio
Convex hull + random insertion	7.7%
Random insertion	9%
Convex hull + random insertion (n times)	3.1%
Random insertion (n times)	3.6%

Table 4.10: Convex hull heuristic comparison

As we can see from this table, the convex hull heuristic definitely improves the solution, which is completely sensible - we are starting with a cycle which is definitely a subsequence of optimal cycle.

4.5 Comparison of the best algorithms

In this section, I am presenting a cross-comparison of the algorithms that were the best in the previous comparisons, with addition of nearest-neighbour heuristic and the ant colony optimisation algorithm. The representative algorithm chosen for this final evaluation are shown in the table below.

Algorithm	Time complexity	Algorithm type
Held-Karp algorithm	$\mathcal{O}(n^2 \cdot 2^n)$	exact
3-opt full	$\mathcal{O}(n^3)$ per it.	improvement
Christofides algorithm	$\mathcal{O}(n^3 \log(n))$	approximation
Nearest neighbour heuristic	$\mathcal{O}(n^2)$	heuristic
Random insertion heuristic	$\mathcal{O}(n^2)$	heuristic
Lin-Kernighan heuristic	$\mathcal{O}(n^2.2)$	heuristic
Ant colony optimization	$\mathcal{O}(cnt\_ants \cdot n^2)$	local search

Table 4.11: Final comparison algorithms

These algorithm will be tested on a set of 10 TSPLIB instances, and their average on 10 runs, as well as the best tour obtained will be shown.

Algorithm	tc1	tc2	tc3	tc4	tc5	tc6	tc7	tc8	tc9	tc10	avg. appr. ratio
H-K algo 3-opt NN heur. RI heur. LK heur. ACO											
winner											
optimal solution											

Table 4.12: k-opt comparison - artificial

Chapter 5

Conclusion

Bibliography

[1] Sanjeev Arora. Polynomial time approximation schemes for euclidean traveling salesman and other geometric problems. *Journal of the ACM (JACM)*, 45(5):753–782, sep 1998.

[2] Richard Bellman. Dynamic programming treatment of the travelling salesman problem. *Journal of the ACM (JACM)*, 9(1):61–63, jan 1962.

[3] N. Christofides and Carnegie-Mellon University of Pittsburgh PA management sciences research group. *Worst-Case Analysis of a New Heuristic for the Travelling Salesman Problem*. Management sciences research report. Defense Technical Information Center, 1976.

[4] G. A. Croes. A method for solving traveling-salesman problems. *Operations Research*, 6(6):791–812, dec 1958.

[5] Balázs Dezső, Alpár JÃijttner, and Péter Kovács. LEMON – an open source c++ graph template library. *Electronic Notes in Theoretical Computer Science*, 264(5):23–45, jul 2011.

[6] Merrill M Flood. The traveling-salesman problem. *Operations research*, 4(1):61–75, 1956.

[7] Michael Held and Richard M. Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics*, 10(1):196–210, mar 1962.

[8] Keld Helsgaun. An effective implementation of the lin–kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126(1):106–130, oct 2000.

[9] Shen Lin and Brian W Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations research*, 21(2):498–516, 1973.

[10] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis. Approximate algorithms for the traveling salesperson problem. In *15th Annual Symposium on Switching and Automata Theory (swat 1974)*. IEEE, oct 1974.

[11] Gerhard J. Woeginger. *Exact Algorithms for NP-Hard Problems: A Survey*, pages 185–207. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

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)