Karlsruher Institut für Technologie

INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

Master thesis

Deterministic construction heuristics for the time-dependent travelling salesman problem

by

Maximilian Zimmermann

Industrial Engineering

Date of submission

30.09.2021

Supervisors KIT:

Prof. Dr. Stefan Nickel

Anika Pomes

Supervisors INSA:

Prof. Dr. Khaled Hadj-Hamou

Dr. Anne-Laure Ladier

Fayçal Atik Touzout

ABSTRACT

The time-dependent travelling salesman problem (TD-TSP) extends the TSP by considering the travel time between two locations as time-dependent. As the TD-TSP is NP-hard, it is difficult to find an optimal solution for large-sized instances. In this case, construction heuristics provide fast and easy to implement solutions. Such solutions can be used in real-world situations where optimality is not necessary. Furthermore, they can be used for improvement heuristics as initial solutions. Although construction heuristics are commonly used for the TSP, only few of them are adapted to the TD-TSP. This thesis adapts and evaluates five of the most prominent construction algorithms of the TSP to the TD-TSP: Cheapest Insertion, Christofides, First Fit, Nearest Neighbour and Savings. An attempt was made to create a new benchmark derived from taxi rides in New York City. In order to compare the performances of the considered heuristics, numerical experiments are performed on one artificial and two real-life based time-dependent benchmarks. The results of the comparison are validated on TSP instances from the literature. The outcomes show that the modified Savings algorithm outperforms all other approaches significantly.

CONTENTS

LIST OF FIGURES

LIST OF ALGORITHMS

ACRONYMS

CP Constraint Programming

DP Dynamic Programming

FIFO first in first out

ILP Integer Linear Programming

TSP travelling salesman problem

TDTSP time-dependent travelling salesman problem

TDVRP time-dependent vehicle routing problem

VRP vehicle routing problem

1 INTRODUCTION

In today's world, processes are required to be increasingly efficient to save time and money. Especially in the field of logistics, characterised by its high level of competition, possibilities are examined to create an advantage and achieve greater welfare. Improvements regarding efficiency can be reached by using superior methods in tour planning, for example. Operations Research helps with analytical techniques to solve numerical problems and produce better, quicker and more robust results.

The *travelling salesman problem* (TSP) is one of the most discussed problems in Operations Research. The objective is to find the Hamiltonian cycle with the optimal objective value in a given graph. A *Hamiltonian cycle* is a cycle that visits each vertex of the graph exactly once except for the depot twice. Initially, the problem was inspired by salesmen who visited multiple cities to sell their products and get back to their hometown, considering travel time, ticket price, and other objectives.

The TSP has different applications in its purest formulation, such as the traffic and transportation sector. Modern logistic systems that deliver packages to their customers and return to the depot in the most economically efficient manner are an example of this. Various problems in the literature such as creating timetables for universities [4], the control of production machines [33], the path control of laser carving [49], X-ray crystallography analysis [8] and more can be reduced to finding a Hamiltonian cycle similar to the TSP. While the TSP has many applications, in most cases, it is not adapted to the individual circumstances because the model strongly simplifies the given problem.

In the original TSP, the objective function is constant over the entire time horizon. In reality, street traffic is highly dependent on traffic congestion which occurs in predictable patterns determined by the day, week, season or year. Since more data is available, it is possible to consider time dependencies and improve the precision of the predictions. Several papers show that considering the time-dependency can significantly improve the solution of TSPs [13, 15]. The resulting problem is the *time-dependent travelling salesman problem* (TD-TSP) which expands the original model by time-dependency to better fit real-world problems.

The TSP is NP-hard [32]. Since the TD-TSP is an extension of the TSP, it is also NP-hard [36]. Furthermore, it tends to be more challenging than its basic counterpart, the TSP, especially for real-life sized problems [17]. For these reasons, scholars propose different

approaches to solve the TD-TSP. Firstly, exact methods that lead to optimal solutions but are currently not capable of solving problems with more than forty vertices [56, 46]. Hence, they can most often not be applied for real-world examples because they include more vertices.

Secondly, heuristics try to arrive at practicable solutions faster than exact approaches. They are used when the costs of computing a solution are higher than the loss in quality. Heuristics can be roughly divided into the two categories construction heuristics and improvement heuristics [47, 52]. *Improvement heuristics* start with a feasible solution and try to improve it. Their procedure often depends on randomisation. When an easy to obtain solution is needed fast, construction heuristics are used. *Construction heuristics* or *constructive heuristics* are heuristic methods that start with an empty solution and run until a feasible solution is obtained. If no randomness is involved in their procedure, they are called deterministic. This thesis considers only deterministic heuristics because their performances are more accessible to compare due to their predictable nature.

Construction heuristics have different applications. Their results can be utilised directly, as an upper bound for exact methods and their performance can be compared with new algorithms. Another use is in the context of improvement heuristics. Suited construction heuristics that already provide different reasonable solutions can increase the quality of improvement heuristics [23].

For the TSP, it is possible to choose between different construction heuristics because compilations and comparisons exist [48, 41, 21]. While there is significant interest in TD-TSP topics, to the author's best knowledge, no paper has been published that compares different construction heuristics for the TD-TSP. Furthermore, for most construction heuristics from the TSP, no adaptation to the TD-TSP exists.

In this thesis, this gap is filled by evaluating construction heuristics on different benchmarks. As heuristics, adaptations of five of the most prominent algorithms are provided: Cheapest Insertion, Christofides, First Fit, Nearest Neighbour, and Savings. All algorithms work for the TSP and are modified to operate on TD-TSP instances. These heuristics are compared on four different benchmarks from Melgarejo et al. [37], Rifki et al. [46], Cordeau et al. [11] and TSPLIBs [7, 57] to evaluate their performance in different environments. These benchmarks are created from real-world data, simulation or artificially. Considered are time-dependent benchmarks and time-independent ones to observe differences in the algorithmic performances.

Each heuristic solution is compared to the best heuristic solution, upper bound or optimal solution of the instance. The only criteria taken into consideration for the comparison is the objective value.

Despite the great scientific interest, only a limited number of benchmarks exist that share characteristics with real road traffic. Therefore, an attempt was made to create a new benchmark derived from real-world traffic with the ”New York Yellow Taxis” data. The data was cleaned, enhanced and a travel time matrix was computed.

This thesis is organised as follows. Section 2 starts by providing a review of existing literature in the field of TD-TSP and construction heuristics. Subsequently, in Section 3, a definition and a mathematical formulation of the TD-TSP is given. Section 4 describes the algorithms used in the later experiments and how they are implemented, namely, Cheapest Insertion, Christofides, First Fit, Nearest Neighbour and Savings. In Section 5, the TSP and TD-TSP literature benchmarks are introduced on which the algorithms are tested. Section 6 presents the attempt to create a benchmark with the data of the New York Yellow Taxis. The results of these experiments are presented in Section 7, where the findings are also discussed.

2 LITERATURE REVIEW

For several years, there has been a trend to extend the TSP to better approximate reality through more complex travel time functions. An example of this is taking into account time dependencies. The first time-dependent model for travel times was introduced by Beasley [5]. In his paper, the Savings algorithm is adapted to a simplified version of the time-dependent vehicle routing problem (TDVRP). Malandraki et al. [36] were the first to take an interest in the TD-TSP. They explain the main framework conditions for the TD-TSP and propose heuristics for the TDVRP and TD-TSP, especially the implementation of the Nearest Neighbour heuristic. They show that the TD-TSP is an NP-hard problem because it is an extension of the TSP. Gendreau et al. [17] provide a good overview of existing travel time modelling, applications and solution methods for time-dependent routing problems in general until 2015. The authors show that time-dependent TSPs are harder to solve than their basic counterparts.

Different methods exist to find the optimal solution for a TD-TSP. Arigliano et al. [2] used an Integer Linear Programming (ILP) approach to find exact solutions for the TD-TSP with time windows. Their branch-and-bound algorithm solves instances with up to forty vertices but had minutes of computation time for large instances and found no optimal solution in most cases. More ILP approaches have been proposed from Sun et al. [51] and Hansknecht et al. [24] that have the same limitations in time and optimality for instances with a bigger number of vertices [46]. Constraint programming (CP) and dynamic programming (DP) are also used in the literature to solve the TD-TSP to optimality. CP, DP and ILP can solve instances with up to thirty instances, but no more [46]. Vu et al. [56] used dynamic discretisation discovery to find optimal solutions for the TSP with time windows with up to forty vertices.

As most of the exact approaches cited above cannot solve real-life sized instances, scholars turn to approximate approaches such as construction heuristics and improvement heuristics. Improvement heuristics cannot be executed independently because they need a feasible solution to start their procedure.

When solutions are needed fast, and the quality of the objective value comes at a second degree, construction heuristics are best suited because they are less complex than exact methods. Therefore, they can provide solutions for TD-TSP problems in less time and for instances with a large number of vertices. This could be the case for logistic companies that have a high number of customers.

Another application for construction heuristics is that they are needed to start improvement heuristics and are often a part of metaheuristics. In general, *metaheuristics* define an abstract sequence of steps that can be applied to any problem, in contrast to problem-specific heuristics. However, the individual steps must again be adapted to fit the TD-TSP. Often they include, combine or modify existing construction and improvement heuristics. Tamaki et al. [53] state that for the TSP, their genetic algorithm performs depending on its initial solution that is created by construction heuristics. Hansen et al. [23] show that their 2-opt improvement heuristic achieved better and faster results with construction heuristics than with random initial solutions. Although better-suited starting tours can improve the solution's quality, there is often no clear reflection about the use of construction heuristics.

Another application of construction heuristics is for comparing the performance of newly proposed methods against them. Ghiani et al. [19] proposed a machine learning approach to solve TD-TSPs. They compare their results with the Nearest Neighbour algorithm.

In total, construction heuristics have lots of applications, and for the TSP, it is easy to choose between different construction heuristics because compilations exist. Nilsson [41] gave an overview of different solving approaches for the TSP. Nearest Neighbour, Cheapest Insertion and Christofides are described briefly for the TSP. Glover et al. [21] put together different construction heuristics for the asymmetric TSP, including Random Insertion and a Recursive Path Contraction algorithm.

When different algorithms are available, it is interesting to measure their performance to always use the best or most appropriate algorithm for given circumstances. Different approaches exist to assess the performances of such algorithms. The theoretical performance of combinatorial optimisation is typically studied by approximation analysis [3]. By doing this, in the end, lower or upper bounds of the worst-case performance can be compared. Rosenkrantz et al. [48] put together different construction and improvement heuristics for the TSP and analysed them. These included the Nearest Neighbour and the Cheapest Insertion algorithm. They proved that Cheapest Insertion provides a solution that is at most twice the optimal solution for a TSP.

Another way to compare algorithms is domination analysis, introduced by Glover et al. [20], that analysis the domination number and domination ratio. The domination number indicates the maximum number of solutions that are not better than all possible solutions. The

domination number is the ratio between the domination number and the number of possible solutions. Gutin et al. [22] apply this approach and state for the Nearest Neighbour algorithm for the asymmetric TSP a domination number of 1. This means that the algorithm does not guarantee any solution quality but can find the worst solution of a given TSP.

Sometimes it is challenging to conduct an approximation or a domination analysis analytically. In these cases, scholars turn to numerical experiments to assess the algorithms [6]. In these experiments, one or multiple algorithms are compared on the same instances to pull exclusion over their quality. Optimal solutions should be preferred for comparisons to evaluate the quality. If they are not available, the best-known solutions should be evaluated. Ong et al. [42] applied that principle and showed that Savings has in their experiments, on average, a better performance than Nearest Neighbour and Cheapest Insertion. A comparison of different construction heuristics for the asymmetric TSP has been made by Glover et al. [21]. They suggest that combined algorithms can be helpful to get robust results for TSP instances. Johnson et al. [29] compare the Nearest Neighbour, a special "Greedy" heuristic, the Savings and Christofides algorithm. They measure the performance on instances from TSPLIB, a TSP library, and randomly generated instances against the Held-Karp lower bound. In their paper, a modified Christofides algorithm performs best after Savings, their greedy algorithm and Nearest Neighbour.

To the best of the author's knowledge, no theoretical proves or computational experiments for the performance of construction heuristics exist for the TD-TSP. Because of the time-dependent aspect, proofs for TSP heuristics do not hold for the TD-TSP. Nevertheless, they are also an indication of the performance of the algorithms for the TD-TSP. For the present work, the algorithms are evaluated with computational experiments.

To rate the performance of algorithms, exemplary data sets are needed that are called *instances*. Instances consist at least of a travel time matrix that describes the travel duration and a set of vertices. Optional, they can include further information to create a model that is closer to real-world traffic. Current research implements the travel time as piece-wise linear functions. A *time step* is one specific segment of the whole time horizon that shares the same travel time. *Benchmarks* combine many instances often with solutions and different characteristics into one collection. The computation time of a TD-TSP algorithm is highly dependent on the number of time steps and the number of vertices.

The first to propose a time-dependent benchmark was Ichoua et al. [28]. They artificially created a benchmark from a Euclidean square with three time steps. Because the benchmark is generated by artificial data, it can not be assumed that it reflects all aspects of real-world traffic.

Fleischmann et al. [15] proposed the first benchmark created from real road traffic. The data is derived from the information system LISB of the city of Berlin. It includes 216 time steps that are not evenly distributed over a whole day. Because it is made from actual traffic data, it leads to more reliable tests for transportation models.

Cordeau et al. [11] established a time-dependent benchmark with three time steps. The vertices were generated randomly in a Euclidean square with the depot locating in the centre. Distinct circulation zones are inspired by real-world traffic by including different congestion patterns in different zones. Arigliano et al. [1] uses an extended version of the proposed benchmark so that it can be applied to the TD-TSP with time windows which was also used in Montero et al. [40].

Melgarejo et al. [37] introduced a new benchmark generated from traffic data in Lyon that contained in the beginning 65 time steps and was later extended by mirroring the data to 130.

Rifki et al. [46] established another benchmark from the Lyon transportation network that was created using a dynamic microscopic simulator called SYMUVIA. They simulated with the software a travel time matrix that respects the traffic flow theory and gives access to the needed level of granularity. Their paper studied the effect of spatio-temporal granularity on the quality of TSP and TD-TSP solutions. They state that not only time-dependency can improve a solution like Fleischmann et al. [15] did but also that spatio-temporal granularity is essential. If complete spatial information is given, Rifki et al. [46] show that smaller time steps lead to better solutions. The benchmark is publicly available.

Although construction heuristics are commonly used, neither is there a collected description of how construction heuristics for the TSP can be implemented for the TD-TSP, nor a ranking of their performances. This thesis fills this gap by adapting five commonly used constructive heuristics of the TSP to the TD-TSP and comparing their performances on a panel of TD-TSP literature benchmarks.

3 PROBLEM DEFINITION AND MATHEMATICAL FORMULATION

This section gives an overview of the TD-TSP, including an Integer Linear Programming (ILP) formulation and a definition of the first in first out property (FIFO).

Let $G = (V,A)$ be a connected graph, with V as the set of vertices and A as the set of all arcs between them. TD-TSP's objective is to find the Hamiltonian cycle that starts with $depot \in V$ and provides the best objective value. In this thesis, the only objective considered is the travel time.

A Hamiltonian cycle can be represented a as path. A path $P = \{v_1, v_2, \ldots, v_k, v_{k+1}, \ldots, v_n\}$, $v_1, v_2, v_k, v_{k+1}, v_n \in V$ defines a sequence of vertices with at least two vertices included. $|P|$ represents the number of vertices in P. A time-dependent path $[P,T]$ is a connection of P with a set of departure times $T = \{t_1, t_2, \ldots, t_k, t_{k+1}, \ldots, t_n - 1\}$ that comprises all starting times of P. This means that v_1 leaves to v_2 at t_1.

In order to compute a solution, a travel time function $f(i, j, t)$ is used that returns a positive value for every given arc $(i, j) \in A$ when leaving vertex i to j at time $t \in T$. In reality, traffic flows dynamically and could be best approximated by continuous functions. Scientific research does not use these so far because the higher computation time and complexity outweigh the continuous functions' value. This thesis uses, like current research, step-wise constant functions as input to calculate the travel time.

The travel time functions are decomposed into different time steps, which represent the smallest unit of time for the whole time horizon. The duration of the time horizon and a time step both depend on the benchmark. M is the set of time steps with $|M|$ as the number of time steps and d as the duration of each time step. In this thesis, for each TD-TSP, all time steps are equal in size.

A disadvantage of step-wise constant functions is that the results could lead to a logical mistake. With these functions, a vehicle may have an incentive to leave later because it could arrive earlier by waiting. If this case can occur, the so-called first in first out (FIFO) or non-passing property would not be satisfied.

The FIFO property states:

$$t_0 < t_1 \rightarrow t_0 + f(i, j, t_0) \leq t_1 + f(i, j, t_1), \forall i, j \in V, t_0, t_1 \in T$$

If a vehicle leaves vertex i to j at t_0 and a second vehicle drives the same way at t_1, which is later than t_0, then the FIFO property ensures that the second vehicle does not arrive earlier

than the first one.

Figure 1 shows a function that represents the travelling time from i to j, $i, j \in V$ depending on the leaving time from i. The length of a time step equals four $(d = 4)$ and the time step starts at zero minutes and ends at twelve minutes $(T \in [0, 12])$. A vehicle that leaves at $t_{l1} = 7$ arrives at $t_{a1} = 13$. Another vehicle that leave at $t_{l2} = 9$ arrives at $t_{a2} = 12$. Hence, the possibility exists that the vehicle that leaves later may arrive earlier. Section 5 explains how the transformation of a non-FIFO function into one that satisfies the FIFO property is done for each benchmark used in the later experiments.

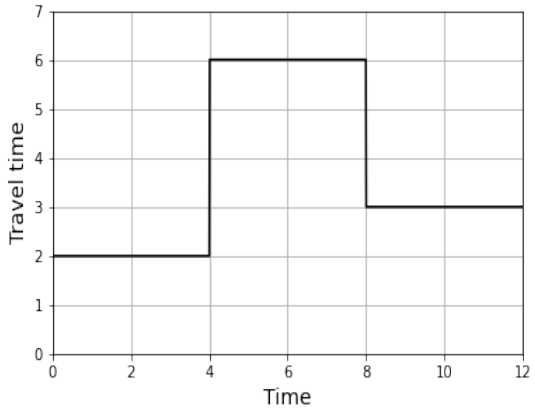

Figure 1. Example for a step-wise constant travel time function

Following an Integer Linear Programming (ILP) formulation of the TD-TSP is given. In line (1) the objective function is defined as the travel time function $f(i, j, t)$ times x_{ij}^t. The auxiliary variable x_{ij}^t is one if the arc (i, j) is activated and zero if not. An *activated arc* is an arc that is included in the Hamilton cycle of the solution. Afterwards, constraints are defined. Equality (2) enforces that every vertex has exactly one outgoing arc. Analogously, in line (3) is defined that every vertex has exactly one incoming arc. Constraint (4) states that for every possible subtour of A called A^*, not all arcs of A^* are activated. This is in line with the Miller-Tucker-Zemlin subtour elimination constraint [38]. Equation (5) enforces that one arc can not be activated on different time steps. This is realised by equating the sum of variable x_{ij}^t and the variable x_{ij}. Line (6) ensures that the tour starts with the depot at time zero. In line (7) is ensured that only feasible paths can be used. A path is infeasible if one departure time in S exists that can not be realised due to the travelling time [39]. An example of this is when

Integer Linear Programming formulation of the TD-TSP

$$\min \quad \sum_{(i,j)\in A}\sum_{t\in T} f(i,j,t)\cdot x_{ij}^{t} \tag{1}$$

$$\text{s.t.} \quad \sum_{j\in V\neq i} x_{ij} = 1 \qquad \forall i \in V \tag{2}$$

$$\sum_{i\in V\neq j} x_{ij} = 1 \qquad \forall j \in V \tag{3}$$

$$\sum_{(i,j)\in A^*} x_{ij} \leq |A^*|-1 \qquad \forall A^* \subseteq A \tag{4}$$

$$\sum_{t\in T} x_{ij}^{t} = x_{ij} \qquad \forall (i,j) \in A \tag{5}$$

$$\sum_{j\in V\setminus\{depot\}} x_{0j}^{0} = 1 \tag{6}$$

$$\sum_{v_k\in P\setminus\{v_n\}}\sum_{t\in T} x_{v_k,v_{k+1}}^{t} \leq |P_{t_o}|-2 \qquad \forall P_{t_o} \text{ infeasible}, t_0 \in T \tag{7}$$

$$x_{ij}, x_{ij}^{t} \in \mathbb{B} \qquad \forall (i,j) \in A, \forall t \in T \tag{8}$$

the start time of a vertex is later than the start time of the following vertex. Finally, in (8), the variables x_{ij} and x_{ij}^{t} are defined as binary.

4 ALGORITHMS

The following section discusses First Fit, Nearest Neighbour, Cheapest Insertion, Savings and Christofides. The algorithms are ordered by the complexity of their implementation. All algorithms take as input a set of vertices V with $|V|$ as the number of vertices included. Also a $depot \in V$ and a (time-dependent) travel time function $f(i,j,t), i,j \in V, t \in T$ between all the given vertices over time horizon T is required. As the result, the algorithms return the solution (sol) as a path and the total travel time of the given solution (t) for the computed path. For a better understanding of the heuristics, algorithmic descriptions are provided.

Also, the results of the algorithms are illustrated through an illustrative example with three time steps. Figure 2 shows the travel time for each time step between the vertices of $V = \{0,1,2,3\}$. The circles represent the vertices with the numbers inside as their index numbers and index zero as the depot. Each arc has an assigned travel time value in minutes. The graph includes three time steps with a time step duration of seven minutes. The time steps can be distinguished by the color of the arc. A black arc represents a ride that starts in time step one, a blue arc in time step two and a green arc in time step three. When a vehicle leaves between time zero and six minutes, it is in the first time step. When it leaves between seven and thirteen minutes, it is in the second, and for all times greater or equal to fourteen minutes, it is in the third time step. In the later experiments, the travel time is not restricted to natural numbers but can also be positive rational numbers.

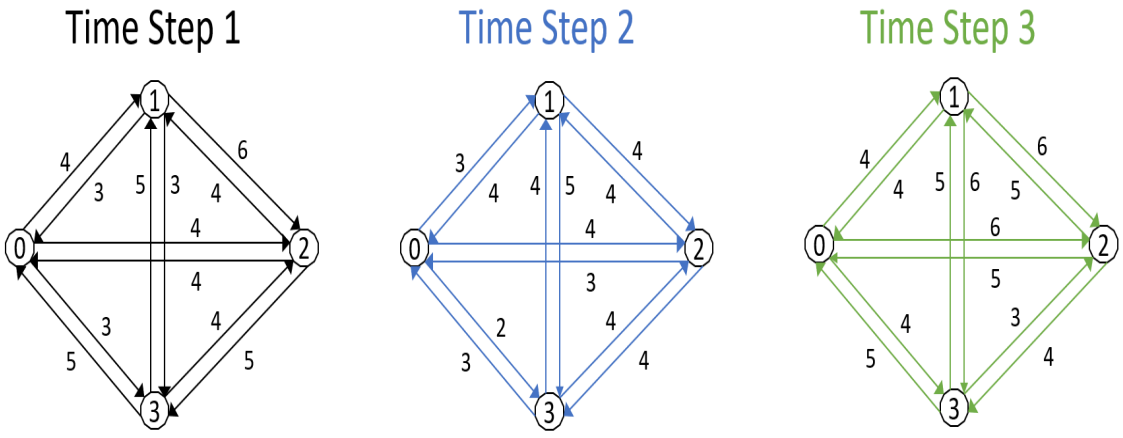

Figure 2. Example graph

4.1 First Fit

First Fit is an intuitive construction heuristic for obtaining a feasible solution for a TD-TSP. Usually, vertices are ordered previously, e.g. by the travel time matrix, and it is typical to take the given structure. A solution is found by up counting and connecting the vertices by their index. Additionally, the last vertex has to be connected with the depot. There are no differences between TSPs and TD-TSPs for this algorithm.

In the following, Algorithm 1 is explained step by step using the example graph of Figure 2. First Fit starts in Algorithm 1 by setting the start as the depot that is vertex zero (line 1). Due to this change current solution is now $sol = \{0\}$. The variables t and i are set equal to zero (line 2 and 3) to store the solution time and the current index. The while loop in line 4 iterates over all vertices except for the depot. Because $|V| = 4$, it iterates three times. In every step, it adds up the time (line 5), stores the current index in the solution (line 6) and increases variable i by one (line 7). Applied for the first iteration, this means that $t = 4$, $sol = \{0,1\}$ and $i = 1$. The second iteration sets $t = 10$, $sol = \{0,1,2\}$ and $i = 2$. The result of the first and second iteration are illustrated in Figure 3. The third iteration works accordingly in the same manner. After the while loop, the time needed for getting back to the depot is added (line 9), and the index is stored in the end of the solution (line 10). As the last step, the solution $sol = \{0,1,2,3,0\}$ and the solution time $t = 19$ are returned (line 11). The final solution is presented in Figure 3. The Hamiltonian cycle starts at vertex zero goes through vertex one, vertex two and vertex three back to vertex zero. The first two arcs are black because both start in the first time step, the arc between vertex two and three is blue because it starts in the second time step ($t = 10$), and the arc between vertex three and vertex zero is green because it starts in the third timestep ($t = 14$).

The complexity of First Fit is $\Theta(|V|)$ because it iterates once over all vertices. The characteristic of the solution is dependent on the given travel time function. If the vertices are ordered in a particular way, e.g. by the position, it can positively or negatively influence the result. With randomly chosen vertices, First Fit's solution can be considered a random solution. First Fit does not guarantee any performance ratio and may return the worst solution for the TSP as for the TD-TSP.

Algorithm 1 First Fit

Require: $G(V,A), f(i,j,t), depot$

1: $sol_0 \leftarrow depot$
2: $t \leftarrow 0$
3: $i \leftarrow 0$
4: **while** $i < |V| - 1$ **do**
5: $\quad t \leftarrow t + f(i, i+1, t)$
6: $\quad sol_i \leftarrow i$
7: $\quad i \leftarrow i + 1$
8: **end while**
9: $t \leftarrow t + f(i, depot, t)$
10: $sol_{|V|} \leftarrow depot$
11: **return** sol, t

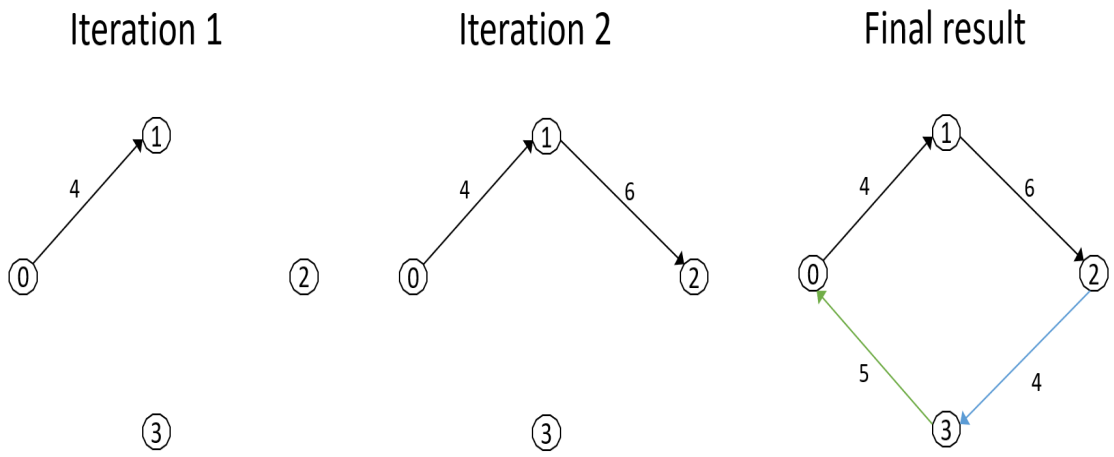

Figure 3. First Fit example

4.2 Nearest Neighbour

Nearest Neighbour is a greedy algorithm. Karg et al. [31] first introduced the application for the TSP. Its idea is to start from the depot as the current vertex and repeatedly take the closest unvisited vertex into the solution and as the new current vertex. It does not change significantly by transforming it from a TSP to a TD-TSP. The main change is that in the calculations of the travel time, time dependencies are now included.

Algorithm 2 describes how the heuristic can be applied. In Figure 4 the first two iteration steps and the final result for the graph of Figure 2 are given. In the following, the algorithm is presented using the example graph.

In Algorithm 2, the depot is the first vertex of the solution, so the solution is $sol = \{0\}$ (line 1). t and i are initialised as 0 and 1 (line 2,3). For executing the algorithm, all vertices except for the depot must be added to a list of unvisited vertices, which results in $unvisitedV = \{1,2,3\}$ (line 4). While the list of unvisited vertices is not empty (line 5), the following is executed. At first, a compare value is initialised as an unreachable number, in this case, represented as infinity (line 6).

Then for every unvisited vertex v (line 7), it is checked if the connection of v to the current vertex is shorter than the current compare value (line 8). In the first iteration, $f(0,1,0) = 4$ and is smaller than infinity. Therefore, vertex one is set as the solution at position i that is 1, so $sol = \{0,1\}$ (line 9) and the compare value is set as four (line 10). In the second repetition of the for loop from line 7, the second vertex is considered. Because $f(0,2,0) = 4$ is not smaller than the current compare value, which is also four, the next vertex is taken into consideration. Because $f(0,3,0) = 3$ is smaller than four, it is set as the new compare value and the solution is overwritten at position 1, it follows $sol = \{0,3\}$. After iterating over all unvisited vertices, the found nearest vertex is deleted from the list of unvisited vertices, $unvisitedV = \{1,2\}$ (line 13). Now the time $t = 4$ and the iteration step $i = 2$ is updated (line 14,15). The state at the end of the first iteration is represented in Figure 4. The dashed arcs represent the other possibilities that not have been taken. The second and third iteration work according to the same system. When all vertices are visited, in lines 17 and 18, the path is completed by inserting an arc that goes back to the depot, and the time and the solution are updated. In the end, the solution $sol = \{0,3,2,1\}$ and the solution time $t = 15$ are returned (line 19).

Nearest Neighbour checks in every iteration all vertices for the nearest unvisited neighbour of the current vertex. Hence, the heuristic can be implemented in $\Theta(|V|^2)$ for the TSP as for the TD-TSP. In both cases, there is no theoretical upper bound for their performances [22].

4.3 Cheapest Insertion

Compared to the Nearest Neighbour algorithm, *Cheapest Insertion* does not add the closest unvisited vertex repetitively to the end of the queue but does insert it at any feasible position that adds the least time to the objective value of the current path. In this thesis, the position

Algorithm 2 Nearest Neighbour

Require: $G(V,A), f(i,j,t), depot$

1: $sol_0 \leftarrow depot$
2: $t \leftarrow 0$
3: $i \leftarrow 1$
4: $unvisitedV \leftarrow V \setminus \{depot\}$
5: **while** $unvisitedV \; != \emptyset$ **do**
6: $\quad compare \leftarrow \infty$
7: $\quad$ **for** `v in unvisitedV` **do**
8: $\quad\quad$ **if** $f(sol_{i-1}, v, t) < compare$ **then**
9: $\quad\quad\quad sol_i \leftarrow v$
10: $\quad\quad\quad compare \leftarrow f(sol_{i-1}, v, t)$
11: $\quad\quad$ **end if**
12: $\quad$ **end for**
13: $\quad unvisitedV \leftarrow unvisitedV \setminus \{sol_i\}$
14: $\quad t \leftarrow t + f(sol_{i-1}, sol_i, t)$
15: $\quad i \leftarrow i + 1$
16: **end while**
17: $t \leftarrow t + f(i, depot, t)$
18: $sol_{|V|} \leftarrow depot$
19: **return** sol, t

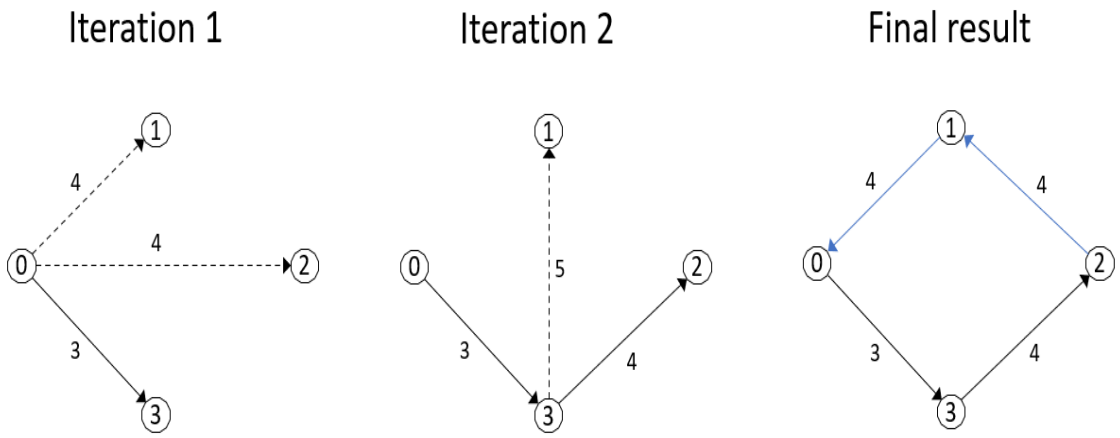

Figure 4. Nearest Neighbour example

of the depot is not flexible but set in the beginning. Hence, insertions can be done anywhere except for the first and last positions that are reserved for the depot.

Algorithm 3 gives an algorithmic explanation of the heuristic. In subsequent, an example of the algorithm is executed on the graph from Figure 2. The results of this procedure after the first and second iteration and the final result are shown in Figure 5.

It starts with the depot at the beginning and the end of the provisional solution, $sol = \{0,0\}$ (line 1,2). Afterwards, all vertices except for the depot are marked as unvisited, $unvisitedV = \{1,2,3\}$ (line 3). The variable i is used to store the current cycle size which is at first set as one (line 4). Subsequently, a while loop is executed until all vertices from the list are visited (line 5). At first, in the loop, a compare value is initialised as a sufficiently large number represented, in this case as infinity (line 6). After this, the test path is set as the provisional solution, $testPath = \{1,2,3\}$ (line 7). The test path is used to evaluate later which insertion is the cheapest available. Then for every unvisited vertex $v \in unvisitedV$ (line 8), in every possible position p (line 9), an insertion is executed (line 10), and the total duration for each test path is calculated. Because $1 \leq p \leq 1$, position one, the position between both depots $sol = \{0,0\}$, is the only possible insertion position. The function ".time" sums the arc weights of a time-dependent path. It always refers to the object that is located in front of the point. When the calculated time is shorter than the compare value (line 11), the current vertex v is stored as the best vertex (line 12). The same is done for position p, stored as the best position so far (line 13). The first unvisited vertex is vertex one, and the only possible insertion

position is position one. Hence, the first possible test path is $testPath = \{0,1,0\}$. Because the total duration of the variable $testPath$ is seven, that is smaller than infinity, $bestV = 1$ and $bestP = 1$. The following repetitions of the for loops from lines 8 and 9 work according to the same principle.

After these loops, the best available insertion is found, which is $testPath = \{0,1,0\}$. Now, the solution is updated by pasting the vertex into the current solution $sol = \{0,1,0\}$ (line 17), and the list of unvisited vertices is updated $unvisitedV = \{2,3\}$ (line 18). Variable i increased by one to always check every feasible insertion position (line 19). In iteration two, there are two possible insertion positions. The results of the first iteration of Algorithm 3 are shown in Figure 5. Now the second iteration works analogously. When the list of unvisited vertices is empty, the time of the solution is calculated, which is fourteen (line 21) and returned with the Hamiltonian cycle $sol = \{0,1,3,2\}$ (line 22).

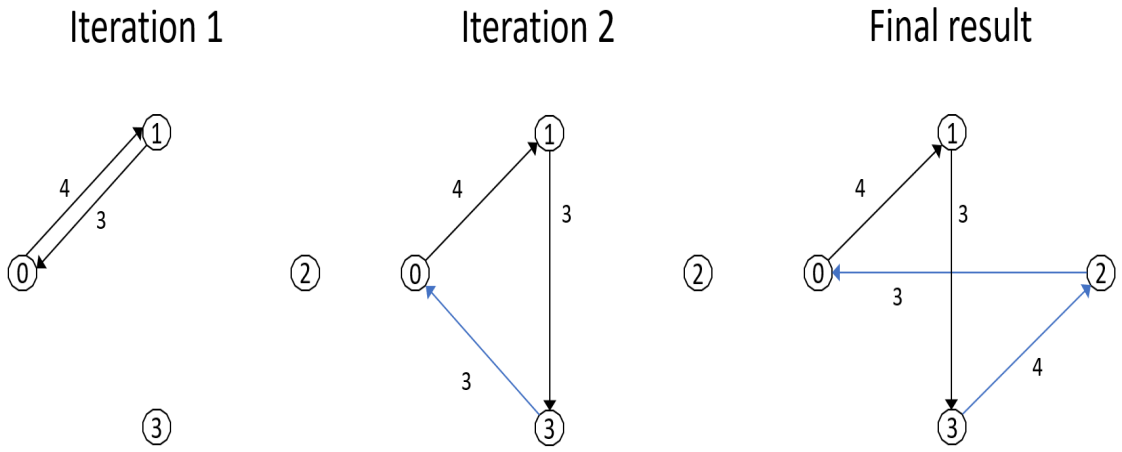

Figure 5. Cheapest Insertion example

Cheapest insertion does not change in terms of logic regardless of whether it is applied on a TSP or a TD-TSP. The algorithm inserts new vertices for both solutions by checking all unvisited vertices for all possible positions for the lowest increase in the current path. A difference is that subsequent connections can change for the TD-TSP in travel time due to a change in the time step. This does not affect the TSP.

Rosenkrantz et al. [48] prove that for an undirected TSP that satisfies the triangle equality Cheapest Insertion returns a solution that is at most twice the optimal solution. In geometry, the triangle equation is a theorem that states that a side of a triangle is at most as long as

Algorithm 3 Cheapest insertion

Require: $G(V,A), f(i,j,t), depot$

1: $sol_0 \leftarrow depot$
2: $sol_1 \leftarrow depot$
3: $unvisitedV \leftarrow V \setminus \{depot\}$
4: $i \leftarrow 1$
5: **while** $unvisitedV \; != \emptyset$ **do**
6: $\quad compare \leftarrow \infty$
7: $\quad testPath \leftarrow sol$
8: $\quad$ **for** `v in unvisitedV` **do**
9: $\quad\quad$ **for** `p=1;p` $\leq$ `i;p++` **do**
10: $\quad\quad\quad testPath \leftarrow$ *insert v at p*
11: $\quad\quad\quad$ **if** $testPath.time < compare$ **then**
12: $\quad\quad\quad\quad bestV \leftarrow v$
13: $\quad\quad\quad\quad bestP \leftarrow p$
14: $\quad\quad\quad$ **end if**
15: $\quad\quad$ **end for**
16: $\quad$ **end for**
17: $\quad sol \leftarrow sol + (insert\ bestV\ at\ bestP)$
18: $\quad unvisitedV \leftarrow unvisitedV \setminus \{bestV\}$
19: $\quad i \leftarrow i + 1$
20: **end while**
21: $t \leftarrow sol.time$
22: **return** sol, t

the sum of the other two sides. Applying the example to road traffic, the triangle equation holds for a travel time function if for any three vertices $i, j, k \in V$, there is no quicker path from i to j than the direct path. Every additional vertex k would be in total at least equal to the direct path. Hence, for a given TSP holds $f(i, j) \leq f(i,k) + f(k, j)$. Some benchmarks like the benchmark from Melgarejo et al. [37] is computed with a point-to-point shortest path algorithm. Therefore, the triangle equation is still ensured. In the later experiments, the travel time matrices from the time-dependent benchmarks are transformed to time-independent ones. Due to this procedure, there is no guarantee that the triangle equation is still satisfied. Hence, there is no guarantee for the performance of Cheapest Insertion when used for a TD-TSP in the later experiments.

The Cheapest Insertion algorithm inserts $|V| - 1$ vertices. Each time it iterates at most $|V| - 1$ times over all unvisited vertices and at most $|V| - 1$ over all possible insertion positions. Hence, the algorithm can be implemented in the complexity $\Theta(|V|^3)$.

4.4 Savings

Clarke et al. [10] initially introduced the *Savings* algorithm that was mainly used to find an optimum routing for a fleet of vehicles with restricted capacities. It follows the idea to connect all customers with the depot and insert the best shortcut called *saving* repetitively afterwards until a stopping condition is met. This circumstance would be the case for the vehicle routing problem, e.g. because of capacity restrictions. TD-TSPs used in this thesis do not have restrictions. Therefore, they run until exactly two vertices are connected with the depot, representing a feasible solution for the TD-TSP. The Savings algorithm newly introduced for the TD-TSP by this thesis differs, besides the number of vehicles that is one, from the existing version from Clarke et al. [10].

The Savings algorithm by Clarke et al. [10] for one vehicle is briefly described below. In their algorithm, every vertex is connected via an outward and return arc with the depot at the beginning. Commuting tours are created. All savings generated are evaluated according to $s_{i,j} = f(i, depot) + f(depot, j) - f(i, j)$. The calculated savings are sorted in decreasing order. Then, the two vertices that have the best remaining saving and at least one arc to the depot get connected. This step is repeated as long as there are more than two arcs connected with the depot. If this is the case, a feasible solution is reached.

Beasley [5] adapted the Savings algorithm to consider time-dependencies for the vehicle

routing problem with capacity restrictions. His benchmark consists of two time steps. For Beasley's Savings algorithm, each saving is

$$s_{ij} = max\{f(depot,i,t_0) + f(j,depot,t_1) - f(i,j,t_1), \\ f(depot,j,t_0) + f(i,depot,t_1) - f(j,i,t_1)\}$$

with t_0 as the start of the first time step and t_1 as the start of the second time step. Afterwards, the savings are applied sequentially and in descending order if the implementation is feasible.

For the final travel time calculation, the speed of a vehicle changes directly to the speed of the new time step when the current time step changes. This condition is ensured by checking if an arc passes by two or more different time steps and calculating the total travel time of one arc by subdividing the arc into parts with different speeds. In 1981 when the paper was published, the concept of FIFO or the non-passing property was not introduced. The proposed approach from Beasley [5] works with a similar concept as the FIFO property later introduced by Ichoua et al. [28]. In this thesis, the FIFO property is always satisfied as stated from the benchmark.

The modified Savings by Beasley [5] shows some disadvantages regarding the inclusion of time dependency. Firstly, a sequential approach of Savings is taken that includes multiple vehicles. Because in this thesis, the TD-TSP with only one vehicle without capacity restrictions is considered, a sequential approach is not reasonable due to the fact that it then strongly resembles the Cheapest Insertion algorithm. Instead, a parallel approach is taken. Secondly, a single calculation of the savings has the drawback of not reacting to changes in the travelling time accordingly. This procedure works because the paper only operates on an example with two time steps, but for benchmarks with more time steps, the paper assumed a too simplified version of time dependency.

For the own adaptation of Savings to the TD-TSP, the algorithm has to be modified to consider time-dependency. At the beginning of the algorithm, all vertices are connected with the depot with an incoming and an outgoing arc. All paths that start and end with the depot but do not include all vertices are called *inner cycle*. Time dependency is newly introduced for every inner cycle of the provisional solution. All inner cycles start with the time $t = t_0$. With this measure, it is possible to estimate a more realistic time step when insertions are done.

A difference between the original Savings algorithm and the one proposed in this thesis is that savings cannot only be realised at the beginning or at the end of an inner cycle but also at

the positions in between that are not connected with the depot. Therefore, more possibilities for savings are created, which results in better solutions found. However, this requires a longer computation time as more possible insertions have to be evaluated.

A description of the modified Savings is given in Algorithm 4. In the following, it is applied to the example graph from Figure 2. It starts by connecting all vertices with incoming and outgoing arcs to the depot. This step is done by setting the variables i and counter to zero and V^* as all vertices except for the depot (line 1,2,3). Afterwards, in a while loop, each vertex is connected with the depot (line 4). This is accomplished by setting the solution as the depot at position i and following at $i+1$ as the position counter of the set V^* (line 5,6). The variable i is increased by 2 to continue filling the initial infeasible solutions with vertices (line 7). The variable counter is increased by one to keep track of the number of inner cycles and to get out of the while loop when all vertices are added (line 8). At line 10, the depot is added at the end of the provisional solution to close the inner cycle. Now, every vertex is connected with arcs to the depot with $sol = \{0,1,0,2,0,3,0\}$.

After creating the commuting tours, in every iteration of the following while loop (line 11), the number of inner cycles and the length of the current solution is reduced by one until a feasible solution is found. At first, in the while loop, a compare value is initialised as infinity (line 12). This value helps to take at the beginning every solution, but after having identified a solution taking no worse than the best found so far. The list *bestPath* is initialised as an empty list (line 13) and gets filled after the first feasible shortcut is identified and is replaced when a better solution is found.

In line 14, a for loop iterates over every inner cycle c in the current solution. The inner cycles are (0,1,0), (0,2,0) and (0,3,0). Another for loop iterates over all valid positions p in the current solution (line 15). A position is valid if it is not at the first and last spot because then it would displace the depot in its function. The feasible positions are 1, 2, 3, 4, 5 and 6, with 0 as the first position. In line 16, the variable $testPath$ is set as the current solution without the inner cycle c except for the first depot of the inner cycle. For the first inner cycle $c = (0,1,0)$, $testPath = \{0,2,0,3,0\}$. Then, the inner cycle c is inserted at position p of the for loop without the connections to the depot at the beginning and end (line 17). For the first position this would result in $testPath = \{0,1,2,0,3,0\}$. By this implementation, the length of the testPath is reduced by one compared to the current solution.

If the total time of the resulting testPath is smaller than the compare value (line 18), the testPath becomes the new bestPath (line 19), and the compare value is set as the time of the bestPath to deny all solutions that are greater than the current bestPath. The summed travel time of the variable $testPath$ is twenty-one. It is essential to regard that each inner cycle starts with time zero, which is important for the time step and, therefore, the travel time calculation.

After each execution of the two for loops that work analogously, the current solution is set as the variable $bestPath$ that was found by inserting every inner cycle at every valid position (line 24). After the first iteration $sol = \{0,1,3,0,2,0\}$ with a travel time of eighteen minutes which is shown in Figure 6. After one more iteration, the final Hamiltonian cycle $sol = \{0,1,3,2,0\}$ is reached with a total travel time of fourteen minutes represented in the same figure. In general, when the solution length reaches the length of $|V|+1$, a feasible solution for the TD-TSP is found, and the solution and time are returned (line 25, 26).

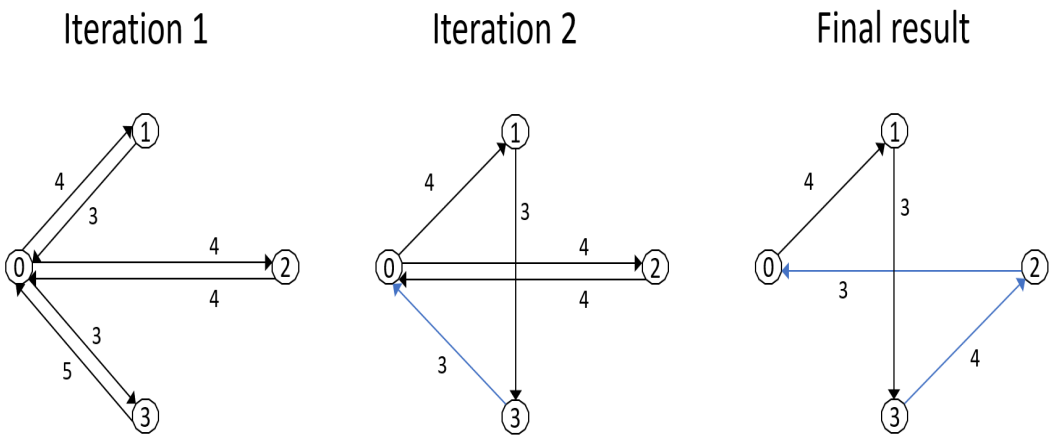

Figure 6. Savings graphical example

Further explanation is provided in Figure 2 which illustrates a numerical example with the graph from Figure 2. Each number represents the index number of the vertex with zero as the depot. Every row shows the current path that is unfeasible until the end because the depot is visited more often than twice. In the beginning, each vertex is inbound and outbound connected to the depot in ascending order of the index number. In Step 1, vertex three is shifted from its original position after vertex one because it has the best savings value. During this process, the former connection from vertex one to the depot is deleted. With every iteration, the number of inner cycles decreases by one because one shortcut is realised.

Algorithm 4 Savings

Require: $G(V,A), f(i,j,t), depot$

1: $i \leftarrow 0$
2: $counter \leftarrow 0$
3: $V^* \leftarrow V \backslash \{depot\}$
4: **while** $counter < |V^*|$ **do**
5: $sol_i \leftarrow depot$
6: $sol_{i+1} \leftarrow V^*_{counter}$
7: $i \leftarrow i+2$
8: $counter \leftarrow counter+1$
9: **end while**
10: $sol_i \leftarrow depot$
11: **while** $|sol| > |V|+1$ **do**
12: $compare \leftarrow \infty$
13: $bestPath \leftarrow \emptyset$
14: **for** $\forall innerCycles$ c in sol **do**
15: **for** all valid positions p in sol **do**
16: $testPath \leftarrow sol \backslash \{c \backslash \{depot_{start}\}\}$
17: $testPath \leftarrow testPath$ with $c \backslash \{depot_{start}, depot_{end}\}$ at position p
18: **if** $testPath.time < compare$ **then**
19: $bestPath \leftarrow testPath$
20: $compare \leftarrow bestPath.time$
21: **end if**
22: **end for**
23: **end for**
24: $sol \leftarrow bestPath$
25: **end while**
26: **return** sol, t

In Step 2, The inner cycle with vertex two is removed and inserted after vertex three. Again the outgoing connection to the depot of the moving inner cycle is deleted. The path $P = \{0,1,3,2,0\}$ is now a Hamiltonian cycle and therefore a feasible solution for the TD-TSP.

During the iterations, the inner cycles without connection to the depot are removed and inserted in a different position afterwards. Hence, the time step of an inner cycle can change completely during one iteration. To preserve the possibility to insert the inner cycle at every valid position but also to take into account the time dependencies, savings are calculated after each iteration to ensure the consideration of changes in the time step. This leads to an increase in computation time but pays more attention to the time dependency.

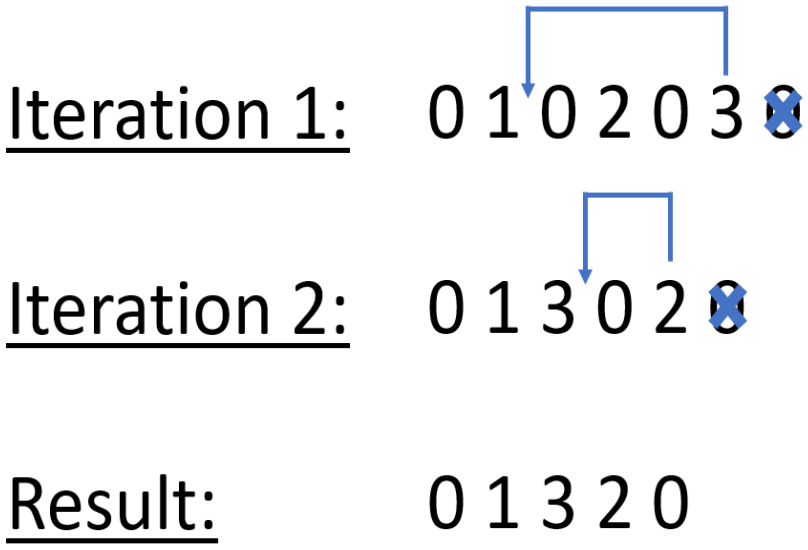

Figure 7. Savings numerical example

To consider the complexity, the first part that generates a provisional solution and the second part that shortens it has to be taken into account. The provisional solution is created by iterating over the list of vertices without the depot. Therefore, it is in $\Theta(|V|)$. The second part that is transforming the provisional solution into a Hamiltonian cycle, iterates in the while loop $|V|-2$ times (line 11). The following two for loops (line 15,16) iterate at most $|V|$ times. Therefore, shortening of the provisional solution can be implemented in $\Theta(|V|^3)$. The second part of the algorithm outweighs the first one. Therefore, the whole procedure requires time $\Theta(|V|^3)$.

4.5 Christofides

The *Christofides* algorithm is named after its inventor Nicos Christofides [9]. In contrast to other approaches, the algorithm is an approximation algorithm for the TSP because it can guarantee a worst-case ratio of 3/2 compared to the exact result. This threshold makes it one of the most interesting construction algorithms for the TSP and makes it worthwhile to test its performance on TD-TSP instances.

For Christofides, the issue exists that some algorithmic steps can not be easily implemented in a time-dependent manner. An example of this is the calculation of a minimum spanning tree. This can be computed for time-independent graphs in complexity $\Theta(|A|log|A|)$ by using Kruskal's algorithm [34]. Huang et al. [27] proposed two concepts for time-dependent minimum spanning trees. The first version is calculated by finding the earliest arrival times for all vertices in the graph. This can be computed in linear time but has the disadvantage that the sum of the weights is not considered. The second version aims to minimise the total weight of the time-dependent minimum spanning tree. This seems more appropriate than the first version, but because the procedure is MAX-SNP hard, it is not practical to use it for construction heuristics that is expected to be fast.

Because the algorithm can not be adapted to the TD-TSP without accepting disadvantages with regard to quality or computation time, a different approach is followed. Instead of modifying the algorithm for the TD-TSP, the graph and the travel time function are transformed to fit an undirected TSP problem. This is done by taking the median value of all trips from the travel time function $f(i,j,t)$ and $f(j,i,t), \forall i,j \in V, t \in T$ which becomes the time-independent travel time function $f(i,j)$. Figure 8 shows how the transformation is done for the example graph from Figure 2. Numbers are rounded up if necessary to avoid decimal numbers. After the computation of the travel time function, the original TSP algorithm is applied, and the travel time is calculated with the achieved solution and the original time-dependent travel time function.

The procedure can be described as in Algorithm 5 by the following steps. At first, a minimum spanning tree T for the underlying graph $G = (V,A)$ is created. Afterwards, a minimum weight perfect matching M in the graph between the vertices with an odd number of connected arcs in the tree T has to be found. By merging T with M, the resulting graph $E = T \cup M$ is then Eulerian. Because the number of arc connections is even for all vertices, an

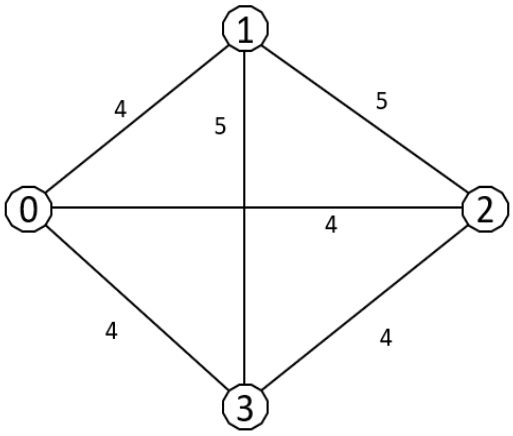

Figure 8. Undirected time-independent example graph

Eulerian tour can be constructed. To construct a Hamiltonian cycle in E, the depot has to be chosen in the beginning and afterwards always select an unvisited connection of the current vertex. If there is no direct connection between the current and an unvisited vertex, a shortcut to any unvisited vertex is taken. In the implementation of this thesis, Christofides always takes the shortcut to the closest, unvisited vertex. The resulting path is a Hamiltonian cycle that is a feasible solution for the TSP.

Algorithm 5 Christofides

Require: $G(V,A), f(i,j), depot$

1: Calculate a minimum spanning tree T
2: Find the set of vertices O with an odd number of connected arcs in T
3: Construct a minimum weight perfect matching M for O
4: Create a Eulerian graph $E \leftarrow T \cup M$
5: Calculate a Hamiltonian cycle (sol) in E by taking shortcuts
6: Calculate t from the original time-dependent travel time function
7: **return** sol, t

The first step of the Christofides algorithm demands a minimum spanning tree. This thesis uses Kruskal's algorithm [34]. Kruskal's algorithm is a greedy algorithm in graph theory introduced by Joseph Kruskal for computing minimum spanning trees of undirected graphs.

A spanning tree is a set of arcs that connects all vertices by precisely one path between

every two vertices. It is called a minimum weight spanning tree or minimum spanning tree if the spanning tree in a weighted graph has the minimum summed weights compared to all possible spanning trees.

Kruskal's algorithm is implemented in Algorithm 6. The input graph must be connected, weighted and finite. For the undirected time-independent example graph in Figure 8, the arcs are first ordered by their travel time and second ordered by the index number. At first, the arcs with a travel time of four are included and afterwards, those with a travel time of five. The sorted list of weights with the fastest arc with the smallest index number is: $sortedA = \{(0,1),(0,2),(0,3),(2,3),(1,2),(1,3)\}$. For sorting the arcs, e.g. Quicksort can be used [26]. Then, the algorithm repeatedly selects among the not yet selected arcs the arc with the shortest travel time that does not form a circle with the already selected arcs. In this example, the first three arcs are the minimum spanning tree, and the arcs afterwards are not considered. The result of the algorithms for the example graph is shown in Figure 9.

Algorithm 6 Kruskal

Require: G(V,A)

1: $sortedA \leftarrow$ Sort A according to its arc weights
2: $T \leftarrow \emptyset$
3: $i \leftarrow 0$
4: **while** $i < |sortedA|$ **do**
5: **if** $T \cup sortedA_i$ does not form cycle **then**
6: $T \leftarrow T + sortedA_i$
7: **end if**
8: $i \leftarrow i + 1$
9: **end while**
10: **return** T

The next step is to find the vertices with an odd number of connected arcs in the minimum spanning tree T. This can be done by counting. O is the resulting set of odd degree vertices in T. For the minimum spanning tree of Figure 9, all four vertices have an odd number of connected arcs.

The Blossom algorithm is an algorithm for finding the maximum weight perfect matching in a graph using the blossom contraction process introduced by Edmonds [14]. The Blossom

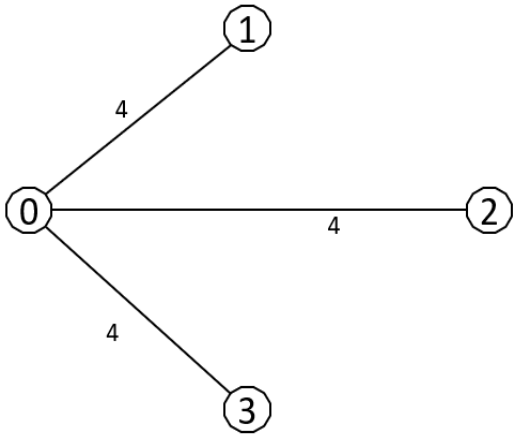

Figure 9. Minimum spanning tree for Christofides example

algorithm is historically significant because it first proved that maximum weight perfect matching can be found in polynomial time and showed that it is in the P-complexity class. By adapting the algorithm or inverting the arc weights of the graph, it can be used to find a minimum weight perfect matching.

Given a weighted graph G, a minimum weight perfect matching finds a subset of arcs so that each vertex $v \in V$ is connected with precisely one arc in the subset. The matching is perfect if G is a complete graph and $|G|$ is even. A perfect matching is the minimum weight perfect matching if it has the smallest summed weights of all possible perfect matchings. The result of the algorithm provides the matching M = $\{(0,1),(2,3)\}$ which is the cheapest possibility to connect the four vertices.

By combining the original graph G with the minimum weight perfect matching M, graph E is created, which are both illustrated in Figure 10. Now Euler's Theorem can be applied, which was first proven by Hierholzer et al. [25]. Euler's Theorem states that a connected graph has an Eulerian cycle if every vertex has an even degree. Because E is connected and all vertices have even degrees due to the preceding matching, there must be an Eulerian cycle in E.

In the next step, a Hamiltonian cycle is found by starting from the depot and taking as next vertex always the next connected vertex that has not been visited so far. If only already visited vertices are connected with the current vertex, a shortcut is done to the closest vertex that has not been visited so far. For the Eulerian cycle in Figure 10 the tour starts from vertex zero,

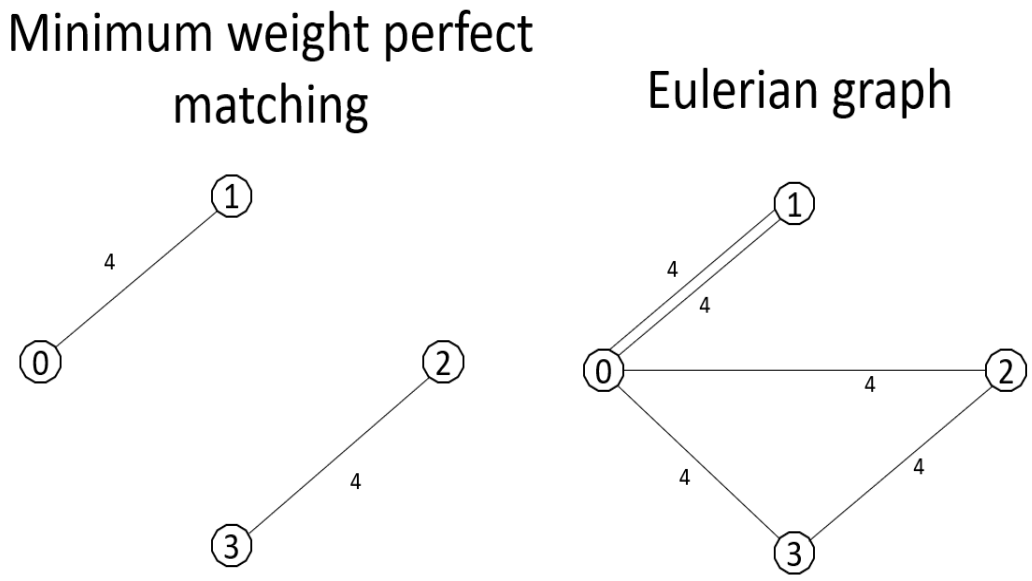

Figure 10. Minimum weight perfect matching and Eulerian graph for Christofides example

which is the depot. Afterwards, the connected vertex with the smallest index number is taken as next connection, which is vertex one. Vertex one does not have any connection that is unvisited. Therefore, a shortcut is taken to the closest unvisited vertex. Because $f(1,2) = f(1,3) = 5$, vertex two is, due to the smaller index number, taken as the next connection. In the next step, vertex two is connected with vertex three and vertex three afterwards with the depot to close the cycle. The results are illustrated in Figure 11.

After computing the Hamiltonian cycle, two possibilities exist to transform the undirected solution into the direct solution $\{0,1,2,3,0\}$ or $\{0,3,2,1,0\}$. The shorter path is taken as final result which is $\{0,3,2,1,0\}$ plotted in Figure 12.

The Christofides algorithm guarantees for undirected TSPs that its solution does not exceed a ratio of 1.5. This threshold will be explained in the following. At first, a minimum spanning tree T is constructed. A minimum spanning tree is at most the best value for the TSP problem because, by definition, it is, in total, the smallest tree that connects all vertices. It cannot exceed the TSP because of its optimality, and the TSP also includes one arc additional. Therefore, it is ensured that the total sum of the arc weights of T is smaller or equal to the total sum of the arc weights of the optimal TSP solution.

Because the weights of the matching are at most equal to the summed weight of the paths

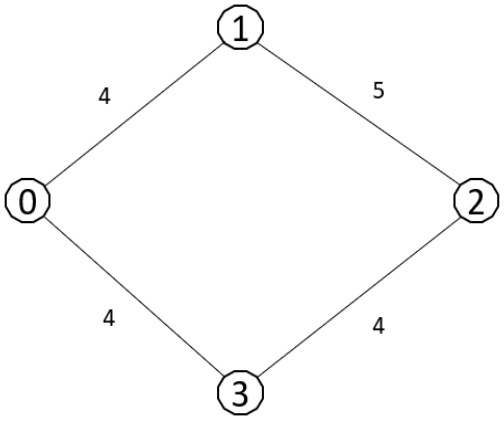

Figure 11. Hamiltonian cycle for Christofides example

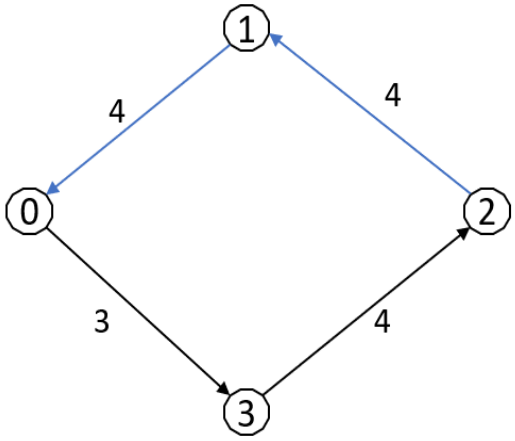

Figure 12. Travel time calculation for Christofides example

and are at most half the number of arcs of the TSP solution, it is ensured that the summed weights of the perfect matching are not greater than 1/2 of the perfect TSP solution.

For the approximation proof, the triangle equation is used. A shortcut can never be longer than a path leading through another vertex if the triangle equation is satisfied. Therefore, shortcuts can, under these circumstances, never increase the total time.

In conclusion, T is in sum not greater than the optimal solution, M is not bigger than half the optimal solution, and the shortcuts do not increase the length. Thus, Christofides has an upper bound of 3/2 for the symmetric TSP. For the TD-TSP, this proof does not hold. The triangle equation does not necessarily hold because of the transformation of the instances

from time-dependent to time-independent ones. Furthermore, shortcuts can change the travel time of all arcs that come subsequently, which leads to an unknown change concerning the optimality. Nevertheless, it is interesting to test if an algorithm that creates reliable results on TSP instances also is able to perform well for TD-TSP instances without considering all the available information.

For estimating the complexity of Christofides, Kruskal's algorithm and the Blossom algorithm have to be considered. All other parts like calculating the set of odd degree vertices are computed in a lower complexity.

Kruskal's algorithm starts with a sorting, e.g. Quicksort that runs in $\Theta(|A|log|A|)$ [26]. Afterwards, checking the sorted list of arcs needs, in the worst case, to check the whole list. Hence, it is in the complexity $\Theta(|A|)$. All other steps also run at most linear. To summarising, Kruskal's algorithm runs in $\Theta(|A|log|A|)$.

The Blossom algorithm makes at most $|A|$ steps to find an alternating path because the cardinality of the minimum perfect matching is at most $|A|/2$. In each iteration of the tree growth, each arc must be considered at most once, as it either creates a cycle, finds an augmenting path or becomes part of the tree. When a cycle is found, a graph can be created by contracting the cycle in $O(|V|)$ by checking all arcs. In the end, there can be at most $|V|$ contraction since at least one node is removed for each contraction. This makes the algorithm perform in total in a run time of $\Theta(|A||V|^2)$. One of the most efficient implementations of the Blossom algorithm is from Gabow [16] that allows solving the same problem in $\Theta(|V|(|A|log|V|)$.

Because Blossom runs in greater complexity than Kruskal's algorithm, the whole Christofides algorithm operates in $\Theta(|A||V|^2)$.

5 BENCHMARKS

Different benchmarks are used for testing the introduced algorithms and compare the results on environments with distinct characteristics. A benchmark for the TD-TSP typically contains a time-dependent travel time matrix and instances. An instance includes a finite set of vertices from the corresponding travel time matrix and a depot. If nothing else is stated in the benchmark, the first vertex is assumed to be the depot. Service times for the vertices can be included. They represent, e.g. loading the goods in the warehouse or delivering them to the customer's location. In the following, the benchmarks from Melgarejo et al. [37], Cordeau et al. [11], Rifki et al. [46] and the TSPLIBs [7, 57] are presented that are used for the experiments of the preceding section.

5.1 Benchmark from Melgarejo et al. [37]

In the paper "A Time-Dependent No-Overlap Constraint: Application to Urban Delivery Problems" from Melgarejo et al. [37] an extension of the no-overlap constraint is presented. The new constraint is tested on a set of benchmarks that are proposed in the paper. The benchmark is generated in the context of the Optimod project Lyon [44]. The Optimod project wants to address mobility problems and improve transportation for individuals, professionals and cargo in an urban environment. Part of this is an improved measurement of urban traffic which was applied by installing sensors in the principal axes of Lyon. A predictive model resulting in a benchmark was created using a time-dependent version of Dijkstra's point-to-point shortest path algorithm. The vertices of the travel time function are chosen randomly from a list of delivery tours of transporters in Lyon. To summarising, it can be assumed that the given data is a realistic benchmark for traffic purposes.

Figure 13 shows the connection between two vertices from the travel time function from Melgarejo et al. [37]. The function is not transformed to satisfy the FIFO function yet. The x-axis describes the time in minutes, starting from 6 am and ending at 7 pm. The y-axis is the travel time from the first to the second vertex in seconds. Originally, there has only been the first part of the time horizon. A longer time horizon was reached by data manipulation. The values from 6:00 am to 0:30 pm are mirrored in the middle to fill the time from 0:30 pm to 7:00 pm. Consequently, traffic congestion occurs in the morning and the evening.

The following describes how the data is structured. In the files, three matrices, 500

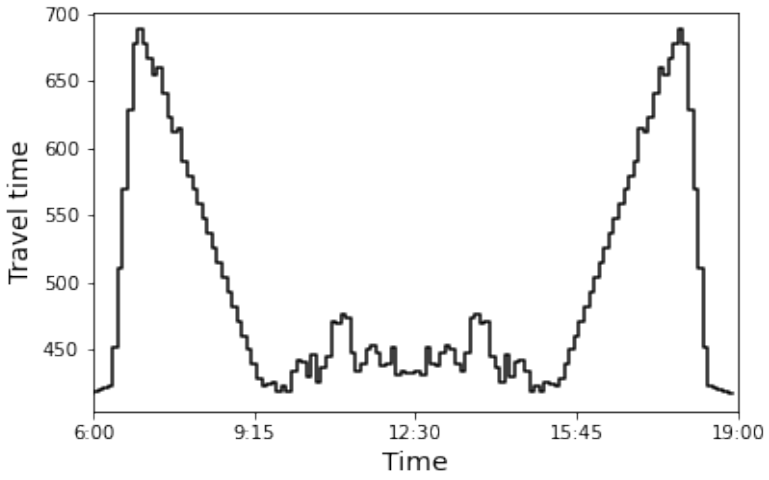

Figure 13. Example for the FIFO transformation of Melgarejo et al. [37]

instances and results are included. Benchmark instances and their results exist with or without defined time windows. For this thesis, only the instances and results without time windows are considered. Instances are available with 10, 20, 30, 50 and 100 vertices. The three matrices contain 255 different locations, the number of time steps is 130, and the duration of a time step is six minutes. In total, the benchmark covers a time horizon of 13 hours. Melgarejo et al. [37] assume that travel time functions tend to underestimate the total time. Therefore they compensate it by enlarging the original function by 10% and 20%. This thesis uses the original, not modified version of the travel time function for all experiments.

Each instance includes service times that indicate how long it takes for a driver to complete the service for a customer. This could, e.g. be delivering food or packages. The service time was selected randomly in a time interval between one and five minutes. The first vertex is always treated as the depot. The depot also has a service time which means that the driver, e.g. loads the vehicle with the needed products.

For most of the considered heuristics, it is important to evaluate a current feasible or unfeasible solution. The calculation is done by adding all travel times of the individual arcs included in the current path. Travel times are calculated by the travel time function f(i,j,t) the time at starting time $t \in T$ from $i \rightarrow j, i, j \in V$.

The given value is modified to satisfy the FIFO property. Melgarejo et al. [37] handle

it in a similar way as Fleischmann et al. [15]. The travel time matrix defines the speed of a vehicle during one time step. At the transition of two time steps, the travel time matrix is smoothed to satisfy the non-passing property. While Fleischmann et al. [15] do this change in both directions, Melgarejo et al. [37] do only modify the travel time value from slower time steps that are followed by a faster time step because this already ensures the FIFO property.

An example for the applied FIFO transformation can be seen in Figure 14. The x-axis shows the time in minutes and the y-axis the travel time from a vertex to another. It pinpoints that it is not necessary to smooth the transition of a faster time step to a slower time step because no possibility exists that a vehicle that leaves later arrives earlier.

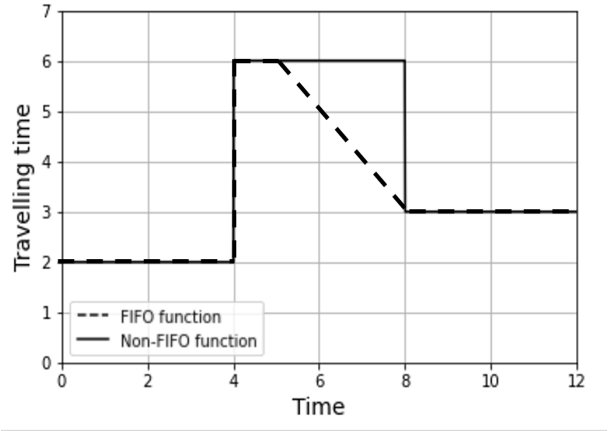

Figure 14. FIFO travelling function transformation used from Melgarejo et al. [37] own representation of [55]

5.2 Benchmark from Rifki et al. [46]

The benchmark of Rifki et al. [46] was introduced in the paper ”On the impact of spatio-temporal granularity of traffic conditions on the quality of pickup and delivery optimal tours”. It is initially designed for time-dependent general pickup and delivery problems (TDGPDP) but can be used for the TD-TSP. The instances were generated from the dynamic microscopic simulator SYMUVIA on a part of the transportation network of Lyon. Simulations like these do not include real traffic data but are a good approximation of the characteristics and give access to the needed granularity of information.

For the experiments, all instances with a spatial coverage of 100% are chosen. The

algorithms got tested on instances with 11, 21, 31, 41, 51 and 61 vertices. For each number of vertices, 150 cost functions exist. These can be subdivided into the time step sizes of six minutes, twelve minutes, twenty-four minutes, one hour and twelve hours, each with thirty different cost functions. The benchmark has a time horizon of twelve hours from 7 am to 7 pm. For all vertices except for the outgoing depot the service time is three minutes.

Figure 15 shows the travel time matrix of one instance with different time step sizes. Matrices with a large number of time steps are very volatile.

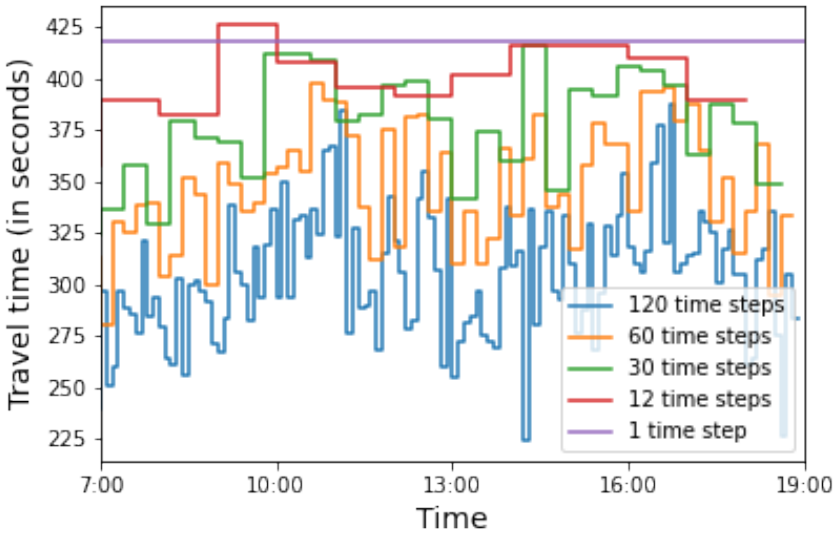

Figure 15. Travel time matrix from Rifki et al. [46]

In each cost function, $|V|^2 * |T|$ values are given. The values are sorted like in the benchmark from Melgarejo et al. [37] that every row represents the arc from i to j over the whole time horizon, and a block of $|V|$ rows consists of all arcs that start from $i \in V$.

Rifki et al. [46] used in their paper like Melgarejo et al. [37] a one-sided approach similar to Fleischmann et al. [15] that satisfies the FIFO property.

5.3 Benchmark from Cordeau et al. [11]

Cordeau et al. [11] proposed their benchmark in the paper ”Analysis and Branch-and-Cut Algorithm for the Time-Dependent Travelling Salesman Problem”. They introduced a new integer formulation for the TD-TSP and solved it with an exact method.

The benchmark is made from a Euclidean square where the depot is located in the middle, and the vertices are situated randomly around the depot using a normal distribution and different standard deviations. Three distinct zones are established with their own travel speed and congestion level characterisation to give the artificial benchmark real traffic characteristics. Each day comprises three time steps representing rush hours in the morning and the evening and a period of lower traffic density in the middle. During the rush hours, the speed is estimated as half of the maximum travel speed. The congestion factor can be influenced by a degradation factor. The time steps are calculated by considering the optimal time of the solution of the asymmetric TSP. Two traffic patterns A and B, are introduced to reflect different types of cities, including low and high traffic in the city centres. This benchmark does not include service times. In contrast to the preceding benchmarks, Cordeau et al. [11] only include three time steps and created the data artificially. Therefore, it can be seen as not as realistic as the the benchmarks from Melgarejo et al. [37] and Rifki et al. [46].

Figure 16 shows two diagrams including the speed and travel time between two vertices of the original benchmark depending on the time. In the travel time diagram, two functions illustrate the Non-FIFO and the FIFO function which is dashed. In their paper, Cordeau et al. [11] implemented the FIFO property as proposed by Ichoua et al. [28]. They assured that no vehicle that leaves later arrives earlier because every vehicle always drives at the current time step speed. When a vehicle does a transition between two time steps, it immediately changes its velocity. Therefore, an unrealistic incentive of waiting is excluded. Figure 16 pinpoints the characteristics of the FIFO function. Contrary to Melgarejo et al. [37] and Rifki et al. [46] the transformation smooths the transition of a faster time step into a slower one but also the transition of a slower time step into a faster time one. This transformation can lead to lesser or greater objective function values than functions that do not satisfy the FIFO property.

The benchmark of Cordeau et al. [11] consists of instances and traffic patterns. The instances include fifteen, twenty, twenty-five, thirty, thirty-five and forty vertices, each size comprising thirty instances. An instance includes the number of vertices, the number of time steps, and the number of clusters. The cluster defines if a vertex position is in the city centre, in a suburb or farther away. The travel speed depends on the cluster. For each instance, there is a distance matrix with $|V|$ vertices, that is of the size $(|V|+1)^2$. The last vertex for each instance is an additional dummy vertex for the depot. The distance between them is initialised

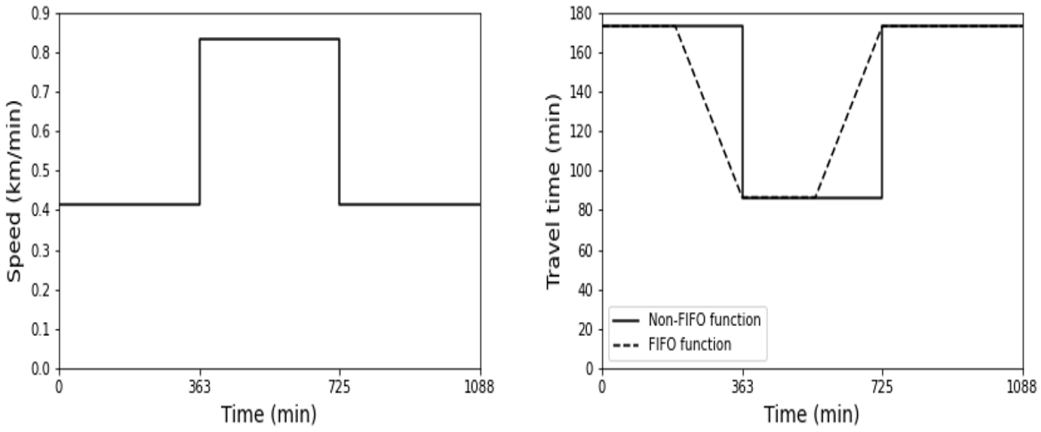

Figure 16. Example for the speed and travel time of Cordeau et al. [11]

as a large enough value to ensure no direct connection between the depot and the dummy vertex.

Furthermore, they provide in their instances solutions of the asymmetric TSP version of the problem, which includes the Hamiltonian cycle, the objective value and the computation time. Their paper postulates that the asymmetric TSP solution is optimal for the TD-TSP solution if all arcs share a common congestion pattern. Last, they deploy a matrix that shows which vertex belongs to which traffic cluster to get the associated travel speed.

5.4 Benchmark from TSPLIBs [7, 57]

TSP libraries have existed for decades and had the function to test time-independent routing problems [45]. They do not contain real-world data and are mainly provided through two-dimensional Euclidean systems with vertices generated randomly or inspired, e.g. by country cities. Because most often only the coordinates are provided, the data has to be seen as artificial and do not represent real-world traffic characteristics. Nevertheless, it is interesting to evaluate the heuristics on time-independent instances to see if and how their ranking changes due to missing time dependencies.

Nine different instances in size ranging from 29 to 194 instances are used. These instances are provided by the University of Waterloo [57] and the Zuse Institute [7]. Only instances are chosen where the vertices are given as coordinates in a two-dimensional Euclidean system. The instances do not include service times.

Figure 23 represents the coordinates of the instance *wi29*. It contains twenty-nine vertices from cities in Western Sahara. The red point is the depot which acts as starting and ending point for every feasible solution. All instances from [57] were created from data of the National Imagery and Mapping Agency in America. Instances from [7] are partly created from Euclidean squares and some instances inspired by geographical conditions. Even though the instances are partly created with real coordinates, they do not reflect real road traffic because they only use the lengths. Therefore, instances of TSPLIBs are artificial.

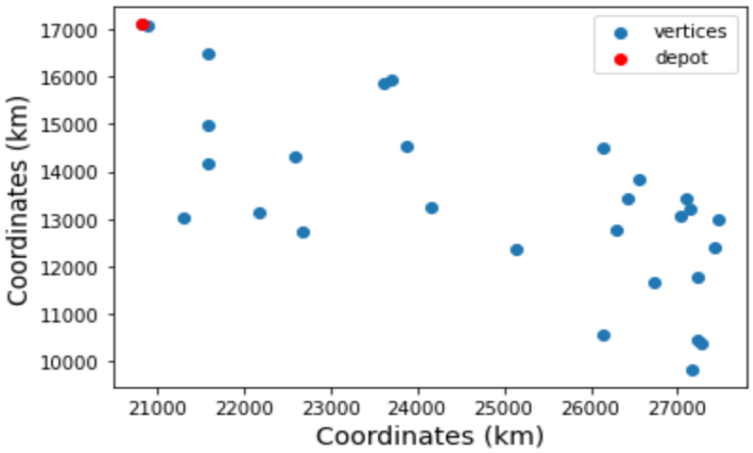

Figure 17. Coordinates of wi29 from TSPLIB [57]

The TSPLIBs instances do, by definition, satisfy the FIFO property because they all operate in the same time step. Therefore, a vehicle arrives later exactly by the amount of time that it left later.

Table 1 summarises the characteristics of the presented benchmarks. The number of vertices ranges for time-dependent instances from 10 to 100 vertices. For time-independent instances, they range from 11 to 194. The benchmark of Rifki et al. [46] is especially interesting for later analysis because the instances are not only given with different numbers of vertices but also with different numbers of time steps. The experimental results of the benchmarks from Melgarejo et al. [37] and Rifki et al. [46] are derived from real-world data and simulation, which is a more accurate representation of traffic. Nevertheless, from an algorithmic, perspective an evaluation on the benchmarks from Cordeau et al. [11] and the TSPLIBs [7, 57] is also interesting.

Name	Melgarejo et al.	Rifki et al.	Cordeau et al.	TSPLIBs
Number of vertices	10 - 100	11 - 61	15 - 40	29 - 194
Number of time steps	130	1 - 120	3	1
Duration time steps	6 min	6 min - 12 h	variable	-
Time horizon	6 am - 7 pm	7 am - 7 pm	-	-
Creation	real traffic	real traffic	artificial	artificial
Number of instances	220	900	180	9

Table 1. Summary of the featured benchmarks [37, 46, 11, 7, 57]

6 BENCHMARK CREATION

An important research topic of Operations Research is the solving of different generalisations of the TSP. Frequently, new exact algorithms, metaheuristics and other heuristics are introduced to provide faster, better and more robust results. To compare these algorithms, benchmarks are needed that are realistic for the possible applications and have the needed granularity of data information. In this section, at first advantages and disadvantages of other benchmarks are analysed, and afterwards, an attempt was made to create a new Benchmark with data from the Yellow Taxis of New York City. In the end, it is described why it was not possible to create a benchmark that would be useful for current research. All the described programming in this chapter has been done with Python.

6.1 Evaluation of other benchmarks

For the TD-TSP, only a limited number of time-dependent benchmarks exist that can be used to test algorithms. First, artificial benchmarks have to be mentioned like the benchmarks from Ichoua et al. [28], Cordeau et al. [11] and Schneider [50]. Artificial benchmarks have the disadvantage that they do not reflect real-world traffic characteristics in their whole complexity. The benchmark from Cordeau et al. [11] is created by a Euclidean matrix that has been subdivided into different zones and different city types to represent different areas of a city. Nevertheless, the benchmark is, front of all, not a representation of real-world data because it only includes three time steps representing morning, noon and evening what is not enough to represent the complexity of urban traffic [18].

Other benchmarks exist that are far from real-world data and are therefore not considered for testing purposes. An example for this is the modified version of the instance BIER127 from Schneider [50]. It includes only two time steps and no further manipulations to imitate real-world traffic. Because of the substantial simplification, it was not considered for testing purposes.

The benchmark of Melgarejo et al. [37] is derived from real-world traffic. Therefore, it should reflect urban traffic characteristics in an optimal way. The downsides of the benchmark are that instances are limited to hundred vertices. For transportation purposes, it is interesting to evaluate an even greater number of instances. Furthermore, the time step duration is fixed to six minutes. For testing purposes, different sizes of time granularity may be interesting. Also,

the time horizon of thirteen hours is created by mirroring the six and a half hours from 6:00 to 12:30 and copying them mirrored to get a benchmark that covers the day until 19:00. It is a strong simplification to assume that afternoon traffic congestion has the same characteristics as morning traffic congestion. Despite the listed downsides, the benchmark is the best publicly available benchmark of real-world traffic data to test algorithms for the TD-TSP.

Rifki et al. [46] provided a benchmark derived from the simulation software SYMUVIA [35]. The provided data respects the complexity of real-world data. SYMUVIA's calculation core contains an assignment module that includes the calculation of vehicle routes and a flow module that includes the calculation of trajectories. The flow module is based on a microscopic model with an extended macroscopic law in order to take into account the specifications of traffic flow in an urban environment like traffic light junctions, public transport, regulation strategy. An advantage of the benchmark is that instances are available with different numbers of vertices and different numbers of time steps. The benchmark provides a large number of testing instances in total, which helps decrease randomness due to the sample size.

In summary, there are only a few benchmarks that can be used to evaluate algorithms for the TD-TSP. A wider range could be helpful to verify results.

6.2 Attempt to create a benchmark

Because a new benchmark can be helpful for recent research and only a limited number of time-dependent benchmarks exist, an attempt was made to create a benchmark. The computation was done with the public data set of "Yellow Taxi Trip Data" from 2015 to 2019 (available at https://data.cityofnewyork.us). Yellow Taxis are able to pick up the passengers in the five boroughs of New York, while Green Taxis have a more limited range. The data is provided by the Mayor's Office of Data Analytics and the Department of Information Technology and Telecommunications.

For the calculation of the travel time matrix, the data from 2015 until 2019 is taking into account. 2020 was excluded because due to the Corona crisis, the number of taxi trips decreased significantly, leading to less congestion. In order to minimise bias in the data, only the years in which a similar road traffic situation can be assumed were considered. Also, the years before 2015 have not been considered because the city's infrastructure changes quickly. Hence, taking a more extended period into account would distort the created travel time matrix.

The data sets provide different information from passenger trips of the taxis. These include the trip distance, information of the pick-up and drop-off position, pick-up time and drop-off time, the costs and more. The year 2019 consists of roughly 113 Million taxi trips.

Real locations from shops in New York are taken as vertices. Figure 18 shows a modified version of the data set that excludes shops that are in the borough of Staten Island or the Newark Liberty International Airport. Shops have been created that are treated as vertices. On the map, they are represented as blue markers. The blue lines separate the locations into different districts. Because there are not enough taxi trips to create a time-dependent travel time function for each location, the benchmark should include not vertices but districts. The travel time from different points in one district to another is assumed to be equal. The district code defines in which zone the location is based. New York 265 taxi districts exist in New York. For the benchmark, only district codes with shop locations are considered that are in total 134.

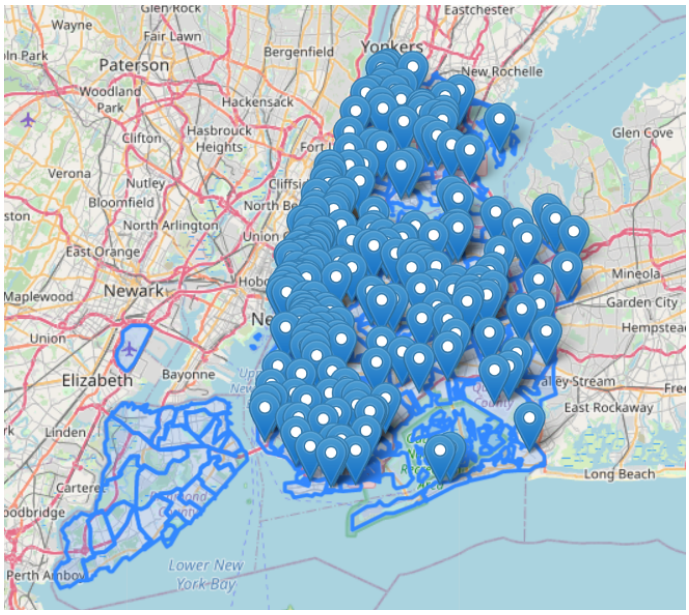

Figure 18. Generated shops in New York, Map from OpenStreetMap [43]

The data was modified for the computation of a travel time matrix with time steps. At first, unimportant information was deleted. The information kept was the trip distance, the pick-up and drop-off coordinates and time, the costs and the district code.

For the data of 2015, the district code of the location was not given. Therefore, it has to be

calculated with the coordinates of the pick-up position and drop-off position. This data was computed with the GeoPandas module of Python. *GeoPandas* [30] extends the data types used by Pandas for easier working with geospatial data. *Pandas* [54] is a tool for data analysis and manipulation that enhances Python. With GeoPandas, it is possible to compute the district code of a vertex by the coordinates and a shape file that describes the form of the districts. The shape file represents the official taxi district map of New York and was also provided by the city of New York (available at: https://data.cityofnewyork.us/Transportation/NYC-Taxi-Zones/d3c5-ddgc). After the calculations for every trip, a pick-up and drop-off district is given.

Besides the location, it is also essential to know how long a trip takes. For each trip, a pick-up time and a drop-off time is available. The trip duration was calculated by taking the difference between both times. It is assumed that the primary traffic occurs in New York from the morning until the evening. Hence, only trips are considered between the twelve hours from 6 am to 6 pm.

After the data modification, some trips lasted with infeasible or biased information. Examples of this are wrong data or waiting times. If a person orders a taxi to wait at a place, this is considered a regular taxi ride, but it does not reflect how long it actually takes to get there. Likewise, data that is obviously wrong, e.g. negative travel time, should also be excluded because inaccurate data can skew the travel time function and can lead to unrealistic results. Nevertheless, long travel times can be explained by congestion that should be included in the benchmark. Therefore, only trips with objectively wrong characteristics were deleted. This includes trips that have a negative travel time, a negative trip distance or a travel duration greater than five hours. The maximum travel duration was chosen by doubling the maximum travel time generated by Google Maps between the southernmost and northernmost points in times of great traffic congestion. By excluding trips with a longer travel duration, the chance of including false data and excluding correct data is minimised.

Now, the travel time and the district codes are available for all trips, and the data was cleaned. In the next step, every trip is evaluated for the travel time matrix. At first, a time step duration has to be set. A duration of five minutes results in 144 time steps. The duration of a time step does not affect the computation steps and can be set as needed. A smaller granularity results in a function that resembles a steady function to a higher degree , which is good for a

realistic benchmark. The downside of a greater number of time steps is that algorithms need more time to solve TD-TSPs because the evaluations have to include more different travel speeds.

The time-dependent travel time matrix is constructed by considering all remaining trips and adding up all trips that start at the same time step and have the same origin and destination district. When all trips are considered, the summed time of the included trips is divided by the number of trips found for each arc at a specific time step.

It can lead to problems if there are no trips for a time step of an arc. This missing data implies that the approximation by given data is not possible, and it is unsure how long trips for this specification take. Surprisingly, it was impossible to create a travel time matrix with a greater number of vertices and time steps due to the given data.

The issues will be explained with the Yellow Taxi data set from 2019. For this year, 113 Million taxi trips are available. From these trips, 85 Million consist of all the necessary information and drive between different districts. By considering only the time steps from 6 am to 6 pm and deleting infeasible trips, the number of trips decreases to 48 Million. Because only the generated shop areas are taken into account, 15 Million rows stay for the matrix calculation. By dividing this by the number of district areas (134) and the number of time steps for a time step duration of five minutes (144), for every arc in a specific time step are 777 trips available. An issue is that these trips are neither equally distributed on the twelve hours nor equally distributed on all district areas. There are neighbouring districts that contain a considerable number of trips, and there exist periods that also include a huge number of trips.

Furthermore, districts exist that are not connected at all with other districts by taxi trips. Affected are some island districts that can not be reached by taxi.

For the data from 2017 to 2019, the 91 districts with the most trips were chosen, and the time step duration was set as fifteen minutes. The number of needed values equals to 397,488 which is $|V|^2 \times |M|$ (91x91x48). From all possible values, 277,375 were zero, which corresponds to roughly 70 % with some vertices that are not connected at all at certain time slots.

With the number of trips and the distribution of these, it was not possible to generate a benchmark that would include a considerable enough number of vertices ($|V| > 30$) and a small time step duration ($d < 1$ hour) without more extensive data manipulation.

It is possible to complete the missing data by computational methods. A first intuitive possibility would be to set all undefined connections as infinity or rather a sufficiently large number. This would lead to avoiding all connections for which no data is available. A downside of this approach is that the assumed travel time is far from reality. Algorithms like Nearest Neighbour that have a limited number of insertion possibilities are more likely obligated to take these connections, which leads to bad results. Because this would influence the performance of specific algorithms due to unrealistic characteristics, this modification was not applied.

A second possible approach is to assume that the missing time step value is approximately as large as the value of the previous time step. Especially for large quantities of time steps, this is a good approximation because the change in the travel time is most of the time insignificant, between, e.g. time steps of five minutes. A problem occurs when a greater share of time steps for a specific arc is missing. If no preceding time step exists, it can not be used to approximate the current value. Then it is possible to take the last existing value and take this value for all following time steps. By doing this, the quality of the created benchmarks diminishes. Hence, it should not be done for a significant number of missing values. Because for several district combinations, the gaps were too big, this approach was rejected.

Another way to approximate the time between two districts is to run a point-to-point shortest path algorithm [12]. This runs through all vertices and checks for the shortest path between each combination. By running this procedure, the triangle equation is automatically satisfied. It also helps fill undefined travel times by assuming that their values are equal to infinity or a sufficiently large number. This approach does not work if a vertex is not connected at all with another vertex. Because several districts are not connected to any other district for some time steps the idea would not work for them. By deleting all the unconnected districts, it is possible to generate a travel time matrix. Nevertheless, the number of unknown travel times stays high for the few remaining vertices. By completing the travel time matrix with a point-to-point shortest path algorithm, the quality would diminish because of the high share of unavailable data.

During the process, it became clear that not every data source is sufficient for creating a new benchmark. It is essential to check if enough data is available to create a data set with a large enough number of vertices and a sufficiently small travel time duration. Also, it is

important to check how the data is distributed.

The data already had the disadvantage that it did not summarise the traffic from locations but districts. This means that the travel time between two vertices would have been estimated by the mean travel time of all trips that share the same pick-up district, drop-off district and time step. This already simplified the travel time matrix to a large extent. By including only a small number of vertices with a long time step duration, the travel time matrix would not help for research purposes because better benchmarks already exist. Also, further data manipulation like the possibilities described would degrade the quality of the real-world data. Therefore, it would not be a contribution to existing research, and only the existing benchmarks will be used for the experiments in the following section.

7 COMPUTATIONAL EXPERIMENTS

In this section, the algorithms from Section 4 are executed on the benchmarks described in Section 5. Depending on the given benchmark, the objective values of the algorithms are compared to an upper bound solution that is either an optimal solution, a given upper bound solution from the instance or the best performing heuristic of each instance. The relative gap to the upper bound describes to which extend the computed solution of the algorithm exceeds the given solution in proportion. For simplicity, the term ”relative gap to the upper bound” is often shortened in this section to ”gap” to improve readability but does always refer to the original formulation.

The heuristics were coded in Java and run on an ASUS TUF A17 with a 3.30-GHz AMD Ryzen 9 and 16 GB of memory.

7.1 Experiments on Melgarejo et al. [37]

In the Benchmark of Melgarejo et al. [37] the results for each instance are given as an optimal or upper bound solution. Because upper bounds are partly used, a heuristic may perform better than the given solution.

In Figure 19 the average gap is shown depending on the number of vertices. The exact numbers can be found in Table 2. The performance of the algorithms does not intersect at any point. This means that the ranking of the heuristics does not change depending on the number of vertices for this benchmark.

The First Fit algorithm performs on average worst, followed by Nearest Neighbour. Both heuristics show an algorithmic increase in their quality in contrast to Cheapest Insertion, Christofides and Savings, which have a more or less constant average gap. It can not be estimated to which extend this effect occurs because the given solutions are for instances with a higher number of vertices, more often not optimal but an upper bound estimation. It is unclear how close the given upper bound is to the optimal solution. Christofides performs better than Cheapest Insertion and Savings best overall.

In Table 2 the numerical results of the experiments are shown. The five tested heuristics are presented with the average, standard deviation, minimum and maximum relative gap to the upper bound. These values are given depending on the number of vertices. The best value for each category is marked with bold letters. For all heuristics, the average gap increases,

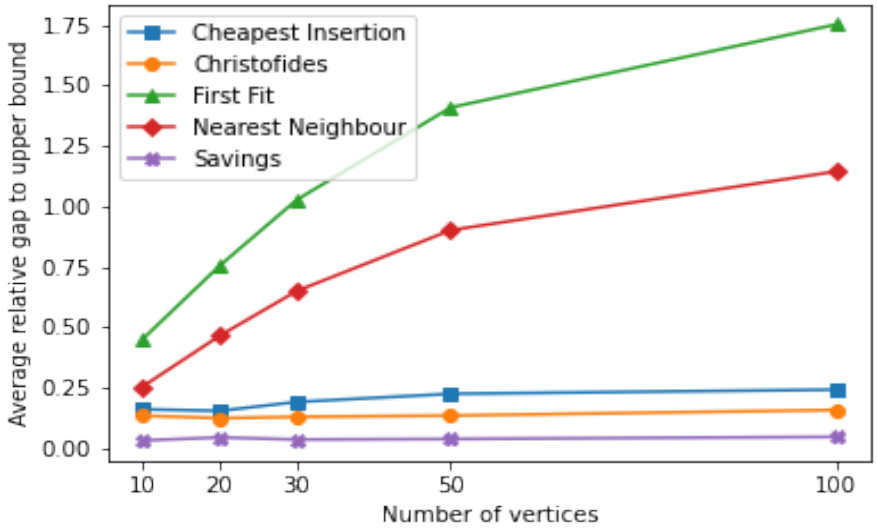

Figure 19. Comparison of the algorithms on the benchmark of Melgarejo et al. [37]

generally speaking, with the number of vertices. Most of the heuristics have a decreasing standard deviation while the number of vertices increases. The minimum gap ranges between -0.011 and 0.694. In the case of a solution smaller than zero, the given upper bound for the instance has a worse objective value than the heuristic.

In this benchmark, Savings outperforms the other heuristics in all categories. For all number of vertices, it had average gaps of less than 5% followed by Christofides with averages lower than 16 % and Cheapest Insertion lower than 25 %. On small instances with ten vertices, Savings and Christofides were able to find a small number of optimal solutions. Savings had compared to Christofides a smaller standard deviation which ranges at a ratio of roughly 50% compared to Christofides. Also, the maximum of Savings was strictly better than all compared algorithms with a maximum gap of 7% to 10%.

In contrast to Savings, Christofides and Cheapest Insertion that performed stable over instances with different numbers of vertices, Cheapest Insertion got relatively worse for an increasing number of vertices. Both had an average gap of more than one for instances with hundred vertices. Conversely, the standard deviation of Nearest Neighbour is much higher.

Heuristic	10 v	20 v	30 v	50 v	100 v
Cheapest Insertion					
- Average gap	0.162	0.155	0.193	0.226	0.243
- Standard deviation of gap	0.070	0.049	0.051	0.044	0.030
- Minimum gap	0.029	0.064	0.076	0.116	0.197
- Maximum gap	0.309	0.292	0.318	0.315	0.307
Christofides					
- Average gap	0.135	0.125	0.132	0.136	0.159
- Standard deviation of gap	0.075	0.043	0.061	0.034	0.036
- Minimum gap	**0**	0.034	-0.004	0.079	0.107
- Maximum gap	0.312	0.248	0.384	0.186	0.235
First Fit					
- Average gap	0.456	0.755	1.031	1.409	1.754
- Standard deviation of gap	0.134	0.169	0.172	0.120	0.095
- Minimum gap	0.195	0.394	0.694	0.203	0.565
- Maximum gap	0.877	0.243	0.536	0.635	0.927
Nearest Neighbour					
- Average gap	0.257	0.463	0.649	0.899	1.146
- Standard deviation of gap	0.122	0.167	0.201	0.180	0.223
- Minimum gap	0.063	0.209	0.301	0.565	0.749
- Maximum gap	0.810	0.094	0.343	0.187	0.532
Savings					
- Average gap	**0.034**	**0.045**	**0.037**	**0.039**	**0.048**
- Standard deviation of gap	**0.032**	**0.019**	**0.023**	**0.017**	**0.019**
- Minimum gap	**0**	**0.004**	**-0.011**	**0.008**	**0.020**
- Maximum gap	**0.101**	**0.090**	**0.091**	**0.071**	**0.096**

Table 2. Results from Melgarejo et al. [37]

7.2 Experiments on Rifki et al. [46]

The results of the heuristics on the benchmark from Rifki et al. [46] were compared with the best performing heuristic of each instance as upper bound. The instances were of different sizes

regarding the number of vertices and the number of time steps. In Figure 20 the performance of the heuristics is compared to the number of vertices that ranged between eleven and sixty-one. For each size, the algorithms were tested on 750 instances. Savings was first in the ranking before Cheapest Insertion, Christofides, Nearest Neighbour and First Fit. The results were similar to the experiments on the benchmark of Melgarejo et al. [37], and Cordeau et al. [11]. Compared to them, the ranking changed between Christofides and Cheapest Insertion. The three best heuristics, Savings, Cheapest Insertion and Christofides, had relatively consistent performances for all number of vertices. By contrast, the average objective value of First Fit and Nearest Neighbour increased with the number of vertices.

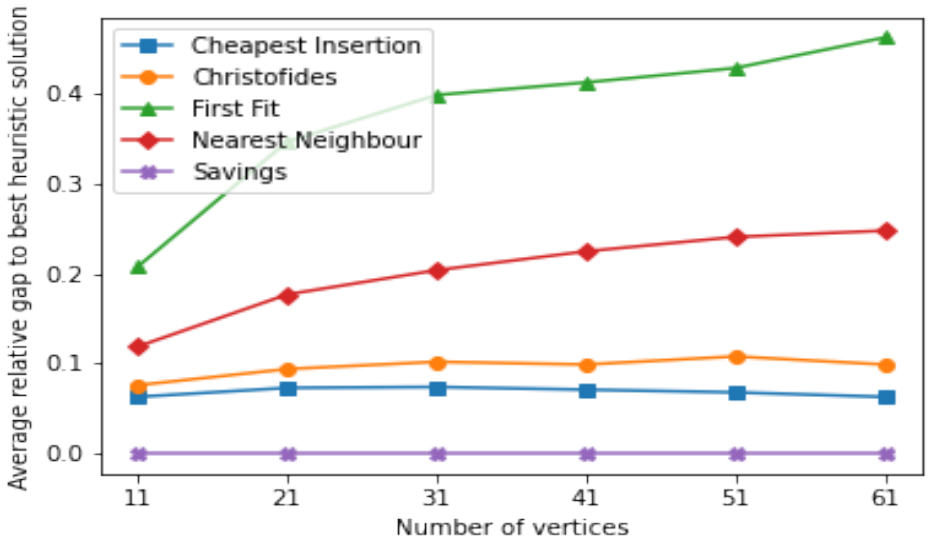

Figure 20. Results generated with the benchmark of Rifki et al. [46] depending on the number of vertices

Table 3 shows the numbers of the average gap from Figure 20 and additional the standard deviation, the minimum and maximum gap of one instance. The standard deviation tends to decrease for an increasing number of vertices. For all heuristics except for Savings, the minimum gap increases with the number of vertices included. While for eleven instances, all heuristics except for Nearest Neighbour were able to find at least one best solution, for instances that include thirty-one vertices or more, Savings consistently ranks first. The maximum gap decreases for Cheapest Insertion and Christofides while it slightly increases for

Nearest Neighbour and First Fit.

Heuristic	11 v	21 v	31 v	41 v	51 v	61 v
Cheapest Insertion						
- Average gap	0.063	0.073	0.074	0.071	0.068	0.063
- Standard deviation of gap	0.040	0.027	0.022	0.023	0.021	0.021
- Minimum gap	**0**	**0**	0.025	0.011	0.021	0.011
- Maximum gap	0.192	0.151	0.127	0.134	0.128	0.120
Christofides						
- Average gap	0.076	0.094	0.102	0.099	0.108	0.099
- Standard deviation of gap	0.039	0.034	0.027	0.019	0.023	0.022
- Minimum gap	**0**	**0**	0.051	0.056	0.062	0.050
- Maximum gap	0.197	0.188	0.194	0.152	0.179	0.177
First Fit						
- Average gap	0.208	0.347	0.399	0.413	0.429	0.463
- Standard deviation of gap	0.074	0.072	0.059	0.055	0.053	0.062
- Minimum gap	**0**	0.199	0.198	0.290	0.308	0.306
- Maximum gap	0.352	0.579	0.516	0.555	0.567	0.641
Nearest Neighbour						
- Average gap	0.119	0.177	0.204	0.225	0.241	0.248
- Standard deviation of gap	0.053	0.058	0.044	0.046	0.050	0.043
- Minimum gap	0.018	0.049	0.099	0.116	0.141	0.144
- Maximum gap	0.285	0.332	0.303	0.355	0.392	0.357
Savings						
- Average gap	**0**	**0**	**0**	**0**	**0**	**0**
- Standard deviation of gap	**0.002**	**0.001**	**0**	**0**	**0**	**0**
- Minimum gap	**0**	**0**	**0**	**0**	**0**	**0**
- Maximum gap	**0.025**	**0.013**	**0**	**0**	**0**	**0**

Table 3. Results from Rifki et al. [46] depending of the number of vertices

Figure 21 measures the performance on the benchmark of Rifki et al. [46] depending on the number of time steps. The whole time horizon of the instances are twelve hours from 7 am

to 7 pm. The number of time steps ranges from 1 to 120. Instances with one time step are TSP instances. 120 time steps equal a time step duration of six minutes. For each of the five time step durations, the heuristics were tested on 900 instances.

The ranking of the algorithms depending on the number of time steps is the same as the one depending on the number of vertices (Figure 20). In distinction to it the relative performance of First Fit and Nearest Neighbour improves significantly and for Cheapest Insertion and Christofides slightly with an increasing number of time steps. It is unclear if the observed effect occurs due to relatively worse results of Savings or improved results of the other algorithms. The more likely explanation of this effect is a change in the relative performance of Savings because this would explain it with the performance of one instead of four algorithms.

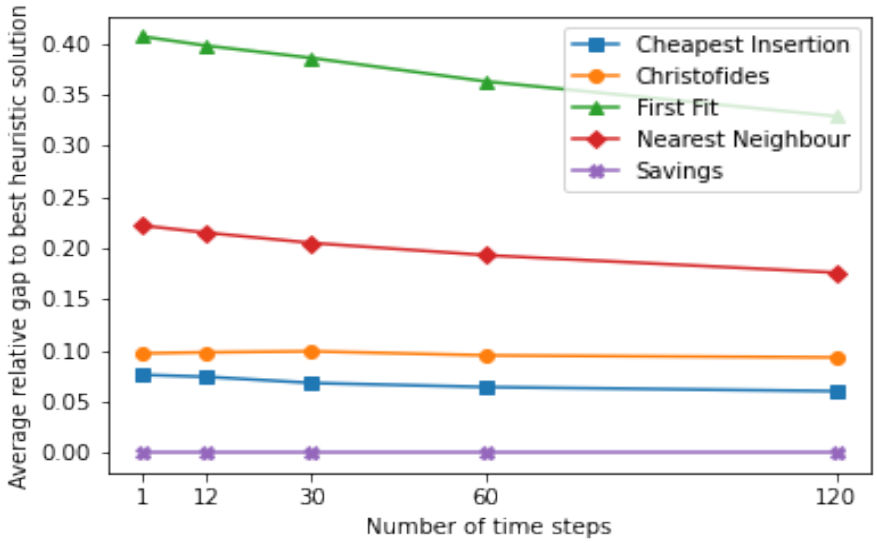

Figure 21. Results generated with the benchmark of Rifki et al. [46] depending on the number of time steps

In Table 4 the heuristics are compared on the benchmark of Rifki et al. [46] depending on the number of time steps. The data correspond to those from the previous tables. The standard deviation stays over all time step sizes relatively stable in contrast to the results from Table 3. Savings has the best minimum and maximum relative gap to the upper bound but does not show clear evidence if the number of time steps influences the relative performance.

Surprisingly, the minimum gap to the upper bound of First Fit is in several cases lower than for Nearest Neighbour. For every subdivision of time, Cheapest Insertion finds at least once the best heuristic solution.

Heuristic	1 t	12 t	30 t	60 t	120 t
Cheapest Insertion					
- Average gap	0.076	0.074	0.068	0.064	0.060
- Standard deviation of gap	0.026	0.025	0.026	0.025	0.029
- Minimum gap	**0**	**0**	**0**	**0**	**0**
- Maximum gap	0.165	0.146	0.134	0.147	0.192
Christofides					
- Average gap	0.097	0.098	0.099	0.095	0.093
- Standard deviation of gap	0.030	0.031	0.032	0.030	0.027
- Minimum gap	0.003	**0**	0.018	0.011	0.014
- Maximum gap	0.197	0.175	0.188	0.194	0.179
First Fit					
- Average gap	0.407	0.398	0.386	0.363	0.329
- Standard deviation of gap	0.110	0.106	0.103	0.094	0.085
- Minimum gap	0.011	0.015	0.027	0.036	**0**
- Maximum gap	0.641	0.614	0.600	0.570	0.518
Nearest Neighbour					
- Average gap	0.222	0.215	0.205	0.193	0.176
- Standard deviation of gap	0.072	0.068	0.060	0.058	0.061
- Minimum gap	0.031	0.018	0.026	0.049	0.018
- Maximum gap	0.392	0.391	0.366	0.330	0.295
Savings					
- Average gap	**0**	**0**	**0**	**0**	**0**
- Standard deviation of gap	**0.001**	**0.002**	**0**	**0**	**0.001**
- Minimum gap	**0**	**0**	**0**	**0**	**0**
- Maximum gap	**0.010**	**0.025**	**0**	**0.006**	**0.011**

Table 4. Results from Rifki et al. [46] depending of the number of time steps

7.3 Experiments on Cordeau et al. [11]

The results of the experiments on the benchmark of Cordeau et al. [11] are shown in Figure 22. For this benchmark, the relative gap to the upper bound is calculated by dividing the objective value of each algorithm with the best objective value among all algorithms of each instance, as done for the benchmark of Rifki et al. [46]. The best heuristic of an instance gets the value one. Each point in the graph represents the average calculation results of one heuristic for instances with a specified amount of vertices. The ranking does again not intersect between instances with different numbers of vertices.

Savings performs best after Christofides, Cheapest Insertion, Nearest Neighbour and First Fit. This is the same ranking as for the benchmark of Melgarejo et al. [37]. The pike at thirty-five vertices for First Fit and Nearest Neighbour can be explained by some instances with an extremely high objective value compared to the best heuristic result. Twenty instances for each considered number of vertices are not enough to compensate for these outliers.

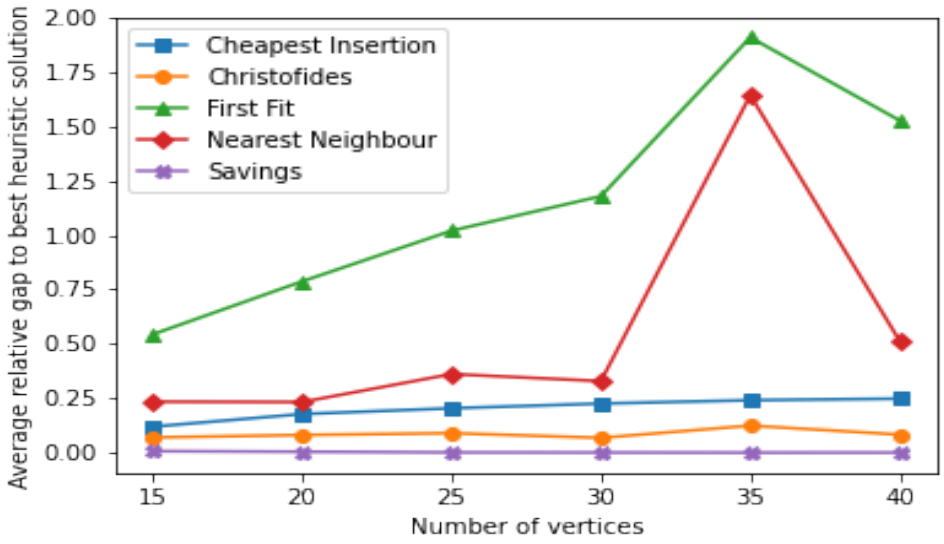

Figure 22. Results generated with the benchmark of Cordeau et al. [11] depending on the number of vertices

Table 5 summarises the numerical results of the experiments from the benchmark of Cordeau et al. [11]. As in Tables 3 and 4, the average, standard deviation, minimum and maximum gap to the best heuristic solution are included for all heuristics depending on the

number of vertices. The number of vertices ranges from five-teen to forty vertices. There is no clear tendency of change depending on the number of vertices for the standard deviation. First Fit and Nearest Neighbour have the highest standard deviation, while Savings has a standard deviation of zero for thirty-five instances because it always performs best for this problem size.

The minimum gap shows that Cheapest Insertion, Christofides and Savings find the best solution for specific instances. Significantly good results performed Savings that had no objective value worse than twelve per cent of the best-found solution on all instances and ranks on average best. First Fit has over all instances on this benchmark the worst maximum gap.

7.4 Experiments on TSPLIBs [7, 57]

Results of the heuristics on instances from the TSPLIBs [7, 57] are presented in Figure 23. On these instances, the computed results of the heuristics are compared with the optimal solutions that are given for all instances. Because no averages are taken, the results are more volatile than in the preceding benchmarks. This also explains the fact that there are more intersections between the graph. First Fit and Nearest Neighbour perform on average worse than Cheapest Insertion, Christofides and Savings.

The numerical results for the TSPLIBs [7, 57] are reported in Table 6. Every row represents one instance with its size and the results of the heuristics. The names consist of an abbreviation and the number of vertices afterwards. They are ordered by the number of vertices they include. The best value for each category is marked with bold letters. On average, Savings produces the best results and after it Christofides, Cheapest Insertion, Nearest Neighbour and First Fit. This is the same ranking as in the experiments from Melgarejo et al. [37] and Cordeau et al. [11].

As in the preceding experiments, Savings performs on most of the instances best. On one instance Christofides' solution had a slightly better objective function value. For most instances, Christofides produces second-best results before Cheapest Insertion. The average gap of Christofides is about half of Cheapest Insertion's value. First Fit performs again worst and Nearest Neighbour second worst. Both algorithms show a high variance in the quality of their solutions.

Heuristic	15 v	20 v	25 v	30 v	35 v	40 v
Cheapest Insertion						
- Average gap	0.090	0.086	0.074	0.067	0.356	0.084
- Standard deviation of gap	0.053	0.041	0.046	0.046	0.104	0.038
- Minimum gap	0.011	**0**	**0**	**0**	0.152	0.001
- Maximum gap	0.205	0.187	0.174	0.166	0.568	0.172
Christofides						
- Average gap	0.070	0.081	0.090	0.068	0.124	0.083
- Standard deviation of gap	0.060	0.071	0.048	0.038	0.070	0.061
- Minimum gap	**0**	**0**	**0**	0.012	**0**	**0**
- Maximum gap	0.271	0.335	0.189	0.210	0.264	0.266
First Fit						
- Average gap	0.544	0.787	1.020	1.179	1.909	1.527
- Standard deviation of gap	0.254	0.281	0.333	0.265	0.328	0.366
- Minimum gap	0.220	0.253	0.367	0.568	1.388	0.623
- Maximum gap	1.349	1.576	1.668	1.622	1.740	1.083
Nearest Neighbour						
- Average gap	0.234	0.233	0.361	0.329	1.643	0.514
- Standard deviation of gap	0.135	0.072	0.154	0.122	0.407	0.155
- Minimum gap	0.075	0.025	0.148	0.163	2.065	0.281
- Maximum gap	0.578	0.365	0.779	0.679	3.625	0.920
Savings						
- Average gap	**0.007**	**0.004**	**0.001**	**0.001**	**0**	**0**
- Standard deviation of gap	**0.023**	**0.012**	**0.004**	**0.002**	**0**	**0.002**
- Minimum gap	**0**	**0**	**0**	**0**	**0**	**0**
- Maximum gap	**0.113**	**0.058**	**0.023**	**0.010**	**0**	**0.009**

Table 5. Results from Cordeau et al. [11]

7.5 Results

For the benchmarks of Melgarejo et al. [37], Cordeau et al. [11], and the TSPLIBs [7, 57] Savings ranked first ahead of Christofides, Cheapest Insertion, Nearest Neighbour and First Fit.

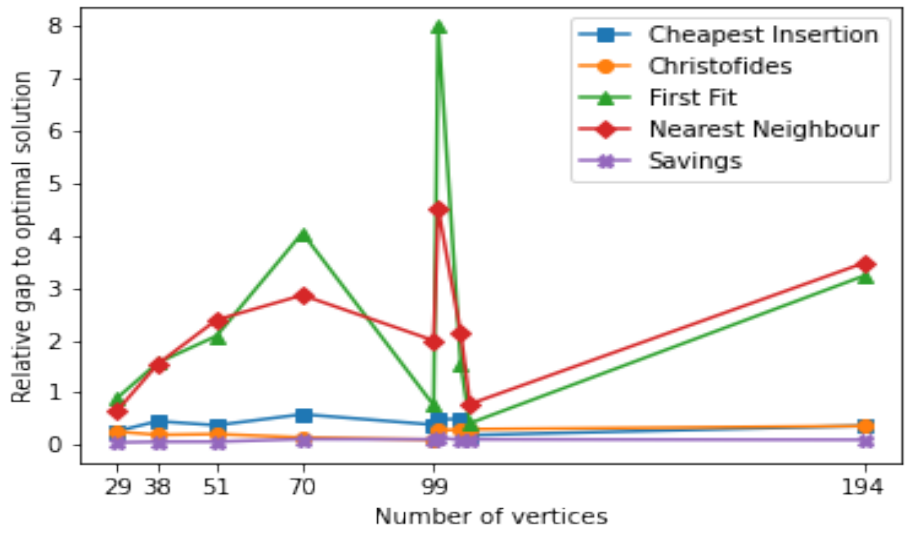

Figure 23. Results generated with the instances from the TSPLIBs [7, 57]

Name	Size	Cheapest Insertion	Christofides	First Fit	Nearest Neighbour	Savings
wi29	29	0.265	0.256	0.894	0.649	**0.049**
dj38	38	0.454	0.194	1.569	1.543	**0.061**
eil51	51	0.377	0.209	2.082	2.390	**0.063**
st70	70	0.587	0.144	4.052	1.862	**0.113**
rat99	99	0.388	**0.102**	0.759	1.996	0.108
kroA100	100	0.491	0.285	7.993	4.519	**0.146**
lin105	105	0.495	0.291	1.537	2.138	**0.092**
pr107	107	0.189	0.301	0.417	0.782	**0.108**
qa194	194	0.366	0.363	3.230	3.476	**0.100**
Average	88	0.401	0.238	2.504	2.262	**0.093**

Table 6. Average gap to optimal solution for the TSPLIBs [7, 57]

The ranking changed for Rifki et al. [46] with regard to Christofides and Cheapest Insertion, which have been interchanged.

Overall, the ranking was surprisingly similar between all benchmarks and even on average equal on three out of four benchmarks. Also, it was unexpected that the rankings for each benchmark were that clear, and intersections in the figures only existed for the TSPLIBs instances. These unambiguous results can be partly explained by the large number of instances used to take the average. Hence, outliers could not be particularly significant for the average results.

The computational tests show that the modified Savings algorithm outperforms all other heuristics significantly with a low average, standard deviation, minimum and maximum gap. Unexpectedly, the results were clear on all considered benchmarks, and Savings even outperformed Christofides on time-independent instances. On three out of four benchmarks, Christofides was on average better than Cheapest Insertion. Although Christofides does not consider time dependencies, it nevertheless produces reliably good results on time-dependent instances without considering time dependency. The standard deviations have been quite similar over all benchmarks. Cheapest Insertion was second-best on average on the benchmark from Rifki et al. [46] that was created by simulation. It is unclear if Cheapest Insertion has an advantage over Christofides on more realistic benchmarks. First Fit was the most simple heuristic and performed for all instances on average worst. The heuristic can be considered random in most cases, which explains that the solution had a high standard deviation. Nearest Neighbour ranked on average second worst. For TD-TSP and TSP instances, the greedy algorithm had a relatively high maximum gap and a worse minimum gap than First Fit in several cases.

In the following, possible explanations are given for the rankings of the experiments. First Fit corresponds to a random search if the vertices are not sorted. It was expected that a random solution performs on average worse than heuristics that operate under a certain principle to provide a better solution.

Nearest Neighbour is a greedy algorithm that connects the vertex closest to the current vertex in each iteration. Because it always only takes the next step into account and has at most $|V|-1$ possibilities for the next iteration, it is worse than Cheapest Insertion that has first more insertion possibilities and second already considers the depot in the end. An advantage of Savings over Cheapest Insertion is that it creates a graph that directly includes all vertices. First Fit, Nearest Neighbour and Cheapest Insertion add with each iteration a new vertex to

the solution. An early extended graph can explain advantages over the other algorithms.

Christofides is a well-performing algorithm for TSP instances but does not consider the time dependency, which leads to worse solutions. For Christofides, the median was taken to compute the time-independent travel time functions. Experiments that used the average instead of the median have shown no clear tendency which transformation fits better. A further examination of this could help to improve the time-dependent implementation of Christofides. The advantages of Savings include the consideration of the time dependency and also directly taking all vertices into account by an extended graph.

The benchmark from Rifki et al. [46] was evaluated depending on the number of vertices and time steps. Interestingly, Savings performed best on every given instance with an increasing number of vertices while there is no clear tendency for an increasing number of time steps. Even though Savings performs on average best on all instances, compared to the other heuristics, the algorithm is particularly strong on instances with a large number of vertices. An explanation for this could be that all vertices are included in the initial solution from the beginning. A result of the experiments is that the number of time steps and vertices affects the standard deviation. While for an increasing number of vertices, the standard deviation for most algorithms decreased, it stayed more or relatively stable over different numbers of time steps.

The results are very similar to those known from the TSP literature. Ong et al. [42] showed that Savings has, in their experiments, on average, a better performance than Nearest Neighbour and Cheapest Insertion. This can be confirmed with the results from the TSP and TD-TSP benchmarks with the adapted algorithms. Johnson et al. [29] show in their experiments that a modified Christofides algorithm performance best after Savings and Nearest Neighbour. In this thesis, the modified Savings algorithm performs on average better than Christofides for TSP and TD-TSP instances. This can be explained due to the fact that the Savings was adapted in an advanced version, and Christofides was adapted in a basic implementation.

Greater adjustments of the algorithms like Johnson et al. [29] done it for Christofides, and it is done in this thesis with Savings can significantly change their performances.

In general, during the experiments could be observed that the construction heuristics have a similar ranking for different numbers of time steps and vertices including the time-independent

case. These results suggest that good construction heuristics for the TSP can perform well on TD-TSP instances if adapted adequately.

8 CONCLUSION

In this thesis, Cheapest Insertion, Christofides, First Fit, Nearest Neighbour and Savings have been investigated on benchmarks from Melgarejo et al. [37], Rifki et al. [46], Cordeau et al. [11] and TSPLIBs [7, 57]. All these deterministic construction heuristics were compared on TSP and TD-TSP instances with an upper bound, which was either a given optimal or upper bound solution or the best performing heuristic for each instance. The heuristic solution was divided by this upper bound to compute a relative indicator. The performance of the heuristics was compared by the average, standard deviation, minimum and maximum of the relative gap to the upper bound. All algorithms were adapted to work for the TD-TSP. The modified Savings algorithm outperformed all other heuristics on TSP and TD-TSP instances significantly on all considered criteria. Christofides also produced reliably good results without taking into account time-dependency.

Also, an attempt was made to create a new time-dependent benchmark with the data of Yellow Taxis in New York. This includes a large number of taxi trips' pick-up and drop-off times and corresponding coordinates. Initially, the idea was to bundle the information using taxi districts for all pick-up and drop-off locations. Afterwards, all given trips for a specific time step from one district to another were averaged to create a benchmark. The result was discarded because the quality of the benchmark was not good enough due to insufficient information in the underlying database.

The present findings might help future research and real-world applications to choose construction heuristics better suited to their purposes. The solutions may serve as a direct solution, the initial solution for an improvement heuristic or an upper bound for exact methods. Furthermore, comparisons of newly established algorithms for the TD-TSP should not be drawn with the weakest algorithm but at least with a well-performing construction heuristic according to the state of the art.

The present study was limited by the number of considered heuristics and existing benchmarks. Further experimental investigation of models that have not yet been adapted, such as additional insertion heuristics, may be helpful to fill the comparison of deterministic construction heuristics for the TD-TSP. For time-dependent construction heuristics, future work on probabilistic approaches is essential. Furthermore, new benchmarks with optimal solutions for the TD-TSP would help to verify results. A particular focus should be on the introduction

of new benchmarks derived from real road traffic that would improve comparability for real applications. Especially benchmarks with a large number of vertices help to compare construction heuristics that are fast on bigger size problems. Future studies should validate the findings of this thesis with the introduction of new benchmarks.

REFERENCES

[1] Anna Arigliano, G. Ghiani, and A. Grieco. “Time Dependent Traveling Salesman Problem with Time Windows : Properties and an Exact Algorithm”. In: 2015.

[2] Anna Arigliano et al. “Time-dependent asymmetric traveling salesman problem with time windows: Properties and an exact algorithm”. In: *Discrete Applied Mathematics* 261 (2019), pp. 28–39.

[3] Giorgio Ausiello et al. *Complexity and approximation: Combinatorial optimization problems and their approximability properties*. Springer Science & Business Media, 2012.

[4] Nagraj Balakrishnan, Abilio Lucena, and Richard T. Wong. “Scheduling examinations to reduce second-order conflicts”. In: *Computers & Operations Research* 19.5 (1992), pp. 353–361. ISSN: 0305-0548.

[5] JE Beasley. “Adapting the savings algorithm for varying inter-customer travel times”. In: *Omega* 9.6 (1981), pp. 658–659. ISSN: 0305-0483.

[6] Vahid Beiranvand, Warren Hare, and Yves Lucet. “Best practices for comparing optimization algorithms”. In: *Optimization and Engineering* 18.4 (2017), pp. 815–848.

[7] Zuse Institute Berlin. *TSPLIB Symmetric Traveling Salesman Problem Instances*. URL: `http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/` (visited on 09/30/2021).

[8] Robert G Bland and David F Shallcross. “Large travelling salesman problems arising from experiments in X-ray crystallography: A preliminary report on computation”. In: *Operations Research Letters* 8.3 (1989), pp. 125–128. ISSN: 0167-6377.

[9] Nicos Christofides. *Worst-case analysis of a new heuristic for the travelling salesman problem*. Tech. rep. Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group, 1976.

[10] Geoff Clarke and John W Wright. “Scheduling of vehicles from a central depot to a number of delivery points”. In: *Operations research* 12.4 (1964), pp. 568–581.

[11] Jean-François Cordeau, Gianpaolo Ghiani, and Emanuela Guerriero. “Analysis and Branch-and-Cut Algorithm for the Time-Dependent Travelling Salesman Problem”. In: *Transportation Science* 48.1 (2014), pp. 46–58. ISSN: 00411655, 15265447.

[12] Ugur Demiryurek, Farnoush Banaei-Kashani, and Cyrus Shahabi. “A Case for Time-Dependent Shortest Path Computation in Spatial Networks”. In: *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems*. GIS ’10. San Jose, California: Association for Computing Machinery, 2010, pp. 474–477. ISBN: 9781450304283.

[13] Alberto V Donati et al. “Time dependent vehicle routing problem with a multi ant colony system”. In: *European journal of operational research* 185.3 (2008), pp. 1174–1191.

[14] Jack Edmonds. “Paths, Trees, and Flowers”. In: *Canadian Journal of Mathematics* 17 (1965), pp. 449–467.

[15] Bernhard Fleischmann, Martin Gietz, and Stefan Gnutzmann. “Time-varying travel times in vehicle routing”. In: *Transportation science* 38.2 (2004), pp. 160–173.

[16] Harold N Gabow. “Data structures for weighted matching and nearest common ancestors with linking”. In: *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*. 1990, pp. 434–443.

[17] Michel Gendreau, Gianpaolo Ghiani, and Emanuela Guerriero. “Time-dependent routing problems: A review”. In: *Computers & Operations Research* 64 (2015), pp. 189–197. ISSN: 0305-0548.

[18] Nikolas Geroliminis and Carlos F. Daganzo. “Existence of urban-scale macroscopic fundamental diagrams: Some experimental findings”. In: *Transportation Research Part B: Methodological* 42.9 (2008), pp. 759–770. ISSN: 0191-2615.

[19] Gianpaolo Ghiani et al. “Lifting the Performance of a Heuristic for the Time-Dependent Travelling Salesman Problem through Machine Learning”. In: *Algorithms* 13.12 (2020). ISSN: 1999-4893.

[20] F Glover and A P Punnen. “The travelling salesman problem: new solvable cases and linkages with the development of approximation algorithms”. In: *Journal of the Operational Research Society* 48.5 (1997), pp. 502–510. eprint: https://doi.org/10.1057/palgrave.jors.2600392.

[21] Fred Glover et al. “Construction heuristics for the asymmetric TSP”. In: *European Journal of Operational Research* 129.3 (2001), pp. 555–568. ISSN: 0377-2217.

[22] Gregory Gutin and Anders Yeo. “Introduction to domination analysis”. In: (2005).

[23] Pierre Hansen and Nenad Mladenović. “First vs. best improvement: An empirical study”. In: *Discrete Applied Mathematics* 154.5 (2006). IV ALIO/EURO Workshop on Applied Combinatorial Optimization, pp. 802–817. ISSN: 0166-218X.

[24] Christoph Hansknecht, Imke Joormann, and Sebastian Stiller. “Cuts, primal heuristics, and learning to branch for the time-dependent traveling salesman problem”. In: *arXiv preprint arXiv:1805.01415* (2018).

[25] Carl Hierholzer and Chr Wiener. “Über die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren”. In: *Mathematische Annalen* 6.1 (1873), pp. 30–32.

[26] Charles AR Hoare. “Quicksort”. In: *The Computer Journal* 5.1 (1962), pp. 10–16.

[27] Silu Huang, Ada Wai-Chee Fu, and Ruifeng Liu. “Minimum spanning trees in temporal graphs”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 2015, pp. 419–430.

[28] Soumia Ichoua, Michel Gendreau, and Jean-Yves Potvin. “Vehicle dispatching with time-dependent travel times”. In: *European Journal of Operational Research* 144.2 (2003), pp. 379–396. ISSN: 0377-2217.

[29] D. Johnson and L. A. McGeoch. *Local Search in Combinatorial Optimization*. 1st. USA: John Wiley & Sons, Inc., 1997, pp. 215–310. ISBN: 0471948225.

[30] Kelsey Jordahl et al. *geopandas/geopandas: v0.8.1*. Version v0.8.1. July 2020.

[31] Robert L Karg and Gerald L Thompson. “A heuristic approach to solving travelling salesman problems”. In: *Management science* 10.2 (1964), pp. 225–248.

[32] Richard M Karp. “Reducibility among combinatorial problems”. In: *Complexity of computer computations*. Springer, 1972, pp. 85–103.

[33] Bernhard Korte. “Steuerung von Produktionsmaschinen”. In: *Unternehmensdynamik*. Springer, 1991, pp. 183–206.

[34] Joseph B Kruskal. “On the shortest spanning subtree of a graph and the traveling salesman problem”. In: *Proceedings of the American Mathematical society* 7.1 (1956), pp. 48–50.

[35] LICIT.

[36] Chryssi Malandraki and Mark Daskin. “Time Dependent Vehicle Routing Problems: Formulations, Properties and Heuristic Algorithms”. In: *Transportation Science* 26 (Aug. 1992), pp. 185–200.

[37] Penélope Aguiar Melgarejo, Philippe Laborie, and Christine Solnon. “A Time-Dependent No-Overlap Constraint: Application to Urban Delivery Problems”. In: *Integration of AI and OR Techniques in Constraint Programming*. Ed. by Laurent Michel. Cham: Springer International Publishing, 2015, pp. 1–17. ISBN: 978-3-319-18008-3.

[38] Clair E Miller, Albert W Tucker, and Richard A Zemlin. “Integer programming formulation of traveling salesman problems”. In: *Journal of the ACM (JACM)* 7.4 (1960), pp. 326–329.

[39] Juan Jose Miranda-Bront, Isabel Mendez-Diaz, and Paula Zabala. “An integer programming approach for the time-dependent TSP”. In: *Electronic Notes in Discrete Mathematics* 36 (2010), pp. 351–358.

[40] Agustín Montero, Isabel Méndez-Díaz, and Juan José Miranda-Bront. “An integer programming approach for the time-dependent traveling salesman problem with time windows”. In: *Computers & Operations Research* 88 (2017), pp. 280–289. ISSN: 0305-0548.

[41] Christian Nilsson. “Heuristics for the traveling salesman problem”. In: *Linkoping University* 38 (2003), pp. 00085–9.

[42] H.L. Ong and H.C. Huang. “Asymptotic expected performance of some TSP heuristics: An empirical evaluation”. In: *European Journal of Operational Research* 43.2 (1989), pp. 231–238. ISSN: 0377-2217.

[43] OpenStreetMap contributors. *Planet dump retrieved from https://planet.osm.org*. `https://www.openstreetmap.org`. 2017.

[44] *Optimod’Lyon*. 2021. URL: `http://www.business.greaterlyon.com/news/optimodlyon-partnership-agreement-signed-in-april-2012-887.html` (visited on 09/30/2021).

[45] Gerhard Reinelt. “TSPLIB—A traveling salesman problem library”. In: *ORSA journal on computing* 3.4 (1991), pp. 376–384.

[46] Omar Rifki, Nicolas Chiabaut, and Christine Solnon. “On the impact of spatio-temporal granularity of traffic conditions on the quality of pickup and delivery optimal tours”. In: *Transportation Research Part E: Logistics and Transportation Review* 142 (2020), p. 102085.

[47] Débora P Ronconi. “A note on constructive heuristics for the flowshop problem with blocking”. In: *International Journal of Production Economics* 87.1 (2004), pp. 39–48. ISSN: 0925-5273.

[48] Daniel J. Rosenkrantz, Richard E. Stearns, and Philip M. Lewis II. “An Analysis of Several Heuristics for the Traveling Salesman Problem”. In: *SIAM Journal on Computing* 6.3 (1977), pp. 563–581.

[49] Liang-zhong RUAN, Li ZHANG, and Chao WU. “A New Tour Construction Algorism and its Application in Laser Carving Path Control”. In: *Journal of Image and Graphics* 6 (2007).

[50] Johannes Schneider. “The time-dependent traveling salesman problem”. In: *Physica A: Statistical Mechanics and its Applications* 314.1 (2002). Horizons in Complex Systems, pp. 151–155. ISSN: 0378-4371.

[51] Peng Sun et al. “The time-dependent capacitated profitable tour problem with time windows and precedence constraints”. In: *European Journal of Operational Research* 264.3 (2018), pp. 1058–1073. ISSN: 0377-2217.

[52] Wahyudin P Syam and Ibrahim M Al-Harkan. “Improvement and comparison of three metaheuristics to optimize flexible flow-shop scheduling problems.” In: *International Journal of Engineering Science and Technology* 4 (2012), pp. 373–383.

[53] Hisashi Tamaki et al. “A comparison study of genetic codings for the traveling salesman problem”. In: *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*. IEEE. 1994, pp. 1–6.

[54] The pandas development team. *pandas-dev/pandas: Pandas*. Version latest. Feb. 2020.

[55] Faycal A Touzout, Anne-Laure Ladier, and Khaled Hadj-Hamou. “Time-dependent travel-time constrained Inventory Routing Problem”. In: *International Conference on Computational Logistics*. Springer. 2020, pp. 151–166.

[56] Duc Minh Vu et al. “Solving time dependent traveling salesman problems with time windows”. In: *Optimization online* 6640 (2018).

[57] University Waterloo. *TSPLIB National Traveling Salesman Problems*. URL: `https://www.math.uwaterloo.ca/tsp/world/countries.html` (visited on 09/30/2021).