THE UNIVERSITY OF MANCHESTER

FINAL YEAR PROJECT

COMPUTER SCIENCE WITH BUSINESS AND MANAGEMENT

Experiments with the Travelling Salesman Problem (TSP)

Author:
Veselin TODOROV

Supervisor:
Dr. Ian PRATT-HARTMANN

April 26, 2013

Abstract

The Travelling Salesman Problem is one of the best knows problems of combinatorial optimisation. It is believed that no polynomial time algorithm exists for solving it.

This report presents some algorithms for solving the problem. The algorithms vary from brute force approaches and heuristics to genetic algorithms. The report also gives testing data on these algorithms and explains the results obtained.

Acknowledgements

I would like to express my gratitude to my supervisor Dr. Ian Pratt-Hartmann for the unreserved support and guidance given to me during this project. I would also want to thank everyone else who offered support and showed interest in this project.

Contents

Chapter 1

Introduction to The Travelling Salesman problem

Imagine a salesman who needs to visit, a set of cities and return to his starting point. How would he decide which city to visit next in order to minimise the travelling cost for the tour? Would the closest one always be the optimal solution? This problem is know as *The Travelling Salesman Problem (TSP)*. For example, suppose we want to plan a holiday that involves visiting some of Europe's major capitals by plane, starting from London. In order to have a low travel cost and minimise the air miles travelled, it would not make any sense to visit for example London, Berlin and then Paris, as this is almost twice the optimal route for these 3 cities. Figure 1.1 shows a possible route with 11 cities (Copenhagen, Madrid, Berlin, Rome, Budapest, Paris, London, Vienna, Amsterdam, Prague and Warsaw) and minimised travel cost.

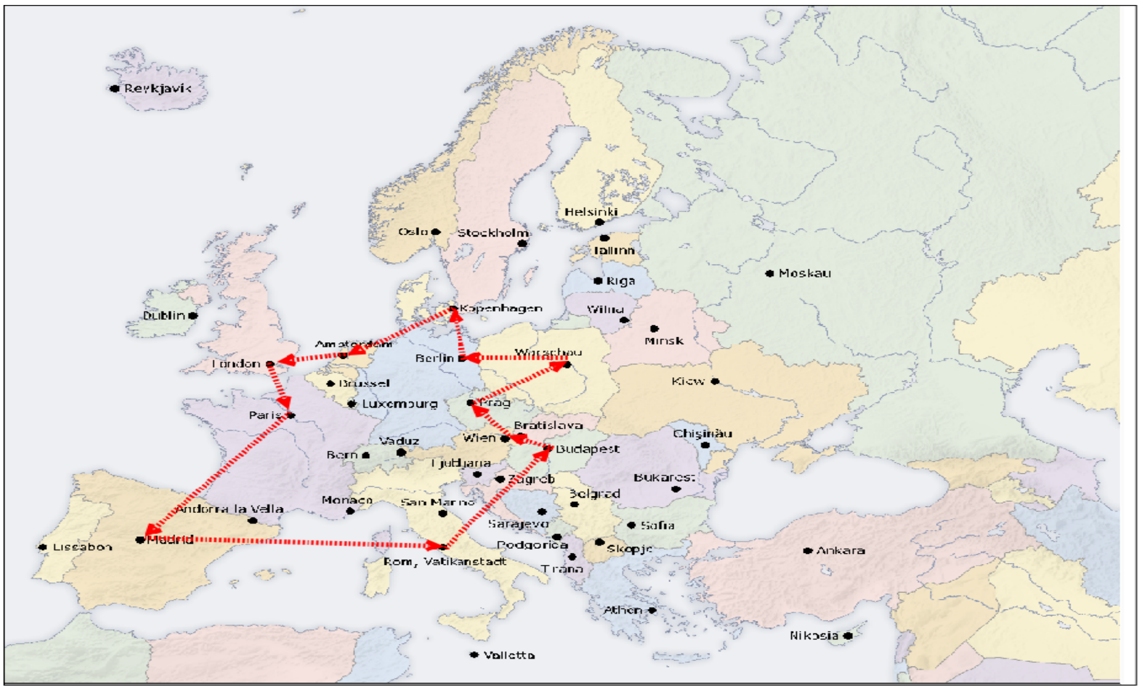


Figure 1.1

1.1 Origins

The Travelling Salesman problem, has a model character in many areas of Computer Science, Mathematics and Operations Research. Heuristics, linear programming, and branch and bound, which are still the main components of today's successful approaches to hard combinatorial optimisation problems, were first formulated for the TSP.

The origins of the problem are unclear. Without any doubt, it is safe to say that people had to deal with this problem long before it was mathematically defined. Small instances of the problem have been and continue to be solved daily. For example when someone has to go shopping and that involves visiting more than one store, the shortest path is always chosen in order to save time and hassle. People do that without even realising it, as the size of the problem is not large at all.

The first formal definition of the problem was done around the 1800s by the British mathematician *Thomas Kirkman* and his Irish colleague *William Rowan Hamilton.* The name 'Travelling Salesman problem' came around the 1930s at *Princeton University* by *Hassler Whitney.* The TSP started to become popular in the late 1950s when it was more deeply researched by members of the *Research and Development (RAND) corporation.* One of the main reasons for the interest raised by this problem, is the fact that people believe a polynomial time algorithm for solving it does not exist.

1.2 Real Life Use

There are various real life applications which take advantage of The Travelling Salesman problem. The following sections give, a brief outline on some problems. The full solutions of these problems can be found in the reference section, as they are not the main subject of this report.

1.2.1 Computer Wiring[1]

This problem arises frequently for example at the Institute for Nuclear Physical Research in Amsterdam during the design of computer interfaces. These interfaces consist of a few modules, which modules have several pins located on them. Module positions on the interface have been determined in advance. The problem here is that, a given subset of pins must be interconnected by wires. In view of possible future changes or corrections and of the small size of the pin, at most two wires are to be attached to any pin. In order to improve ease and neatness of wirability and to avoid signal cross-talk, the total wire length must be minimised.

This type of wiring problem exists in almost every telecommunication company that provides cabling services. As when, a new cable is laid out for a set of points, the cost of doing that must always be minimised in order to decrease expenses. This example emphasises the importance of TSP in computer and telecoms wiring.

1.2.2 Vehicle Routing[1]

Vehicle routing is one of the main real life applications when it comes to The Travelling Salesman problem. The following two problems are just an example of what companies in the logistics sector encounter everyday.

The first problem comes from the national postal service(PTT) in Denmark. In 28 towns in the North-Holland province, the postal service have installed telephone boxes. A technical crew must visit either once or twice a week to empty the coin box. The crew also has to perform minor repairs and replace directories, if necessary. The PTT workers must visit all cities in just one working day, starting and finishing in the same city of Haarlem. The problem here is to minimise the total travelling time and also minimise the number of days in which all telephone boxes can be visited.

A similar problem is observed in the city of Utrecht, where about 200 mail boxes have to emptied daily. The problem here gets more sophisticated as all the boxes must be emptied only in the limits of one hour. The goal is to find the minimum number of trucks that can perform the task. All trucks operate from the city's central railway station.

These problems are just, a overview of the type of problems that every postal service experience. It is not different when it comes to delivering mail. As a whole having an efficient algorithm for solving the TSP is vital for logistics companies and services all over the world.

1.2.3 Machine scheduling[2]

Another example of using in real life the Travelling Salesman problem is scheduling, a machine over a given set of repeated operations. For example, a machine tool may be required for performing, a specific set of operations repetitively. The operator of the machine, would have the task to choose, a sequential order for the operations that minimises the time to complete the set.

1.2.4 X-Ray crystallography[3]

An important use of the TSP occurs in the analysis of the structure of crystals. A X-ray diffractometer is used to gain information about the structure of the material crystalline. The intensity of X-ray reflections are measured by, a detector from various positions. These measurements can be accomplished quite fast, but there is a considerable overhead in positioning time since up to hundreds of thousands positions have to be taken into consideration for some projects. The positioning involves four motors. The time can be computed very accurately that it takes to reposition the heads. The result of this experiment does not depend on the sequence in which measurements are taken. The important thing here is the total time needed, which depends on the sequence. This implies that the problem consists of finding a sequence that minimises the total positioning time, which leads to a travelling salesman problem.

These are just small examples from the endless applications that the Travelling Salesman problem has. Some uses are more complicated and not so clear as others, but the fact remains that it is widely used. Other uses of the TSP are: Overhauling gas turbine engines, handling materials in warehouses and control for robot motions.

1.3 Aim of this report

The aim of this report is to present results from experiments conducted on the Travelling Salesman problem. As there numerous algorithms for solving the problem, this report shows just, a fraction of them. The algorithms for solving it, can be classified into groups.

- One of these groups is the Heuristics' group. Heuristics' refers to experience-based techniques for problem solving, learning and discovery. Heuristic approaches are useful where an exhaustive search is impractical. They produce, a satisfactory result in less time, using shortcuts.

- Genetic algorithms are another class of algorithms for solving the Travelling Salesman Problem. More details on Genetic and Nature inspired algorithms is given in Chapter 6.

The report describes three heuristic approaches for solving the TSP, a relatively simple genetic algorithm and also gives detailed information on exhaustive search. Information is also given about, a greedy approach to the problem.

In order to produce better results, different algorithms have been combined. The most common combination is the greedy algorithm combined, with a heuristic one. More information about these combinations is given in a later chapter.

The report also presents data on how each different algorithm performs on a given set of problems. These results are then compared so that it can be clear which algorithm performs best for the given size of problems.

The algorithms in this report are: Exhaustive Search, Greedy Approach, Heuristics Approach - 2 opt swap[4], Heuristics Approach - 3 opt swap[4], Heuristics Approach - Christofides' Algorithm[5], Genetic Algorithm for solving the TSP without special crossover and mutation . A separate chapter describes in detail every algorithm.

1.4 Preliminaries

Weighted Graph - a representation of a set of object where some object are connected by links. These links have costs in a weighted graph. Figure 1.2 shows an example weighted graph.

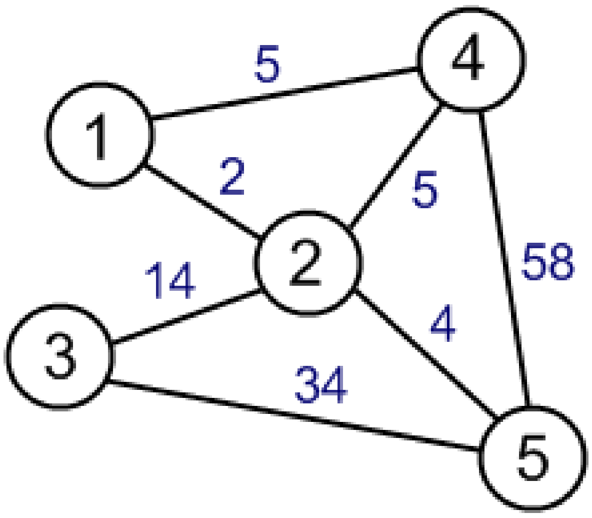


Figure 1.2

Undirected Graph - An undirected graph is one in which edges have no orientation

Tree is an undirected graph in which any two vertices are connected by exactly one simple path.

Tour - A sequence where v_i are vertices, e_i are edges, and for all i the edge e_i connects the vertices v_i - 1 and v_i is called a walk. A walk with no repeated edges is called a tour.

Tour Cost - the cost of all edges in a given tour.

Chapter 2

Exhaustive Search and Greedy Approach

2.1 Exhaustive Search

In order to have, a solid base for comparison between different algorithms, the optimal tour for a given problem must be computed. The only way to obtain such, a result is by performing an exhaustive search. This approach is very simple to implement. The cost on the other hand is time. Because the exhaustive search has to check $n!$ possibilities, this option quickly becomes infeasible to run on, a normal computer. This becomes the case for problems containing around 30 cities. For problems containing 20, or less cities, the time taken to find the optimal solution is considerably more tolerable. Just to give an example this is how 20! looks like - 2432902008176640000 - and this is 30! - 265252859812191058636308480000000. When we consider that the number of computations needed for, a problem of size 30!, is 30! times the number of operations needed to calculate the cost of one tour, the numbers become even bigger.

The exhaustive search algorithm is quite simple. It always starts from one, testing all possible solutions with one and then moves until it reaches n, where n is the size of the problem. An optimisation of the algorithm is achieved by representing all the tours as, a tree. This way large amounts of possible tours can be eliminated from an early stage, if a tour which has not yet visited all cities and it is already exceeding the cost of the currently best one that is complete. The first path computed, becomes the best one so far. Once more paths are computed, when a better one is encountered, it becomes the best.

2.2 Greedy Approach

The Greedy approach is also one of the easiest and straightforward methods for producing, a solution to the Travelling Salesman Problem. The essence of the greedy approach is that it always goes to the nearest node/city, which has not yet been visited, from the current one. This way it tries to minimise the total cost route. This algorithm terminates in just n steps as it always knows which one will be the next node to go to. In this implementation, a small improvement has been made in order to minimise the final cost. For any given problem of size n, the algorithm computes all possible greedy routes. This means that it compares the all the n routes starting each time from different nodes. Due to the randomness when generating the problems, described in the Testing and Evaluation chapter, this approach gives an improvement in the final path cost.

Chapter 3

Heuristics Approach - 2 opt swap[4]

There are two main ways of solving the TSP. The first is using exact methods, such as for example, the Exhaustive Search approach. The problem with these methods is that they are very time-consuming as n grows. The second way is using heuristic approaches. Although they terminate much faster than the exact methods, the cost of the tour produced for a given problem is not optimal.

In this chapter we consider one of the simplest and most naive heuristic approaches for solving the TSP. Given a tour $T1 = (u, ...u_i, u_{i+1}, ...u_j, u_{j+1}, ...u_n)$ the '2-opt' algorithm consists of repeatedly breaking the connection between four vertices and reconnecting them, in a way that decreases the total cost of the tour. In order to optimise $T1$ we need to find two edges and reconnect the vertices, that they connect in such, a way that a vertex from edge one $E1 = (u_i, u_{i+1})$ is now connected to a vertex from edge two $E2 = (u_j, u_{j+1})$. The new edges produced from this are: $E1 = (u_i, u_j)$ and $E2 = (u_{i+1}, u_{j+1})$. This operation is called, a *swap*. After the new edge $E1$ is constructed, all vertices between u_{j+1} and u_{i+1} must be added to the path, before adding $E2$. To be able to perform a swap, the cost of the edges $(u_i, u_{i+1}) + (u_j, u_{j+1})$ must be bigger than $(u_i, u_j) + (u_{i+1}, u_{j+1})$. This produces, a new optimised tour $T2 = (u, ...u_i, u_j, ...u_{i+1}, u_{j+1}, ...u_n)$. The algorithm stops when an optimisation cannot be made to the route any more by this technique. An example of this swap operation is shown below. Figure 2.1A represents an unoptimised trip for, a given problem with $n = 8$. Figure 2.1B shows how the tour is optimised after the first iteration of the algorithm.

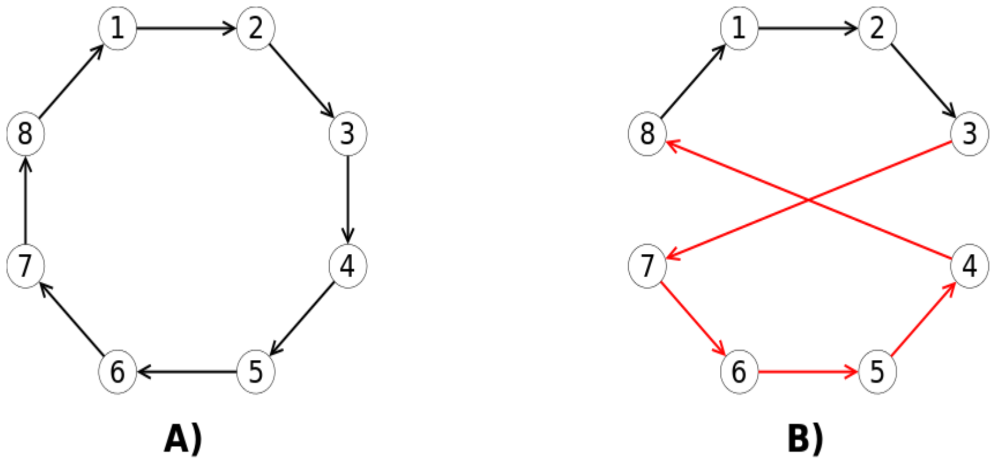


Figure 2.1

Chapter 4

Heuristics Approach - 3 opt swap[4]

The approach taken in the 3-opt heuristic algorithm is almost the same as with the 2-opt, discussed in a Chapter 3. The most significant difference is that in this case we need to find 3 pairs of vertices in order to improve the tour. This is done by breaking the edges between them and connecting the vertices in such a way that the total tour cost is improved. This algorithm possesses the same property concerning the run time and the final result - it terminates significantly faster than exact methods, but at the cost of a somewhat degraded performance produced for, a given problem. For problems where n is small, there is a possibility of an optimal solution to be produced. As n grows, this possibility decreases significantly. Evidence for this claim is produced in Testing and Evaluation chapter.

In this algorithm there are eight possible ways to execute, a *swap* as Figure 3.1 shows. In this project only one is implemented, namely shown in Figure 3.1(h), is explained in more detail. The idea of the other seven is the same, with the difference that the 6 vertices are connected in different ways.

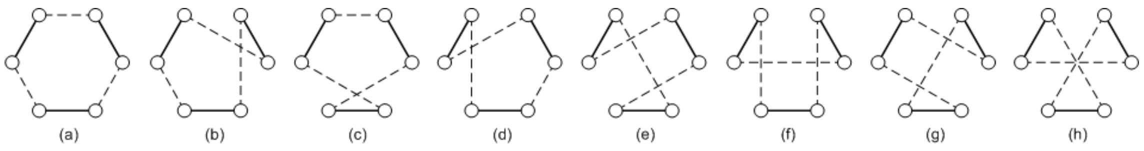


Figure 3.1

We are given a tour $T1 = (u, ...u_i, u_{i+1}, ...u_j, u_{j+1}, ...u_k, u_{k+1}, ...u_n)$ and $E1 = (u_i, u_{i+1})$, $E2 = (u_j, u_{j+1})$ and $E3 = (u_k, u_{k+1})$. In order to construct the optimised tour, the cost of the edges $(u_i, u_{i+1}) + (u_j, u_{j+1}) + (u_k, u_{k+1})$ must be bigger than the cost of $(u_i, u_{j+1}) + (u_k, u_{i+1}) + (u_j, u_{k+1})$. The new produced tour is $T2 = (u, ...u_i, u_{j+1}, ...u_k, u_{i+1}, ...u_j, u_{k+1}, ...u_n)$. This edge swap operation is done until no further improvement can be made.

As there are eight possible ways to swap the edges, a further improvement to the algorithm would be to check all possible combinations of connecting the six vertices in order to maximise the improvement in the final tour. Figure 3.2 shows an example of an unoptimised tour, consisting of 11 nodes, and how the it looks after one iteration of the algorithm is complete.

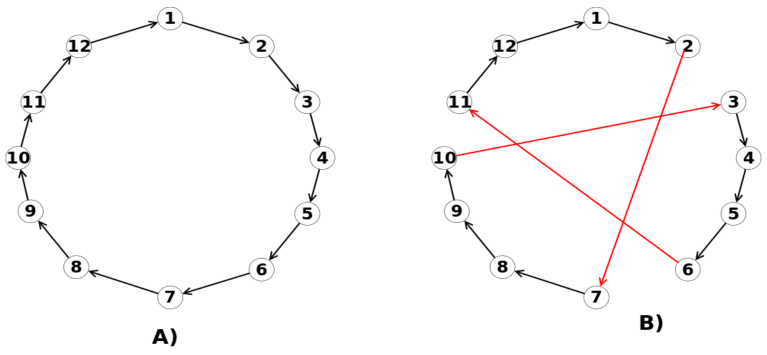


Figure 3.2

Chapter 5

Heuristics Approach - Christofides' Algorithm[5]

5.1 Christofides' Algorithm

This chapter gives details about another heuristic approach. Christofides' algorithm, named after its inventor Nicos Christofides, has the goal of finding a solution to the TSP, given that the resources available are limited in terms of time and space. Christofides' algorithm follows, a few simple steps in order to get, a solution to, a given problem. The following detailed description is an approximation of Christofides' algorithm. More details why this is the case are given in the Testing and Evaluation chapter. Testing results are shown along with the official performance mark for Christofides' algorithm.

The first step in order to find, a solution for the problem, Christofides' algorithm starts by finding, a Minimum spanning tree[7] (MST) S of the given graph. Christofides' algorithm does not specify any particular method how to undertake this. In this implementation of the algorithm, Kruskal's Algorithm[8] is used to obtain, a MST of a graph. Kruskal's Algorithm is explained in detail, in a later section of this chapter. An also efficient alternative to Kruskal's algorithm for finding, a MST of a graph is Prim's Algorithm.

Once the MST of the graph is obtained, the next step of Christofides' Algorithm is to identify the nodes with an odd number of edges. This is done by traversing the already found MST and recording the number of edges for each node.

The next step of Christofides' Algorithm, once the nodes with odd edge degree have been found, is to match these nodes in pairs. As there is only one set of nodes to be matched, this problem is know as 'Algorithm for solving 'The roommates problem'[9]'. Detailed information of a variation of Gale-Shapley algorithm is given in a later section of this chapter for solving the roommates problem. The only restriction for doing this matching is for the total cost of all new edges created to be as low as possible. If an edge already exists between two nodes, this edge is created again as when the final tour is being constructed one of these edges would be deleted. Let N be the set of new edges constructed. Now Let J consist of the minimum spanning tree S and the newly added edges N.

Once the new set of edges - J, consisting of the matchings and the minimum spanning tree, is constructed, the edge degree of each node is calculated again. If all nodes have exactly 2 edges, then the final tour is constructed. A final check of the tour must be done as in some cases the algorithm produces several separate paths. These paths are connected by breaking, a link in each one in a way that minimises the total cost. New links are created to connect all the paths into one. However if this is not the case, a further step must be carried out to delete or add some edges to the set J. In this case we are interested in all nodes with 4 or more edges. Consider for example having an edges uv and vw which are from J. If we delete these edges and add only the edge uw, the graph stays connected and all nodes have an even degree again. This operation is repeated until all nodes in the set J have degree 2. Once this is complete, a check for separate paths must be done.

In order to obtain an optimal result, the final graph containing the minimum spanning three and the additional edges, the triangle inequality must be satisfied along with nonnegative edge costs. The triangle inequality theorem states that any side of a triangle is always shorter than the sum of the other two sides[6]. However due to the random generation of problems, this inequality is not satisfied. This is one of the reasons why this implementation of Christofides' Algorithm underperforms. More detail and evidence of this is shown in Testing and Evaluation.

5.2 Minimum spanning tree[7]

In a given weighted graph, a spanning tree connects all vertices within the graph. A spanning tree cannot contain any loops as it always has n-1 edges, where n is the number of nodes in the graph, and it must satisfy the condition of all nodes to be connected.For the purpose of implementing Christofides algorithm a spanning tree with a minimum cost of the edges must be found, as different spanning trees can be constructed from a given graph with more than 2 nodes.

5.3 Kruskal's Algorithm[8]

Kruskal's algorithm is a greedy algorithm used in graph theory in order to find, a minimum spanning tree of a weighted graph. The first step of this implementation of the algorithm is to create, a sorted array of all the edges in a given graph. The next step is to add the edge with the smallest cost to an array which will hold only the MST. The only condition that must be fulfilled, is to check whether, a loop is created by adding the new edge.

If a loop is created the condition for a spanning tree is violated and therefore this edge cannot be used. This step is repeated until we have $n-1$ edges in the array holding the minimum spanning tree. Kruskal's algorithm is a suitable choice for finding, a MST within Christofides' algorithm as it has a good performance and it only has to be ran once for a given graph. Figure 5.1 shows the steps of Kruskal's algorithm on a small graph.

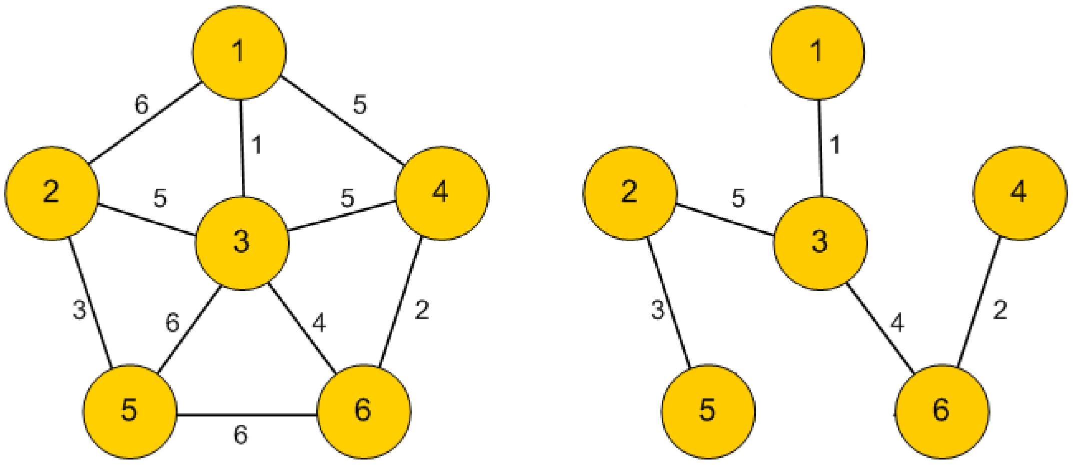


Figure 5.1

The sorted array of edges is:

1. [1,3] = 1	6. [2,3] = 5
2. [4,6] = 2	7. [3,4] = 5
3. [2,5] = 3	8. [1,2] = 6
4. [3,6] = 4	9. [3,5] = 6
5. [1,4] = 5	10.[5,6] = 6

Because we do not have any edges in our spanning tree yet, the first $[1, 3]$ and second $[4, 6]$ edges of the sorted array are always part from the spanning tree. This is because no loops can be created with 2 edges. We then consider the third edge $[2, 5]$ as it is the next smallest. By adding this edge no loops will be made, this means that it will also be a part of the MST. The next edge $[3, 6]$ again does not create any loops, so it must be part of the MST. Edge number $5[1, 4]$ on the other hand, if it is added to the current tree it will form a loop. Thereby this edge cannot be used and we move to the next one. Edge number $6[2, 3]$ does not form a loop so it can be added to the tree. At this point all 6 nodes of the tree are connected without forming any loops. Kruskals algorithm terminates here without checking any more edges as there is no point. The minimum spanning tree has been constructed.

5.4 Algorithm for solving 'The roommates problem'[9]

The stable marriage problem was introduced by Gale and Shapley in the context of matching applicants to colleges taking into consideration their choices. In the marriage problem a set of boys and a set of girls is given. Each boy has a preference list showing the desirable girl he prefers. The goal in this case is to produce a maximum amount of pairs of boys and girls. In order to produce a stable matching, every boy has to be matched to a girl. The second requirement for a stable matching is that there is no boy or girl of the same couple who prefer each other to their actual partner.

In this case, in order to implement Christofides' Algorithm, a version of the above described problem must be solved. 'The roommates problem' is essentially the same, with the difference that there is only one set given. From this set, being always an even number, $n/2$ pairs must be produced.

The algorithm implemented consists of a sequence of proposals made from every person to every other one. In this case 'person' is represented by a node. Once the list of proposals is complete, every person starts to pursue his desirable match. In other words every node wants to be, a match with its first choice, as that will always be the cheapest. For example if x receives, a proposal from y, there are two options for x. The first one is to hold the proposal, if it is the first one offered and keep it until, a better one comes. The second option is to reject the proposal, simply because it is worse(more costly) than an early one. An individual x always proposes in the order of the preference list created by him. Proposals are kept, once they are promised, until the other person breaks it if he receives, a better one. At that point x continues to propose to the next one in the preference list. Proposals from one individual to another can be only one at a time. This naturally means that once the first proposal is made, a chain of proposals and rejections starts from it. Below is a example preference list followed by the proposals.

Example 5.1 : N == 6.

1	4 6 2 5 3
2	6 3 5 1 4
3	4 5 1 6 2
4	2 6 5 1 3
5	4 2 3 6 1
6	5 1 4 2 3

Following the above rules, this would be sequence of proposals produced:

1 proposes to 4;	4 holds 1;	- first proposal to 4
2 proposes to 6;	6 holds 2;	- first proposal to 6
3 proposes to 4;	4 rejects 3;	- 1 is higher in 4s preference list
3 proposes to 5;	5 holds 3;	- 5 is next in preference list
4 proposes to 2;	2 holds 4;	- 4 holds 2 as 6 has not confirmed
5 proposes to 4;	4 holds 5 and rejects 1;	- 5 is higher than 1 in 4s preference list
1 proposes to 6;	6 holds 1 and rejects 2;	- 1 is higher than 2 in 4s preference list
2 proposes to 3;	3 holds 2;	- first proposal to 3
6 proposes to 5;	5 rejects 6;	- proposed to 4 and still not rejected
6 proposes to 1;	1 holds 6;	- 1 already proposed to 6

This part of the algorithm can terminate in two ways. The first one is with a stable matching as the example above shows. The second option occurs only when there are 4 persons/nodes. In this case number 4 can be rejected by all other. This violates the second rule for having a stable matching stated above. For small graphs this is likely to happen, in which case the algorithm has been configured to test all possible combinations and choose the best one.

Chapter 6

Genetic Algorithm for solving the TSP without special crossover and mutation

In the field of computer science, a genetic algorithm is a search heuristic that applies the principles of natural evolution. Genetic algorithms are part of a larger evolutionary algorithms class. One of the purposes of this class algorithms is to produce useful results to optimisation problems, such as is the Travelling Salesman problem and search problems. The techniques that Genetic algorithms use are inspired from nature. These techniques are: inheritance, mutation, selection and crossover.

6.1 Inheritance

Inheritance is widely observed everywhere in living organisms. The most obvious example of inheritance is seen with humans. When a child is born, its DNA is mainly constructed from the DNA of its parents. This most likely means that the child will have visual similarities and other properties such as height, blood type, eye colour etc.

In the case of the TSP, things are almost the same. For example, given we have randomly generated two relatively good solutions to a given problem, these two solutions can be mixed in order to produce a better one. The new route is called also a child. This can be done by simply taking the first half of the solution from the first parent and the second half of the second parent. This must be done carefully as most likely there will be repetitions of nodes in the child.

6.2 Mutation

Mutation is also a technique used by Genetic algorithms. If mutation did not exists, there would not be any evolution. In genetics mutation is a mutation is a change of the nucleotide sequence of the genome of an organism. In the algorithm described below, mutation is not used. In most genetic algorithms this is not the case as mutation is one of the key factors in order to produce as optimal as possible result.

6.3 Selection

During the selection process, individuals may be chosen from a population. The population could be sorted by the appropriate factor, in most cases called fitness. Once the generated population is sorted, it is up to the designer of the algorithm to pick the individuals, from which a new population will be generated.

6.4 Crossover

In genetic operations the Crossover operator determines the amount of chromosomes carried on from one generation to the next. There are different techniques to choose the chromosomes for crossover.Some of them are Roulette wheel selection, Tournament selection and Rank Selection. There are also different crossover techniques. One and two point crossover, Cut and splice and Uniform Crossover and half Uniform crossover. The technique used in the algorithm described in a later section of this chapter is called Partially matched crossover (PMX).

6.5 A simple Genetic Algorithm[10]

This chapter gives detailed description of a simple Genetic algorithm. The algorithm does not uses mutation, or any special crossover techniques. The crossover is done using the PMX method[10]. It will first be explained how the population is generated and after that a visiual example of the crossover. Information also is provided when the algorithm terminates and at what conditions.

In order to start the genetics, an initial population of individuals (routes) must be generated. There are no fixed rules about how to undertake this. In this algorithm the size of the population is always 500 times the size of the problem. This value can be easily increased or decreased. Parents in this case are represented by routes in the format ”5 2 4 6 1 3”. Parents are randomly generated one by one. This is done using the random class in Java to generate, a number between 1 and the size of the problem. When a new number is generated, a check is done if it is already in the route before it is added. This does not give any guarantees that one route can be generated more than one time, but this is highly unlikely even for not so large problems. Once all the population is generated and stored in an array, it is sorted. The sorting is based on fitness and the best individual from the population is always the first. The purpose of the sorting is because it is easier when the algorithm starts to perform the crossovers.

Figure 6.1 gives a visual example of how crossover is done in this implementation.

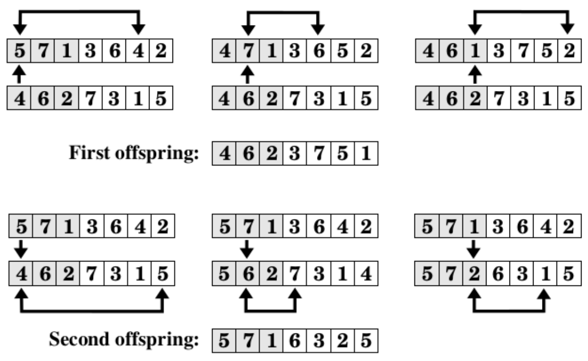


Figure 6.1

Once we have parent A (5 7 1 3 6 4 2) and parent B (4 6 2 7 3 1 5), two new childs can be produced. Two childs are produced as the size of the population will stay the same. This is a choice of preference as we may decide to increase or decrease it. It depends on the size of the problem how many chromosomes (cities or nodes) will be crossed. The algorithm decides that by dividing the size by 3, taking the first part. This number is always rounded down.

A copy of parent A is always taken before the crossover starts as it will become the new offspring. The first step of the crossover is to take the first chromosome of the second parent and replace it at the same position in the copy of the first. The number that was at that position in parent A is swapped in the same parent with the new number that came on its place. This is done so that no route contains one city more than once. As the example shows, we take 4 from parent B and replace it with the 5 in the copy of parent A. The 5 then goes to the position of 4 in that parent. This produces the child C (4 7 1 3 6 5 2). The algorithm then repeats the same step for all the number of chromosomes chosen to be crossed. After finishing a new child is produced. In order to keep the size of the population parent A and parent B now swap their roles so A becomes B and B becomes A. The process is then repeated to produce the second child. The resulting childs now have the same cities in the same positions as parent B between the crossover points. It also has the same cities outside the crossover interval as parent A, where this was not a conflict during the crossover.

Once the new population is generated, the fitness of each new individual is calculated. The array holding the population is now sorted in the same way, the individual with the best fitness is first. The way the algorithm works is that it starts again to produce childs from the new population. At every new population, the best individual is recorded and compared with the previous best. If it is better, it becomes the best solution to the problem found so far. If the algorithm does not find an improvement in as many populations as is the size of the problem, it terminates. If an improvement is made, the algorithm restarts the counter and again tries to do N populations without improvements. Again these boundaries are not strictly set and depend of the type of the problem and the designer. They can easily be altered if more time could be spent in computation.

6.6 Multithreaded implementation

A multithreaded version of this algorithm was implemented. Due to the single additions and comparisons, it has been implemented as a 4 threaded program. The 4 threads do not interfere with variable while they compute their best results. The best result is taken from the 4 best results produced. Again this is something that can be changed depending on the machine that it runs.

Genetic Algorithms do not produce in most cases an optimal result for larger problems. There are more complicated algorithms that use wisely the mutation and selection operators. Due to the random elements in Genetic Algorithms, it is safe to say that they are, a highly educated guess.

Chapter 7

Testing and Evaluation

The aim of this chapter is to present the experiments conducted on The Travelling Salesman Problem. It first explains how the generation of random problems is done. It then moves to the results obtained from the algorithms described in this report. All generated problems have been solved by all algorithms. This is done in order to have, a solid base for comparison. Once the results are explained and proper evaluation has been done, the chapter gives details on the Genetic algorithm. As there is a factor of randomness in the GA, it has been tested separately. The results are still comparable with the other heuristic approaches.

7.1 Generation of Input data

All distances in every set of cities are generated on a random basis. The set is represented as a two dimensional symmetrical array (matrix), which allows instant access for the cost of memory. The distances between cities are generated using Java.random(). As a distance is generated we enter it twice in the matrix in order to make it symmetrical for ease of access. The maximum matrix that can be generated using this method consists of 15174 cities.
In order to test all algorithms on the same matrix, a function for reading a precomputed matrix from a file is used, the file being use as a parameter.

0	**1**	**2**	**3**	**4**	**5**
1	0	3	69	24	45
2	3	0	17	29	68
3	69	17	0	37	82
4	24	29	37	0	8
5	45	68	82	8	0

Example matrix - size 5

7.2 Testing & evaluation with problems of size 10

In this section information on how the tests were conducted is given. A considerable amount of the tests is done for problems of size below 25 nodes. This is because in order to have a solid base for comparison, a large amount of tests must be done. The essence of the problem is that it takes, a considerable amount of time to perform an exhaustive search on a problem with over 25 nodes.
In order to have, a solid base for comparison between different algorithms, the optimal result for each problem has to be computed. This can be done only by performing an exhaustive search

first, on each matrix. The exhaustive search is the only algorithm at the moment that can produce, a 100% accurate results, simply because it is testing all possible route combinations. Once the optimal solution is computed for, a given problem it could be proceeded with the other algorithms subject to this report.

For small problems around 10 nodes, the exhaustive search dominates the preference list when it comes to choosing which one is the best. This is due to the small number of possible routes that it has to compute. The time that takes exhaustive search to compute, a solution for problems with 10 nodes is almost instant. As we can obtain the optimal result at the press of, a button the other algorithms simply cannot compete.

The other algorithms which are tested are the described in earlier chapter, with a few mixtures between them. These mixtures of algorithms are between the greedy approach along with the 2-opt heuristic and the 3-opt heuristic. The mixture is done in the following way. As the 2 & 3 opt algorithm can take any route for, as a starting point and after that optimise it, they are given the result from the greedy approach. These algorithms will be called Greedy2Swap and Greedy3Swap in order not to mix things up. This mixture of algorithms tends to perform better than the tree algorithms performing solo. For example when comparing data on problems of size 10, after performing 1000 tests on different matrices, the Greedy 2 & 3 swaps perform the best after the exhaustive search, out of all. On average these two algorithms manage to produce results, which are 59% for the Greedy2Swap and 60% for the Greedy3Swap above the optimal. When 60% above the optimal is said, this means that if for example the optimal tour cost for a given problem is 100, 60% above would mean cost of 160. The other algorithm's performance is not that good. The only one close to these results is the pure Greedy approach which manages to produce results around 66% above the optimal. The heuristic algorithms perform even worse than this. The 2 opt algorithm produces results around 109% more than the optimal and the 3 opt - 94% above.

Christofides' algorithm has, a slightly different case. As this implementation of Christofides' algorithm is an approximation, due not fulfilling the triangle inequality and not producing optimal matching when it comes to the node matching, the algorithm underperforms. The official benchmark of the proper algorithm is that it always produces, a result less than 2/3 times the optimal[11]. More information on why this is the case is given in the references. In the case with just 10 nodes, Christofides' algorithm manages to produce a result of 80% times the optimal. The standard deviation of the results from each algorithm has also been recorded. With this we can see how consistent is a given algorithm when it comes to performance. The higher the standard deviation, the more inconsistent algorithm when it produces results. For the 10 city problems, the standard deviation for the three best algorithms, Greedy Approach, Greedy2Swap and Greedy3Swap is between 24% and 28%. This means that they produce stable results compared to the normal 2 - 3 swap and Christofides algorithm. Their standard deviation varies between 38% for Christofides, 42% for 3 swap and 50% for the 2 swap.

7.3 Testing & evaluation with problems of size 15 to 22

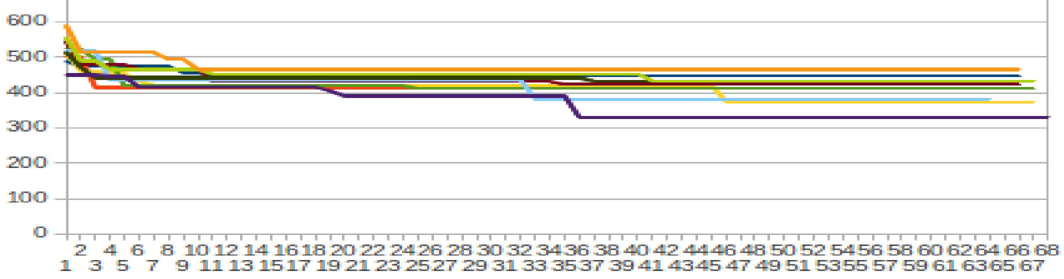
It becomes more interesting to see how the algorithms perform on some problem with size bigger than 10. In this section the results for 8000 problems with size between 15 and 22 are presented. This sections provides valuable information, because as the size of n grows, the time that takes the exhaustive search algorithm to produce the optimal result increases significantly. This means that if, a problem has to be solved in a matter of minutes and a suboptimal solution can satisfy the needs, the algorithm chosen to find it must be as accurate as possible. Because of the methods and techniques that the algorithms use, a suboptimal solution can be obtained relatively fast. For example it takes the exhaustive search several hours to compute the optimal tour cost for a problem of size 20. It takes, on the other hand, a couple of minutes the heuristic ones to produce, a suboptimal.

In this case with size of the problem 15 the best performing two algorithms are the Greedy2Swap and Greedy3Swap. The average offset from the optimal result that they managed to produce is 52%. The standard deviation for this result is 20% for both algorithms. These two algorithms again outperform the rest. The Greedy algorithm produces results with 65% offset, and a deviation of 25% for those results. The heuristics' 2 swap, 3 swap and Christofides' perform the worse. All of them achieve results of above 80% the optimal solution. The deviation with those algorithms is not good also - above 30%. Once the size starts to increase, the algorithms start to behave differently. For example the best algorithm so far, the Greedy3Swap, tends to improve by a small percentage for bigger problems. As it reaches the problems with 20 cities, it gives an offset of 46%, while at the same time the deviation goes down to 15%. The results are the same for the Greedy2Swap. At the point when it reaches problems of size 20, the average offset that it produces goes down from 53% with 15 cities to 49% with 20 cities. The deviation of those results also drops from 20% to 17%. These results show, a trend for these two algorithms improving as the size gets bigger. The greedy approach on the other hand produces consistent results. With 20 nodes it still produces results which are around 65% more than the optimal. The deviation drops down to 17% as the problems get bigger. Again from these results it can be seen that the greedy algorithm produces results which are 65% of the optimal. Considering the fact that it is the simplest and fastest these results can be useful for some problems. The result of the other 3 heuristic approaches is not the same. Christofides' algorithm produces worse results in general. From the 85% offset with 15 nodes, it gradually goes to above 90% when the problem size increases. The standard deviation on the other hand decreases by a few percents down to 35%. The 3 swap algorithm shows an improvement of almost 10% between size 15 and size 20. The deviation for those results also goes does down by 5%. The 2 swap algorithm shows a minor improvement of 3% offset and 5% with the deviation of those results. When it comes to small problems with less than 20 nodes, these results tend in favour of the Greedy3Swap algorithm as the best of the ones tested. This is due to the kickstart optimisation by the greedy algorithm.

7.4 Genetic Algorithm testing

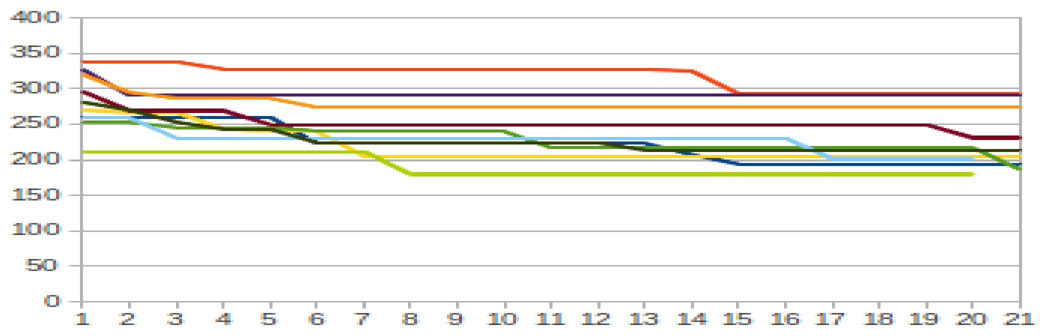
This section gives details about results from the testing of the Genetic algorithm implemented. As the genetic algorithms has, a random factor when it generates solutions, the results are different every time it runs. The test for this algorithm are conducted in a different way of the testing for the other algorithms. This algorithm has been tested with problems starting from size 15 up to size 22. Because of the way the algorithm terminates, the time that each run lasts is different.

When the algorithm computes the costs of every million childs it reaches, the best result is recorded. This way it can be seen how the best result produced progresses as the algorithm runs. Each run represents 4 simultaneous runs due to the multithreaded implementation of the algorithm. For every problem size, 10 runs have been computed. Graph 7.1 gives an example of the runs with size 22. On the left is the cost of the solution and on the bottom is the sample size, or in other words the best solution at each million. Each color represents, a different run of the algorithm of the same size. The optimal solution for this problem is 139.



Graph 7.1

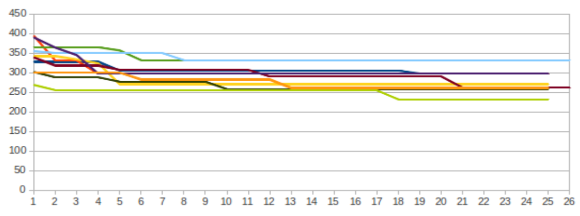
From this graph we can see that all the runs improve significantly in the first couple of millions, up to around 10. After the 10 million mark, the results are still being improved, but with minor values. It also becomes clear from this graph that no two runs are the same. The best solution found by the Genetic algorithm for this problem has a cost of 330, which is a little over 137% more than the optimal. The results for the different problem sizes are not consistent when it comes to the Genetic algorithm. The same number of tests done on a problem of size 15, produced a result which is 71% of the offset. This is how the graph looks like for a smaller problem.



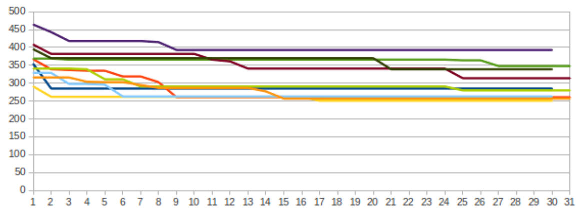
Graph 7.2

It is clear to see here, that the different runs are not so much alike as with 22 cities. The algorithm also terminates earlier at the point of 21 million childs computed. In some of the runs significant improvements are made towards the end. The optimal solution for this problem has a cost of 105, and the best produced from these runs is 180.

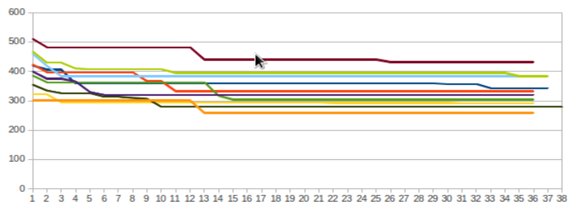
Graphs 7.3 to 7.8 present the results for problems of size 16, 17, 18, 19, 20 and 21.



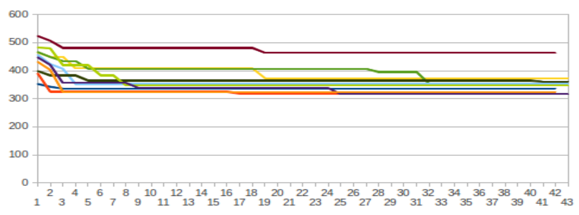
Graph 7.3



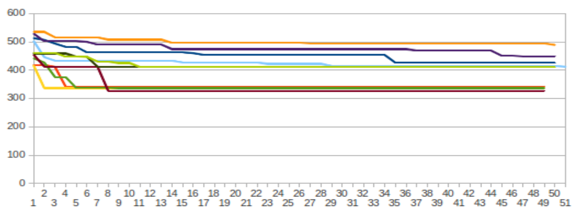
Graph 7.4



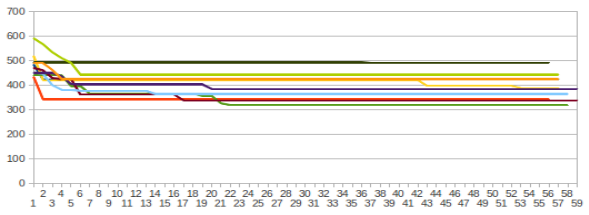
Graph 7.5



Graph 7.6



Graph 7.7



Graph 7.8

Chapter 8

Conclusion

8.1 What did the report show?

The Travelling Salesman Problem is one of the hardest problems in Computer Science. This is one of the reasons why this topic was chosen for the final year project.

In this report six different algorithms have been presented and explained in detail, how to be implemented. Namely these algorithms are: Exhaustive search, Greedy approach, Heuristics' 2 opt swap, Heuristics' 3 opt swap, Christofides' algorithm and a simple Genetic Algorithm. These algorithms are by far not the best when it comes to performance. Most of the algorithms tested perform with more than, a 50% offset of the optimal solution for, a given problem. The aim, however of this report, was not to present a new unseen algorithm for solving the TSP. The aim was to present different types of algorithms and observations on their performance over different problems. For example this way heuristic algorithms can be compared with other heuristic algorithms or brute force approaches. These algorithms have just outlined the basics and laid the foundations for more sophisticated algorithms to be implemented.

8.2 Further work

Due to the fact that no polynomial time algorithm exists, for solving the Travelling Salesman Problem, this remains an open topic. One possibility for further research of the topic, is the mixture of different algorithms. Combining the algorithms may produce that one algorithm which the world is searching for. Mixing algorithms can also lead to new possibilities to explore and new ways of solving the problem. For example, a possible mixture can be done between the heuristics algorithms and the genetic ones. One way of mixing them would be to generate all the initial population of the Genetic Algorithm using the heuristics. This would mean that the genetic algorithm will start with, a very strong population which is not that far away from the optimal result.

Another improvement for the future to be done is the full implementation of Christofides' Algorithm. By full it is considered, that the algorithm for 'the roommates problem' produces stable and optimal solution. This would make the algorithm perform, a lot closer to its' capabilities, which are that it always gives, a solution which is no bigger than 2/3 of the optimal. The ideal way for Christofides' algorithm to work in an optimal way would be to make the initial problem matrix fulfill always the triangular inequality. All of these seem to be minor improvements, but when they are summed up the results would improve significantly. This approximation of Christofides' algorithm was done in a non-optimal way due to the time constraints on the project. This implementation gives, a general overview of the algorithm and its capabilities.

The area of Genetic and Nature Inspired algorithms is also one that can be deeply explored. Complex algorithms can be implemented in order to find, a solution to The Travelling salesman problem. Algorithms involving sophisticated mutation operators and crossovers are just, a small part of what can be done. Another way is also to mix different algorithms with genetic algorithms in order to get the best results. Due to the random factor included in genetic algorithms the possibilities that can be produced from this are endless.

8.3 Personal improvement

This project has taught me many things. The most important of them being that work has to be done by the deadline. The weekly submission done during the course of the project have taught me how to divide the workload and balance many tasks. This project has also boosted my abilities to research, a given topic. Due to the nature of the algorithm I have learned to implement various algorithms in an efficient way. My programming skills were severely boosted, especially in the object orientated programming with Java. This project also has taught me how to write bash scripts in order to make the machine do all the hard work instead of wasting precious time. Performing so many tests was not also an easy task to do as this had to be also balanced. This is due to the fact that the computers used for testing were also workstations for many students.

By doing this project I have achieved the sense of completing a real life task over the course of my final year. The experience gained from this cannot be substituted by anything else.

Chapter 9

References

1. Some Simple Applications of the Travelling Salesman Problem
Author(s): J. K. Lenstra and A. H. G. Rinnooy Kan
Source: Operational Research Quarterly (1970-1977), Vol. 26, No. 4, Part 1 (Nov., 1975), pp.717-733
Published by: Palgrave Macmillan Journals on behalf of the Operational Research Society
Stable URL: http://www.jstor.org/stable/3008306 .
Accessed: 21/04/2013 05:31

2. The Traveling-Salesman Problem
Author(s): Merrill M. Flood
Source: Operations Research, Vol. 4, No. 1 (Feb., 1956), pp. 61-75
Published by: INFORMS
Stable URL: http://www.jstor.org/stable/167517 .
Accessed: 21/04/2013 07:32

3. M.O. Ball et al., Eds., Handbooks in OR & MS, VoL 7
Author(s): Michael Jnger, Gerhard Reinelt, Giovanni RinaMi
Source: Chapter 4, The Travelling Salesman Problem, pp. 231-232
Published by: Elsevier Science B.V.

4. COMBINING 2-OPT, 3-OPT AND 4-OPT WITH K-SWAP-KICK PERTURBATIONS FOR THE TRAVELING SALESMAN PROBLEM
Author(s): Andrius Blazinskas, Alfonsas Misevicius
Source: Kaunas University of Technology, Department of Multimedia Engineering

5. Combinatorial Optimization 1998
Author(s): William J.Cook, William H. Cunningham, William R.Pulleyblank, Alexander Schrijver
Source: Chapter 7 - The Travelling Salesman Problem
Published by: John Wiley & Sons, Inc

6. Math Open Reference[Online]
Stable URL: http://www.mathopenref.com/triangleinequality.html
Accessed: 18/04/2013 15:47

7. Princeton Education[Online]
Stable URL: http://algs4.cs.princeton.edu/43mst/
Accessed: 18/04/2013 16:29

8. The University Of Manchester, School of Computer Science[Online]
Stable URL: http://www.cs.man.ac.uk/ graham/cs2022/greedy/
Accessed: 19/04/2013 09:46

9. An Efficient Algorithm for the 'Stable Roommates' Problem
Author: Robert W. Irving
Source: Journal Of Algorithms 6, pp. 577-595(1985)
Department of Mathematics, University of Salford - May 1 1984

10. Genetic Algorithm Solution of the TSP Avoiding Special Crossover and Mutation
Author: Gokturk Ucoluk
Department of Computer Engineering, Middle East Technical University, 06531 Ankara, Turkey

11. Christofides' Heuristics
Author: Jung-Sheng Lin
February 16, 2005, IEOR 251 Facility Design and Logistics