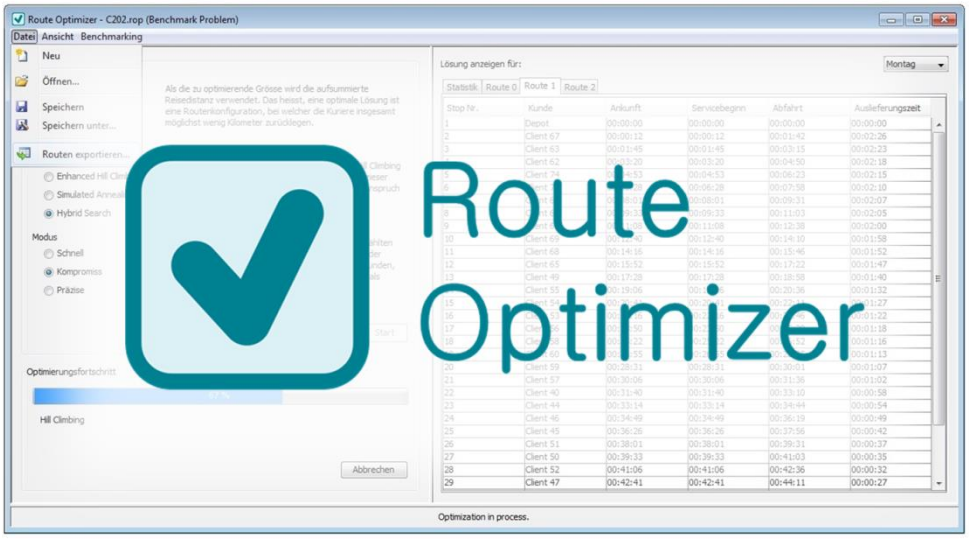
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Kartografie und Geoinformation
Institute of Cartography and Geoinformation

The Vehicle Routing Problem with Time Windows applied to a real-world example

Route optimisation for the Zentrum für Labormedizin St. Gallen



Semester:	6. Semester
Course of studies:	BSc Geomatic Engineering and Planning
Type of work:	Bachelor's thesis
Author:	Katharina Henggeler katharina.henggeler@hispeed.ch khenggel@student.ethz.ch
Supervisor:	Prof. Dr. Martin Raubal
Advisors:	Dominik Bucher, Ioannis Giannopoulos
Submission date:	29.05.2015

Acknowledgements

First, I wish to give my thanks to Prof. Dr. med. Wolfgang Korte and Heinz Ryffel from the Zentrum für Labormedizin St. Gallen for having given me the opportunity to write this thesis. They have generously provided me with all the data I needed and took time off from their busy schedules to answer all the questions I had.

A special thank you goes to my advisors Dominik Bucher and Ioannis Gianopoulos from the ETH Institute of Cartography and Geoinformation. Without their kind assistance and expert support, this thesis would have not been possible.

I would like to thank Prof. Dr. Martin Raubal for supervising this thesis. As the topic for this bachelor's thesis was my own suggestion, its aim was still vague at the beginning. I appreciated that this thesis has been possible despite that.

I owe my gratitude to Gillian Lienhard. She generously offered to proofread this thesis even though time was short. I thank her a lot for her effort.

Last but not least this thesis would not have been possible without the support of my family and my mother and father especially. They put up with my late-night shifts, cheered me up when I got stuck, and provided me with the best writing environment possible, thus allowing me to fully concentrate on this thesis.

Abstract

Instances of the Vehicle Routing Problem with Time Windows (VRPTW) can be found everywhere in the real world. But most of the time they show certain variations compared to the abstract mathematical definition of the problem. This thesis analyses such a real-world instance and proposes a possible implementation of an application in Java. It discusses the optimisation algorithms used, as well as how the additional constraints of this special variation can be modelled to fit into the standard definition of the VRPTW. The structure of the application and the design of the user interface are illustrated and explained. Finally, ideas to further improve the application are proposed and shortly discussed.

Contents

List of figures and tables

1 Introduction

The Vehicle Problem with Time Windows (VRPTW) is a special case of the Vehicle Routing Problem (VRP) where each customer additionally has a time window during which service needs to take place. The VRP describes the problem of having a set of customers and a fleet of vehicles that need to deliver a certain service to each of these customers exactly once. All vehicles must start at a unique depot and after completing their route return to that same depot again. Additionally, each customer specifies the goods it demands and each vehicle has a limited capacity. In the most basic case all vehicles have the same capacity. The objective is then to minimise the total distance travelled and the number of vehicles needed.

As the VRPTW is a NP-hard combinatorial optimisation problem, determining the optimal solution deterministically and in a reasonable amount of time is only possible for reduced customer numbers. Hence, for more complex problems heuristics and meta-heuristics are used to find approximate solutions (de Oliveira, et al., 2008). As of today, a lot of research has been done regarding the VRPTW and a variety of approaches have been proposed for solving the VRPTW. For example, Genetic Algorithms, Simulated Annealing, and Tabu Search methods have been researched with the aim to minimise travel distance (Thangiah, et al., 1994).

In the last decades, the optimisation of such problem instances has become increasingly necessary, especially with the increasing mobility of people and goods and the logistic expenses that grow along with it. Studies suggest that 10 – 15 % of the final value of traded goods correlates to its transportation costs (de Oliveira, et al., 2008).

An institution that was confronted with that exact problem gave the impetus for this thesis. The institution, the Zentrum für Labormedizin St. Gallen (ZLMSG), provides a courier service for clients that delivers consumables and retrieves samples that are to be analysed by the ZLMSG. The client list includes hospitals and medical practices as well as a few other laboratories. Their spatial distribution is roughly shown in Figure 1. At the moment, there are about 30 clients in total; however, the ZLMSG would be happy to expand. The ZLMSG has employed three couriers, each covering more or less the same route every day. Each of these routes has been manually calculated using google maps. The criteria used for this calculation was the total distance travelled by all couriers and the number of couriers needed. However, there is no guarantee that a solution obtained from such a calculation is optimal, especially when a new client needs to be integrated. Also, there are many additional constraints resulting from the nature of ZLMSG courier service that this method cannot take into consideration. They can be modelled with the time windows of the VRPTW, though. (See chapter *2.1 Analysis of the special constraints posed by the ZLMSG courier service* on page 7).

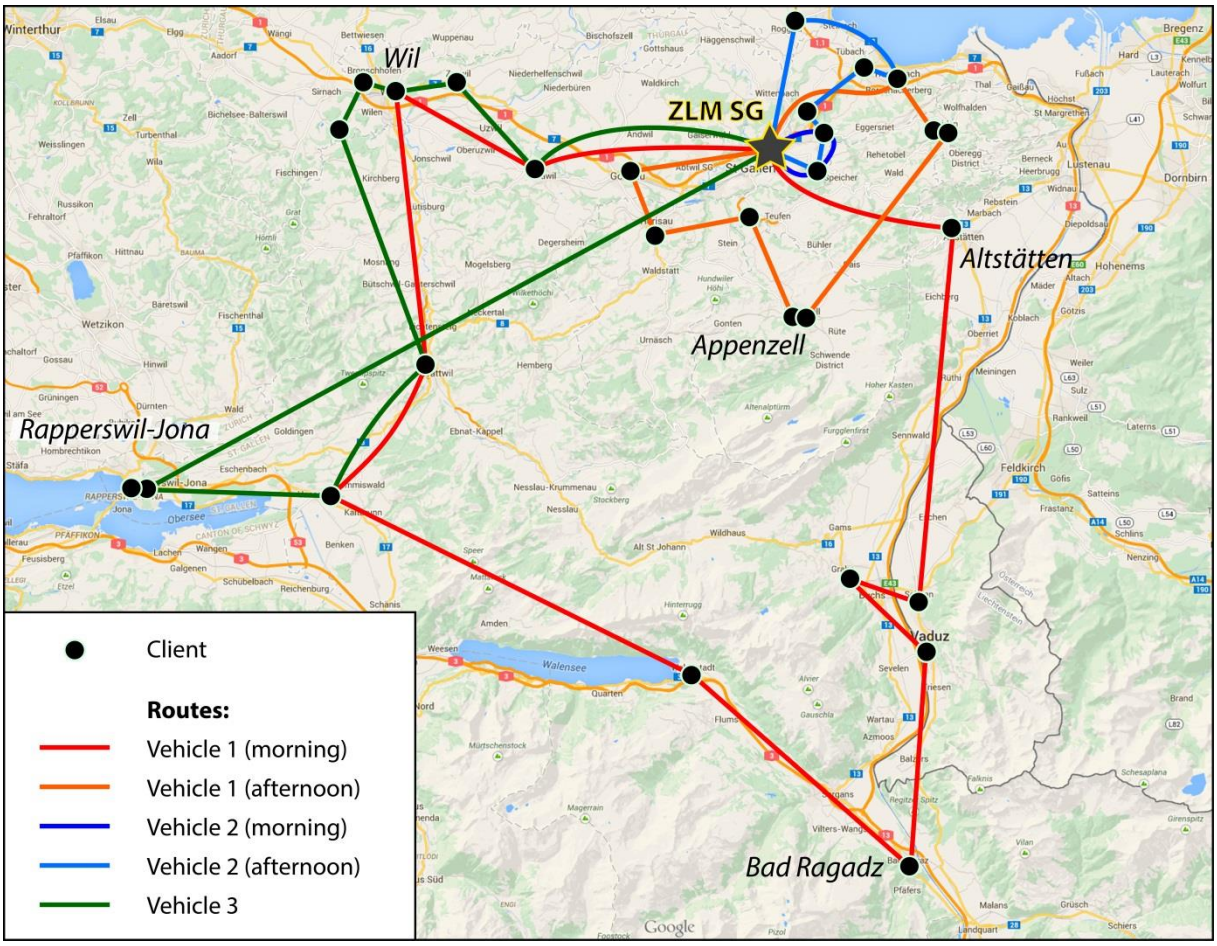


Figure 1: Sketch map of the spatial distribution of the ZLMSG clients.

Therefore, the aim of this thesis is to develop an application that can solve the special variation of the VRPTW posed by the ZLMSG courier service. The application should be simple and intuitive to navigate as the users are most likely unfamiliar with the topic of route optimisation. In addition it should provide an easy way to add and edit clients, offer a way to export the solution found, and be able to save and load the current courier service configuration.

In this thesis, the terms client and customer will be used to differentiate between the abstract customer concept used to define the VRPTW and a client as a model of a client or customer in the real world. This distinction will be of especial importance when talking about the modelling of the ZLMSG courier service. (See chapter *2.1 Analysis of the special constraints posed by the ZLMSG courier service* on page 7).

2 Concept

2.1 Analysis of the special constraints posed by the ZLMSG courier service

As this thesis focuses on developing an application for one client, the ZLMSG, with their specific variation of the VRPTW, the very first step was to learn of the current situation and find out about the ZLMSG's wishes and expectations of the application.

Consequently, requirements engineering was necessary. It showed that the situation of the ZLMSG courier service cannot be directly modelled by the VRPTW since there are several additional constraints that needed to be considered. The two main constraints are the fact that a customer can demand to be visited multiple times and that each courier must return to the ZLMSG during lunch time. The following chapters will give a detailed discussion of these constraints and how they were modelled to fit into the standard definition of the VRPTW.

2.1.1 Return at lunch time and lunch break of the courier

Since the client's objective is to get the result of the sample analysis as fast as possible, the sample needs to be transported back to the ZLMSG as quickly as possible. In the best case this would mean a pick-up service on call where each sample would be retrieved separately and transported straight back to the ZLMSG. Obviously, this is expensive and impractical, especially for large quantities of samples. Instead, the couriers drive both on a morning and an afternoon tour and always return to the ZLMSG at lunch time. This way the morning batch of samples can already be analysed right after lunch and the clients get the results for their samples in the early afternoon. The results for the afternoon batch are transmitted to the clients in the early evening. In addition to that, couriers have a one hour lunch break. Its point in time is more or less flexible.

This two-batch-system was retained for the route optimisation. It was modelled by creating a fictitious customer, an intermediate depot (definition in chapter *2.3.2.3.1 Intermediate depots* on page 24) that is present in every route by default and cannot be removed. This customer has a time window from 11.30 to 13.30, a service time of 1 hour, and the location of the ZLMSG.

Splitting the problem into a morning and afternoon part is not possible as there are clients who are visited only once per day. And as the lunch break may be handled flexibly but still needs to have a minimum duration of 1 hour, the morning and afternoon route schedules would influence each other. Hence, the optimisation algorithm has to be run over the whole day.

2.1.2 Time Windows: multiple visits per day and predefined visits

The other main difference to the standard VRPTW besides the mandatory lunch break at the ZLMSG (the depot) is the use of the time windows. Firstly, some clients wish to be visited up to three times a day. This concerns mainly large hospitals that accumulate quite an amount of samples during the day. So in order to get the samples transported to the ZLMSG as soon as possible (and get the analysis results as soon as possible too), they prefer to be visited multiple times per day.

Also, depending on the type of client (hospital or practice) and the number of visits paid to that client certain times of the day are more preferable than others for retrieving the samples. For example there is not much use in planning to retrieve samples from a practice early in the morning because not many samples will have been taken by then. But for a large hospital that is visited multiple times a day it will be possible to plan the first visit early in the morning. This means that the calculation of the visits' time windows is dependent on whether the client is a hospital or a practice. Figure 2 shows the preferred visiting times as they were specified by the ZLMSG.

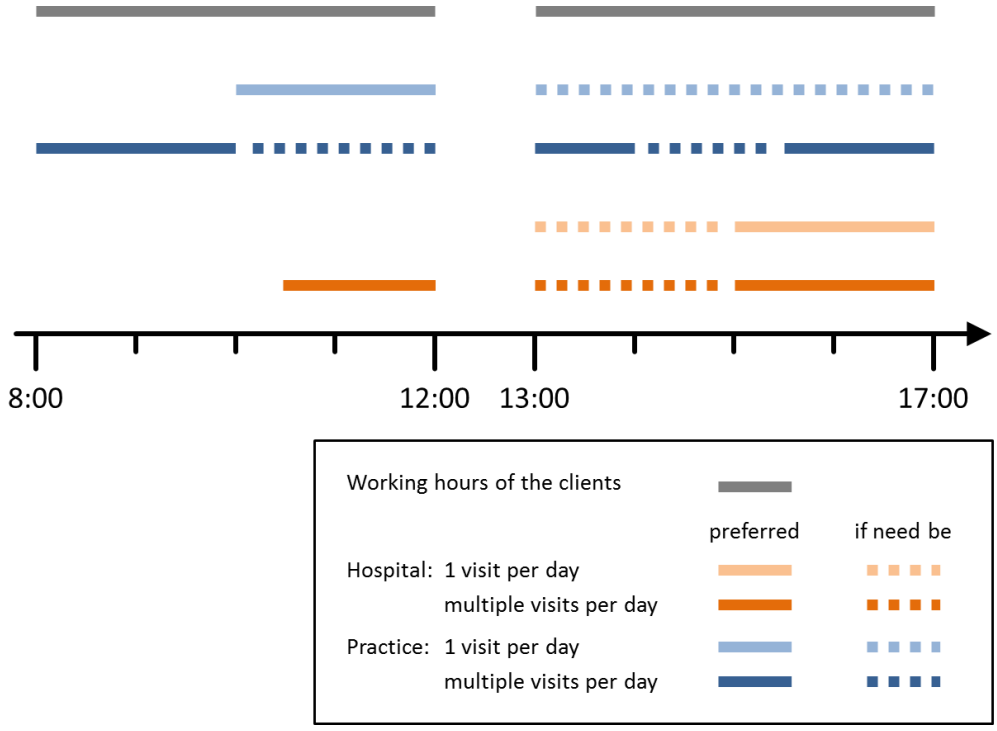


Figure 2: Preferred visiting times of the clients of the ZLMSG.

Additionally, there exists a courier from the Kantonspital St. Gallen (KSSG) that visits some of the clients of the ZLMSG as well. However, as this courier is not administered by the ZLMSG its route and clients have to be considered as fixed. Consequently, the times at which the KSSG courier's clients are visited should be taken into consideration when calculating the optimal routes for the ZLMSG couriers (reference visits). Plus, there are also clients who, themselves, wish to specify the time window in which they are visited (fixed visits). In both cases, the time windows of the remaining visits need to be arranged around the time windows of those predefined visits (PV).

And lastly, the couriers naturally have fixed working hours.

To allow for a client to be visited multiple times a day, the VRPTW was defined through all visits of all the clients instead of just the clients to be served. Meaning, each visit of a client was counted as a 'customer' in the VRPTW. A visit essentially represents the time window of that visit. For a closer understanding see also chapter *2.3.2.3 The Problem Model* (page 23) on the modelling of a problem instance.

The scheduling of time windows with regards to the constraints discussed above is actually an optimisation problem in itself. However, as this time scheduling problem was not part of this thesis, a simple deterministic algorithm was developed to calculate the time windows for each client. (See chapter *2.3.2.3.3 Deterministic algorithm for scheduling the visits of a ZLMSG client* on page 24).

2.1.3 Capacity Limit

As samples only take up very little space in the vehicle the capacity for retrieving samples is very high and can be assumed to be limitless. Thus, the checking of a vehicle's load capacity was simply omitted during the route optimisation process.

2.1.4 Exclusion of clients on certain weekdays and constancy for clients

Some clients don't need to be visited on every day of the week. The optimal route configuration may therefore differ for each weekday. Still, for the clients it is important that the time they are visited does not change every other day. Currently, this problem is solved by having slightly different route schedules on each weekday and repeating these every week. This concept was retained and realised by calculating the optimal routes for each weekday separately.

2.2 Technical specifications and data used

The application was written in Java using the eclipse Juno environment. Besides the standard System Library (JavaSE-1.7) the external libraries listed in Table 1 were referenced. The compiler compliance level was set to 1.7 as well. Hence, a system needs to have Java 7 installed, or a later version, to be able to run the Route Optimizer application.

Library	**Used for**
Gson: – gson-2.3.1	Saving and opening existing projects using the JavaScript Object Notation (JSON) format
From the Eclipse WindowBuilder: – forms-1.3.0 – miglayout15-swing	Designing the GUI
From the Apache POI Project: – poi-3.12 – poi-ooxml-3.12 – poi-excelant-3.12 – poi-ooxml-schemas-3.12 – poi-scratchpad-3.12	Exporting a solution to an Excel file
– xmlbeans-2.6.0	POI dependency

Table 1: External libraries referenced in the source code of the Route Optimizer application.

As calculating the distance between two clients of a real-world problem instance was not part of this thesis, the mapquest web service (MapQuest, Inc.) was used for this purpose.

For testing purposes, two sets of data were used: The Solomon instances set for 100 customers (Networking and Emergency Optimization) and the data from the ZLMSG.

The Solomon instances were used to test the implemented algorithms in general. The structure of the text files storing the instances can be found in *Appendix C – Solomon instances file structure*. The ZLMSG data was then used to test whether the application handles the additional constraints of the ZLMSG courier service correctly. However, due to data protection reasons, the ZLMSG data can only be published in pseudonymised form. Even the client's type (hospital or practice) needs to remain undefined. This pseudonymised form of the data can be found in *Appendix B – Data ZLMSG*.

2.3 Application architecture and development process

This section discusses the different parts of the implementation, looking at algorithms and concepts used, and the respective Java classes and interfaces.

Initially, it was planned to proceed in three consecutive steps: Firstly, implement the 'standard' VRPTW, then adapt the program to fit the constraints of the ZLMSG, and finally implement the Graphical User Interface (GUI). However, it soon became apparent that this was not practicable, because adapting a program for a particular client, in this case the ZLMSG, always implicitly demands adapting the user interface as well. The best software is useless, if the targeted users can't orientate themselves in it. Thus the workflow was shortened to two steps only:

1. Implementation of the method to solve the 'standard' VRPTW
2. Adaption of the program to fit the additional constraints of the ZLMSG courier service and simultaneous implementation of the graphical user interface (GUI) for the application

The initially planned three steps can still be seen though in the structure of the application. Figure 3 shows a diagram of the application's structure and its three modules: Visuals, Project, and the VRPTW Solver. The Visuals module mainly contains the GUI, and the VRPTW Solver contains everything that is needed to solve a standard VRPTW. The Project module in turn is the actual core of the application. It functions as the interface between the two other modules and is responsible for providing a project file representing the current session and courier service configuration. Thus, the additional constraints of the ZLMSG courier service are taken care of within the Project module as well.

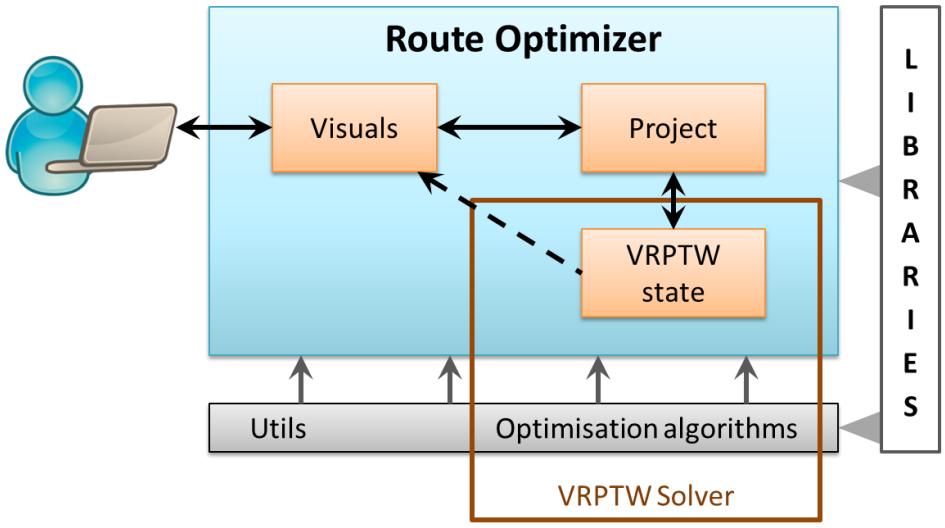


Figure 3: Overview of the Route Optimizer application's architecture.

In addition to the three modules, an utils package was developed to collect repeated general actions and services at a single place. Like the optimisation algorithms the utils package is practically stand-alone and could theoretically be reused for other applications or programming projects. Thus, they were separated from the actual Route Optimizer bundle. The following chapters will now discuss these three modules and the utils package in detail.

In order to test whether the application performs the desired actions, two sets of data were used. The set of Solomon instances for 100 customers was used during the development process as these problem instances can be read into the program easily. Later, the client data from the ZLMSG courier service was used to test the application for usability with real-world instances.

2.3.1 The VRPTW Solver module

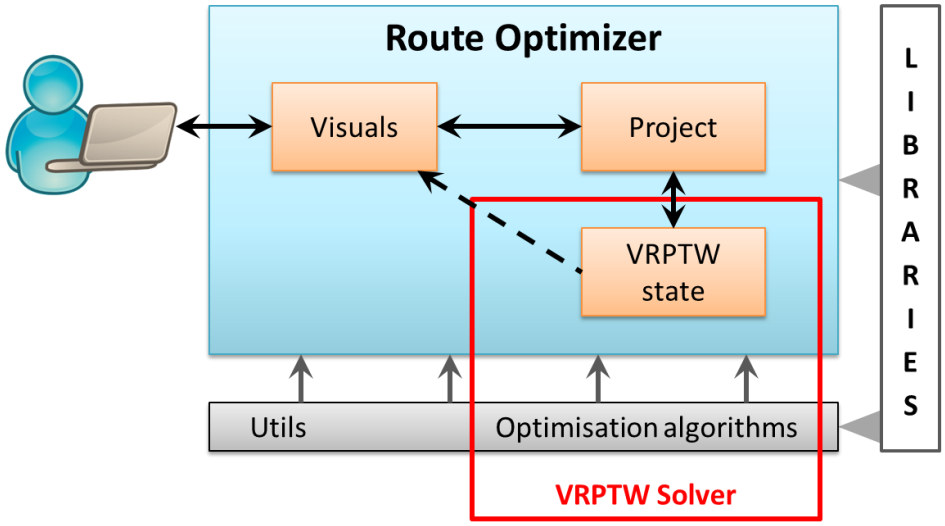


Figure 4: Overview of the Route Optimizer application's architecture.

The VRPTW Solver Module is again a composition of two separate components. While one component provides the functionality, resp. the algorithms, for solving an optimisation problem, the other component models the optimisation problem considered in this thesis, the VRPTW. The following two chapters will discuss these two components in more detail.

2.3.1.1 Optimisation algorithms

The first step in implementing the VRPTW solver was to decide on which algorithm to use. As mentioned in the introduction chapter (page 5), there exists a variety of approaches to solving the VRPTW. But since this thesis focuses on a specific instance of the VRPTW, the algorithm chosen to solve it should primarily work efficiently for this particular instance.

With this in mind, three algorithms belonging to the class of iterative local search methods were eventually chosen: Enhanced Hill Climbing (EHC), Simulated Annealing (SA), and the Hybrid Search (HS) method introduced by de Oliveira (de Oliveira, et al., 2008).

As the most robust, precise, and efficient method the Hybrid Search method was chosen. Compared to methods proposed in previous works it gave better or equal results for problem instances with few couriers in the solution and a high load capacity. And this is exactly the type of problem instance looked at in this thesis.

In contrast, the EHC and SA algorithms were chosen to give the application a certain degree of flexibility. The HS method is essentially built on the combination of Simulation Annealing with simple Hill-Climbing. Thus, in situations where the user can't invest a long period of time to search for a precise solution but still wishes to get an approximate solution quickly, he may choose one of these two methods. They might also become of use, if the problem instance at one point reaches such a size that the HS method would take too long to complete. Moreover, the EHC and SA methods nicely complement each other. While the EHC method works effectively on a search space with few optima, the SA method is more effective for a search space with many optima.

2.3.1.1.1 The state interface

The optimisation algorithms were chosen with respect to the needs of the problem considered in this thesis, the VRPTW. But aiming at reusability, they were implemented in a general manner. In fact, the three local search algorithms chosen for the Route Optimizer application are not problem-specific but can find the optimum in any search space consisting of 'states'. However, the following three operations or properties need to be defined for this search space in order for the local search algorithms to work (see also Figure 5 for a better understanding):

A) Definition of a neighbouring 'state'
B) Specification of the measure to be optimised
C) Provide a mechanism for random initialisation

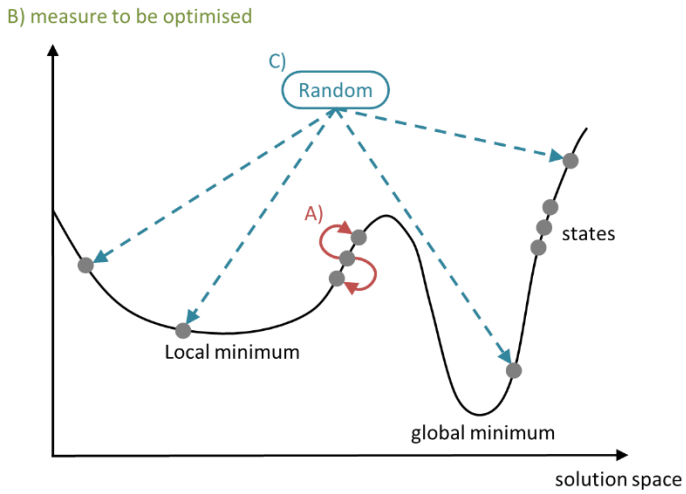


Figure 5: Operations or properties of the state interface.

These three requirements are summarised in a state interface and the EHC and SA algorithms will be explained using the notion of this state interface.

For a given VRPTW, the search space consists of all possible solutions to the VRPTW. This means a solution is an implementation of the state interface.

2.3.1.1.2 Enhanced Hill Climbing

Enhanced Hill Climbing combines stochastic Hill Climbing (Neller, 2005) with Random Restart. Stochastic Hill Climbing starts from an initial state in the search space S and at each iteration step evaluates a randomly generated neighbouring state $s' \in S$ is with respect to the optimisation measure $f(s)$. If s' is more optimal than the current state $s \in S$, that is, if $f(s') < f(s)$, then s' is accepted as the new optimal state and the search is continued from s' onwards. Otherwise, the neighbouring state s' is discarded and the search continues from the old state s onwards.

However, as mentioned before, Stochastic Hill Climbing is a local search algorithm. The more local optima a search space has, the higher the probability of the HC method getting stuck at such a locally optimal state. In order to avoid this, the stochastic Hill Climbing algorithm is then restarted a specified number of times. Each restart takes as input a random state of the search space. In the end, the optimal state of all restarts is chosen. The idea to enhance the algorithm with the Random Restart technique originated from the paper on the Hybrid Search method by de Oliveira (de Oliveira, et al., 2008).

2.3.1.1.3 Non-Monotonic Simulated Annealing

The strategy of the Simulated Annealing algorithm is similar to the one of the stochastic Hill Climbing method (Neller, 2005). At every iteration, it randomly generates a neighbouring state $s' \in S$ and compares it to the current state $s \in S$ with respect to the measure to be optimised $f(s)$. However, in order to escape from local optima the Simulated Annealing algorithm also accepts less optimal states during the search process.

This method is considered inspired by the annealing process in metallurgy, where a solid at higher temperature has a greater probability of reaching a state of higher energy. As the system cools, the solid will go towards a state of minimum energy but it can still reach a state of higher energy with a certain probability. This probability depends directly on the temperature of the system and can be mathematically expressed with the Metropolis Criteria:

$$P(x) = e^{-\Delta/T} \qquad \text{with } \Delta = f(s') - f(s) \text{ and } T \text{ as the temperature of the system}$$

Applying this concept to the VRPTW is straight-forward as Table 2 illustrates.

Var.	General description	Metallurgy	VRPTW
s	current state of the system	current state of the metal	current solution to the VRPTW
s'	next state of the system	next state of the metal	neighbouring solution of s
$f(s)$	objective function	energy function	measure to be minimized
T	temperature of the system		

Table 2: Meanings of the variables used in the definition of the optimisation algorithms.

With this the condition for accepting a neighbouring state s' is:

$$\Delta \leq 0 \vee \theta < e^{-\Delta/T} \qquad \text{where } \theta \in [0,1)$$

To give the algorithm even more flexibility, a non-monotonic cooling schedule is employed (de Oliveira, et al., 2008). This means that the algorithm does not continuously decrease the system's temperature but also occasionally allows a temperature increase.

2.3.1.1.4 Hybrid Search

As stated before, the Hybrid Search algorithm was taken from a work by de Oliveira (de Oliveira, et al., 2008). that aimed at proposing a robust and efficient method for solving the VRPTW. Starting from an initial state the algorithm first tests and finds varied but coarser solutions with non-monotonic Simulated Annealing. The found solution is then refined with the stochastic Hill Climbing method. The Hill Climbing itself is executed multiple times to take into account the fact that different executions of a stochastic Hill Climbing may result in different solutions.

In order to give the hybrid system the desired robustness, the combination of Simulated Annealing and Hill Climbing is re-initialized a number of times with a random solution of the search space. In the end the best solution of all random restarts is chosen. The flow of the Hybrid Search algorithm can also be viewed in Figure 6.

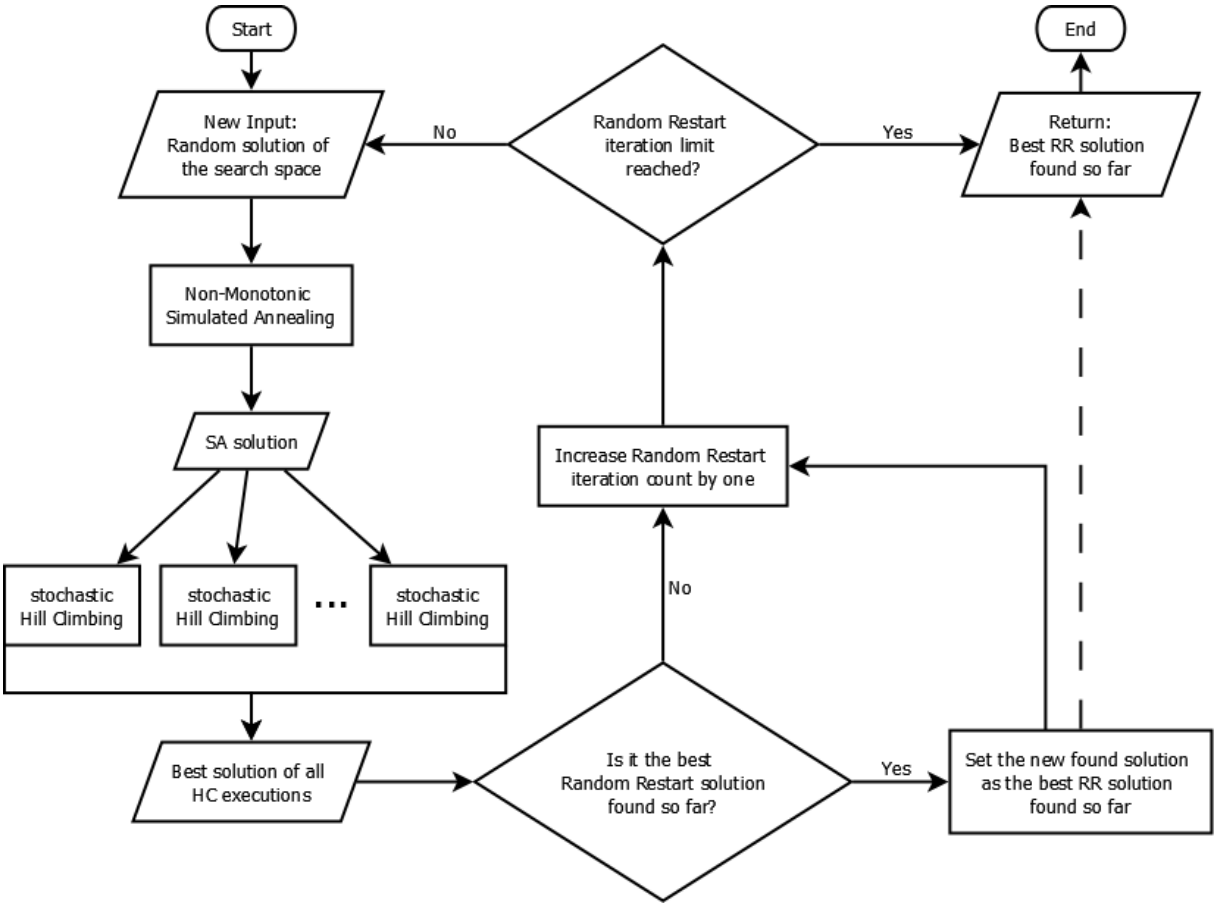


Figure 6: Overview of the hybrid system. (Adapted from (de Oliveira, et al., 2008)).

2.3.1.1.5 Implementation

During the development process the Hill Climbing and Simulated Annealing algorithms were first implemented and then combined to the Hybrid Search method. In fact, these algorithms were implemented twice in order to provide both, a static version as well as a runnable one.

As stated before, the optimisation algorithms were implemented as a separated component (or package in Java terminology) and in a generic manner. Except for the Java Swing dependency of the Optimizer class, this package is completely stand-alone, too. The following class table gives a short overview of the classes present in the package.

Class	Short description
Parameters	Container class storing all parameters that need to be defined for the optimisation methods.
OptimizationMethod	An enum (enumeration class) specifying the three methods that can be used for the optimisation: Enhanced Hill Climbing, Non-monotonic Simulated Annealing, and Hybrid Search.
Optimizations	Class containing static implementations of the three optimisation methods.
Optimizer	Implementation of the Java SwingWorker that optimises an initial state with the specified optimisation method.
State	Interface for defining the general state concept.

Table 3: Classes of the optimisation algorithms component of the VRPTW Solver module.

2.3.1.2 The solution to the VRPTW as an implementation of the state interface

The definition of a solution by itself is quite straight-forward: A solution contains a list of routes and each route consists of a list of customers. But a solution also needs to fulfil the requirements of the state interface (see chapter *2.3.1.1.1 The state interface* on page 12). This means it must define what neighbouring solutions are, it must specify the measure to be optimised, and it needs to have a mechanism for random initialisation. In addition it needs to ensure that the intermediate depot (definition in chapter *2.3.2.3.1 Intermediate depots* on page 24) representing the courier's lunch break is present in every route. In order to realise all these requirements in a proper, efficient, and comprehensible way, a few more concepts are needed. Table 4 lists the classes that represent these concepts.

Class	Short description
Problem	An abstract class with the purpose of defining what a problem instance of the VRPTW needs to look like. It specifies the customers, the depot, the distance matrix, and the travel time matrix.
Solution	An implementation of the state interface consisting of a list of routes.
Route	Essentially a list of route stops framed with route stops pointing to the depot at the beginning and end.
RouteStop	Placeholder for a position in a route that points to a certain customer. Stores this customers service begin in the route. (See also chapter *2.3.1.2.4 Route stops* on page 19).

Class	Short description
Customer	Representation of a client (or a visit: see chapter *2.3.2.3.2 Visits* on page 24) that is passed in the problem instance. Most of its fields are final.
Insertion	Abstraction of a possible insertion of a customer (resp. a route stop) into a certain route at a certain position. An insertion may be time-feasible or not. (See also chapter *2.3.1.2.1 Random initialisation through the Push-Forward Insertion Heuristic (PFIH)* on page 16).
OptimizationMeasure	An enum (enumeration class) specifying the three measures that can be optimised: distance, travel time, and delivery time. (See also chapter *2.3.1.2.2 Measure to be optimised* on page 17).
MutationMethod	A nested enum (enumeration class) inside the Solution class specifying the method to be used when generating a neighbouring solution. (See also chapter *2.3.1.2.3 Neighbourhood operators* on page 17).

Table 4: Classes of the VRPTW state component of the VRPTW Solver module.

2.3.1.2.1 Random initialisation through the Push-Forward Insertion Heuristic (PFIH)

A new solution is instantiated from a given problem instance using the Push-Forward Insertion Heuristic (PIFH) (Thangiah, et al., 1994). It is a constructive method that builds the solution with respect to the customers' insertion costs. The method is described below.

1. Open a new empty route.

2. The first customer to be inserted is chosen by the following cost function:

$$c_i = -\alpha d_{depot,i} + \beta l_i + \gamma \left(\frac{p_i}{2\pi}\right) d_{depot,i}$$

   with:

   c_i: initial insertion cost of customer i
   $d_{depot,i}$: distance from the depot to customer i
   l_i: latest arrival time (due date) of customer i
   p_i: polar coordinate angle of customer i with respect to the depot

   This cost function allows comparing customers with respect to the measure to be optimised, the latest arrival time and the angular value of the customer. The customer with the lowest cost is then inserted. In order to introduce the desired randomness when initialising a solution, the weights for the three criteria α, β, and γ are taken from a normal distribution $\mathcal{N}(\mu, \sigma)$ with $\sigma = 1$ and $\mu_\alpha = 0.7$, $\mu_\beta = 0.1$, and $\mu_\gamma = 0.2$ respectively. These values were found to be optimal (de Oliveira, et al., 2008). For one execution of the PFIH algorithm the same values for α, β, and γ are used.

3. For the current route all possible insertions (definition below) are generated with all the still unrouted customers and tested for time-feasibility (Solomon, 1987). From the time-feasible insertions the one with the lowest increase in the measure to be optimised is then set, meaning the client is definitely inserted into the route. This step is repeated until there are no more time-feasible insertions for the current route. Then, the current route is closed.

4. Steps 1 to 3 are repeated until there are no more unrouted customers.

An insertion symbolises a possible insertion of a certain customer, resp. a route stop, into a certain route at a certain position. An insertion may be time-feasible or time-infeasible (see glossary). Time-feasibility means all time windows of all customers in the route can be adhered to. An insertion also specifies the increase in the measure to be optimised that would result from inserting the specified customer. Thus a collection of insertion instances may be sorted with respect to this measure increase.

2.3.1.2.2 Measure to be optimised

The optimisation algorithms used for this thesis are described with the aim to minimise the total distance travelled. But in order to give the user a choice two additional optimisation measures were added in this thesis: Travel Time and Delivery Time. Delivery time in this case refers to the time it takes to travel from the customer, resp. the route stop, in question to the next depot. This measure is introduced with the ZLMSG in mind, for their main interest is to transport the client's samples back to the ZLMSG as fast as possible.

2.3.1.2.3 Neighbourhood operators

For local search methods, the definition of what is local in the search space, resp. the definition of the neighbourhood for a given solution, is essential. As a solution is a set of ordered lists without repetition (the routes), four basic permutation operators are enough to generically capture the neighbourhood $N(s)$ of a solution s (de Oliveira, et al., 2008). These operators are not based on heuristics, instead they more or less randomly perform a permutation of the solution s to generate a new solution $s' \in N(s)$. Thus they bring diversity to the search method. Additionally, de Oliveira introduced a fifth neighbourhood operator called Heuristic Mutation to balance out the search process and allow a more refined search once a promising solution is found. The Heurisitc Mutation is also a mutation operator, however, in contrast to the four basic ones, it uses information on the problem to construct a new solution $s' \in N(s)$. Studies suggest that the use of such an operator significantly increases the quality of the final solution produced by the optimisation algorithm (de Oliveira, et al., 2008). This was especially the case for problem instances such as the one defined by the courier service of the ZLMSG: random and clustered customer distribution, high load capacity, and few routes in the solution.

The first basic operator is called Swap Mutation (or '2-change' when applied to the Traveling Salesman Problem (TSP)) and is formally defined as:

$swap(s) = \{s' : s' \in S \wedge s'$ is obtained from s by swapping two random customers (C_i, C_j) of any routes $(R_k, R_l)\}$.

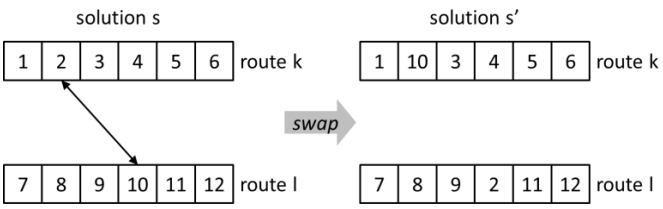


Figure 7: Diagram illustrating the Swap Mutation operator for two different routes in the solution.

The second basic operator called Insert Mutation can be described as:

$insert(s) = \{s' : s' \in S \wedge s'$ is obtained from s by removing a random customer C_i from any route R_k of s and inserting C_i again in any route R_l of $s\}$.

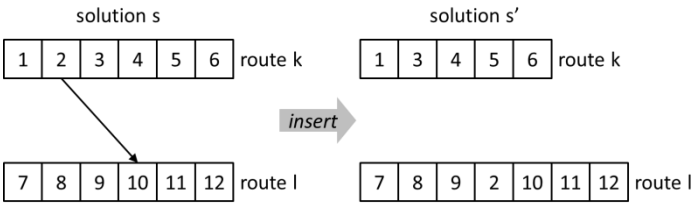


Figure 8: Diagram illustrating the Insert Mutation operator for two different routes in the solution.

The third basic operator is called Scramble Mutation and is defined as:

$scramble(s) = \{s' : s' \in S \wedge s'$ is obtained from s by randomly choosing a continuous sequence q of customers in any route R_k of s and mixing the order of the customers in q to create a sequence q' which will substitute q in $R_k\}$.

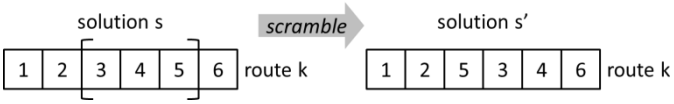


Figure 9: Diagram illustrating the Scramble Mutation operator.

The fourth basic operator called Invert Mutation can be described as:

$invert(s) = \{s' : s' \in S \wedge s'$ is obtained from s by randomly choosing a continuous sequence q of customers in any route R_k of s and inverting the order of the customers in q to create a sequence q' which will substitute q in $R_k\}$.

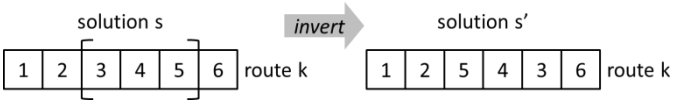


Figure 10: Diagram illustrating the Invert Mutation operator.

The Heuristic Mutation operator first removes a random number of customers from each route R_k of the solution s. The number of withdrawn customers varies for each route. This creates an incomplete solution h. Then, the withdrawn customers are again inserted into h using the PFIH method.

The above described neighbourhood operators were applied to all three optimisation algorithms used in this thesis. For each iteration during the search process, one of the five mutation operators is called randomly to generate a neighbouring solution $s' \in N(s)$. However, as there is a possibility that the solution s' obtained from one of the four basic operators does not satisfy all constraints of the VRPTW (e.g. time-feasibility), the solution s' is only accepted if it does satisfy all constraints. In case of constraint violation, the optimisation algorithm keeps the initial solution s and moves to the next iteration. The Heuristic Mutation operator always returns a solution s' satisfying every constraint of the VRPTW as the PFIH automatically ensures all constraints are fulfilled.

2.3.1.2.4 Route stops

In order to facilitate the implementation of the VRPTW solution, especially of a route of this solution, the concept of a route stop was introduced. In a route, the definition of previous and next for an element in the route can be very useful, for example when calculating the push forward in time when inserting a new customer (see chapter *2.3.1.2.1 Random initialisation through the Push-Forward Insertion Heuristic (PFIH)* on page 16). However, defining the notion of previous and next for a customer doesn't make much sense, even more so, when the customer is not part of a route.

For this reason the route stop concept was implemented as an additional class in the VRPTW module. With this, a customer that is a spatially fixed client can be differentiated from a route stop which is a position in a route with a previous and next route stop. This also has the advantage that customers can always be passed and stored as a reference, making customer comparison very simple. Naturally, the service begin is also being stored in the route stop instance as it keeps changing during the optimisation process.

2.3.1.2.5 Ensuring the presence of the intermediate depot in every route

Whether there exists an intermediate depot (definition in chapter *2.3.2.3.1 Intermediate depots* on page 24), is specified in the problem instance that is used to initialise a solution. If there is one, the solution needs to guarantee that every route of this solution always contains this intermediate depot. This was achieved by simply including the insertion of the intermediate depot into the route as the last step during route creation. Additionally, it was ensured that a depot can never be removed from a route. If the intermediate depot were to be removed during the application of one of the five neighbourhood operators, the method simply removes the route stop, resp. the customer, before the intermediate depot. If there is no route stop before the intermediate depot, the route stop after the intermediate depot is removed. However, this approach slightly increases the possibility of removal for route stops, resp. their customers, that are scheduled just before or after an intermediate depot. But this was deemed to be acceptable or favourable even, because customers that are scheduled just before a depot are more flexible with regard to their service begin.

2.3.2 The Project module

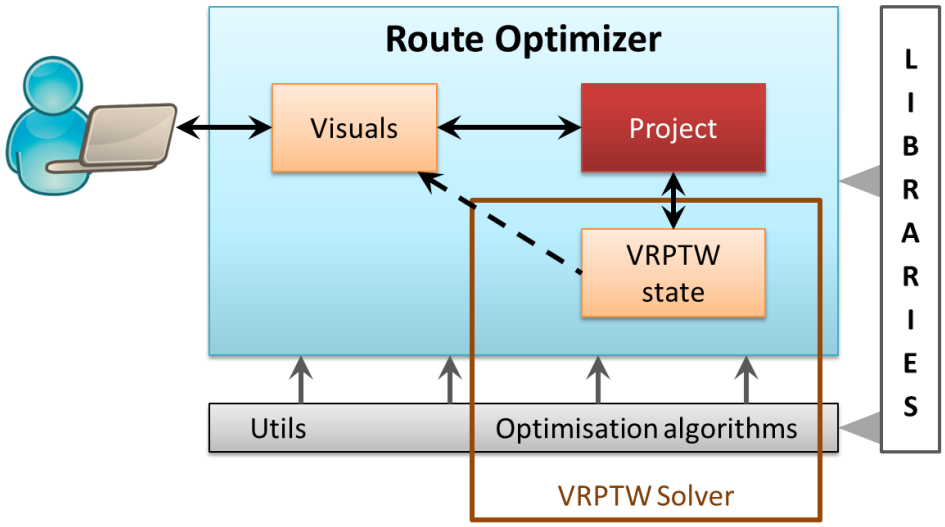


Figure 11: Overview of the Route Optimizer application's architecture.

The Project module's main role of operating the application was realised through the introduction of a Project class. An instance of this class corresponds to a Route Optimizer project, which is like what a Word document is for Microsoft Word. There exists two types of projects, benchmark and real-world projects. This distinction was needed in order for the Route Optimizer application to be able to handle both, benchmark and real-world problem instances. For a more detailed discussion of this issue, see chapter *2.3.2.1 Benchmark vs. real-world instances* on page 22.

Each such project stores its filename, folder path as well as a table model (see chapter *2.3.4 The utils package* on page 34) containing the last found solution. But most importantly, it stores the courier service configuration defined by the user in a so-called problem model (definition in chapter *2.3.2.3 The Problem Model* on page 23). This is also where the modelling of the additional constraint of the ZLMSG courier service takes place.

When the user starts an optimisation process, this problem model is used to generate a benchmark or a real-world problem instance depending on the project's type. This generated problem instance is then used to calculate the initial solution that is passed to the optimisation algorithm as input. Once the optimisation is finished, the solution is retrieved from the optimisation algorithm and stored in the project as a table model. A visualisation of these interactions can be seen in Figure 12.

Naturally, the Route Optimizer application needs to provide a way to create, save, and open projects. This topic is covered in chapter *2.3.2.4 Creating, opening, and saving projects* on page 27. Additionally, a mechanism for exporting the last found solution, resp. the route configuration, to an Excel file was implemented. (See chapter *2.3.2.5 Exporting found solutions* on page 28). This is due to the fact that the Route Optimizer application can only display a found solution. It does not support printing or any other functionality that users might wish for.

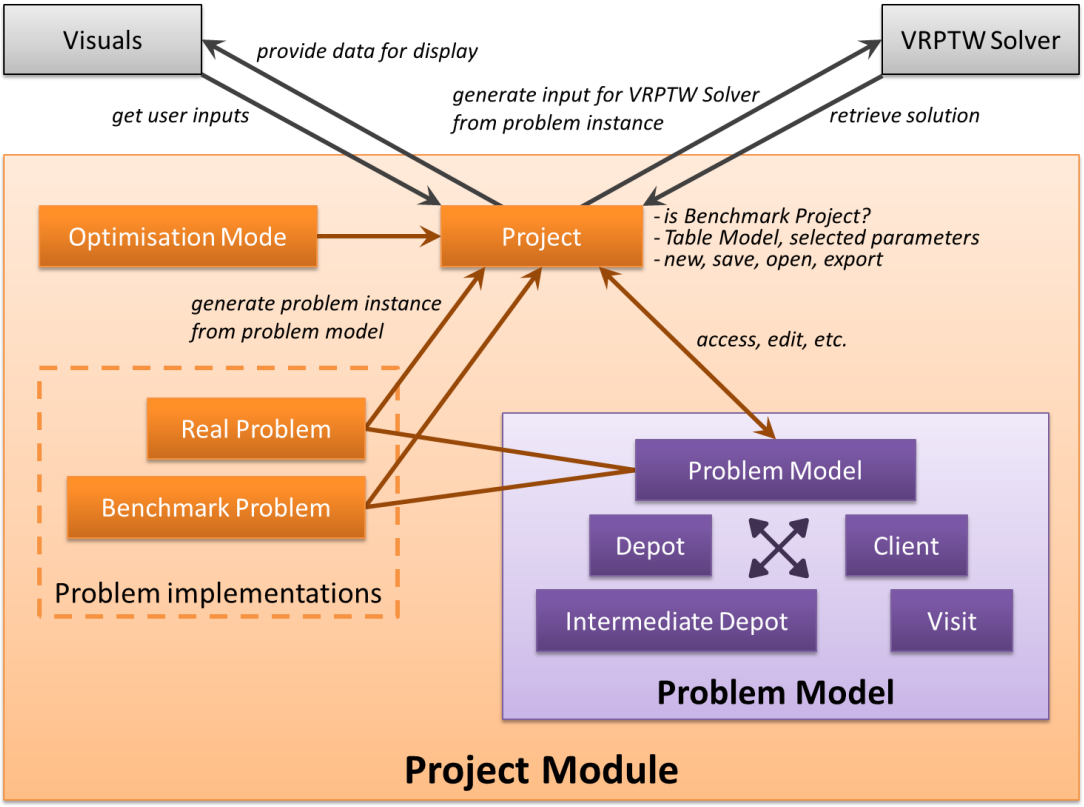


Figure 12: Diagram illustrating the Project module and its interaction with the other two modules.

Figure 12 indicates that the parameters selected by the user in the GUI (optimisation measure, method, and mode) are stored in the project as well. However, they are not written into the file when a project is saved. Although by saving the parameters to the file one could store something like optimisation preferences but it would create the possibility of a user mistakenly run an optimisation with the wrong parameters. By clearing the parameter selection after each optimisation run and not saving them to the project file, the user is asked to think about the selection before each optimisation. Moreover, selecting the parameters only takes three clicks at minimum, which is an acceptable number.

Lastly, Project module defines the three optimisation modes provided in the Route Optimizer application: fast, compromise, and precise. (See chapter *2.3.2.6 Optimisation mode* on page 29).

Again, Table 5 gives a short overview of the classes present in the Project module. For the descriptions of the classes of the problem model see chapter *2.3.2.3 The Problem Model* on page 23.

Class	Short description
Project	Representation of a Route Optimizer Project (ROP) file that can be created, saved, and opened. A project may be a benchmark or real-world instance.
RealProblem	Implementation of the abstract Problem defined in the VRPTW Solver module for real-world problem instances.
BenchmarkProblem	Implementation of the abstract Problem defined in the VRPTW Solver module for benchmark problem instances.
OptimizationMode	An enum (enumeration class) specifying the three supported optimisation modes: fast, compromise, and precise. (See also chapter *2.3.2.6 Optimisation mode* on page 29).

Table 5: Classes of the Project module.

2.3.2.1 Benchmark vs. real-world instances

As stated above, the Route Optimizer application can solve both benchmark and real-world problem instances. A benchmark problem is a predefined abstract problem. Entire sets of benchmark problems are often used to compare different optimisation algorithms and to indicate their efficiency. On the other hand, a real-world problem describes an optimisation problem of the real world, such as the one posed by the ZLMSG courier service.

In the application, the distinction is made by storing a Boolean value in each project that denotes whether the current project is a benchmark project or not. This Boolean is false by default and the project thus a real-world project. The field storing this value is final, needs to be defined at creating time of the project, and can't be changed afterwards. This is due to the fact that distance and travel time calculations work completely different for benchmark and real-world problem instances. And the coordinates needed as input for these calculations are different as well. While for a customer in a benchmark problem the coordinates can be any random real number, for a client in the real world it mandatorily must be the longitude and latitude of its location. Hence, a conversion from a benchmark project to a real-world project has no real use and thus isn't possible in the Route Optimizer.

However, instead of having a Boolean determine the project type it would have been more elegant to solve the issue with inheritance and polymorphism. But as the Gson library used for saving and opening projects to a file cannot deal with abstract types unless custom serializers and deserializers are defined, it was decided to implement the less elegant version with the Boolean field. For further explanation regarding the saving and opening mechanisms see chapter *2.3.2.4 Creating, opening, and saving projects* on page 27.

2.3.2.2 Supporting optimisation by weekdays

In order to support optimisation by weekdays, which is one of the ZLMSG's additional constraints, each client stores for each weekday whether it needs to be visited on that weekday. For each weekday, a real problem is then generated with only the clients that need to be visited on this weekday. Each of these problems is then optimised separately.

For benchmark problem models all weekdays are set to 'need a visit' by default and the optimisation is run on the weekday 'Monday'. But this is simply an implementation issue.

Definitions of benchmark and real problems are given in chapter *2.3.2.1 Benchmark vs. real-world instances* on page 22.

2.3.2.3 The Problem Model

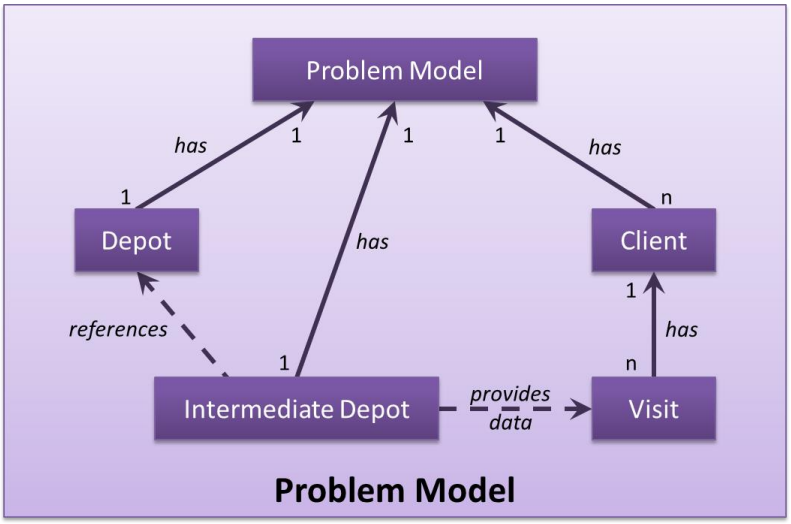


Figure 13: Visualisation of the relations inside the problem model.

A problem model models a courier service configuration with its clients, the depots, and all the additional constraints considered in this thesis. It consists of one depot, which needs to be defined at creation time of the problem model, a list of clients, and possibly an intermediate depot (see also Figure 13). Additionally, a client may have one or more visits. While the definitions of depot and client are trivial, the purposes of an intermediate depot and of a visit require further explaining.

The problem model also provides a way to schedule the specified number of visits for a client by implementing a simple deterministic algorithm (see chapter *2.3.2.3.3 Deterministic algorithm for scheduling the visits of a ZLMSG client* on page 24).

Table 6 lists the classes needed in the problem model and gives a short description for each class.

Class	Short description
ProblemModel	Model of the problem instance to be solved. A problem model may have one depot, one intermediate depot, and several clients.
Client	The class representing a client in the real world. A client may have one or more visits.
Depot	The class representing a depot in the real world.
IntermediateDepot	Concept that allows forcing a solution to only have route schedules where the vehicles must return to the depot somewhere in between.
Visit	Representation of the time window during which the service should take place that provides additional functionality.

Table 6: Classes needed in the problem model.

2.3.2.3.1 Intermediate depots

The concept of an intermediate depot was introduced in order to model the lunch break constraint of the ZLMSG courier service. However, this concept would also be applicable for other problem instances. An intermediate depot is very similar to the 'standard' depot, and, in fact, the intermediate depot actually references the 'standard' depot for properties like name or coordinates. The only difference is that instead of a service time the intermediate depot has a so-called scheduled stay. This scheduled stay denotes the time interval during which the 'service' would preferably take place. This information is needed when scheduling the visits of a client in the problem model (see chapter *2.3.2.3.3 Deterministic algorithm for scheduling the visits of a ZLMSG client* on page 24). Thus, the intermediate depot mandatorily needs to be defined before the clients' visits can be scheduled. The duration of the scheduled stay is the service time of the intermediate depot. An intermediate depot also has a time window which must cover the scheduled stay.

During the optimisation process, only the duration of the scheduled stay and the time window are used. However, the implementation of a solution is such that each route mandatorily must contain the intermediate depot at all times. This issue is discussed in chapter *2.3.1.2.5 Ensuring the presence of the intermediate depot in every route* on page 19.

2.3.2.3.2 Visits

In order to allow for a client to be visited multiple times a day, the visit concept was developed. A visit essentially stands for the time window during which service should take place. But it also provides the additional functionality which is needed for scheduling a client's visit throughout the day.

For once, a visit can be shifted with respect to an intermediate depot (definition in chapter *2.3.2.3.1 Intermediate depots* on page 24). This means that if the start of the intermediate depot's scheduled stay lies within the visit's time window, the time window of the visit is prolonged by the duration of the intermediate depot's scheduled stay. And if the visit is later than the start of the intermediate depot's scheduled stay, the entire time window of the visit is shifted by the duration of the intermediate depot's scheduled stay. If the visit is earlier than the intermediate depot's scheduled stay, then nothing happens.

Also, a visit is either predefined or not. Predefined means, it is either a fixed visit (visits specified by the client) or a reference visit (visits paid by the KSSG courier).

2.3.2.3.3 Deterministic algorithm for scheduling the visits of a ZLMSG client

To calculate the visits, resp. the time windows, of a ZLMSG client, a simple deterministic algorithm was developed. As input, it takes the reference visits and the fixed visits, if there are any for the current client, as well as the total number of visits to be paid by the fleet of vehicles (the ZLMSG couriers). It returns all visits, resp. all time windows, of the client that are relevant for the route optimisation (reference visits are excluded).

The algorithm is divided in three main steps. First a default distribution (DD) is generated as a function of the total number of visits paid to the client. This includes the predefined visits (fixed and reference visits) and the visits to be defined (variable visits). In a second step, for each fixed visit the closest visit in the default distribution is determined and replaced by the fixed visit. And in the last step, for each reference visit the closest visit in the default distribution that is not a fixed visit is removed.

This algorithm was tested with all the ZLMSG clients that have predefined visits and the resulting visit distributions were found to be acceptable. Acceptable in this case means, that the resulting time windows always matched with the route schedule that is currently employed by the ZLMSG courier service.

This algorithm is run prior to every route optimisation. In fact it would be sufficient to recalculate the visits' time windows of a client only after a change has been made to the client. However, as the application in its current stage does not provide a way to directly edit a client, the recalculation of the visits was made to be a pre-processing step to the optimisation. The visit list of a client resulting from this calculation is not saved to the ROP file for the same reason. Instead, the number of visits and the predefined visits are saved to the file.

2.3.2.3.3.1 Generating the default distribution (DD) for a ZLMSG client

Generating a default distribution for a ZLMSG client's visits, resp. its time windows, means distributing the specified number of visits uniformly over the whole day. However, this uniform distribution was slightly modified to fit the preferred visiting times of the clients of the ZLMSG. Hence, the visit distribution generated with this method is tailored to ZLMSG clients and most likely will not work for other real-world problem instances.

The method aims at finding a visit distribution that approximates the preferred visiting times described in chapter *2.1.2 Time Windows: multiple visits per day and predefined visits* (page 7) as best as possible. However, the VRPTW does not support 'preferred' time windows. The time windows in the VRPTW need to be defined absolutely. Thus, the time windows found by the described method are hard time windows, even though this does not reflect the entire situation in the real world. See also chapter *4.3 Improvements regarding the modelling of the ZLMSG courier service's constraints* on page 38.

The flow of the method is described below.

A) If only one visit needs to be scheduled, the preferred time window shown in Figure 2 on 8 is adopted one-to-one and the method terminates.

B) Otherwise, the following calculation steps take place:

1. Calculate the number of parts p needed with the following formula:

$$p = 2n - 1 \qquad n: \text{ number of visits to schedule}$$

2. Calculate the time span t_{alloc} during which visits may be scheduled. For example for a practice this would be $t_{alloc} = 17{:}00 - 10{:}30 - 1\,\mathrm{h} = 5\,\mathrm{h}\ 30\,\mathrm{min}$. The one hour subtracted is the lunch break.

3. Calculate the duration of the earliest visit $d_{earliest}$, resp. its time window with the following formula:

$$d_{earliest} = \frac{t_{alloc}}{p} - \frac{t_{alloc}}{\alpha \cdot p}$$

p: number of parts needed
α: scaling parameter

The scaling parameter α is chosen in such a way that the earliest visit exactly corresponds to the time window specified in Figure 2 on page 8. For hospitals this gives $\alpha_H = 4$ and for practices $\alpha_P = 6$ respectively.

4. Set the earliest visit with the above calculated duration $d_{earliest}$.

5. Calculate the duration of the remaining visits d_i $(1 < i \leq n)$, resp. its time windows with the following formula:

$$d_{remaining} = \frac{t_{alloc}}{p} + \frac{t_{alloc}}{\alpha(p-1)}$$

p: number of parts needed
α: scaling parameter

6. Calculate the start s_{i+1} and end e_{i+1} of the next visit, resp. its time window, the following way:

$$s_{i+1} = e_i + d_{remaining} \qquad e_{i+1} = s_{i+1} + d_{remaining}$$

7. Set the next visit with the above calculated start s_{i+1} and end e_{i+1}.

8. Repeat steps 6 and 7 until all visits are set.

2.3.2.3.3.2 Determining the closest visit

As the method to determine the closest visit, resp. the closest time window, is a rather complex structure of conditionals, it is best illustrated in a flow diagram (Figure 14). The main principle is to loop over the default distribution until one reaches a visit concurrent with or later than the predefined visit (PV). In the first case, the visit which has just been reached is replaced or removed. In the latter case, it is then determined whether the visit which has just been reached or the previous one is closer to the predefined visit. In contrast to the method used for generating the default distribution (DD), this algorithm can be applied to other real-world problem instances as well.

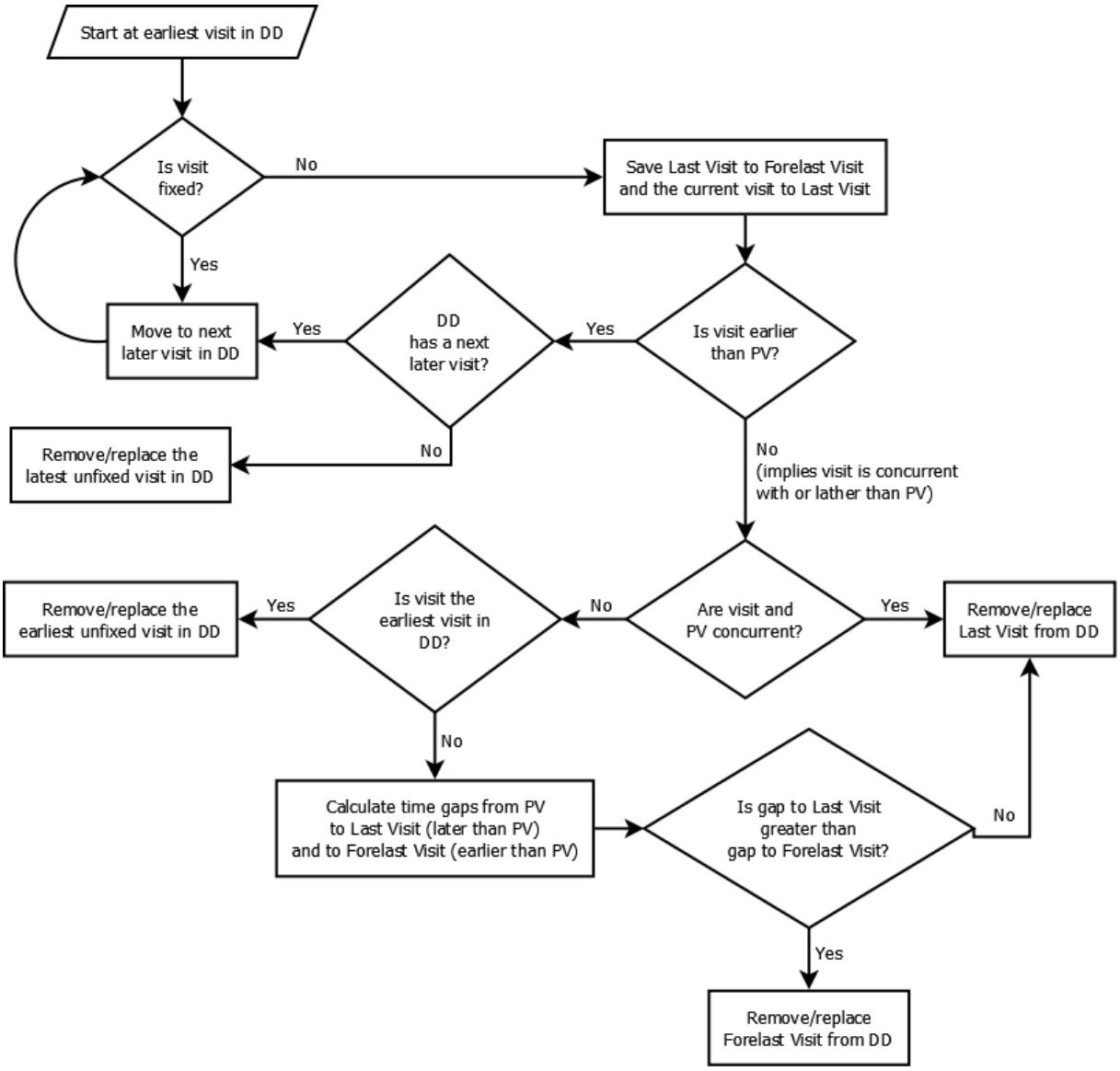


Figure 14: Flow of the algorithm employed to determine the closest visit.

However, this algorithm assumes that the predefined visits are distributed over the day in a manner that matches with the total number of visits this client desires. In an extreme case where for example two predefined visits are concurrent, the algorithm may return a visit distribution showing unreasonably large gaps between time windows. This is due to the fact, that the algorithm always starts with a default distribution and then simply deletes or replaces the visit closest to the predefined visit. It does not rearrange the visits afterwards to give a better distribution.

2.3.2.4 Creating, opening, and saving projects

The functionality of creating, opening and saving these project files is directly handled by the Project class itself. For this purpose a new file type with a specific file extension was created. This file type is called Route Optimizer Project File (*.rop) and corresponds to an instance of the Project class.

For the serialization and deserialization (saving and opening) of a project the Gson library from Google was used (Google). This library provides a way to convert Java objects into their JSON representation and vice versa. JSON is the abbreviation for JavaScript Object Notation, which is a lightweight data-interchange format. It is both, easy for humans to read and write and easy for machines to parse and generate (JSON Group). Hence, this ROP file is in fact nothing but a JSON file (*.json) and can be opened with any other standard text-editing program. However, not all text editors recognise the line breaks. For example, WordPad does recognise them, while Notepad doesn't.

In order to improve the readability of the resulting JSON file a GsonBuilder instance with the following configuration was used to write (save) and read (open) the file:

- Fields without the Expose Annotation were excluded.
- Pretty printing was used.
- Nulls were serialized.
- A custom TypeAdapter was registered for the Enum Weekday.
- Complex map key serialisation was enabled.

Additionally, custom InstanceCreators were used to deal with the EnumMaps that are needed to support optimisation by weekdays (see chapter *2.3.2.2 Supporting optimisation by weekdays* on page 22).

The creation procedure for a new project depends on the type of project one wishes to create. For real-world projects it simply initialises a new real-world project instance with an empty problem model. Creating benchmark projects is more complicated. Since the Solomon benchmark instances used in this thesis are stored in text files, a benchmark project needs to be created from such a text file. This is done by reading the information about the clients, their coordinates, time windows, and service times from the file and then correctly filling it into the problem model of the created benchmark project.

The Route Optimizer application can only have one open project at a time. Unfortunately, at the application's current stage, if a new project is opened, the currently running project will be overwritten even if there are unsaved changes in the previous project. (See also chapter *4.1 Usability improvements* on page 36).

2.3.2.5 Exporting found solutions

The Route Optimizer application provides a way to export a found solution, resp. a route configuration, to an Excel file. For this purpose the external libraries from the Apache Poi Project were used (The Apache Software Foundation). However, this export function does not export the actual solution. Instead, it exports the table model (see chapter *2.3.4 The utils package* on page 34) that is used for the display of the solution in the display panel (see chapter *2.3.3.4 Solution display panel* on page 33). This way, the general information about the solution, like for example the total distance travelled, is also written into the Excel file.

2.3.2.6 Optimisation mode

A mode stands for the parameter configuration used when running the optimisation process. (See also chapter *2.3.1.1 Optimisation algorithms* on page 11). Table 7 displays the exact values used for each mode.

Parameter	Fast	Compromise	Precise		
Number of random restarts	3	6-7	30		
Initial temperature used in for SA	100	100	100		
Decrement used for SA	0.5	0.1	0.05		
Number of SA iterations	3'000	15'000	30'000		
Number of SA iterations to cooling	10	50	100		
Number of SA iterations to heating	100	500	1'000		
Number of HC executions	1	2	3		
Number of HC iterations	100	500	1'000		
Ratios:				**F : C**	**C : P**
Total iterations EHC	300	3'000	30'000	0.10	0.10
Total iterations SA	3'000	15'000	30'000	0.20	0.50
Total iterations HS	12'400	112'000	1'023'000	0.11	0.11

Table 7: Parameters used for the three optimisation modes Fast, Compromise, and Precise.

For the precise mode the parameters were taken from the paper by de Oliveira (de Oliveira, et al., 2008). The fast mode was defined with the aim at to be approximatively a hundred times faster than the precise one. Thus iterations and restarts were all divided by ten and the decrement multiplied by ten to keep the ratio despite the unchanged initial SA temperature. The compromise mode is, as its name already states, a compromise between fast and precise. It is aimed at being about ten times faster than the precise mode and about ten times slower than the fast mode (see ratios in Table 7). But as a statistically significant linear correlation was found between the quality of the optimisation's final solution and the decrement used in the SA (de Oliveira, et al., 2008), the decrement for the compromise mode was intentionally set lower. As a consequence, more iterations were needed to keep the balance. To still reach the desired reduction in processing time, the number of random restarts was reduced instead. However, the SA method by itself performs no random restarts. Hence when using this optimisation method, the three-to-three ratio cannot be kept.

2.3.3 The Visuals module

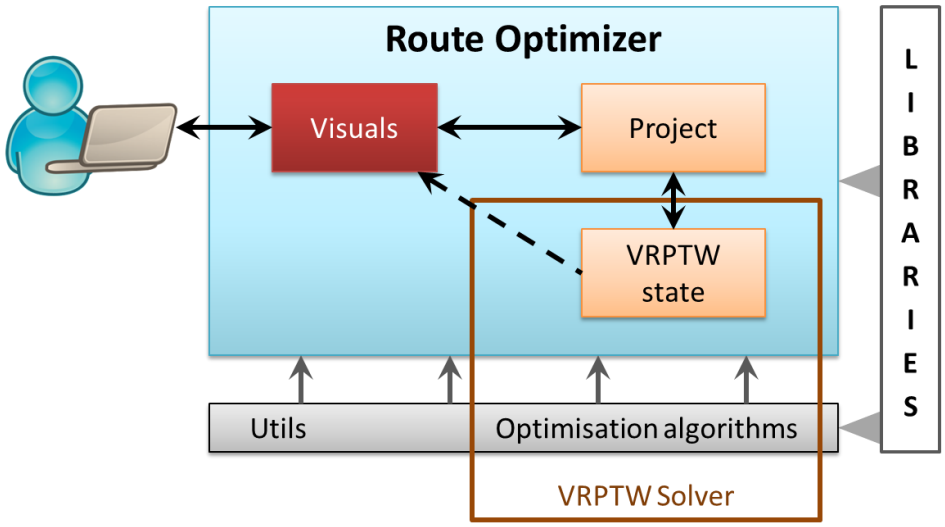


Figure 15: Overview of the Route Optimizer application's architecture.

The role of the Visuals module is to coordinate the interaction with the user. Thus the main part of this module is the GUI. The aim while designing the GUI was to keep it simple yet intuitive. For the latter, it was decided to use the same basic layout (Figure 16) that can be found in many other applications. The idea to have the title bar display the file name of the currently open project was also taken from other applications.

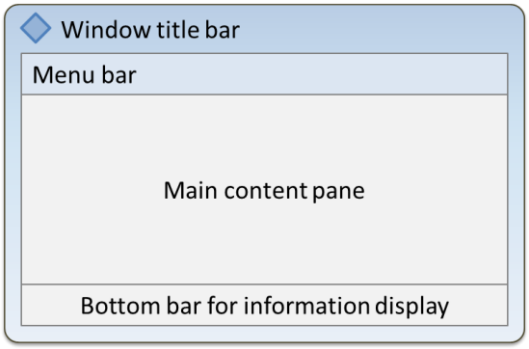


Figure 16: Basic layout of the Route Optimizer application.

The main content pane was then split into a left and a right panel: One to show the optimisation control and progress and one for displaying the calculated solutions. As one needs to use the controls and run the optimisation before the solution can be displayed, and since in our region people read from left to right, the controls and the progress display were assigned to the left panel and the solution display to the right panel. And because the optimisation progress is only needed after the controls have been set, the progress display was placed below the controls. Figure 17 also illustrates the described setup.

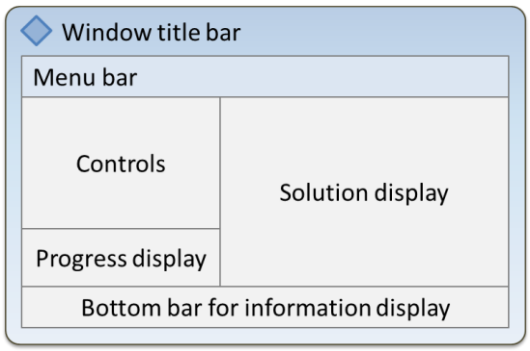


Figure 17: Layout of the GUI of the Route Optimizer application.

The following chapters now discuss these sections of the GUI in detail. The Visuals module only contains one class:

Class	Short description
GUI	Implementation of the application's graphical user interface (GUI).

Table 8: Classes of the Visuals module.

2.3.3.1 Menu bar

The terms used in the menu bar were again taken from other applications. First there is a File menu where new projects can be created or existing ones saved and opened. The functionality to export a calculated solution to an Excel file can also be found in the File menu. (See also chapter *2.3.2.5 Exporting found solutions* on page 28). Then there is a menu for editing the currently open project. At the applications current stage, this menu only contains a button for recalculating the visits of all clients, though. For the algorithm used see chapter *2.3.2.3.3 Deterministic algorithm for scheduling the visits of a ZLMSG client* on page 24. Next there is the View menu which only contains one item. From this item a new dialog displaying the currently stored Clients can be opened. Editing clients is not possible in this dialog. Finally, there is a Benchmarking menu where benchmark problem instances can be loaded into a new project (see chapter *2.3.2.4 Creating, opening, and saving projects* on page 27).

2.3.3.2 Controls panel

The controls panel is the place where the user can choose between all the provided optimisation options discussed in the previous chapters (see Table 9). The targeted users for the Route Optimizer application are most likely unfamiliar with the topic of route optimisation and thus might have a hard time understanding the meaning of these parameters. Therefore, a short description of the selected option is shown to the right of each radio button group.

Parameter	Options	Shown description
Measure to be optimised	Distance	The total distance travelled is used as the measure to be optimised. This means an optimal solution is a route configuration where the couriers collectively cover the shortest distance.
	Travel time	The total time travelled is used as the measure to be optimised. This means an optimal solution is a route configuration where the couriers collectively spend the least time on the road. Waiting time and service time are included in the calculation.
	Delivery time	The average delivery time is used as the measure to be optimised. This means an optimal solution is a route configuration where the travelling time from each client back to the depot is on average minimal.
Optimisation method	Enhanced Hill Climbing	EHC is a simple and fast algorithm. But it isn't very robust and in certain cases, especially in combination with the mode 'fast', it may yield less optimal solutions. This algorithm is efficient, when it is obvious that there are only a few optimal solutions to begin with.
	Non-monotonic Simulated Annealing	Simulated Annealing works similarly to Enhanced Hill Climbing but is more robust and thus slower. This algorithm is primarily suitable for situations where there are likely to be many optimal solutions.
	Hybrid Search	Hybrid Search is a combination of Enhanced Hill Climbing and Simulated Annealing. It is a robust and precise algorithm and accordingly demands a lot of time to complete.
Optimisation mode	Fast	The calculation time is always dependant on the options chosen for the parameters above and on the size and configuration of the problem instance (clients, visits/day). 'Fast' is approx. 10 times faster than 'Compromise' and approx. 10 times faster than 'Precise'.
	Compromise	The calculation time is always dependant on the options chosen for the parameters above and on the size and configuration of the problem instance (clients, visits/day). 'Compromise' is approx. 10 times slower than 'Fast and approx. 10 times faster than 'Precise'.
	Precise	The calculation time is always dependant on the options chosen for the parameters above and on the size and configuration of the problem instance (clients, visits/day). 'Precise' is approx. 10 times slower than 'Fast' and approx. 10 times slower than 'Compromise'.

Table 9: Explanations displayed in the Route Optimizer for the parameter options.

The controls panel also contains the run button at the very bottom in the right corner for starting the optimisation process. This button is implemented in such a way that an error message pops up, if the button is clicked when there are still unset parameters or no problem model (definition in chapter *2.3.2.3 The Problem Model* on page 23) has been defined yet.

2.3.3.3 Progress panel

By default, the progress panel shows only a message stating there is currently no optimisation in process. It is only when the run button is clicked that the panel displays the optimisation progress in a progress bar with a percentage declaration. Below the progress bar, a short description also states the stage the optimisation algorithm is currently in.

The percentage for the progress bar is derived from the iteration count of the algorithm. The iteration count corresponding to 100 % is calculated from the iterations specified by the chosen optimisation mode.

The progress bar also provides a cancel button in the lower right corner. Cancelling stops the running optimisation process and it returns the currently found optimal solution. This solution is then displayed in the solution display panel and set as the latest solution in the running project.

2.3.3.4 Solution display panel

The solution display panel displays the data stored in the table models (see chapter *2.3.4 The utils package* on page 34) of the project that is currently open in the Route Optimizer. In other words, it displays the general information about the found solution and the route schedules of the solution. Each route schedule is displayed in a separate tabbed pane in order to save screen space and since the number of routes may vary.

Additionally, if the currently open project supports optimisation by weekdays, meaning it is a real-world project, in that case, one can switch between displaying the route schedules of each weekday.

2.3.3.5 Icon

The icon (see Figure 18) which was used as the application's logo is the result of the idea to condense the purpose of the application into one icon. The purpose is simple: given a set of clients, efficiently compute a route configuration that minimises a chosen measure. Surprisingly, a single three-point polyline is perfectly sufficient. For once, it is the simplest abstract route possible[1]. Secondly, if one interprets the line as a graph, it shows a minimum. And lastly, its most apparent meaning, a checkmark, symbolises efficiency.

Figure 18: Icon of the Route Optimizer application.

[1] It needs at least one intermediate point to distinguish it from a simple connection.

2.3.3.6 Language

As the targeted users are employees of a Swiss medical laboratory, the language was set to German. But unfortunately, there are still unsolved issues regarding this matter at the application's current state. For example the tool tip messages for closing, minimizing, and maximising a window are still shown in English.

2.3.4 The utils package

The utils package contains practically stand-alone classes and methods that would be reusable in other applications. Only for the TableModel class the Java Swing library is needed. The following table gives a short overview:

Class	Short description	Used in
TableModel	An implementation of the Java Swing AbstractTableModel used to display tables.	VRPTW Solver: Solution, Route Project: Project, ProblemModel Visuals: GUI
Time	Very basic data structure to represent time (hh:mm:ss) and time periods. Supports Addition/subtraction, multiplication/division with Integers, and comparison to other Time.	VRPTW Solver: Problem, Solution, Route Project: Project, ProblemModel, Client, Depot, IntermediateDepot, Visit Utils: TimeWindow
TimeWindow	Very basic data structure to represent a time interval. Supports comparison to other TimeWindows and Time.	Project: Project, ProblemModel, Depot, IntermediateDepot, Visit
Weekday	An enum (enumeration class) to represent the five weekdays.	Project: Project, Client Visuals: GUI
Utils	Collection of static methods: • Method for getting the system's new-line character • Method for sending an HTTP request	VRPTW Solver: Problem, Solution, Route Project: RealProblem, Client Visuals: GUI

Table 10: Classes in the utils package.

3 Results

This thesis analysed a variation of the VRPTW where clients need to be visited more than once. Generally, the time windows for these visits are not given though. Instead, there exist preferred times depending on the client's type. Still, there can be cases where there are fixed visits or where some specified time period must be avoided for a certain client. Also, in addition to starting and finishing its route at a unique depot, each vehicle needs to make an intermediate stop at that same depot during a given time window. The minimum duration of this intermediate stop is specified as well.

Further, this thesis proposed a possible implementation of an application, the Route Optimizer, to solve this variation of the VRPTW. Figure 19 shows a screenshot of the Route Optimizer's GUI.

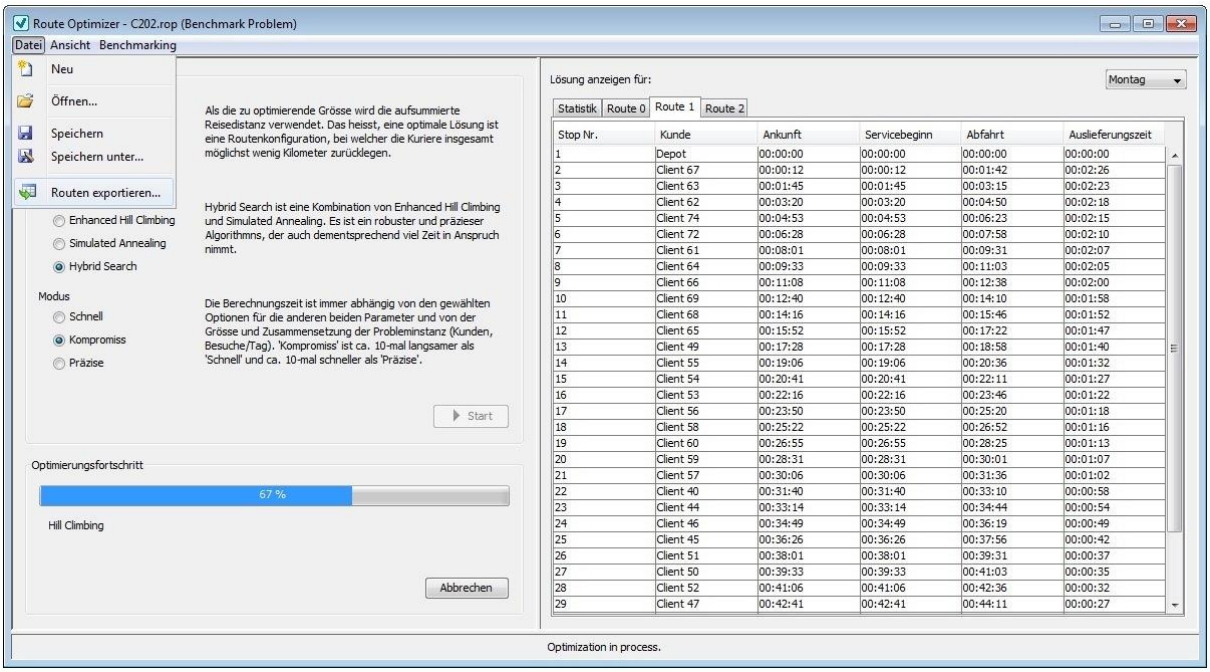


Figure 19: Screenshot of the Route Optimizer application during a running optimisation process.

The Route Optimizer works with project files that store the problem model defined by the user. These projects can be saved and opened again. The Route Optimizer also supports creating new empty projects. In addition to optimising real-world problem instances, benchmark problem instances can be loaded into a new project and processed with the same optimisation algorithms. In order to give the user a choice and to give the application certain flexibility, the user may choose between three optimisation methods, three measures to be optimised, and three modes. When running the optimisation, the Route Optimizer then displays the progress in a progress bar and also offers a possibility to cancel the optimisation. Finally, the found solution, resp. its route schedules, is displayed in a set of tables, one for each route plus an additional table for displaying general information about the solution. These tables can also be exported for further processing and visualisation.

The optimisation algorithms were tested with the Solomon benchmark instances and similar results as listen in de Oliveira's work (de Oliveira, et al., 2008) were achieved. The application was also tested with the ZLMSG data and a reasonable solution was found. However, publishing this solution was dispensed with because without the information about the client's type (see 2.2 Technical specifications and data used on page 9) the found solution is no longer reproducible.

4 Outlook

Although the application developed in this thesis is able to complete the tasks it was designed for, it still has plenty of room for improvement. Especially in terms of usability and application design, there were many issues that could not be considered in this thesis. The same goes for issues concerning the modelling of the constraints of the ZLMSG courier service.

Also, the Route Optimizer application has not been tested with respect to usability or usefulness in this thesis. Hence, this would definitely be one of the first issues to look at. The Route Optimizer application could be tested with other real-world data as well and a kind of survey could be conducted in order to find out whether the calculated solutions are practicable or not.

Additionally, one could implement an app that would access the route calculated for a certain courier and display it on a map. This would also offer the possibility to send a short message to the ZLMSG as soon as a courier falls behind schedule.

4.1 Usability improvements

The most important issue concerning usability improvements is the currently missing implementation of a way to edit the problem model directly in the Route Optimizer application. An idea could be to add another menu item to the edit menu that would open a similar window like the 'display clients' menu item in the view menu does now. But this window would additionally provide an edit button for every client that would open yet another window where the client's properties could be edited. For this the current application structure would most likely have to be adapted a little, especially concerning the serialisation and deserialization processes.

Further improvement is also possible with regard to the optimisation modes (definition in chapter *2.3.2.6 Optimisation mode* on page 29). Currently, the parameters of the three predefined modes are hard-coded. A more elegant way would be to provide an opportunity for the user to arbitrarily set the parameters defined in a mode. A quick algorithm would then estimate the time it would take to execute the optimisation with these parameters.

With the intention of giving more information about the progress of a running optimisation, the short description below the progress bar could be implemented differently in order to display more information. For example, it would be nice to know how many random restarts have already been performed. In order to document the entire optimisation process in detail, a log file could also be created.

Another issue concerning display is the format used when displaying time (time windows). Right now, the application always displays it as 'hh:mm:ss'. However, giving second-precise information might not always be needed. Therefore, some users might welcome the opportunity to leave away the seconds when displaying time. Implementing this functionality is less trivial than it looks though, as the way an instance of the Time class is displayed is simply defined through the 'toString' method. Maybe the application of an existing implementation of time (e.g. Joda Time library, new Time API for JDK 8) would offer an easier solution.

It would be nice to solve the language issue discussed in chapter *2.3.3.6 Language* on page 34. Additionally, the usability of the application could be increased, if the user was able to choose the language used in the GUI. The control for this functionality would have to be in the view menu.

Finally, there are several basic issues that still need to be resolved. For example, currently, the user must remember to save a project before closing the application or opening a new project. If he doesn't, the old project will be discarded without warning. The implementation of tool tip messages would also improve the usability of the Route Optimizer. Additionally, the bottom bar for displaying information (see chapter *2.3.3 The Visuals module* on page 30) could be expanded. For instance, it could show a small progress bar when opening a new project. And it could display the numbers of clients and/or visits of the current project. And one could think about whether it would be beneficial to ask for confirmation whenever a previously found optimal solution is to be overwritten with a new solution that is worse than previous one. However, one would have to define what a 'worse' solution actually is, since the optimisation measure used to find the previous and the new solution might not be the same.

4.2 Improvements concerning the implementation of the application

In its current sage, the architecture of the application mainly consists of the three modules introduced in chapter *2.3 Application architecture and development process* (page 10): one for coordinating the interaction with the user (Visuals), one for solving a VRPTW instance (VRPTW Solver), and one for cording the interactions between these two modules (Project). Yet the application is actually started from the main method in the GUI class of the Visuals module. This is a contradiction that could be solved by introducing a coordinator class that does not belong to any module and would administer the entire Route Optimizer application. It would store the currently running project, an instance of the GUI class and the Optimizer class, and would be responsible for opening, saving, and creating projects. The Project module's only role would then be to provide the file concept. However, this solution would require a lot more event based programming.

Another structural improvement would to move the distinction between a benchmark and real-world instance from the project to the problem model. In fact, it is the problem which is a benchmark instance, and not the project.

In the problem model, the classes Depot, Client, and IntermediateDepot share a lot of code, especially fields. Thus it would be good practice to collect these shared features in a superclass. However, this would again require writing custom serializers and deserializers for the GsoBuilder which was used to save and open project files (see chapter *2.3.2.4 Creating, opening, and saving projects* on page 27).

Further improvement of the optimisation algorithms could also be attained by increasing the probability of applying the Heuristic Mutation operator $P_{HM}(x)$ when exploring a solution's neighbourhood. In the current implementation all five neighbourhood operators are applied with equal probability ($P_{HM}(x) = 0.2$). However, studies suggest that the quality of the final solution will increase, if $P_{HM}(x)$ is increased (de Oliveira, et al., 2008).

The solutions found by the current implementation of the VRPTW Solver module contain route schedules where the service begins are always set to the earliest possible time. This is perfectly fine when the optimisation measure is the distance travelled. However, when a solution is to be optimised with respect to travel time or delivery time, it could be beneficial to set the service begin to the latest possible time. To solve this issue, the implementation of a simple post-processing step where the service begins inside each route are shifted to the last possible time would be enough (Solomon, 1987).

In order to reduce the average delivery time, the concept of an intermediate depot (definition in chapter *2.3.2.3.1 Intermediate depots* on page 24) was introduced in this thesis. This idea could be developed further. For example, one could analyse whether the insertion of additional intermediate depots into the routes of a found solution can further reduce the average delivery time.

Lastly, the current implementation for taking into account the different client configurations on each day of the week may be quite inefficient, especially in cases where the exact same client configuration is used for more than one weekday. In these cases, multiple optimisations are run for the exact same problem instances. Such redundancy could be avoided, if the problem instances generated for each weekday were checked for equality prior to the execution of the optimisation algorithms.

4.3 Improvements regarding the modelling of the ZLMSG courier service's constraints

It was stated in this thesis that the use of hard time windows does not model the true situation in the real world. Instead, soft time windows would be more suitable. In order to realise this, the divergence from the time window could be included in the cost function when inserting a new customer into the route instead of using the time windows as a hard constraints. There also exist several works and papers about this topic.

But there are also issues to consider, if one were to stay with the current model with hard time windows. The algorithm used to calculate a client's visits, resp. its time windows, is implemented specifically for the ZLMSG's constraints. With the aim of making the Route Optimizer applicable for a broader problem space, this algorithm could be replaced by a more generic one.

However, as stated in chapter *2.3.2.3.3.1 Generating the default distribution (DD) for a ZLMSG client* (page 25), the scheduling of visits is actually an optimisation problem in itself anyway. Therefore, it should be considered using a different method altogether, i.e. an optimisation algorithm. The measure to be minimised would then be the standard deviation of the time in between the visits, resp. its time windows.

The Route Optimizer application at its current stage does not support the constraint of a maximum number of vehicles even though this is part of the VRPTW definition and probably is an essential feature for many real-world applications. It could be realised by implementing a check box in the controls panel that would activate an entry field, if checked. The user would then be asked to enter the maximum number of vehicles into that entry field and this number would be passed as an additional constraint to check for when accepting neighbourhood solutions (see chapter *2.3.1.2.3 Neighbourhood operators* on page 17).

5 List of references

de Oliveira, Humberto César Brandão and Vasconcelos, Germano Crispim. 2008. *A hybrid search method for the vehicle routing problem with time windows.* US : Springer Science+Business Media, 2008. 0254-5330.

Google. Google code. *google-gson.* [Online] [Zitat vom: 15. 05 2015.] https://code.google.com/p/google-gson/.

Houghton Mifflin Company. *The American Heritage® Science Dictionary.* [Online] [Zitat vom: 27. 05 2015.] http://dictionary.reference.com/browse/GUI.

JSON Group. JSON. *JSON.* [Online] [Zitat vom: 26. 05 2015.] http://json.org/.

MapQuest, Inc. MapQuest Open Directions API Web Service. *mapquest developers.* [Online] MapQuest, Inc. [Zitat vom: 24. 05 2015.] http://developer.mapquest.com/web/products/open/directions-service.

Neller, Todd W. 2005. *Teaching Stochastic Local Search.* Computer Science, Gettysburg College. Gettysburg : American Association for Artificial Intelligence, 2005. S. 6.

Networking and Emergency Optimization. *Networking and Emergency Optimization.* [Online] [Zitat vom: 26. 04 2015.] http://neo.lcc.uma.es/vrp/vrp-instances/capacitated-vrp-with-time-windows-instances/.

Solomon, Marius M. 1987. *Algorithms for the Vehicle Routing and Scheduling Problems with Time Window Constraints.* Boston, Massachusetts : Operations Research Society of America, 1987.

Thangiah, Sam R, Osman, Ibrahim H und Sun, Tong. 1994. *Hybrid Genetic Algorithm, Simulated Annealing and Tabu Search Methods for Vehicle Routing Problems with Time Windows.* 1994.

The Apache Software Foundation. Apache POI Project. *Apache POI - Download Release Artifacts.* [Online] [Zitat vom: 16. 05 2015.] https://poi.apache.org/download.html.

Appendix A – Glossary

Term	Abbreviation	Explanation
Vehicle Routing Problem with Time Windows	VRPTW	An optimisation problem belonging to the class of NP-hard problems.
Time-feasibility		A route is time-feasible, if the time windows of all customers in the route can be adhered to. A solution is time-feasible, if all routes of the solution are time-feasible.
Insertion		An insertion specifies the customer to be inserted as well as the route and the position in the route where the customer is to be inserted. An insertion may be time-feasible or time-infeasible. Setting an insertion means inserting the specified customer in the specified route and position.
Time window	TW	Time interval during which the 'service' (as defined by the VRPTW) must take place.
Visit		Representation that essentially stands for the time window during which a visit needs to be paid to a client.
Predefined Visit	PV	A predefined visit is a visit that has not been calculated by the application but was defined by the user. A predefined visit may be a fixed visit or a reference visit
Fixed visit		A fixed visit is a predefined visit to which the fleet of vehicles must deliver a service. (I.e. the visit was specified by the customer.)
Reference visit		A reference visit is a predefined visit to which the fleet of vehicles does not deliver a service (i.e. visits paid by the KSSG courier.)
Default distribution	DD	Default visit schedule for a client depending on the client's type. It is 'default' in the sense that predefined visits have not been considered yet.
Graphical User Interface	GUI	'An interface that is used to issue commands to a computer by means of a device such as a mouse that manipulates and activates onscreen images.' (Houghton Mifflin Company)
Optimisation measure		The measure to be optimised during the optimisation process.
Distance		One of the three optimisation measures: the total distance travelled.
Travel time		One of the three optimisation measures: the total time taken.
Delivery time		One of the three optimisation measures: the average travel time from a customer to the next depot.
VRPTW Solver module		The part of the application developed in this thesis that is responsible for solving a given problem instance of the VRPTW.
Project module		The part of the application developed in this thesis that is responsible for operating the application and providing a file concept for creating, saving, and opening work sessions.
Visuals module		The part of the application developed in this thesis that is responsible for coordinating the interaction with the user.
Problem model		Model of a courier service configuration or a VRPTW instance.

Term	Abbreviation	Explanation
Client		Model of a client or customer in the real world.
Customer		The abstract customer concept used in the definition of the VRPTW.
Intermediate depot		Concept for a mandatory return to the depot in each route of a solution.
Courier service configuration		Synonym used to refer to real-world problem instances of the VRPTW or small variations of it.
Zentrum für Labormedizin St. Gallen	ZLMSG	The courier service looked at in this thesis belongs to the ZLMSG.
Kantonsspital St. Gallen	KSSG	
JavaScript Object Notation	JSON	A lightweight data-interchange format. It is both, easy for humans to read and write and easy for machines to parse and generate. (JSON Group)

Appendix B – Data ZLMSG

Route Schedules:

Vehicle 1	Time	
Start: Depot	07:30:00	
Client 1	08:00:00	
Client 2	08:25:00	
Client 3	08:50:00	
Client 4	06:00:00	
Client 5	10:00:00	
Client 6	10:25:00	
Client 7	10:45:00	no service on Wednesday
Client 8	10:55:00	
Client 9	11:15:00	
Client 10	11:50:00	
End: intermediate Depot	12:15:00	
Start: intermediate Depot	13:30:00	
Client 11	14:00:00	
Client 12	14:10:00	
Client 13	14:20:00	
Client 14	15:00:00	
Client 15	15:15:00	
Client 16	15:35 -15:40	
Client 17	16:00:00	
Client 18	16:05:00	
Client 19	16:10:00	
Client 20	16:20:00	service only on Tuesday and Friday
End: Depot	16:30:00	

Vehicle 2	Time
Start: Depot	10:20:00
Client 21	11:10:00
Client 22	11:15:00
End: intermediate Depot	11:30:00
Start: intermediate Depot	15:10:00
Client 23	15:25:00
Client 11	10:40:00
Client 24	10:50:00
Client 25	11:00:00
Client 21	11:10:00
Client 22	11:15:00
End: Depot	11:25:00

	Time	
Vehicle 3	Mo, Di, Do	Mi, Fr
Start: Depot	12:45:00	12:45:00
Client 26	13:45:00	13:45:00
Client 27	14:00:00	14:00:00
Client 28	14:25:00	14:25:00
Client 3	14:45:00	14:45:00
Client 29	15:15:00	15:15:00
Client 30	15:30:00	15:30:00
Client 2	15:45:00	15:45:00
Client 31	-	16:00:00
Client 1	16:10:00	16:20:00
End: Depot	16:35:00	16:45:00

Client table:

Pseudo-name	number of visits by courier	reference visits	fixed visits	predefined visits	duration of stay [min]	days of visit
	1	0	0	0	5	daily
Client 1	2	0	0	0	5	daily
Client 2	2	0	0	0	5	daily
Client 3	2	0	0	0	5	daily
Client 4	1	0	0	0	5	daily
Client 5	1	0	0	0	5	daily
Client 6	1	0	0	0	5	daily
Client 7	1	0	0	0	5	Mon, Tue, Thu, Fri
Client 8	1	0	0	0	5	daily
Client 9	1	0	0	0	5	daily
Client 10	1	0	0	0	5	daily
Client 11	2	1	0	1	5	daily
Client 12	1	0	0	0	5	daily
Client 13	1	0	0	0	5	daily
Client 14	1	0	0	0	5	daily
Client 15	1	0	0	0	5	daily
Client 16	1	0	1	1	5	daily
Client 17	1	0	0	0	5	daily
Client 18	1	0	0	0	5	daily
Client 19	1	0	0	0	5	daily
Client 20	1	0	0	0	5	Tue, Fri
Client 21	2	0	0	0	5	daily
Client 22	1	0	0	0	5	daily
Client 23	1	0	0	0	5	daily
Client 24	1	1	0	1	5	daily
Client 25	1	1	0	1	5	daily
Client 26	1	0	0	0	5	daily
Client 27	1	0	0	0	5	daily
Client 28	1	0	0	0	5	daily
Client 29	1	0	0	0	5	daily
Client 30	1	0	0	0	5	daily
Client 31	1	0	0	0	5	Wed, Fri
Client 32	1	0	0	1	60	daily

Appendix C – Solomon instances file structure

<name>

<vehicle number> <capacity>

<customer number> <xCoord.> <yCoord.> <demand> <ready time> <due date> <service time>

<client 0 is the depot>

Appendix D – Application structure

- math
  - optimization
    - Parameters.java
    - OptimizationMethod.java
    - Optimizations.java (static version)
    - Optimizer.java (runnable version)
    - State.java
- routeOptimizer
  - project
    - BenchmarkProblem.java
    - OptimizationMode.java
    - Project.java
    - RealProblem.java
    - problemModel
      - Client.java
      - Depot.java
      - IntermediateDepot.java
      - ProblemModel.java
      - Visit.java
  - visuals
    - GUI.java
    - *images*
  - vrptw
    - Customer.java
    - Insertion.java
    - OptimizationMeasure.java
    - Problem.java
    - Route.java
    - RouteStop.java
    - Solution.java
- Utils
  - TableModel.java
  - Time.java
  - TimeWindow.java
  - Utils.java
  - Weekday.java