universität wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

„Ring Star Problem with User Equilibrium Constraints“

verfasst von / submitted by

Nadja Friesen

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Master of Science(M.Sc.)

Wien, 2017/ Vienna 2017

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	A 0660920
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Qunatitative Economics, Management and Finance
Betreut von / Supervisor:	Univ.-Prof. Mag. Dr. Walter Gutjahr

Acknowledgments

I would like to thank my parents and my sister for their never ending support and for always believing in me, and for being patience and understanding during my work on this thesis.

In addition, I would like to thank my supervisor Prof. Dr. Walter Gutjahr for giving me the opportunity to work on this interesting topic and his support.

Furthermore, I would like to thank my friends for being there for me in stressful times and being optimistic in various ways.

Abstract

In this Master's Thesis the bi-objective Ring Star Problem is solved. Different to the general formulation of the RSP, the assignment problem was extended by User Equilibrium constraints. In that way the assignment to ring is not just based on the distance to it, but also on the "service quality", respectively the incoming flow to a node on the ring. As a solution approach the NSGA-II was implemented, with nested Clarke&Wright savings algorithm and Frank Wolfe algorithm. The savings algorithm was used for solving a TSP for each solution. For the approximation of the User Equilibrium the Frank Wolfe algorithm was applied, after the assignment problem was transferred into a network flow problem. Since no benchmark solution for such a formulation of a RSP exists, test instances were generated and enumerated. These were used for the evaluation of the quality of the implemented solution method.

Zusammenfassung

In dieser Masterarbeit wird das Ring Star Problem (RSP) als ein bikriterielles Problem gelöst. Im Unterschied zu der allgemeinen Formulierung des RSP, wird hier die Zuweisung der Knoten zu dem Ring mit Hilfe des Benutzergleichgewichts bestimmt. Folglich ist diese Zuweisung nicht nur basierend auf der Entfernung eines Knotens zu Ring, sondern auch auf der "Service Qualität" dort. Diese wird an jedem Knotenpunkt an dem Ring ermittelt und hängt von den insgesamt zugewiesenen Bedarf ab. Als Lösungsmethode wurde der NSGA-II implementiert. Die einzelnen Zielfunktionen wurden jeweils mit dem Clarke&Wright Algorithmus und dem Frank Wolfe Algorithmus gelöst. Wobei der Savings-Algorithmus ein Traveling Salesman Problem (TSP) für den Ring gelöst hat und der Frank Wolfe Algorithmus die Zuweisung der Knoten zu dem Ring. Um diesen anwenden zu können wurde die zweite Zielfunktion in ein Netzwerk-Problem umformuliert. Da keine Benchmark Lösungen für dieses Mehrziel-Problem vorliegen wurden kleine Test Instanzen generiert und enumeriert. Anhand dieser wurde die Lösungsqualität der implementierten Metaheuristik evaluiert.

List of Abbreviations

GA Genetic Algorithm

MCP Median Cycle Problem

MOOP Multi-objective Optimization Problem

NSGA-II Non-dominated Sorting Genetic Algorithm II

OD Origin-Destination

RSP Ring Star Problem

SP Service Point

TSP Travelling Salesman Problem

WE Wardrop Equilibrium

List of Tables

List of Figures

Contents

1. Introduction

The purpose of this work is to implement a solution approach to solve a Ring Star Problem (RSP) which has extended the assignment problem in a way that the service quality at the ring is also considered. Additionally the RSP will be solved as a multi-objective optimization problem.

The Ring Star Problem is a two-stage optimization problem. Firstly, a decision maker chooses from a given set of vertices a subset, the so called Service Points (SP). These provide a homogeneous good to all other vertices and themselves. The SP's are connected by a ring with each other, which represents a tour passing all of them. By this tour the SP's are supplied. On the second stage the demand of each vertex is assigned to one of the SP's. Each node has a positive demand, which will be fully supplied.

Example of a real-world applications for such a problem formulation is the implementation of a public transportation system. In this example the stops of a tour of a vehicle are represented by the SP's. Additional examples are the planning of location of postal boxes or offices and the implementation of a network of branch stores. A widely discussed application of the RSP is for the setup of an emergency facilities network after natural catastrophes, which helps to provide relief goods to the affected people(cf.[10],[9]).

Two conflicting cost functions arising in this optimization problem: the ring costs and the assignment costs. The ring costs consist of all costs concerning the ring, such as the tour costs dependent on its length. These are decreasing if less SP's are opened. The assignment costs consist in main literature of the overall travelling costs to the ring and are minimal if at each vertex a SP is set up. It is observable that the people in real world application do not always choose the closest stop to supply their demand and take additional decision factors into consideration. To model such a behaviour a function representing the service quality at a service point is introduced. The service quality of a SP decreases with rising demand assigned to it. In sum the assignment problem consists in optimization of the travelling costs to a SP and the service quality.

In this Master's Thesis firstly a bi-objective formulation of the RSP is introduced with an extension of the assignment problem by a service quality function. The two partial problem are solved independently. Whereby the first stage problem is solved as a travelling salesman problem for the SP's. The assignment problem is transferred into the User Equilibrium formulation to solve it. A multi-objective optimization approach was used to find a set of non-dominated solution, which will be presented to a decision manager. To do so the non-dominated sorting genetic algorithm II (NSGA-II) was implemented with the aim to provide a good non-dominated set. Hence this approach is a random search method, solution heuristics for the partial problems were used. The TSP for the ring was solved by the Clarke&Wright savings algorithm. The Frank Wolfe algorithm was used to solve the assignment problem. The performance of the NSGA-II was tested on generated

test instances. The test results were compared to non-dominated sets resulting out of enumeration of the test instances.

The Master's Thesis is organised as followed: In the 2. chapter the mathematical problem is set up. Firstly, the partial optimization problems are defined and then the overall formulation is introduced. For better understanding, in chapter 3 the concepts of multi-objective optimization and genetic algorithms are shortly described. A closer look at most significant present literature is taken in chapter 4. This is followed by the introducing and explanation of the optimization method, NSGA-II. Also, the solution methods for the partial problems are explained in the 5th chapter. Namely the Clarke&Wright savings algorithm for the TSP and the Frank Wolfe algorithm for solving the User equilibrium. Followed up by the analysis and presentation of the computational results in chapter 6.

2. Problem Fomulation

The Ring Star Problem will be solved as a bi-objective optimization problem. Given a set of connected vertices a decision maker selects a subset of them, which covers the demand for a service or a good for all vertices. Further on it is referred to them as Service Points (SP's). A cycle through all SP's is formed, the so-called ring. The first objective is to minimize the ring costs by solving a Travelling Salesman Problem (TSP). The ring costs include the non-negative costs, which are connected to the length of the ring. No other costs related to the ring are considered. Given the information which vertices are forming the ring, the other nodes are assigned to the vertices on the cycle. In the classical formulation of the RSP, the closest SP is selected. In this work additionally to the distance as decision factor of assignment the service quality at a SP is taken into account. The resulting second objective of the problem is to minimize the non-negative traveling costs [21] and maximize the service quality at a SP.

In the beginning section of this chapter the basic framework of the bi-objective RSP will be described. The following sections are splitting the RSP in two different partial problems. Firstly, the TSP is introduced with its assumptions and constraints. Followed by the extension of the assignment problem with a service quality function as an additional decision factor. The last section will outline the whole bi-objective RSP with equilibrium constraints.

2.1. Basic Model

The model of bi-objective RSP presented by Liefhooge et al. 2010 [21] and the Median Cycle Problem (MCP) of Labbé et al. 1999 [19] are used as basic framework in this work. A complete mixed graph $G = (V, E, A)$ is given, where $V = \{v_1, v_2, ..., v_n\}$ is the set of vertices, $E = \{[v_i, v_j] | v_i, v_j \in V, i < j\}$ the set of edges and $A = \{(v_i, v_j) | v_i, v_j \in V\}$ the set of arcs. Please note that A also contains loops (v_i, v_i). In real-world applications the n vertices describe settlements or population centre [9]. In this work each vertex is considered as population centre with specific demand p_i which represent the number of inhabitants of it [9]. The subset $V' \subseteq V$ contains the vertices which are denoted as service points. At least one vertex is in V'. An empty set V' would not make any sense, because then the demand of no one can be satisfied [10]. It follows that in total there are $(2^n - 1)$ possible combinations of either a vertex is on the ring or not [10]. The SP's are supplying the total demand $P = \sum_{v_i \in V} p_i$. The demand at a SP is usually covered by itself [19]. To each edge $[v_i, v_j] \in E$ non-negative ring costs c_{ij} are assigned. Consequently, the total ring costs are connected to the length of the ring, which is formed by edges between the SP's. By d_{ij} the costs of traversing the arc $(v_i, v_j) \in A$ are given.

These costs are proportional to the length of the corresponding arc.

The first objective function of the bi-objective RSP as proposed in [21] is to minimize the costs of the ring and is formulated as followed

$$\sum_{[v_i,v_j]\in E} c_{ij}x_{ij} \quad \rightarrow \min \tag{2.1}$$

where the variable x_{ij} is binary and equal to 1 if and only if the edge $[v_i, v_j] \in E$ is assigned to the ring.

$$x_{ij} = \begin{cases} 1 & \text{if edge } [v_i, v_j] \in E \text{ belongs to the ring} \\ 0 & \text{else.} \end{cases}$$

Liefhooge et al. 2010 [21] proposed as the second objective to minimize the total assignment costs

$$\sum_{v_i\in V\setminus V'} \min_{v_j\in V'} d_{ij} \quad \rightarrow \min . \tag{2.2}$$

For the assignment of a non-visited vertices to the ring the arc with the minimum costs is chosen. This is equivalent of choosing the closest SP for each population centre. Similar formulations of the second cost function are found in the most works on the RSP. Hence the assignment cost are funded by the inhabitants at the settlements is it reasonable to multiply each arc by p_i, which fulfils $(\min_{v_j\in V'} d_{ij}|_{i\in V\setminus V'})$.

2.2. The Travelling Salesman Problem

Since the 19th century the Traveling Salesman Problem (TSP) is very popular in combinatorial optimization [14] and widely discussed in different research fields like Mathematics, Operations Research, Computer Science or Physics and Biology [23].

Given a list of clients the salesman needs to find a closed tour starting from his home on which he visits each of his clients exactly once and returns home. The distance between each pair of two stops and between his home and each client is known in advance to the salesman. The cost of travelling to the clients are connected to the distance needs to be passed. In general, the salesman is searching for the shortest tour. In the past different cases of the TSP were introduced, for example the asymmetric TSP and symmetric TSP or that not all clients have a link between each other. In this work the symmetric case is considered, where travelling from client i to client j implies the same costs as travelling reverse. These assumptions give us $(n-1)!/2$ feasible solution of a TSP with n vertices in a tour. Furthermore, it is assumed that all travelling costs are non-negative.

Given a complete graph $\hat{G} = (V', E)$ with the set of vertices $V' = \{v_1, v_2, ..., v_m\}$ and $E = \{[v_i, v_j] | v_i, v_j \in V', i < j\}$ as the set of edges. The set of vertices contains all stops on the closed tour of the salesman and his home. Each edge $[v_i, v_j]$ has the cost c_{ij} of travelling from vertex v_i to v_j. Note that for the TSP only the SP's are considered. The

mathematical formulation of the TSP is:

$$\sum_{[v_i,v_j]\in E} c_{ij}x_{ij} \quad \rightarrow \min! \tag{2.3}$$

$$\text{s.t.} \quad \sum_{\substack{v_i\in V' \\ i\neq j}} x_{ij} = 1 \quad \forall\, v_j \in V' \tag{I}$$

$$\sum_{\substack{v_j\in V' \\ i\neq j}} x_{ij} = 1 \quad \forall\, v_i \in V' \tag{II}$$

$$\sum_{\substack{v_i\in S \\ v_j\in V'\setminus S}} x_{ij} \geq 2 \quad \forall\, S \subset V' \tag{III}$$

$$x_{ij} \in \{0,1\} \quad \forall\, v_i, v_j \in V' \tag{IV}$$

Equation (2.3) is the objective function, where the decision variable x_{ij} is equal to 1 if and only if the salesman is visiting v_j after visiting v_i. In other words, if the edge $[v_i, v_j]$ will be passed on the tour. Otherwise x_{ij} is equal to zero. The constraints (I) and (II) ensure that each city is just visited once. The additional constraints (III) are preventing any subtours and are called the subtour elimination constraints. These constraints determine that any two disjoint partitions of the set V' should be connected by at least two edges.

The TSP is known to be a NP-hard combinatorial problem.

As one can see the first objective of the basic model of RSP in section 2.1 is equal to the objective of the TSP. Since the basic model of RSP fulfils the assumptions of the TSP, as like completeness of the graph, symmetry of the edges and non-negativity of all c_{ij}, we can use it to find a ring connecting all SP's on it.

2.3. User Equilibrium

To describe a service orientated assignment of the vertices to the SP's, the User Equilibrium will be introduced. The User Equilibrium is used, with its assumptions on the behaviour of travellers, to describe the flows of people and goods on links in a network under given traffic conditions. So, the assignment problem of the RSP needs to be transmitted to a system of a flow network. To do so the works of Džubur (2013) [10], under consideration of the basic model of the Wardrop Equilibrium (WE) described by Correa and Stier-Moses (2010) [6] were taken into account. In her Master's Thesis Džubur [10] optimized a Warehouse Location Problem applying the concept of the User Equilibrium. Further on in this section the second objective with a service quality function is introduced and explained.

The User Equilibrium with its assumption is adapted mainly in the research field of transportation and telecommunication networks, where the flows within the networks are optimized. Since Wardrop was the first who formulated its principles in his work from 1952 [27], the User Equilibrium is also referred to as Wardrop Equilibrium. The WE is a

steady state which is obtained after an adjustment phase of the travellers behaviour till no one can improve his own objective [6]. One objective can be for example to minimize the cost of travelling a route in terms of distance or time needed to traverse it. Each route is described by an origin-destination (OD) pair. Given the traffic conditions each traveller is optimizing non-cooperatively his own objective and chooses a route which appears to him as most efficient [6]. This non-cooperative behaviour is not necessarily optimizing the overall performance. Following this Wardrop formulated the following principles:

Definition 1 (Wardrop's Principles)**.** *1. The journey times on all routes actually used are equal or less than those which would be experienced by a single vehicle on any unused route.*
2. At equilibrium the average journey time is minimum.

The first principle refers to the selfish behaviour of all participants and describes the User Equilibrium. It says that no one can improve his own travelling cost in the WE by changing his behaviour while all other participants keep theirs. The Wardrop Equilibrium refers just to the first principle of Wardrop. The second principle is added here for the sake of completeness and describes a Social Equilibrium, where the travellers cooperatively optimize the total travel costs [6].

To obtain a WE in order to optimize the second objective of the RSP introduced in section 2.1, this will be formulated as a network system. The complete graph $\tilde{G} = (V \cup \{t\}, \tilde{A})$ with a dummy vertex t is given with the set of arcs $\tilde{A} = \{(v_i, v_j), (v_i, t) | \, v_i, v_j \in V\}$. $\tilde{A}$ contains all possible links between the vertices, as given by A in section 2.1, and additionally links to the vertex t. Note that for each arc $(v_i, v_j)_{\forall v_i, v_j \in V} \in A$ the cost of transfer this, are given by d_{ij}. The set of commodities $C = \{(v_k, t) \, | \, v_k \in V\}$ is represented by disjoint OD-pairs, where v_k is the origin and t the destination. Hence the elements of C are just distinguished by the origin, the OD-pairs are indicated just by the origin, as $k \in C$. The demand of vertex v_k is defined by the number of inhabitants p_k within it. Given the set of the vertices on the ring $V' \subseteq V$ the aim is to find an efficient route for the demand p_k from its origin to a vertex on the ring and then to the destination. In other words, all routes are going from a vertex $v_k \in V$ passing one of the vertices $v_j \in V'$ to t. The purpose of introducing a dummy variable t is to extend the graph by arcs to which the total incoming demand in each node on the ring can be assigned [10]. Figure 2.1 shows an example of such a network with all possible routes.

For each OD-pair k the set of routes is given by $R_k = \{(v_k, v_j, t) | \, v_k \in V, v_j \in V'\}$ and the set of all possible routes is given by the union $R = \bigcup_{k \in C} R_k$. Through the pair (v_k, v_j) each route $r \in R$ is uniquely defined and two kinds of link flows can be determined. Firstly, the link flow between a vertex $v_k \in V$ and a vertex $v_j \in V'$ is given by $f_{(v_k, v_j)} = f_{kj}$. This flow represents the number of people who live in vertex v_k and decided to travel to v_j. It is assumed, that the demand p_k is arbitrarily divisible, because the decision on a route of a single participant has an insignificant small impact on other participants [6]. Secondly, the link flow from v_j to t is defined by $f_{(v_j, t)} = f_j$, which represents the demand occurring within the node v_j on the ring. Consequently, it is the

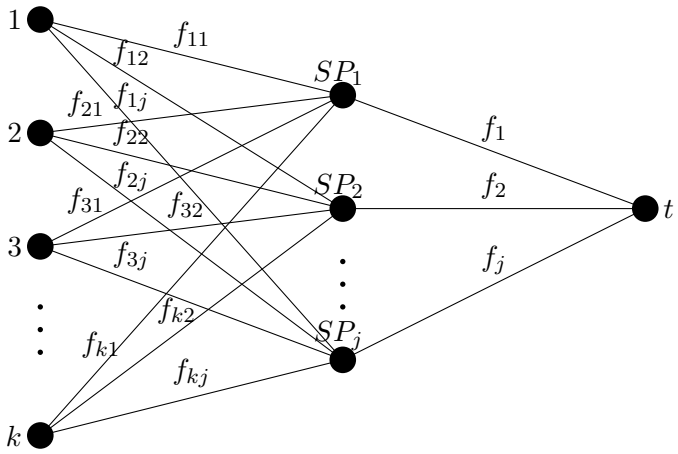

Figure 2.1.: Illustration of flows (cf.[10])

total number of incoming people to v_j from different vertices. The non-negative vector of link flows $f = ((f_{kj}), (f_j))_{v_k \in V, v_j \in V'}$ contains all link flows in the network.

Since each route r is uniquely and well-defined, each element of non-negative vector of route flows $h = (h_{kj})_{v_k \in V, v_j \in V'}$ provides the unique link flow f_{kj} and it holds $h_{kj} = f_{kj}$ [10]. As a consequence the demand constraints are given by $\sum_{v_j \in V'} h_{kj} = p_k \; \forall \; v_k \in V$. Hence the flow conversation constraints need to hold for every $v_j \in V'$, which say that the incoming flow to v_j needs to be equal to the outcoming flow, the flow f_j is determined by $f_j = \sum_{k \in C} h_{kj} = \sum_{k \in C} f_{kj}$. Because of the bijective relation of link flows and route flows, everything can be expressed in terms of link flow [10]. The feasible set of flows (f, h) is given by X. Its projection into the link flow space X_f is defined by the flow conversation constraints and demand constraints. It follows X_f is a polytope.

Each participant within the network wants to optimize two quantities, the distance to the ring and the service quality at a SP. These will be summarized in a weighted sum of two cost function. By minimizing the distance between $v_k \in V$ and $v_j \in V'$ the participant are choosing the closest vertex from the set V'. Note that by d_{kj}, as defined above, costs related to the length of an arc connecting two vertices are given. By defining a link travel cost function $t_{kj}(f_{kj}) = \phi(d_{kj})$ the flow vector $(f_{kj})_{v_k \in V, v_j \in V'}$ is mapped to its costs on each arc connecting the vertices with the ring. Hence there are no further restriction on the link capacities, nor the travelling distance and d_{kj} is independent from a flow on the corresponding link, ϕ is a constant function regarding to flow. It follows that ϕ is non-negative, since costs are positive, in addition to it ϕ is also continuous and non-decreasing. To have a linear function, ϕ will be defined as a constant value $\alpha > 0$ [10],

$$\phi(d_{kj}) = \alpha \, d_{kj}. \tag{2.4}$$

The service quality at a SP is dependent on the amount of people assigned to it. It is diminishing with every additional person going to the same SP. As a consequence, the

service quality at a SP is higher, if less people are assigned to this. In order to minimize the overall objective of a person the function of service quality need to be expressed as a cost function. To do so the costs of the diminished service quality by every additional person at a SP are proposed. A function which is growing with each additional person coming to the SP, needs to represent this sacrifice. Hence no further restriction at a SP are made like a capacity, this function needs to be non-decreasing, non-negative and continuous on $\mathbb{R}_{\geq 0} \cup \{\infty\}$ with respect to the amount of incoming people to a SP, described by f_j. The function $\omega(f_j)$ in (2.5) fulfils the stated properties for the sacrifice in service quality.

$$\omega(f_j) = (f_j)^2 \tag{2.5}$$

Let $\omega(f_j)$ be the function of loss in service quality conditional on f_j and $\beta > 0$ a constant price, then the cost function is: $t_j(f_j) = \beta\,\omega(f_j)$.

The total cost of the flow f on the route $k \in C$ are given by

$$c_{kj}(f) = \phi(d_{kj}) + \beta\,\omega(f_j). \tag{2.6}$$

This cost function corresponds to the one each person is optimizing independently.

To get the Wardrop Equilibrium the total cost function needs to fulfil the conditions of non-negativity, continuity and it needs to be non-decreasing. Hence the both partial costs function are fulfilling these properties, consequently they hold also for the sum of them. Since all routes in R_k for each $k \in C$ are uniquely defined by the pair (k, j), the flow f is Wardrop Equilibrium if and only if

$$c_{kj}(f) = \min_{j'} c_{kj'}(f) \tag{2.7}$$

holds for all (k, j) with a flow f_{kj} bigger than zero ([10], [6]). If there are two routes (k, j, t) and (k, j', t) with a flow bigger than zero, the saving on the travelling costs by choosing the shorter path compensates for the loss in service quality.

The Wardrop Equilibrium can be computed by optimizing the minimum-cost multi-commodity flow problem with separable objective function as like in (2.8).

$$\min_f \Big\{ \sum_{k,j} \int_0^{f_{kj}} \phi(d_{kj})dz + \int_0^{f_j} \beta\,\omega(z)dz \mid f \in X_f \Big\}. \tag{2.8}$$

Beckmann et al. [4] showed such a minimum exists. Hence the cost function $t_{kj}(f_{kj})$, $t_j(f_j)$ and $c_{kj}(f)$ are non-negative, non-decreasing and continuous and the feasible set X_f is a polytope, it follows a convex minimization problem needs to be solved.

By calculating the integrals, we get the following minimization problem:

$$\min_f \Big\{ \alpha \sum_{v_k \in V} \sum_{v_j \in V'} d_{kj} f_{kj} + \beta \sum_{v_j \in V'} \frac{1}{3} (f_j)^3 \Big\} \tag{2.9}$$

$$\text{s.t.} \quad \sum_{v_j \in V'} f_{kj} = p_k \quad \forall\, v_k \in V \tag{I}$$

$$\sum_{v_k \in V} f_{kj} = f_j \quad \forall\, v_j \in V' \tag{II}$$

The demand constraints are stated in (I) and the flow conversation constraints are given by (II).

Since the service quality was reformulated as a cost function of sacrifice in service quality is it reasonable to sum up the travelling costs to a SP with it. Both costs are funded by the inhabitants of the population centres, differently to the ring costs. Additionally, the two cost in the above introduced assignment problem are not conflicting each other and have no influence on each other.

2.4. RSP with User Equilibrium Constraints

In this section all the constraints and assumptions will be brought together to give an overview of the problem, which will be optimized in this Master's Thesis.

$$\min \Big(\sum_{[v_i, v_j] \in E} c_{ij} x_{ij},\ \alpha \sum_{v_i \in V} \sum_{v_j \in V'} d_{ij} f_{ij} + \beta \sum_{v_j \in V'} \frac{1}{3} (f_j)^3 \Big) \tag{2.10}$$

$$\text{s.t.} \quad \sum_{\substack{v_i \in V' \\ i \neq j}} x_{ij} = 1 \quad \forall\, v_j \in V' \tag{I}$$

$$\sum_{\substack{v_j \in V' \\ i \neq j}} x_{ij} = 1 \quad \forall\, v_i \in V' \tag{II}$$

$$\sum_{\substack{v_i \in S \\ v_j \in V' \setminus S}} x_{ij} \geq 2 \quad \forall\, S \subset V' \tag{III}$$

$$\sum_{v_j \in V'} f_{ij} = p_i \quad \forall\, v_i \in V \tag{IV}$$

$$\sum_{v_i \in V} f_{ij} = f_j \quad \forall\, v_j \in V' \tag{V}$$

$$x_{ij} \in \{0, 1\} \quad \forall\, v_i, v_j \in V' \tag{VI}$$

The two objectives of (2.10) will be solved simultaneously and independent from each other after the SP's are chosen. The constraints (I)-(III) and (VI) are similar to the

constraint of a TSP. (I)-(II) ensures that each vertex on the ring is exactly once on a tour along the ring. Since it is desirable to have one tour, (III) gives the subtour elimination constraints. The decision variables are binary (VI), where x_{ij} is equal to 1 if and only if the edge $[v_i, v_j]$ is within the cycle, otherwise 0. The demand constraints for every vertex are given by (IV) and the flow conversation constraints, which are required for solving the WE, are stated by (V).

As a consequence of introducing the User Equilibrium and its constraints to the bi-objective RSP in the special case of $V' = V$ still a bi-objective optimization problem is required to be solved. Since there are cases, in which inhabitants of populous SP's are willing to travel to another SP's in return of a gain in service quality. This behaviour is strongly related on the choice of β.

3. Multiobjective Optimization and Genetic Algorithms

The intention of the following chapter is to give an overview on multi-objective optimization. Further on the ideas of genetic algorithms are described for a better understanding of the remainder of this work.

Multiobjective Optimization

Most real-world problems cannot be solved efficiently by optimizing a single objective function. Often, there are several different objectives, which need to be optimized simultaneously. Often they also tend to be conflicting with each other, in a way that by optimizing one of them the other are influenced to perform worse. Since a single objective is easier to handle and one optimum exists, many of the multi-objective optimization problems (MOOP) are solved as a single objective problem. One method to transfer a MOOP into a single optimization problem is to sum up all objectives and optimize the value of that sum. To reflect priorities of a decision maker it is possible to multiply each objective with a different weight. Since often no further information are given about the complex solution environment prior, it is hard to mirror the preferences of a decision maker. Another approach is to consider only one of all objectives and constrain it by the other objectives. In this case an upper or lower bound is set to each of the remaining objectives. A solution of such approaches is mostly not an adequate one, hence it is not able to represent the characteristics of the general optimization problem. The result of a simultaneous optimization of all objectives is seldom a single optimum, it is rather a set of solutions which can be arranged into different ranks in terms of domination. Without loss of generality we can assume that all objectives are minimized [21], further on domination, also called pareto efficiency, of a decision vector x is defined as in Definition 2.

Definition 2 (Domination)**.** *Given n objective functions and m decision variables and a MOOP defined as follows:*

$$\begin{aligned} \min \quad & y = f(x) = (f_1(x), f_2(x), ..., f_n(x)) \\ s.t. \quad & x \in X \end{aligned}$$

where the $x = (x_1, x_2, ..., x_m)$ is the set of decision variables and X describes the set of feasible solutions in the decision space. The objective space is described by the set

Y, which contains all $y = (y_1, y_2, ..., y_n)$. A decision vector $x \in X$ dominates another decision vector $x' \in X$ ($x \succ x'$) if and only if

$$\forall i \in 1, 2, ..., n : f_i(x') \geq f_i(x) \quad \wedge$$
$$\exists j \in 1, 2, ..., n : f_j(x') > f_j(x).$$

Within each resulting non-dominated rank all solutions are equally efficient. Consequently, one set of solutions which are non-dominated by any other solution exists. Such set is called pareto efficient, pareto optimal or pareto set and is defined as in Definition 3. The image of the pareto set is called the pareto front.

Definition 3 (Pareto set)**.** *Given the objective set Y, the non-dominated set of solutions $Y' \subset Y$ are those solutions, which are not dominated by any element in $Y \setminus Y'$.*

Often it is not possible to find all elements of the pareto front, due to the complexity and range of the problem [21]. Therefore, a good solution method of solving a MOOP should provide a good approximation of the pareto front. More precisely this approximation should have minimum distance to the real pareto front and less cluster of elements along it. Moreover, it is preferable to have uniform spread along the generated pareto front and as many elements as possible.

Genetic algorithms perform highly in the search of multiple solutions, since they work with several solutions in one single simulation run. In the following section this method is discussed deeper, to understand the underlying mechanism.

Genetic Algorithms

A Genetic Algorithm (GA) is a class of optimization methods from the wide field of Evolutionary Algorithms [2]. The principles of the evolutionary theory are mimicked to optimize a solution of a problem.

The operations which are inspired by the evolutionary theory are applied to a population of solutions, the so-called chromosomes. A chromosome is one possible encoding of a solution composed by genes. Each decision variable is represented by one gene and the value of it is called allele. These values are problem specific and can be for example binary. As in the biological theory each chromosome in the population get a fitness value assigned. By intuition it is the objective value of the solution. For a MOOP the fitness can also be represented as the non-domination rank. The fitness value is required to compare chromosomes with each other. The goal of an optimization method is to raise the overall fitness of a population, preferable with each iteration. In other words, with each generation. A generation is a state of a population at a specific iteration throughout the algorithm. Therefore, some operations are required to transfer and optimize a population from one generation to another. To do so some parent chromosomes are chosen for reproduction of offspring chromosomes. The general procedure of a GA with the most common methods used is shortly summarized in the following recital. Please note that beyond these there are also many other variations of each steps.

(i) **Initialization of the population**
An initial population is generated, this is most commonly done randomly. The size of it needs to be set in advance and is equal to the population size for every generation. Every member of the population gets a fitness value assigned in terms of its objective values and the corresponding non- domination rank.

(ii) **Selection for reproduction**
The parents for reproduction are chosen randomly, whereby each chromosome has the same probability to be selected. Another possible method is the fitness proportionate selection. In this method the chance to be chosen is related to the fitness of a chromosome. To put it another way, the *"better"* solution in terms of fitness values is more likely to be select for reproduction. Joining the selection, a tournament between the selected members of the population is taken. Usually a binary tournament is implemented. Two members are compared by their fitness and the *"winner"* is selected for recombination.

(iii) **Recombination**
The parent chromosomes are recombined into offspring chromosomes. A crossover operator is used for the recombination, which splits the parent chromosomes at least in one position and recombines them.

(iv) **Mutation**
A mutation operator is modifying a solution usually at one random selected gene and operates with a low probability on a chromosome. Using the mutation possibility diversity is added to the population. This makes it possible to explore deeper the solution space. The mutation rate needs to be set in advance. Since, it depends on the problem [2].

(v) **Replacement**
In the replacement phase the next generation is chosen out of the set containing the current population and the generated offspring. One possibility to do so is to replace the previous generation totally by the offspring. This approach has the disadvantage of a high risk of losing *"good"* existing solutions. A good replacement operator should be preventive of losing non-dominated chromosomes and provide elitism.

(vi) **Repeat (ii)-(vi) till a termination condition is met** The most popular termination condition is a maximum number of generations.

4. Literature Review

In present literature the Ring Star Problem is solved in different formulations and as well with different solution methods. Labbé et al. (1999) [19] introduced the Median Cycle Problem (MCP) in two different versions and solved them with the branch-and-cut algorithm. In general, the MCP is defined on a complete mixed graph with given non-negative routing costs and non-negative assignment costs on each edge respectively on each arc. Like in RSP the aim is to find a cycle through a subset of the given vertices with minimum routing costs, which are determined by the edges on the cycle and with minimum assignment cost. The assignment costs are determined by choosing the arc with minimum costs connecting a vertex outside the cycle with one on it. The two versions of the MCP presented by the authors are distinguished mainly in their objectives. The first variant is minimizing a weighted sum of assignment cost and routing cost and in further literature often also referred as RSP [20]. The second variant is minimizing the routing cost under the constraint of from above bounded assignment costs. This version is mainly called MCP [20]. Although there are two conflicting cost, the MCP in the work named above was solved as single objective problem. The two variants were also solved by Moreno Pérez et al (2003) [22] with a variable neigbourhood tabu search and by Renaud et al. (2004) [24] using a multi-start greedy add heuristic and the random keys evolutionary algorithm.

In 2004 Labbé et al. [18] published a polyhedral analysis of the Ring Star Problem as a single objective optimization problem, with the intention to proof that the RSP can be solved by a convex optimization method. A weighted sum of the routing cost on the ring and the assignment costs was considered as the objective. As in [19] the authors provided a mixed-integer linear program formulation and used the branch-and-cut algorithm to solve the optimization problem.

The RSP was generally solved as a single objective optimization problem of minimizing of a weighted sum of both costs, routing and assignment costs. So did Dias et al. (2006) [8] with a hybrid metaheuristic. A General Variable Neighbourhood Search (GVNS) was used to improve the Greedy Randomized Adaptive Search Procedure (GRASP) and the results were compared to the results obtained by the proposed solution approach of Pérez et al.[22].

A bi-objective formulation of the RSP was proposed by Liefooghe et al. (2008) [20] and (2010) [21]. The same costs were considered as in the works of Labbé et al. [19], [18] and [17], namely the ring costs, which are defined by the edges on the ring and assignment costs defined by the shortest arc between a vertex on the ring and one not on the ring. Unlike single objective optimization the authors optimized two objectives separately. The search for an approximation of the pareto front was done with four metaheuristics, particularly IBMOLS (indicator-based multi-objective local search), IBEA (indicator-

based evolutionary algorithm), NSGA-II (non-dominated sorting genetic algorithm II) and SEEA (simple elitist evolutionary algorithm). The last three are evolutionary algorithm and the first one is a local search algorithm. The performance of these solution methods was measured by their hypervolume and the additive ϵ-indicator. The two best methods in terms of the measurement methods were chosen to build a hybrid metaheuristic. The hybrid version combined the SEEA, with a good performance in diversification, with IBMOLS, which is good at intensification.

A real-world application of the Ring Star Problem was introduced by Dörner et al. (2007) [9]. The authors optimized a tour planning problem for mobile healthcare facilities for a region in Senegal. Three objectives were considered for this multi-objective optimization problem with one mobile facility. More precisely, the minimization objectives presented were: (1) effectiveness of workforce employment, which is connected to the tour length of the ring, (2) average accessibility, this is related to the distance which every inhabitant needs to travel to the next stop of the mobile facility and (3) the coverage. Hence the inhabitants of the settlements (within a vertex) are not able or willing to walk every distance, the intention is to minimize the uncovered demand. This optimization problem was solved with an ant colony optimization metaheuristic and two multi-objective genetic algorithms.

5. Solution Method

The genetic metaheuristic NSGA-II (non-dominated sorting genetic algorithm II) will be applied to approximate the pareto front of the bi-objective RSP with User-Equilibrium constraint. Section 5.1 will outline the main functions of the NSGA-II and its implementation. Hence a GA, in this case the NSGA-II, is minimizing the fitness value in terms of the non-domination rank (cf. [7]) further solutions methods are needed to solve the partial problems TSP and User Equilibrium. The Clarke&Wright Savings Algorithm will be used to find a feasible tour on the ring and calculate its costs for each chromosome in a population. The assignment of the inhabitants within each settlement under the conditions of the User Equilibrium is done by the Frank-Wolfe Algorithm. These two heuristic methods are described in section 5.2.

5.1. NSGA-II

5.1.1. General functions of the NSGA-II

The NSGA-II was introduced by Deb et al. (2002) [7]. The characteristic parts of it are the fast non-dominated sorting approach and the diversity preservation by introducing the crowded-comparison operator using a crowding distance. Based on them an improved replacement procedure of the current generation in the main loop is proposed. The first NSGA (1994) [25] was outperformed by NSGA-II in terms of diversity of the found solution set, elitism and computational complexity [7]. Before the main loop will be introduced, the non-dominated sorting and the crowded comparison operator are described in the following paragraphs.

The aim of the non-dominated sorting operator shown in Algorithm 1 is to assign a rank to each chromosome in a population $\mathcal{P}$. This rank corresponds to the front the chromosome is a member of and represents its fitness. The authors of [7] introduced two entities to make the sorting faster, these are a domination count n_p and a set of solutions $\mathcal{S}_p$. The domination count n_p of a chromosome p gives the number of chromosomes by which p is dominated. Whereby the set $\mathcal{S}_p$ contains all chromosome which are dominated by p. Within a population $\mathcal{P}$ every chromosome needs to be compared to the others regarding dominance. Algorithm 1 line 5 to 9 show how the domination count n_p and the set $\mathcal{S}_p$ are evaluated. If a solution is not dominated by any other solution, it consequently belongs to the first front and gets the rank 1 assigned. To identify the next front with its members the set $\mathcal{S}_p$ of every element of the first front is taken under consideration. For all the solution q in the set $\mathcal{S}_p$ of $p_{rank} = 1$ the domination count n_p will be reduced by one [7]. If n_q is equal to zero now, then it gets the rank 2, $q_{rank} = 2$. This procedure is

repeated until all elements in the population get a rank assigned, respectively all fronts are identified.

Algorithm 1 Fast-Non-Dominated-Sorting [7]

```
1:  function FAST-NON-DOMINATED-SORT(P)
2:      for each p ∈ P do
3:          S_p = ∅
4:          n_p = 0
5:          for each q ∈ P do
6:              if p ≺ q then
7:                  S_p = S_p ∪ {q}
8:              else if q ≺ p then
9:                  n_p = n_p + 1
10:             if n_p = 0 then
11:                 p_rank = 1
12:                 F_1 = F_1 ∪ {p}
13:     i = 1
14:     while F ≠ ∅ do
15:         Q = ∅
16:         for each p ∈ F_i do
17:             for each q ∈ S_p do
18:                 n_q = n_q − 1
19:                 if n_q = 0 then
20:                     q_rankd = i + 1
21:                     Q = Q ∪ {q}
22:         i = i + 1
23:         F_i = Q
```

Another attribute additional to the rank needs was introduced, which makes it possible to compare chromosome with the same rank. This should also preserve a good spread of solutions, since diversity is a goal of a good result of a GA [7]. Therefore, the crowding distance between chromosome within a front is defined. This will provide more information of the density around a particular solution. For each solution the average distance between its next top and bottom neighbour along each objective is determined. This gives an estimation of the perimeter of a cuboid formed by the nearest neighbours of a particular solution [7]. Figure 5.1 illustrates the idea of the crowding distance with two objectives. Note that the points are members of the same front.

The crowding distance for every member of a front $\mathcal{I}$ is calculated according to the Algorithm 2, which was proposed by [7]. After the initialization of the distance for each $i \in \mathcal{I}$, the solutions within a front are sorted regarding each objective. The chromosomes with boundary values get an infinite value assigned as their crowding distance. For all other solutions the distance is calculated as set in Algorithm 2 line 9. Note that a solution with a lower crowding distance has a higher density and therefore is more

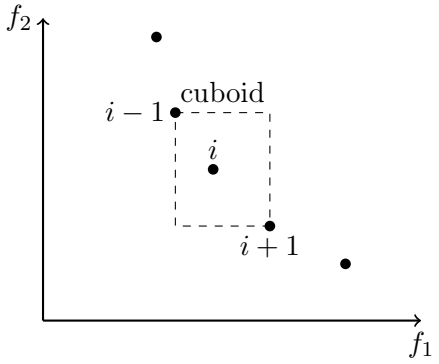

Figure 5.1.: crowding distance calculation with two objectives (cf. [7])

crowded. Consequently, chromosome within a front with a higher crowding distance are preferred.

Algorithm 2 Assignment of the crowding distance [7]

```
1: function CROWDING-DISTANCE-ASSIGNMENT(I)
2:     l = |I|                                   ▷ number of solutions in I
3:     for each i ∈ I do
4:         I[i]_distance = 0
5:     for each objective m do
6:         I = sort(I, m)
7:         I[1] = I[l] = ∞
8:         for i = 2 to (l − 1) do
9:             I[i]_distance = I[i]_distance + (I[i + 1].m − I[i − 1].m)/(f_m^max − f_m^min)
```

Using the crowding distance and the rank of a chromosome the authors of [7] defined the crowded-comparison operator ($\prec_n$).

Definition 4 (Crowded-Comparison Operator $\prec_n$)**.** *Given for each $i \in P$ a non-dominated rank i_{rank} and a crowding distance $i_{distance}$, a partial order $\prec_n$ is defined as*

$$i \prec_n j :\Leftrightarrow$$

$$\textit{if } (i_{rank} < j_{rank})$$

$$\textit{or } (i_{rank} = j_{rank}) \textit{ and } (i_{distance} > j_{distance}).$$

Since the less crowded solution are more likely to be chosen the population is pushed with every generation to an uniform spread along the approximated pareto front [7].

The procedure of the NSGA-II in Figure 5.2 describes main loop in the t-th generation. It starts with a fusion of the current population P_t and the generated offspring Q_t. Each has the same size N. The resulting population of size $2N$ is sorted into non-domination

fronts. The next generation is filled first with the members of the best non-domination fronts. As long as the number of solution has not reached N all elements of a front are added to it, beginning with the first. When the size of a front exceeds the number of free places in the next generation P_{t+1}, the solutions within this front with the highest crowding distance are chosen to be in it.

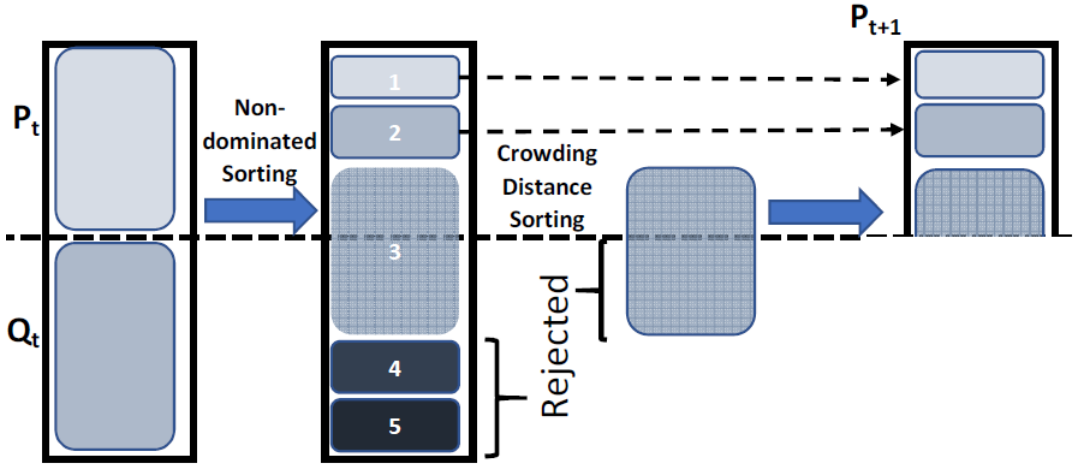

Figure 5.2.: Main NSGA-II procedure (cf. [7])

By combining the offspring with the current population and sort them according to the non-domination the elitism is retained [7], since the efficient solutions are kept.

All other steps belonging to a GA, like initialization of the population, crossover and mutation are free to be selected according to the optimization problem and coding and will be explained in the following section.

5.1.2. Implementation of the NSGA-II

To gain a good result to the optimization problem stated in chapter 2 the choice of the encoding, crossover operator and the mutation operator need to be done carefully. Since they influence the approximation to the pareto front significantly. During the implementation phase different methods were used and analysed and the following remarks are describing the ones which were used in the final version of the NSGA-II in this thesis.

A chromosome which describes a solution of the optimization problem defined in section 2.4 contains $n = |V|$ genes. Each gene is representing a vertex $v_i \in V$. An intuitive encoding of a chromosome is the usage of binary variables. A gene at position i take the value 1 if and only if a SP is opened at v_i, else it has the value 0. Note that each chromosome contains just the information about which vertex is on the ring. One member of a population in generation t is defined by its chromosome, its costs, number of the front, which it belongs to and the crowding distance. These attributes are used to describe the fitness of a solution.

To initialize the algorithm a population of solutions and their fitness is needed. Therefore, an initial population of a in advanced set population size N is created. This is done randomly, where a uniform distribution is used to evaluate a value for each gene in each chromosome. Furthermore, the corresponding ring costs and assignment costs are determined with the savings algorithm and Frank-Wolfe algorithm. Based on these costs the non-dominated sorting is identifying the fronts within the starting population.

In each generation a reproduction step is performed, where N new solutions are generated out of the existing population, the so-called offspring. Four members of the population are chosen randomly for a binary tournament. In which always two of the four members are compared to each other by using the crowded-comparison operator. Afterwards an one-point crossover operator is performed on the resulting two solution to created two new solutions. One splitting point is randomly selected, at which the two chromosomes are cut and put crossed together. In figure 5.3 this procedure is illustrated. The probability of all possible split points is uniformly distributed. The possible crossover points were limited to be between after the first gene and before the last one. An advantage of this crossover operator is it easy to implement and nevertheless it is providing diversity. Since it can change the solution in a way that a different cycle within the vertices is created. This new cycle differ in size and members of it. Additionally to secure diversity with every crossover it was checked that not two identical solutions were selected for the crossover. Furthermore the resulting offspring was tested for being a duplicate of an existing solution (cf. A.4).

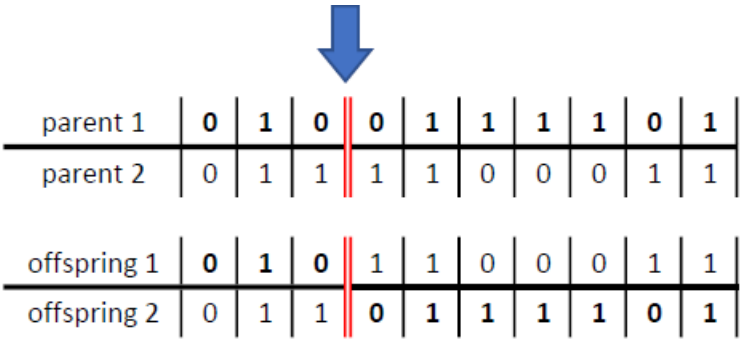

Figure 5.3.: One point crossover operator of two solutions

By applying of a mutation operator, the solution space can be explored deeper by changing randomly some of the found solutions. It is performing with a low probability on the offspring chromosomes. This probability need to be adjusted to the problem and test instances. A too low probability does not provide the desired effect of higher diversity and a too high probability will make the search too random, which leads to unstable results. A mutation is performed on each gene of the chromosome of an offspring after the crossover, with the respective mutation rate. It operates by changing the gene values to 1 if it was previously 0 and to 0 it was 1.

The main loop of the metaheuristic is implemented as proposed by Deb et al. [7] except

one modification. Throughout the test runs the algorithm produced many duplicates and this contradicts the goal of diversity. So, an additional condition was added to the loop of selecting the population in the next generation, which did not allow to accept duplicates. Since the chromosomes were encoded binary just the chromosomes of each member are compared (cf. A.4).

Set the population size N and maximal number of generations T the following NSGA-II version is adapted:

- **While** $|\mathcal{P}_0| < N$
  - create random chromosome
  - calculate the ring cost using the Clarke&Wright Savings Algorithm
  - calculate the assignment costs using the Frank-Wolfe Algorithm
- **For t = 1 : T**
  - non-dominated sorting ($\mathcal{P}_{t-1}$)
  - for each front $\mathcal{F}_i$ in $\mathcal{P}_{t-1}$ calculate for each member of it the crowding distance
  - Reproduction
    - choose 4 random members out of the current population
    - binary tournament
    - one-point crossover operator
    - mutation operator
    - calculate the ring cost using the Clarke&Wright Savings Algorithm
    - calculate the assignment costs using the Frank-Wolfe Algorithm
    - repeat *Reproduction* till $|\mathcal{O}_t| = N$
  - merge current population with offspring: $\mathcal{R}_t = \mathcal{P}_{t-1} \cup \mathcal{O}_t$
  - non-dominated sorting($\mathcal{R}_t$)
  - for each front $\mathcal{F}_i$ in $\mathcal{R}_t$ calculate for each member of it the crowding distance
  - select next generation $\mathcal{P}_t$

5.2. Solution Methods of Solving the Partial Problems of the RSP

5.2.1. Clarke&Wright Savings Algorithm for the TSP

The Savings Algorithm introduced by Clarke and Wright (1964) [5] was intending to solve Vehicle Routing Problems. Nevertheless, this heuristic was also used to build a feasible tour for the TSP ([12], [13]). The general idea of the Savings Algorithm is to make connections between nodes in a way that the saving in costs or distance are maximized compared to the previous setting. The algorithm consist of the following main steps:

1. select a vertex as a depot (starting node) and denote it with 1
2. compute savings: $s_{ij} = c_{i1} + c_{1j} - c_{ij}$
3. sort the savings in a decreasing order
4. starting at the top of the savings list and moving downwards, form lager tours by linking appropriate cities i and j
5. repeat step 4 until a complete tour is formed.

Please note that the tour build by the Saving Algorithm is not optimal.

5.2.2. Frank-Wolfe Algorithm for Solving the WE

For each chromosome the assignment problem with User Equilibrium constraints is solved by the Frank-Wolfe Algorithm. This section will give a general idea of this solution approach. The Frank-Wolfe Algorithm is implemented in the way Džubur [10] introduced it in her Master's Thesis about the optimization of a Warehouse Location Problem.

Frank and Wolfe introduced 1956 [11] a gradient and interpolation method, further known as the Frank-Wolfe Algorithm, to solve constraint non-linear programming problems. This method is also popular for solving traffic assignment problems, hence to its inexpensive memory requirements and its simplicity [1]. The Frank-Wolfe Algorithm is applied to optimize convex problems of the form

$$\{\min \omega(f) | f \in X \subset \mathbb{R}^n_{\geq 0}\},$$

which satisfy the following conditions

- $\omega(f)$ is a convex function and continuously differentiable and
- $X \subset \mathbb{R}^n_{\geq 0}$ is a compact and convex set of feasible solutions.

Given these conditions a solution for that kind of problem exists and it is unique if ω is strictly convex. The formulation of the assignment of the nodes to the ring using the concept of the WE in section 2.3 fulfils these conditions.

In each iteration a linear approximation of the objective through the current solution $(\omega(f) \cong \omega(f^{(t)}) + \bigtriangledown\omega(f^{(t)})(f - f^{(t)}))$ is minimized within in the constraint set $X \subset \mathbb{R}^n_{\geq 0}$ to find a search direction $(s^{(t)})$ and detemine a new and better feasible solution $(f^{(t+1)})$. Please note that a solution to the problem discussed in this work is given by the flows on different routes between all OD-pairs. As a consequence, in each iteration a single route for each OD-pair is considered [15]. The search direction is defining the solution used in the next iteration and is the minimizer of the linearisation:

$$\begin{aligned} s^{(t)} := arg \min_{s \in X} (\omega(f^{(t)}) + \bigtriangledown\omega(f^{(t)})(s - f^{(t)}) \\ \Leftrightarrow \quad arg \min_{s \in X} \bigtriangledown \omega(f^{(t)}) s. \end{aligned}$$

A sequence of solutions is created throughout the algorithm, which is converging against the optimum of the problem. Although the Frank-Wolfe Algorithm is converging to an equilibrium, the convergence is asymptotic. It means around the optimum it slows significantly down in term of convergence and tends to behave in *"zig zag"* manner [15]. This is due to the fact that the algorithm prefers to accept corner solution as a consequence of avoiding infeasibility [15]. To handle this behaviour a modification was introduced.

The line search approach is using a fixed step size $\xi^t \in [0,1]$ to determine the next solution. This is defined by a convex combination of the current solution and the search direction:

$$f^{(t+1)} = (1 - \xi^t) f^{(t)} + \xi^t s^{(t)}.$$

As in her work, Džubur [10] set the step size to $\xi^t = \frac{2}{t+2}$ in iteration t. With each iteration the weight is more on the previous solution than on the search direction due to the fact that $\lim_{t\to\infty} \frac{2}{t+2} = 0$. This setting was also adopted in the implementation in this work.

Applying the Frank-Wolfe Algorithm to the assignment problem of section 2.3 one needs to remind that due to the fact that everything can be expressed in terms of f_{kj} only the demand constraints are of interest for all vertices $k = 1, ..., n$. Given the $n \times m$-matrix, $(f_{kj})_{k\in V; j\in V'}$ each vector k describes the distribution of the demand p_k on the m vertices on the ring. Therefore the vector $(f_{k1}, f_{k2}, ..., f_{km})'$ is an element of the standard simplex $S_m^{p_k}$ in $\mathbb{R}^m$, which is by the factor p_k enlarged (cf. [10]).

$$S_m^{p_k} = \{(x_1, ..., x_m) \in \mathbb{R}^m | x_1 + ... + x_m = p_k, x_j \geq 0, j = 1, ..., m\}$$

It follows that the feasible set X_f is polyhedron, precisely a Cartesian product of enlarged simplices $(S_m^{p_1} \times ... \times S_m^{p_n})$ [10]. This fact leads to the conclusion that with each iteration step the search direction $s^{(t)}$ is not a point, but rather a $(n \times m)$ matrix

$$s^{(t)} := arg\min\{\sum_{k=1}^{n}\sum_{j=1}^{m} \frac{\partial\omega(f^{(t)})}{\partial f_{kj}} s_{kj} | s = (s_{kj}) \in S_m^{p_1} \times ... \times S_m^{p_n}\}.$$

This problem can be broken down into n partial problems, for each $k \in V$ [6], [10]:

$$s_k^{(t)} = arg\min\{\sum_{j=1}^{m} \frac{\partial\omega(f^{(t)})}{\partial f_{kj}} s_{kj} | s_k = (s_{k1}, ..., s_{km})' \in S_m^{p_k}\}.$$

For each problem the solution is given by a corner point of the corresponding simplex. It follows $s^{(t)} = (s_1^{(t)}, ..., s_n^{(t)}) = (p_1 e_{j^*(1)}, ..., p_n e_{j^*(n)})'$, where the index of the minimizer of the k-th problem is given by $j^*(k)$ and e_j is the j-th unit vector in $\mathbb{R}^m$ [10]. Using $f_k = \frac{1}{m}\sum_{j=1}^{m} p_k e_j = \frac{1}{m}\sum_{j=1}^{m} p_k(1, ..., 1)'$ for the flows for each simplex and $\xi^0 = 1$ as initial values, algorithm 3 describes the implemented version of the Frank-Wolfe Algorithm.

Since the algorithm is converging to the optimal solution $\omega(f^*)$ in a *zig zag manner*, it is reasonable to define an upper and a lower bound of the objective value within

Algorithm 3 Frank-Wolfe Algorithm [10]

function FRANK-WOLFE(f)
  for $t = 1 : t_{\max}$ **do**
    for $k = 1 : n$ **do**
      Find the optimal corner point $s_k = d_k e_{j^*(k)}$ whitin the simplex $S_m^{d_k}$ s.t.
      $j^*(k) = arg\ \min\{...\}$
    Set $s = (s_1, ..., s_n)$
    Set $\xi = \dfrac{2}{t+2}$
    Set $f = (1-\xi)f + \xi s$
    for $j = 1 : m$ **do**
      $\sum_{k=1}^{n} f_{kj} = f_j$

each iteration. Using the best known upper and lower bounds throughout the algorithm is providing an alternative termination criterion by considering the gap between these values. The equation below gives a relation which is used to define the lower (LBD_t) and upper (UBD_t) bound in each iteration:

$$\omega(f^{(t)}) + \bigtriangledown\omega(f^{(t)})(s^{(t)} - f^{(t)}) \underbrace{\leq}_{\text{(I)}} \omega(f^{(t)}) + \bigtriangledown\omega(f^{(t)})(f^* - f^{(t)}) \underbrace{\leq}_{\text{(II)}} \omega(f^*) \underbrace{\leq}_{\text{(III)}} \omega(f^{(t+1)}).$$

The inequality (II) and (III) follow from the convexity of ω and optimality of $\omega(f^*)$. Since $(s^{(t)} - f^{(t)})$ is the optimal search direction within iteration t which is not necessarily true for $(f^* - f^{(t)})$ in t, inequality (I) holds. It follows that $LBD_t := \omega(f^{(t)}) + \bigtriangledown\omega(f^{(t)})(s^{(t)} - f^{(t)})$ and $UBD_t := \omega(f^{(t+1)})$. The interval $[LBD_t, UBD_t]$ contains the optimal value and is decreasing, since the sequence LBD_t approximates from below to the optimal objective value and UBD_t from above. In chapter 6 the gap, more precisely the relative gap, between the best known upper (UBD) and lower (LBD) bound is used to analyse the computational results of the Frank-Wolfe Algorithm and evaluate the quality of its results. As in the work of Džubur [10] the relative gap is:

$$\frac{UBD - LBD}{LBD}.$$

6. Computational Results

All the results discussed in the following chapter were calculated by using the software "Microsoft Visual C++ 2015 Express Edition" on the same PC with 2.60 GHz and 8 GB RAM.

6.1. Test Instances

To the best knowledge of the author the bi-objective RSP was not solved with User Equilibrium constraints until the present time, therefore there are no existing instances with a known pareto front or best-known results. For that reason, test instances were generated and enumerated. Due to the computational effort of the enumeration of all possible combinations of a binary encoded solution and the connected calculation of the optimal objectives, each test instance contains 10 vertices. The vertices were randomly generated in a $[0, 100]$x$[0, 100]$ square. The Euclidean distance between two nodes was rounded to the next integer and corresponds to the costs of travelling from one vertex to another. The instances were created with different distributions among the vertices, namely uniform distribution and two different clusters. The demand within the vertices was also randomly generated within a given interval. Whereby the instances are also distinguished by a weak spread of demand, $[20, 40]$ and a strong spread $[20, 80]$. Table 6.1 shows all generated instances with their characteristics. The first letter in the name of the instance indicates which distribution of the vertices was used and the following number gives the information about the demand spread. Three instances with same characteristics of each kind were created and the last number in the name distinguishes them. Overall 18 test instances were created to test the performance of the proposed solution approach.

For the evaluation of the optimal ring costs for each possible combination of the vertices all permutation of them were enumerated. To calculate the exact and optimal solution of the second objective would go beyond the scope of this Master's Thesis. Hence, the result of Frank-Wolfe Algorithm is approximating closer to the optimum with more iterations, it was reasonable to solve this problem with the heuristic method with more iterations. The assignment costs for each solution of each instance were calculated by the Frank-Wolfe Algorithm with a termination condition of maximum of iterations. This was set to 2000 iterations, based on the results of the testing phase described in section 6.3.1. Since it appeared that some solutions achieve a significant small rate of change between two iterations after less than 2000 iterations, an additional termination condition was plugged into the algorithm. This condition applied only if the change rates between two iterations of all f_{kj} were smaller than 0.001 and additional the change rates of all f_j

were smaller than 0.01. These extra conditions were set with the intention of saving computational time.

The values of weights α and β were set based on the analysis and discussion of them in section 6.3.1: $\alpha = 1$ & $\beta = 0.25$. The computational time of the enumeration was on average for one instance 90 minutes.

Name	**Distribution**	**Demand-Interval**	**Pareto-Front size**	**Max Ring Cost**	**Max Assignment Cost**
U 40 1	uniform	[20,40]	19	283	2 161 190.7500
U 40 2	uniform	[20,40]	24	352	1 432 044.3750
U 40 3	uniform	[20,40]	32	270	2 599 781.7500
U 80 1	uniform	[20,80]	27	284	9 838 769.0000
U 80 2	uniform	[20,80]	22	247	5 639 906.0000
U 80 3	uniform	[20,80]	37	328	15 964 275.0000
C2 40 1	2 clusters	[20,40]	28	237	2 095 425.3750
C2 40 2	2 clusters	[20,40]	25	294	2 270 927.0000
C2 40 3	2 clusters	[20,40]	34	231	1 690 995.6250
C2 80 1	2 clusters	[20,80]	22	208	9 535 862.0000
C2 80 2	2 clusters	[20,80]	28	263	10 454 375.0000
C2 80 3	2 clusters	[20,80]	27	217	11 151 995.0000
C3 40 1	3 clusters	[20,40]	24	333	1 623 154.3750
C3 40 2	3 clusters	[20,40]	30	279	3 042 835.7500
C3 40 3	3 clusters	[20,40]	24	322	1 676 903.6250
C3 80 1	3 clusters	[20,80]	26	304	7 575 794.5000
C3 80 2	3 clusters	[20,80]	33	311	8 688 215.0000
C3 80 3	3 clusters	[20,80]	28	321	10 832 994.0000

Table 6.1.: Test Instances

6.2. Metric for Performance Measurement

To measure the performance of the proposed solution approach a metric needs to be introduced. This should provide information about the above-named quality goals of an approximation of the pareto front. Namely small distance to the true pareto front, a good spread along it and high number of elements.

A measurement metric which performs satisfactorily is the hypervolume metric, which was initially proposed by Zitzler and Thiele (1999) [28]. The area in the objective space which is covered by the set of non-dominated solutions is the target of it. For each member of the non-dominated set an area defined by its objective values and an in advance chosen reference point is considered. In the case of bi-objective optimization they are forming a rectangle. The hypervolume H is defined by the union of those rectangles of all members of the non-dominated set. Figure 6.1 illustrates a hypervolume in a two dimensional

space bounded by the points 'a' to 'e' and the reference point.

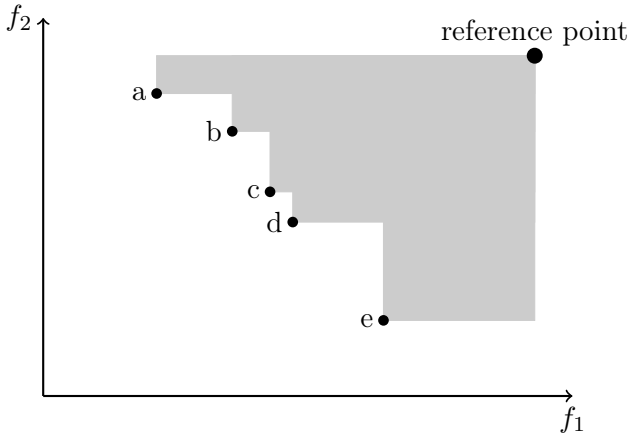

Figure 6.1.: Illustration of a hypervolume in a bi-objective solution space

Frequently the relative Hypervolume $\tilde{H} \in [0, 1]$ is used, where the fraction of coverage of the area between the zero point and reference point is measured. Therefore, a better non-dominated set has a higher $\tilde{H}$. Another approach of the Hypervolume measurement is to use the Hypervolume difference indicator I^-. This indicator is computed by taking the difference of two Hypervolumes. This approach is often used if a reference set is given in advance, such as the pareto set. A good approximation of the pareto front has a small value for the I^-. The Hypervolume difference indicator can be calculated by using values of the area calculation or the relative hypervolume values. The scaling of the volume or the objective values has no influence on the comparison analysis [16]. Therefore, the relative values of the Hypervolume will be taken under consideration further in the analysis.

The choice of the reference point should be made more thoughtfully, since it is directly affecting the size of the areas. The reference point is influencing the comparison of two different sets in terms of performance measure (cf.[16]). Since the goal of the optimization problem in this work is to get a good approximation of the pareto front, for each test instance the maximum of both objective values will give the reference point. These maximum values are taken from the enumeration results.

Note that the hypervolume metric gives only meaningful information if the true pareto front and the objective value space are convex [26]. As shown in chapter 2 is it the case here.

6.3. Parameter Settings

For the section of testing different parameter setting an instance with uniform distribution of the vertices in a $[0, 100]$x$[0, 100]$ square with a strong spread of the demand $[20, 80]$ among the vertices was created and enumerated. The tables A.1 and A.2 in the appendix

give the exact demand values for each node and the Euclidean distance between them. All results in this section are rounded to the fourth decimal place.

6.3.1. Parameters Analysis - Frank Wolfe Algorithm

To run the Frank Wolfe Algorithm three parameters need to be analysed in advance, since they influence the outcome and the approximation of the result to the equilibrium. The weights α and β in the second objective are crucial due to the fact that they describe the willingness to travel additional distance to gain more service quality in exchange. Therefore, these two weights have also a big impact on the flows of the model. The number of iterations of the Frank Wolfe Algorithm will be set as the termination criterion of the algorithm used in the NSGA-II implementation. This decision is based on the recommendation in [10]. Nevertheless, the maximal number of iterations need to be chosen carefully, since with more iterations the algorithm approaches closer to the equilibrium and simultaneously the CPU time raises. Additionally, the approximation slows down while getting closer to the equilibrium.

No further assumptions were made on capacities at the SP's nor on the maximum travelling distance to the ring in the model discussed in this work. Further on the relation between the weights will be analysed with reason to get an idea of their influence on the different kind of flows and on the approximation of the algorithm.

Three different cases are of a special interest in this analysis, namely:

- $\frac{\beta}{\alpha} = 1 \Leftrightarrow \alpha = \beta$
- $\frac{\beta}{\alpha} > 1 \Leftrightarrow \alpha < \beta$
- $0 < \frac{\beta}{\alpha} < 1 \Leftrightarrow \alpha > \beta$.

It is assumed that both weights, α and β, are bigger than zero.

The setting of $\beta = 0$ and $\alpha = 1$ is describing the general RSP, where the demand of a vertex is served by the closest SP regardless the service quality at it. The results of the assignment are equal for any $\alpha > 0$. Hence the focus of this work is to analyse a solution approach of solving the RSP with User Equilibrium constraints, this setting will be not observed further.

The opposite case is to set $\alpha = 0$ and choose β to be bigger than zero. Here the participants are willing to travel any distance to gain more service quality vice versa to reduce their costs of sacrifice in service quality. The equilibrium in this case is given by the static incoming flows $f_j = \frac{\sum_{k \in V} d_k}{|V'|} \, \forall j \in V'$. Each SP has the same incoming flow and therefore also the same service quality, which results in the same assignment costs for each participant. No one can improve his costs by changing his behaviour. From the relation $f_j = \sum_{k \in V} f_{kj}$ for all $j \in V'$ and that the distance to a SP is irrelevant, one can follow that $f_{kj} = \frac{d_k}{|V'|} \, \forall k \in V, \forall j \in V'$ are the equilibrium flows between each vertex and the available SP's. Since these results are obvious and taking just one cost into account of the assignment, this setting was also excluded from the further analysis. Please note that these results were also tested to be true.

To get a first idea of the influence of the weights on the output, the case where α is equal to β was analysed. Table 6.2 presents the results of running the Frank-Wolfe Algorithm with setting $\alpha = \beta = 1$ with different numbers of iterations. The first three columns give information about the settings of the test runs, to be exact the weight values and the number of iterations. These are followed by the relative gap between the best known UBD and LBD (relatively to the LBD). The next three columns show the change of the objective value in percent, the absolute change of the flow f_j and the absolute change of the flows between a vertex and a SP (f_{kj}). The change rate is always considered between the last two iteration steps. The CPU running time is listed in the last column and gives the total running time of the Frank Wolfe Algorithm for all possible solutions ($2^n - 1$). Since the stated setting were run for all possible solutions the first value is the average result and is followed by the maximum in brackets. The results of each test run with the other relation between α and β are presented in the appendix section A.2 in tables A.3 to A.8 and will be considered into the discussion in this section. Table 6.2 and the tables A.3 to A.8 shows that all rates decline with rising number of

α	β	# **iteration**	**(UBD-LBD)/LBD**	**change of objective value**	**change** f_j	**change in** f_{kj}	**CPU time**
1	1	50	0.0248 *(0.1)*	0.0486% *(1.3339%)*	5.8147 *(15.9525)*	0.5815 *(2.5571)*	2min 6s
		100	0.0176 *(0.0677)*	0.0093% *(0.4445%)*	2.9355 *(7.9738)*	0.2937 *(1.2488)*	4min 14s
		200	0.0151 *(0.0445)*	0.0070% *(0.1001%)*	1.4497 *(3.9461)*	0.1473 *(0.6410)*	8min 36s
		500	0.0095 *(0.0173)*	0.0023% *(0.0123%)*	0.4886 *(1.5070)*	0.0553 *(0.2709)*	20min 48s
		1000	0.0043 *(0.0081)*	0.0006% *(0.0024%)*	0.1754 *(0.7241)*	0.0232 *(0.1344)*	46min 43s
		1500	0.0024 *(0.0045)*	0.0003% *(0.0013%)*	0.0914 *(0.4828)*	0.0131 *(0.0901)*	1h 1min 44s

Table 6.2.: Test runs of Frank Wolfe algorithm with $\alpha = \beta = 1$

iterations.

It occurs that the average change rates of the objectives values are less than 1% even with few iterations. By taking a closer look at the average rates within the different setting of the weights and number of iterations it stands out that there is no clear tendency of the rates by changing the values of α and β. If α is fixed and β diminishing, the average rates of the objective values for the different iteration numbers do not change in a same manner. In table 6.3 the observations are summarized. For 50, 100 and 200 iterations the average rates are rising with a diminishing β and declining for 1000 and 1500 iterations. If the algorithm stops after 500 iterations no tendency is noticeable. For a fixed β and diminishing α the average rates of the objective values are rising if the

iterations are higher than 50.

The analysis of the influence of the different settings of the weights on the change rates of the relative gap between the best known UBD and LBD shows also that it is difficult to say how they are adjust. For a diminishing β the rates are rising for the termination criterion of 50, 100 and 200 iterations. For higher tested iteration it is falling with a declining β. Similar behaviour was observed for a diminishing α (table 6.3).

The rates of changing of the flows f_{kj} and f_j seem to react uniformly throughout the different tested iterations (cf. tables 6.3, A.3 - A.8). Note that considering the rates after 50 iterations of the Frank Wolfe Algorithm one might not detect a trend how the flow rates are influence by a declining β. For higher iterations the average rates of the flows are falling with a diminishing β. However, the rates are rising if α is diminished. These leads to the assumption that the flows adjust fast for a lower β value in the setting $\alpha > \beta$ and it takes more iterations to adjust them for a lower α, when $\alpha < \beta$.

Iterations	50	100	200	500	1000	1500	Average rate of change
$\alpha = 1 > \beta \downarrow$	$\uparrow$	$\uparrow$	$\uparrow$	$-$	$\downarrow$	$\downarrow$	objective value
$\alpha \downarrow < \beta = 1$	$-$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	
$\alpha = 1 > \beta \downarrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\downarrow$	$\downarrow$	$\downarrow$	(UBD-LBD)/LBD
$\alpha \downarrow < \beta = 1$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$-$	$\uparrow$	
$\alpha = 1 > \beta \downarrow$	$-$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	f_{kj}
$\alpha \downarrow < \beta = 1$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	
$\alpha = 1 > \beta \downarrow$	$-$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	f_j
$\alpha \downarrow < \beta = 1$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	

Table 6.3.: Changes of average rates by declining β, resp. α with different terminations of the Frank Wolfe Algorithm ($\uparrow$: rates are rising, $\downarrow$: rates are falling, $-$: no tendency)

In the further discussion a closer look will be taken onto the flows and how their approximation behaviour is influenced by different weights. For this purpose, the Frank Wolfe Algorithm was run for each solution until the change rate between two iterations is smaller than a certain value (ϵ) for all flows. Additionally it was sophisticated between two kinds of flows, f_{kj} and f_j and different setting of α and β. Table 6.4 and 6.5 present the average and maximum number of iterations needed until the termination condition was met.

Concerning the flows f_{kj}, if $\epsilon = 1$ the average number of iterations until the algorithm stops does not vary significantly. It is constant if the value of α is diminished. For a declining β the average number of iterations slightly declines. The lowest average number within the test runs settings, is reached with $\beta = 0.25$ and is 6.48% less than for $\beta = \alpha = 1$. The same observations hold for the maximum values. If ϵ is set to 0.1, bigger variations of the average number of iterations are observed by changing the weights. The lowest value is reached again with the set-up of $\beta = 0.25$ and $\alpha = 1$. The average value is reduced by 24.40% from the initial setting of $\alpha = \beta = 1$. The average number of iterations until the termination condition is met is declining with a diminishing

β ($\beta < \alpha$) and rising for a diminishing α ($\alpha < \beta$) (cf. table 6.4). By comparing the average numbers of the two threshold values, it appears that for the different weights the increase is not the same. For $\alpha = \beta = 1$ it takes about 8 times more iterations on average to fulfil the stopping condition of $\epsilon = 0.1$ than $\epsilon = 1$. For $\beta = 0.25$ it is about 6.5 times more for a threshold value of 0.1 than 1. Even though the average and maximum values of the iterations do not vary for different α with $\epsilon = 1$, the increase of them does differ if ϵ is set to 0.1. The average number of iterations is increased by about 10 times of the number for ϵ equal to 1 and $\alpha = 0.25$ by using 0.1 as threshold value. For $\alpha = 0.75$ it is 8.5 times more. It follows that the weights α and β have an influence on how fast the flows f_{kj} approximate to a stable state.

		$\epsilon = 1$			$\epsilon = 0.1$		
α	β	avg # iterations	max # iterations	CPU time	avg # iterations	max # iterations	CPU time
1	1	108	127	3min 51s	869	1068	32min 4s
1	0.75	108	127	3min 53s	815	1064	27min 11s
1	0.5	107	123	3min 43s	747	1028	24min 35s
1	0.25	101	118	3min 28s	657	959	22min 2s
0.75	1	108	127	3min 51s	924	1136	33min 9s
0.5	1	108	127	3min 51s	995	1183	37min 29s
0.25	1	108	127	3min 51s	1072	1217	40min 10s

Table 6.4.: Frank Wolfe Algorithm with stopping condition: $f_{kj} < \epsilon \; \forall \, k \in V, j \in V'$

Table 6.5 shows the results of testing the Frank Wolfe Algorithm with a termination condition using a threshold value ϵ on the absolute change rate between two iterations of the flows f_j. The first what is noticed is that it requires more iterations than using the threshold value for f_{kj} until the algorithm is determined, for both values of ϵ. As above the analysis will begin with $\epsilon = 1$. In the reference case of $\alpha = \beta = 1$ on average 434 iterations are required, and maximum 724, until the algorithm stops. Likewise, above the lowest number is reached with setting $\beta = 0.25$ with an average number of iterations of 211, which is 51.38% less than the reference case. Unlike above the number of iterations differ for different values of α. They rise with a declining α. For $\alpha = 0.25$ the average number of iterations is increased by 57.83% from the case $\alpha = 1 (= \beta)$. The same trends are observable for $\epsilon = 0.1$, the number of iterations after which the termination condition is met are falling with a declining β and rising with a declining α. The change of the threshold ϵ from 1 to 0.1 is approximately tripling the average number of iterations for all weights. The highest increase is with $\alpha = 0.25$ and $\beta = 1$ of about 3.7 times of the 685 to 2534 average number of iterations.

Since there are no restrictions on the flows or the capacities and after the analysis of the weights the decision on how to set them is made with respect to their influence on the approximation to a stable state. A β which is smaller than α is preferred. Since it takes less iteration until the changes between two steps are adequate.

		$\epsilon = 1$			$\epsilon = 0.1$		
α	β	avg # iterations	max # iterations	CPU time	avg # iterations	max # iterations	CPU time
1	1	434	724	15min 14s	1183	6039	38min 8s
1	0.75	374	680	11min 20s	1027	4878	35min 47s
1	0.5	296	679	10min 41s	857	4530	30min 38s
1	0.25	211	604	6min 32s	665	4530	21min 58s
0.75	1	499	725	16min 39s	1378	6039	44min 37s
0.5	1	593	755	20min 29s	1723	6040	55min 1s
0.25	1	685	792	30min 36s	2534	6795	1h 22min 4s

Table 6.5.: Frank Wolfe Algorithm with stopping condition:$f_j < \epsilon \; \forall \; j \in V'$

6.3.2. Parameter Analysis - NSGA-II

Mutation Rate

The influence of the mutation rate to the outcome of the proposed NSGA-II will be analysed using the same test instance as above. For evaluation of the outcome the relative Hypervolume was used. The enumeration of the instance gave the reference point at maximum ring cost of 308 and maximum assignment costs of 7 775 196. For the test runs the population size was set to 40 and the NSGA-II stopped after 20 generations. Table 6.6 presents the average results of 5 runs with each setting and the standard deviation values, whereby only the mutation rate was changed.

mutation rate	average $\tilde{H}$	standard deviation
0	0.8878	0.00785
0.1	0.8944	0.00016
0.2	0.8891	0.01059
0.5	0.8713	0.02351
0.8	0.8778	0.00732

Table 6.6.: Test runs with different mutation rates

The rates of 0.5 give lower average values of the Hypervolume with the highest deviation. Whereby a probability of 0.8 returns on average a slightly better Hypervolume with a much smaller standard deviation than a mutation rate of 0.5. A rate of 0, means that the test runs were done without any randomness provided by a mutation operator. Nevertheless, the outcome of this setting did not give the worst results, neither in the average value nor in the standard deviation. The best performance gave the setting of 0.1 mutation probability. It gave the highest average $\tilde{H}$ and the smallest deviation. These results suggest the conclusion that a rate of 0.1 gives stable results and provides still some randomness to the solution search.

Population Size and Number of Generations

The parameter population size is giving the number of solutions in each generation and how many offspring chromosomes are generated. Therefore, it has an impact on the search of a good non-dominated set in the objective space, since it defines the intensity and how widely the objective space is explored in combination with number of generations.

The enumeration results are giving a pareto set for each test instance. Given the size of these pareto fronts it is argumentative to conclude that the size of a population should not undercut these. As it is in table 6.1 shown the biggest pareto set has 37 solutions and the average size of the non-dominated sets over all test instances is about 27 members. Thus the minimum number of solutions within a population will be set to be 40.

The performance of the NSGA-II was tested with different settings of the population size, 40, 60 and 80. For each of them the genetic algorithm was run three times with 100 generations. Figures 6.2 to 6.4 show how the relative Hypervolume changed through the algorithm in three runs. The figures are distinguished by the size of the population and each set of points represents one test run. It appears that all the test runs are approximating the same maximum relative Hypervolume value, this is for all three settings the same (0.8944). This maximum is just slightly less than the relative Hypervolume which is cover by the pareto front (0.8948) of the test instance.

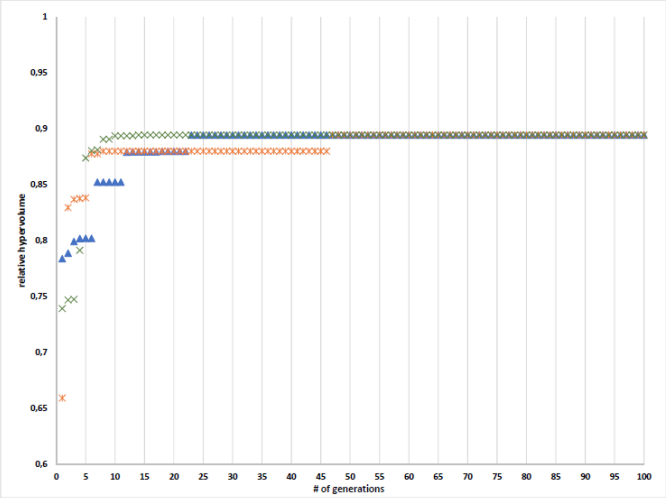

Figure 6.2.: Evaluation of $\tilde{H}$ through the NSGA-II with 100 generations and population size of 40 (results of 3 runs)

The spread of figure 6.2, with a population size of 40, is higher than the spread of the other figures with a bigger population size for low generation numbers. In one test run more than 45 generations were needed to reach the maximum value of 0.8944. Whereby with a population size of 80 solutions each of the three test runs got to it within less than 15 generations. To see the impact clearer another 5 test runs where made with this setting but with just 20 generations. For a population size of 40 the average $\tilde{H}$ was 0.8886

(standard deviation: 0.008), for 60 it was 0.8944 (standard deviation: 0.000006). For the size of each generation of 80 the average $\tilde{H}$ was 0.8944 without any variance. In other words, with a larger population less generations are needed until the best approximation of the pareto set is found.

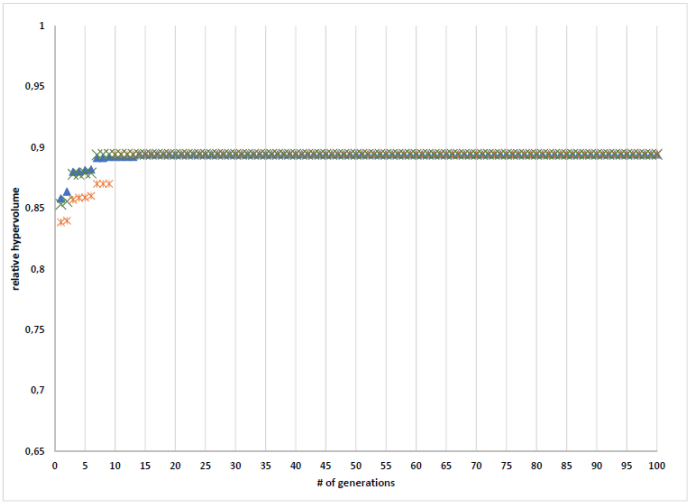

Figure 6.3.: Evaluation of $\tilde{H}$ through the NSGA-II with 100 generations and population size of 60 (results of 3 runs)

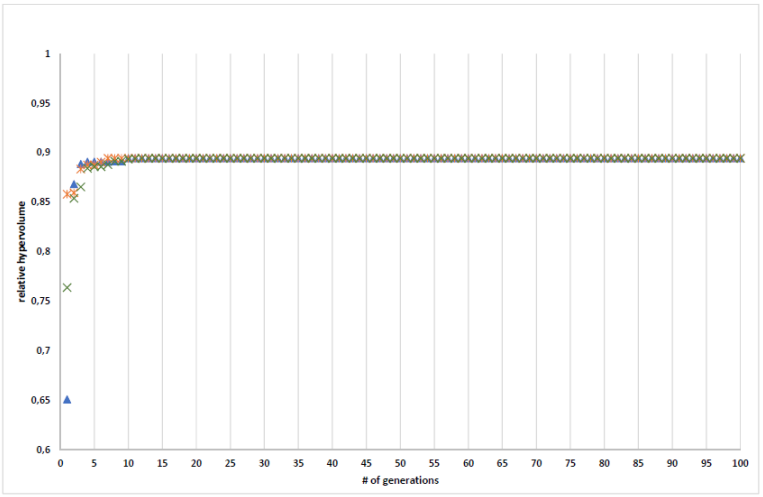

Figure 6.4.: Evaluation of $\tilde{H}$ through the NSGA-II with 100 generations and population size of 80 (results of 3 runs)

By analysing in particular the elements of the best found non-domination set and the pareto front given from the enumeration it appeared that the elements differ slightly from

each other. Because of the heuristic approach of solving the partial problems there exists an upper bound for the relative hypervolume formed by the best found non-domination set. Whereby this best-found approximation is very close to the pareto front resulting from the enumeration, as shown in figure 6.5 and resulting difference between the relative hypervolume.

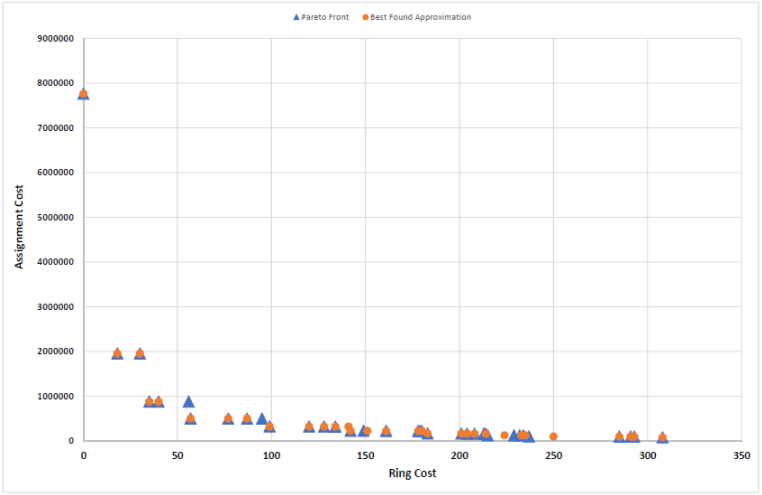

Figure 6.5.: Pareto front and the best found approximation of the test instance for the parameter setting

Moreover, through the test runs another observation was made for all the settings of population size and generation numbers. The number of fronts within a population is shrinking through the generations. To validate this for all test runs the number of fronts was tracked in each test run. Figure 6.6 shows one series of number of fronts for a test run with a population size of 80 solutions and 100 iterations. The initial population was sorted in terms of domination into 10 fronts. With each iteration the number of them is falling until it reaches the first time a minimum of 3 in the 9th generation. In further execution of the algorithm it switches between 3 and 4 fronts, whereby there are mostly just 3 fronts in the population. The enumeration results display that the 80 best solutions can be sorted into three fronts by applying the non-domination sorting. This is supported by the test results. All test runs for N = 80 the number of fronts was not smaller than 3.

For the sake of completeness, the number of fronts for a population inclusive the offspring was also tracked. Figure A.1 in the appendix shows the series of number of fronts of generation including the offspring before the selection of the next generation. From this figure it follows that there is no trend for a size of fronts for population with its offspring. Even though this number is pushed for the population to a certain minimum, the operations of reproduction and mutation create different new solutions in terms of objective functions and as well of domination.

The named observation on the progress of the fronts within a population was made also

on other test runs. In figure 6.4 is shown that for a population size of 80 the NSGA-II finds roughly after 15 iterations it best found non-dominated set. It follows that there is no clear connection between the number of fronts and the quality of the approximation of the pareto front. The number of fronts can be used as an indicator for a best found solution, but not as a quality measurement or a stopping condition for the NSGA-II.

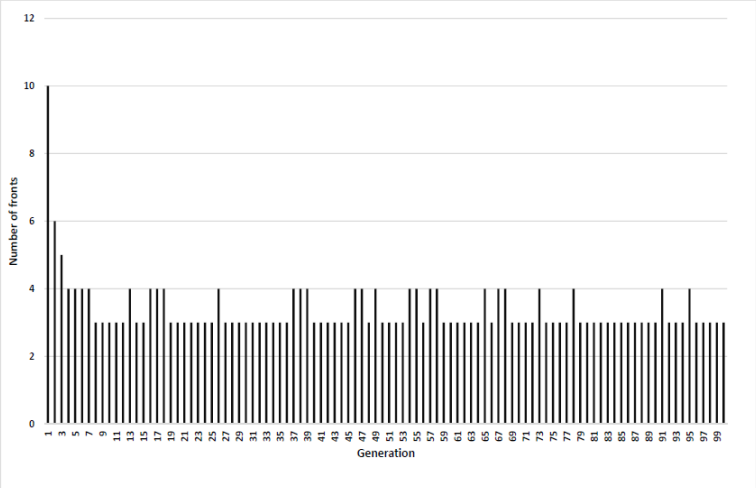

Figure 6.6.: Number of fronts within a population of size 80 in each generation

6.4. Overall Computational Results with the Test Instances

The final test runs using the test instances were done with the setting of a population size of 40 and the generation number 50. Given the results discussed above it followed that either a big population size with less generations or a smaller population with more generation need to be set to gain adequate results. The computational time is rising with both parameters and execution of the NSGA-II with a population of 80 solutions and 20 generations takes about 45 minutes and as well with the chosen setting. The mutation rate is set to be 0.10 for each gene on a chromosome. The Frank Wolfe Algorithm was executed with 700 iterations and $\alpha = 1$ and $\beta = 0.25$.

Each test instance had three independent test runs and table 6.7 presents the resulting average difference indicator of the relative Hypervolumes and the standard deviation. The last column is showing the maximal computing time of the test runs for each instance. The instances with a standard deviation of 0 had the same results in all three runs. The overall performance of the implemented solution method was good, in terms of the difference indicator $I^-_{\tilde{H}}$. The uniformly distributed instances were slightly better solved by the NSGA-II than the clustered. 4 out of 6 instances were solved without a deviation and had a difference to the Hypervolume of the pareto front of less than 0.001. The Hypervolume of these instances was less than 0.1% smaller than the Hypervolume covered by the pareto set. The results of the clustered instances are less stable and

deviate more. Nevertheless, the average difference indicator of the cluster instance is not exceeding 0.1. The biggest difference resulted from the instance "C2 80 3", which was on average 0.0136. The NSGA-II is performing with nearly the same quality on the uniform distributed instances, as on instance with clusters. The observed results do not distinguish between the different demand spread in the instances.

Instance	**Average** $I_{\tilde{H}}^{(-)}$	**Standard Deviation**	**max CPU**
U 40 1	0.0002	0	49min 16s
U 40 2	0.0007	0	43min 5s
U 40 3	0.0057	0.0095	41min 24s
U 80 1	0.0002	0	47min
U 80 2	0.0002	0	52min 15s
U 80 3	0.0093	<0.0001	59min 9s
C2 40 1	0.0003	0.0003	47min 18s
C2 40 2	0.0005	<0.0001	41min 28s
C2 40 3	0.0045	0.0037	38min 54s
C2 80 1	0.0001	<0.0001	53min 4s
C2 80 2	0.0002	0	1h 1min 59s
C2 80 3	0.0136	0.0011	46min 30s
C3 40 1	0.0003	0	47min 49s
C3 40 2	0.0002	<0.0001	38min 36s
C3 40 3	0.0004	0	52min 43s
C3 80 1	0.0053	0	45min 29s
C3 80 2	0.0001	<0.0001	40min 45s
C3 80 3	0.0033	0.0054	47min 2s

Table 6.7.: Computational Results of the Test Instances

7. Conclusion

The bi-objective Ring Star Problem was solved by the genetic algorithm NSGA-II. Whereby the assignment problem was extended by User Equilibrium constraints. This made it possible to modify the assignment of the vertices to a Service Point based on distance and additionally based on the service quality. A function, which represents the costs of sacrifice in the service quality dependent on the total served demand at a SP, was introduced. To solve this partial problem, it was formulated as a network system, for which the User Equilibrium could be solved by the Frank Wolfe Algorithm. The calculation of the ring costs was done by solving the TSP using the Clarke&Wright Savings Algorithm. The quality of the implemented NSGA-II was measured on generated test instances by using the Hypervolume metric.

During the adjustment phase of the parameter settings it got noticeable that the implemented solution approach was performing very good on these small instances. This early guess was proven to be true by the final test runs. The computation time of the approximated pareto fronts was halved compared to the enumeration time. Whereby the results on average covered just less than 1% of the Hypervolume given by the pareto set. The NSGA-II gave stable results independently of the characteristics of the test instance.

The explanation for the good performance of the solution method can be assumed to be the encoding of the solution. This allows a fast and easy comparison of two chromosomes. This gave the advantage that during the selection phase of the population, duplicates of already selected chromosome where excluded independent of their dominance.

For future research it is suggested to test the NSGA-II on bigger test instance and compare it with other optimization methods. For example, methods with different encoding of the chromosomes as like the random keys sequencing (cf. [3], [21]). The random keys sequencing might improve the overall runtime of the optimization method, hence no additional heuristic of solving the TSP is required.

References

[1] S. Arrache and R. Ouafi. Accelerating convergence of the frank-wolfe algorithm for solving the traffic assignment problem. *IJC-SNS International Journal of Computer Science and Network Security*, 8(5), May 2008.

[2] T. Bäck. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford university press, 1996.

[3] J. C. Bean. Genetic algorithms and random keys for sequencing and optimization. *ORSA Journal on Computing*, 6(2):154–160, 1994.

[4] M. Beckmann, C. McGuire, and C. Winsten. *Studies in the Economics of Transportation*. Yale University Press, 1956.

[5] G. Clarke and J. W. Wright. Scheduling of vehicles from a central depot to a number of delivery points. *Operations Research*, 12(4):568–581, 1964.

[6] J. R. Correa, N. E. Stier-Moses, J. J. Cochran, L. A. Cox, P. Keskinocak, J. P. Kharoufeh, and J. C. Smith. *Wardrop Equilibria*. John Wiley & Sons, Inc., 2010.

[7] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[8] T. C. S. Dias, G. F. de Sousa Filho, E. M. Macambira, L. dos Anjos F. Cabral, and M. H. C. Fampa. *An Efficient Heuristic for the Ring Star Problem*, pages 24–35. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[9] K. Dörner, A. Focke, and W. J. Gutjahr. Multicriteria tour planning for mobile healthcare facilities in a developing country. *European Journal of Operational Research*, 179(3):1078–1096, 2007.

[10] N. Džubur. Optimization of warehouse locations nased on wardrope equilibria. Master's thesis, University of Vienna, 2013.

[11] M. Frank and P. Wolfe. An algorithm for quadratic programming. *Naval Research Logistics (NRL)*, 3(1-2):95–110, 1956.

[12] B. Golden, L. Bodin, T. Doyle, and W. Stewart Jr. Approximate traveling salesman algorithms. *Operations research*, 28(3-part-ii):694–711, 1980.

[13] B. L. Golden. A statistical approach to the tsp. *Networks*, 7(3):209–225, 1977.

[14] A. J. Hoffman and P. Wolfe. *The Traveling Salesman Problem*, chapter History, pages 1–15. John Wiley & Sons, 1985.

[15] R. Jayakrishnan, W. T. Tsai, J. N. Prashker, and S. Rajadhyaksha. A faster path-based algorithm for traffic assignment. *University of California Transportation Center*, 1994.

[16] J. Knowles and D. Corne. *On Metrics for Comparing Non-Dominated Sets*, pages 711–716. Institute of Electrical and Electronics Engineers, United States, 2002.

[17] M. Labbé, G. Laporte, I. R. Martın, and J. J. S. González. Locating median cycles in networks. *European Journal of Operational Research*, 160(2):457–470, 2005.

[18] M. Labbé, G. Laporte, I. R. Martín, and J. J. S. González. The ring star problem: Polyhedral analysis and exact algorithm. *Networks*, 43(3):177–189, 5 2004.

[19] M. Labbé, G. Laporte, I. R. Martín, and J. J. S. González. The median cycle problem, 1999.

[20] A. Liefooghe, L. Jourdan, and E.-G. Talbi. Metaheuristics and their hybridization to solve the bi-objective ring star problem: a comparative study. *arXiv preprint arXiv:0804.3965*, 2008.

[21] A. Liefooghe, L. Jourdan, and E.-G. Talbi. Metaheuristics and cooperative approaches for the bi-objective ring star problem. *Computers & Operations Research*, 37(6):1033 – 1044, 2010.

[22] J. A. M. Pérez, J. M. Moreno-Vega, and I. R. Martín. Variable neighborhood tabu search and its application to the median cycle problem. *European Journal of Operational Research*, 151(2):365 – 378, 2003. Meta-heuristics in combinatorial optimization.

[23] G. Reinelt. *The Traveling Salesman: Computational Solutions for TSP Applications*. Springer-Verlag, Berlin, Heidelberg, 1994.

[24] J. Renaud, F. F. Boctor, and G. Laporte. Efficient heuristics for median cycle problems. *The Journal of the Operational Research Society*, 55(2):179–186, 2004.

[25] N. Srinivas and K. Deb. Muiltiobjective optimization using nondominated sorting in genetic algorithms. *Evolutionary computation*, 2(3):221–248, 1994.

[26] D. A. van Veldhuizen and G. B. Lamont. Multiobjective evolutionary algorithm test suites. In *Proceedings of the 1999 ACM Symposium on Applied Computing*, SAC '99, pages 351–357, New York, NY, USA, 1999. ACM.

[27] J. G. WARDROP. Road paper. some theoretical aspects of road traffic research. *Proceedings of the Institution of Civil Engineers*, 1(3):325–362, 1952.

[28] E. Zitzler and L. Thiele. Multiobjective evolutionary algorithms: A comparative case study and the strength pareto approach. *IEEE Transactions on Evolutionary Computation*, 3(4):257–271, 1999.

A. Appendix

A.1. Test Instance for Parameter Settings

Vertex	1	2	3	4	5	6	7	8	9	10	$\sum$
Demand	56	65	47	71	56	34	34	25	24	41	453

Table A.1.: Demands of the vertices in the test instance for parameter setting

Vertex	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	0	24	85	83	65	57	27	77	82	52
2	24	0	89	91	87	47	40	81	83	56
3	85	89	0	16	77	53	59	9	12	34
4	83	91	16	0	63	62	56	15	27	36
5	65	87	77	63	0	91	51	69	83	64
6	57	47	53	62	91	0	44	48	43	30
7	27	40	59	56	51	44	0	50	57	27
8	77	81	9	15	69	48	50	0	14	26
9	82	83	12	27	83	43	57	14	0	30
10	52	56	34	36	64	30	27	26	30	0

Table A.2.: Distance matrix between the vertices in the test instance for parameter setting

A.2. Additional Results of Test Runs for Frank Wolfe Algorithm

α	β	# iteration	(UBD-LBD)/LBD	change of objective value	change f_j	change in f_{kj}	CPU time
1	0.75	50	0.0296 *(0.1077)*	0.0559% *(1.4089%)*	5.8139 *(15.8159)*	0.5815 *(2.58706)*	2min 11s
		100	0.0223 *(0.0725)*	0.0142% *(0.4320%)*	2.9289 *(7.9737)*	0.2936 *(1.2482)*	4min 23s
		200	0.0184 *(0.0453)*	0.0097% *(0.0988%)*	1.4027 *(3.8657)*	0.1462 *(0.6605)*	9min 26s
		500	0.0095 *(0.0174)*	0.0026% *(0.0131%)*	0.4265 *(1.4476)*	0.0521 *(0.2632)*	26min 7s
		1000	0.0038 *(0.0073)*	0.0006% *(0.0022%)*	0.1467 *(0.6790)*	0.0206 *(0.1384)*	40min 12s
		1500	0.0021 *(0.0045)*	0.0002% *(0.0010%)*	0.0787 *(0.4527)*	0.0114 *(0.0940)*	1h 12min 21s

Table A.3.: Test runs of Frank Wolfe Algorithm with $\alpha = 1,\ \beta = 0.75$

α	β	# iteration	(UBD-LBD)/LBD	change of objective value	change f_j	change in f_{kj}	CPU time
1	0.5	50	0.0389 *(0.1402)*	0.0665% *(1.4663%)*	5.8283 *(15.7925)*	0.5832 *(2.5849)*	2min 2s
		100	0.0305 *(0.0872)*	0.0300% *(0.4260%)*	2.8949 *(7.9410)*	0.2936 *(1.3238)*	3min 57s
		200	0.0227 *(0.0431)*	0.0140% *(0.0847%)*	1.2771 *(3.8636)*	0.1421 *(0.6599)*	7min 47s
		500	0.0087 *(0.0150)*	0.0025% *(0.0102%)*	0.3500 *(1.4469)*	0.0462 *(0.2705)*	19min 12s
		1000	0.0031 *(0.0067)*	0.0005% *(0.0024%)*	0.1178 *(0.6790)*	0.0171 *(0.1303)*	39min 35s
		1500	0.0016 *(0.0033)*	0.0002% *(0.0009%)*	0.0618 *(0.4526)*	0.0092 *(0.0888)*	1h 10min 8s

Table A.4.: Test runs of Frank Wolfe Algorithm with $\alpha = 1,\ \beta = 0.5$

α	β	# iteration	(UBD-LBD)/LBD	change of objective value	change f_j	change in f_{kj}	CPU time
1	0.25	50	0.0607 *(0.1439)*	0.1305% *(1.5333%)*	5.6865 *(15.5018)*	0.5781 *(2.6620)*	2min 1s
		100	0.0433 *(0.0917)*	0.0542% *(0.3080%)*	2.5298 *(7.4944)*	0.2810 *(1.3359)*	3min 54s
		200	0.0228 *(0.0436)*	0.0144% *(0.0571%)*	0.9869 *(3.6081)*	0.1229 *(0.6813)*	7min 42s
		500	0.0061 *(0.0130)*	0.0019% *(0.0106%)*	0.2338 *(1.4458)*	0.0338 *(0.2823)*	19min 22s
		1000	0.0020 *(0.0048)*	0.0003% *(0.0019%)*	0.0791 *(0.6034)*	0.0118 *(0.1398)*	45min 51s
		1500	0.0010 *(0.0033)*	0.0001% *(0.0007%)*	0.0425 *(0.4024)*	0.0063 *(0.0917)*	1h 21min 57s

Table A.5.: Test runs of Frank Wolfe Algorithm with $\alpha = 1,\ \beta = 0.25$

α	β	# iteration	(UBD-LBD)/LBD	change of objective value	change f_j	change in f_{kj}	CPU time
0.75	1	50	0.0211 *(0.0879)*	0.0448% *(0.3498%)*	5.8236 *(15.9900)*	0.5824 *(2.4971)*	2min 36s
		100	0.0139 *(0.0631)*	0.0074% *(0.0445%)*	2.9415 *(8.0735)*	0.2942 *(1.2642)*	5min 17s
		200	0.0118 *(0.0441)*	0.0037% *(0.1150%)*	1.4723 *(4.0030)*	0.1477 *(0.6489)*	486s
		500	0.0088 *(0.0347)*	0.0021% *(0.0141%)*	0.5273 *(1.5493)*	0.0575 *(0.2633)*	20min 53s
		1000	0.0047 *(0.0087)*	0.0006% *(0.0028%)*	0.2062 *(0.7243)*	0.0254 *(0.1333)*	42min 26s
		1500	0.0028 *(0.0047)*	0.0003% *(0.0012%)*	0.1098 *(0.4828)*	0.0148 *(0.0928)*	1h 5min 44s

Table A.6.: Test runs of Frank Wolfe Algorithm with $\alpha = 0.75,\ \beta = 1$

α	β	# iteration	(UBD-LBD)/LBD	change of objective value	change f_j	change in f_{kj}	CPU time
0.5	1	50	0.0173 *(0.0737)*	0.0445% *(0.4251%)*	5.8323 *(15.9789)*	0.5832 *(2.5078)*	2min 18s
		100	0.0101 *(0.0457)*	0.0065% *(0.0378%)*	2.9459 *(8.0721)*	0.2946 *(1.2656)*	4min 25s
		200	0.0082 *(0.0348)*	0.0012% *(0.0090%)*	1.4801 *(4.0067)*	0.1480 *(0.6281)*	9min 15s
		500	0.0070 *(0.0180)*	0.0013% *(0.0159%)*	0.5722 *(1.5815)*	0.0590 *(0.2639)*	23min 21s
		1000	0.0048 *(0.0087)*	0.0006% *(0.0034%)*	0.2459 *(0.7543)*	0.0278 *(0.1304)*	45min 9s
		1500	0.0031 *(0.0058)*	0.0003% *(0.0012%)*	0.1380 *(0.4830)*	0.0170 *(0.0897)*	57min 15s

Table A.7.: Test runs of Frank Wolfe Algorithm with $\alpha = 0.5,\ \beta = 1$

α	β	# iteration	(UBD-LBD)/LBD	change of objective value	change f_j	change in f_{kj}	CPU time
0.25	1	50	0.0136 *(0.0549)*	0.0457% *(0.3201%)*	5.8324 *(15.9716)*	0.5832 *(2.5078)*	2min 16s
		100	0.0064 *(0.0275)*	0.0055% *(0.0571%)*	2.9458 *(8.0711)*	0.2946 *(1.2656)*	4min 55s
		200	0.0044 *(0.0204)*	0.0008% *(0.0049%)*	1.4802 *(4.0564)*	0.1480 *(0.6359)*	8min 45s
		500	0.0039 *(0.0148)*	0.0003% *(0.0126%)*	0.5931 *(1.6075)*	0.0594 *(0.2531)*	22min 3s
		1000	0.0035 *(0.0091)*	0.0003% *(0.0037%)*	0.2875 *(0.7760)*	0.0296 *(0.1330)*	49min 26s
		1500	0.0030 *(0.0061)*	0.0002% *(0.0015%)*	0.1784 *(0.5174)*	0.0192 *(0.0866)*	1h 8min 30s

Table A.8.: Test runs of Frank Wolfe Algorithm with $\alpha = 0.25,\ \beta = 1$

A.3. Additional Results of Test Runs for the NSGA-II

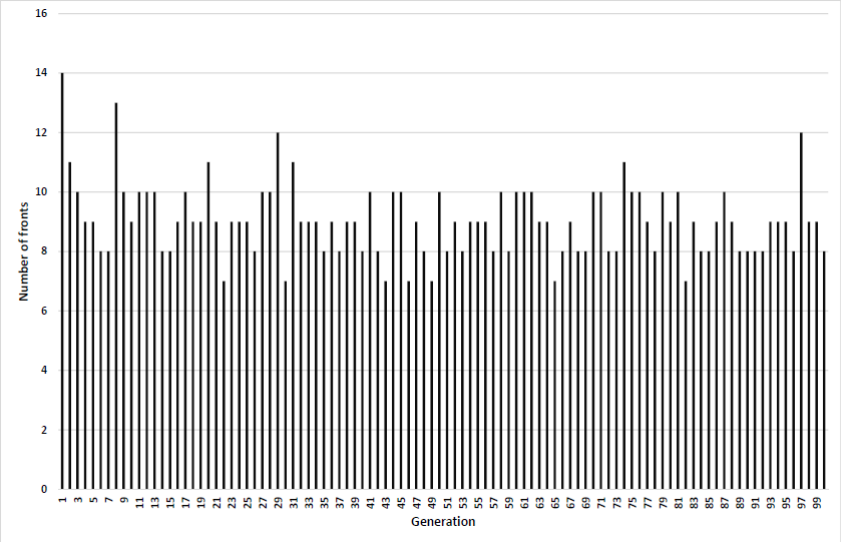

Figure A.1.: Number of fronts within a population of size 80 including a offspring population in each generation

A.4. Source Code of the Implemented NSGA-II

```
// library includes
#include <iostream>
#include <fstream>
#include <vector>
#include <random>
#include <time.h>
#include <Windows.h>
#include <cmath>
#include <algorithm>
#include <numeric>
#include <map>
#include <list>
#include <ctime>
#include <iomanip>

using namespace std;

// ************ global variables, structures and constans *************

// INPUT-VARIABLES: initialization
vector<vector<float>> distanceMatrix;    // symmetric matrix with distances between the vertices
vector<float> demand;                    // vector of the demand at each vertex
```

```
23
24 float alpha = 1;                          // cost of traveling a length unit
       on an arc
25 float beta = 0.25;                        // cost of loss in service quality
       per an additional person coming to a SP
26 int nrVertices = 10;                      // total number of vertices = V = {
       v_1,...,v_n}
27
28 int N = 40;                               // size of the population in every
       generation (NSGA-II)
29 int T = 50;                  // number of generations (NSGA-II)
30
31 float maxR = 283;                         // maximum Ring Cost of the
       instance (result from the enumeration)
32 float maxA = 2161190.75000;               // maximum Assignment Cost of the
       instance (result from the enumeration)
33
34 // VARIABLES FOR THE SAVINGS-ALGORITHM
35 struct EDGE                               // structure of two nodes, which
       form an edge, needed for calculation of savings
36 {
37   int c1, c2;
38 };
39
40 // VARIABLES FOR THE NSGA II
41 struct INDIVIDUAL                         // individual member of the
       popualtion
42 {
43   vector<int> chromosome;                // binary coded chromosome of size
         nrVertices
44   float ringCost;
45   long float assignmentCost;
46   int domCount;                          // number of solutions which
         dominate the current solution
47   vector<int> domSet;                    // the set of individuals which are
         dominated by current individual
48   int rank;                              // the rank of the individual =
         fitness = front
49   float crowdingDist;                    // crowding distance of chromosome
         between other chromosomes within the same rank
50 };
51 vector<INDIVIDUAL> population;           // vector of all members of size N
52
53 vector<vector<int>> fronts;              // vector of all fronts/ranks with
       their members
54
55 vector<int> nrFrontsPop;          // contains the number of fronts in each
       generation after selection phase (population)
56 vector<int> nrFrontsOff;                 // contains the number of fronts
       in each generation inclusise the offspring (population + offspring)
57 int runTime;                 // CPU time of the algorithm
58 float hv;                  // hypervolume covered by the 1. fronts after T
       generations, using the max values as reference point
```

```
59
60
61  // ******************** FUNCTIONS ******************************
62
63  // reading a .txt file, which contains the demand values and the distance
        matrix
64  void readDataFile()
65  {
66    ifstream dataFile("U_20_40_1_e.txt");
67    int tmp = 0;                          //temporary variable
68    // read the values of the demand vector
69    for (int i = 1; i <= nrVertices; i++)
70    {
71      dataFile >> tmp;
72      if (!dataFile.fail())
73      {
74        // tmp is inserted into the vector from behind
75        demand.push_back(tmp);
76      }
77    }
78
79    // read the values for the distance Matrix
80    vector<float> tmp1;
81    for (int i = 1; i <= nrVertices; i++)
82    {
83      int tmp2 = 0;
84      for (int j = 1; j <= nrVertices; j++)
85      {
86        dataFile >> tmp;
87        if (!dataFile.fail())
88        {
89          tmp1.push_back(tmp);
90        }
91      }
92      distanceMatrix.push_back(tmp1);
93      tmp1.clear();
94    }
95    dataFile.close();
96  }
97
98  // writing a .txt file with the results
99  void writeDataFile()
100 {
101   ofstream output;
102   output.open("Result_U_20_40_1.txt");
103   output << "Runtime_:_" << runTime << endl; // in seconds
104   output << "Hypervolume:_" << fixed << setprecision(10) << hv << endl;
105   output << endl << "List_of_fronts_within_a_generation:_" << endl << fixed
          << setprecision(0);
106   for (int i = 0; i < nrFrontsPop.size(); i++)
107   {
108     output << nrFrontsPop[i] << endl;
109   }
```

```
110   output << endl << "List of fronts with 2N: \n";
111   for (int j = 0; j < nrFrontsOff.size(); j++)
112   {
113     output << nrFrontsOff[j] << endl;
114   }
115   output << endl << "Solution Number" << "    " << "Ring Cost" << " " << "
          Assignment Cost" << "   " << "Solution " << "   " << "Rank \n";
116   for (int i = 0; i < population.size(); i++)
117   {
118     output << (i + 1) << "    " << fixed << setprecision(0) << round(
          population[i].ringCost) << "   " << fixed << setprecision(5) <<
          population[i].assignmentCost << "    ";
119     output << fixed << setprecision(0);
120     for (int p = 0; p < nrVertices; p++)
121     {
122       output << population[i].chromosome[p] << " ";
123     }
124     output << "   " << population[i].rank << "\n";
125   }
126   output.close();
127 }
128
129 // ************************FUNCTIONS FOR THE SAVINGS ALGORITHM
      ***************************************************************************

130 // creating the saving list, with double entrys of the saving and the
      corresponding edge
131 void calcSavings(int start, multimap<float, EDGE> &listSav, INDIVIDUAL
      chrom, vector<int> cities)
132 {
133   EDGE edge;
134   // the savings of all edges are calculated
135   for (int i = 0; i < cities.size(); i++)
136   {
137     // if the vertex i is open -> =1
138     if (cities[i] != 0 && i != start)
139     {
140       for (int j = 0 + i; j < cities.size(); j++)
141       {
142         if (i != j && cities[j] != 0) //
143         {
144           float sav = 0.0;
145           edge.c1 = i;
146           edge.c2 = j;
147           // since the multimap is sorting the saving automaticly in rising
                order
148           sav = (-1)*(distanceMatrix[i][start] + distanceMatrix[start][j] -
                distanceMatrix[i][j]);
149           listSav.insert(make_pair(sav, edge));
150         }
151       }
152     }
153   }
```

```
154 }
155
156 // initialize routes, from start is to every visited vertex a route created
157 void initRoutes(int start, INDIVIDUAL chrom, list<list<int>> &initialRoutes
        , vector<int> cities)
158 {
159   // for every vertex i in the ring, which is not the start of the tour a
          route is created (start, i, start)
160   // whereby the start node is added at the end of the savings algorithm
161   for (int i = 0; i < cities.size(); i++)
162   {
163     if (cities[i] == 1 && i != start)
164     {
165       list<int> route;
166       route.push_back(i);
167       initialRoutes.push_back(route);
168     }
169   }
170 }
171
172 // clear routes which are merged
173 void clearRoute(int tour, INDIVIDUAL chrom)
174 {
175   // after two routes are merged, the old routes are cleared
176   int dim = count(chrom.chromosome.begin(), chrom.chromosome.end(), 1);
177   for (int i = 0; i < dim; i++);
178   {
179
180   }
181 }
182
183 // merging two edges if there is a positiv saving
184 void mergeEdges(int start, list<int> &tour, list<list<int>> &initRoutes,
        multimap<float, EDGE> &listSav, INDIVIDUAL chrom)
185 {
186   EDGE e;
187   // find the edge belonging to the saving
188   e = listSav.begin()->second;
189
190   // variables for merging two edges
191   list<int> t1;
192   list<int> t2;
193   list<list<int>>::iterator it1;
194   list<list<int>>::iterator it2;
195   bool b1 = false, b2 = false;
196
197   // iteration through initial routes to find the vertices of the edge and
          merge them to one route
198   for (list<list<int>>::iterator it = initRoutes.begin(); it != initRoutes.
          end(); it++)
199   {
200     // find two routes, so the edge e can be added
201     if (!b1 && it->front() == e.c1)
```

```
202       {
203         it1 = it;
204         t1 = *it;
205         b1 = true;
206       }
207       else if (!b1 && it->front() == e.c2)
208       {
209         it1 = it;
210         t1 = *it;
211         b1 = true;
212       }
213       else if (!b2 && it->back() == e.c1)
214       {
215         it2 = it;
216         t2 = *it;
217         b2 = true;
218       }
219       else if (!b2 && it->back() == e.c2)
220       {
221         it2 = it;
222         t2 = *it;
223         b2 = true;
224       }
225
226       if (b1 && b2)
227       {
228         // delete the two initial routes from initial routes list: (start, a,
                start) & (start, b, start)
229         initRoutes.erase(it1);
230         initRoutes.erase(it2);
231
232         // add one merged tour : (start, a,b, start)
233         list<int> newTour;
234         newTour.insert(newTour.end(), t2.begin(), t2.end());
235         newTour.insert(newTour.end(), t1.begin(), t1.end());
236
237         // clear the routes which are merged
238         if (t1.size() > 1)
239         {
240           list<int>::iterator it = t1.begin();
241           it++;
242           while (it != t1.end() && next(it) != t1.end())
243           {
244             clearRoute(*it, chrom);
245             it++;
246           }
247         }
248         if (t2.size() > 1)
249         {
250           list<int>::iterator it = t2.begin();
251           it++;
252           while (it != t2.end() && next(it) != t2.end())
253           {
```

```
254             clearRoute(*it, chrom);
255             it++;
256           }
257         }
258         // add the new route to the list of routes, whereby at the end this
                list contains just one final route
259         initRoutes.push_front(newTour);
260         return;
261       }
262     }
263     // delete the used saving from list
264     multimap<float, EDGE>::iterator itDelete = listSav.begin();
265     listSav.erase(itDelete);
266   }
267
268   // claculation of the tour length
269   float tourLength(list<int> tour)
270   {
271     float distance = 0.0;
272     list<int>::iterator it1 = tour.begin();
273     list<int>::iterator it2 = tour.begin();
274     it2++;
275     while (it2 != tour.end())
276     {
277       distance += distanceMatrix[*it1][*it2];
278       it2++;
279       it1++;
280     }
281     return distance;
282   }
283
284   // calculation of ring costs for each soultion using the Clarke&Wright
        Savings-Algorithm
285   void calcRingCost(INDIVIDUAL &chrom)
286   {
287     double costRoute = 0.0;
288     vector<int> open = chrom.chromosome;
289
290     // if more than one vertex is chosen to be visited calc ring cost
291     // else the ring cost are zero, because there is no ring
292     int test = count(open.begin(), open.end(), 1);
293     if (test > 1)
294     {
295       multimap<float, EDGE> listSav;           // list of all savings with
            corresponding edge
296       list<list<int>> initialRoutes;           // initial Routes for start
            the savings algorithm
297       list<int> finalTour;                     // list of the sequence of the
             final tour
298       int start;
299
300       // w.l.o.g. the first visited node is the starting node of the route
301       for (int i = 0; i < open.size(); i++)
```

```
302       {
303         if (open[i] != 0)
304         {
305           start = i;
306           break;
307         }
308       }
309
310       // SAVINGS-ALGORITHM
311       // calculation of the savings
312       calcSavings(start, listSav, chrom, open);
313
314       // initialization of (n-1) routes: for each visited vertex a route from
              start and back is created
315       initRoutes(start, chrom, initialRoutes, open);
316
317       // as long as the savings are positiv(<=> smaller than zero, since they
              multimplied by (-1)), merge routes
318       while (listSav.size() > 0)
319       {
320         multimap<float, EDGE>::iterator itListSav = listSav.begin();
321         if (itListSav->first <= 0)
322         {
323           mergeEdges(start, finalTour, initialRoutes, listSav, chrom);
324         }
325         else if (itListSav->first > 0)
326           break;
327       }
328
329       list<list<int>>::iterator tourIt = initialRoutes.begin();
330       finalTour.insert(finalTour.end(), (*tourIt).begin(), (*tourIt).end());
331
332       // add start to the beginging and end of the final tour sequence -> for
              calculation of the length
333       finalTour.push_front(start);
334       finalTour.push_back(start);
335
336       // calculate the tour length
337       costRoute = tourLength(finalTour);
338     }
339     chrom.ringCost = costRoute;
340   }
341
342
343   // ********************FUNCTIONS FOR FRANK-WOLFE ALGORITHM: calculation of
          user equilibrium*************************************
344   /* The Frank-Wolfe Algorithm is used to calculated the Wardrop Equilibrium
          (= User Equilibrium).
345      The output of this function is a matrix of total inflows to each visited
              village.*/
346
347   // matrix multiplication
348   vector<vector<float>> multiplicationMatrix(vector<vector<float>> matrixA,
```

```
    vector<vector<float>> matrixB, int sizeMatrix)
{
  vector<vector<float>> AB;
  vector<float> lineVec;
  for (int i = 0; i < sizeMatrix; i++)
  {
    for (int j = 0; j < sizeMatrix; j++)
    {
      double sum = 0;
      for (int k = 0; k < sizeMatrix; k++)
      {
        sum += (matrixA[i][k] * matrixB[k][j]);
      }
      lineVec.push_back(sum);
    }
    AB.push_back(lineVec);
    lineVec.clear();
  }
  return AB;
}

// multiplication of a matrix A with a vector b -> Ab
vector<float> multiplicationMatVec(vector<vector<float>> matrix, vector<
    float> vec)
{
  int n = vec.size();
  vector<float> Ab;
  for (int i = 0; i < n; i++)
  {
    float sum = 0.0;
    for (int j = 0; j < n; j++)
    {
      sum += (matrix[i][j] * vec[j]);
    }
    Ab.push_back(sum);
  }
  return Ab;
}

// initialize matrizes with zeros
vector<vector<float>> initZeros(int row, int column)
{
  vector<float> zeroVec(column, 0.0); // initialize the vector of size
      column and with zeros
  vector<vector<float>> matrix(row, zeroVec);
  return matrix;
}

// Frank-Wolfe Algorithm
void FrankWolfAlg(int nrIterations, INDIVIDUAL chrom, long float &
    assignCost)
{
  // number of total vertices
```

```
398     int n = chrom.chromosome.size();
399     // number of SP's
400     int m = count(chrom.chromosome.begin(), chrom.chromosome.end(), 1);
401     // copy chromosome coding to a vector -> easier to handle
402     vector<int> open = chrom.chromosome;
403     // holds the distances from all vertices to the SP
404     vector<vector<float>> distanceSP;
405
406     //~~~~~~~~initialization of matrices and vectors~~~~~~~~
407
408     // indicator matrix with zero entries
409     vector<vector<float>> indicatorMatrix = initZeros(n, n);
410
411     // set the diagonal values equal to one, if vertex i is a SP
412     for (int i = 0; i < n; i++)
413     {
414       if (open[i] == 1)
415       {
416         indicatorMatrix[i][i] = 1;
417       }
418     }
419
420     // multiplication of indicator matrix with distance matrix
421     // only the entries of distances to SP's are left
422     distanceSP = multiplicationMatrix(indicatorMatrix, distanceMatrix, n);
423
424     // testing vector with ones
425     vector<int> test(n, 1);
426
427     // multiplication the columns of the distanceToStop matrix with test
          vector, for testing if the row i is leading to a SP
428     // if a row contains just zero-entries, it is erased
429     for (int i = (n - 1); i >= 0; i--)
430     {
431       float tmp = 0;
432       for (int j = 0; j < n; j++)
433       {
434         tmp += (distanceSP[i][j] * test[j]);
435       }
436       if (tmp == 0)
437       {
438         distanceSP.erase(distanceSP.begin() + i);
439       }
440     }
441
442     // setup flow matrix with m rows and n columns -> m SP's and n total
          vertices
443     // flow from vertex j..n to SP i..m
444     vector<vector<float>> flowSPVil = initZeros(m, n);
445
446     // setup a matrix for search directions / optimal cornerpoints
447     vector<vector<float>>  matrixG = initZeros(m, n);
448
```

```
449   // vector of incoming flows into each SP
450   // sum of all incoming flows to a SP
451   vector<float> flowSP(m, 0);
452
453   // stepsize of Frank-Wolfe -> linearization of the search
454   float xi = 1;
455
456   // matrix with cornerpoints of the j-th simplex
457   vector<vector<float>> cornerpoint;
458   for (int k = 0; k < n; k++)          // for each vertex
459   {
460     cornerpoint = initZeros(m, m);  // setup of a m*m simplex for each
          village
461     for (int j = 0; j < m; j++)      // for each SP
462     {
463       // extend simplex with demand values
464       cornerpoint[j][j] = demand[k];
465     }
466     for (int i = 0; i < m; i++) // for each SP
467     {
468       float sum = accumulate(cornerpoint[i].begin(), cornerpoint[i].end(),
              0);
469       flowSPVil[i][k] = sum / m;
470     }
471     cornerpoint.clear();
472   } // end k
473
474   // total flows within each DC
475   for (int j = 0; j < m; j++)
476   {
477     flowSP[j] = accumulate(flowSPVil[j].begin(), flowSPVil[j].end(), 0.0);
478   }
479   // ~~~~~~~~~~~~~~~~~~ Frank-Wolfe Algorithm loop
        ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
480
481   for (int t = 0; t < nrIterations; t++)
482   {
483     // defining previous flows
484     vector<vector<float>> f1 = flowSPVil;
485     vector<float> flow1 = flowSP;
486
487     // determination of increment per iteration ( xi = 2/(t+2) )
488     xi = 2 * xi / (xi + 2);
489
490     // service quality values at each SP
491     vector<float> serviceQuality(m, 0); // mx1 vector with zeros
          initialized
492     for (int i = 0; i < m; i++)
493     {
494       // the service quality depends on the incoming total flow to a SP
495       serviceQuality[i] = ((flowSP[i] * flowSP[i]));
496     }
497
```

```
498       float sum = accumulate(serviceQuality.begin(), serviceQuality.end(),
              0.0);
499
500       // determination of search direction (vector to edgepoint) with optimal
              value within each simplex k
501       for (int k = 0; k < n; k++) // for each vertex
502       {
503         // starting value for search for optimal edge
504         float minval = LLONG_MAX;
505
506           // overwrite previous values of cornerpoint with zeros
507         cornerpoint = initZeros(m, m);
508
509         // extend simplex with demand values
510         for (int j = 0; j < m; j++) // for each SP
511         {
512           cornerpoint[j][j] = demand[k];
513         }
514
515         for (int j = 0; j < m; j++) // for each SP
516         {
517           // calculation total distance and service for a solution
518           long float tmpSumDis = 0.0;
519           long float tmpSumSer = 0.0;
520           // multiplication of distanceMatrix's line vector k with
                  cornerpoints matrix's vector j
521           // multiplication of serviceQuality vector with cornerpoints matrix
                  's vector j
522           for (int dc = 0; dc < m; dc++)
523           {
524             tmpSumDis += (distanceSP[dc][k] * cornerpoint[dc][j]);
525             tmpSumSer += (serviceQuality[dc] * cornerpoint[dc][j]);
526           }
527
528           // calculation of objective function
529           float obj = (alpha*tmpSumDis) + (beta*tmpSumSer);
530           if (obj < minval)
531           {
532             minval = obj;
533             // update search directions if better objective value is found
534             for (int d = 0; d < m; d++)
535             {
536               // the edge with the better objective gets a new search
                      direction
537               matrixG[d][k] = cornerpoint[d][j];
538             }
539           } // end if-loop
540         } // end j : determination of matrixG
541
542         // determination of flows on the interval between the previous flow
              and the search direction dep. of increment
543         for (int d = 0; d < m; d++)
544         {
```

```
545           flowSPVil[d][k] = ((1 - xi)*flowSPVil[d][k]) + (xi*matrixG[d][k]);
546         }
547         cornerpoint.clear();
548       }// end k..n
549
550
551        // update the flow values within the SP from newly determined flows
552       for (int j = 0; j < m; j++)
553       {
554         flowSP[j] = accumulate(flowSPVil[j].begin(), flowSPVil[j].end(), 0.0)
              ;
555       }
556     } //end Frank-Wolfe loop
557
558     // calculation of assignment cost after max iterations
559     // first the distances
560     for (int k = 0; k < n; k++)
561     {
562       for (int j = 0; j < m; j++)
563       {
564         assignCost += alpha*distanceSP[j][k] * flowSPVil[j][k];
565       }
566     }
567     for (int j = 0; j < m; j++)
568     {
569       long float b = beta*((pow(flowSP[j], 3)) / 3);
570       assignCost += b;
571     }
572   }
573   // calculation assignment costs based on the flows resulting from Frank-
        Wolfe algorithm
574   long float calcAssignCost(INDIVIDUAL chrom)
575   {
576     long float assignCost = 0.0;
577     int iterations = 700;
578     FrankWolfAlg(iterations, chrom, assignCost);
579     return assignCost;
580   }
581
582   // ************************************FUNCTIONS FOR THE NSGA II
        ***************************************************************************************

583   // the fast non domination sort introduced by Deb et al.
584   void FastNonDominatedSort()
585   {
586     // before sorting clear previous fronts, because of new sorting and
          domination relations
587     fronts.clear();
588     // set with all p in population which has no rank yet... first just
          solutions with dominanceCount = 0, are assigned to rank 1
589     vector<int> setNoRank;
590     // initialization of ranks
591     vector<int> Rank_i;
```

```
592     // initialize the dominance count and clear domSet
593     for (int p = 0; p < population.size(); p++)
594     {
595       population[p].domCount = 0;
596       population[p].domSet.clear();
597     }
598     // compare each individual with each other regarding their fitness in
            terms of the objectives
599     for (int p = 0; p < population.size(); p++)
600     {
601       for (int q = (p + 1); q < population.size(); q++)
602       {
603         if (population[p].ringCost <= population[q].ringCost && population[p
              ].assignmentCost <= population[q].assignmentCost)
604         {
605           if (population[p].ringCost < population[q].ringCost && population[p
                ].assignmentCost <= population[q].assignmentCost)
606           {
607             population[p].domSet.push_back(q);
608             population[q].domCount++;
609           }
610           else if (population[p].ringCost <= population[q].ringCost &&
                  population[p].assignmentCost < population[q].assignmentCost)
611           {
612             population[p].domSet.push_back(q);
613             population[q].domCount++;
614           }
615         }
616         if (population[p].ringCost >= population[q].ringCost && population[p
              ].assignmentCost >= population[q].assignmentCost)
617         {
618           if (population[p].ringCost > population[q].ringCost && population[p
                ].assignmentCost >= population[q].assignmentCost)
619           {
620             population[q].domSet.push_back(p);
621             population[p].domCount++;
622           }
623           else if (population[p].ringCost >= population[q].ringCost &&
                  population[p].assignmentCost > population[q].assignmentCost)
624           {
625             population[q].domSet.push_back(p);
626             population[p].domCount++;
627           }
628         }
629       }//q-pop.size
630
631       // initialization of the first front
632       if (population[p].domCount == 0)
633       {
634         population[p].rank = 1;
635         Rank_i.push_back(p);
636       }
637       else
```

```
638       {
639         setNoRank.push_back(p);
640       }
641     }//p-pop.size()
642 
643     fronts.push_back(Rank_i);  // keep track of ranks and its members
644 
645     // detemination of other fronts = ranks
646     // best rank has the number "0"
647     int i = 0;
648 
649     // number of solution which need to be assigned to a rank
650     int n = setNoRank.size();
651 
652     // while there are solution without a rank, assign ranks
653     while (setNoRank.size() > 0)
654     {
655       i++;  // rank number
656       vector<int> tmp = Rank_i;
657       Rank_i.clear();
658       for (int j = 0; j < tmp.size(); j++)
659       {
660         // the set of dominated solutions by p
661         for (int q = 0; q < population[tmp[j]].domSet.size(); q++)
662         {
663           int a = population[tmp[j]].domSet[q];
664           // reduce dominance count of the members of the set by one
665           population[a].domCount--;
666           // if dominance count is zero now, pop[a] is a member of the next
                front/rank and can be assigned to it
667           if (population[a].domCount == 0)
668           {
669             population[a].rank = (i + 1);
670             Rank_i.push_back(a);
671             setNoRank.erase(find(setNoRank.begin(),setNoRank.end(), a));  //
                  remove the assigned solution from the set without ranks
672           }
673         }
674       }
675       fronts.push_back(Rank_i);
676     }
677   }
678 
679   // calculation of the crowding distance for each solution within a front /
        with the same rank
680   // introduced by Deb et al.
681   void crowdingDistance()
682   {
683     // idetify all solution in same rank
684     for (int i = 0; i < fronts.size(); i++)
685     {
686       // placeholder for a sort by ring cost and a sort by assignment cost
          each front and the corresponding chromosome number in the
```

```
        population vector
687     multimap<float, int> sortR;
688     multimap<long float, int> sortA;
689     int n = fronts[i].size();
690     for (int j = 0; j < fronts[i].size(); j++)
691     {
692       population[fronts[i][j]].crowdingDist = 0;
693       sortR.emplace(make_pair(population[fronts[i][j]].ringCost, fronts[i][
            j]));
694       sortA.emplace(make_pair(population[fronts[i][j]].assignmentCost,
            fronts[i][j]));
695     }
696
697     vector<int> sortByRing;
698     vector<int> sortByAssign;
699
700     // the corresponding integers are inserted in sorted order into a
          vector, we need just to know the solutions belonging to min and max
701     for (multimap<float, int>::iterator itR = sortR.begin(); itR != sortR.
          end(); itR++)
702     {
703       sortByRing.push_back(itR->second);
704     }
705
706     for (multimap<long float, int>::iterator itA = sortA.begin(); itA !=
          sortA.end(); itA++)
707     {
708       sortByAssign.push_back(itA->second);
709     }
710
711     // set boundary points to infinity -> very big number
712     population[sortByRing[0]].crowdingDist = population[sortByRing[n - 1]].
          crowdingDist = population[sortByAssign[0]].crowdingDist =
          population[sortByAssign[n - 1]].crowdingDist = 100000000;
713     float maxRing = population[sortByRing[n - 1]].ringCost;
714     float minRing = population[sortByRing[0]].ringCost;
715     long float maxAssign = population[sortByAssign[n - 1]].assignmentCost;
716     long float minAssign = population[sortByAssign[0]].assignmentCost;
717
718     for (int k = 1; k < (n - 1); k++)
719     {
720       population[sortByRing[k]].crowdingDist += (population[sortByRing[k +
            1]].ringCost - population[sortByRing[k - 1]].ringCost) / (maxRing
            - minRing);
721       population[sortByAssign[k]].crowdingDist += (population[sortByAssign[
            k + 1]].assignmentCost - population[sortByAssign[k - 1]].
            assignmentCost) / (maxAssign - minAssign);
722     }
723   }
724 }
725
726 // initialization of a population of size N, with individuals (=solutions)
      coded binary
```

```
727 // entry i is 1 iff vertex i is visited, else 0
728 void initPopulation()
729 {
730   INDIVIDUAL chrom;
731   random_device rd;
732   uniform_int_distribution<int> distribution(0, 1);
733   mt19937 engine(rd());
734   // N different individuals need to be created
735   for (int i = 0; i < N; i++)
736   {
737     // code the binary chromosome
738     for (int gene = 0; gene < nrVertices; gene++)
739     {
740       int geneValue = distribution(engine);
741       chrom.chromosome.push_back(geneValue);
742     }
743     // calculation of fitness values
744     calcRingCost(chrom);
745     chrom.assignmentCost = calcAssignCost(chrom);
746     population.push_back(chrom);
747     chrom.chromosome.clear();
748   }
749   // calculation of rank and crowding distance for the initial population
750   FastNonDominatedSort();
751   nrFrontsPop.push_back(fronts.size());  // keep track of number of fronts
        through the algorithm for the population of size N
752   crowdingDistance();
753 }
754
755 // check for duplicats and delete them: returns true, if chrom is a
      duplicat of an individual in the current population
756 bool findDuplicate(vector<INDIVIDUAL> &somePop, vector<int> chrom)
757 {
758   bool duplicate = false;
759   if (somePop.size() != 0)
760   {
761     for (int i = 0; i < somePop.size(); i++)
762     {
763       if (equal(somePop[i].chromosome.begin(), somePop[i].chromosome.end(),
            chrom.begin(), chrom.end()))
764         duplicate = true;
765     }
766   }
767   return duplicate;
768 }
769
770 // mutation operator -> flip a gene with a specific mutation rate
771 void mutation(vector<int> &candidate)
772 {
773   random_device rd;
774   uniform_int_distribution<int> distribution(1, 100);
775   uniform_int_distribution<int> uni(0, nrVertices-1);
776   mt19937 engine(rd());
```

```
777
778   for (int i = 0; i < nrVertices; i++)
779   {
780     int value = distribution(engine);
781     // 10% probability for each gene
782     if (value >=10 && value < 20)
783     {
784       int gene = uni(engine);
785       if (candidate[i] != 0)
786         candidate[i] = 0;
787       else
788         candidate[i] = 1;
789     }
790   }
791 }
792
793 // crossover operator, INPUT: two parent solutions, OUTPUT: two offspring
      solutions and calculations of the objectives
794 void crossOver(vector<INDIVIDUAL> parents, vector<INDIVIDUAL> &offspring)
795 {
796   // possible crossover points: (0, 1, 2, 3, ..., (nrVertices-1)) -> cut
       will be made in front of the chosen position
797   // hence there is no sense to cut before 0, or just flip one gene of each
       solution
798   int min = 1;
799   int max = nrVertices-2; //
800
801   random_device rd;      // only used once to initialise seed
802   mt19937 rng(rd());     // random-number engine used
803   uniform_int_distribution<int> uni(min, max);
804   int position = uni(rng);
805
806   vector<int> parent_1 = parents[0].chromosome;
807   vector<int> parent_2 = parents[1].chromosome;
808
809   INDIVIDUAL off_1;
810   INDIVIDUAL off_2;
811   // first check if the chromosome are identic, if not continue
812   if (!equal(parent_1.begin(), parent_1.end(), parent_2.begin(), parent_2.
       end()))
813   {
814     vector<int> gene_1;
815     vector<int> gene_2;
816     // crossover
817     for (int i = 0; i < position; i++)
818     {
819       gene_1.push_back(parent_1[i]);
820       gene_2.push_back(parent_2[i]);
821
822       parent_1[i] = gene_2[i];
823       parent_2[i] = gene_1[i];
824     }
825     // check with the parent population if a duplicate is produced
```

```
826       // findDuplicate returns true, if offspring is a duplicate of an
              existing member of the population
827       bool dupl_1 = findDuplicate(population, parent_1);
828       bool dupl_2 = findDuplicate(population, parent_2);
829       if (!dupl_1 | !dupl_2) // if at least one is not a duplicate
830       {
831         if (dupl_1 && !dupl_2) // if parent_1 is a duplicate and parent_2 not
                (these are now the offspring)
832         {
833           parent_1.clear();
834           for (int i = 0; i < nrVertices; i++)
835           {
836             off_2.chromosome.push_back(parent_2[i]);
837           }
838           mutation(off_2.chromosome);
839           calcRingCost(off_2);
840           off_2.assignmentCost = calcAssignCost(off_2);
841           offspring.push_back(off_2);
842         }
843         if (dupl_2 && !dupl_1) // if parent_2 is a duplicate and parent_1 not
                (these are now the offspring)
844         {
845           parent_2.clear();
846           for (int i = 0; i < nrVertices; i++)
847           {
848             off_1.chromosome.push_back(parent_1[i]);
849           }
850           mutation(off_1.chromosome);
851           calcRingCost(off_1);
852           off_1.assignmentCost = calcAssignCost(off_1);
853           offspring.push_back(off_1);
854         }
855         if (dupl_2 && dupl_1)  // if both are duplicates
856         {
857           parent_1.clear();
858           parent_2.clear();
859         }
860       }
861       if (parent_1.size() != 0 && parent_2.size() != 0)  // if both not
              duplicates and not cleared
862       {
863         for (int i = 0; i < nrVertices; i++)
864         {
865           off_1.chromosome.push_back(parent_1[i]);
866           off_2.chromosome.push_back(parent_2[i]);
867         }
868         mutation(off_1.chromosome);
869         mutation(off_2.chromosome);
870
871         calcRingCost(off_1);
872         off_1.assignmentCost = calcAssignCost(off_1);
873
874         calcRingCost(off_2);
```

```
875         off_2.assignmentCost = calcAssignCost(off_2);
876
877         offspring.push_back(off_1);
878         offspring.push_back(off_2);
879       }
880     }
881   }
882
883   // random operator for choosing two solutions for tournament
884   void randomSelectionForTournament(INDIVIDUAL &a, INDIVIDUAL &b)
885   {
886     // need to select two solutions out of the population
887     int min = 0;
888     int max = (N - 1); // since the counting starts with zero not with one
889     vector<int> numbers;
890     random_device rd;
891     // two random numbers are needed
892     mt19937 rng(rd());
893     uniform_int_distribution<int> uni(min, max);
894     a = population[uni(rng)];
895     b = population[uni(rng)];
896   }
897
898   // two tournaments between two solutions, winner is taken to reproduction,
        returns two solutions for reproduction = crossover
899   //  CROWDED COMPARISON OPERATOR is the tournament procedure
900   void tournament(vector<INDIVIDUAL> &winner)
901   {
902     // two binary tournaments
903     for (int i = 0; i < 2; i++)
904     {
905       INDIVIDUAL a;
906       INDIVIDUAL b;
907       randomSelectionForTournament(a, b);
908       // compare the rank, if "a" has smaller rank, it wins
909       if (a.rank < b.rank)
910         winner.push_back(a);
911       // if equal rank, compare the crowding distance -> solution with bigger
            crowding distance wins
912       else if(a.rank == b.rank)
913       {
914         if(a.crowdingDist > b.crowdingDist)
915           winner.push_back(a);
916         else
917           winner.push_back(b);
918       }
919       else
920         winner.push_back(b);
921     }
922   }
923
924   // selection of the members of the next generation
925   //vector<INDIVIDUAL>
```

```
926 void selectNextGeneration(vector<INDIVIDUAL> &nextGeneration)
927 {
928   for (int i = 0; i < fronts.size(); i++)
929   {
930     if ((N - nextGeneration.size()) != 0)
931     {
932       // as long all members of one whole rank fitting into next generation
            without exceeding N, insert all members of the front
933       if (fronts[i].size() <= (N - nextGeneration.size()))
934       {
935         for (int j = 0; j < fronts[i].size(); j++)
936         {
937           // just feasable solution -> at least one SP is open
938           if (count(population[fronts[i][j]].chromosome.begin(), population
                [fronts[i][j]].chromosome.end(), 1) > 0)
939           {
940             bool d = findDuplicate(nextGeneration, population[fronts[i][j
                  ]].chromosome); // no duplicates are allowed
941             if (d != true)
942             {
943               nextGeneration.push_back(population[fronts[i][j]]);
944             }
945           }
946         }
947       }
948       // just the (N-nextGeneration.size()) chromosome sorted by crowding
            distance are select to get into next generation
949       // bigger crowding distane is better
950       else if (fronts[i].size() > (N - nextGeneration.size()))
951       {
952         vector<int> tmp(fronts[i].begin(), fronts[i].end());
953         int l = (N - nextGeneration.size());
954         for (int m = 0; m < l; m++)
955         {
956           int pos = 0; ;
957           float max = 0;
958           bool c = false;
959           for (int k = 0; k < tmp.size(); k++)
960           {
961               if (count(population[tmp[k]].chromosome.begin(), population[
                    tmp[k]].chromosome.end(), 1)>0)
962               {
963                 if (population[tmp[k]].crowdingDist > max)
964                 {
965                   max = population[tmp[k]].crowdingDist;
966                   pos = tmp[k];
967                   c = true;
968                 }
969               }
970             }
971             if(c)  // a max value was found, the if loop was executed
972             {
973               bool d = findDuplicate(nextGeneration, population[pos].
```

```
                    chromosome);  // no duplicates are allowed
974                 if (d != true)
975                 {
976                   nextGeneration.push_back(population[pos]);
977                 }
978                 tmp.erase(find(tmp.begin(), tmp.end(), pos));
979               }
980             }
981           if (nextGeneration.size() == N)
982             break;
983         }
984       }// end of if ((N - nextGeneration.size()) == 0)
985       else // if ((N - nextGeneration.size()) == 0)
986         break;
987     }
988   }
989
990   // calculation of the hypervolume using the first front of the generated
          popualtion after t generations
991   // reference point is given by the max ring cost and max assignment costs
          from the enumeration results
992   // this algorothm just works for bi-objective optimization problems
993   void calcHypervolumen(vector<INDIVIDUAL> &allSol, vector<int> &pareto,
          float &maxRing, float &maxAssignment, float &hypervolume)
994   {
995     multimap<long float, int> sortA;
996     // sort all member of the pareto front according to there assignment
          costs in rising order
997     for (int i = 0; i < pareto.size(); i++)
998     {
999       sortA.emplace(make_pair(allSol[pareto[i]].assignmentCost, pareto[i]));
1000    }
1001    // iterate through the points if the sorted list according the assignment
          costs
1002
1003    multimap<long float, int>::iterator it = sortA.end();
1004    it--; // pointing to the last element with the biggest assignment costs
1005    // initialization of the hypervolume
1006    hypervolume += (maxRing - allSol[it->second].ringCost)*(maxAssignment -
          allSol[it->second].assignmentCost);
1007    it--; // pointing to the second biggest assignment costs
1008
1009    multimap<long float, int>::iterator it2 = sortA.end();
1010
1011    for (it; it != sortA.begin(); it--)
1012    {
1013      it2--;
1014      hypervolume += (maxRing - allSol[it->second].ringCost)*(allSol[it2->
          second].assignmentCost - allSol[it->second].assignmentCost);
1015    }
1016    // calc relative coverage
1017    hypervolume = hypervolume / (maxRing*maxAssignment);
1018  }
```

```
1019 // **** main function ****
1020 int main()
1021 {
1022   // holds the INDIVIDUALS of next generation
1023   vector<INDIVIDUAL> nextPop;
1024 
1025   // read .txt Data File
1026   readDataFile();
1027 
1028   // start counting the time
1029   clock_t start = clock();
1030   time_t time(NULL);
1031 
1032   // initialize first generation of size N & calculation of all values of a
          solution at once
1033   initPopulation();
1034 
1035   // T generations
1036   for (int t = 0; t < T; t++)
1037   {
1038     vector<INDIVIDUAL> offspring;
1039     vector<INDIVIDUAL> parents_CO;  // parents for cross over
1040 
1041     // create offspring as long number of offspring is lower than N
1042     while (offspring.size() < N)
1043     {
1044       tournament(parents_CO); // includes the procedure of random choosing
              the parents
1045       crossOver(parents_CO, offspring); // includes the mutation operator
              on the offspring
1046       parents_CO.clear();
1047     }
1048 
1049     // merge parent generation with offspring generation
1050     for (int i = 0; i < offspring.size(); i++)
1051     {
1052       population.push_back(offspring[i]);
1053     }
1054 
1055     // calc rank for everyone: 2N chromosomes (population + offspring)
1056     FastNonDominatedSort();
1057     nrFrontsOff.push_back(fronts.size());  // keep track of the number of
          fronts within (population + offspring)
1058     // calc crowding distance for all ranks (2N chromosomes (population +
          offspring))
1059     crowdingDistance();
1060 
1061     selectNextGeneration(nextPop);
1062 
1063     // replace current population with nextPop
1064     population.clear();
1065     for (int p = 0; p < nextPop.size(); p++)
1066     {
```

```
1067            population.push_back(nextPop[p]);
1068        }
1069        nextPop.clear();
1070
1071        // calc rank for everyone for new population of size N
1072        FastNonDominatedSort();
1073        nrFrontsPop.push_back(fronts.size());
1074        // calc crowding distance
1075        crowdingDistance();
1076
1077    }// end t
1078
1079    // stop clock
1080    clock_t stop = clock() - start;
1081    runTime = stop / CLOCKS_PER_SEC;
1082
1083    calcHypervolumen(population, fronts[0], maxR, maxA, hv);
1084
1085    // create .txt File with results
1086    writeDataFile();
1087
1088    system("pause");
1089    getchar();
1090
1091    return EXIT_FAILURE;
1092 }
```