CS EE World
https://cseeworld.wixsite.com/home
23/34 (B)
May 2022

"e-mail: nadia [dot] hoffmann123 [at] gmail.com
My name is Nadia, I got admitted to several unis, but going to Warsaw University
My supervisor expected my EE to be graded higher, but I'm happy with the result nonetheless, as the entire process of writing the essay was surprisingly quite pleasant. If you have any questions regarding CS EE, CAS or IB in general, I'll be more than happy to help! Good luck with your essay :)"

Investigating algorithms for solving the Traveling Salesman Problem

To what extent Branch and Bound algorithm, Greedy algorithm and the Christofides' algorithm are efficient ways of solving the Travelling Salesman Problem (TSP)?

A Computer Science Extended Essay

4000 words

Table of Contents

1. Introduction.......... **1**

2. Background information.......... **3**

2.1 The Structure of the TSP.......... 3

2.2 The Complexity of the TSP.......... 5

2.3 An Overview of Some Classical Algorithms for Solving the TSP.......... 6

2.3.1 Example of Exact Algorithm: The Branch and Bound Algorithm.......... 7

2.3.2 Example of Approximate Algorithm: The Christofides'Algorithm.......... 9

2.3.3 Example of Heuristic Algorithm: The Greedy Algorithm.......... 10

3. Hypothesis and Experiment Methodology.......... **12**

3.1 Hypothesis and Applied Theory.......... 12

3.2 The Independent Variable.......... 13

3.3 The Dependent Variable.......... 13

3.3.1 Time.......... 13

3.3.2 Accuracy.......... 14

3.4 The Controlled Variables.......... 14

3.5 The Experimental Procedure.......... 15

4. The Experimental Results.......... **17**

4.1 The Tabular Data Representation.......... 17

4.1.1 The Average Execution Time.......... 18

4.1.2 The Accuracy of the Calculated Tour Lengths.......... 20

4.2 The Graphical Data Representation 22
4.2.1 The Average Execution Time 22
4.2.2 The Average Accuracy of the Calculated Tour Lengths 25
4.3 Data Analysis 25
4.4 Possible Improvements and Further Research Opportunities 26
5. Conclusions 28
6. Reference list I
7. Appendices VI
Appendix A: Code of the Algorithms Used VI
Appendix B: The Datasets and Their Representation XV
Appendix C: Raw Data Tables XXXI

1. Introduction

The Traveling Salesman Problem (TSP) is an NP-hard optimization problem. Despite its simplicity at the first glance, it is one of the most known and studied problems in Mathematics and Computer Science. It is dated back to the 18th century, when it was studied by Irish and British mathematicians Sir William Rowan Hamilton and Thomas Penyngton Kirkman. Its idea is: given the cost or travel time between cities in the set find the shortest or cheapest (therefore the most optimal) way through all of them, visiting each of them exactly once and returning to the first one.[1] Additional assumptions are that distances are positive and, since every city is visited only once, there is a finite number of solutions. There are also many variations of the problem: some require that the distance between city A and B is the same as one between B and A (these are called symmetric TSP); in others distance from A to B might be different than from B to A (asymmetric TSP).[2] In this work focus will be placed on symmetric TSP cases.

TSP is a problem which the optimal solutions to are very beneficial for the wide amount of fields. The obvious ones are transport or postal businesses, where goods have to be delivered in the fastest way possible. The optimal route problem also occurs in manufacturing circuit boards or other objects - the order of the holes has to be scheduled so that the machine has to make as little movement as possible to save time and, therefore, money. However, TSP algorithms or their elements are also used in much more scientific and rather surprising areas, such as genome sequencing, planning satellite routes or x-ray crystallography. [3] The variety of

---

[1] Rajesh Matai, Surya Singh, and Murari Lal, "Traveling Salesman Problem: An Overview of Applications, Formulations, and Solution Approaches" in *Traveling Salesman Problem, Theory and Applications* (Rijeka: InTech, 2010), pp. 1 10.5772/12909>.

[2] Chetan Chauhan, Ravindra Gupta, and Kshitij Pathak, "Survey of Methods of Solving TSP along with Its Implementation Using Dynamic Programming Approach", *International Journal of Computer Applications*, 52.4 (2012), 12 <https://research.ijcaonline.org/volume52/number4/pxc3881550.pdf> [accessed 19 February 2021].

[3] Matai, Singh and Lal, "Traveling Salesman Problem: An Overview of Applications, Formulations, and Solution Approaches", pp. 2-4

situations where the algorithms for solving the TSP can be applied shows how broad are applications of them and how important it is to find the most efficient one of them.

To investigate and evaluate the algorithms for solving the TSP, the experiments on their accuracy and execution time were made in order to find the most efficient one, here defined as the one with the highest accuracy and smallest execution time. Different data sets were used and the execution time and accuracy were analyzed. Mathematical and logical explanations for the results obtained were presented and discussed to answer the research question: **to what extent Branch and Bound algorithm, Greedy algorithm and the Christofides' algorithm are efficient ways of solving the Travelling Salesman Problem (TSP)?**

2. Background information

2.1 The Structure of the TSP

TSP can be modelled in several ways, one of which is using the graph theory.

Definition 1: A graph G can be defined as a pair (V, E), where V is a set of vertices, and E is a set of edges between the vertices $E \subseteq \{(u, v) \mid u, v \in V\}$. If the graph is undirected, the adjacency relation defined by the edges is *symmetric*, or $E \subseteq \{\{u, v\} \mid u, v \in V\}$ (sets of vertices rather than ordered pairs).[4]

Several other definitions are needed to model TSP using the graphs. The length or the cost of the routes is the crucial element of the problem. Therefore, the weight function must be applied on every edge.

Definition 2: A weight function or distance function on a graph $G = (V, E)$ is a function $w : E \to \mathbb{R}$ that gives every edge of E a real number.[5]

In order to be able to fully understand the problem, description of some more characteristics of the graph are necessary.

Definition 3: A path is a list of *vertices* of a *graph* where each vertex has an *edge* between it and the next vertex.[6]

Definition 4: Hamiltonian cycle is a *path* through a *graph* that starts and ends at the same *vertex* and includes every other vertex exactly once.[7]

---

[4] Paul E. Black and Paul J. Tanenbaum, "graph", *Dictionary of Algorithms and Data Structures*, 2005 <https://xlinux.nist.gov/dads/HTML/graph.html> [accessed 29 June 2021].
[5] Daniël Vos, "Basic Principles of the Traveling Salesman Problem and Radiation Hybrid Mapping" (unpublished Bachelor Thesis, 2016) <https://repository.tudelft.nl/islandora/object/uuid:0ebb0f3b-e352-4f3c-8465-5c6be2812a90/datastream/OBJ/download> [accessed 19 February 2021].
[6] Paul E. Black, "path", *Dictionary of Algorithms and Data Structures*, 2005 <https://xlinux.nist.gov/dads/HTML/path.html> [accessed 29 June 2021].
[7] Paul E. Black, "Hamiltonian cycle", *Dictionary of Algorithms and Data Structures*, 2005 <https://xlinux.nist.gov/dads/HTML/hamiltonianCycle.html> [accessed 29 June 2021].

Definition 5: A complete graph is an *undirected graph* with an *edge* between every pair of *vertices*.[8]

Therefore, the definition of the TSP might be as follows: "Given an graph *G* and a weight function *w* on *G*, determine the minimum weight Hamiltonian cycle in *G*".[9] Logically, for the graph to have a Hamiltonian cycle, it has to be complete. Therefore, the number of possible tours can be calculated using the formula $\frac{(n-1)!}{2}$, where n is the number of nodes (vertices) of the graph.[10] The more vertices, the more possible tours, and the number of them increases very quickly, which is illustrated in the table 2.1.1 below:

S. No.	Nodes (n)	Edges (Arcs) $\frac{(n-1)n}{2}$	Tours $\frac{(n-1)!}{2}$
1	1	0	0
2	2	1	½
3	3	3	1
4	4	6	3
5	5	10	12
6	6	15	60
7	7	21	360
8	8	28	2520
9	9	36	20160
10	10	45	181440

[8] Paul E. Black, "complete graph", *Dictionary of Algorithms and Data Structures*, 2005 <https://xlinux.nist.gov/dads/HTML/completeGraph.html> [accessed 29 June 2021].
[9] Vos, "Basic Principles of the Traveling Salesman Problem and Radiation Hybrid Mapping"
[10] Antima Sahalot and Sapna Shrimali, "A Comparative Study of Brute Force Method, Nearest Neighbour and Greedy Algorithms to Solve the Travelling Salesman Problem", *IMPACT: International Journal of Research in Engineering & Technology*, 2.6 (2014), 60–61 10.1.1.684.8937>.

Table 2.1.1: Number of edges and tours for complete graph with n nodes. Adapted from Sahalot and Shrimali, "A Comparative Study of Brute Force Method, Nearest Neighbour and Greedy Algorithms to Solve the Travelling Salesman Problem"

2.2 The Complexity of the TSP

To classify the TSP to the computational complexity class, the investigation of the time complexity of the known solving algorithms is needed. In order to do that, the definition of the decision problems is necessary. These are ones the answer for which may be only 'yes' or 'no'.[11] The TSP itself is not a decision problem, but can be easily reformulated in order to become one. The decision version of TSP is: given $b \in R$, does the tour with length less than b exist for a certain TSP?[12] The decision problem belongs to a class P if the polynomial-time algorithm for solving it exists, and to the class NP if the polynomial-time nondeterministic algorithm for solving it exists. Nondeterministic algorithm is one where more than one path of computation is possible.[13] It is proved that "there is a polynomial-time algorithm for the TSP if and only if there is a polynomial-time algorithm for TSP decision".[14] However, Karp in his work proved that decision problem of TSP is NP-complete. When a problem belongs to NP-complete class, it means it is "a decision problem such that if it can be solved in polynomial time then every problem in NP can be solved in polynomial time".[15] A problem such that its decision problem is in NP-complete class is called NP-hard, which is the case of the Traveling Salesman Problem.[16]

[11] Tiago Salvador, "The Traveling Salesman Problem: A Statistical Approach", (2010), pp. 8–12 <http://www.math.lsa.umich.edu/~saldanha/Files/Report%20TSP.pdf> [accessed 29 June 2021].
[12] Vos, "Basic Principles of the Traveling Salesman Problem and Radiation Hybrid Mapping"
[13] Salvador, "The Traveling Salesman Problem: A Statistical Approach"
[14] Salvador, "The Traveling Salesman Problem: A Statistical Approach"
[15] Vos, "Basic Principles of the Traveling Salesman Problem and Radiation Hybrid Mapping"
[16] Vos, "Basic Principles of the Traveling Salesman Problem and Radiation Hybrid Mapping"

2.3 An Overview of Some Classical Algorithms for Solving the TSP

There are many classical algorithms developed for solving the TSP. It is important to distinguish between main types of the solutions: the exact and non-exact ones, which can be divided into the approximate and the heuristic ones.[17] Each of them have their advantages and disadvantages, which are described in the table 2.3.1 below.

Exact solvers	Approximate solvers	Heuristic solvers
Guarantee of finding the optimal solution	In worst case approximation for the found solution within a known bound	Find feasible solution
Longer execution time	Usually faster than exact algorithms	Usually faster than exact algorithms
More space used	Usually less space used than exact algorithms	Usually less space used than exact algorithms

Table 2.3.1: Comparison of types of algorithms. Adapted from Chauhan, Ravindra, and Kshitij, "Survey of Methods of Solving TSP along with Its Implementation Using Dynamic Programming Approach"

It is easy to confuse approximate and heuristic algorithms, as those names are sometimes used interchangeably. For example, the Christofides' algorithm is sometimes described as a heuristic algorithm, whereas in other papers it is an approximate one. However, there is an important distinction between those two types of solutions. Heuristics is following certain steps in order to solve a very time-consuming problem quicker or find an approximate solution when exact one is impossible to find using classic methods. It trades accuracy and/or optimality for speed.[18]

---

[17] Chauhan, Ravindra, and Kshitij, "Survey of Methods of Solving TSP along with Its Implementation Using Dynamic Programming Approach"
[18] Judea Pearl, *Heuristics : Intelligent Search Strategies for Computer Problem Solving* (Reading, Mass.: Addison-Wesley Pub. Co, 1984).

However, there is no proof on the quality of the given result. An algorithm is called an approximate one when the quality of the solution in the worst-case scenario is known and mathematically proved.[19] Therefore, as the Christofides' algorithm has a mathematical proof of the worst-case results bound, and Greedy does not have one, Christofides' method is considered as an approximation solution and Greedy algorithm is heuristic in this paper.

2.3.1 Example of Exact Algorithm: The Branch and Bound Algorithm

The Branch and Bound algorithm is an example of an exact solution for TSP. Its main idea is to break the problem into several sub-problems, calculate lower bounds for them and therefore find routes whose distances are less than bound.[20] The mathematical approach to this principle as stated by Y. Narahari is as follows:

"Let S be a subset of solutions, $L(S)$ – a lower bound on the cost of any solution belonging to S and C – a cost of the best solution so far. If $C \leq L(S)$, there is no need to explore S because it does not contain better solution. If $C > L(S)$, then we need to explore S because it may contain a better solution."[21]

There might be different bounding functions used, therefore the exact time complexity depends on the function chosen.[22] However, in the worst case of never being able to prune a node the

[19] David P Williamson and David Bernard Shmoys, *The Design of Approximation Algorithms* (New York: Cambridge University Press, 2011).

[20] Mirta Mataija, Mirjana Rakamarić Šegić, and Franciska Jozić, "Solving the Travelling Salesman Problem Using the Branch And Bound Method," *Zbornik Veleučilišta U Rijeci*, 4.1 (2016), 259–70 <https://hrcak.srce.hr/file/236378> [accessed 16 January 2022].

[21] Y. Narahari, "8.4.2 Optimal Solution for TSP Using Branch and Bound," *Gtl.csa.iisc.ac.in*, 2001 <https://gtl.csa.iisc.ac.in/dsa/node187.html> [accessed 15 January 2022].

[22] Weixiong Zhang, *Branch-And-Bound Search Algorithms and Their Computational Complexity*, 1996 <https://apps.dtic.mil/sti/pdfs/ADA314598.pdf> [accessed 16 January 2022].

complexity of this algorithm is the same as the brute force algorithm's, so $O(n!)$.[23] The steps for the algorithm used in this investigation are as follows, according to Y. Narahari[24]:

1. Calculate the lower bound on the cost of any tour with the formula:

$$\frac{1}{2}\sum_{v\in V}(Sum\ of\ the\ costs\ of\ the\ two\ least\ cost\ edges\ adjacent\ to\ v)$$

where V is the set of all vertices.

2. Prepare a solution tree. An example solution tree for 5 nodes is:

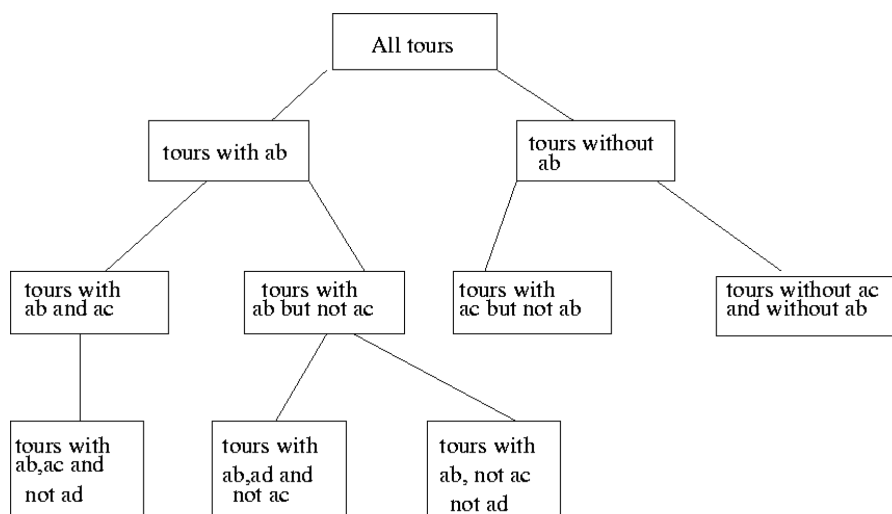


Figure 2.3.1.1: A solution tree example for 5 nodes. Adapted from Y. Narahari, "8.4.2 Optimal Solution for TSP Using Branch and Bound"

The nodes are in the lexicographic order.

3. In the step of branching, while considering two children of the node, decide which edges should be excluded or included from the route, using the following rules:

---

[23] Mataija, Rakamarić Šegić, and Jozić, "Solving the Travelling Salesman Problem Using the Branch And Bound Method"
[24] Y. Narahari, "8.4.2 Optimal Solution for TSP Using Branch and Bound,"

a. “If excluding the edge *(x, y)* would make it impossible for nodes *x* or *y* to have as many as two adjacent edges in the tour, then *(x, y)* must be included.

b. If including (x, y) would cause *x* or *y* to have more than two edges adjacent in the tour, or would complete a non-tour cycle with edges already included, then *(x, y)* must be excluded.”[25]

4. Calculate lower bounds for both children nodes, excluding the nodes that need to be omitted according to the solution tree. If the lower bound of the child is greater or equal to lowest cost of the tour calculated so far, this child node may be ignored and so its descendants.
5. If no child node can be ignored, the child with smaller lower bound is considered first. After examining this node, the sibling node has to be investigated again, as there might be new best solution found.[26]

2.3.2 Example of Approximate Algorithm: The Christofides'Algorithm

An example of an approximate algorithm for TSP may be the Christofides’ algorithm. It was developed by Nicos Christofides as an extension of one of the earlier-developed algorithms with ratio of 2 in worst case scenario (which means the tour found could be utmost twice as long as the optimal solution).[27] The Christofides’ method has ratio of 3/2 to the shortest possible route in the worst case, which is an improvement to previous algorithms. It performs worse in terms of time complexity, though, as it is equal to $O(n^3)$ comparing to earlier $O(n^2 log_2(n))$.[28] The algorithm’s procedure is:

---

[25] Y. Narahari, “8.4.2 Optimal Solution for TSP Using Branch and Bound,”
[26] Y. Narahari, “8.4.2 Optimal Solution for TSP Using Branch and Bound,”
[27] Nicos Christofides, *Worst-Case Analysis of a New Heuristic for the Travelling Salesman Problem*, 1976 <https://apps.dtic.mil/sti/pdfs/ADA025602.pdf> [accessed 15 January 2022].

[28] Christian Nilsson, *Heuristics for the Traveling Salesman Problem*, 2003 <http://160592857366.free.fr/joe/ebooks/ShareData/Heuristics%20for%20the%20Traveling%20Salesman%20Problem%20By%20Christian%20Nillson.pdf> [accessed 15 January 2022].

1. Build a Minimum Spanning Tree from the inputted graph. Minimum Spanning Tree (MST) is the tree connecting all vertices of the undirected, edge-weighted graph with the lowest possible total weight and without any cycles.[29]
2. Create a Minimum Weight Matching (MWM) in the subgraph of nodes with an odd degree in the MST. MWM is finding a set of edges without common vertices with the minimized sum of weights.[30]
3. Add edges from MWM to the MST.
4. Create an Euler cycle (so the cycle that includes every edge[31]) from the pseudograph generated in the previous steps. Take shortcuts to avoid visiting a node twice.[32]

2.3.3 Example of Heuristic Algorithm: The Greedy Algorithm

The Greedy algorithm is a simple example of a heuristic approach for solving the TSP.[33] In order to find the optimal tour, it searches for the shortest edge from the current node and adds it to the path unless it would create a cycle which has less edges than the number of nodes or it "increases the degree of any node to more than 2".[34] What is more, one edge cannot be added twice. The algorithm's steps are as follows:

1. Sort all edges.
2. Choose the shortest edge that follows the rules above.

---

[29] Jaroslav Nešetřil, Eva Milková, and Helena Nešetřilová, "Otakar Borůvka on Minimum Spanning Tree Problem Translation of Both the 1926 Papers, Comments, History," *Discrete Mathematics*, 233.1-3 (2001), 3–36 <https://www.sciencedirect.com/science/article/pii/S0012365X00002247> [accessed 16 January 2022].

[30] William Cook and André Rohe, "Computing Minimum-Weight Perfect Matchings," *INFORMS Journal on Computing*, 11.2 (1999), 138–48 <https://www.math.uwaterloo.ca/~bico/papers/match_ijoc.pdf> [accessed 16 January 2022].

[31] Emanuel Lazar, *5.6 Euler Paths and Cycles*, 2016 <https://www2.math.upenn.edu/~mlazar/math170/notes05-6.pdf> [accessed 16 January 2022].

[32] Christian Nilsson, *Heuristics for the Traveling Salesman Problem*
[33] Christian Nilsson, *Heuristics for the Traveling Salesman Problem*
[34] Christian Nilsson, *Heuristics for the Traveling Salesman Problem*

3. Repeat step 2 as long as tour consists of less than N edges, N being a number of cities (nodes).

The time complexity of the Greedy algorithm is $O(n^2 log_2(n))$.[35]

[35] Christian Nilsson, *Heuristics for the Traveling Salesman Problem*

3. Hypothesis and Experiment Methodology

3.1 Hypothesis and Applied Theory

The algorithms were described and explained carefully, including their time efficiency. The comparison of the complexities in the big O notation is:

$$O(n^3) > O(n^2 \log_2 n) > O(n!) \text{ for n} < 5,$$

$$O(n^3) > O(n!) > O(n^2 \log_2 n) \text{ for n} = 5$$

and

$$O(n!) > O(n^3) > O(n^2 \log_2 n) \text{ for n} > 5.$$

Therefore it can be concluded that the comparative time efficiency of the algorithms depends on the datasets sizes. In case of less than 5 cities, the most time-efficient algorithm should be Branch and Bound, then Greedy and the last one should be Christofides'. For the problems with 5 cities, the best method should become the Greedy, second should be the Branch and Bound, and the Christofides' algorithm should perform the worst. However, when the number of cities will become greater than 5, the time efficiency of the Christofides' method should get better than the Branch and Bound one.

The most accurate algorithm will be the Branch and Bound one, as it gives the exact solutions. It is proved that the results given by the Christofides' method will be within the bound of 3/2 of the optimal tour. However, it is hard to predict the accuracy of the Greedy algorithm, as there is no proof to its correctness.

In conclusion, Branch and Bound should be the fastest and most accurate, so the most efficient one for small datasets. For bigger datasets, Greedy algorithm should have shorter execution time than the Christofides', however, its accuracy might be worse.

3.2 The Independent Variable

The independent variable that will be changed during the experiment is the **dataset size**. As the number of all possible tours grows very quickly, the number of the cities in consecutive datasets are increased by 5, starting from 5 and ending at 25 included. This way a sufficient range of results will be obtained, resulting in not too many points on the graph, but enough to analyze the algorithms. What is more, some algorithms may perform good on particular datasets, while in reality their complexity is much bigger than what is represented by the time took to give the solution. Therefore 3 different data matrices will be tested within the dataset size in order to obtain more reliable results.

3.3 The Dependent Variable

In order to answer the research question and find the **efficiency** of the algorithms investigated, the dependent variable in this experiment consists of two elements – **time** and **accuracy** of the result obtained. Both of them are crucial, as an algorithm that gives the shortest tour length, but takes very long time to find it, is less efficient than one that gives tour length very close to the shortest one, but in much less time. On the other hand, algorithm giving quick, but very inaccurate results is not very efficient as well.

3.3.1 Time

Execution time of the algorithms is going to be measured in nanoseconds using System.nanoTime() method. The time elapsed from some arbitrary but fixed origin time[36] will be measured before and after calling the function starting the algorithms to avoid influence on the results by additional calculations (for example one connected to creation of graph or reading the data matrix). Then results will be subtracted from each other to get the completion time of

---

[36] Oracle, "System (Java Platform SE 8)," *Docs.oracle.com* <https://docs.oracle.com/javase/8/docs/api/java/lang/System.html#nanoTime--> [accessed 13 January 2022].

the algorithm. The average execution time in a dataset size is going to be taken into account, as algorithms' calculation time might be different for different data. Some may be the worst-case scenario for the algorithms, especially the exact one, therefore the results will be calculated much longer than with other data.

3.3.2 Accuracy

In order to calculate the accuracy, the shortest possible tour is necessary to be known. This will be obtained through execution of the branch and bound algorithm, which is an exact algorithm and will give accurate shortest path. Then, a formula will be applied:

$$\frac{shortest\ path}{path\ calculated} = accuracy$$

in order to calculate the ratio of the paths and the calculated path accuracy.

3.4 The Controlled Variables

Several variables may have an influence on the efficiency of the algorithms, especially time, therefore they need to be controlled. They are shown in the table 3.4.1 below.

Variable	**Description**
The same code used for each algorithm	The algorithms from *Appendix A* will be used
The same data type used	In all the sets tested the data type used will be int (32-bit integer)
Computer and operating system used	The experiment will be conducted on one computer, an Acer Aspire A515-52G personal laptop. Specifications: **Operational System**: Windows 10 Home, version 21H1

	Processor: IntelI Core i5-8265U CPU @ 1.60GHz **Memory**: 8GB 2400 MHz DDR3
The same number of programs opened on the computer during tests	The only program opened by the user (apart from the background tasks executed independently by the operating system) is going to be IntelliJ IDEA executing the code
Integrated Development Environment (IDE) used	**IDE**: IntelliJ IDEA Educational 2021.2.3 **Build**: #IE-212.5457.63 **Java Runtime Environment**: 1.8.0_271-b09 x86_64 **Java Virtual Machine**: OpenJDK 64-Bit Server VM

Table 3.4.1: Controlled variables and their descriptions

3.5 The Experimental Procedure

The experimental procedure is as follows: first, the 1st dataset matrix for 5 cities will be inserted into the code of Branch and Bound algorithm. It will be run 5 times, recording the execution time and tour length calculated each time. Next, the same will be done with datasets 2 and 3 for 5 cities, and all the rest of datasets (datasets 1, 2 and 3 for 10, 15, 20 and 25 cities). The Greedy and Christofides' algorithms will be tested analogously. The code of the algorithms was taken from the sources represented in table 3.5.1.

Algorithm	**Source**
The Branch and Bound algorithm	Rai, Anurag, "Traveling Salesman Problem Using Branch and Bound," *GeeksforGeeks*, 2016

	<https://www.geeksforgeeks.org/traveling-salesman-problem-using-branch-and-bound-2/> [accessed 8 January 2022]
The Greedy algorithm	JGraphT, “jgrapht/jgrapht-core/src/main/java/org/jgrapht/alg/tour/GreedyHeuristicTSP.java,” *GitHub*, 2022 <https://github.com/jgrapht/jgrapht/blob/master/jgrapht-core/src/main/java/org/jgrapht/alg/tour/GreedyHeuristicTSP.java> [accessed 27 January 2022]
The Christofides’ algorithm	JGraphT, “jgrapht/jgrapht-core/src/main/java/org/jgrapht/alg/tour/ ChristofidesThreeHalvesApproxMetric TSP.java,” *GitHub*, 2022 <https://github.com/jgrapht/jgrapht/blob/master/jgrapht-core/src/main/java/org/jgrapht/alg/tour/ ChristofidesThreeHalvesApproxMetric TSP.java> [accessed 27 January 2022]

Table 3.5.1: The sources of the code of the algorithms.

Please refer to the *Appendix A* for the code used and to *Appendix B* for the datasets tested and their representation.

4. The Experimental Results

4.1 The Tabular Data Representation

Examples of the raw data can be seen in the table 4.1.1 and 4.1.2 below.

For 5 cities	Dataset 1 [ns]	Dataset 2 [ns]	Dataset 3 [ns]
Branch and Bound	194.400	183.400	59.500
	223.000	171.500	132.500
	134.000	158.900	74.300
	130.300	166.300	68.100
	103.700	156.000	91.900
Greedy	26673.400	31479.400	22900.800
	28784.600	27000.100	27029.200
	24204.100	30058.700	27804.000
	25795.200	24730.900	27396.300
	31906.500	27571.800	31047.700
Christofides'	86633.700	53487.800	57711.300
	88809.100	51471.500	61522.000
	67883.700	58842.500	44465.700
	75021.400	52108.600	78825.400
	79499.600	59324.400	51810.400

Table 4.1.1: The raw data table of execution time of the algorithms for 5 cities.

For 5 cities	Dataset 1	Dataset 2	Dataset 3
Branch and Bound	19	67	1209
Greedy	21	69	1209
Christofides'	21	69	1209

Table 4.1.2: The raw data table of the tours' lengths of the algorithms for 5 cities.

Such tables were prepared for each of the datasets' sizes, giving total of 225 results for time and 45 for tour lengths. For the rest of the raw data please refer to *Appendix C*. In the tables below, the processed data will be presented.

Data was processed using Microsoft Excel, therefore precise values were used in calculations. In the tables below, numbers were rounded to 3 decimal digits for clear representation.

4.1.1 The Average Execution Time

For 5 cities				
	Dataset 1 [ns]	Dataset 2 [ns]	Dataset 3 [ns]	Average [ns]
Branch and Bound algorithm	157.080	167.220	85.260	136.520
Greedy algorithm	27472.760	28168.180	27235.600	27625.513
Christofides' algorithm	79569.500	55046.960	58866.960	64494.473
For 10 cities				
	Dataset 1 [ns]	Dataset 2 [ns]	Dataset 3 [ns]	Average [ns]
Branch and Bound algorithm	8475.960	2509.160	6525.200	5836.773
Greedy algorithm	32070.120	27393.640	28904.280	29456.013
Christofides' algorithm	75827.920	82613.760	66070.360	74837.347
For 15 cities				
	Dataset 1 [ns]	Dataset 2 [ns]	Dataset 3 [ns]	Average [ns]

Branch and Bound algorithm	26930.660	140106.840	1095020.020	420685.840
Greedy algorithm	33352.820	30587.940	31114.000	31684.920
Christofides' algorithm	71988.960	71816.740	84018.380	75941.360
For 20 cities				
	Dataset 1 [ns]	Dataset 2 [ns]	Dataset 3 [ns]	Average [ns]
Branch and Bound algorithm	10030093.880	1271482.000	3448042.480	4916539.453
Greedy algorithm	34619.840	32828.660	35834.960	34427.820
Christofides' algorithm	74797.660	79019.260	79421.360	77746.093
For 25 cities				
	Dataset 1 [ns]	Dataset 2 [ns]	Dataset 3 [ns]	Average [ns]
Branch and Bound algorithm	34076004.580	No data	No data	No data
Greedy algorithm	44401.460	38624.000	34615.240	39213.567
Christofides' algorithm	77445.980	71615.460	86470.760	78510.733

Table 4.1.1.1: Average execution time in the datasets and the average for the algorithms.

For 5 and 10 cities there are big differences between average execution times of the exact solution and other algorithms. However, for 15 cities the Branch and Bound algorithm's execution time significantly exceeds two other ones. The differences between times of Greedy and Christofides' methods are pretty consistent throughout all trials. The Branch and Bound

algorithm for datasets 2 and 3 with 25 cities did not give any results after 5 hours of calculations, therefore the program was terminated and no execution time was obtained.

4.1.2 The Accuracy of the Calculated Tour Lengths

The Branch and Bound algorithm is the exact algorithm, therefore the tour calculated by it is the shortest possible one. Therefore the accuracy ratio for this algorithm is always going to be equal to 1.

For 5 cities				
	Accuracy ratio for dataset 1	Accuracy ratio for dataset 2	Accuracy ratio for dataset 3	Average accuracy ratio
Branch and Bound algorithm	1.000	1.000	1.000	1.000
Greedy algorithm	0.905	0.971	1.000	0.959
Christofides' algorithm	0.905	0.971	1.000	0.959
For 10 cities				
	Accuracy ratio for dataset 1	Accuracy ratio for dataset 2	Accuracy ratio for dataset 3	Average accuracy ratio
Branch and Bound algorithm	1.000	1.000	1.000	1.000
Greedy algorithm	0.882	0.823	0.913	0.873
Christofides' algorithm	0.938	0.948	1.000	0.962
For 15 cities				

	Accuracy ratio for dataset 1	Accuracy ratio for dataset 2	Accuracy ratio for dataset 3	Average accuracy ratio
Branch and Bound algorithm	1.000	1.000	1.000	1.000
Greedy algorithm	1.000	0.948	0.912	0.953
Christofides' algorithm	0.895	0.846	0.940	0.894
For 20 cities				
	Accuracy ratio for dataset 1	Accuracy ratio for dataset 2	Accuracy ratio for dataset 3	Average accuracy ratio
Branch and Bound algorithm	1.000	1.000	1.000	1.000
Greedy algorithm	0.790	0.879	0.974	0.881
Christofides' algorithm	0.887	0.969	0.881	0.912
For 25 cities				
	Accuracy ratio for dataset 1	Accuracy ratio for dataset 2	Accuracy ratio for dataset 3	Average accuracy ratio
Branch and Bound algorithm	1.000	No data	No data	1.000
Greedy algorithm	0.946	No data	No data	0.946
Christofides' algorithm	0.919	No data	No data	0.919

Table 4.1.2.1: Accuracy ratio of the results and the average accuracy ratio for the algorithms.

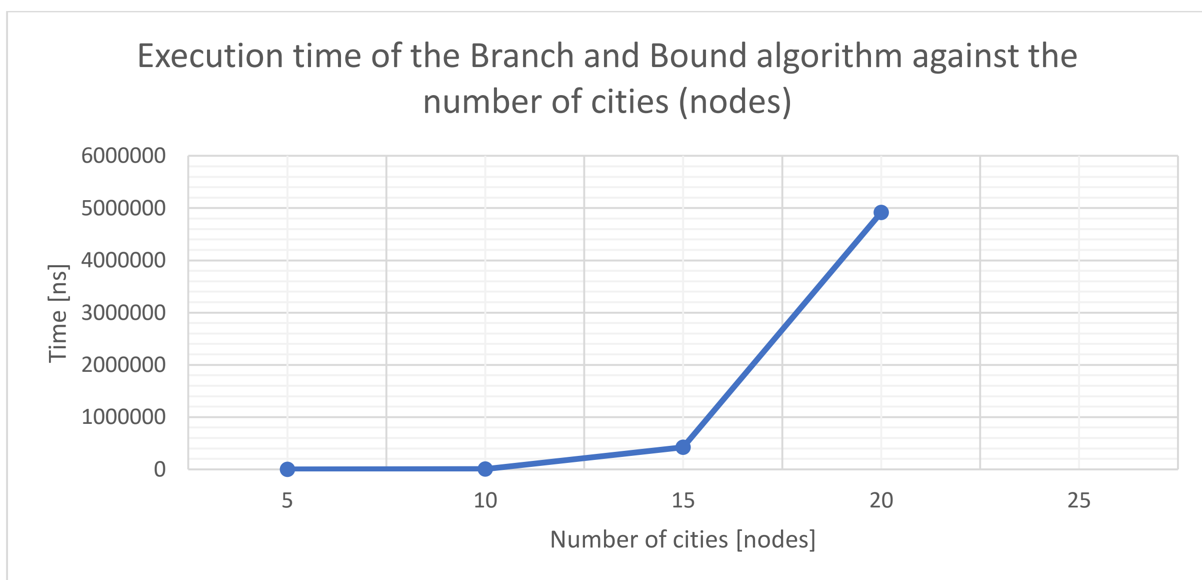
The accuracy ratios of the Greedy and Christofides' algorithms do not follow a consistent pattern. The Branch and Bound algorithm for datasets 2 and 3 with 25 cities did not give any results after 5 hours of calculations, therefore the program was terminated. Since no exact shortest tour length was obtained for both of the datasets, it was impossible to calculate the accuracy ratio for other algorithms in datasets 2 and 3. Thus the accuracy ratio for the dataset 1 was used as an average for the case of 25 cities.

Algorithm	Average accuracy ratio
Branch and Bound algorithm	1.000
Greedy algorithm	0.922
Christofides' algorithm	0.929

Table 4.1.2.2: Overall average accuracy ratio of the algorithms.

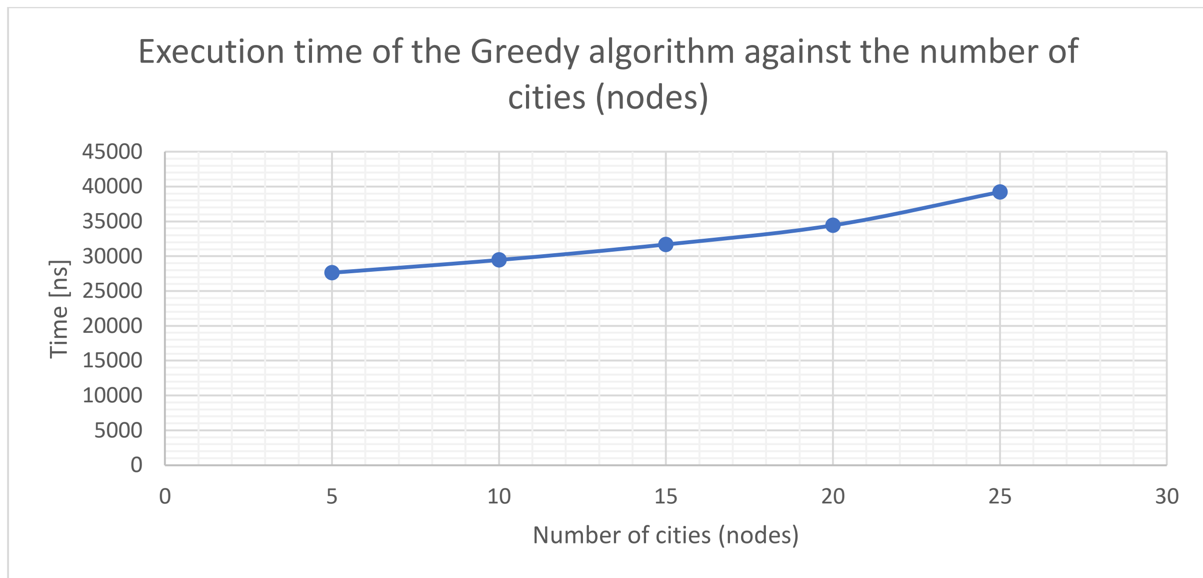
4.2 The Graphical Data Representation

4.2.1 The Average Execution Time



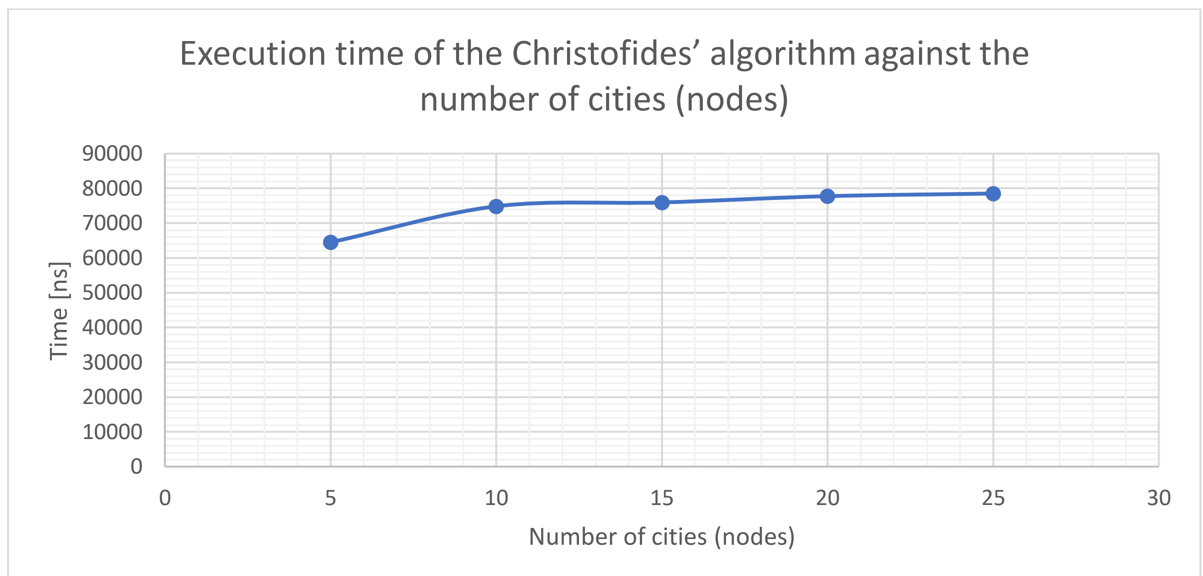
Graph 4.2.1: Execution time of the Branch and Bound algorithm against the number of cities (nodes)

The graph 4.2.1 shows the fast significant increase in the execution time of the Branch and Bound algorithm when the datasets get bigger.



Graph 4.2.2: Execution time of the Greedy algorithm against the number of cities (nodes)

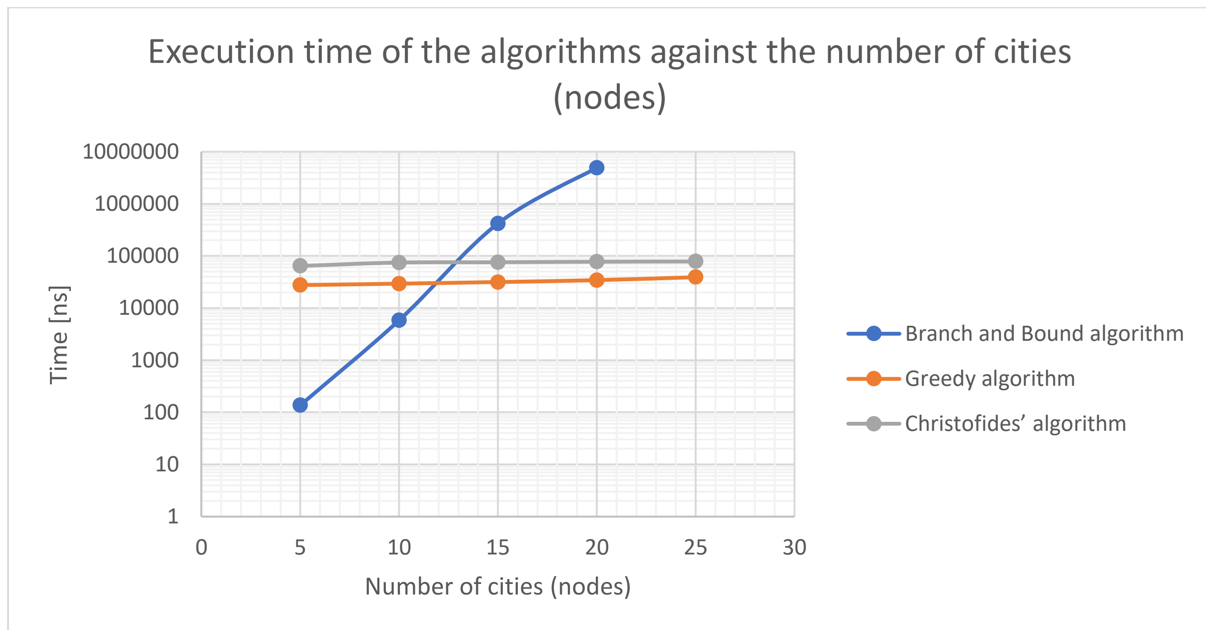
The Greedy algorithm's execution time presented on the graph 4.2.2 increases slowly, but steadily.



Graph 4.2.3: Execution time of the Christofides' algorithm against the number of cities (nodes)

The Christofides' algorithm, as shown in the graph above, slows down more significantly when number of cities is increased from 5 to 10; later its execution time grows steadily.

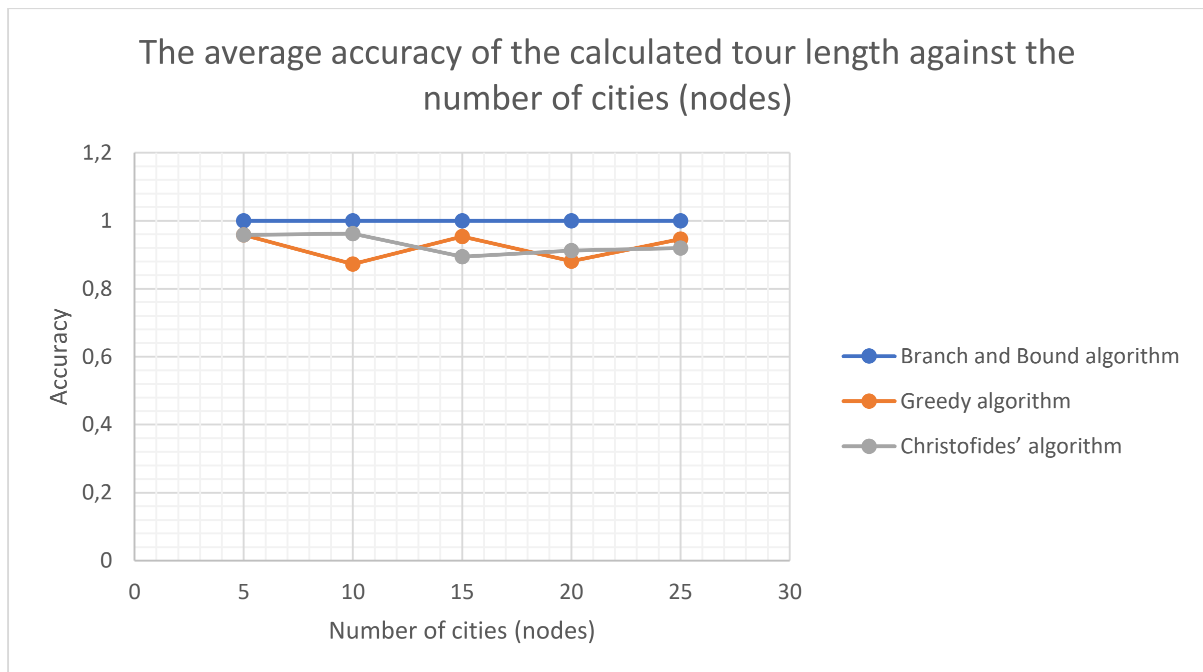
A graph below has a logarithmic scale for clearer data representation.



Graph 4.2.4: Execution time of all algorithms against the number of cities (nodes)

The points of intersection of the Branch and Bound algorithm's graph and other ones are for 12 cities in case of the Greedy algorithm and 13 for Christofides' algorithm.

4.2.2 The Average Accuracy of the Calculated Tour Lengths



Graph 4.2.1: The average accuracy of the calculated tour path by all algorithms against the number of cities (nodes)

The accuracy of the Branch and Bound algorithm is the same all the time. Both Greedy and Christofides' algorithms have high accuracy which tend to a straight line. It seems that Christofides' accuracy has a falling tendency, while Greedy's converges to some middle value.

4.3 Data Analysis

The hypothesis of dependence of the execution time of the algorithms on the datasets' sizes was partially correct. The Greedy algorithm's performance was overall better than Christofides' one, and the Branch and Bound heuristic had longer and longer execution time. For the datasets of 5 and 10 cities its times were smaller than ones of the two other algorithms and for 15 and 20 cities they became bigger. What is interesting are the points of interception on the graph 4.2.4. From the hypothesis it could be expected that between 4 and 5 cities there should be the point of interception of the Branch and Bound and Greedy algorithms' execution time, and between 5 and 6 cities would be the intercept of Branch and Bound and Christofides'

time graph. However, this is not the case. The Branch and Bound and Greedy algorithms' graphs intercept at 12 cities, and Branch and Bound and Christofides' at 13 cities. The points of interception are therefore moved by between 7 and 9 on the x axis comparing to what was expected from the mathematical analysis and hypothesis.

According to Muhammad Shaaban, some processors may do some operations quicker than other ones, as they might require less micro operations on them.[37] This means that the time of making one calculation by the processor is not the same in every case. The big O notation does take into account how quickly the number of operations that have to be made in the worst case scenario in order to find the solution grows.[38] Therefore the actual execution time might not be representative for the number of the operations done.

The hypothesis about the accuracy of the algorithms was overall right. The Greedy algorithm has lower general average accuracy ratio than the Christofides' heuristic, however, this is not a significant difference. What is more, each algorithm was more accurate than the other one equal number of times. In the case of 5 cities, the accuracy of the methods was the same; Greedy algorithm was more accurate for 15 and 25 cities, and Christofides' was better for 10 and 20 cities. The Branch and Bound, being the exact algorithm, gave the most optimal route for each dataset. However, its execution time became insufficiently long for the set of 25 cities.

4.4 Possible Improvements and Further Research Opportunities

In order to better analyze the algorithms, more datasets and bigger number of cities could be tested. More precise range of datasets sized could be used, so the datasets' sizes could be

---

[37] Muhammad Shaaban, *Central Processor Unit (CPU) & Computer System Performance Measures: CPI, CPU Execution Equation, Benchmarking, MIPS Rating, Amdahl's Law*, 2011 <meseec.ce.rit.edu/eecc550-winter2011/550-12-6-2011.pdf> [accessed 19 January 2022].

[38] Austin Mohr, *Quantum Computing in Complexity Theory and Theory of Computation*, 2007 <http://www.austinmohr.com/Work_files/complexity.pdf> [accessed 19 January 2022].

consecutively raised by 2, which could potentially lead to more precise results. An important limitation was the processor used in the experiment. As it was discussed in part 4.3, some processors may do some operations faster or slower. What is more, there might have been some processes running in the background when the algorithms' performance was measured, which might have influenced the final results. The investigation could be conducted on computers with different processors in order to further analyze the efficiency of the algorithms.

Moreover, there are many other types of algorithms that were not covered in this study. An important one could be dynamic programming approach, an example of which is Held-Karp algorithm, which has the complexity of $O(n^2 2^n)$ and is one of the most known approaches to the TSP.[39]

The execution time of the algorithm depends strongly on the computational power of the machine it is tested on. As the field of the quantum computing becomes increasingly significant, it is seen by some as the future of the computer science. As quantum computers can hypothetically have much more computational power than classical computers, the possibility of solving the TSP quicker than ever emerges. There even exists an algorithm for solving the TSP on the quantum machine, using IBM Quantum Experience cloud quantum computer.[40] However, it has its limitations, as the maximum number of q-bits (quantum bits) available currently on IBM's machines is 127.[41]

---

[39] Michael Held and Richard M. Karp, "A Dynamic Programming Approach to Sequencing Problems", *Journal of the Society for Industrial and Applied Mathematics*, 10.1 (1962), 196–210.

[40] Karthik Srinivasan and others, *Efficient Quantum Algorithm for Solving Travelling Salesman Problem: An IBM Quantum Experience*, 2018 <https://arxiv.org/pdf/1805.10928.pdf> [accessed 19 January 2022].

[41] IBM, "IBM Quantum Processor Types", *IBM Quantum*, 2021 <https://quantum-computing.ibm.com/services/docs/services/manage/systems/processors> [accessed 19 January 2022].

5. Conclusions

The aim of the study was to use the theory behind the Traveling Salesman Problem and three examples of different types of the algorithms for solving it and apply it practically to investigate the relationship between the execution time and accuracy of the calculated tour and the number of cities in the problem. As expected, the execution time of the Greedy heuristic is overall better than the Christofides' method, and the Branch and Bound algorithm tends to perform better than those two on smaller sets of data, but worse on the big ones. However, the number of cities which the Branch and Bound algorithm becomes slower is different than the one expected, which may be caused by processor's role in the computations. As the Branch and Bound is an exact algorithm and always outputs the shortest tour, its accuracy was the best. The accuracy of the Greedy and Christofides' solutions was not following any pattern when compared to the number of cities in the dataset. The overall accuracy of the Christofides' method was by 0.007, so 0.7% better than one of the Greedy algorithm.

Therefore, the answer to the research question of this essay – "to what extent Branch and Bound algorithm, Greedy algorithm and the Christofides' algorithm are efficient ways of solving the Travelling Salesman Problem (TSP)?" – is not simple. **For the datasets consisting of less than 12 cities the most efficient is the Branch and Bound algorithm, as it gives the exact result in the shortest time. For bigger datasets, the Christofides' has worse execution time than the Greedy method, however the second one is less accurate. That means that the efficiency of those two algorithms for datasets bigger than 12 cities is comparable.**

6. Reference list

Bibliography

Abdulkarim, Haider, and Ibrahim Alshammari, “Comparison of Algorithms for Solving Traveling Salesman Problem,” *International Journal of Engineering and Advanced Technology (IJEAT)*, 4.6 (2015), 76–79 <https://www.researchgate.net/publication/280597707_Comparison_of_Algorithms_for_Solving_Traveling_Salesman_Problem> [accessed 17 February 2021]

Black, Paul E., “Dictionary of Algorithms and Data Structures,” *Xlinux.nist.gov*, 2005 <https://xlinux.nist.gov/dads/> [accessed 29 June 2021]

Black, Paul E., and Paul J. Tanenbaum, “Graph,” *Dictionary of Algorithms and Data Structures*, 2005 <https://xlinux.nist.gov/dads/HTML/graph.html> [accessed 29 June 2021]

Bondy, John Adrian, and Uppaluri Siva Ramachandra Murty, *Graph Theory with Applications* (New York: North Holland, 1982) <https://www.iro.umontreal.ca/~hahn/IFT3545/GTWA.pdf> [accessed 28 June 2021]

Chauhan, Chetan, Ravindra Gupta, and Kshitij Pathak, “Survey of Methods of Solving TSP along with Its Implementation Using Dynamic Programming Approach,” *International Journal of Computer Applications*, 52.4 (2012), 12 <https://research.ijcaonline.org/volume52/number4/pxc3881550.pdf> [accessed 19 February 2021]

Christofides, Nicos, *Worst-Case Analysis of a New Heuristic for the Travelling Salesman Problem*, 1976 <https://apps.dtic.mil/sti/pdfs/ADA025602.pdf> [accessed 15 January 2022]

Clausen, Jens, *Branch and Bound Algorithms - Principles and Examples*, 1999 <https://imada.sdu.dk/~jbj/heuristikker/TSPtext.pdf> [accessed 24 August 2021]

Cook, William, and André Rohe, "Computing Minimum-Weight Perfect Matchings," *INFORMS Journal on Computing*, 11.2 (1999), 138–48 <https://www.math.uwaterloo.ca/~bico/papers/match_ijoc.pdf> [accessed 16 January 2022]

Held, Michael, and Richard M. Karp, "A Dynamic Programming Approach to Sequencing Problems," *Journal of the Society for Industrial and Applied Mathematics*, 10.1 (1962), 196–210 10.1137/0110015>

IBM, "IBM Quantum Processor Types," *IBM Quantum*, 2021 <https://quantum-computing.ibm.com/services/docs/services/manage/systems/processors> [accessed 19 January 2022]

jddeep003, "Travelling Salesman Problem | Greedy Approach," *GeeksforGeeks*, 2020 <https://www.geeksforgeeks.org/travelling-salesman-problem-greedy-approach/> [accessed 8 January 2022]

JGraphT, "jgrapht/jgrapht-core/src/main/java/org/jgrapht/alg/tour/ ChristofidesThreeHalvesApproxMetric TSP.java," *GitHub*, 2022 <https://github.com/jgrapht/jgrapht/blob/master/jgrapht-core/src/main/java/org/jgrapht/alg/tour/ ChristofidesThreeHalvesApproxMetric TSP.java> [accessed 27 January 2022]

JGraphT, “jgrapht/jgrapht-core/src/main/java/org/jgrapht/alg/tour/GreedyHeuristicTSP.java,” *GitHub*, 2022 <https://github.com/jgrapht/jgrapht/blob/master/jgrapht-core/src/main/java/org/jgrapht/alg/tour/GreedyHeuristicTSP.java> [accessed 27 January 2022]

Lazar, Emanuel, *5.6 Euler Paths and Cycles*, 2016 <https://www2.math.upenn.edu/~mlazar/math170/notes05-6.pdf> [accessed 16 January 2022]

Matai, Rajesh, Surya Singh, and Murari Lal, “Traveling Salesman Problem: An Overview of Applications, Formulations, and Solution Approaches,” in *Traveling Salesman Problem, Theory and Applications* (Rijeka: InTech, 2010), pp. 1–24 10.5772/12909>

Mataija, Mirta, Mirjana Rakamarić Šegić, and Franciska Jozić, “Solving the Travelling Salesman Problem Using the Branch and Bound Method,” *Zbornik Veleučilišta U Rijeci*, 4.1 (2016), 259–70 <https://hrcak.srce.hr/file/236378> [accessed 16 January 2022]

Mohr, Austin, *Quantum Computing in Complexity Theory and Theory of Computation*, 2007 <http://www.austinmohr.com/Work_files/complexity.pdf> [accessed 19 January 2022]

Narahari, Y., “8.4.2 Optimal Solution for TSP Using Branch and Bound,” *Gtl.csa.iisc.ac.in*, 2001 <https://gtl.csa.iisc.ac.in/dsa/node187.html> [accessed 15 January 2022]

Nešetřil, Jaroslav, Eva Milková, and Helena Nešetřilová, “Otakar Borůvka on Minimum Spanning Tree Problem Translation of Both the 1926 Papers, Comments, History,” *Discrete Mathematics*, 233.1-3 (2001), 3–36

<https://www.sciencedirect.com/science/article/pii/S0012365X00002247?via%3Dihub> [accessed 16 January 2022]

Nilsson, Christian, *Heuristics for the Traveling Salesman Problem*, 2003 <http://160592857366.free.fr/joe/ebooks/ShareData/Heuristics%20for%20the%20Traveling%20Salesman%20Problem%20By%20Christian%20Nillson.pdf> [accessed 15 January 2022]

Oracle, “System (Java Platform SE 8),” *Docs.oracle.com* <https://docs.oracle.com/javase/8/docs/api/java/lang/System.html#nanoTime--> [accessed 13 January 2022]

Pearl, Judea, *Heuristics : Intelligent Search Strategies for Computer Problem Solving* (Reading, Mass.: Addison-Wesley Pub. Co, 1984)

Rai, Anurag, “Traveling Salesman Problem Using Branch and Bound,” *GeeksforGeeks*, 2016 <https://www.geeksforgeeks.org/traveling-salesman-problem-using-branch-and-bound-2/> [accessed 8 January 2022]

Sahalot, Antima, and Sapna Shrimali, “A Comparative Study of Brute Force Method, Nearest Neighbour and Greedy Algorithms to Solve the Travelling Salesman Problem,” *IMPACT: International Journal of Research in Engineering & Technology*, 2.6 (2014), 59–72 10.1.1.684.8937>

Salvador, Tiago, “The Traveling Salesman Problem: A Statistical Approach” (2010), pp. 8–12 <http://www.math.lsa.umich.edu/~saldanha/Files/Report%20TSP.pdf> [accessed 29 June 2021]

Shaaban, Muhammad, *Central Processor Unit (CPU) & Computer System Performance Measures: CPI, CPU Execution Equation, Benchmarking, MIPS Rating, Amdahl's Law*, 2011 <http://meseec.ce.rit.edu/eecc550-winter2011/550-12-6-2011.pdf> [accessed 19 January 2022]

Srinivasan, Karthik, Saipriya Satyajit, Bikash Behera, and Prasanta Panigrahi, *Efficient Quantum Algorithm for Solving Travelling Salesman Problem: An IBM Quantum Experience*, 2018 <https://arxiv.org/pdf/1805.10928.pdf> [accessed 19 January 2022]

Vos, Daniël, "Basic Principles of the Traveling Salesman Problem and Radiation Hybrid Mapping" (unpublished Bachelor Thesis, 2016) <https://repository.tudelft.nl/islandora/object/uuid:0ebb0f3b-e352-4f3c-8465-5c6be2812a90/datastream/OBJ/download> [accessed 19 February 2021]

Williamson, David P, and David Bernard Shmoys, *The Design of Approximation Algorithms* (New York: Cambridge University Press, 2011)

Zhang, Weixiong, *Branch-And-Bound Search Algorithms and Their Computational Complexity*, 1996 <https://apps.dtic.mil/sti/pdfs/ADA314598.pdf> [accessed 16 January 2022]

7. Appendices

Appendix A: Code of the Algorithms Used

A.1: The Branch and Bound Algorithm

The code was taken from Geeks for Geeks website: https://www.geeksforgeeks.org/traveling-salesman-problem-using-branch-and-bound-2/

```
// Java program to solve Traveling Salesman Problem
// using Branch and Bound.
import java.util.*;

public class TSP_Branch {
    static int N = 25;

    // final_path[] stores the final solution ie, the
    // path of the salesman.
    static int final_path[] = new int[N + 1];

    // visited[] keeps track of the already visited nodes
    // in a particular path
    static boolean visited[] = new boolean[N];

    // Stores the final minimum weight of shortest tour.
    static int final_res = Integer.MAX_VALUE;

    // Function to copy temporary solution to
    // the final solution
    static void copyToFinal(int curr_path[])
    {
        for (int i = 0; i < N; i++)
            final_path[i] = curr_path[i];
```

```
    final_path[N] = curr_path[0];
}

// Function to find the minimum edge cost
// having an end at the vertex i
static int firstMin(int adj[][], int i)
{
    int min = Integer.MAX_VALUE;
    for (int k = 0; k < N; k++)
        if (adj[i][k] < min && i != k)
            min = adj[i][k];
    return min;
}

// function to find the second minimum edge cost
// having an end at the vertex i
static int secondMin(int adj[][], int i)
{
    int first = Integer.MAX_VALUE, second = Integer.MAX_VALUE;
    for (int j=0; j<N; j++)
    {
        if (i == j)
            continue;

        if (adj[i][j] <= first)
        {
            second = first;
            first = adj[i][j];
        }
        else if (adj[i][j] <= second &&
                adj[i][j] != first)
```

```
                second = adj[i][j];
        }
        return second;
}

// function that takes as arguments:
// curr_bound -> lower bound of the root node
// curr_weight-> stores the weight of the path so far
// level-> current level while moving in the search
//         space tree
// curr_path[] -> where the solution is being stored which
//             would later be copied to final_path[]
static void TSPRec(int adj[][], int curr_bound, int curr_weight,
                    int level, int curr_path[])
{
    // base case is when we have reached level N which
    // means we have covered all the nodes once
    if (level == N)
    {
        // check if there is an edge from last vertex in
        // path back to the first vertex
        if (adj[curr_path[level - 1]][curr_path[0]] != 0)
        {
            // curr_res has the total weight of the
            // solution we got
            int curr_res = curr_weight +
                    adj[curr_path[level-1]][curr_path[0]];

            // Update final result and final path if
            // current result is better.
            if (curr_res < final_res)
```

```
            {
                copyToFinal(curr_path);
                final_res = curr_res;
            }
        }
        return;
    }

    // for any other level iterate for all vertices to
    // build the search space tree recursively
    for (int i = 0; i < N; i++)
    {
        // Consider next vertex if it is not same (diagonal
        // entry in adjacency matrix and not visited
        // already)
        if (adj[curr_path[level-1]][i] != 0 &&
                visited[i] == false)
        {
            int temp = curr_bound;
            curr_weight += adj[curr_path[level - 1]][i];

            // different computation of curr_bound for
            // level 2 from the other levels
            if (level==1)
                curr_bound -= ((firstMin(adj, curr_path[level - 1]) +
                        firstMin(adj, i))/2);
            else
                curr_bound -= ((secondMin(adj, curr_path[level - 1]) +
                        firstMin(adj, i))/2);

            // curr_bound + curr_weight is the actual lower bound
```

```
            // for the node that we have arrived on
            // If current lower bound < final_res, we need to explore
            // the node further
            if (curr_bound + curr_weight < final_res)
            {
                curr_path[level] = i;
                visited[i] = true;

                // call TSPRec for the next level
                TSPRec(adj, curr_bound, curr_weight, level + 1,
                        curr_path);
            }

            // Else we have to prune the node by resetting
            // all changes to curr_weight and curr_bound
            curr_weight -= adj[curr_path[level-1]][i];
            curr_bound = temp;

            // Also reset the visited array
            Arrays.fill(visited,false);
            for (int j = 0; j <= level - 1; j++)
                visited[curr_path[j]] = true;
        }
    }
}

// This function sets up final_path[]
static void TSP(int adj[][])
{
    int curr_path[] = new int[N + 1];
```

```
    // Calculate initial lower bound for the root node
    // using the formula 1/2 * (sum of first min +
    // second min) for all edges.
    // Also initialize the curr_path and visited array
    int curr_bound = 0;
    Arrays.fill(curr_path, -1);
    Arrays.fill(visited, false);

    // Compute initial bound
    for (int i = 0; i < N; i++)
        curr_bound += (firstMin(adj, i) +
                    secondMin(adj, i));

    // Rounding off the lower bound to an integer
    curr_bound = (curr_bound==1)? curr_bound/2 + 1 :
                curr_bound/2;

    // We start at vertex 1 so the first vertex
    // in curr_path[] is 0
    visited[0] = true;
    curr_path[0] = 0;

    // Call to TSPRec for curr_weight equal to
    // 0 and level 1
    TSPRec(adj, curr_bound, 0, 1, curr_path);
}

// Driver code
public static void main(String[] args)
{
    //Adjacency matrix for the given graph
```

```
        int adj[][] = {
           //here goes the adjacency matrix
        };
        long start = System.nanoTime();
        TSP(adj);
        long end = System.nanoTime();
        System.out.println("Location: " + (end - start));

        System.out.printf("Minimum cost : %d\n", final_res);
    }
}

/* This code contributed by PrinciRaj1992 */
```

A.2: The Greedy Algorithm and the Christofides' Algorithm

Both of the codes for those algorithms were taken from the jgrapht.alg.tour library. They are available as the GitHub repository, the Greedy algorithm code: https://github.com/jgrapht/jgrapht/blob/master/jgrapht-core/src/main/java/org/jgrapht/alg/tour/GreedyHeuristicTSP.java

and the Christofides' method code:

https://github.com/jgrapht/jgrapht/blob/master/jgrapht-core/src/main/java/org/jgrapht/alg/tour/ChristofidesThreeHalvesApproxMetricTSP.java.

Therefore the code used in the experiment looked like this:

```
import org.jgrapht.*;
import org.jgrapht.alg.tour.GreedyHeuristicTSP;
import org.jgrapht.graph.*;
import java.net.*;
```

```
public class TSP_Greedy {
    public static void main (String[] args) throws URISyntaxException {

        int adj[][] = {
                //here was the adjacency matrix
        };

        Graph<URI, DefaultWeightedEdge> g = new
DefaultUndirectedWeightedGraph<>(DefaultWeightedEdge.class);

//here was the code for creating the nodes and assigning the edges' weight

        GreedyHeuristicTSP algorithm = new GreedyHeuristicTSP();

        long start = System.nanoTime();
        // Function Call
        GraphPath result = algorithm.getTour(g);
        long end = System.nanoTime();

        System.out.println(result.getWeight());
        System.out.println("Location: " + (end - start));
    }
}
```

for the Greedy algorithm and so:

```
import org.jgrapht.*;
import org.jgrapht.alg.tour.ChristofidesThreeHalvesApproxMetricTSP;
import org.jgrapht.graph.*;
```

```
import java.net.*;

public class TSP_Christofides {
    public static void main (String[] args) throws URISyntaxException {

        int adj[][] = {
                //here was the adjacency matrix
        };

        Graph<URI, DefaultWeightedEdge> g = new
DefaultUndirectedWeightedGraph<>(DefaultWeightedEdge.class);

//here was the code for creating the nodes and assigning the edges’ weight

        ChristofidesThreeHalvesApproxMetricTSP algorithm = new
ChristofidesThreeHalvesApproxMetricTSP();

        long start = System.nanoTime();
        // Function Call
        GraphPath result = algorithm.getTour(g);
        long end = System.nanoTime();

        System.out.println(result.getWeight());
        System.out.println("Location: " + (end - start));
    }
}
```

for the Christofides’ algorithm.

Appendix B: The Datasets and Their Representation

The Branch and Bound as an input needed a matrix representation of the weight of the edges between the vertices. That means that if there is a square matrix of a size n, the [i][j] element represents the weight of the edge between the nodes i and j.

The Greedy and Christofides' algorithms needed a graph representation of the problem. The jgrapht library was used to obtain those and assign the weights of the edges using the data from the matrices. An example below created a graph with two nodes and one undirected weighted edge in order to show functions that were used to create the graphs:

```
//adjacency matrix for the example graph
int adj[][] = {
{0, 1},
{1, 0}
};

//defining the undirected weighted graph g
Graph<URI, DefaultWeightedEdge> g = new
DefaultUndirectedWeightedGraph<>(DefaultWeightedEdge.class);

//defining the vertices A and B(nodes/cities)
URI A = new URI("A");
URI B = new URI("B");

//assigning the vertices A and B to the graph
g.addVertex(A);
g.addVertex(A);

//creating an edge between A and B
g.addEdge(A, B);
```

```
//assigning the weight to the edge

DefaultWeightedEdge AB = g.getEdge(A, B);

g.setEdgeWeight(AB, adj[0][1]);
```

The code was analogous for the bigger sets of data.

B.1 Matrices for 5 cities datasets

Dataset 1 for 5 cities problem

0	3	4	2	7
3	0	4	6	3
4	4	0	5	8
2	6	5	0	6
7	3	8	6	0

Dataset 2 for 5 cities problem

0	27	12	17	11
27	0	16	11	29
12	16	0	6	12
17	11	6	0	18
11	29	12	18	0

Dataset 3 for 5 cities problem

0	64	378	519	434
64	0	318	455	375
378	318	0	170	265
519	455	170	0	223
434	375	265	223	0

B.2 Matrices for 10 cities datasets

Dataset 1 for 10 cities problem

0	8	20	31	12	48	36	2	5	39
8	0	38	9	33	37	22	6	4	14
50	38	0	11	55	1	23	46	41	17
31	9	11	0	44	13	16	19	25	18
12	33	55	44	0	54	53	30	28	45
48	37	1	13	54	0	26	47	40	24
36	22	23	16	53	26	0	29	35	34
2	6	46	19	30	47	29	0	3	27
5	4	41	25	28	40	35	3	0	20
39	14	17	18	45	24	34	27	20	0

Dataset 2 for 10 cities problem

0	300	325	466	217	238	431	336	451	47
300	0	638	180	595	190	138	271	229	236
325	638	0	251	88	401	189	386	565	206

466	180	251	0	139	371	169	316	180	284
217	595	88	139	0	310	211	295	474	130
238	190	401	371	310	0	202	122	378	157
431	138	189	169	211	202	0	183	67	268
336	271	386	316	295	122	183	0	483	155
451	229	565	180	474	378	67	483	0	299
47	236	206	284	130	157	268	155	299	0

Dataset 3 for 10 cities problem

0	2451	713	1018	1631	1374	2408	213	2571	875
2451	0	1745	1524	831	1240	959	2596	403	1589
713	1745	0	355	920	803	1737	851	1858	262
1018	1524	355	0	700	862	1395	1123	1584	466
1631	831	920	700	0	663	1021	1769	949	796
1374	1240	803	862	663	0	1681	1551	1765	547
2408	959	1737	1395	1021	1681	0	2493	678	1724
213	2596	851	1123	1769	1551	2493	0	2699	1038
2571	403	1858	1584	949	1765	678	2699	0	1744
875	1589	262	466	796	547	1724	1038	1744	0

B.3 Matrices for 15 cities datasets

Dataset 1 for 15 cities problem

0	29	82	46	68	52	72	42	51	55	29	74	23	72	46

29	0	55	46	42	43	43	23	23	31	41	51	11	52	21
82	55	0	68	46	55	23	43	41	29	79	21	64	31	51
46	46	68	0	82	15	72	31	62	42	21	51	51	43	64
68	42	46	82	0	74	23	52	21	46	82	58	46	65	23
52	43	55	15	74	0	61	23	55	31	33	37	51	29	59
72	43	23	72	23	61	0	42	23	31	77	37	51	46	33
42	23	43	31	52	23	42	0	33	15	37	33	33	31	37
51	23	41	62	21	55	23	33	0	29	62	46	29	51	11
55	31	29	42	46	31	31	15	29	0	51	21	41	23	37
29	41	79	21	82	33	77	37	62	51	0	65	42	59	61
74	51	21	51	58	37	37	33	46	21	65	0	61	11	55
23	11	64	51	46	51	51	33	29	41	42	61	0	62	23
72	52	31	43	65	29	46	31	51	23	59	11	62	0	59
46	21	51	64	23	59	33	37	11	37	61	55	23	59	0

Dataset 2 for 15 cities problem

0	141	134	152	173	289	326	329	285	401	388	366	343	305	276
141	0	152	150	153	312	354	313	249	324	300	272	247	201	176
134	152	0	24	48	168	210	197	153	280	272	257	237	210	181
152	150	24	0	24	163	206	182	133	257	248	233	214	187	158
173	153	48	24	0	160	203	167	114	234	225	210	190	165	137
289	312	168	163	160	0	43	90	124	250	264	270	264	267	249
326	354	210	206	203	43	0	108	157	271	290	299	295	303	287
329	313	197	182	167	90	108	0	70	164	183	195	194	210	201

285	249	153	133	114	124	157	70	0	141	147	148	140	147	134
401	324	280	257	234	250	271	164	141	0	36	67	88	134	150
388	300	272	248	225	264	290	183	147	36	0	33	57	104	124
366	272	257	233	210	270	299	195	148	67	33	0	26	73	96
343	247	237	214	190	264	295	194	140	88	57	26	0	48	71
305	201	210	187	165	267	303	210	147	134	104	73	48	0	30
276	176	181	158	137	249	287	201	134	150	124	96	71	30	0

Dataset 3 for 15 cities problem

0	633	257	91	412	150	80	134	259	505	353	324	70	211	268
633	0	390	661	227	488	572	530	555	289	282	638	567	466	420
257	390	0	228	169	112	196	154	372	262	110	437	191	74	53
91	661	228	0	383	120	77	105	175	476	324	240	27	182	239
412	227	169	383	0	267	351	309	338	196	61	421	346	243	199
150	488	112	120	267	0	63	34	264	360	208	329	83	105	123
80	572	196	77	351	63	0	29	232	444	292	297	47	150	207
134	530	154	105	309	34	29	0	249	402	250	314	68	108	165
259	555	372	175	338	264	232	249	0	495	352	95	189	326	383
505	289	262	476	196	360	444	402	495	0	154	578	439	336	240
353	282	110	324	61	208	292	250	352	154	0	435	287	184	140
324	638	437	240	421	329	297	314	95	578	435	0	254	391	448
70	567	191	27	346	83	47	68	189	439	287	254	0	145	202
211	466	74	182	243	105	150	108	326	336	184	391	145	0	57
268	420	53	239	199	123	207	165	383	240	140	448	202	57	0

B.4 Matrices for 20 cities datasets

Dataset 1 for 20 cities problem

0	144	114	105	31	109	135	132	85	79	158	20	73	162	127	190	156	58	87	71
144	0	144	181	147	76	195	73	64	114	220	135	71	18	39	60	37	101	62	146
114	144	0	49	86	169	51	78	130	42	76	94	114	154	105	151	125	137	94	46
105	181	49	0	73	189	31	124	152	67	52	88	135	195	146	197	169	147	123	40
31	147	86	73	0	128	104	119	97	57	126	17	82	164	122	184	151	80	85	40
109	76	169	189	128	0	212	126	38	128	238	112	54	92	95	137	110	51	77	148
135	195	51	31	104	212	0	129	174	85	26	118	157	206	157	201	176	173	141	67
132	73	78	124	119	126	129	0	92	65	153	115	84	80	35	73	47	118	55	98
85	64	130	152	97	38	174	92	0	90	200	82	17	82	66	120	89	36	39	112
79	114	42	67	57	128	85	65	90	0	111	59	73	128	80	137	106	95	57	33
158	220	76	52	126	238	26	153	200	111	0	141	183	231	182	224	201	198	167	91
20	135	94	88	17	112	118	115	82	59	141	0	67	153	114	177	142	63	75	52
73	71	114	135	82	54	157	84	17	73	183	67	0	90	64	123	89	35	28	95
162	18	154	195	164	92	206	80	82	128	231	153	90	0	49	47	35	119	79	161
127	39	105	146	122	95	157	35	66	80	182	114	64	49	0	62	28	99	40	113
190	60	151	197	184	137	201	73	120	137	224	177	123	47	62	0	34	156	102	170
156	37	125	169	151	110	176	47	89	106	201	142	89	35	28	34	0	123	68	139
58	101	137	147	80	51	173	118	36	95	198	63	35	119	99	156	123	0	63	106
87	62	94	123	85	77	141	55	39	57	167	75	28	79	40	102	68	63	0	85
71	146	46	40	40	148	67	98	112	33	91	52	95	161	113	170	139	106	85	0

Dataset 2 for 20 cities problem

0	**74**	**4110**	**3048**	**2267**	**974**	**4190**	**3302**	**4758**	**3044**	**3095**	**3986**	**5093**	**6407**	**5904**	**8436**	**6963**	**6694**	**6576**	**8009**
74	**0**	**4070**	**3000**	**2214**	**901**	**4138**	**3240**	**4702**	**2971**	**3021**	**3915**	**5025**	**6338**	**5830**	**8369**	**6891**	**6620**	**6502**	**7939**
4110	**4070**	**0**	**1173**	**1973**	**3496**	**892**	**1816**	**1417**	**3674**	**3778**	**2997**	**2877**	**3905**	**5057**	**5442**	**4991**	**5151**	**5316**	**5596**
3048	**3000**	**1173**	**0**	**817**	**2350**	**1172**	**996**	**1797**	**2649**	**2756**	**2317**	**2721**	**3974**	**4548**	**5802**	**4884**	**4887**	**4960**	**5696**
2267	**2214**	**1973**	**817**	**0**	**1533**	**1924**	**1189**	**2498**	**2209**	**2312**	**2325**	**3089**	**4401**	**4558**	**6342**	**5175**	**5072**	**5075**	**6094**
974	**901**	**3496**	**2350**	**1533**	**0**	**3417**	**2411**	**3936**	**2114**	**2175**	**3014**	**4142**	**5450**	**4956**	**7491**	**5990**	**5725**	**5615**	**7040**
4190	**4138**	**892**	**1172**	**1924**	**3417**	**0**	**1233**	**652**	**3086**	**3185**	**2203**	**1987**	**3064**	**4180**	**4734**	**4117**	**4261**	**4425**	**4776**
3302	**3240**	**1816**	**996**	**1189**	**2411**	**1233**	**0**	**1587**	**1877**	**1979**	**1321**	**1900**	**3214**	**3556**	**5175**	**4006**	**3947**	**3992**	**4906**
4758	**4702**	**1417**	**1797**	**2498**	**3936**	**652**	**1587**	**0**	**3286**	**3374**	**2178**	**1576**	**2491**	**3884**	**4088**	**3601**	**3818**	**4029**	**4180**
3044	**2971**	**3674**	**2649**	**2209**	**2114**	**3086**	**1877**	**3286**	**0**	**107**	**1360**	**2675**	**3822**	**2865**	**5890**	**4090**	**3723**	**3560**	**5217**
3095	**3021**	**3778**	**2756**	**2312**	**2175**	**3185**	**1979**	**3374**	**107**	**0**	**1413**	**2725**	**3852**	**2826**	**5916**	**4088**	**3705**	**3531**	**5222**
3986	**3915**	**2997**	**2317**	**2325**	**3014**	**2203**	**1321**	**2178**	**1360**	**1413**	**0**	**1315**	**2511**	**2251**	**4584**	**2981**	**2778**	**2753**	**4031**
5093	**5025**	**2877**	**2721**	**3089**	**4142**	**1987**	**1900**	**1576**	**2675**	**2725**	**1315**	**0**	**1323**	**2331**	**3350**	**2172**	**2275**	**2458**	**3007**
6407	**6338**	**3905**	**3974**	**4401**	**5450**	**3064**	**3214**	**2491**	**3822**	**3852**	**2511**	**1323**	**0**	**2350**	**2074**	**1203**	**1671**	**2041**	**1725**
5904	**5830**	**5057**	**4548**	**4558**	**4956**	**4180**	**3556**	**3884**	**2865**	**2826**	**2251**	**2331**	**2350**	**0**	**3951**	**1740**	**1108**	**772**	**2880**

8436	8369	5442	5802	6342	7491	4734	5175	4088	5890	5916	4584	3350	2074	3951	0	2222	2898	3325	1276
6963	6891	4991	4884	5175	5990	4117	4006	3601	4090	4088	2981	2172	1203	1740	2222	0	684	1116	1173
6694	6620	5151	4887	5072	5725	4261	3947	3818	3723	3705	2778	2275	1671	1108	2898	684	0	432	1776
6576	6502	5316	4960	5075	5615	4425	3992	4029	3560	3531	2753	2458	2041	772	3325	1116	432	0	2174
8009	7939	5596	5696	6094	7040	4776	4906	4180	5217	5222	4031	3007	1725	2880	1276	1173	1776	2174	0

Dataset 3 for 20 cities problem

0	39	22	59	54	33	57	32	89	73	29	46	16	83	120	45	24	32	36	25
39	0	20	20	81	8	49	64	63	84	10	61	25	49	81	81	58	16	72	60
22	20	0	39	74	18	60	44	71	73	11	46	6	61	99	61	37	10	51	40
59	20	39	0	93	27	51	81	48	80	30	69	45	32	61	97	75	31	89	78
54	81	74	93	0	73	43	56	104	76	76	77	69	111	72	46	56	84	49	53
33	8	18	27	73	0	45	61	71	88	8	63	22	57	87	77	54	18	68	56
57	49	60	51	43	45	0	85	88	115	52	103	60	75	64	85	79	63	83	78
32	64	44	81	56	61	85	0	74	43	55	23	40	81	97	17	8	50	8	7
89	63	71	48	104	71	88	74	0	38	69	51	75	16	35	75	77	61	77	80
73	84	73	80	76	88	115	43	38	0	81	28	72	53	55	38	49	70	42	50
29	10	11	30	76	8	52	55	69	81	0	55	16	57	91	71	48	11	62	50
46	61	46	69	77	63	103	23	51	28	55	0	44	59	81	32	26	46	29	29
16	25	6	45	69	22	60	40	75	72	16	44	0	67	105	56	33	16	46	35
83	49	61	32	111	57	75	81	16	53	57	59	67	0	39	88	82	51	87	85
120	81	99	61	72	87	64	97	35	55	91	81	105	39	0	84	104	90	93	104
45	81	61	97	46	77	85	17	75	38	71	32	56	88	84	0	23	67	9	21
24	58	37	75	56	54	79	8	77	49	48	26	33	82	104	23	0	44	14	3
32	16	10	32	84	18	63	50	61	70	11	46	16	51	90	67	44	0	58	47
36	72	51	89	49	68	83	8	77	42	62	29	46	87	93	9	14	58	0	12
25	60	40	78	53	56	78	7	80	50	50	29	35	85	104	21	3	47	12	0

B.5 Matrices for 25 cities datasets

Dataset 1 for 25 cities problem

0	83	93	129	133	139	151	169	135	114	110	98	99	95	81	152	159	181	172	185	147	157	185	220	127
83	0	40	53	62	64	91	116	93	84	95	98	89	68	67	127	156	175	152	165	160	180	223	268	179
93	40	0	42	42	49	59	81	54	44	58	64	54	31	36	86	117	135	112	125	124	147	193	241	157
129	53	42	0	11	11	46	72	65	70	88	100	89	66	76	102	142	156	127	139	155	180	228	278	197
133	62	42	11	0	9	35	61	55	62	82	95	84	62	74	93	133	146	117	128	148	173	222	272	194
139	64	49	11	9	0	39	65	63	71	90	103	92	71	82	100	141	153	124	135	156	181	230	280	202
151	91	59	46	35	39	0	26	34	52	71	88	77	63	78	66	110	119	88	98	130	156	206	257	188
169	116	81	72	61	65	26	0	37	59	75	92	83	76	91	54	98	103	70	78	122	148	198	250	188
135	93	54	65	55	63	34	37	0	22	39	56	47	40	55	37	78	91	62	74	96	122	172	223	155
114	84	44	70	62	71	52	59	22	0	20	36	26	20	34	43	74	91	68	82	86	111	160	210	136
110	95	58	88	82	90	71	75	39	20	0	18	11	27	32	42	61	80	64	77	68	92	140	190	116
98	98	64	100	95	103	88	92	56	36	18	0	11	34	31	56	63	85	75	87	62	83	129	178	100

99	89	54	89	84	92	77	83	47	26	11	11	0	23	24	53	68	89	74	87	71	93	140	189	111
95	68	31	66	62	71	63	76	40	20	27	34	23	0	15	62	87	106	87	100	93	116	163	212	132
81	67	36	76	74	82	78	91	55	34	32	31	24	15	0	73	92	112	96	109	93	113	158	205	122
152	127	86	102	93	100	66	54	37	43	42	56	53	62	73	0	44	54	26	39	68	94	144	196	139
159	156	117	142	133	141	110	98	78	74	61	63	68	87	92	44	0	22	34	38	30	53	102	154	109
181	175	135	156	146	153	119	103	91	91	80	85	89	106	112	54	22	0	33	29	46	64	107	157	125
172	152	112	127	117	124	88	70	62	68	64	75	74	87	96	26	34	33	0	13	63	87	135	186	141
185	165	125	139	128	135	98	78	74	82	77	87	87	100	109	39	38	29	13	0	68	90	136	186	148
147	160	124	155	148	156	130	122	96	86	68	62	71	93	93	68	30	46	63	68	0	26	77	128	80
157	180	147	180	173	181	156	148	122	111	92	83	93	116	113	94	53	64	87	90	26	0	50	102	65
185	223	193	228	222	230	206	198	172	160	140	129	140	163	158	144	102	107	135	136	77	50	0	51	64
220	268	241	278	272	280	257	250	223	210	190	178	189	212	205	196	154	157	186	186	128	102	51	0	93

127	179	157	197	194	202	188	188	155	136	116	100	111	132	122	139	109	125	141	148	80	65	64	93	0

Dataset 2 for 25 cities problem

0	146	145	181	77	147	76	100	195	151	49	84	64	114	29	108	135	32	64	88	18	153	145	39	188
146	0	35	40	80	40	148	51	67	30	109	99	112	33	165	65	52	154	92	74	161	20	50	113	43
145	35	0	69	69	5	129	45	100	66	100	79	97	52	158	43	19	160	104	60	162	20	15	119	66
181	40	69	0	120	73	189	91	31	33	149	140	152	67	203	105	88	185	122	115	195	49	84	146	13
77	80	69	120	0	71	70	29	141	96	31	27	32	57	89	31	58	97	57	11	94	81	68	60	123
147	40	5	73	71	0	128	48	104	71	102	79	97	57	160	43	17	163	107	61	164	24	10	122	69
76	148	129	189	70	128	0	98	212	167	50	50	38	128	61	86	112	109	109	74	92	146	121	95	190
100	51	45	91	29	48	98	0	114	69	59	50	61	32	116	26	40	115	61	24	117	53	49	74	94
195	67	100	31	141	104	212	114	0	45	168	164	174	85	219	132	118	193	132	138	206	79	114	157	41
151	30	66	33	96	71	167	69	45	0	123	120	129	39	174	90	82	152	90	94	163	50	81	114	44

49	109	100	149	31	102	50	59	168	123	0	35	18	83	58	60	88	76	59	40	68	112	98	48	152
84	99	79	140	27	79	50	50	164	120	35	0	21	82	87	36	63	111	82	25	102	96	73	80	140
64	112	97	152	32	97	38	61	174	129	18	21	0	90	65	54	82	93	75	38	82	111	92	66	155
114	33	52	67	57	57	128	32	85	39	83	82	90	0	136	57	59	120	59	56	128	48	64	80	73
29	165	158	203	89	160	61	116	219	174	58	87	65	136	0	117	146	58	91	98	35	170	156	67	208
108	65	43	105	31	43	86	26	132	90	60	36	54	57	117	0	28	128	82	19	126	60	38	90	105
135	52	19	88	58	17	112	40	118	82	88	63	82	59	146	28	0	154	102	47	153	39	10	114	85
32	154	160	185	97	163	109	115	193	152	76	111	93	120	58	128	154	0	62	109	23	165	163	41	193
64	92	104	122	57	107	109	61	132	90	59	82	75	59	91	82	102	62	0	67	73	105	110	25	131
88	74	60	115	11	61	74	24	138	94	40	25	38	56	98	19	47	109	67	0	106	73	58	71	117
18	161	162	195	94	164	92	117	206	163	68	102	82	128	35	126	153	23	73	106	0	169	163	49	202
153	20	20	49	81	24	146	53	79	50	112	96	111	48	170	60	39	165	105	73	169	0	34	124	46

145	50	15	84	68	10	121	49	114	81	98	73	92	64	156	38	10	163	110	58	163	34	0	123	79
39	113	119	146	60	122	95	74	157	114	48	80	66	80	67	90	114	41	25	71	49	124	123	0	153
188	43	66	13	123	69	190	94	41	44	152	140	155	73	208	105	85	193	131	117	202	46	79	153	0

Dataset 3 for 25 cities problem

0	681	335	725	1215	657	210	1325	372	227	33	121	1175	1371	697	190	462	202	305	1846	1319	294	923	929	660
681	0	675	152	1537	1080	886	1494	694	454	700	560	810	1435	892	709	417	605	755	2099	828	510	399	316	685
335	675	0	827	1547	405	364	1660	704	518	355	412	1422	1689	1032	522	753	537	80	2178	1503	585	1074	793	951
725	152	827	0	1470	1232	930	1373	657	498	744	604	658	1294	825	753	365	605	907	1989	676	501	247	430	604
1215	1537	1547	1470	0	1825	1378	580	843	1136	1192	1172	1172	681	672	1136	1135	1018	1517	631	1414	1027	1452	1853	937
657	1080	405	1232	1825	0	447	1949	982	879	633	773	1827	1995	1321	699	1114	853	402	2456	1908	946	1479	1198	1312

210	886	364	930	1378	447	0	1502	535	432	186	326	1380	1548	874	252	667	406	284	2009	1524	499	1128	1113	865
1325	1494	1660	1373	580	1949	1502	0	967	1204	1316	1268	885	101	628	1260	1107	1123	1630	712	1127	1095	1296	1803	809
372	694	704	657	843	982	535	967	0	293	349	329	937	1013	339	293	292	175	674	1474	1094	184	751	1010	393
227	454	518	498	1136	879	432	1204	293	0	246	106	983	1213	576	255	236	151	488	1767	1093	109	696	770	468
33	700	355	744	1192	633	186	1316	349	246	0	140	1194	1362	688	167	481	220	325	1823	1338	313	942	962	679
121	560	412	604	1172	773	326	1268	329	106	140	0	1054	1277	640	185	341	154	382	1803	1198	173	802	859	539
1175	810	1422	658	1172	1827	1380	885	937	983	1194	1054	0	784	731	1164	749	997	1436	1597	242	881	411	1007	562
1371	1435	1689	1294	681	1995	1548	101	1013	1213	1362	1277	784	0	674	1306	1048	1169	1659	813	1026	1104	1195	1724	750
697	892	1032	825	672	1321	874	628	339	576	688	640	731	674	0	632	505	495	1002	1207	973	467	834	1208	271

190	709	522	753	1136	699	252	1260	293	255	167	185	1164	1306	632	0	451	167	492	1767	1308	283	941	1025	643
462	417	753	365	1135	1114	667	1107	292	236	481	341	749	1048	505	451	0	284	723	1712	857	168	490	733	298
202	605	537	605	1018	853	406	1123	175	151	220	154	997	1169	495	167	284	0	507	1649	1141	116	774	921	476
305	755	80	907	1517	402	284	1630	674	488	325	382	1436	1659	1002	492	723	507	0	2148	1580	555	1154	873	921
1846	2099	2178	1989	631	2456	2009	712	1474	1767	1823	1803	1597	813	1207	1767	1712	1649	2148	0	1839	1658	1940	2415	1425
1319	828	1503	676	1414	1908	1524	1127	1094	1093	1338	1198	242	1026	973	1308	857	1141	1580	1839	0	1025	429	923	804
294	510	585	501	1027	946	499	1095	184	109	313	173	881	1104	467	283	168	116	555	1658	1025	0	658	826	366
923	399	1074	247	1452	1479	1128	1296	751	696	942	802	411	1195	834	941	490	774	1154	1940	429	658	0	596	563
929	316	793	430	1853	1198	1113	1803	1010	770	962	859	1007	1724	1208	1025	733	921	873	2415	923	826	596	0	1001

XXX

660	685	951	604	937	1312	865	809	393	468	679	539	562	750	271	643	298	476	921	1425	804	366	563	1001	0

Appendix C: Raw Data Tables

Table C.1: Execution time for 5 cities datasets

For 5 cities	Dataset 1 [ns]	Dataset 2 [ns]	Dataset 3 [ns]
Branch and Bound	194.400	183.400	59.500
	223.000	171.500	132.500
	134.000	158.900	74.300
	130.300	166.300	68.100
	103.700	156.000	91.900
Greedy	26673.400	31479.400	22900.800
	28784.600	27000.100	27029.200
	24204.100	30058.700	27804.000
	25795.200	24730.900	27396.300
	31906.500	27571.800	31047.700
Christofides'	86633.700	53487.800	57711.300
	88809.100	51471.500	61522.000
	67883.700	58842.500	44465.700
	75021.400	52108.600	78825.400
	79499.600	59324.400	51810.400

Table C.2: Execution time for 10 cities datasets

For 10 cities	Dataset 1 [ns]	Dataset 2 [ns]	Dataset 3 [ns]
Branch and Bound	10615.900	2404.400	5880.600
	8711.900	2260.600	6846.100
	6532.200	1984.900	6455.400

	8301.900	2568.900	6950.700
	8217.900	3327.000	6493.200
Greedy	39850.000	27820.800	34472.000
	31839.500	27435.500	27422.900
	34083.500	25443.900	30412.400
	27501.200	29586.600	27898.300
	27076.400	26681.400	24315.800
Christofides'	78907.500	92093.400	66730.800
	84158.200	74781.500	59165.400
	84910.300	82083.900	63720.200
	67502.100	85253.400	72450.600
	63661.500	78856.600	68284.800

Table C.3: Execution time for 15 cities datasets

For 15 cities	Dataset 1 [ns]	Dataset 2 [ns]	Dataset 3 [ns]
Branch and Bound	29420.000	141492.100	1127321.300
	24925.800	151618.200	1116487.200
	27818.600	140057.100	1078828.600
	25152.200	136331.400	1075252.900
	27336.700	131035.400	1077210.100
Greedy	30412.800	34184.300	29985.500
	31361.400	28867.800	33465.700
	31635.800	26841.000	29038.500
	36072.800	33101.500	37826.400

	37281.300	29945.100	25253.900
Christofides'	76245.800	72271.300	81706.100
	64483.300	73846.100	81223.900
	78416.400	79065.100	100042.600
	69372.500	55722.300	72376.600
	71426.800	78178.900	84742.700

Table C.4: Execution time for 20 cities datasets

For 20 cities	Dataset 1 [ns]	Dataset 2 [ns]	Dataset 3 [ns]
Branch and Bound	10265884.500	1284115.000	3416386.000
	9896777.400	1256880.500	3353258.200
	10062314.000	1300893.400	3385011.400
	9949965.200	1251569.200	3563624.100
	9975528.300	1263951.900	3521932.700
Greedy	29410.700	24944.000	28944.300
	30436.600	29621.300	31052.900
	35510.600	34741.200	38610.200
	34822.100	31258.600	42995.000
	42919.200	43578.200	37572.400
Christofides'	75347.800	74532.400	97315.200
	85982.200	79440.900	63186.000
	67654.900	70260.800	65660.500
	65193.900	80358.300	80174.700
	79809.500	90503.900	90770.400

Table C.5: Execution time for 25 cities datasets

For 25 cities	Dataset 1 [ns]	Dataset 2 [ns]	Dataset 3 [ns]
Branch and Bound	35605942.600	no data	no data
	33783605.200	no data	no data
	34211855.800	no data	no data
	32453998.500	no data	no data
	34324620.800	no data	no data
Greedy	41037.800	40994.300	33195.700
	47728.400	40201.500	35287.100
	49821.700	43434.700	35546.600
	52225.800	32861.800	34650.000
	31193.600	35627.700	34396.800
Christofides'	68726.800	83356.200	63665.800
	92017.800	83038.500	91585.000
	72834.200	76036.100	76473.800
	76135.900	61849.200	107655.200
	77515.200	53797.300	92974.000

Table C.6 Tour lengths and accuracy for 5 cities datasets

For 5 cities	Dataset 1	Dataset 2	Dataset 3
Branch and Bound	19	67	1209
Greedy	21	69	1209

Christofides’	21	69	1209

Table C.7 Tour lengths and accuracy for 10 cities datasets

For 10 cities	Dataset 1	Dataset 2	Dataset 3	Ratio 3
Branch and Bound	135	1435	6811	1
Greedy	153	1744	7463	0.912635669
Christofides’	144	1513	6811	1

Table C.8 Tour lengths and accuracy for 15 cities datasets

For 15 cities	Dataset 1	Dataset 2	Dataset 3
Branch and Bound	291	1194	1908
Greedy	291	1260	2091
Christofides’	325	1411	2030

Table C.9 Tour lengths and accuracy for 20 cities datasets

For 20 cities	Dataset 1	Dataset 2	Dataset 3
Branch and Bound	769	23339	444
Greedy	973	26564	456
Christofides’	867	24076	504

Table C.10 Tour lengths and accuracy for 25 cities datasets

For 25 cities	Dataset 1	Dataset 2	Dataset 3

Branch and Bound	902	No data	No data
Greedy	953	796	9100
Christofides’	981	787	8842