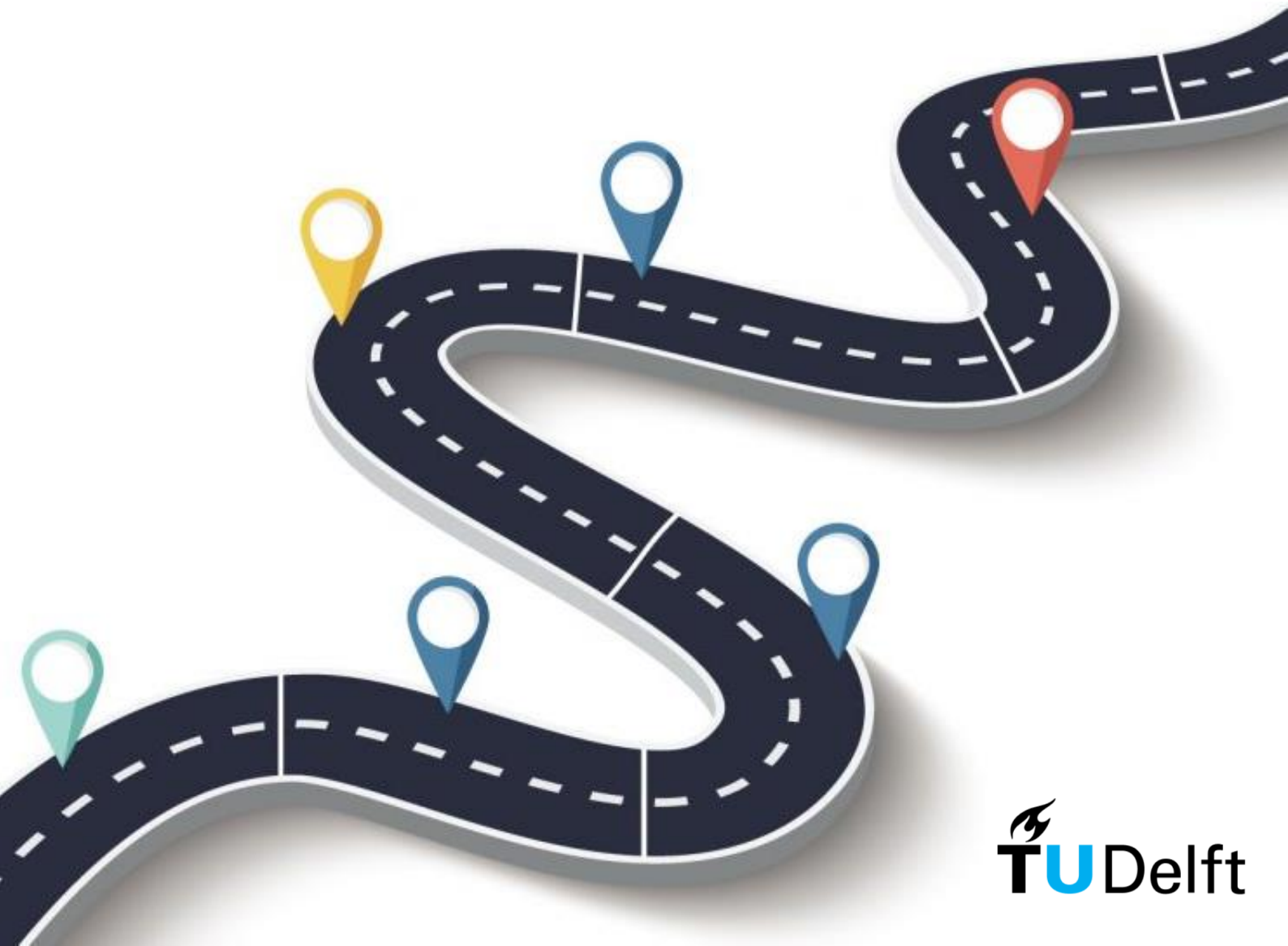
Route optimization for maintaining road quality

F.B.J. Hemler

Supervisor:	Theresia van Essen
Examiner:	Cor Kraaikamp
Faculty:	Applied Mathematics
Date:	22-8-2023



TU Delft

Title figure:
https://onderwijsbureauvanleeuwen.nl/wp-content/uploads/2018/01/coach-route-1-768x543.jpg

Abstract

Velotech Solutions (VTS) is a company specialised in detecting damages in public spaces, such as road damages, non-working light posts or crooked traffic signs. This detection happens by bike or by car, and covers every street in the city or neighbourhood. The routes for navigation are currently created by hand. In this thesis, a proposal is given for the first step of automating this process meeting the requests of Velotech Solutions. A mathematical model is formulated for creating the routes. Firstly, the basic model is formulated to minimize the total distance of all routes that are created. Secondly, minimization of self crossings in every route is added. This is the main request, to prevent routes from becoming too complicated for the navigation devices and cyclists. Thirdly, two solution methods are presented. In the first one, all routes are created at once. In the second one, routes are created one-by-one. The methods are applied to the neighbourhoods Parkwijk and Boeier. From the results, the conclusion is drawn that the first solution method gives the most logical routes. However, the second method, is able to handle larger sets of data, since the solution space is smaller when creating only one route at the time. Lastly, recommendations for further research are given. These include research on the input parameters, the behaviour of the second method on larger datasets and using heurisitcs to solve this problem instead of exact solution methods.

Table of contents

1 Introduction

Without our infrastructure we will get nowhere, literally. It is therefore important that our road network is well maintained. Think about the quality of the road surface, the placement of traffic signs and the operation of light posts. In order to maintain the aforementioned things as efficiently as possible, it is necessary to identify the locations where the roads or public assets are inadequate. The start-up Velotech Solutions has come up with a solution for this.

1.1. The work of Velotech Solutions

Velotech Solutions (VTS) is a company founded by Willem Schellekens and Sjors van Eerten. They provide digital inspections of public assets by bicycle. They have developed hardware and software to detect multiple problems at once when biking along the way. For example: road damage, road markings, damages on traffic signs or light posts, tiltedness of traffic signs or light posts and stickers on traffic signs or light posts. These inspections mostly take place by bike, since this is the most sustainable option. To optimize the assets that can be checked in a certain time period, routes are constructed manually before the cyclist starts. Until now, they were not able to find a routing software that gives routes as output that meet the criteria of Velotech Solutions. Their goal is to be able to create routes automatically, and in the end, to have software that combines all steps of detecting damage, analyzing it and giving an output on how to solve it. In the next section, the problem is further defined, such that we can model this optimization problem mathematically.

1.2. Problem definition

The problem addressed by VTS is a routing problem. In this section, we specify the constraints of VTS for this routing problem.
This problem has multiple objectives, namely:

1. Minimizing the total distance of the routes created
2. Minimizing the number of times a route crosses itself

The first objective is common for routing problems and in navigation services such as google maps this happens automatically. The second objective is unique for this problem. VTS faces two problems that cause them not to use a standard navigation service that only minimizes the distance. Firstly, their software used on the bikes is very newly developed and is hardly able to handle the data when certain locations are visited more than once. Secondly, cyclists do not understand routes that are difficult. By difficult is meant that the routes often go back and forth or contain many turns. Both these problems can be tackled when we do not allow the routes to cross theirselves. However, in practice, this is impossible (dead ends, for example). Therefore, we want to minimize the number of intersections of streets that are visited more than once, in other words: the self crossings per route.
To model this, we need the following input data:

- The streets that need to be checked
- The length of every street that needs to be checked

- The start- and endpoint of the routes
- The maximum length per route
- The maximum number of routes that are created

Once this information is provided, we can create a model to solve the problem of VTS. In Chapter 3, this model is constructed mathematically.

1.3. Thesis outline

The goal of this thesis is to find out whether it is possible to meet the requirements of Velotech Solutions and create a model that automatically creates routes that are useful in practice. In Chapter 2, previous research is discussed. In Chapter 3, the model is formulated mathematically. In Chapter 4, the data generation process and the data used is described. The results are discussed in Chapter 5. In Chapter 6, the conclusion is discussed along with recommendations for further research. In Appendix A, the Python code can be found and in Appendix B the data can be found.

2

Literature review

In this chapter, we review previous literature on this subject. The problem addressed by VTS is an application of a routing problem. Below, previous literature regarding routing problems is discussed.

The General Routing Problem (GRP) as explained in [1] is a problem where a minimum cost tour must be found in a road network that visites specific required vertices and traverses specific required edges. This problem is defined on a connected graph that represents a network of streets. The sets E_R of required edges and V_R of required nodes are defined, which contain the elements that need to be visited by the minimum cost tour. The representation of the nodes and edges is determined by the type of routing problem that is faced.

2.1. Vehicle Routing Problems

In a Vehicle Routing Problem (VRP), the objective is to find multiple minimum cost tours in a network where the nodes represent locations that need to be visited. Well-studied examples of this problem are postal delivery, waste collection and school bus routing. In these problems, the nodes are represented by the post addresses, the places the waste needs to be collected or the houses where students need to be picked up. A certain set of nodes is required, whereas the edges are solely to connect these places. That is, we do not require the vehicles to visit certain edges, hence $E_R = \emptyset$.

2.1.1. Travelling Salesman Problem

The most famous special case of a VRP is the Travelling Salesman Problem (TSP). In the TSP, the nodes of the graph are represented by cities and the edges by the ways connecting them. The objective is to find a minimum cost tour starting and ending in the same city, while visiting all cities. In real life, this problem can be translated to a travelling salesman that sells their product in different cities. The salesman wants to minimize the travelling time or cost by finding a minimum cost tour along the cities. We present a formulation for this problem as stated in [2].
Let n be the number of cities that are required to be visited by the salesman. Let the city from where the tour starts be indexed by 0, which is the base city. For simplicity, we want the salesman to travel all required cities before returning to the base city. That is, it is not allowed to visit city 0 during the tour, only at the beginning and the end. We let d_{ij} be the distance travelled between city i and city j. Let x_{ij} be a binary variable that is

1 when city j is visited directly after city i. We minimize the total distance travelled.

$$\text{Min} \sum_{i=0|i\neq j}^{n} \sum_{j=0,j|i}^{n} d_{ij} x_{ij}$$

Subject to:

$$\sum_{i=0|i\neq j}^{n} x_{ij} = 1, \quad j = 1,\cdots,n \tag{2.1}$$

$$\sum_{j=0|j\neq i}^{n} x_{ij} = 1, \quad i = 1,\cdots,n \tag{2.2}$$

$$\sum_{i=1}^{n} x_{i0} = 1, \tag{2.3}$$

$$u_i - u_j + n x_{ij} \leq n-1, \quad i,j = 1,\cdots,n,\ i \neq j \tag{2.4}$$

$$x_{ij} \in \{0,1\}, \quad i,j = 0,\cdots,n \tag{2.5}$$

$$1 \leq u_i \leq n, \quad i = 1,\cdots,n \tag{2.6}$$

Constraints (2.1) and (2.2) ensure that every city is entered and left exactly once. In Constraint (2.3), this condition is described for the base city, too. Constraints (2.4) are known as the Miller-Tucker-Zemlin or MTZ-constraints. It is a programming-friendly way of ensuring that every city is visited exactly once. This is done with the auxiliary variable u_i. Using this expression, every city (ecxept the base city) receives a "rank". It states that every city that is next visited in the tour, should have a higher rank than the one before. Hence, a city that already has been visited cannot be visited again since its rank will always be lower than the cities that are visited later in the tour [2].

2.1.2. Multiple vehicles

The above formulated model for the TSP creates one route that visits all cities. This is the simplest case of a vehicle routing problem. When a large set of cities (in the case of TSP) or other locations need to be visited, it can be more efficient to use multiple vehicles. That is, creating multiple tours that start and end at the same node. This can be one and the same node for all routes, or this can differ per route. The main objective stays the same: minimizing the cost. When multiple routes are created, this cost can be minimized by minimizing the total cost over all routes. Other ways include for example minimizing the tour of maximum cost [3]. The way chosen depends on the problem that needs to be solved.

2.2. Arc Routing Problems

In contrary to a VRP, in an Arc Routing Problem (ARP), the required set to visit is a set of edges instead of nodes. Again, the goal is to find a least-cost traversal. The applications of ARP's are partly similar to the applications of VRP's. Most ARP's can easily be transformed into VRP's [4]. The main difference lies in the approach of the problem. For example, in school bus routing problems, one can choose to model this as a VRP where the nodes are the locations students need to be picked up. However, when we do not allow the school bus to make u-turns (this can take up a lot of time when turning is difficult), it is more efficient to model this as an ARP. [1] In this case, the nodes represent the crossings and the edges the street sections in between. Whether an edge is a required edge is determined by the presence of a pick-up location on that certain edge. Other examples of applications of ARPs are snow removal, garbage collection, postal delivery and street sweeping. Below two examples of ARPs are explained. These form the basis for our formulation in Chapter 3.

2.2.1. K-Chinese Postman Problem

The Chinese Postman Problem (CPP) was introduced by the Chinese mathematician Meigu Guan and originates from the postal delivery service. It is formulated by Guan (1962): "A mailman has to cover his assigned segment before returning to the post office. The problem is to find the shortest walking distance for the mailman." This problem is defined on a connected graph and the objective is to find a minimum cost tour that travels each edge at least once. This is essentially the same problem as the routing problem of VTS. The K refers to the variant of the problem where K vehicles are available so multiple routes can be created. The K-CPP is a special version of the K- Rural Postman Problem. The difference is that in the K-CPP a minimum cost

route is searched that traverses every edge at least once whereas in the K-Rural Postman Problem (K-RPP) only a subset of the edges needs to be traversed.

2.2.2. K-Windy Rural Postman Problem

As stated in Section 2.2.1, the K-CPP is essentially the K-RPP, but the set of required edges E_R consists of the complete set of edges E. A more general version of this problem is the K-Windy Rural Postman Problem (K-WRPP). In this problem, costs in different directions are allowed to differ. The term "windy" refers to the wind force that makes travelling in one direction easier than in the opposite direction.
In this formulation, for simplicity, it is assumed that every vertex is incident to at least one required edge, which is not a restriction. Let $S, S' \subseteq V$. The set $(S : S')$ denotes the set of edges with one endpoint in S and the other in S'. The set $\delta(S)$ consists of all edges that have a node in S and not in and $E(S) = \{(i, j) \in E : i, j \in S\}$. Now the binary variables x_{ij}^k and x_{ji}^k are defined to represent whether edge $e = (i, j)$ is traversed by vehicle k from i to j or in the opposite direction. If e is a required edge, the variables y_{ij}^k are defined. This variable takes the value 1 if edge e is serviced by vehicle k and 0 otherwise. Note that we make a distinction between traversing an edge and serving an edge. The latter implies that the service that needs to be carried out is finished (for example garbage is picked up, whereas a street can also be traversed without picking it up). In the case of VTS, this distinction is not necessary. The formulation becomes:

$$\text{Minimize} \quad \sum_{k=1}^{K} \sum_{(i,j)\in E} (c_{ij} x_{ij}^k + c_{ji} x_{ji}^k)$$

$$s.t.:$$

$$\sum_{k=1}^{K} y_e^k = 1, \quad \forall e \in E_R \tag{2.7}$$

$$x_{ij}^k + x_{ji}^k \geq y_e^k, \quad \forall e = (i, j) \in E_R,\ k = 1, \cdots, K \tag{2.8}$$

$$\sum_{(i,j)\in\delta(i)} (x_{ij}^k - x_{ji}^k) = 0, \quad \forall i \in V,\ k = 1, \cdots, K \tag{2.9}$$

$$\sum_{(i,j)\in\delta(S)} (x_{ij}^k + x_{ji}^k) \geq 2 y_e^k, \quad \forall S \subset V \setminus \{1\} \text{ with } |E_R(S)| \geq 1,\ e \in E_R(S),\ k = 1, \cdots, K \tag{2.10}$$

$$x_{ij}^k, x_{ji}^k \geq 0 \text{ and integer}, \quad \forall (i, j) \in E,\ k = 1, \cdots, K \tag{2.11}$$

$$y_e^k \in \{0, 1\}, \quad \forall e \in E_R,\ k = 1, \cdots, K \tag{2.12}$$

In the objective function, the total cost of the routes is minimized. Constraints (2.7) ensure that every required edge is serviced by exactly one vehicle. In Constraints (2.8), the vehicles that serve an edge are forced to traverse this edge as well. In Constraints (2.9), connectivity between the starting node and the served edges is ensured and in Constraints (2.10), the subtour elimination constraints are given. This model forms the basis for our formulation in Chapter 3.

3

Methodology

In this chapter, the mathematical formulation for our model is given and substantiated. Furthermore, certain factors that influence the model are discussed.

3.1. Motivation modelling choices

The problem addressed by VTS can be interpreted mathematically in different ways. In this section, we explain why we chose our model formulation as it is.
VTS is a company that delivers a method of collecting data. This method is a service that is usable for different purposes. Although the service can be applied to any purpose in the same way, this is different for creating the routes. Therefore, we make a distinction between creating the routes based on a data set of locations of assets that need to be visited and creating routes without such a data set. The first case is when they need to check street lights, and the second case is the case for all other purposes such as checking the road quality. However, in practice, it is often more efficient to combine these purposes into one set of routes. Therefore, VTS wants to be able to obtain their routes without having to receive a data set of assets from their client. This is a very important condition for our mathematical formulation. We can either construct a model for an arc routing problem or for a vehicle routing problem. Since VTS wants to be able to create routes without data points of assets needed, we choose for an arc routing problem. In this case, the nodes of our graphs are crossings between streets and the edges are the intermediate street sections.
The conditions given by VTS that determine what type of model we use to formulate our problem are the following.

1. Every street in a certain neighbourhood needs to be visited
2. Multiple routes need to be created to cover the whole neighbourhood

These conditions are indicative of a model based on the K-Chinese postman problem, which is, as explained in Chapter 2, a special case of the K-Windy Rural Postman problem. We use both models to formulate our model for VTS. We approach the way of solving the routing problem for VTS in the following two ways:

1. *Construct all routes at once.* This means that the model creates the total number of routes needed at once, and these routes can then be executed by the cyclists.
2. *Construct routes step by step based on what streets have not been visited earlier.* That is, create a route of maximal length, let the cyclist travel this route and then create the next route by going along the streets that have not been travelled yet or that the cyclist has skipped.

In the first approach, the model is based on the K-Chinese Postman Problem, since we want to create multiple routes in an ARP at once and minimize over the total length. Therefore, we first look at this problem. In the second approach, we ask the model to create a route where not all streets have to be traversed. Hence, in this approach, we base our model on the K-Windy Rural Postman Problem. In Chapter 6, we compare the two approaches.

3.2. Mathematical formulation for creating the routes

In this section, the mathematical formulation for the basic model for creating the routes is given.

3.2.1. Formulation of the basic model

Our model is formulated in the same way as in [3]. Let $G = (V, E)$ be an undirected multigraph. Note that in our case most of the streets are indeed undirected, but a small subset is however directed. We can easily fix this by assigning costs to both directions of the edge. Let c_{ij} be the cost of traversing the edge $e = (i, j)$ from i to j. In real life, this cost can be either the time needed to bike through the street section, or the distance between the start and end of a street section. Obviously, we can set $c_{ij} = \infty$ if traversing the edge is not possible in that direction, which is the case in one-way streets. Note that thus every edge $e_{ij} = (i, j)$ is assigned two distinct costs c_{ij} and c_{ji}, but in most cases these costs are equal.
In the K-Windy postman problem, not all edges need to be traversed. Hence, we create the set E_R of edges that are required to traverse, where $E_R \subseteq E$. We want to find a maximum of K routes that have minimum total cost while all edges in E_R are traversed at least once. If it is optimal to create less than K routes, we allow this. Therefore, K is a maximum.
We let x_{ij}^k be a binary variable indicating whether edge e_{ij} is traversed in route k or not. Let y_{ijl}^k be the variable that indicates if edge e_{jl} is traversed directly after edge e_{ij} by route k. We formulate our model in the following way:

$$\text{Min} \quad \sum_{k=1}^{K} \sum_{(i,j)\in E} c_{ij} x_{ij}^k$$

$$s.t.:$$

$$\sum_{k=1}^{K} (x_{ij}^k + x_{ji}^k) \geq 1, \quad \forall (i,j) \in E_R \tag{3.1}$$

$$\sum_{(i,j)\in E} (x_{ij}^k - x_{ji}^k) = 0, \quad \forall i \in V, \forall k = 1, \cdots, K \tag{3.2}$$

$$\sum_{(0,j)\in E} x_{0j}^k = 1, \quad \forall j \in V, \forall k = 1, \cdots, K, \tag{3.3}$$

$$\sum_{(j,l)\in E | j\neq 0} y_{ijl}^k = x_{ij}^k, \quad \forall (i,j) \in E, \forall k = 1, \cdots, K \tag{3.4}$$

$$\sum_{(l,i)\in E | i\neq 0} y_{lij}^k = x_{ij}^k, \quad \forall (i,j) \in E, \forall k = 1, \cdots, K \tag{3.5}$$

$$y_{ijl}^k \leq x_{ij}^k, \quad \forall (i,j) \in E, \forall (j,l) \in E, \forall k = 1, \cdots, K \tag{3.6}$$

$$y_{ijl}^k \leq x_{jl}^k, \quad \forall (i,j) \in E, \forall (j,l) \in E, \forall k = 1, \cdots, K \tag{3.7}$$

$$u_{jl}^k - u_{ij}^k \geq 1 - M(1 - y_{ijl}^k), \quad \forall (i,j) \in E, \forall (j,l) \in E, \ \forall k = 1, \cdots, K \tag{3.8}$$

$$\sum_{(i,j)\in E} c_{ij} x_{ij}^k \leq L_{max}, \quad \forall k = 1, \cdots, K \tag{3.9}$$

$$1 \leq u_{ij}^k \leq n \quad \forall (i,j) \in E, \ i \neq 0, \ \forall k = 1, \cdots, K \tag{3.10}$$

$$u_{ij}^k \geq 0, \text{ integer}; \quad \forall (i,j) \in E, \ i \neq 0, \ \forall k = 1, \cdots, K \tag{3.11}$$

$$x_{ij}^k \in \{0,1\}, \quad \forall (i,j) \in E, \ \forall k = 1, \cdots, K \tag{3.12}$$

$$y_{ijl}^k \in \{0,1\}, \quad \forall (i,j) \in E, \forall (j,l) \in E, \ \forall k = 1, \cdots, n \tag{3.13}$$

The objective function is to minimize the total distance travelled in all the routes. Constraints (3.1) ensure that every street section is travelled at least once among all routes. Constraints (3.2) ensure that every route is connected. In Constraints (3.3), the starting node is added to every route. In Constraints (3.4)-(3.7), an additional auxiliary variable y_{ijl}^k is defined to indicate in which order a route travels along the street sections. In Constraints (3.8), a rank is allocated to every street section for each vehicle, such that no street sections are traversed twice in the same direction within the same route. This is based on the idea of the MTZ-constraints explained in Chapter 2. Input parameter M is used to make the constraint valid for all cases and is defined as

the length of a maximum length route divided by the smallest edge. In this way, M is big enough to not cause problems with this constraint. In Constraints (3.9), every route is prevented to be longer than the maximum length of a route. Lastly, in Constraints (3.10)-(3.13) bounds for the decision variables are defined. In Table 3.1 at the end of this chapter, an overview of all sets, parameters and variables used is given.

3.2.2. Minimizing self crossings

In Subsection 3.2.1, the basic model for the problem of VTS is formulated. Allowing routes to traverse a street only once in the same direction prevents the model from creating confusing and inefficient routes for cyclists to travel. However, this can be further improved by minimizing the number of times a route crosses itself. The main improvement lies in the fact that a route can still go up and down a street, and only the distance becomes longer. Therefore, this might in some cases be more efficient than not travelling up and down a street and taking a detour. However, when we minimize the number of self crossings, this is punished since for travelling up and down a street the corresponding nodes need to be visited multiple times. Note that it is not possible in reality to never travel a street up and down, therefore we do allow it but we punish it in the objective vunction value. Therefore, when minimizing the self crossings, routes will become easier to travel. This is the main request from VTS since the navigation easily fails when routes cross themselves too often. In addition to this, from their experience, the cyclists are more likely to make mistakes in cycling along the route when it crosses itself often.

To include this in the model, we let a_i^k be an integer variable that represents the number of times node $i \in V$ is visited on route k, where a_i^k is greater than or equal to zero. Furthermore, let z_i^k be an integer variable that indicates the number of times node $i \in V$ was visited more than once by route k. We minimize the total sum of z_i^k over all nodes and routes along with the total distance. We apply weighting factor w_C to remain flexible in giving one of the two objectives a priority. The objective function then becomes:

$$\text{Min} \quad \sum_{k=1}^{K} \sum_{(i,j)\in E} c_{ij} x_{ij}^k + w_C \sum_{k=1}^{K} \sum_{i\in V} z_i^k$$

The following constraints are added to the model:

$$a_i^k = \frac{1}{2} \sum_{j\in V|(i,j)\in E} (x_{ij}^k + x_{ji}^k), \quad \forall i \in V, \forall k = 1, \cdots, K \tag{3.14}$$

$$a_i^k \geq z_i^k, \quad \forall i \in V, \forall k = 1, \cdots, K \tag{3.15}$$

$$a_i^k - 1 \leq z_i^k, \quad \forall i \in V, \forall k = 1, \cdots, K \tag{3.16}$$

$$a_i^k \text{ integer}, \; a_i^k \geq 0, \quad \forall i \in V, \; \forall k = 1, \cdots, K \tag{3.17}$$

$$z_i^k \text{ integer}, z_i^k \geq 0, \quad \forall i \in V, \; \forall k = 1, \cdots, K \tag{3.18}$$

Constraints (3.14) define a_i^k as the number of times a node is visited. In Constraints (3.15) and (3.16), z_i^k is linked to a_i^k, by forcing $z_i^k \geq 1$ whenever $a_i^k \geq 2$. Note that Constraints (3.16) are not sufficient, since it allows z_i^k to be -1. Therefore, Constraints (3.15) are needed. Lastly, in Constraints (3.17) and (3.18), the bounds for the variables are given.

3.3. Determining of input parameters

In this section, the input parameters for the model are discussed and how they can be determined.

3.3.1. Starting point

The starting point of the cyclist is the depot where the equipment for the bike needs to be picked up. Therefore, it is fixed. However, since our model creates routes per neighbourhood and not per city, the starting point is most of the time not a fixed node in a given neighbourhood. Therefore, it is the most logical to choose the starting point as the node where the cyclist is most likely to enter the neighbourhood. This matters for the final outcome: suppose we choose the starting node in the middle of the neighbourhood instead of on the edge, we obtain different solutions when we allow multiple vehicles. When only one route is created, the starting point does not change the route. The determination of the number of routes that can be created is explained in the next subsection.

3.3.2. Estimation of K

VTS requests a minimum of two routes to be created. This is, because from their experience, the cyclists never succeed to cycle along all required streets in one route. Either because the route would become too long or because they are confused and skip certain streets. This minimum will almost never be a problem in practice, because for large neighbourhoods or even whole cities, K needs to be big enough to cover the total distance of streets that need to be travelled without exceeding the maximum distance of a route, otherwise the model would be infeasible. It is recommendable to set K quite a bit larger than this lower bound since the model minimizes the total distance. Creating an extra route is therefore in most cases less efficient than extending an existing one.
It is important to note that by increasing K, the solution space increases significantly due to extra constraints and variables. This causes an increase of symmetrical solutions. Two solutions are symmetrical when in one solution an optimal route is assigned to "route 1" and in another to "route 2", where the routes created are the same. Therefore, it is important to set K high enough, but as low as possible.

3.3.3. Weighting factor

The weighting factor has a great impact on the outcome of the model. Note that the value of the sum over the distances is much higher than the value of the sum of the crossings. Therefore, the model will practically ignore the minimization of self crossing in routes and solely focus on the distance minimization when w_C equals 1. When minimizing the distance and crossings with the same priority, it is therefore obvious to set w_C as the average length of the edges in E. However, for VTS, minimizing self crossings in a route has priority. Therefore, a significantly high value for w_C is the most logical. In Chapter 5, the results of different values for the weighting factor is discussed.

3.4. Solution methods

In this section, two solution methods are discussed. After this, a combination of these two is proposed.

3.4.1. Creating all routes at once

The first approach we discuss is creating all routes at once. The above described model follows this approach when we set $E_R = E$. By creating all routes at once, we set K as the maximum number of routes created. The model then decides the optimal number of routes meeting the constraints. The output is a set of routes of which the distance and self crossings in total are minimized.

3.4.2. Creating routes based on previous routes

The second approach we discuss is creating the routes one by one based on previous routes. To reach this approach, we set $K = 1$ and start with $E_R = E$. We create a route larger than a certain minimum distance, while letting the model choose which streets to traverse while meeting the constraints. The next route is then created based on a fixed set of streets E_R. This set contains the streets that have not been traversed earlier. This can either be streets that the model did not choose to add to the first route or streets that should have been traversed but are skipped by the cyclist. This approach changes our model slightly. For creating the first route, we let L_{min} be the minimum length of the route we want to create. Then, we can simply replace Constraint (3.1) from the basic model by:

$$\sum_{(i,j)\in E} c_{ij} x^1_{ij} \geq L_{min}, \tag{3.19}$$

In this way, we make sure that the model creates a route at all. Then, for creating the second route, we run the model again, but now we need to maximize the number of edges traversed in E_R, to prevent the model from making random or empty routes. Let p_{ij} be a binary auxiliary variable. The following constraint ensures that p_{ij} can only equal one when both edges between i and j are not yet removed from E_R.

$$x^1_{ij} + x^1_{ji} \geq p_{ij}, \quad \forall (i,j) \in E_R \tag{3.20}$$

We then add the maximization of the sum of p_{ij} to our objective function, along with a weighting factor:

$$\text{Min} \quad w_D \sum_{(i,j)\in E} c_{ij} x^1_{ij} + w_C \sum_{i\in V} z^1_i - w_E \sum_{(i,j)\in E_R} p_{ij} \tag{3.21}$$

The model created for the second route can be used for the remaining routes that need to be created until the set E_R is empty. Note that in this model we still include Constraint (3.19) to make sure that all streets are traversed. For this it is important to experiment with the length of L_{min} to prevent the model from adding random streets just to reach the minimum length of a route. The output consists of one route at the time where the distance and self crossings are minimized and the number of edges used is maximized per route. Note that we have added two input parameters: L_{min} and w_E. In Chapter 5, certain values are compared and the argumentation for the used values is given. In Table 3.1, all sets, input parameters and decision variables are shown.

Table 3.1: Table of used sets, parameters and variables.

Sets	
V	$= \{1, \cdots, n\}$ set of nodes
E	$= \{e_{ij} = (i, j) : i, j \in V \text{ and } i \neq j\}$ set of arcs
$E_R \subseteq E$	$= \{e \in E : e \text{ is required to be traversed in the optimal route}\}$
Input parameters	
c_{ij}	$= \begin{cases} \text{distance from node } i \text{ and } j, & \text{if } i, j \text{ adjacent} \\ \infty, & \text{if } i, j \text{ non-adjacent} \end{cases}$
L_{max}	= maximum length of route
L_{min}	= minimum length of route
K	= number of routes that need to be created
M	$= \frac{L_{max}}{\min_{i,j \in V} c_{ij} + 1}$
w_C	= weighting factor for minimizing self crossings
w_E	= weighting factor for maximizing edges used from E_R
Decision variables	
x_{ij}^k	$= \begin{cases} 1, & \text{if } (i, j) \in E \text{ is traversed in route } k \\ 0, & \text{otherwise} \end{cases}$
y_{ijl}^k	$= \begin{cases} 1, & \text{if edge } (j, l) \text{ is traversed directly after edge } (i, j) \text{ by vehicle } k \\ 0, & \text{otherwise} \end{cases}$
u_{ij}^k	= rank of edge (i, j) in route k
p_{ij}	= auxiliary variable for maximizing edges used in the routes
z_i^k	= auxiliary integer variable equal to the number of visits of route k to node i more than once
a_i^k	= auxiliary integer variable equal to the number of visits of route k to node i

4

Data

In this chapter, the generation of the data is clarified. For both data sets, the graph was created manually using graphonline.ru. The distances are measured using Afstandmeten.nl and the edges together with these distances are stored in an Excel file.
The choice for neighbourhoods in Lelystad originates from the fact that VTS has already received data from the municipality about the city. Furthermore, Lelystad consists of a lot of small neighbourhoods like Parkwijk that are perfect for testing our model.

4.1. Data generation from OpenStreetMap

The goal of VTS is to automate the creation of routes. Therefore, our model is the first step, however the in- and output is still processed manually. This can be automated by using OpenStreetMap (OSM). In Figure 5.1, Parkwijk is shown when generated via OSM as a graph in Python. However, this graph contains too many paths that are not required to be checked by VTS (footpaths for example). Before running this in the model, it should be simplified. Furthermore, this graph does not contain the distances corresponding to the edges. This is quite a bit harder to obtain, and beyond this bachelor's thesis in mathematics. Therefore, instead of this data, manually created datasets are used in this thesis.

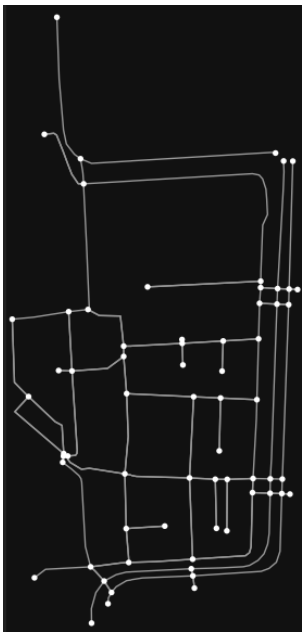
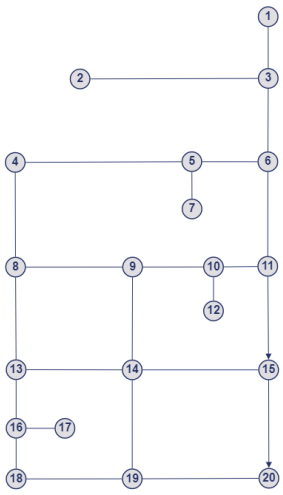


Figure 4.1: Parkwijk automatically generated.

4.2. Parkwijk, Lelystad

This neighbourhood was chosen because it is small and contains all possible complicated situations for the model, such as: a one-way street and dead-end streets. Below a map of Parkwijk is shown together with the corresponding graph.

(a) Parkwijk.



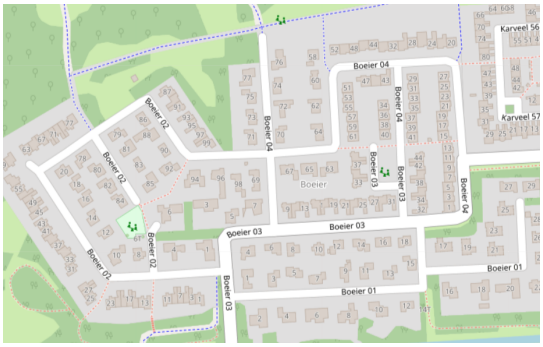
(b) Graph corresponding to Parkwijk.

Figure 4.2: Visualisation of data set.

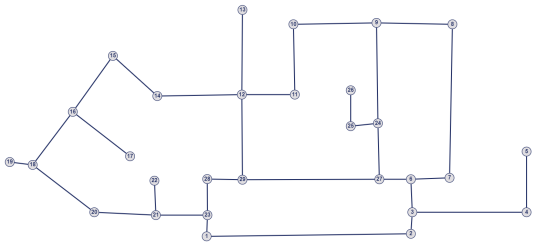
The arrows indicate that it is a one-way street. In the model, this is represented by two edges in opposite direction, with a distance of L_{max} for the edge direction that is not possible.

4.3. Boeier, Lelystad

The second neighbourhood used is Boeier, which is a bit larger. Boeier and its corresponding graph are shown below.



(a) Boeier.



(b) Graph corresponding to Boeier.

Figure 4.3: Visualisation of data set.

5

Results

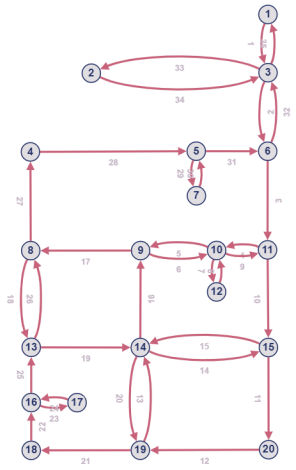
In this chapter, the results from the different models and additions to the models are clarified. Several input values are tested and compared to each other.

5.1. Parkwijk

In this section, we discuss the results achieved by running the first model where the distance is minimized together with the number of times a route crosses itself. The routes are created following the first solution method. As explained in Chapter 3, the choice of K, w_C, w_E and the starting point can influence the outcome significantly. Therefore, we discuss multiple solutions with combinations of different values for these parameters.

5.1.1. Solution method 1: Creating all routes at once

We start with Parkwijk. The starting node is chosen as $v_0 = 1$. For argumentation, see Section 3.3.1. We let $K = 3$. Parkwijk is small as the total distance of the streets together is 1345m, which is far below $L_{max} = 35000$. Therefore, creating two routes should be sufficient. We set $K = 3$ to be sure that it is sufficiently large. For w_C we choose the average length of the streets such that minimizing the number of self crossings and the distance has roughly the same priority. We compare results for $w_C = 1$ and $w_C = 55$. In Figure 5.1, we see the result for $w_C = 1$. The red lines indicate the edges traversed by the route, in the direction of the arrow.

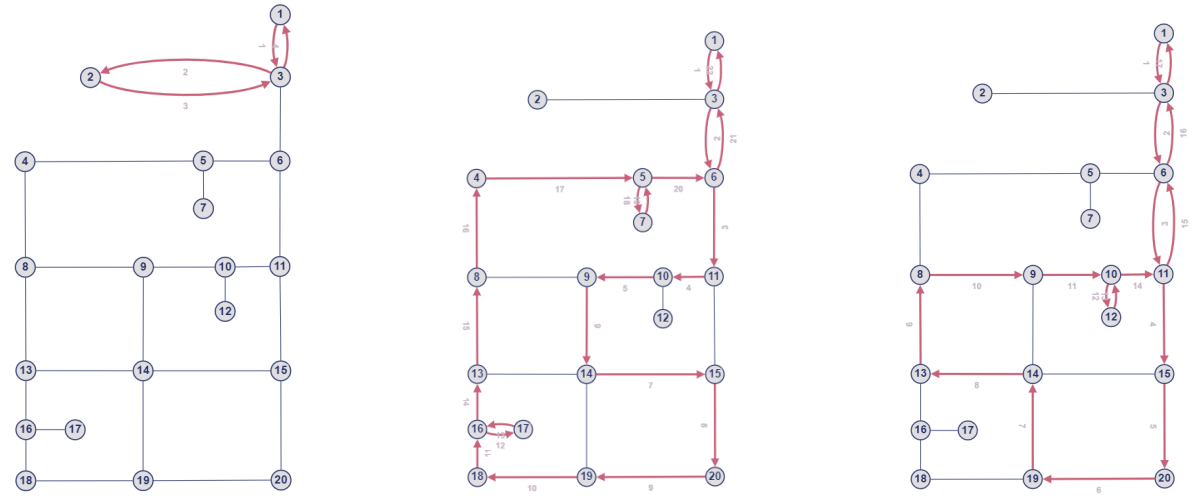


(a) Route 1.

Figure 5.1: Results when distance and number of self crossings are minimized, $K = 3$, $v_0 = 1$, $w_C = 1$.

The numbers indicate the rank of the edge in the route. As we can see, when w_C is set low only one route is created, that crosses itself quite often. This is as expected, since it is more beneficial to minimize the distance than to minimize the number of self crossings.

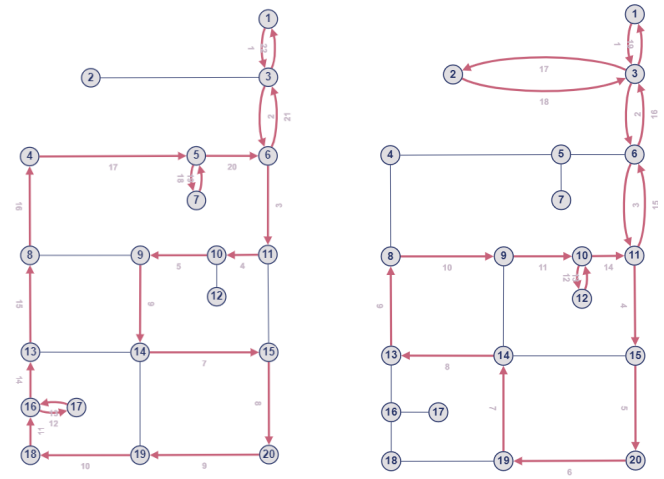
In Figure 5.2, w_C is taken as the average distance of the edges. That is, in the objective function, reducing one of the crossings or adding an edge roughly weighs the same. Now three routes are created, instead of one.



(a) Route 1. (b) Route 2. (c) Route 3.

Figure 5.2: Results when distance and number of self crossings are minimized, $K = 3$, $v_0 = 1$, $w_C = 55$.

Note that nodes 3, 5, 6, 10 and 16 need to be double visited in any situation. As we can see, in this set of routes, no other nodes are visited more than once. Hence, the number of self crossings of the route is minimal. So apparently, in this case, $w_C = 55$ is sufficiently high to meet our requirements. Indeed, all edges are traversed and the routes seems logically constructed: not unnecessarily traversing edges double. Most remarkable is the small route displayed in Figure 5.2a. In practice, this would not be an efficient solution. It would be easier to just let the cyclist of one of the other routes shooting up and down that street rather than installing all equipment all over again on a new bike only to traverse two streets. However, when minimizing the number of self crossings, it is the best option to create an extra route because in distance there is no difference and node 3 is now never visited more than twice by one route. When we change K to 2, we obtain the same results, but now the edges (2,3) and (3,2) are added to one of the routes.



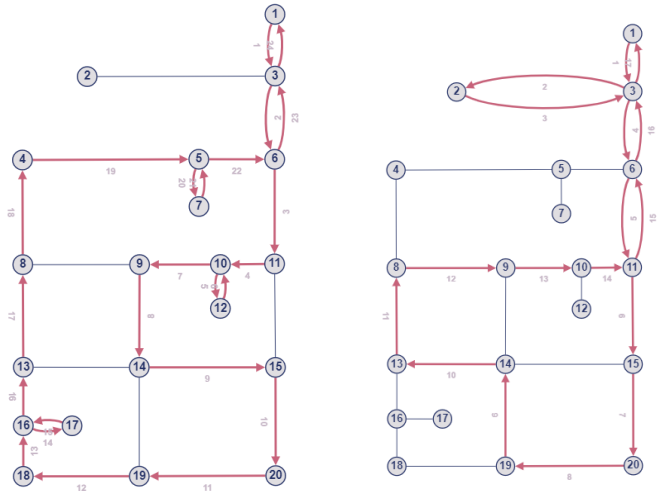
(a) Route 1. (b) Route 2.

Figure 5.3: Results when distance and number of self crossings are minimized, $v_0 = 1$, $w_C = 55, K = 2$.

5.1.2. Solution method 2: Creating routes step-by-step

Next, we look at the second solution method, where one route is created at a time. In this method, new input parameters are added that influence the outcome: L_{min} and w_E. The constraint where L_{min} is used

is not necessary when w_E is sufficiently high. However, when we set w_E too high, removing edges from E_R has priority over minimizing self crossings and distance of a route. Then, it is optimal to use as much edges as possible in one route without giving the minimization of the number of self crossings and distance enough priority. But since the constraint for traversing each street at least once is removed in this solution method, if we set w_E too low, not all edges are traversed after all routes are created. Therefore, by adding the constraint with L_{min}, we force the model to create routes of a minimum length so that we can find a balance between these input parameters more easily. We want to choose L_{min} sufficiently large, otherwise a cyclist is done way too soon. However, if we choose L_{min} too high, we force the route to add edges that have been traversed before, which are only useless extra travelled meters. It turns out that $L_{min} = 1000$ works best for this dataset. Now we set w_E to 100. Since $w_C = 55$, it is now twice as important to traverse all edges compared to minimizing the crossings. When choosing w_E lower than this, not all edges are traversed since too much priority is given to minimizing distance and number of self crossings. The outcome is that two routes are created as shown in Figure 5.2.



(a) Route 1. (b) Route 2.

Figure 5.4: Results when distance and number of self crossings are minimized, $v_0 = 1$, $w_C = 55$, $w_E = 100$, $L_{min} = 1000$.

As we can see, the routes created are the same as in the first method. The number of crossings is minimal and no edges are unnecessarily traversed by both routes.

5.1.3. Comparison of solution methods

Both methods have practically the same output. However, in the first method, we needed to adjust K to obtain the best applicable set of routes in practice, whereas in the second method automatically the optimal routes were created. However, in the second method, choosing the right weighting factors is important, whereas in the first method only the input for K matters significantly.

5.2. Boeier

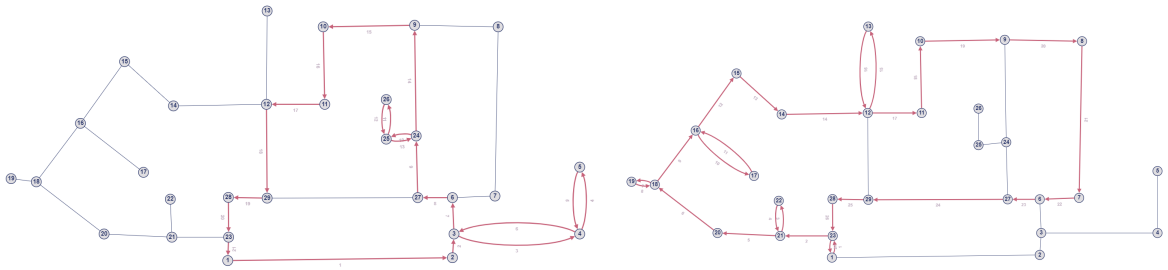
In this section, we compare the results of the solution methods for the neighbourhood Boeier.

5.2.1. Solution method 1: Creating all routes at once

The input parameters are set the same as for Parkwijk. We choose $w_C = 69$ as the average of the length of streets again. Since the size of Boeier is comparable to that of Parkwijk, we let $K = 3$. In Figure 5.5, the results are displayed.

Note that no nodes are visited double unnecessarily in one route, except for node 23 in Figure 5.5b. When we increase w_C to 200, we obtain the results shown in Figure 5.6.

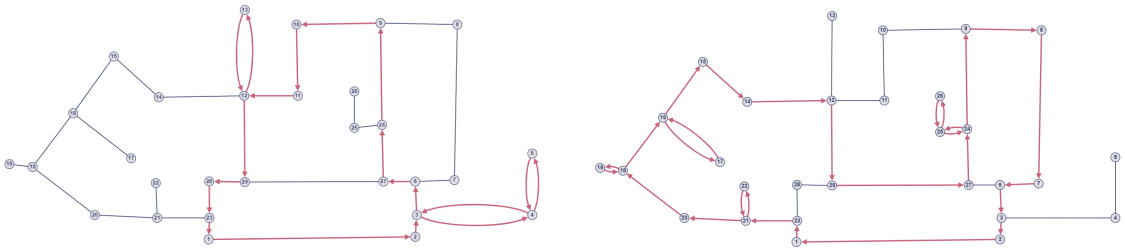
As we can see, the routes differ slightly, and indeed, no nodes are crossed double unnecessarily in the second set of routes. The total meters travelled when $w_C = 69$ is 3165m whereas for $w_C = 200$ this is 3282m. One can argue that it is more efficient for the route to cross itself one time too many rather than adding 100 meters to the route. On the contrary, if this causes the navigation equipment to crash, it is of course better to



(a) Route 1.

(b) Route 2.

Figure 5.5: Results when distance and number of self crossings are minimized, $v_0 = 1$, $w_C = 69$.



(a) Route 1.

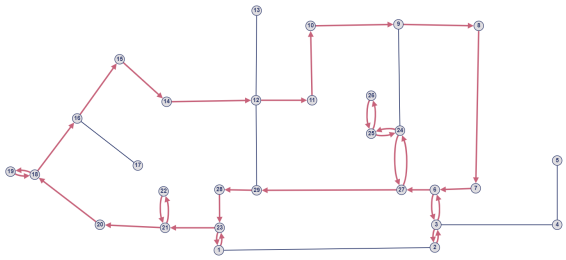
(b) Route 2.

Figure 5.6: Results when distance and number of self crossings are minimized, $v_0 = 1$, $w_C = 200$.

take a 100 meter detour.

5.2.2. Solution method 2: Creating routes step-by-step

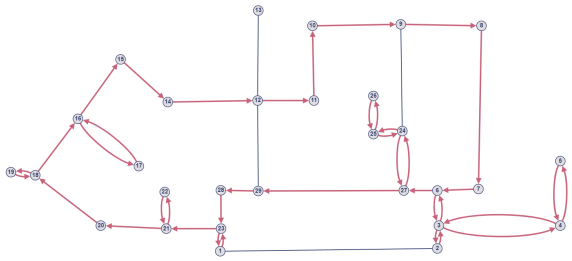
For the second solution method, we first need to determine the value of the input parameters again. Since Boeier and Parkwijk are roughly of the same size, we let $L_{min} = 1000$. That means that every route created has to be at least 1000m. Now to begin with, we keep w_C and w_E the same. The output consists of as many as four routes. This cannot be optimal, since in the previous section, we have shown that two routes should be sufficient to meet our constraints. For the sake of completeness, the first of 4 routes is shown in Figure 5.7. It immediately stands out that other streets than dead ends are traversed double.



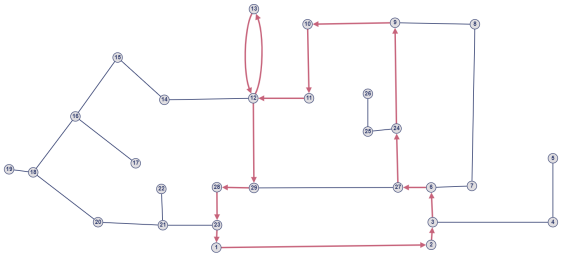
(a) Route 1.

Figure 5.7: Results when distance and number of self crossings are minimized, $v_0 = 1$, $w_C = 69$, $w_E = 100$, $L_{min} = 1000$.

We now increase w_E to 150, which means more priority is given to traversing all edges from E_R in one route. Note that lowering L_{min} does not work, since then some edges are never reached because it benefits more to not add those since our route can be small. Therefore w_E has to be increased. Now two routes are created as shown in Figure 5.8. The first route is practically the route created before, but now the remaining



(a) Route 1.



(b) Route 2.

Figure 5.8: Results when distance and number of self crossings are minimized, $v_0 = 1$, $w_C = 200$, $w_E = 150$, $L_{min} = 1000$.

edges are put together in one route instead of three.

5.2.3. Comparison of solution methods

In contrast to our conclusion for Parkwijk, the routes created in the first solution method are better than the ones in the second solution method. The second set of routes created contains more self crossings and has a greater total distance.

5.3. Combining the solution methods

We conclude that the first method works best for creating the routes, although the right value for K has to be found by trial and error, to prevent the model from creating routes of too few streets. On the contrary, in the second solution method, we risk adding too many edges to one route which reduces the quality of the route because the value for w_E has to be sufficiently large. Therefore, the proposition for the method used in practice is to combine both methods in the following way:

1. Use the first solution method to gain routes where the total distance and number of crossings is minimal.
2. Execute the longest route created by this method and remove the edges from the set E_R. By removing the longest route from E_R the set is as small as possible after removing one route from the first solution method. Moreover, by executing the longest route, the chances of mistakes are the highest, therefore these mistakes can be included immediately by creating the following routes. After this, the remaining routes are created by the second solution method.
3. Apply the second solution method.

The results for both Parkwijk and Boeier are as expected. For Parkwijk, the routes in Figure 5.4 are created, but now at once instead of having to adjust K. For Boeier, the results are the same as in Figure 5.6. Therefore, combining the two methods is a way of finetuning the route generation. Moreover, adjustments to the set E_R can be made after every generation of a new route. In this way, mistakes made by the cyclists or navigation can be included in creating the next route.

6

Conclusion

The goal of this thesis was to automate the creation of routes for Velotech Solutions while meeting their requirements. Three methods for constructing these routes were implemented and compared.

The first solution method creates routes where the number of self crossings together with the distance is minimized. A weighting factor w_C for the number of self crossings with a value equal to the average of the length of the edges for w_C results in solutions with a good balance between these two objectives. However, it may appear that the above mentioned value for w_C is too high, and redundant routes are created. It suffices then to reduce the value for K. In this thesis, K was determined by reviewing the data set and estimating the number of routes needed based on this. For safety's sake, this estimation was chosen on the high side. This increases the computing time significantly. A recommendation for further research would therefore be to analyze what values for the number of routes K and w_C are optimal for a specific data set. Moreover, research on when a solution is exactly optimal is also interesting for in the future. For small data sets like Parkwijk and Boeier this is easily determined intuitively, but for larger datasets this is more difficult.

The second solution method can handle slightly larger data sets due to the fact that $K = 1$, which means that not all routes are created at once. However, since the constraint for traversing every edge is removed, the number of unused edges per route must be maximized. This influences the kind of routes that are created. Adding unused edges to a route receives priority over minimizing crossings and distance. Therefore, the routes are not optimal, in comparison to the first method. Further research is therefore to be done on the input values for w_C, w_E and the value for L_{min}. This does however not mean that this method is entirely useless. When constructing the routes one by one, after every route adjustments to the set E_R can be made manually. That is, if a cyclist made mistakes in following the route, this can be included before creating the next route.

The third solution method is proposed lastly and is a combination of the two mentioned above. By applying the first solution method at first and finetuning this with the second solution method we obtain realistic routes that can be adjusted in between the execution of the checks of VTS. However, the same problems as discussed for both solution methods remain.

The automation of the route creation for VTS has not succeeded in the way hoped. It turns out that the automatic generation of data into the right format is difficult. This is definitely a subject for further research, probably in the field of computer science. Furthermore, both models cannot handle large datasets. This is caused by the fact that we use an exact solution method. Further research could be done on which heuristics can be used to solve the problem of VTS such that it is a useful approach in practice.

We can conclude that both methods work, whereas the first method gives better routes in terms of minimal distance and self crossings. The second method however, is able to process larger data sets and gives - depending on the input parameters- acceptable solutions. Therefore, the recommendation for VTS based on this thesis is to combine the two solution methods as explained in Section 5.3.

7

References

[1] Eglese, R. W., Letchford, A. N. (2008b). General routing problem. In Springer eBooks (pp. 1252–1254). https://doi.org/10.1007/978-0-387-74759-$0_2$17

[2] Miller, C. E., Tucker, A. W., Zemlin, R. A. (1960b). Integer programming formulation of traveling salesman problems. Journal of the ACM, 7(4), 326–329. https://doi.org/10.1145/321043.321046

[3] Benavent, E., Corberán, Á., Plana, I., Sanchis, J. M. (2009). Min-Max K -vehicles windy rural postman problem. Networks, 54(4), 216–226. https://doi.org/10.1002/net.20334

[4] Gordenko, M., Avdoshin, S. (2018). The Variants of Chinese Postman Problems and Way of Solving through Transformation into Vehicle Routing Problems. Trudy Instituta Sistemnogo Programmirovaniâ, 30(3), 221–232. https://doi.org/10.15514/ispras-2018-30(3)-16

Appendices

A

Appendix A: Python code

A.1. Solution method 1

Listing A.1: Python Code solution method 1

```
from pulp import *
import math
import numpy as np
import pandas as pd
import networkx as nx

#############################################################################
data = pd.read_excel(r"C:\Users\fjehe\OneDrive\Documenten\BEP\Test data .xlsx")
Start_node = pd.DataFrame(data, columns=['Start'])
Dest_node = pd.DataFrame(data, columns=['Dest'])
Dist_edge = pd.DataFrame(data, columns=['Dist'])
distance_list = []
id_list = []
start_id_list = []
dest_id_list = []
dist_dict = {}
totaldist = 0
v0 = 1

#Parkwijk
##for i in range(1,21):
##    id_list.append(i)

#Boeier
for i in range(1,30):
    id_list.append(i)

n = len(id_list)

for i in range(0,len(data)):
    edge = (data['Start'][i], data['Dest'][i])
    dist_dict[edge] = data['Dist'][i]
    if dist_dict[edge] != 10000:
        totaldist += dist_dict[edge]


#############################################################################################
#LP Problem
#INPUT
K = 3#int(input('How many routes do you want to create? '))
```

```
L_max = 35000 #int(input('What is the maximum length in meters per route? '))
#M = L_max/25 + 1 #Parkwijk
M = L_max/11 + 1 #Boeier
wD = 1
wC = 200


#define problem
problem = LpProblem('K-CPP', LpMinimize)

#VARIABLES
x = LpVariable.dicts('x', ( range(n+1), range(n+1), range(K) ), 0, 1, LpBinary)
y = LpVariable.dicts('y', ( range(n+1), range(n+1), range(n+1), range(K) ), 0, 1, LpBinary)
u = LpVariable.dicts('u', ( range(n+1), range(n+1), range(K) ), 0, 2*n, LpInteger)
a = LpVariable.dicts('a', ( range(n+1), range(K) ), 0, n, LpInteger)
z = LpVariable.dicts('z', ( range(n+1), range(K) ), 0, n, LpInteger)

#OBJECTIVE
problem += wD*lpSum([ dist_dict[(i,j)] * (x[i][j][k]) for (i,j) in dist_dict for k
in range(K) ])
+ wC*lpSum([ z[i][k] for i in range(n+1) for k in range(K) ])



#CONSTRAINTS
for (i,j) in dist_dict:
    problem += lpSum([x[i][j][k] + x[j][i][k] for k in range(K)]) >= 1 #visit every edge
    at least once

#connectivity: every route consists only of cycles
for k in range(K):
    for i in range(n+1):
        problem += lpSum([x[i][j][k] - x[j][i][k] for j in range(n+1) if (i,j) in dist_dict ]
        == 0 #connectivity
    problem += lpSum([x[v0][j][k] for j in range(n+1) if (v0,j) in dist_dict ]) == 1
    #v0 is included in a cycle
    problem += lpSum([x[j][v0][k] for j in range(n+1) if (j,v0) in dist_dict ] ) == 1

#y can only be 1 if both x are 1, otherwise 0
for k in range(K):
    for (i,j) in dist_dict:
        if j != 1:
            problem += lpSum([ y[l][j][i][k] for l in range(1,n+1) if (l,j) in dist_dict ])
            == x[j][i][k]
            problem += lpSum([ y[i][j][l][k] for l in range(1,n+1) if (j,l) in dist_dict])
            == x[i][j][k]
            for l in range(n+1):
                if (j,l) in dist_dict:
                    problem += y[i][j][l][k] <= x[i][j][k]
                    problem += y[i][j][l][k] <= x[j][l][k]

#route continuity and penalizing subtours based on edge visit ipv nodes
for k in range(K):
    for (i,j) in dist_dict:
        for l in range(n+1):
            if (j,l) in dist_dict:
                problem += u[j][l][k]- u[i][j][k] >= 1- M*(1 - y[i][j][l][k])
                #subtour elimination using edges

#max length of route
for k in range(K):
```

```
    problem += lpSum([dist_dict[(i,j)]*(x[i][j][k]) for (i,j) in dist_dict]) <= L_max

#define a
for k in range(K):
    for i in range(n+1):
            problem += 2*a[i][k] == lpSum([x[j][i][k] + x[i][j][k] for j in range(n+1) ])

#define z
for k in range(K):
    for i in range(n+1):
        problem += a[i][k] >= z[i][k]
        problem += a[i][k] - 1 <= z[i][k]

#solve and print
problem.solve(GUROBI(timeLimit=5))
route_length = 0
for k in range(K):
    for (i,j) in dist_dict:
        if value(x[i][j][k])>=1:
            print(str(i) + '_' + str(j) + '_' + str(k) + ': ' + str(value(x[i][j][k])) +
            ' u is '+str(value(u[i][j][k]))  )
            if dist_dict != L_max:
                route_length += dist_dict[(i,j)]

for i in range(n+1):
    print('a_'+ str(i)+' is '+str(value(a[i][k]))+
    ' and z_'+str(i)+' is '+str(value(z[i][k])))
print('Total length of routes should be '+str(route_length) + ' m')
print('Total number of edges: '+str(len(dist_dict)))
print('Total length of streets: '+ str(totaldist) + ' m')
print('Total length of routes: ' + str(value(problem.objective)) + ' m')
print('Status: '+ str(problem.status))
```

A.2. Solution method 2

Listing A.2: Python Code Solution method 2

```
from pulp import *
import math
import numpy as np
import pandas as pd
import networkx as nx

#############################################################################
data = pd.read_excel(r"C:\Users\fjehe\OneDrive\Documenten\BEP\Test data .xlsx")
Start_node = pd.DataFrame(data, columns=['Start'])
Dest_node = pd.DataFrame(data, columns=['Dest'])
Dist_edge = pd.DataFrame(data, columns=['Dist'])
distance_list = []
id_list = []
start_id_list = []
dest_id_list = []
dist_dict = {}
dist_dict_copy = {}
totaldist = 0
v0 = 1

#Parkwijk
##for i in range(1,21):
##    id_list.append(i)
```

```
#Boeier
##for i in range(1,30):
##    id_list.append(i)
##
n = len(id_list)

for i in range(0,len(data)):
    edge = (data['Start'][i], data['Dest'][i])
    dist_dict[edge] = data['Dist'][i]
    dist_dict_copy[edge] = data['Dist'][i]
    if dist_dict[edge] != 10000:
        totaldist += dist_dict[edge]

for (i,j) in dist_dict:
    if dist_dict[(i,j)] == 35000:
        del dist_dict_copy[(i,j)]

#############################################################################################
#LP Problem
#INPUT
K = 1#int(input('How many routes do you want to create? '))
L_max = 3500 #int(input('What is the maximum length in meters per route? '))
#M = L_max/25 + 1 #Parkwijk
M = L_max/12 + 1 #Boeier
wD = 1
wC = 200
wE = 1000
L_min = 0
count = 0

#define problem
problem = LpProblem('K-CPP', LpMinimize)
problem2 = LpProblem('WRPP', LpMinimize)
#VARIABLES
x = LpVariable.dicts('x', ( range(n+1), range(n+1), range(K) ), 0, 1, LpBinary)
y = LpVariable.dicts('y', ( range(n+1), range(n+1), range(n+1), range(K) ), 0, 1, LpBinary)
u = LpVariable.dicts('u', ( range(n+1), range(n+1), range(K) ), 0, 2*n, LpInteger)
a = LpVariable.dicts('a', ( range(n+1), range(K) ), 0, n, LpInteger)
z = LpVariable.dicts('z', ( range(n+1), range(K) ), 0, n, LpInteger)
p = LpVariable.dicts('p', ( range(n+1), range(n+1) ), 0, 1, LpBinary)


#objective function
problem += wD*lpSum([ dist_dict[(i,j)] * (x[i][j][k]) for (i,j) in dist_dict
for k in range(K) ])
+ wC*lpSum([ z[i][k] for i in range(n+1) for k in range(K) ]) - wE*lpSum([p[i][j]
for (i,j) in dist_dict])


#for creating the first route, use the following constraint
for k in range(K):
    problem += lpSum([ dist_dict[(i,j)]*(x[i][j][k]) for (i,j) in dist_dict ]) >= L_min

for k in range(K):
    for i in range(n+1):
        #connectivity
        problem += lpSum([x[i][j][k] - x[j][i][k] for j in range(n+1) if (i,j) in dist_dict
        == 0
        #v0 is included in every route
    problem += lpSum([x[v0][j][k] for j in range(n+1) if (v0,j) in dist_dict ]) == 1
```

```
    problem += lpSum([x[j][v0][k] for j in range(n+1) if (j,v0) in dist_dict ] ) == 1

#define y
for k in range(K):
    for (i,j) in dist_dict:
        if j != 1:
            problem += lpSum([ y[l][j][i][k] for l in range(1,n+1) if (l,j) in dist_dict ])
            == x[j][i][k]
            problem += lpSum([ y[i][j][l][k] for l in range(1,n+1) if (j,l) in dist_dict])
            == x[i][j][k]
            for l in range(n+1):
                if (j,l) in dist_dict:
                    problem += y[i][j][l][k] <= x[i][j][k]
                    problem += y[i][j][l][k] <= x[j][l][k]

#traversing every edge once & subtour elimination
for k in range(K):
    for (i,j) in dist_dict:
        for l in range(n+1):
            if (j,l) in dist_dict:
                problem += u[j][l][k]- u[i][j][k] >= 1- M*(1 - y[i][j][l][k])

#max distance of route
for k in range(K):
    problem += lpSum([dist_dict[(i,j)]*(x[i][j][k]) for (i,j) in dist_dict]) <= L_max

#define a
for k in range(K):
    for i in range(n+1):
        problem += 2*a[i][k] == lpSum([x[j][i][k] + x[i][j][k] for j in range(n+1) ])

#koppel z aan a
for k in range(K):
    for i in range(n+1):
        problem += a[i][k] >= z[i][k]
        problem += a[i][k] - 1 <= z[i][k]

#definieer p zo dat: maximaliseer edges uit E_R om langs te gaan
for k in range(K):
    for i in range(n+1):
        for j in range(n+1):
            if (i,j) in dist_dict_copy:
                problem += x[i][j][k] + x[j][i][k] >= p[i][j]
                #problem += x[i][j][k] + x[j][i][k] >= p[j][i]

while dist_dict_copy != {} and count <= 5:
    count += 1
    #define new objective with maximization of p added
    problem2 += wD*lpSum([ dist_dict[(i,j)] * (x[i][j][k]) for (i,j) in dist_dict
    for k in range(K) ]) + wC*lpSum([ z[i][k] for i in range(n+1) for k in range(K) ])
    - wE*lpSum([p[i][j] for (i,j) in dist_dict_copy])

    for k in range(K):
        problem2 += lpSum([ dist_dict[(i,j)]*(x[i][j][k]) for (i,j) in dist_dict ])
        >= L_min


    for k in range(K):
        for i in range(n+1):
            problem2 += lpSum([x[i][j][k] - x[j][i][k] for j in range(n+1) if (i,j)
            in dist_dict ])
```

```
        == 0 #connectivity
    problem2 += lpSum([x[1][j][k] for j in range(n+1) if (v0,j) in dist_dict ]) == 1
    #v0 is included in a cycle
    problem2 += lpSum([x[j][1][k] for j in range(n+1) if (j,v0) in dist_dict ] ) == 1

for k in range(K):
    for (i,j) in dist_dict:
        if j != 1:
            problem2 += lpSum([ y[l][j][i][k] for l in range(1,n+1) if
            (l,j) in dist_dict ])
            == x[j][i][k]
            problem2 += lpSum([ y[i][j][l][k] for l in range(1,n+1) if
            (j,l) in dist_dict])
            == x[i][j][k]
            for l in range(n+1):
                if (j,l) in dist_dict:
                    problem2 += y[i][j][l][k] <= x[i][j][k]
                    problem2 += y[i][j][l][k] <= x[j][l][k]

for k in range(K):
    for (i,j) in dist_dict:
        for l in range(n+1):
            if (j,l) in dist_dict:
                problem2 += u[j][l][k]- u[i][j][k] >= 1- M*(1 - y[i][j][l][k])

for k in range(K):
    problem2 += lpSum([dist_dict[(i,j)]*(x[i][j][k]) for (i,j) in dist_dict]) <= L_max

for k in range(K):
    for i in range(n+1):
        problem2 += 2*a[i][k] == lpSum([x[j][i][k] + x[i][j][k] for j in range(n+1) ])

for k in range(K):
    for i in range(n+1):
        problem2 += a[i][k] >= z[i][k]
        problem2 += a[i][k] - 1 <= z[i][k]

for k in range(K):
    for i in range(n+1):
        for j in range(n+1):
            if (i,j) in dist_dict_copy:
                problem2 += x[i][j][k] + x[j][i][k] >= p[i][j]
                #problem2 += x[i][j][k] + x[j][i][k] >= p[j][i]


#add constraint for p
for k in range(K):
    for (i,j) in dist_dict_copy:
        problem2 += x[i][j][k] + x[j][i][k] >= p[i][j]

#solve problem again
problem2.solve(GUROBI())
#delete used edges from edge dictionary
for k in range(K):
    for (i,j) in dist_dict:
        if value(x[i][j][k]) >= 1:
            print(str(i) + '_' + str(j) + '_' + str(k) + ': ' + str(value(x[i][j][k]))
            + ' u is '+str(value(u[i][j][k]))  )
            #als xij =1 dan (i,j) en (j,i) weghalen uit E_R
            if (i,j) in dist_dict_copy:
                del dist_dict_copy[(i,j)]
```

```
                if (j,i) in dist_dict_copy:
                    del dist_dict_copy[(j,i)]

    print(dist_dict_copy)
    for (i,j) in dist_dict_copy:
        print("p_"+str(i)+"_"+str(j)+ " is "+str(value(p[i][j])))

problem2.writeLP("Problem 2")
problem.writeLP("Problem")
```

B

Appendix B: Data

Start	Dest	Dist
1	3	25
2	3	107
3	1	25
3	2	107
3	6	56
4	5	95
4	8	45
5	4	95
5	6	33
5	7	28
6	5	33
6	3	56
6	11	58
7	5	28
8	4	45
8	9	64
8	13	74
9	8	64
9	10	26
9	14	76
10	9	26
10	11	35
10	12	45
11	10	35
11	6	58
11	15	71
12	10	45
13	8	74
13	14	63
13	16	52
14	9	76
14	13	63
14	15	60
14	19	76
15	14	60
15	11	35000
15	20	71
16	13	52
16	17	38
16	18	32
17	16	38
18	16	32
18	19	62
19	14	76
19	18	62
19	20	53
20	19	53
20	15	35000

(a) Parkwijk

Start	Dest	Dist
1	2	192
1	23	25
2	1	192
2	3	16
3	2	16
3	4	99
3	6	50
4	3	99
4	5	63
5	4	63
6	3	50
6	7	46
6	27	18
7	6	46
7	8	156
8	9	58
8	7	156
9	8	58
9	10	65
9	24	119
10	9	65
10	11	76
11	10	76
11	12	66
12	11	66
12	13	116
12	14	82
12	29	73
13	12	116
14	12	82
14	15	70
15	14	70
15	16	67
16	15	67
16	17	81
16	18	67
17	16	81
18	16	67
18	19	12
18	20	125
19	18	12
20	18	125
20	21	48
21	20	48
21	22	49
21	23	74
22	21	49
23	1	28

(b) Boeier

Start	Dest	Dist
23	21	74
23	28	39
24	9	119
24	25	27
24	27	36
25	24	27
25	26	39
26	25	39
27	6	18
27	24	36
27	29	128
28	23	39
28	29	47
29	12	79
29	27	128
29	28	47

(c) Boeier

Figure B.1: Data of Parkwijk and Boeier