UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

A Simulation and Visualization Tool for the Dynamic and Stochastic Vehicle Routing Problem with Time Windows

Dissertation presented by
Simon CANDAELE

for obtaining the Master's degree in
Computer Science and Engineering

Supervisor(s)
Yves DEVILLE

Scientific Advisor(s)
Michael SAINT-GUILLAIN

Reader(s)
Yves DEVILLE, Michael SAINT-GUILLAIN , Pierre SCHAUS

Academic year 2017-2018

Abstract

The Vehicle Routing Problem consists in optimizing a set of routes for a fleet of vehicles in order to serve a set of requests issued by customers. Many variants exist, including dynamic variants where the requests appear while the vehicles are moving. It is then necessary to modify the existing solution to take into account the new requests that appear. These problems are heavily studied by the scientific community and many algorithms have been proposed to solve them. When implementing a solver for a dynamic problem, it is imperative to be very careful when each request is 'revealed' and can then be taken into consideration to build a solution. The first goal of this master thesis is to provide a tool that simulates the Dynamic Vehicle Routing Problem to a solver, so that the solver no longer has to deal with the simulation of the problem. In addition to simplifying the development of new solvers, this tool also aims to be used to perform simulations, recover all the solutions returned by a solver during a simulation, and thus facilitate the analysis of solver performance. Finally, the tool we propose also make it possible to visualize a problem instance as well as the solutions returned by a solver. The tool allows to visualize the simulation process, showing the current solution, the position of the vehicles and the routes assigned to them. In this these, we also show how to use the tool and how to integrate it into a solver.

Acknowledgements

After a long and intensive work, I would like to thank all the people who helped and supported me during this master thesis. I would like to express my sincere gratitude to my supervisor Professor Yves Deville and to the assistant Michael Saint-Guillain without whom this work could not have been carried out successfully. I also thank them for the comments and advice, numerous and very helpful, that they have given me throughout my work.

My most sincere thanks are for Michael who has devoted a huge amount of time to help me throughout this master thesis. Without his availability and commitment, this work would not have been possible. I can never thank him enough for his precious help.

Finally, I have a thought for my family and friends who have always supported me.

Contents

1 Introduction **1**

2 Problem Description **3**

2.1 The Vehicle Routing Problem 4

2.2 The Vehicle Routing Problem with Time Windows 5

2.3 Extensions of the VRP 7

2.3.1 Multiple Depots 7

2.3.2 Heterogeneous fleet 7

2.3.3 Multiple requests per nodes 8

2.4 VRP with dynamic requests 8

2.4.1 Waiting points 10

2.5 The Dynamic and Stochastic Vehicle Routing Problem with Time Windows . . . 11

3 Descriptions of the tool's functionnalities **13**

3.1 General Overview and main usage 13

3.1.1 Presentation of the tool 13

3.1.2 The command line simulator 15

3.1.3 Using scripts 16

3.1.4 The graphical interface of the simulator 17

3.1.5 Displaying soutions on a real map 18

3.1.6 Functionnalities offered to the solver 18

3.2 Input files, Output file and simulated problems 19

3.2.1 The Graph file 20

3.2.2 The Carrier file . . . 21
3.2.3 The Customer file . . . 23
3.2.4 The Scenario file . . . 27
3.2.5 The Solution format . . . 29

4 How to use the tool **31**

4.1 Using the simulator in command line . . . 31
4.2 Using a script . . . 34
4.3 Using the Graphical User Interface . . . 35
4.4 Displaying on an interactive map . . . 37
4.5 The API . . . 38

5 Implementation of the tool **43**

5.1 General Overview . . . 43
5.2 Communication between the simulator and the solver . . . 45
5.2.1 The SimulatorMessages service . . . 45
5.2.2 The SolverMessages service . . . 47
5.3 The Simulator Components . . . 48
5.3.1 User Input Manager and Script Manager . . . 48
5.3.2 the Graphical User Interface . . . 48
5.3.3 The Simulation Manager . . . 49
5.3.4 The Simulation Process . . . 50
5.3.5 The communication API of the simulator . . . 50
5.4 The interactive web browser map . . . 51
5.5 The Solver API . . . 51
5.6 The mechanism to start a simulation . . . 52

6 Proof of Concept **54**

6.1 Data Sets . . . 54
6.1.1 Solomon data set . . . 54

6.1.2 Lyon data set 55
6.1.3 Turin data set 56
6.2 Experiments 56
6.2.1 Solomon case study 57
6.2.2 Lyon case study 61
6.2.3 Turin case study 62
6.3 Conclusion 63

7 Conclusion **65**

A Graphical User Interface Screenshot **69**

B Source Code Parts **76**

B.1 Skeleton code to use the API 76
B.2 Simulator scripts used in the Proof of Concept 78

C Quantitative data about the code **81**

Chapter 1

Introduction

Nowadays, the transport and delivery of goods or services have become omnipresent in our modern societies. It creates a need to optimize routes to minimize costs, economic or environmental, and achieve objectives specific to each situation. For about fifty years now, the scientific community has been working on this family of problems, which it has named the *Vehicle Routing Problems* or VRP. These problems consist initially in determining a routing plan for a set of vehicles in order to visit a set of customers while optimizing an objective function that generally require minimizing the total distance travelled by vehicles.

Over time, the formulations and variants of the VRPS have evolved considerably in order to get closer and closer to real life situations. In the earliest formulations of the problem, vehicles were already modeled with finite transport capacity that limits the number of customers that can be served by a single vehicle. The vehicles also had to start and finish their route at a location called the depot. A major variant of the VRP is to associate the customers with time windows defining when a vehicle can visit a given customer in order to serve him. These variants are called the *Vehicle Routing Problem with Time Windows* or VRPTW. Additional constraints can be added to the formulation such as the use of several depots to store vehicles, or the use of a heterogeneous fleet where each vehicle has a different transport capacity and moves at its own speed between the different locations involved in the problem. Another very important category of VRP are the *Dynamic Vehicle Routing Problems* or D-VRP. In these problems, customers may appear while the vehicles are performing the routing plan that has been computed. In order to serve these new customers, it is therefore necessary to adapt the routing plan currently applied so that one or more vehicles change their future itinerary to serve the new demands. Other variants of the VRP model some parameters as random variables. These problems are called *Stochastic Vehicle Routing Problem* of S-VRP. The parameters that can be modelled in a more or less random way are numerous and vary from one problem to another. These can be customer-specific parameters, such as the amount of goods to be delivered, or the time window to visit it. It may also include vehicle-specific parameters such as the travel time between two locations at a given time. It is sometimes possible to take into account the probabilities associated with these random variables in order to try to anticipate the values they will take. All this brings us to a final variant, which combines all the one we have mentioned, the *Dynamic and Stochastic Vehicle Routing Problem with Time Windows* or DS-VRPTW.

In this master thesis, we design and implement a tool to simulate an instance of a *Dynamic and Stochastic Vehicle Routing Problem with Time Windows* to a solver, in order to separate the simulation of the problem from its resolution. The formulation of these problems is complex and involves many constraints on when the various information describing the problem must be

known by the solver. The tool we propose facilitates the development and implementation of a solver for problems in the family of DS-VRPTW by offering a toolbox that is easy to integrate into a solver. The tool simulates a DS-VRPTW by communicating to the solver all information related to the problem at the right time during its execution so that the solver does not have to bother with this task. The simulator is therefore responsible in particular for communicating to the solver the customers who appear while the vehicles are moving. The tool also recovers all the solutions returned by a solver during the simulation of a problem.

The tool that we propose also makes it possible to visualize the solutions during a dynamic problem and to see them evolve during the execution of a problem. This forms thus a simulation and visualization tool for the DS-VRPTW, making it easier to develop and test solvers, and to visualize problems and solutions proposed by a solver.

This dissertation is organized as follows. In Chapter 2, we present the different variants of the *Vehicle Routing Problem* that lead to the variant we call the *Dynamic and Stochastic Vehicle Routing Problem with Time Windows*. This chapter begins with section 2.1, which introduces the basic variant of the VRP. We then introduce the VRP with Time Windows in section 2.2 and additional variants in section 2.3. The section 2.4 discusses the VRP with dynamic requests. Section 2.5 closes the chapter by adding stochastic data to the formulation of the problem, thus forming the *Dynamic and Stochastic Vehicle Routing Problem* we are interested in. Chapter 3 is intended to provide an overview of the tool and its functionality. First, the functionalities are described in section 3.1, then the problems that can be simulated and the files to describe them are presented in section 3.2. In Chapter 4, we explain how to use the tool. The simulator itself can be used with three different interfaces described in the sections 4.1, 4.2 and 4.3. Section 4.4 shows one way to display solutions on an interactive map. Section 4.5 closes the chapter by explaining how a solver must do to use the simulator. The implementation of the tool is presented in the chapter 5. This chapter begins with an overview of the architecture in section 5.1. Section 5.2 explains how the solver communicates with the simulator. Section 5.3 discusses the components of the simulator. We explain how the solutions are displayed on an interactive map in the section 5.4. Details about the toolset developed specificaly for the solver to communicate with the simulator are presented in section 5.5. The chapter concludes with Section 5.6, which presents an example illustrating the interactions between different components of the system. In Chapter 6, we perform several case studies on different datasets using our tool to perform a proof of concept. First we present the dataset used in the section 6.1, then we show our experiments in the section 6.2 and we comment the whole in section 6.3. Finally, chapter 7 closes this dissertation with a conclusion on the work done as well as on the possibilities of future work to continue what we have started.

We also would like to point out that the tool is available at the following address: `https://github.com/SimonCandaele/simulator_VRP`

Chapter 2

Problem Description

In this chapter, we will present the *Vehicle Routing Problem*, or VRP, and some of its variants. The objective is not to review all existing variants but to present the different variants that lead to the one that can be simulated with the tool, which we present in the following chapters.

The VRP was first introduced by [12], as a generalization of the well-known *Traveling-Salesman Problem* or TSP. It consists in finding the shortest path linking a set of geographical points and returning to its starting point. Figure 2.1 presents an example of a TSP and what a solution looks like.

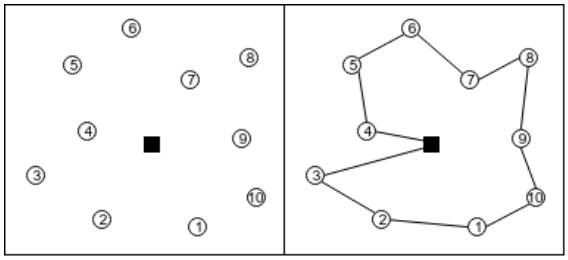


Figure 2.1: Example of a *Travelling Salesman Problem* : On the left is the problem to solve. A route passing by each circle one time, starting and finishing at the black square has to be found. On the right is an example of such a route.

The VRP generalizes the TSP in the sense that there is no longer one, but several routes that must be found to connect the points. Figure 2.2 illustrates the concept of VRP by generalizing the example of TSP in figure 2.1. The points are now connected by three routes instead of one.

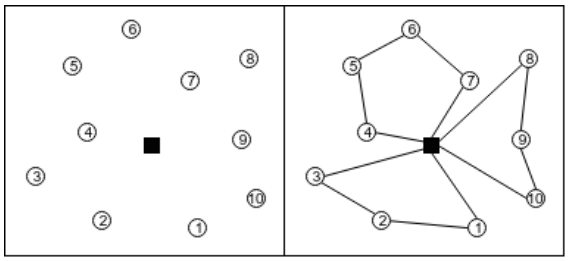


Figure 2.2: Example of a *Vehicle Routing Problem* : On the left is the problem to solve. The example solution contains three routes that each connect different circles. Each route begins and ends at the black square.

It is well known that the *Travelling salesman problem* is a NP-complete problem. As a

natural generalization of the TSP, the VRP is by nature a NP-complete problem. No exact algorithm solving a NP-complete problem in polynamial time has ever been found. It is therefore highly probable that there is no exact algorithm solving the VRP in a polynomial time. In practice, an exact algorithm for this kind of problem will always have a complexity that grows exponentially with the size of the problem to solve. To solve a VRP of realistic size (e.g. hundreds of clients or more), algorithms using approximations or heuristics are generally used.

The chapter is organized as follows. First of all, section 2.1 describes a first basic formulation of the *Vehicle Routing Problem*. Second, we present the variant named *Vehicle Problem with Time Window* in the section 2.2. Then the section 2.3 introduces several extensions of the VRP. Section 2.4 presents the *Dynamic Vehicle Routing Problem*. This leads to the section 2.5 that summarizes all the extensions previously introduced to form the VRP variant that we consider in the next chapters : the *Dynamic and Stochastic Vehicle Routing Problem with Time Windows* noted DS-VRPTW.

2.1 The Vehicle Routing Problem

[12] was the first to study the *Vehicle Routing Problem*. It is this variant, which we call the *Capacitated Vehicle Routing Problem* or CVRP, that we will describe here, using our own formulation. Note also that, unless otherwise stated, the variants we will present later are all capacitated variants, or can be trivially extended to be so. We will therefore not always use the letter C in the acronyms to indicate a capacitated variant of VRP.

In this variant, we consider a set of locations or clients where a certain quantity of a good must be delivered or collected. In order to serve these customers, a set of vehicles is available. However, vehicles have a limited capacity, and therefore cannot deliver more good than they can carry. The vehicles are kept at a location called the depot, where they must start and end their route. The goal of the problem is therefore to find a set of routes, so as to visit each customer while minimising the total travel time and not exceeding the capacity of the vehicles.

Solution

The CVRP applies to a set of nodes $V = \{0, 1, ..., N\}$. The node 0 is the depot and the others are the customers. A travel cost c_{ij} is associated to each pair of nodes $v_i, v_j \in V$. The costs are not necessarily symmetric. Depending on the context, c_{ij} can be interpreted as the distance or the travel time to join node j starting from node i. Each customer $i \in V \setminus \{0\}$ is associated with a quantity or demand q_i to be delivered by a vehicle.

A vehicle route is a sequence of nodes $\lambda =< v_1, ..., v_{|\lambda|} >, v_i \in V$ where $|\lambda|$ is the number of elements in the sequence. $|\lambda|$ is also called the length of the route. In the example of figure 2.2, there are thre routes that can be noted $\lambda_1 =< 1, 2, 3 >$, $\lambda_2 =< 4, 5, 6, 7 >$ and $\lambda_3 =< 8, 9, 10 >$.

$cust(\lambda) = \{v : v \in \lambda, v \in V \setminus \{0\}\}$ is the set of customers in the route λ.

There are K vehicles that can be used to serve the clients. They each have the same defined capacity Q. A routing plan ρ is a set of K routes $\{\lambda_1, \lambda_2, ..., \lambda_K\}$ with one route for each vehicle. The route λ_i being associated to the i-th vehicle, with $1 \leq i \leq K$. A solution is a routing plan ρ that meets the constraints in the next section. An optimal solution is a solution that minimizes

the total cost of the solution, $\sum_{\lambda \in \rho} \sum_{i=1}^{|\lambda|-1} c_{v_i, v_{i+1}}$.

Constraints

The capacity of each vehicle route does not exceed the vehicle capacity.

$$\forall \lambda \in \rho \ : \ \sum_{i \in cust(\lambda)} q_i \leq Q \tag{2.1}$$

Each customer is served exactly once by one vehicle.

$$\bigcap_{\lambda \in \rho} cust(\lambda) = \emptyset \quad \wedge \ \bigcup_{\lambda \in \rho} cust(\lambda) = V / \{0\} \tag{2.2}$$

The first and last node of each route is the depot.

$$\forall \lambda \in \rho \ : \ v_1 = v_{|\lambda|} = 0 \tag{2.3}$$

A practical example

Let us now describe how this problem can be encountered in everyday life. An example is a company with a large warehouse where it stores large quantities of the product it manufactures. The company also owns several stores in the city where it sells this product. Every day, the stores have to be supplied and they each need a different quantity of the product. To serve the stores, the company has a fleet of trucks at its depot. So every day, a route has to be found for each truck so that they can replenish the shops. The company can find the best solution to its delivery problem by solving the VRP corresponding to its situation.

2.2 The Vehicle Routing Problem with Time Windows

A very common variant of the VRP is the *Vehicle Routing Problem with Time Windows*, or VRPTW, which introduces a time dimension to the problem. In this variant, each customer is associated to a time window and a service duration. To serve a customer, a vehicle must begin to serve the customer during the time window associated to the customer, for a period of time greater than or equal to the service duration of this customer. The vehicle can arrive to the customer before the beginning of the time window, but have to wait until the start of the time window to serve the customer. [6] is a nice review of algorithms for solving the VRP with time windows constraints. [24] and [11] are also interesting to read about the VRPTW.

Solution

The VRPTW applies to a set of nodes V, as we defined it for the CVRP. The travel time to reach node j from node i is given by c_{ij}.

Each node $i \in V$ is associated with a service duration d_i and a time window $[e_i, l_i]$ with $0 \leq e_i < l_i$. The time window is the time interval during which a vehicle can arrive at the node

and begin to serve it. For the depot node, the service duration is zero $d_0 = 0$, and the time window is defined as $e_0 = 0$, $l_0 = +\infty$.

A vehicle route is a sequence of actions noted

$$\lambda =< a_1, a_2, ..., a_{|\lambda|} >=< (v_1, t_1^-, t_1^+), (v_2, t_2^-, t_2^+), ..., (v_{|\lambda|}, t_{|\lambda|}^-, t_{|\lambda|}^+) >$$

An action $a_i = (v_i, t_i^-, t_i^+)$ is a tuple with different values. The node visited during the action a_i is indicated by v_i. The arrival and departure time are given by t_i^- and t_i^+, they indicate respectively when the vehicle reaches and leaves the associated node v_i.

We define some important functions. $v(a)$ gives the node associated to the action a. $t^+(a)$ and $t^-(a)$ give respectively the values t^+ and t^- of the action a. The function $d(a)$ gives the service duration of the node $v(a)$. For two successive actions a_i and a_j, the travel time between the nodes of these two actions is noted $c_{v(a_i),v(a_j)}$. We simplify this notation by introducing the function $c(a_i, a_j) = c_{v(a_i),v(a_j)}$.

A routing plan ρ is a set of route $\{\lambda_1, \lambda_2, ..., \lambda_K\}$ containing K routes. There is one route for each vehicle. A solution is a routing plan that respects the constraints described in the following section. The best solution is the one that optimizes the objective function defined by the user. The aim is often to minimise the total distance travelled by all vehicles.

Constraints

First of all, constraints (2.1), (2.2) and (2.3) described in section 2.1 must be respected. The additional constraints are as follows:

For any route λ in a routing plan ρ, for any action a_i in λ, the arrival time always precedes the departure time.

$$\forall \lambda \in \rho, \forall a \in \lambda \ : \ t^-(a) < t^+(a) \tag{2.4}$$

For two successive actions, a_i and a_{i+1}, in a route λ of a routing plan ρ, the time between the depature time $t^+(a_i)$ and the arrival time $t^-(a_{i+1})$ is always greater than or equal to the travel time $c(a_i, a_{i+1})$ between the associated nodes.

$$\forall \lambda \in \rho, \forall a_i, i < |\lambda| \ : \ t^+(a_i) + c(a_i, a_{i+1}) \leq t^-(a_{i+1}) \tag{2.5}$$

For each action a in a route λ of a routing plan ρ, the vehicles stays at the node for a time interval greater or equal to the service duration of the node. This time interval cannot start before the beginning of the time window $e_{v(a)}$, but the vehicle can arrive to the node before. It is not possible to arrive after the end of the time window $l_{v(a)}$.

$$\forall \lambda \in \rho, \forall a \in \lambda \ : \ \ t^-(a) < l_{v(a)} \ \ \wedge \ \ t^+(a) \geq max(t^-(a), e_{v(a)}) + d(a) \tag{2.6}$$

A practical example

To illustrate the VRPTW, take as an example a delivery company. More precisely, it is a company transporting packages by picking them up from the shippers and delivering them to the addresses indicated. Each shipper calls the company to indicate the weight of the package to be sent as well as the time window during which the company can pick up the package. The company will

pick up the package the next day to bring it back to the warehouse. The day after, the package will be delivered to the address indicated by the sender.

In this example, we can distinguish two VRP. The first concerns the collection of packages, and it is a VRPTW. The second is the delivery of the packages, the day after they are picked up. There are no time constraints for delivery, so it is a simple VRP.

2.3 Extensions of the VRP

We will now briefly introduce several additional extensions and constraints for the *Vehicle Routing Problem*. The extensions described in this section serve to give an overview of the diversity of variants of the VRP, and to introduce concepts that we will explain in detail in the section 2.5.

2.3.1 Multiple Depots

A first possible extension to the VRP is the possibility to have several depot nodes. Variants of VRP with multiple depot were studied among others by [14] and [27]. In these variants, the set of nodes of the graph can be written as $V = O \cup N$ where $O = \{o_1, o_2, ..., o_{|O|}\}$ is the set of depot nodes and $N = \{n_1, n_2, ..., n_{|N|}\}$ is the set of customers or client nodes. In a route λ, the first and the last node must belong to the set O, the other nodes must be elements of N. In some variants called Open VRP, vehicles do not have to return to a depot at the end of their tour. [26] and [15] address such problems.

We can reformulate the constraint (2.3) as

$$\forall \lambda \in \rho, \forall a_1, a_{|\lambda|} \in \lambda \ : \ v(a_1) \in O \wedge v(a_{|\lambda|}) \in O \tag{2.7}$$

2.3.2 Heterogeneous fleet

There are variants where the vehicles making up the fleet are not identical. Such variants have been studied by [25] and [5] among others. These are problems where vehicles have different capacities and move differently between nodes.

This can be modelled by defining the capacity and the travel costs independantly for each of the K vehicles. The capacity of the vehicle travelling the route λ_k is Q_k. The travel time of this vehicle between two nodes $i, j \in V$ is given by c_{ijk}.

Some variants also consider travel times varying with time. For these variants, the travel time for the vehicle associated to the route λ_k, to join node j from node i, leaving i at time t is given by c_{ijkt}

We update the function $c(a_i, a_j)$ that we introduced in section 2.2. We note it now $c_{k,t}(a_i, a_j) = c_{v(a_i),v(a_j),k,t}$ to take into account the vehicle and the departure time. This function now gives the travel time between the nodes associated with actions a_i and a_j for the vehicle k leaving at time t.

Some previously defined constraint have to be reformulated to consider the differences between the vehicles. The constraint (2.5) is now formulated as:

$$\forall \lambda_k \in \rho, \forall a_i, i < |\lambda_k| \ : \ t^+(a_i) + c_{k,t}(a_i, a_{i+1}) \leq t^-(a_{i+1}) \tag{2.8}$$

The constraint (2.1) is reformulated as:

$$\forall \lambda_k \in \rho : \sum_{i \in cust(\lambda_k)} q_i \leq Q_k \tag{2.9}$$

2.3.3 Multiple requests per nodes

Some problems studied assume that several requests or clients can be associated with the same location. [20] and [21] present problems with that possibility. A node can then be visited several times by one or more vehicles, so that the requests associated with it are served.

To model this, we no longer refer to the graph node as clients. Instead, a set of requests $R = \{r_1, r_2, ..., r_{|R|}\}$ is used. Each request r_i is described with a tuple $(q_i, v_i, d_i, e_i, l_i)$. It is then possible for two request r_i and r_j to be associated to the same node : $v_i = v_j$. The values d_i e_i l_i still describe the service duration and the time window of request r_i. The value q_i is the quantity of good to be delivered or collected at this node.

2.4 VRP with dynamic requests

Many variants of the VRP consider problems where input data is revealed while the solution is being executed. The current solution must then be adapted to produce a new solution that takes into account the new data. These variants are *Dynamic Vehicle Routing Problems* D-VRP. [18] and [19] are two excellent reviews of D-VRP.

[18] explains that the main source of dynamism in the vehicle routing problems is the arrival of customer requests during the execution of the routes by the vehicles. Other sources of dynamism can be the travel times, the demands of already known customers or the vehicle availability.

Concerning the dynamic problem, the execution is divided in two parts : the static (or offline) execution and the dynamic (or online) execution. At the beginning of the offline execution, a set of requests is known and a first routing plan can be computed. There is no dynamic event during the offline execution. At the beginning of the online execution, the last routing plan of the offline execution is applied. But new requests will appear while the vehicles are on their way. The goal during the online execution is to modify the partially applied routing plan to serve the new requests as they are revealed.

The time horizon

For dynamic problems in this paper, we will consider a continuous horizon ranging from 0 to H. All events of a problem and its solution, such as the revelation of a request or the movement of a vehicle, take place at a time t within this horizon. The events of the offline execution take place at time $t = 0$. The events of the online execution take place in the interval $0 < t \leq H$. The time horizon is divided into H time units $\{tu_1, tu_2, ..., tu_H\}$. Each time unit tu_i is a time interval, ranging from the time $t = i - 1$, not included, to the time $t = i$, included, : $tu_i =]i - 1, i]$. Figure 2.3 illustrates, with a timeline, how the horizon is divided in time units.

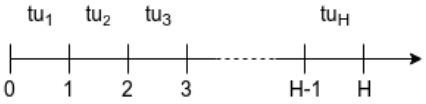


Figure 2.3: The time horizon and the time units.

The Scenario

The scenario is the set of requests we defined previously where each request is associated to a reveal time. To represent the scenario, we extend the notation of the set of requests R introduced in section 2.3.3. The scenario is noted $R = \{\tau_1, \tau_2, ..., \tau_{|R|}\}$. Each element τ_i of R is a tuple (r_i, t_i) where r_i is a request defined with the attributes we described in previous sections, and t_i is the associated reveal time of the request. The reveal time lies within the horizon : $0 \leq t_i \leq H$. If $t_i = 0$, the request r_i is known for the offline execution. If $0 < t_i \leq H$, the request r_i is an online request that will be known at time t_i.

At any time t, the set of requests R is split in two subsets: the set of revealed requests $R^t = \{r_i \in R : t_i \leq t\}$ and the set of hidden requests $R \setminus R^t$. Only the requests of R^t are known and can be included in a solution.

Solutions of a Dynamic VRP

As the requests are dynamicaly revealed while the vehicles are travelling, it is necessary to update the routing plan to serve all the requests. The solution of a dynamic VRP is a routing plan which can undergo modifications during the online execution of the problem.

We define σ as the set of successive routing plans that are applied during the execution : $\sigma = \{\rho^{t_1}, \rho^{t_2}, ..., \rho^{|\sigma|}\}$. Each routing plan ρ^{t_i} is issued at time t_i $(0 \leq t_i \leq H)$, and is applied until the following routing plan, $\rho^{t_{i+1}}$, is issued at time $t_{i+1} > t_i$. We introduce the function $pred(\rho)$ which gives the routing plan preceding the routing plan ρ. Each routing plan is a set of routes. The routes are defined in section 2.2, but we now add some precision in the notation.

The routing plan $\rho^t = \{\lambda_1, ..., \lambda_K\}$ issued at time t. There is one route for each vehicle. The route λ_k^t is the route associated to the k-th vehicle in the routing plan ρ^t and is a sequence of actions $\lambda_k^t =< a_1^t, a_2^t, ..., a^t_{|\lambda_k^t|} >$.

Each routing plan ρ^t is associated to a set A^t. This set is the set of requests served in the routing plan ρ^t.

The best solution is the solution that optimizes a function defined by the user while respecting the constraints described in the next section. The goal is often to maximize the total number of requests served. Additional objectives such as minimizing total travel time can also be targeted.

New Constraints

The dynamic VRP must first respect the constraints we previously defined. There are also new or reformulated constraints that we give here:

The constraint (2.2) is reformulated. Each request served is served by exactly one vehicle,

but not all requests are necessarily served anymore :

$$\bigcap_{\lambda \in \rho} cust(\lambda) = \emptyset \tag{2.10}$$

A routing plan must always serve all the requests served in a previous routing plan :

$$\forall t, t' \text{ such that } t < t' \ : \ A^t \subseteq A^{t'} \tag{2.11}$$

A routing plan ρ^t cannot change the destination of a moving vehicle at the time t it is issued:

$$\begin{aligned} \forall \rho^t, \rho^{t'}, \forall \lambda_k^t \in \rho^t, \lambda_k^{t'} \in \rho^{t'}, n \quad \text{such that} \quad & \rho^{t'} = pred(\rho^t) \\ & t^+(a_{n-1}^{t'}) < t \\ \text{we impose} \quad & v(a_n^t) = v(a_n^{t'}) \end{aligned} \tag{2.12}$$

A routing plan ρ^t must be identical to the previous routing plan for all the events prior to the time t it is issued:

$$\begin{aligned} \forall \rho^t, \rho^{t'}, \forall \lambda_k^t \in \rho^t, \lambda_k^{t'} \in \rho^{t'}, n \quad \text{such that} \quad & t' < t \\ & d^+(a_n^{t'}) < t \\ \text{we impose} \quad & a_n^t = a_n^{t'} \end{aligned} \tag{2.13}$$

The constraints we have just described here are commonly called the *nonanticipativity constraints* [23].

Example of a Dynamic VRP

We present an example of Dynamic VRP in figure 2.4. This example shows the execution of a D-VRP at three successive moments. At time $t = 0$, only a subset of customers is known and a first routing plan is applied. Four new customers are revealed at time t'. To serve them, the first routing plan partly executed has to be modified. One planned section in each route is deleted, and several new sections are added. The new routing plan is completely executed when the execution ends at time $t = H$.

2.4.1 Waiting points

Some approaches to solve D-VRP use waiting strategies. This consists in sending vehicles to certain strategic locations, called the waiting points, and wait for requests near these locations to appears. Some examples of waiting strategies can be founded in [9], [7] and [20].

It is then necessary to define which are the places where a vehicle can be positioned to wait for future requests. To do that, we redefine the set of nodes as a union of three subsets : $V = \{1, 2, ..., |V|\} = O \cup N \cup W$. O is the set of depot nodes, N is the set of customer nodes to which one or several requests can be associated. W is the set of waiting point. Note that the intersection of the sets N and W is not required to be empty. A node can be both a waiting point and a customer node, or only one of them.

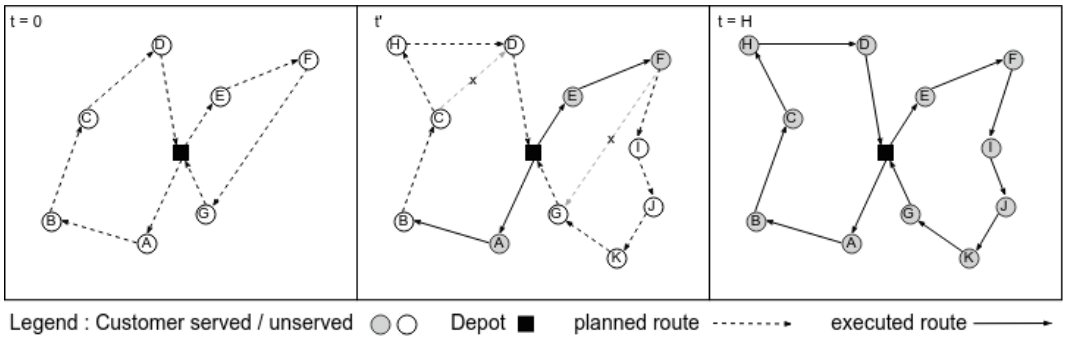


Figure 2.4: Example of a *Dynamic Vehicle Routing Problem*. When the routing plan is modified to serve the new requests, the route sections (■ A), (A B), (■ E) and (E F) are already executed and thus are not modidied to respect the *nonanticipativity constraints*.

New notation for the routes

For these variants, we redefine the notion of route of section 2.2. A route λ is a sequence of actions noted $\lambda =< a_1, a_2, ..., a_{|\lambda|} >$. An action a_i is a tuple that can take two forms. It can be an action that serves a request $r \in R$, the tuple is thus $a_i = (r_i, t_i^-, t_i^+)$ with r_i the request served. The action can also be a waiting action. In that case, the tuple take the form $a_i = (v_i, t_i^-, t_i^+)$ with $v_i \in W$ or $v_i \in O$, and it is not associated with any request.

2.5 The Dynamic and Stochastic Vehicle Routing Problem with Time Windows

In this section, we present a final variant of the VRP, the *Dynamic and Stochastic Vehicle Routing Problem with Time Windows* or DS-VRPTW. This variant will be the reference for the following chapters. We present our own formulation of the DS-VRPTW although it is partly inspired by the formulations in [21] and [20].

This type of problem is similar to the DVRP. The difference lies in the presence of uncertainty in the problem data, we detail this in the next subsection. The notations for the DS-VRPTW are therefore quite similar to those of the D-VRPTW.

Potential requests and Scenario

[13] defines a stochastic vehicle routing problem as a VRP where some element of the problem is random. The stochasticity source can be the quantity of a request, the time window boundaries or wheter or not the request is to be served. The random data must be taken into account to build a solution.

In this paper, the stochasticity source that we use concerns wether or not a set of potential requests will be issued. We therefore consider a set of potential request $\hat{R} = \{r_1, r_2, ..., r_{|\hat{R}|}\}$ that contains $\left|\hat{R}\right|$ requests. A potential request $r_i \in \hat{R}$ is defined as $r_i = (q_i, v_i, d_i, e_i, l_i)$. These attributes are defined in the previous sections.

We also define a function that gives, for each request $r \in \hat{R}$, the probability for the request

to be part of the scenario : $p(r) = P(r \in R), \forall r \in \hat{R}$.

The scenario of an execution is a subset of the set of potential requests $\{r \in R\} \subseteq \hat{R}$. Each potential request $r_i \in \hat{R}$ corresponds to one request in R or no request at all.

Each element τ of the scenario R is a tuple $\tau = (r, t)$, where r is a request from the set of potential requests $r \in \hat{R}$, and t is the reveal time associated to this request such taht $0 \leq t \leq H$.

The constraints

The constraints defined in the previous sections must be respected. As a reminder, we cite these constraints without giving the formulation.

- The arrival time at a node always precedes its departure time. (2.4)
- A vehicle must start serving a request during its time window and respect the service duration. (2.6)
- A route begins and ends with a depot. (2.7)
- A route must always respect the travel time between two nodes (2.8)
- Total customer demand on a route cannot exceed the capacity of the vehicle. (2.9)
- A request is served by at most one vehicle (2.10)
- A routing plan must always serve all the requests served in a previous routing plan (2.11)
- A routing plan cannot change the destination of a moving vehicle at the time it is issued. (2.12)
- A routing plan cannot modify events already executed from the previous routing plan. (2.13)

A practical example

To illustrate the DVRPTW, we will again consider the delivery company from the last practical example. But this time, the company undertakes to collect and deliver the packages within 24 hours. If this is not the case, the bill will be smaller for the clients. The goal for the company is therefore to maximize the number of packages delivered on time. The company offers a simple service and an express service. For the simple service, the customer calls and indicates the time slot of the next day where the company can come to collect the parcel. The express service is similar except that the company comes to collect the parcel the same day. Finally, the company can maintain statistics on its customers in order to know which ones send many packages and are likely to call them during the day.

This problem can be broken down into two sub-problems. The collection problem, and the delivery problem. The collection problem is a DS-VRPTW. The offline request correspond to the client of the simple service, the online request to the client of the express service. The vehicles collecting the packages bring them back to the warehouse for delivery the next day. The delivery problem is a VRPTW. It is not dynamic as all packages to deliver were collected the day before. But there is a time window associated with each package so that it is delivered in less than 24 hours.

Chapter 3

Descriptions of the tool's functionnalities

The goal of this master thesis is to propose a tool that can simulate a *Dynamic and Stochastic Capacitated Vehicle Routing Problem with Time Windows* to a solver, retrieve the solutions of the solver and visualize them. Solving the problem is not the subject of this thesis. But the objective is also to provide an API for someone developing a solver or testing its capacities, to separate the resolution of a dynamic `VRP` from its simulation.

In this chapter, we present the features of the tool that has been developed. First, section 3.1 presents an overview of the tool as well as the features offered by the simulation tool and the functionnalities proposed to facilitate the development of a solver. Second, in section 3.2, we present the contents of the input files of the tool, and thus what are precisely the problems that can be simulated.

3.1 General Overview and main usage

In this section, we will present in a general way the tool as well as the functionalities it offers. Section 3.1.1 first provides an overview of the tool and explains some general concepts. The sections 3.1.2, 3.1.3 and 3.1.4 explains the functionnalities offered by the simulator for the three different modes in which it can be used. Section 3.1.5 presents a way to visualize the solutions of the problem on an interactive map. Finally, section 3.1.6 shows the functionnality proposed for a solver using this tool.

3.1.1 Presentation of the tool

The simulator is a tool to simulate the execution of a `VRP` to a solver. The problem must be described using the file format described in section 3.2. The simulator performs only one simulation at a time. For each simulation, it communicates with only one solver. The simulator and solver are two distincts processes. To use a solver with the simulator, the solver must use the API provided for this purpose. Figure 3.1 provides an overview of the simulator and solver. First, we see that a user interacts with the simulator and communicates input files to it. We then see that the simulator interacts with an API used by a solver to simulate the execution of a problem.

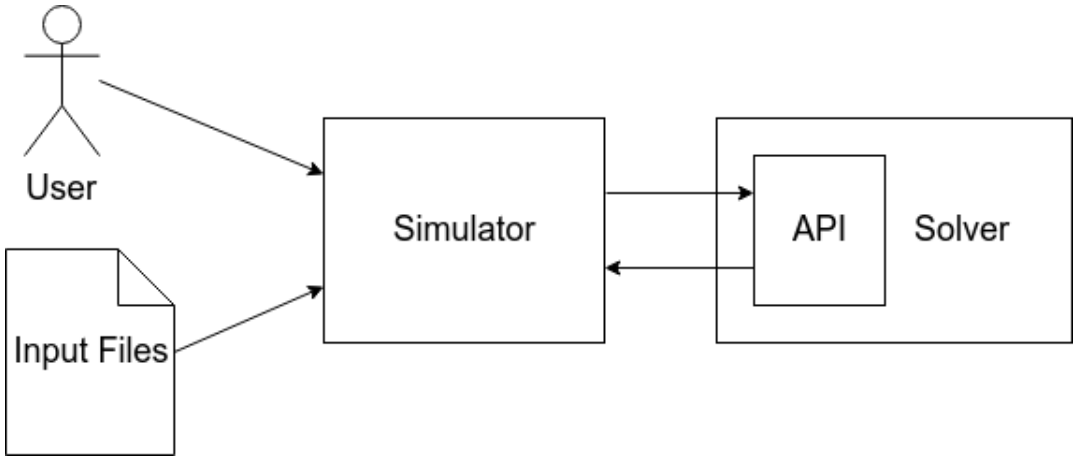


Figure 3.1: global solver and simulator diagram

Let us also clarify what is a simulation. A simulation can be defined by a set of interactions between two components: the simulator and the solver. The role of the simulator is to present (or simulate) a problem to a solver who must solve it and return the solutions obtained. A simulation is therefore the set of interactions, distributed over time, between the solver and the simulator, allowing the simulator to present a problem to the solver and to receive all the solutions computed by the solver. Figure 3.2 shows an example of a simulation. The simulator begins by communicating the input data to the solver. Then comes the offline simulation where the simulator starts by sending all the offline requests, and then retrieves the solution(s) returned by the solver during the time allowed for the offline simulation. Finally comes the online simulation. At each time unit of the online simulation, the simulator sends the requests to be revealed. The simulator also retrieves the solutions that the simulator can send at any time during the time allowed for the online simulation. The simulation ends after the end of the online simulation.

The simulator is designed to simulate a `VRP` with a static part followed by a dynamic part. A simulation thus includes several steps among which are two sub-simulations: the offline simulation (the simulation of the static part including all the requests known a priori) and the online simulation (the simulation of the dynamic part with the requests that appear in an online fashion). Figure 3.3 shows the different steps in a simulation as a state transition diagram. Let's briefly explain what these steps or states correspond to.

PreSimulation is the initial state. This is when the user loads the input files and configures the parameters of the problem.
OfflineComputation is the state corresponding to the offline simulation, when the solver is computing a solution. All offline requests are communicated to the solver at the start of the offline simulation.
OfflinePause is the state when the user commands to pause the offline simulation. Both the solver and the simulator are temporarily and synchronously suspended.
OfflineEnd is the state following the offline simulation, before the online simulation is started.
OnlineComputation is the state corresponding to the online simulation, when the solver tries to find solutions to the dynamic part of the problem.
OnlinePause is the state when the user pauses the online simulation. Both the solver and the simulator are temporarily and synchronously suspended.
PostSimulation is the state reached once the simulation is finished.

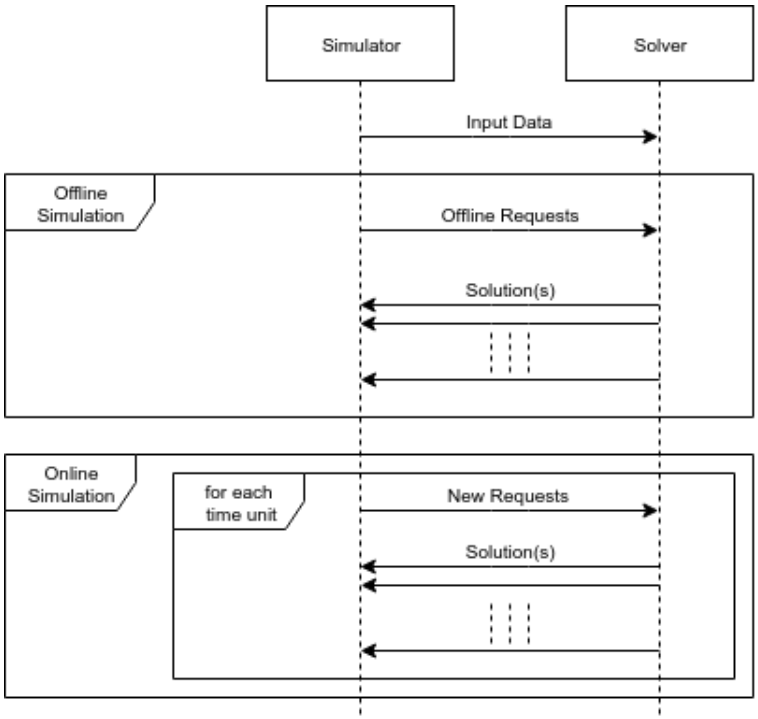


Figure 3.2: Example of a sequence of interactions between the simulator and a solver composing a simulation

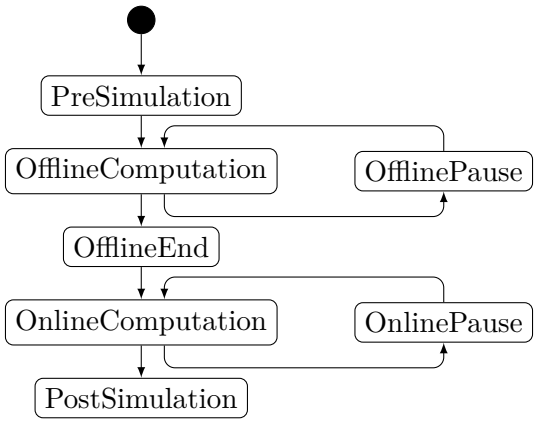


Figure 3.3: The steps of a complete simulation

3.1.2 The command line simulator

In this section we will see what it is possible to do with the command line simulator.

At startup, the simulator is always in the *PreSimulation* state. In this state, it is possible to load the input files *Carrier*, *Customer*, *Graph* and *Scenario*. Once the customer file is loaded, it is possible to generate one or more scenarios and save them for reuse.

Several parameters can be consulted and modified before starting the offline simulation. This includes the offline time and the computation time. The vehicles in the fleet and the capacity of the vehicles types. It is also possible to see the different types of vehicles, to delete the vehicles in the fleet or to add new vehicles.

It may be useful to provide additional data to the solver, which cannot be described with any

of the four input files, but which the simulator does not need to use. It is therefore possible to load additional files, provided that their contents respect the JSON format, in order to communicate them to the solver when starting the offline simulation.

Once the input files have been loaded and the parameters set, the simulator must be instructed to send all this data to the solver. Only then can the simulator be ordered to start the offline simulation. The simulator will then allow the solver to start its calculations to find solutions to the static problem. For its calculations, the simulator has the time indicated by the offline time parameter. The user can command to pause the simulation. The solver then interrupts its calculations and will continue only when the user commands the simulator to resume simulation.

At the end of the offline simulation, the simulator is in an intermediate state between offline and online simulation. The simulator starts running the online simulation at the user's command.

At any time after the start of the offline simulation, the user can save either all the solutions returned by the solver, or the last valid solution only.

After the end of the offline simulation and before starting the online simulation, it is also possible to load a solution that is in a file in the format indicated in the section 3.2.5. The simulator will communicate this solution to the solver. This makes it possible, for example, to switch off the simulator between an offline and online simulation. Simply save the last solution at the end of the offline simulation, restart the simulator, and run a new simulation without offline time, and submit the saved solution at the start of the online simulation.

It is also possible to stop the simulation in progress, the simulator is then back to the initial state in order to prepare another simulation.

3.1.3 Using scripts

The command line interface is convenient because of the control it offers the user throughout the simulation. However, it can also be practical to be able to automate the simulation process. It is indeed easy to forget to launch an online simulation after launching an offline simulation that last several minutes. For this reason, it is possible to define a script to execute a sequence of commands and thus avoid having to type each command in the command-line interface.

A script is a text file containing one command at each line. When the simulator is launched with a script, it executes each command one after the other, and shut down after the last command was handled. The simulator writes to the terminal in the same way as when it is used manually. In this way, it is possible to follow the script's progress.

In order to know what it is possible to do with a script, it is important to understand how the simulator executes the script commands. When the simulator is used in command line and an offline or online simulation is running, it is possible to enter new commands before the end of the simulation. When used with a script file, the simulator does not work like this. When the script contains a command to run an offline or online simulation, the simulator will wait for the end of this simulation before executing the next command. Without this, it would be impossible to launch an online simulation, because we would try to launch the online simulation before the end of the offline simulation. It would also be impossible to record the results obtained at the end of the simulation.

It should also be noted that it is not currently possible to run several different simulations with a single script file.

3.1.4 The graphical interface of the simulator

The graphical user interface allows to use the simulator in a more user-friendly way. But there are still some differences between the features offered by the GUI and those of the command line interface. An overview of what the GUI looks like is provided in the figure 3.4.

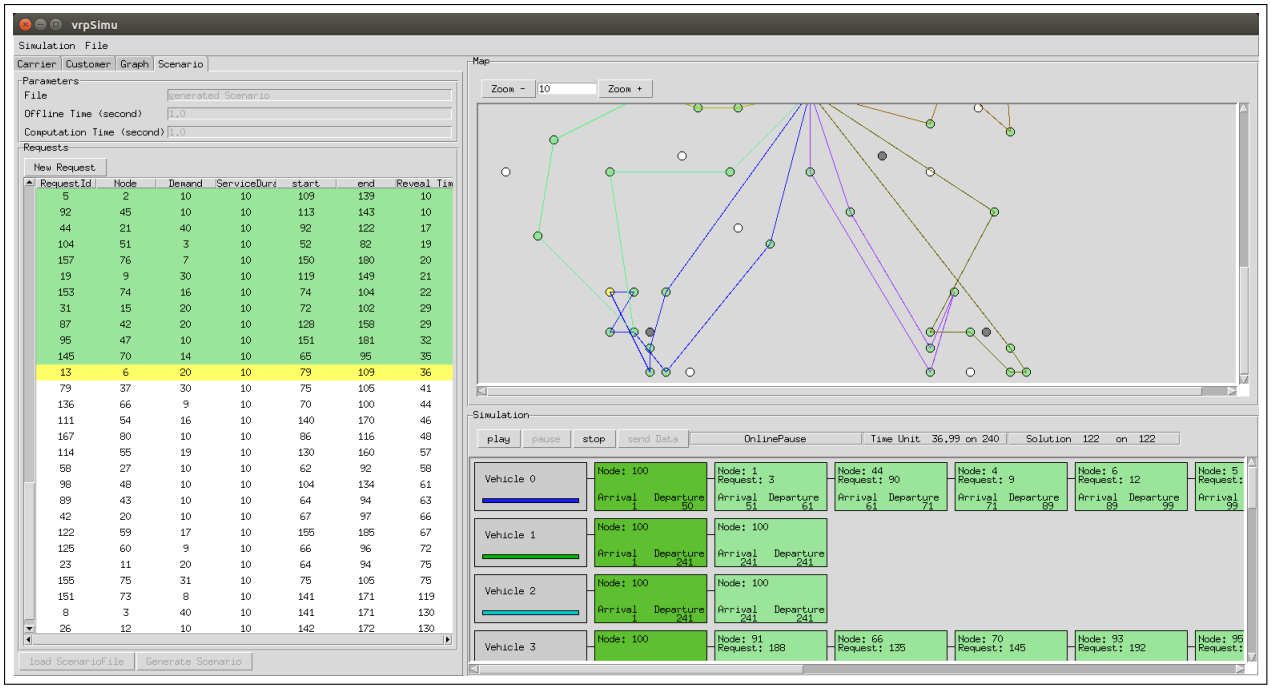


Figure 3.4: Overview of the interface

At startup, or after interrupting a simulation, the simulator will be in the presimulation state. It is then possible to load input files and generate a scenario as with the command line interface. Moreover the graphical interface allows to visualize the contents of these files. The contents of the files are displayed intuitively, easier to understand than a plain text file. The interface allows to display the requests in the customer file or the scenario file by sorting them according to certain criteria, like the node associated to the request, the time window or the reveal time of the request.

The graphical interface also has a map on which the graph can be displayed. To do this, each node described in the graph file (see section 3.2.1) must have a `MapCoord` attribute describing the coordinates to use to display the node on the map. If this is not the case, fictitious coordinates are generated from one of the travel times matrices contained in the carrier file.

Offline and online simulations can be started from the graphical interface. During simulation, the graphical user interface displays the last valid solution returned by the simulator. On the map, the various roads are marked out in such a way as to distinguish the paths already travelled from those yet to be travelled. The informations of each route are also displayed with more details in another area of the interface. These are the arrival time and departure time of the vehicles for the nodes in their itinerary, but also which request are served.

The interface also allows during simulation to easily distinguish the status of each request from the scenario. That is, whether the request has been revealed or not, whether it has already been or will be served, or whether it has not been served.

After the end of the offline simulation and before starting the online simulation, it is possible

to load a solution into a file for the simulator to communicate it to the solver. This makes possible to perform an offline simulation, save the solution and use it later to perform the following online simulation, after having restarted the simulator for example.

As with the command line interface, it is possible to pause and resume simulation. It is also possible to interrupt a simulation to return to the simulator's initial state.

3.1.5 Displaying soutions on a real map

Beside the simulator, it is possible to display the last solution returned by the solver in an interactive map that use the gps coordinates of the graph nodes to represent them. This map is updated with each new solution returned by the solver. This feature can be used regardless of the simulator mode used (command line, with script or with the GUI). Moreover, this map can be opened while the simulator is off. An overview of what the map looks like is provided in figure 3.5.

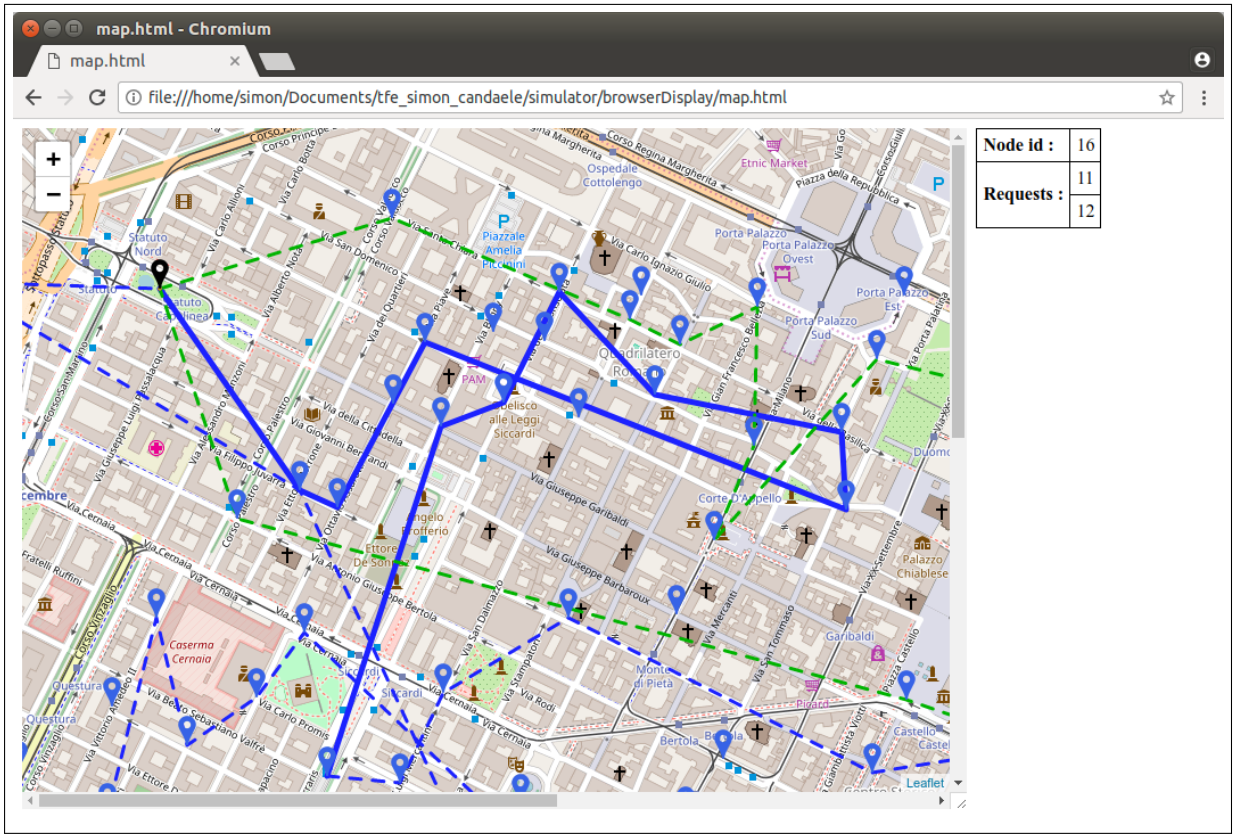


Figure 3.5: Preview of the display in the browser

On this map is first of all represented the graph, the roads of each vehicle are also displayed. It is thus possible to see directly the evolution of the solution on a real map, and to consult on it which requests are served to which nodes.

3.1.6 Functionnalities offered to the solver

Solving a VRP is a difficult task and a lot of research is done in this area. Particularly for dynamic problems, the timing at which each data describing the problem is communicated to

the solver is important. One of the goals of this thesis was to provide a tool that could manage this task in order to facilitate the development of a solver. We therefore developed a `C++` API that makes the link between the simulator and a solver using this API.

The API facilitates the access to input files. A single function call returns the input files sent by the simulator. The contents of these files are then available in an easy to use form for the solver. This ensures that there are no errors when loading files.

The solver receives the requests thanks to the API. The solver is thus assured never to access a request before the reveal time associated with it.

The API also facilitates time management in the solver. Functions allow the solver to know what the current unit of time is and whether the simulation is finished or not.

Using the API also ensures that the solver will not be able to cheat on the time constraints imposed on him. Each time the solver communicates a new solution to the simulator, the API checks the exact moment when that solution is submitted. The simulator can then check that the different time constraints are respected. If this is not the case, the solution will not be registered by the simulator as a valid solution. To be valid, a solution must be submitted within the time allowed for the simulation.

In addition to the constraints concerning when the solver can submit a solution, the simulator checks that the solution never violates the travel times between the nodes and that the capacity of the vehicles are never exceeded.

When the solver submits a solution that the simulator deems invalid, the simulator will communicate the last valid solution received to the solver. The solver can then continue its calculations based on the last valid solution.

3.2 Input files, Output file and simulated problems

Before simulating a problem, it is obviously necessary to describe it unambiguously to the simulator. Many file formats exist to describe a VRP. Each is designed to describe some desired constraints. Constraints that can be modelled in one format can not always be modeled in another format. Many formats are therefore already used to specify VRP. However, we could not find a format to specify all the constraints that we wanted to be able to model. And in the absence of real consensus on a file format to describe a `VRP`, we made the decision to design our own file format which generalizes the format introduced in [17]. This allows us to describe in a single format the constraints coming from different other formats that we considered interesting.

We split the specification of a VRP instance into four distinct parts. Each one being described in a corresponding file. The following sections will present the formats we have defined and will also each present an example file. These example files are consistent with each other in order to present an example of a problem specified in this format.

Graph contains information concerning the different geographical points of the problem. This information is the locations of the different points as well as their types (depot, potential customer or waiting point).

Carrier describes the vehicles types available to serve the requests and how each vehicle type behaves in the graph. A vehicle type is defined by its capacity and its travel-times between

the points on the graph. The travel-times may vary with time and these variations are described here.

Customer describes the potential requests that may be emitted and their probabilities to be emitted. These emission probabilities can be used by the solver to anticipate the emission of future requests.

Scenario contains the requests that will actually be emitted during the simulation. The requests are communicated to the solver, each at the reveal-time associated with it.

Note that the **Scenario** is the only one that the solver does not have access to from the beginning. It contains the dynamic information that must be progressively revealed during the simulation. However, he will have access to the other three from the start of the simulation as they describe essential information for the solver.

We have also defined a format to describe a solution. The solver must respect this format when communicating a solution to the simulator.

These files are in JSON format, it allows to have a structure understandable for a human and the simulator can also load the files very easily. The use of the JSON format also makes it easy in the future to extend the file format as required. The JSON format is described in [10]. Let us recall some principles of this format. There are two structures used in JSON, the object and the array. An object is a collection of name/value pairs, the name being a string. An array is an ordered list of values. A value can be an object, an array, a string, a number, a boolean or null.

There is, to our knowledge, no formal grammar to formally describe the structure of any JSON file. In order to formally describe the structure, or format, of our files, we use a kind of mini formal grammar with just the necessary rules to define our files. Here is how it works.

The symbols `* ? | ( )` are part of the grammar. They are not part of the file. Strings written exclusively with capital letters and underscores are a symbol of the grammar that can be replace using the convenient rule. A rule has a forme like this : `A := expression`. It means that the symbol `A` can be replaced by a sequence of character respecting what is defined in the `expression`. `B*` matches 0 or more occurence of `B`. `C?` means that `C` is optionnal. It can occurs 0 or one time. `D|E` matches an occurence of either `D` or `E`. Parentheses, `(` and `)` are used to group several symbols together. `NUMBER` matches a number and `STRING` matches a string.

Please note that some constraints are not described in the figures corresponding to each file, with the grammar we defined, but in the text to simplify the grammar syntax.

3.2.1 The Graph file

The graph file contain several informations for all the nodes of the graph used in the instance of the problem we want to describe. The file must always consist of a JSON object, the graph object, whose structure is defined in Figure 3.6. A simple example, that we will detail at the end of this is section, is provided in the figure 3.7.

The first pair of the graph object has the name *FileType*. It is used to tell the simulator what type this file is. Here, the value is a string which is : *Graph*. It is mandatory to mention it, otherwise the simulator would have trouble knowing what type of file it is.

The second pair has the name *Nodes*. Its value is another JSON object describing the nodes

```
GRAPH_OBJECT := {
    "FileType": "Graph",
    "Nodes": { NODE_ID : NODE_VALUE (,NODE_ID : NODE_VALUE)* }
}

NODE_ID  := STRING

NODE_VALUE :=  {
        ("GpsCoord" : {"lat": NUMBER, "lng": NUMBER},)?
        ("MapCoord" : {"X":NUMBER, "Y":NUMBER},)?
        "NodeType" : ["Depot"] |
                     ["Customer" | "WaitingPoint"] |
                     ["Customer", "WaitingPoint"]
}
```

Figure 3.6: Structure of a graph file

of the graph. For each node, there is a pair whose name is the id of the node (different for each node), and whose value is an object describing the node. The `NODE_ID` must be a string representing an integer greater or equal to 0.

Each node is described with an object that can have up to three pairs. The names of these pairs are *GpsCoords*, *MapCoord* and *NodeType*. The value of *GpsCoords* is an object with the gps coordinates of the node. The value of *MapCoord* is an object containing X, Y coordinates to display the node on a map embedded in the graphical interface of the simulator. The pair *GpsCoords* and *MapCoord* are not necessary for the proper operation of the simulator. But they must either be specified for each node without exception or for no nodes at all. The value of *NodeType* is an array that specifies the type(s) of the node. There are currently three types taken into account by the simulator.

The type *Depot* defines a node from which vehicles start and end their routes. This type can not be combined with another type. The type *Customer* defines a node from where requests can be emitted. The type *WaitingPoint* defines a node that a vehicle can visit without it being to serve a request, but simply to wait at a strategic point. If it is not combined with *Customer*, no request should appear on this node.

The figure 3.7 shows an example of a graph file. In this example, the graph described contains slightly more than a hundred nodes, but we do not show them all. For each node its geographical coordinates are indicated as well as X,Y coordinates for the map inside the graphical interface of the simulator. Each node also has a list indicating the node types. We see that node `"1"` is the depot. The node `"4"` is a waiting point only, so no request can be emitted from this node. The node `"3"` is both a customer and a waiting point node, so the vehicles can come to this point to wait on a strategic position, and requests can be emitted from this node.

3.2.2 The Carrier file

The carrier file contains information about the vehicles available to build a solution. The file must always contain a JSON object whose structure is given in Figure 3.8. You can also find an example file in figure 3.9 that we will explain at the end of this section.

The first pair we see, with name *FileType*, is again used to tell the solver what type of file it is. This field is mandatory for the simulator.

The second pair, with name *VehicleTypes*, gives us information about the types of vehicles

```
{
    "FileType": "Graph",
    "Nodes": {
        "1": {
            "GpsCoord": { "lat": 45.07629, "lng": 7.67068},
            "MapCoord": { "X": 0.8840000000000001, "Y": 98.1712},
            "NodeType": [ "Depot"]
        },
        "2": {
            "GpsCoord": { "lat": 45.07967, "lng": 7.666022},
            "MapCoord": { "X": 0.4182, "Y": 97.8332},
            "NodeType": [ "Customer"]
        },
        "3": {
            "GpsCoord": { "lat": 45.07468, "lng": 7.689804},
            "MapCoord": { "X": 2.7963999999999998, "Y": 98.3322},
            "NodeType": [ "Customer", "WaitingPoint"]
        },
        "4": {
            "GpsCoord": { "lat": 45.05943, "lng": 7.688095},
            "MapCoord": { "X": 2.6255, "Y": 99.8572},
            "NodeType": [ "WaitingPoint"]
        },
        ...
        ...
        ...
        "105": {
            "GpsCoord": { "lat": 45.075718, "lng": 7.676971},
            "MapCoord": { "X": 1.5131000000000001, "Y": 98.22840000000001},
            "NodeType": [ "Customer"]
        }
    }
}
```

Figure 3.7: Example of Graph File Content

```
CARRIER_OBJECT := {
    "FileType" : "Carrier",
    "VehicleTypes" : [{
        "Capacity" : NUMBER,
        "VehicleType" : STRING
        }],
    ("Vehicles" : {
        VEHICLE_ID : VEHICLE_TYPE (, VEHICLE_ID : VEHICLE_TYPE)*
    },)?
    "TimeSlotsNumber" : NUMBER,
    "TravelTimes" : [
        { "Vehicle" : STRING,
          "VehTravelTimes" : TRAVEL_TIMES_MATRIX
          "TimeSlot" : TIMESLOTS
        }
        (,{ "Vehicle" : STRING,
          "VehTravelTimes" : TRAVEL_TIMES_MATRIX
          "TimeSlot" : TIMESLOTS
        })*
    ],
    "Unit" : "Hour" | "Minute" | "Second"
}

VEHICLE_ID          := STRING
VEHICLE_TYPE        := STRING
TRAVEL_TIMES_MATRIX := [[NUMBER(, NUMBER)*] (, [])* ]
TIMESLOTS           := [NUMBER(, NUMBER)*]
```

Figure 3.8: Structure of a carrier file

that can form the fleet of vehicles. The value of this pair is an array containing an object for each type of vehicle. A vehicle type is described using two pairs. The pair with name *VehicleType* is a

string that is different for each type. The pair with name *Capacity* which is the maximum load of the vehicle.

The third pair, with name *Vehicles*, describes the fleet of vehicles available for the solver. The value of this pair is an object with a pair for each vehicle in the fleet. For each pair in this object, the name is a `VEHICLE_ID`, which is a string that must describe an integer and differs from all the other `VEHICLE_ID`. The value of each of these pairs is a `VEHICLE_TYPE`. This is a string that must be identical to one of those used in the array of the pair *VehicleTypes*. It should be noted that the simulator does not impose any constraints on the composition of the vehicle fleet. A given type of vehicle may therefore not be present in the fleet although described in the array of the pair *VehicleTypes*.

The fourth field, *TimeSlotsNumber*, indicates the number of timeslot existing in the instance that this file describes. There must always be exactly one timeslot for the offline simulation, and at least one timeslot for the online Simulation.

The next pair, with name *TravelTimes*, contains an array of objects, each describing the travel times between each point for a given vehicle type. More precisely, each of these objects contains three pairs named *Vehicle*, *VehTravelTimes* and *TimeSlot*. The value of *Vehicle* indicates to which type of vehicle these travel times correspond to. The value of *VehTravelTimes* must always contain a double array representing a square matrix, of dimension $n \times n$ and not necessarily symmetric, representing the travel times between nodes 0 to $n-1$. The Yth element of the Xth array therefore represents the travel time to reach the node with id Y from the node with id X. To model the variation of travel times as a function of time, the value of *TimeSlot* contains an array of the timeslots during which the vehicle must start its trip between two points for the associated matrix to be valid.

It is important to note that although all nodes mentioned in the Graph file must be included in the travel times, not all nodes mentioned in the travel times must be present in the graph file. If the travel time matrices are of dimensions n, we cannot mention, in the graph file, nodes with an id greater or equal to n, or strictly smaller than 0. Thus, from a very large graph, it is possible to get several subgraphs to perform different simulations.

The last pair, named *Unit* mention the unit used to specify the travel times. Travel times are not indicated in time units, and they are thus independent of the number of time units used.

In the example file in figure 3.9, we see that there are two possible vehicle types : bike and car. But the solver can use three vehicles in one solution, two bike and one car. Travel times, which vary according to time, are also indicated for each type of vehicle. The travel times for the bikes do not change during the simulation. However, travel times for cars are not the same for timeslots 1 and 2 as for timeslots 3 and 4.

3.2.3 The Customer file

The customer file contains information about the requests that can be emitted. The file must always contain a JSON object whose structure is given in Figure 3.10. An example file can be found in figure 3.11 and will be detailed at the end of this section.

As for the other files, the pair with name *FileType* is mandatory.

The value of the pair named *HorizonSize* specify the number of time Units present in the online time. These time units are distributed in different timeslots. The number of timeslots is

```
{    "FileType": "Carrier",
     "VehicleTypes": [
        { "Capacity": 200, "VehicleType": "car"},
        { "Capacity": 20,  "VehicleType": "bike"}],
    "Vehicles": {
        "0": { "VehicleType": "bike" },
        "1": { "VehicleType": "bike" },
        "2": { "VehicleType": "car" }
    },
    "TimeSlotsNumber": 5,
    "TravelTimes": [
        {
            "Vehicle": "car",
            "VehTravelTimes" : [[0, 1502, 1575, ... , 1412], [1499, 0, 151, ... , 162],
                ... , [1325, 170, 204, ... , 0]],
            "TimeSlot" : [1,2]
        },{
            "Vehicle": "car",
            "VehTravelTimes" : [[0, 1625, 1765, ... , 1502], [1550, 0, 163, ... , 164],
                ... , [1370, 124, 240, ... , 0]],
            "TimeSlot" : [3,4]
        },{
            "Vehicle" : "bike",
            "VehTravelTimes" : [[0, 1600, 1721, ... , 1502], [1683, 0, 130, ... , 127],
                ... , [1456, 124, 167, ... , 0]],
            "TimeSlot" : [1,2,3,4]
        }
    ],
    "Unit" : "Second"
}
```

Figure 3.9: Example of Carrier File Content

```
CUSTOMER_OBJECT := {
    "FileType"         : "Customers",
    "HorizonSize"      : NUMBER,
    "TimeSlotsNumber"  : NUMBER,
    "TimeSlots"        : [NUMBER, NUMBER (, NUMBER)*],
    "RealDurationPerTimeUnit" : NUMBER,
    "RealTimeUnit"     : "Second",
    "PotentialRequests" : [REQUEST (, REQUEST)*]
}
REQUEST := {
    "ArrivalProbability": [NUMBER, NUMBER (, NUMBER)*],
    "Demand": NUMBER,
    "Node": NUMBER,
    "RequestId": NUMBER,
    "ServiceDuration": NUMBER,
    "TimeWindow": TIME_WINDOW
}
TIME_WINDOW := {
    "start": NUMBER,
    "end": NUMBER,
    "TWType": "absolute"
    } | {
    "TWType": "relative",
    "TimeWindowDuration"
}
```

Figure 3.10: Structure of the customer file

indicated by the value of the pair named *TimeSlotsNumber*. The first time unit of each timeslot is indicated in the array of the pair named *TimeSlots*. The first timeslot is always the offline timeslot. This timeslot always has only one time unit, and it is a bit special because, as we will see in the scenario file, the time accorded to the solver for this one can be different from that of the other time units.

```
{   "FileType": "Customers",
    "HorizonSize": 480,
    "TimeSlotsNumber": 5,
    "TimeSlots": [-1, 0, 120, 240, 360],
    "RealDurationPerTimeUnit": 60,
    "RealTimeUnit": "Second",
        "PotentialRequests": [
        {
            "ArrivalProbability": [100, 0, 0, 0, 0],
            "Demand": 1,
            "Node": 6,
            "RequestId": 1,
            "ServiceDuration": 4,
            "TimeWindow": { "TWType": "absolute", "start": 0, "end": 480 }
        },{
            "ArrivalProbability": [ 0, 30, 0, 0, 0],
            "Demand": 3,
            "Node": 8,
            "RequestId": 2,
            "ServiceDuration": 4,
            "TimeWindow": { "TWType": "absolute", "start": 100, "end": 130 }
        },{
            "ArrivalProbability": [10,20,50,5,0],
            "Demand": 3,
            "Node": 9,
            "RequestId": 3,
            "ServiceDuration": 4,
            "TimeWindow": {"TWType": "relative", "TimeWindowDuration": 50}
        },{
            "ArrivalProbability": [ 0, 50, 0, 0, 0],
            "Demand": 3,
            "Node": 9,
            "RequestId": 4,
            "ServiceDuration": 4,
            "TimeWindow": {"TWType": "absolute", "start": 300, "end": 350}
        },{
            "ArrivalProbability": [ 0, 0, 50, 0, 0],
            "Demand": 3,
            "Node": 9,
            "RequestId": 5,
            "ServiceDuration": 4,
            "TimeWindow": {"TWType": "absolute", "start": 300, "end": 350}
        },
        ...
        ]
}
```

Figure 3.11: Example of Customer File Content

The value of the pair named *RealDurationPerTimeUnit* is the real duration of a time unit. This is not the duration that will be used by the simulator for each time unit during the simulation, but rather the duration in the real world. This duration is measured with the unit indicated by the value of the pair named *RealTimeUnit*. This duration can be different from the fictitious duration, the computation time, that the simulator will actually use during the simulation.

For example, if there are 240 time units for the online simulation, and the real duration of a time unit is 60 seconds. The total duration, in real life, will therefore be 240 minutes. But it is possible for the simulator to give the solver a different computation time (e.g. 5 seconds) for each time unit. In this case the simulation will last only 20 minutes.

The value of the pair named *PotentialRequests* is an array of potential requests that can be emitted. Each value in this array is an object that describes a potential request. We will now explain the different attributes of a request.

First of all, each request can be identified thanks to its unique Id indicated by the value of the

pair named *RequestId*. A request must always be associated with a node whose id is specified in the value of the pair named *Node*. There can be several requests associated with the same node. The value of the pair named *Demand* indicates the demand of the request, and is an integer greater than or equal to 0. To serve a request, a vehicle needs to stay at the corresponding node for a certain amount of time indicated by the value of the pair named *ServiceDuration*.

The value of the pair named *TimeWindow* is an object describing the time window during which a vehicle must arrive at the nod to start serving the request. There are two types of time window: *absolute* and *relative*. The type of a time window is specified by the value of the pair named *TWType* in the object describing this time window. If the *TWType* of a time window is *absolute*, then it must have the pairs named *start* and *end*, whose values indicate respectively the first and last time unit of the time window. If the *TWType* of a time window is *relative*, then it must have the pair named *TimeWindowDuration* whose value indicates the length of the time window in number of time units. The first time unit of the time window will be the one when the request will be revealed, if the request is emitted. The last time unit is equal to the first plus the value of *TimeWindowDuration*.

The last field, *ArrivalProbability*, requires a little more concentration to understand it. It represents the distribution probability of the request emission on the different timeslots. It contains a array of length equal to the total number of timeslot (offline and online). Each element of this array indicates the probability that the request is emitted during the corresponding time slot. Since this array represents a probability of distribution, the sum of all items in the array must always be smaller or equal to 100. The probability that the request will not be emitted at all is equal to $100 - \sum_{i=0}^{i=n-1}(p_i)$ where n is the length of the array, and p_i is the i-th element of the array. When a request is emitted during a given timeslot, it means that this request will be revealed to the solver at a time unit belonging to that timeslot. All time units belonging to a given timeslot have the same probability of being the time unit when the request will be revealed, if it is revealed during that timeslot. If the request is revealed during the offline timeslot, it will be revealed to the solver at the very beginning of the offline time.

We see that it is therefore possible to describe requests emitted in stochastic ways. The number of different ways to fill in an arrival probability table makes it possible to describe a multitude of different statistical behaviours on the part of users. It is also possible to describe requests emitted in a deterministic way. For that, the arrival probability of one timeslot should be set to 100. All these information can be used by the solver since the customer file is communicated to it.

Let's explain how to interpret the arrival probabilities in the example file in 3.11. We see that there are two potential requests, with the respective *RequestId* 0 and 1. Let's call request 0 and request 1. You can see that request 0 has 50% chance to be revealed in the timeslot 0 at the beginning of the offline simulation thus, 10% chance to be revealed during the timeslot 1 and 5% chance to be revealed during the timeslot 2. The timeslot 1 and 2 both contains eighty time units and they span respectively from time unit 0 to 79 and from time unit 80 to 159. Time units ranging from 0 to 79 therefore all have a $10/80 = 0.125\%$ chance of being the time unit where the request will be revealed. Time units ranging from 0 to 79 all have a $5/80 = 0.0625\%$ chance of being the time unit where the request will be revealed. And there's a $100 - 50 - 10 - 5 = 35\%$ chance the request won't be emitted at all.

The request 1 has $100 - 40 - 40 = 20\%$ chance of not being emitted at all. The request cannot be emitted during either the timeslot 0 or the last timeslot. There is a 20% chance that the request will be emitted during timeslot 1 and a 60% chance that the request will be emitted during timeslot 2. Time units from 80 to 159 form timeslot 2, and for each of them, the

probability that it is the one during which the request is emitted is therefore $60/80 = 0.75\%$.

Now let's describe the example customer file shown in figure 3.11. First of all, we see that the online simulation includes 480 time unit distributed into 4 timeslot. With the timeslot corresponding to the offline simulation, there are five timeslots. We also see the five first requests that can be emitted. Request 1 will be emitted with 100 percent probability at the start of the offline simulation. Request 2 has only a 30 percent chance of being issued, and it will be during the first online timeslot if this is the case. Request 3 will be emitted with 85 percent probability, it is likely to appear during all timeslots except the last. Unlike other requests, the time window of this one is not known in advance, it will be determined according to the moment when the request will be revealed to the solver. This request illustrates how the emission probability can be spread over several timeslots. Requests 4 and 5 are identical except for the timeslot where they can be issued. This shows how one can model the same request that can be emitted several times at different moments.

3.2.4 The Scenario file

The last input file is the scenario file. The file must always contain a JSON object whose structure is given in Figure 3.12. An example is provided in Figure 3.13. This scenario is an example of what can be generated from the example customer file presented previously.

```
SCENARIO_OBJECT := {
    "FileType": "Scenario",
    "ComputationTime": NUMBER,
    "OfflineTime": NUMBER,
    "Requests" : [REQUEST (, REQUEST)*]
}
REQUEST := {
    "Demand": NUMBER,
    "Node": NUMBER,
    "RequestId": NUMBER,
    "RevealTime": NUMBER,
    "ServiceDuration": NUMBER,
    "TimeSlot": NUMBER,
    "TimeWindow": {
        "start": NUMBER,
        "end": NUMBER
    }
}
```

Figure 3.12: Structure of the scenario file

The value of the pair named *FileType* indicates the type of the file.

The values of the pairs named *OfflineTime* and *ComputationTime* indicate respectively how long the offline simulation will last, and how long each time unit will last during the online simulation. Both are in seconds.

The value of the pair named *Requests* is an array. Each value of this array is an object describing one of the requests that make up the simulation. The requests are described in a similar way in the customer file, so we won't explain each pair again. However there are two new pair for each request, named *RevealTime* and *TimeSlot*.

The value of *RevealTime* indicates the time unit when the request is revealed to the solver. And the value of *TimeSlot* indicates to wich timeslot this time unit belongs to. The offline timeslot is indicated by -1, and the first online timeslot is indicated by 1.

```
{   "FileType": "Scenario",
    "ComputationTime": 10.0,
    "OfflineTime": 1.0,
    "Requests": [
        {
            "RevealTime": -1,
            "TimeSlot" : -1,
            "Demand": 1,
            "Node": 6,
            "RequestId": 1,
            "ServiceDuration": 4,
            "TimeWindow": {"start": 0, "end": 480}
        },{
            "RevealTime" : 62,
            "TimeSlot" : 1,
            "Demand": 3,
            "Node": 8,
            "RequestId": 2,
            "ServiceDuration": 4,
            "TimeWindow": {"start": 100, "end": 130}
        },{
            "RevealTime" : 250,
            "TimeSlot" : 2,
            "Demand": 3,
            "Node": 9,
            "RequestId": 3,
            "ServiceDuration": 4,
            "TimeWindow": {"start": 250, "end": 300}
        },{
            "RevealTime": 100,
            "TimeSlot" : 1,
            "Demand": 3,
            "Node": 9,
            "RequestId": 4,
            "ServiceDuration": 4,
            "TimeWindow": {"start": 300, "end": 350}
        },
        ...
        ]
}
```

Figure 3.13: Example of Scenario File Content

There is an important constraint to respect. A scenario file and a customer file used together to specify a problem must match each other. This means that two requests, one in the scenario file, the other in the customer file, which have the same *RequestId*, must have corresponding values for all their other attributes.

However it is not mandatory for a request in the scenario file to have a request with the same *RequestId* in the customer file. This enables new requests to be emitted without matching a request in the customer file.

Figure 3.13 shows an example scenario file that can be generated from the example customer file presented in the previous section. Request 1 is obviously present because it was described with 100 percent probability to timeslot offline. Request 2, which had only a 30 percent chance of appearing in the scenario, will be revealed to the solver at time unit 62. Request 3 will be revealed to the solver in time unit 250. The time window of this request was relative in the customer file, but in the scenario the first and last time unit of this time window were fixed. Finally, we see that among requests 4 and 5 in the customer file, only request 4 appears in the scenario.

3.2.5 The Solution format

In addition to the input files, we will also describe an output file. This is a file created by the simulator, containing the solutions returned by the solver. Like the input files, this file contains a JSON object whose structure is presented in Figure 3.14. Inside this file, each solution is represented using a single JSON object. When sending a solution to the simulator, the solver must respect the structure of the solution defined here.

```
SOLUTION_FILE := {
    "NumberOfSolutions" : NUMBER,
    "Solutions" : [(SOLUTION_OBJECT (, SOLUTION_OBJECT)*)?],
    "SolutionsNotValid" : [(SOLUTION_OBJECT (, SOLUTION_OBJECT)*)?]
}

SOLUTION_OBJECT := {
    "TimeUnitOfSubmission": NUMBER,
    "Routes": { VEHICLE_ID : ROUTE (, VEHICLE_ID : ROUTE)*}
}

VEHICLE_ID := STRING

ROUTE := [(ROUTE_NODE (, ROUTE_NODE)*)?]

ROUTE_NODE := {
    "ArrivalTime" : NUMBER,
    "DepartureTime" : NUMBER,
    "NodeId" : NUMBER,
    ("RequestId" : NUMBER)?
}
```

Figure 3.14: Structure of a solution object

The main JSON object in the solution file must contain three pairs named *NumberOfSolutions*, *Solutions* and *SolutionsNotValid*. The value of the first is the number of valid solutions. The value of the second is an array containing an object describing each valid solution returned by the solver. The value of the third is an array containing an object describing each invalid solution returned by the solver.

An object describing a solution has two pairs named *TimeUnitOfSubmission* and *Routes*. The value of the first one is the exact moment when the solution was submitted, in time unit. This is not an integer value but a double. This allows to know to which time unit, and at which moment during this time unit, the solution was submitted. The value of the pair named *Routes* is an object with a pair for each vehicle of the fleet. The name of these pairs is a string representing the id of a vehicle. The value of each pair is an array. Each value in this array is an object with four pairs describing a step in the itinerary of the vehicle. The value of the pair named *NodeId* is the id of the node visited. The value of the pair named *RequestId* is the id of the request being served at this node. This pair is not mandatory as a vehicle is not constrained to serve a request at each node. The value of the pair named *ArrivalTime* is the exact moment when the vehicle arrives at the node, expressed in time unit. The value of the pair named *DepartureTime* is the exact moment when the vehicle leaves the node, expressed in time unit.

The figure 3.15 shows an example of a solution described using the JSON format. We see that this solution was submitted approximately halfway through the time unit 260. The requests in the example scenario file were thus already all revealed to the solver and they are all served in the solution.

```
{
    "TimeUnitOfSubmission": 260.45,
    "Routes": {
        "1": [{
                "ArrivalTime": 0,
                "DepartureTime": 56.2,
                "NodeId": 1
            },{
                "ArrivalTime": 70.4,
                "DepartureTime": 95,
                "NodeId": 6,
                "RequestId": 1
            },{
                "ArrivalTime": 102,
                "DepartureTime": 120,
                "NodeId": 8,
                "RequestId": 2
            }, ... ,{
                "ArrivalTime": 450,
                "NodeId": 1
            }],
        "2": [{
                "ArrivalTime": 0,
                "DepartureTime": 10,
                "NodeId": 1
            }, ... , {
                "ArrivalTime": 241,
                "DepartureTime": 320,
                "NodeId": 9,
                "RequestId" : 3
            },{
                "ArrivalTime": 350,
                "NodeId": 1
            }],
        "3": [{
                "ArrivalTime": 0,
                "DepartureTime": 310,
                "NodeId": 1
            },{
                "ArrivalTime": 330,
                "DepartureTime": 360,
                "NodeId": 9,
                "RequestId": 4
            },{
                "ArrivalTime": 390,
                "NodeId": 1
            }]
    }
}
```

Figure 3.15: Example of a solution described with the JSON format

Chapter 4

How to use the tool

In this chapter, we will give all the necessary information to be able to use the tool. First section 4.1 will present how to use the command line interface. Second, we explain how to use and define a script in section 4.2. Then, we present the graphical user interface in section 4.3 and the interactive map in section 4.4. Finally, we explain how to use the `C++` API within a solver in section 4.5.

We remind you that the tool is available at the following address: `https://github.com/SimonCandaele/simulator_VRP`

4.1 Using the simulator in command line

To launch the simulator in the command line mode, use the following command in a terminal.

```
$ python3 main.py
```

When the simulator is started, it is in the *PreSimulation* state. At that point, the first thing to do is to load the different input files. This can be done with the following commands:

```
--:loadCarrier  path/to/CarrierFile.json
--:loadCustomer path/to/CustomerFile.json
--:loadGraph    path/to/GraphFile.json
--:loadScenario path/to/ScenarioFile.jsona
```

As you can see, these commands are simple to use. They each take as argument the path of the file to load. If the file is not valid, the simulator will signal it.

Instead of loading a scenario file, it is possible to generate a scenario from the Customer file previously loaded with the following command:

```
--:generateScenario --ot NUMBER --ct NUMBER
```

This command also has two optional parameters, `--ot` and `--ct`, which can be used to indicate respcectively the offline time and the online time. To use these parameters, simply enter the name of the parameter as indicated and replace `NUMBER` by the time, in seconds, that the paramater must be set to.

The offline time and the computation time can also be configured after loading or generating a scenario. The first two following commands can be used to print their values, and the two others to set the values to the indicated number, always in seconds. The computationTime can still be changed later but before starting the online simulation obviously.

```
--:offlineTime
--:computationTime
--:setOfflineTime NUMBER
--:setComputationTime NUMBER
```

It is possible to use the simulator only to generate scenario and save them into files before using them later. The next command saves the current scenario in the file indicated as argument. One can then generate other scenarios, from the same customer file or not, or simply continue with the current scenario.

```
--:saveScenario path/to/file.json
```

The next two commands can be used respectively to show the vehicle types described in the carrier file, and the list of vehicles available that can be used to build a solution. We can also see the output that would be displayed if the example carrier files in section 3.2 had been loaded beforehand.

```
--:showVehicleType
{'Capacity': 200, 'VehicleType': 'car'}
{'Capacity': 20, 'VehicleType': 'bike'}
--:showVehicles
{'0': {'VehicleType': 'bike'}}
{'1': {'VehicleType': 'bike'}}
{'2': {'VehicleType': 'car'}}
  Number of "car" : 1
  Number of "bike" : 2
```

The next two commands can be used to modify the vehicle fleet. The first one deletes all the vehicles of the current fleet. The second must be used by indicating, at least one pair `NUMBER`, `TYPE_NAME`. For each pair, the simulator adds the indicated number of vehicles with the indicated type to the fleet.

```
--:deleteVehicles
--:addVehicles [NUMBER TYPE_NAME]+
```

The next command indicates if the solver is connected or not. The solver and simulator should automatically connect when they are launched.

```
--:testConnection
```

Once the solver is connected, it must be sent the Carrier, Customer, Graph input files as well as the various parameters. Everything can be sent with the first next command, otherwise files can be sent individually with the following ones. We also show what the output looks like.

```
--:sendAll
  Carrier sent
  Customer sent
  Graph sent
  ComputationTime sent
  OfflineTime sent
--:sendCustomersToSolver
  Customer sent
```

```
--:sendCarrierToSolver
  Carrier sent
--:sendGraphToSolver
  Graph sent
```

It is also possible to provide additional files to the solver that the simulator does not need to manipulate, unlike the files defined so far. The next command takes two arguments. The first one is the path to the file. The second one is a key that the solver will use to access the file. The file must respect the json format.

```
--:sendFile path/to/file.json KEY_NAME
```

Once all the necessary data for a simulation has been defined, the simulation can be started with the following command. We also show an example of output at the end of the offline simulation. The first line of the ouput is quickly printed, and the two others are printed at the end of the simulation.

```
--:startOfflineSimulation
   The solver is ready, offline simulation will start very shortly
   END OF THE OFFLINE SIMULATION
   # valid solution   7  #invalid solution 0   total number of solution 7
```

The simulator will therefore go to the OfflineComputation State.

The simulation can be paused and resumed with the following commands :

```
--:pause
--:continue
```

With these commands, the simulator will change between the two states *OfflineComputation* and *OfflinePause*.

At the end of the defined offlineTime, the simulator will automaticaly reach the state *OfflineEnd*. At this point, before launching the online simulation, it is possible to load a solution in a file to communicate it to the solver. The loaded file should only contain a solution in the form of a JSON object as described in section 3.2.5. The command to load this solution takes the path of the file in argument and looks like this:

```
--:loadNewSolution path/to/file
```

It is then possible to launch the online simulation using the following command. We also show a possible output for these commands. The first line of the output is printed just after entering the command while the two others are printed at the end of the simulation.

```
--:startOnlineSimulation
   The solver is ready, online simulation will start very shortly
   END OF THE ONLINE SIMULATION
   SolverEndOnline # valid solution     55  #invalid solution 0   total number
       of solution 55
```

Once the online simulation has started, it can be paused in the same way as the offline simulation.

At the end of the time allowed for the online simulation, the simulator state will automatically change to *PostSimulation*.

The following two commands can be used to save either all solutions returned by the solver or only the last one. The second command produces a file that can be loaded using the command we saw earlier.

```
--:saveSolutions    path/to/file
--:saveLastSolution path/to/file
```

These command can actually be used regardless of the simulator state, since solutions can be received as early as the offline simulation.

To start a new simulation, the simulator must return to the *PreSimulation* state. This is done through the following command :

```
--:stopSimulation
```

This command can be used not only in the *PostSimulation* state but in any state. However, it also deletes from the simulator memory all the solutions returned by the solver. In order not to be lost, these must be saved before using this command.

Finally, the command to turn off the simulator is :

```
--:close
```

4.2 Using a script

To use the simulator with a script file, it must be started in a terminal as before, but with the `--script` parameter followed by the path of the file to use as script.

```
$ python3 main.py --script path/to/script_file
```

The script file must be a text file containing a command at each line. The commands that can be used in a script file are described in the previous section. However, when the `startOfflineSimulation` or `startOnlineSimulation` commands are used, and an offline or online simulation is started, the next command in the script will only be executed after the end of the offline or online simulation. Therefore, the `pause` and `continue` commands cannot be used in script mode.

The solver must also be started before running the simulator with the script. As soon as it is started, the simulator will execute the script and assumes that if a simulation must be performed, the solver is already ready and listening to the simulator.

A script file is suitable to perform a complete simulation, i.e. an offline simulation followed by an online simulation. But it is not possible to run a second simulation in the same script file.

The figure 4.1 shows an example script. First the files Carrier, Customer and Graph are loaded. Then, two scenarios are generated, with different compution time and offline time, and saved. The computation time and the offline time are then again changed as well as the vehicle capacity of the vehicle type `car`. The input files loaded and the last generated scenario are then sent to the solver, and the offline simulation is started. After the offline simulation, the solutions returned by the solver are saved and the online simulation is launched. The whole solution set is saved at the end of the online simulation.

```
loadCarrier /home/Documents/Carrier.json
loadCustomer /home/Documents/Customers.json
loadGraph /home/Documents/Graph.json
generateScenario --ct 10 --ot 600
saveScenario /home/Documents/scenario_1.json
generateScenario --ct 5 --ot 300
saveScenario /home/Documents/scenario_2.json
setComputationTime 15
setOfflineTime 500
setVehicleCapacity car 1000
sendAll
startOfflineSimulation
saveSolutions /home/Documents/solutions_1.json
startOnlineSimulation
saveSolutions /home/Documents/solutions_2.json
```

Figure 4.1: Example of a script file

4.3 Using the Graphical User Interface

The simulator with the graphical interface is started by using the following command in a terminal:

```
$ python3 main.py --gui
```

Before explaining how to use the graphical interface, it is useful to provide an overview of it. You will find it in Figure 3.4 in section 3.1.4. More detailed screenshots are also provided in Appendix A. The interface is divided into three parts. On the left, there are four tabs, one for each problem input file. At the top right is a map showing the solutions. And at the bottom right is the simulation frame which holds control buttons and shows the current solution details.

Each input file can be loaded in the simulator from its corresponding tab. Once the files are loaded, their contents are displayed in the tabs. We will now explain what can be seen in the different tabs.

The carrier tab is divided in four parts: the Parameters, the Vehicle Types, the Vehicle Fleet and the Travel Times. A screenshot of the carrier tab is available in Figure A.1. Here is the description of the different parts.

Parameters shows three informations. First there is the name of the loaded file, then the total number of timeSlot, and finally the travel time unit, which is the unit of the travel times displayed lower. **Vehicles Types** shows the different vehicle types. The name and transport capacity are displayed for each vehicle type. **Vehicle fleet** shows all the Id and the vehicle type of all the vehicles currently in the fleet. Before starting the simulation, the fleet can be modified thanks to the two buttons above the table. **Travel Times** shows the travel times for the timeslot and the vehicle type selected. The ids of the starts node are in the row headings and the column headings show the end nodes.

The customer tab is divided in three parts: the Parameters, the Potential Requests and the Arrival Probabilities. A screenshot of the customer tab is presented in Figure A.2. Here is the description of the different parts.

Parameters shows four informations. The name of the loaded file, the number of timeslots, the horizon size which is the number of time units for the online simulation, and the real duration

duration of a time unit. The last one should not be confused with the computation time allowed for each time unit. **Potential Requests** contains all the potential requests described in the file. Each column corresponds to an attribute of the requests. The requests can be sorted according to an attribute by clicking on its name. Clicking on a request also highlights the associated node on the map **Arrival Probabilities** contains the arrival probabilities of each request.

The graph tab is divided in two sections: the Parameters and the Nodes. A screenshot of the customer tab is presented in Figure A.3. Here is the description of the different parts.

Parameters shows the name of the file loaded. **Nodes** shows each node with his id and his types. Clicking on a node highlights the corresponding node on the map.

The scenario tab is divided in two sections: the Parameters and the Requests. A screenshot of the scenario tab is presented in Figure A.5. Here is the description of the different parts.

Parameters first shows either the name of the file or if it was generated from the customer file. Then there is the offline time and the computation time. The offline time can be modified before starting the offline simulation, and the online time can be changed before starting the online simulation. **Requests** shows all the requests of the scenario. Like in the customer tab, the requests can be sorted according to an attribute by clicking on the column heading. Clicking on a request also highlights the associated node on the map. New requests can be added by clicking on the button above the table. These requests can be added before and during the simulation.

After the start of the offline simulation, the color of each request also provides information about the request. A request with a white background is a request whose reveal time is in the future. A request with a yellow background has been revealed to the solver, but the request is not yet served in any solution. A request with a red background is a rejected request. A request with a light green background is an accepted request which will be served later. A request with a dark green background is a request which was already served or which is being served.

The map allows to visualize the graph and the solutions in a two-dimensional space. The coordinates contained in the graph file are used to display each node of the graph. If there are none, fictitious coordinates are generated from the travel times in the carrier file. Clicking on a node on the map highlights it, but also the corresponding entries in the tabs graph, customer and scenario to easily get the informations about this node.

Depending on the progress of the simulation, different information will be displayed on this map. In the *preSimulation* state, the node on the map have three possible colors. A depot node is black. A node associated to at least one request in the scenario is grey. The other nodes are just white.

After the start of the offline simulation, the itineraries of the vehicles are displayed. Each vehicle has an associated color, which is used to draw a line connecting each node of its route. A thin line between two nodes indicates that the vehicle will start to travel between these two nodes in the future. On the other hand, a thick line between two nodes indicates that the vehicle has already joined these nodes, or is currently travelling between the two nodes.

The color of a node during the simulation also gives information about the requests emitted from this node. A yellow node means that a request on this node has been revealed to the solver, but that no solution serving this request has yet been sent by the solver. A red node indicates a

request attached to this node that has been rejected. A light green node means that a request attached to this node is included in the solution and will be served in the future. A dark green node means that a request attached to this node has begun to be served or has been served.

Multiple requests can be associated to the same node, requiring to choose which color to display in priority. If a request is accepted, it can also be seen by a route that passes through this node. Green therefore has no priority over the other colours. Yellow, unlike red, indicates temporary information and will eventually turn red or green. For these two reasons, the order of priority is yellow, red and then only green.

The simulation frame is presented in the Figure A.6. It first contains some important control buttons. The button *send Data* sends the input files to the solver before the simulation. The button *play* is used to start the offline and the online simulation. It also resumes the simulation when it has been paused with the button *pause*. The button stop is used to stop the current simulation and come back to the *PreSimulation State*. Beside the butons are displayed the current state of the simulator, the current time unit and the number of solution the solver sent.

The last solution is presented below the buttons. The route of each vehicle is presented horizontally thanks to a succession of rectangles. The first rectangle shows the vehicle involved and the color of its road on the map. By clicking on this rectangle, the corresponding road on the map above can be hidden or displayed. The next ones represent the successive step constituting the route of the vehicle. These rectangle shows the id of the corresponding node, the arrival and departure time for this node and the id of the request served if this is the case. The rectangle in light green are the rectangle the vehicle has not yet reached. When the current time unit reaches the arrival time indicated in a rectangle, this color of this rectangle will turns to dark green.

4.4 Displaying on an interactive map

Regardless of the mode (in command line or with the GUI) in which the simulator was launched, it is possible to display the last solution sent by the solver in a web browser on a realistic map. Figure 3.5, in section 3.1.5, presents a screenshot of the map.

This map can be opened while a simulation is running or not, to view the last solution returned by the solver. To open the map, open the `map.html` file, located in the simulator files, with a web browser. The map will look for the information to display in a file updated by the simulator. However, the file must be specified in the browser after opening the html file. Instructions for finding this file are displayed in the browser.

While the simulation is running, the map is updating at each time unit, and each time a new solution is returned by the solver.

The graph is represented on the map with the markers. A black marker represents a depot, the other nodes are represented by blues markers. For each vehicle, a colored line connects each point on the road assigned to that vehicle. A continuous line between two points means that the vehicle has already connected these two points or has already left the first and will join the second. A dashed line between two points means that the vehicle has not yet started the travel between these two points.

By clicking on a marker, the id of the node corresponding to this marker, and the requests served at this node in the solution are displayed on the right of the page.

4.5 The API

We will now present the necessary information for the use of the API in a solver written in `C++`. To present the functions of the API, we will use a skeleton code presented in appendix B.1, and a pseudocode summarizing the skeleton code. The pseudocode of the solver can be seen in 4.2. We will often indicate sections of the skeleton code to illustrate the use of the functions presented here. This code is divided into several sections. The first one corresponds to the preSimulation state, where the different input data are recovered. The second one corresponds to the offline simulation. The third one corresponds to the OfflineEnd state, between the end of the offline simulation and the beginning of the online simulation. The fourth one corresponds to the online simulation.

The API we have developed uses a JSON library that needs to be installed in order to use ours. All information about this library can be found here `https://github.com/nlohmann/json`. Some functions we present return objects of the json class coming from this library. To know how to use them, we invite the reader to consult the very complete and detailed resources at the address provided.

The api that was developed to allow communication between the simulator and a solver has been implemented in `C++`. To use the API in a `C++` solver, one must add, in the file where the API will be used, the following :

```
#include "vrpAPI.h"
```

To use the API, the first thing to do is to instantiate an object from the `vrpApi` class. Then one must call a method that allows the API to receive messages from the simulator. As long as this function is not called, the connection with the simulator will not happen and it will be useless to use the other functions.

```
void startListeningSimulator(bool verboseAPI);
```

This function returns nothing and take one boolean argument which is used to set the API to a verbose mode or not. In the skeleton code, this function is called at line 11, with the parameter set to `false` to not use the verbose mode.

After calling the previous function, the API is ready to communicate with the simulator, but the simulator may not yet have been started. To find out if the connection is established with the simulator, the following function can be used.

```
bool testConnection();
```

This function takes no argument and returns a Boolean value indicating whether the connection with the simulator is established or not. In the skeleton code, this function is used at line 13, where the solver waits for a connection before continuing.

Once the connection is established with the solver, one will generally want to retrieve the initial data of the problem. The following function is made for this purpose. It takes no argument and returns a JSON object, from the library indicated at the beginning of this section, containing all the initial data sent so far by the simulator. The returned object will be empty if the simulator has not sent anything yet. Note that it is not with this function that the solver will access the scenario requests.

```
json getVrpData();
```

```
#include "vrpAPI.h"

int main(int argc, char** argv) {

  vrpApi myVRPAPI;
  myVRPAPI.startListeningSimulator(false);

  while(not myVRPAPI.testConnection()){
    // wait for simulator to be connected
  }

  while(not dataReceived){
    if(myVRPAPI.vrpDataChanged()){
      myVRPdata = myVRPAPI.getVrpData();
  }}

  myVRPAPI.readyForOffline();

  while(myVRPAPI.isNewRequests()){
    json newRequest = myVRPAPI.popNewRequest();
    // do something with the new request
  }

  while(myVRPAPI.getCurrentSimulationState() == "offlineComputation"){
    // perform computation, find a solution
    myVRPAPI.updateBestSolution(mySolution);
  }

  myVRPAPI.readyForOnline();
  while(myVRPAPI.getCurrentSimulationState() == "onlineComputation"){
    while(myVRPAPI.isNewRequests()){
      json nR = myVRPAPI.popNewRequest();
      // do something with the new request
    }
    cu = myVRPAPI.getCurrentTimeUnit();

    // compute solution
    myVRPAPI.updateBestSolution(mySolution);
  }
}
```

Figure 4.2: minimal pseudo code of a solver

The data that the simulator must always send before starting the offline simulation are the **carrier**, the **graph**, the **customer** and the **offlineTime**. To know if each of them has already been sent by the simulator, it is necessary to inspect the object returned by the function `getVrpData()`. The object returned represents a JSON object whose internal structure is presented in Figure 4.3.

```
{
    "Carrier"           : CARRIER_OBJECT,
    "Customers"         : CUSTOMERS_OBJECT,
    "Graph"             : GRAPH_OBJECT,
    "OfflineTime"       : DOUBLE,
    "OnlineTime"        : DOUBLE,
    ADDITIONAL_FILE_KEY : ADDITIONAL_FILE_OBJECT
}
```

Figure 4.3: Internal Structure of the json object returned by the function getVrpData()

As you can see, it is a JSON object with several pairs. Each of these pairs is only present in the object if the simulator has already sent the corresponding data. Pairs named **Carrier**, **Customers** and **Graph** respectively contain the objects **Carrier**, **Customers** and **Graph** loaded by the simulator and then sent to the solver. The solver thus receives JSON objects with internal structures that we have detailed in the section 3.2. The attentive reader will recall that we explained in section 4.1 how to send one or more files with additional data to the solver. As a reminder, to send an additional file, this one had to represent a JSON object, and a key allowing the solver to find this file had to be given. We now see how the solver can concretely access these data. Each of the additional files is in the object returned by the function `getVrpData()`, in a pair whose name is the key given to the solver when loading the file, and whose value is the json object described in the file. The time allocated for the entire offline simulation is provided as the value of the pair named *OfflineTime*. The time allocated for each time unit of the online simulation is provided as the value of the pair named *OnlineTime*. Both are of the type `double`. One can see in the skeleton code starting at line 38, how we create JSON objects containing the different data loaded.

In the skeleton code, we show how it is possible to wait for the simulator to send all the expected data before moving on. In this case, it is an example with a while loop starting at line 21 of the code. Just after, on line 22, the following function is used.

```
bool vrpDataChanged();
```

This function takes no argument, and returns a boolean indicating if the simulator sent new data after the last call to the function `getVrpData()`. The return value is `True` if there is new data, `False` otherwise. In our example, it is used to find out if it is useful to reload the data.

To finish with functions `vrpDataChanged()` and `getVrpData()`, note that it also works to load the data after the start of the offline simulation. We are now done with the first section of the skeleton code, and we move on to the second section.

In order to start the offline simulation, it is mandatory to use the following function, like at the line 49 of the example skeleton code.

```
void readyForOffline();
```

This function blocks the execution of the solver, until the simulator starts the offline simulation. At the exact moment when this function returns, and the execution of the simulator resumes, the offline simulation begins. The solver therefore has exactly the time indicated by *offlineTime* to calculate and submit solutions to the simulator. Note also that if one only wants to perform a dynamic simulation whitout resolving a static problem, it is still mandatory to use this function. The simulator assumes that there is always an offline part, but it is possible to set the time allocated to it to zero, and use the solver accordingly.

The solver can submit solutions as long as the offline simulation is running. To find out the simulation state, the solver can use the following function :

```
std::string getCurrentSimulationState();
```

This function returns a string describing the current simulation state. The value returned is self explanatory, and can be one of the following strings : `"preSimulation"` before the offline simulation, `"offlineComputation"` during the offline simulation, `"offlineEnd"` after the offline simulation and before the online simulation, `"onlineComputation"` during the online simulation, `"onlineEnd"` after the online simulation. In the skeleton code, at line 58, we suggest to use this function in a while loop, to keep computing solutions while the simulation is in the offline state.

The other way to know the progress of the offline simulation is by using the following function:

```
double getCurrentTimeUnit();
```

During the offline simulation, this function returns a value between -1 and 0, proportional to the advancement of the offline time. At the start of the offline simulation, the value returned is near to -1, and at the end, the value is near to 0. As the offline time is known, it is possible to calculate the number of seconds remaining as in the skeleton code at line 66. When the offline simulation is over and the value returned by the function `getCurrentSimulationState` is `"offlineEnd"`, the value returned by the function `getCurrentTimeUnit()` is -2. It has been chosen not to return zero because this can correspond to the end of offline time but also to the beginning of online time. Instead, when the current state does not match either the online simulation or the offline simulation, a value that does not belong to any of the time intervals is returned. This value is -2.

To receive scenario requests, the solver can use a specific function which is:

```
json popNewRequest();
```

This function takes no argument and returns one JSON object describing one request from the scenario. Each request is poped only once by this function. The function only returns requests whose reveal time has passed. When several requests wait to be poped, the request with the smaller reveal time is poped first. When no request is available and the function is called, an empty json object is returned. In the skeleton code, this function for popping offline requests is used at line 52. Since all offline requests are revealed at the beginning of the simulation, the skeleton code suggests using these functions only once at the beginning of the offline simulation. However, during the online simulation, requests may appear throughout the simulation, so these functions should be used more regularly.

The next function can be used to know if there are requests waiting to be poped using the previous function. It is a function taking no argument and returning a boolean value indicating if there are new request that the solver can retrieve with the previous function. This function is used in the skeleton code at line 51.

```
bool isNewRequests();
```

To send a solution to the simulator, the solver must represent this solution with a json object respecting the structure defined in section 3.2.5. Once the solution is ready, it can be sent to the solver with the following function:

```
bool updateBestSolution(json newSolution);
```

This function takes as argument a json object describing a solution. The returned is a boolean equal to `true` if the solution was sent to the solver and `false` if not. If the returned value is `false`, it means that the either the offline simulation or the online simulation is over, depending on which one was started. In the skeleton code, this function is used at line 70.

When the offline simulation is over, the state of the simulation is `"offlineEnd"`. That corresponds to the section 3 of the skeleton code. In that section, one can see that the computation time is retrieved in the same way the previous data were retrieved. Remember that the simulator offers the possibility to define or redefine the computation time between the offline and online simulation. The solver must be written according to how you want to use it. It is up to the solver to check if the computation time value has not been changed when it starts the online simulation. But it is not necessary if this parameter is defined at the beginning and is not modified thereafter.

We now move on the fourth and final section of the skeleton code, which corresponds to the online simulation. This section is very similar to the offline simulation section. First of all it is mandatory to use the following function:

```
void readyForOnline();
```

This function has the same role as the function `readyForOffline()`, except that it is for the online simulation. This function will block the execution of the solver until the simulator starts the online simulation. When the function returns, it corresponds to the beginning of the online simulation. This function must also always be used after function `readyForOffline()` has been used, as the order of the different steps of a simulation must always be respected.

During the online simulation, the function `getCurrentTimeUnit()` returns a value between 0 and the Horizon Size, corresponding to the current time unit.

When the simulator submits a solution, it is possible that it is false. In this case, the simulator will send the last valid solution received to the solver. The solver can use a first function to find out if a solution has been sent to it by the simulator, and a second to retrieve that solution. The first function is:

```
bool isNewCurrentSolution();
```

This function takes no argument, and return a boolean value. If the returned value is `true`, then the simulator sent a solution. If the returned value is `false`, then there is no solution to retrieve. To retrieve the solution sent by the simulator, the solver can use the following function:

```
json getNewCurrentSolution();
```

This function takes no argument and returns a json object representing the last solution that the simulator sent.

Let us also remember that on the simulator side, before starting the online simulation, it is possible to send a solution to the solver. This means that offline and online simulations do not always have to be run in sequence, but the intermediate solution can be saved and the online simulation performed later. In the skeleton code, we suggest to use the two previous function to eventually retrieve this solution at the beginning of the online simulation, just after calling the function `readyForOnline()`

Now let's explain how the solver can be paused by the simulator. In order for the simulator to pause the solver, it must be enabled when writing the solver. It is up to the solver to check if a pause is not requested by the simulator. To do so, it must use the following function :

```
void pauseIfSimulatorAsks();
```

This function takes no argument and returns nothing. If no pause was asked by the simulator, this function returns immediately. If, on the contrary, the simulator has requested a pause, then the function will block the execution of the simulator. And the function will only return when the simulator ends the pause. If the solver must be able to be paused by the simulator, it must therefore use this function regularly during offline and online simulations. When the simulator wants to pause the simulation, the pause will only start when the solver uses this function. In the skeleton code, it is suggested to use this function in a loop calculating solutions while the simulation is still running, as seen at lines 59 and 104.

Chapter 5

Implementation of the tool

In this section, we will discuss the implementation of the tool. More precisely, we will review the different components of the system formed by the simulator and the API developed. First, a general overview of the different components will be presented in section 5.1. Second, we will present the framework that we used to communicate between the simulator and the API in the section 5.2. Third, the section 5.3 will explain the components of the simulator. Then, the API will be examined in section 5.5. The section 5.4 will talk about the map inside a web browser. Finally, in section 5.6, we will make the link between the different elements with an example illustrating how some of the components interact with each other.

The tool and all source files are available here: `https://github.com/SimonCandaele/simulator_VRP`

5.1 General Overview

The general architecture of the solver, the simulator and their sub-components is presented in figure 5.1. On the far left stand the different entries of the simulator, which are the user actions and input files. The simulator is represented by the big rectangle at the center. The solver is represented by the rectangle on the right. These two processes are executed on the same machine and communicate on localhost. Above the simulator, the interactive map embedded in a web browser is shown. It accesses an output file of the simulator.

We therefore distinguish two main components, the simulator and the solver, themselves comprising several sub-components. The simulator was implemented with Python 3. The solver API was implemented in `C++`. In order to communicate between these two processes developed with different languages, we used an open-source remote procedure call framework called `gRPC`. To display the solutions on a map inside a web browser, we use HTML, CSS and JavaScript.

One of the ambitions of the simulator is to show that it is not necessary that the solver be implemented in the same language. This is why the system architecture we present here offers the possibility of being extended to solvers written in other languages. The `gRPC` framework allows processes using different languages to communicate. It is therefore possible to implement a solver API in Java or Python that would communicate with the existing simulator, without requiring any modification on the simulator side.

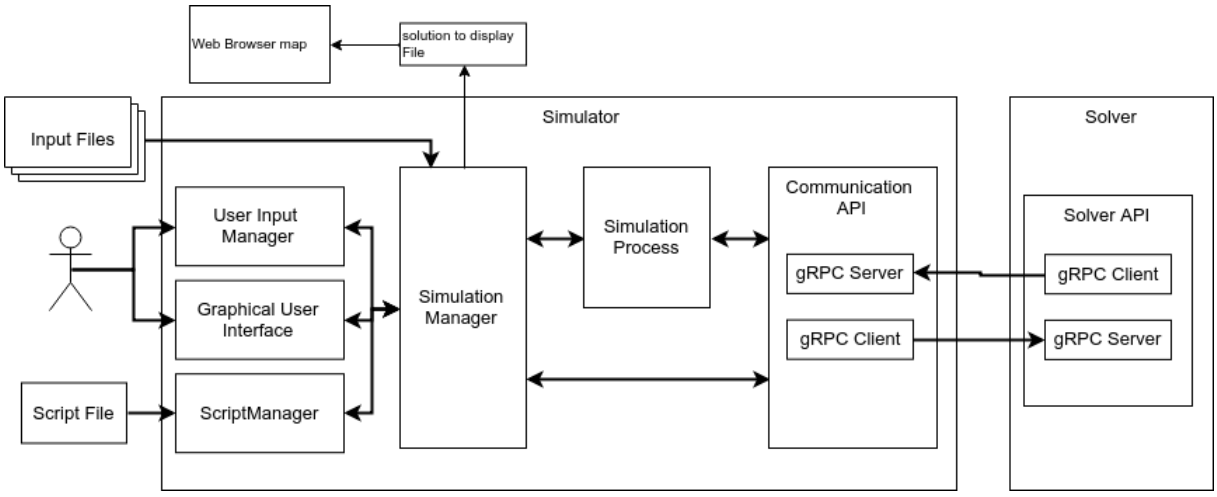


Figure 5.1: General architecture of the simulator and the solver

- **The Simulator** is the most important part. Its role is to perform the simulation of a VRP problem to a solver, according to the instructions received by a user. It is composed of several sub-parts, each with a specific role.
  - **User Input Manager, GUI and script Script Manager**. The simulator must first receive instructions from a user to perform the desired simulation. For this, there are three possible components that can take the role of retrieving these instructions. Each one corresponds to one of the modes of use of the simulator. Depending of the mode in which the simulator is launched, the corresponding component is used. The User Input Manager is the component managing the command line interface. The Script Manager is the component that will read the script file. The Graphical User Interface must communicates the actions of the user to the Simulation Manager, but also present the evolution of the simulation to the user.
  - **Simulation Manager**. The main role of this component is to interpret the instructions given to it by one of the first three components, and to perform the appropriate action based on these instructions.
  - **Simulation Process**. The role of this part is to take care of the real timing issues during the offline and online simulation. Its main task is to ensure that requests are communicated to the solver at the right time.
  - **Communication API**. This component contains all the functions that allow the other components to communicate with the solver. It is in this component that the `gRPC` framework is used by the simulator.
- **The solver** is the process that will solve the problem simulated by the simulator. However, we are not interested here in how to solve a problem. We will therefore not explain this component, but rather the API and its components that allows him to communicate with the simulator
  - **The Solver API** allows a solver to communicate with the simulator. To do this, it uses the `gRPC` framework.
- **The interactive map** can be open with a web browser to display the last solution returned by the solver. It reads the information to display in a dedicated output file of the simulator, which is updated as the simulation proceeds.

5.2 Communication between the simulator and the solver

The communication between the simulator and the solver is done with a framework called `gRPC`, which is an open source remote procedure call framework developed by Google. All information about this framework are available on the website [1]. The main reasons why this framework has been chosen are the ease of use of the framework and that it works across a variety of languages and platforms. This makes it easy to establish a communication between the simulator and the solver even though they are developed in different languages.

A lot of remote procedure call protocols, or RPC protocols, exist. They all have a common goal, to make the process of executing code on a remote machine as simple and straighforward as calling a local function [4]. These protocols are defined as a client-server interaction. On the client side, a procedure call is made, and when the results are returned, the execution proceeds. The server define the routines that it wants to made available for remote callers. The RPC system makes the link between the two.

`gRPC` is based on the idea of defining a service, that is a set of methods that can be called remotely with their parameters and return types. On one side, the interface is implemented and a `gRPC` server is running to handle client calls. On the other side, a client has a `gRPC` stub providing the same methods as the server. The Interface Definition Language, in which the services are described, that is used by default by `gRPC` is the *protocal buffers*. For more information about *protocal buffers*, we invite the reader to consult [3].

For our tool, there are two services defined between the simulator and the solver API. This way, the simulator and the solver API can send messages when they want to communicate information to each other. This is shown in Figure 5.1, where in both the simulator communication API and the solver API, there are a `gRPC` server to receive messages, and a `gRPC` stub to send messages.

The first step to use `gRPC` was to define the two services. This is done in a *proto* file, which is a text file with a *.proto* extension. The *proto* file of our tool is named *vrpAPI.proto*. After defining the services, we used the protocol buffer compiler `protoc` with a `gRPC` plugin to generate code from this *proto* file in both Python and `C++`. That is how we easily get generated the client and server code both for the Python simulator and the `C++` API with a single framework, `gRPC`.

The two services defined in our *proto* file are named *SimulatorMessages* and *SolverMessages*. The first one is used by the simulator communication API to send messages to the solver API. The second one is used in the other direction, by the solver API to send messages to the simulator. The content of our *proto* file describing the two services is presented in Figure 5.2. The behaviour of the RPCs in the two services are detailed in the sections 5.2.1 and 5.2.2.

The protocol buffer data is structured as *messages*. The parameters and the return types of the RPC methods in the defined services are specified as protocol buffer messages. Four message types are defined in our *proto* file. The types `JsonMessage`, `DoubleMessage` and `BoolMessage` are used to transmit respectively a string (that is expected to represent a Json structure), a double and a boolean. The message type `Empty` is used as an empty parameter or return value.

5.2.1 The SimulatorMessages service

This service is a set of RPC designed to be used by the simulator. These RPCs are used in the simulator communication API, which is itself used by the simulator to send and request information from the solver and its API. We will now explain the behaviour of each RPC.

```
service SimulatorMessages {
  rpc isReadyForOffline(Empty) returns (BoolMessage) {}

  rpc isReadyForOnline(Empty) returns (BoolMessage) {}

  rpc loadJson (JsonMessage) returns (Empty) {}

  rpc loadNewRequests (JsonMessage) returns (Empty) {}

  rpc pauseSimulation (Empty) returns (DoubleMessage) {}

  rpc continueSimulation (Empty) returns (DoubleMessage) {}

  rpc testConnection (Empty) returns (Empty) {}

  rpc setCurrentSolution(JsonMessage) returns (Empty) {}

  rpc shutdown (Empty) returns (Empty) {}

  rpc startOfflineSimulation (Empty) returns (DoubleMessage) {}

  rpc startOnlineSimulation (Empty) returns (DoubleMessage) {}
}

service SolverMessages {
  rpc notifyEndOffline(Empty) returns (Empty){}

  rpc notifyEndOnline(Empty) returns (Empty){}

  rpc sendBestSolution(JsonMessage) returns (Empty) {}

  rpc testConnection(Empty) returns (Empty) {}
}

message JsonMessage {
  string jsonstring = 1;
}
message DoubleMessage{
  double value = 1;
}
message BoolMessage{
  bool b = 1;
}
message Empty {}
```

Figure 5.2: Content of the *proto* file, defining the two services between the simulator and the solver

testConnection takes an empty parameter and returns an empty value. This RPC is used to check is the solver API is connected by trying to execute an empty function on the solver system and to get back the empty response.

loadJson is used by the simulator to transmit a JSON object, represented with a string, to the solver API. More specifically, this is the RPC used to transmit the input files *Carrier*, *Customers* and *Graph* as well as any additional file the user want to transmit to the solver.

isReadyForOffline and isReadyForOnline are two RPC taking an empty parameter and returning a boolean value. They are used respectively before the offline and the online simulation. The returned value is a message with a boolean value indicating if the solver is ready to start the offline or online simulation. Before starting a simulation, the simulator waits for a positive returned value using one of these two methods.

startOfflineSimulation and startOnlineSimulation are the two RPCs that are used to launch respectively an offline or an online simulation. Both take an empty paramater, and return a message with a double value. The double value represents the moment that will be taken as the start time. This value is the number of seconds separating the start time of the simulation and the epoch time of the system.

loadNewRequests is used by the simulator to communicate a new request to the solver API. This RPC takes JsonMessage as a parameter. The string of the message should describe the request as a json object.

pauseSimulation is the RPC that is used to tell the solver API that a pause has to be started. It takes no parameter and returned a double value describing the start time of the pause. This time is described in same way as the start time of the simulation.

continueSimulation is the RPC that is used to tell the API to stop the simulation pause. It takes no parameter and return a double value describing the time point where the simulation resume. The time point is described in the same way than with the start time of the simulation.

setCurrentSolution is used to communicate a solution to the solver. It takes a JsonMessage as a parameter. The string of the message should describe a solution.

shutdown is the RPC used to inform the solver API of the end simulation.

5.2.2 The SolverMessages service

This service defined the RPC that are used by the solver API to send messages to the simulator. The solver never directly use these procedures.

notifyEndOffline and notifyEndOnline are two RPC used respectively at the end of the offline simulation and online simulation. The purpose of these RPCs is to inform the simulator that the solver reach the end of the simulation. After calling one of these RPC, the simulator has the confirmation that the offline or online simulation is finished on the solver side. He knows then that no more solution for the corresponding simulation will be sent.

sendBestSolution is the RPC that is used to communicate a new solution to the simulator. The function used by the solver to send a solution to the simulator, uses itself this RPC. The json object describing a solution transmited by the solver is converted in a string that is passed inside the RPC parameter.

testConnection is a RPC similar to the RPC with the same name in the other service. This is a RPC taking an empty parameter and returning an empty value. It is used to verify if the connection with the simulator is working.

5.3 The Simulator Components

The real core of the simulator is the Simulation Manager that we present in the section 5.3.3. It is around this component that all the others are articulated. Depending on the mode in which the simulator is used, on the three components presented in the section 5.3.1 and 5.3.2 will communicate the orders of the user to the Simulation Manager. When running an online or offline simulation, the Simulation Manager use a simulation process that we present in section 5.3.4. And to communicate with the solver API, the simulator use its own API that we present in section 5.3.5.

5.3.1 User Input Manager and Script Manager

These two components both works in a similar way. The User Input Manager is active when the simulator is used in command line, and the Script Manager is active when the simulator was launched with a script.

The User Input Manager consists of a thread, whose role is to retrieve commands entered in the terminal by the user. When the user enters a command, it is sent to the Simulation Manager thread via a fifo queue. Then the User Input Manager wait for a notification from the Simulation Manager, signifying that the next command can be sent. The User Input Manager then allows the user to enter a new command.

The Script Manager is also a thread, but it retrieves the command from a file specified when launching the whole simulator. After sending one line of the script file to the Simulation Manager, the Script Manager also waits for a notification before reading and sending the command at the next line in the file. However, the Simulation Manager does not send this notification always at the same time for the Script Manager and the User Input Manager. When a simulation (offline or online) is started, the Simulation Manager waits until the end of this simulation before informing the Script Manager that the next command can be send. Without this, it would not be possible to perform an offline simulation followed by an online simulation with a script.

When the Script Manager reach the end of the script file, It automatically sends a command to the Simulation Manager to shutdown the simulator.

5.3.2 the Graphical User Interface

The graphical user interface (GUI) was developed using the Python `tkinter` package. This package is available on most Unix platforms and on Windows systems and usualy comes bundled with python. So it is unnecessary to install any additional library to use the graphical user interface and that simplifies things. These reasons influenced the choice of this library.

The GUI also runs in a thread distinct from that of the Simulation Manager. User actions on the interface are translated into commands and then sent to the Simulation Manager using a fifo queue like the User Input Manager and the Script Manager. The Simulation Manager

thus receives the same commands as with the User Input Manager. But contrary to the User Input Manager, the GUI needs to receive messages from the Simulation Manager to display the solutions and other informations. For that, there is a second fifo queue that is used by the GUI to receive messages from the Simulation Manager.

The main window of the GUI is managed within an object of the class `mainWindow`. The main window holds six components which are the four tabs that each correspond to one of the input file, the embedded map and the simulation frame where the solution details are displayed. Each of the six component and their inner widgets are managed with an object of a dedicated class. They are thus seven classes used for the GUI management.

The GUI keeps track of the current simulation status (see figure 3.3) as it necessary to adapt the GUI to the current state of the simulation, i.e. by enabling or disabling buttons.

The Simulation Frame and the Map frame use each use a canvas widget. The Canvas widget is a general purpose widget which can be used for drawings. Each item drawn on a canvas (like a line, a rectangle, a text, etc...) are canvas items which tkinter keeps track off. We discovered during the development of the GUI, that redrawing all of the many items in a canvas at the same time to display a new solution was an operation that takes a significant amount of time. This resulted in an interface that had a response time too high to be used properly when lots of solutions were sent by the solver in a short amount of time. Optimizations in the display of the solutions therefore had to be made.

The first trick used to optimize the use of the canvas is to not update the display of the canvas if it was done less than a second ago. Updating the canvas with shorter delays would not be useful, as no one would be able to look at something that would be displayed so briefly. All the solutions returned by the simulator will thus not be displayed because, with each canvas update, it will be the last solution returned until then which will be displayed.

The second trick used is to optimize the number of canvas manipulations at each update. The simulation frame and the map frame keeps track of each item drawn on their canvas and their attributes (i.e. the coordinates and the color of the item) in order to not use `tkinter` methods to retrieve informations on items previously created. To display a new solution, item creation and deletion are used as a last resort, with modification of existing items being used first. The amount of additional RAM used is not significant and greatly speeds up canvas updates.

5.3.3 The Simulation Manager

This part is the core of the simulator. Its role is to manage the overall operation of the simulator and coordinates the other components by reacting to three kind of messages. These messages are the user commands, the messages coming from the simulation processes we will desbribe in a further section, and the messages coming from the solver.

This component is a thread whose running function consists in a big loop that will run from the simulator start-up to the simulator shut-down. Inside the loop, three fifo queues are being read. A first queue is used by either the User Input Manager, the Script Manager or the GUI to communicate the user commands to the Simulation Manager. The Simulation Process we will describe further use a second queue to send messages concerning the unfolding of the offline or online simulation. The third queue is used to receive messages coming from the solver.

The loop is executed while there are elements in the queues. When the queues are empty, the thread wait for a notification from the threads putting elements in the queues to retrieve and

manage the new messages.

The Simulation Manager keeps track of the current simulation state in order to react to the inputs in an apporpriate way. However, the Simulation Manager use two more states in addition to those presented in the figure 3.3. These two states are named `OfflinePauseAsked` and `OnlinePauseAsked`. When a pause is aked by the user, the message is communicated to the solver. The pause will only begin when the solver reads this message, send an acknowledgement message to the simulator and start the pause. Since the simulator can't predict how long it will take before the solver starts the pause, two new states are used to manage the manage the transition between when the pause was asked and when the pause realy start. When the simulator is in one of these two states, it proceeds the simulation and wait for the message from the solver confirming that the pause has been started.

5.3.4 The Simulation Process

When the simulator perform a simulation, offline or online, it needs to deal with real-timing issues. The simulator must ensure that the offline simulation will last exactly the time indicated by the *offlineTime* parameter. For the online simulation, each time unit must last the time indicated by the *computationTime* parameter. The Simulator must also ensure that these constraints are respected even if pauses were made during the simulation. And most importantly, each request must be communicated to the solver to be available when the reveal time of the request is reached.

To manage these issues, the Simulation Manager launch another thread which is of the class `simulationOfflineThread` or `simulationOnlineThread`. Such a thread is started when the Simulation Manager receives a command to start an offline or online simulation. The simulation thread finish its execution after reaching the end of the simulation or after being stoped. The two classes are similar but are each design to meet the specific needs of the offline and online simulation.

5.3.5 The communication API of the simulator

The simulator uses several classes to communicate with the solver. These classes are all in the file `simulatorAPIClasses.py`.

The class `simulatorAPI` encapsulates the RPCs of the service named *SimulatorMessages* defined in the proto file of figure 5.2. This class allows to use the RPCs while ignoring the manipulation details of the generated `gRPC` code. The functions of this class are used both by the Simulation Manager thread and the Simulation Process to send messages to the solver API. The Simulation Manager use some function to send the different input files. Some other functions are used by the simulation process to communicate the requests that arise during the simulation, or start the simulation.

The class `SolverListenerThread` define a thread inside which the `gRPC` server is running. When the server receives a message in the form of a remote procedure call performed by the solver API, it executes the code of the corresponding function of the class `SolverMessagesImpl`. The skeleton of this class was generated by the `protoc` compiler as explained in section 5.2. This skeleton was then modified to obtain the desired behaviour when receiving a message from the simulator. The functions in this class are the function that are remotely called by the solver API. The functions of this class will simply transmit the received message to the Simulation Manager

using the queue intended for this purpose. The functions are very short in order to be executed quickly so that the solver API receive the return value of the RPC fast, and can continue its execution.

5.4 The interactive web browser map

The purpose of this part, is to allow the user to visualize the evolution of the solution during the simulation, on a map whose aspect is familiar to many people, like those of Google Maps or OpenStreetMap.

We implemented this feature in a single html file, using CSS to style the page and Javascript for some interacting features. This file has to be open in a web browser. For the map, we use the javascript library *Leaflet* [2]. This library has the advantages of being open-source, lightweight and easy to use. For the rendering of the map, we use the tile layer of OpenStreetMap. We do not use any other library.

When the page is opened, the user is asked to load the file that will contain the solutions to display. During the simulation, this file will be regularly updated by the Simulation Manager. The Simulation Manager actually updates it at each new time unit, and at each new solution from the solver. When the page detect that the file has been modified, it updates the solution displayed on the map according to the new content of the file.

5.5 The Solver API

The solver API consists of three classes but only one is used by the solver directly. The class `vrpApi` is the one used directly by the solver. The functions of this class that are used by the solver were presented in section 4.5.

The class `SimulatorMessagesHandler` contains the function that are remotely called by the simulator. These functions are an imlementation of the RPC defined in the service `SimulatorMessages` inside the proto file presented in figure 5.2. In order to serve the remote calls of procedure performed by the simulator, a `gRPC` server has to run in a dedicated thread. This server is actually started when the solver calls the function named `startListeningSimulator` of the class `vrpAPI` that we already presented in the section 4.5.

When a RPC is used by the simulator to communicate an input file or a new request to the solver, it is actually a function inside the class `SimulatorMessagesHandler` that will be executed first. To transmit an input file or a a request, the function inside this class will save the information inside the dedicated attribute of the object of the class `vrpApi` used by the solver. The solver will access the information by using its dedicated function of the API.

The class `SolverMessagesImpl` defines a set of functions that effectively perform the call to the remote procedure of the service SolverMessages. This class make use of the code generated `gRPC` code. When the solver calls function of the class `vrpApi` to communicate a new solution to the simulator, the function will actually use a function of the class `SolverMessagesImpl` to send the message to the simulator.

5.6 The mechanism to start a simulation

This section will aim to illustrate how some of the different components we have presented interact through a concrete example. All the components we have presented will therefore not be involved in this example. So we can focus on the components a little more complex than some others.

Two important steps in a simulation are the starts of the offline and the online simulations. We will describe here how the start of the online simulation takes place. More precisely, we are interested in how the simulator and the solver API coordinate to start. The mechanism to start offline simulation is identical but uses the functions dedicated to offline simulation. Figure 5.3 shows a sequence diagram representing the sequence of interaction between the system components leading to the start of the online simulation. Keep in mind that two distinct programs are represented on this diagram. The Simulation Manager, the Simulation Process and the Simulator API are part of the simulator, while the SimulatorMessagesHandler, the vrpAPI and the Solver constitutes a second Program. These two Programs are executed on the same machine and communicate on localhost.

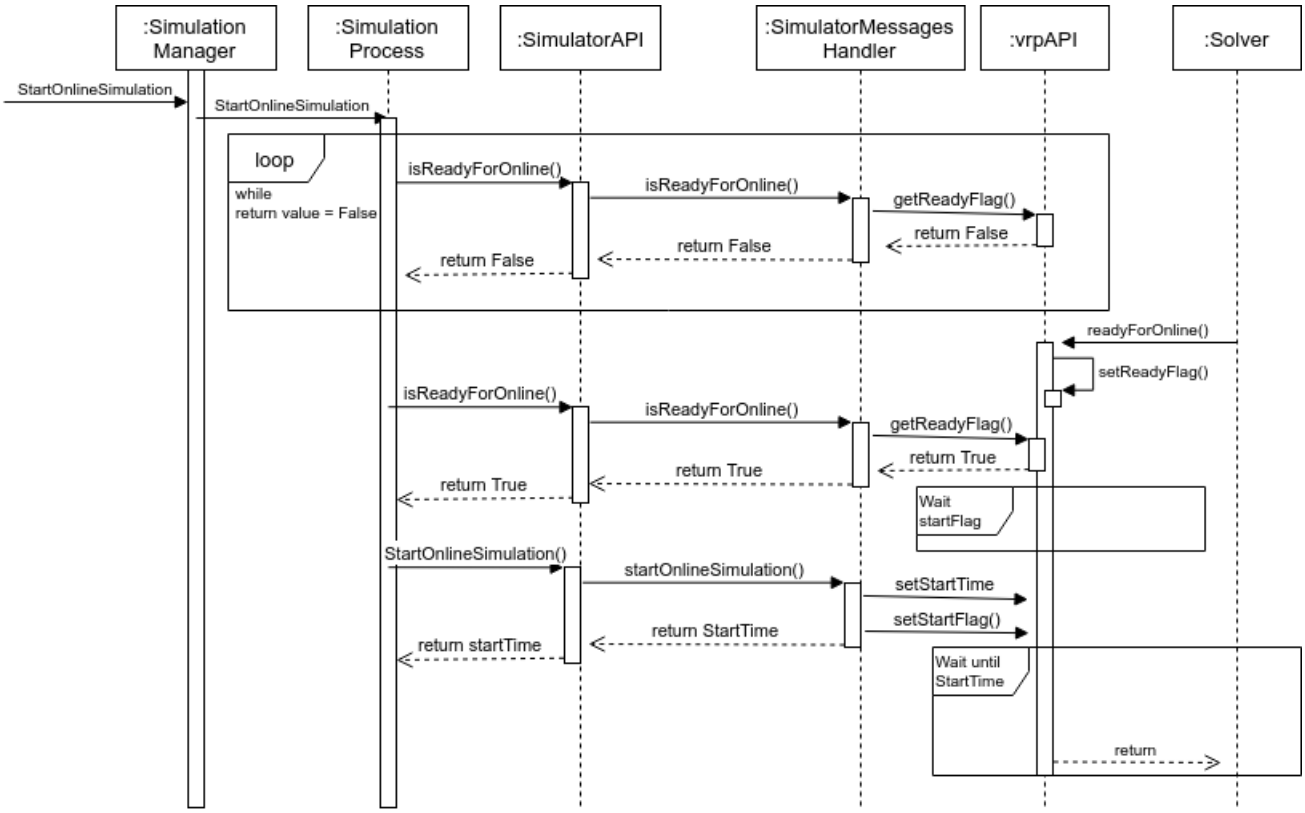


Figure 5.3: Sequence diagram representing the start of the online simulation

The mechanism to launch the online simulation starts with the Simulation Manager that receives the command *startOnlineSimulation*. This causes the Simulation Manager to start the a Simulation Process and then continue its execution. To be precise, the Simulation Process here is a thread of the class `simulationOnlineThread`.

The Simulation Process then enters a loop where it waits until the solver is ready to start the simulation. In this loop, it uses the `isReadyForOnline()` function of the simulator API. This function uses the RPC with the same name which cause the corresponding function of the `SimulatorMessagesHandler` class, inside the solver API, to execute. This procedure get the

value of a flag inside the `vrpApi` instance and returns it. The flag is used only to indicate if the solver called the function. The Simulation Process will stay in this loop as long as the returned value is false.

The flag is set to True by the function `readyForOnline()` which is called by the solver. This function will wait for another flag, the startFlag, to be set to True. When this flag is set, the function will then look what is the time point defined as the start time of the online simulation. Both the flag and the start time are set by an exterior function. The function will until the defined start time, and then will return. At this moment, the online simulation begins.

Let's get back to the Simulation Process. When the `isReadyForOnline()` function returned the value True, the Simulation Process then called another function, `startOnlineSimulation()`. This function uses the RPC of the same name. The remotely called procedure defined in the `SimulatorMessagesHandler` class decide a time point located in a few seconds in the future of the current time point. This time point will be the startTime of the simulation. The procedure set the communicate this time point and set the flag on which the function of the class `vrpApi` is waiting. The value of the time point is also returned to the Simulation Process, via the return value of the RPC.

This mechanism illustrates several things. It shows how the simulator and the solver, two programs executing on the same machine, find a time point that they will both used as the start time of the simulation. By knowing exactly when the solver starts calculating a solution, the time given to the solver to submit solutions can be controlled with great accuracy. This also allows both the solver and the simulator to know exactly the current time unit of the simulation. Finally, this mechanism allows a better understanding of the relationships between the different components of the system.

Chapter 6

Proof of Concept

In this chapter, we put ourselves in the situation of a person wishing to test the efficiency of his algorithm on different data sets. The goal will be to use the simulator to automate a set of simulations and to recover the solutions found by the solver for later analysis. Then we will perform several case studies to analyze the impact of different parameters for the simulations that we will have carried out.

Section 6.1 presents the three different datasets that we will use. Then we do some case studies in section 6.2 with problems from the three datasets. For each dataset, we translated the files in the format used by our simulator. The first dataset is described in section 6.1.1. With this dataset, we do two case studies in sections 6.2.1, which deal with the impact of offline and online computation times. The second dataset describes problems occuring in the city of Lyon, France. It described is described in section 6.1.2. With this data set, we make a case study in section 6.2.2 to see how a solver reacts to different scenarios generated from the same stochastic data. The last dataset describes problems in the city of Turin, Italy. It is presented in section 6.1.3. We then make a case study where we compare the use of bicycles and motor vehicles in section 6.2.3.

6.1 Data Sets

To perform our simulations, we used several datasets from different sources, and we rewrote them in the format of our tool that we describe in the section 3.2. Section 6.1.1 presents the coming from the well-known Solomon's benchmarks. Then we present a dataset taking place in the city of Lyon, France, in section 6.1.2. Finally, section 6.1.3 presents a data set with problems taking place in the city of Turin, Italy, and for which we have the exact coordinates of the locations involved.

6.1.1 Solomon data set

The first set of data we use comes from [8]. As explained in this article, this dataset has been adapted from Solomon's benchmarks, and so we will use this name to refer to it. As In these instances, the graph consists of a hundred nodes and each node (except the depot) is likely to emit one or more requests. The time horizon is divided into 240 time units, itself divided into three time slots plus the offline time slot. This dataset offers different instances, some with many

offline requests and few online requests, some with reverse distribution. We want to use two different types of instances. First instances with many offline requests to study the impact of offline computation time. Then, instances with a majority of online requests in order to study the impact of online computation time.

The graph involved in these instances is presented in Figure 6.1. To represent it, we used the map embedded in the graphical interface of the simulator.

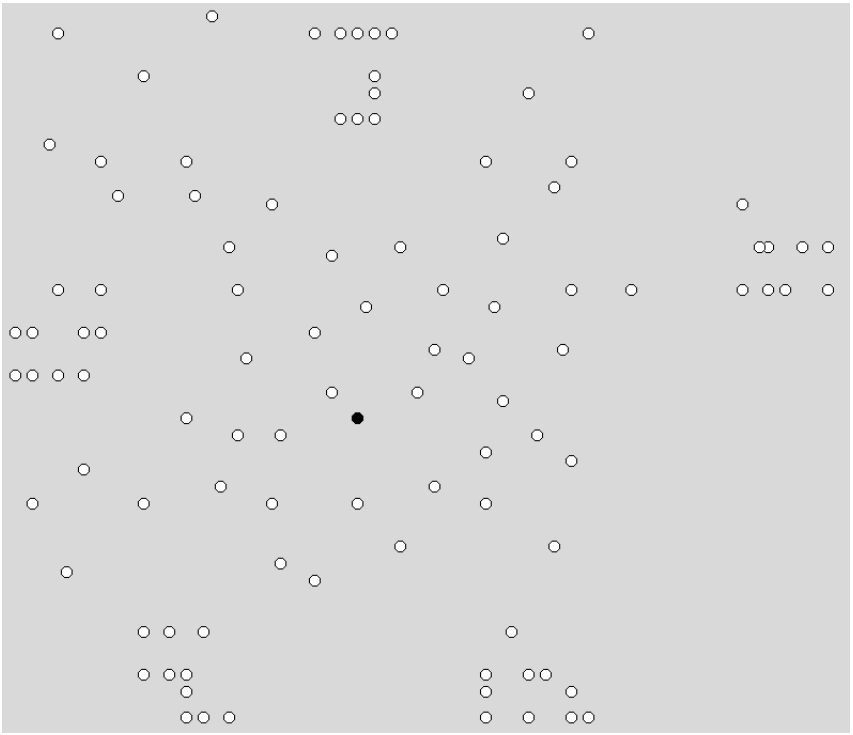


Figure 6.1: Graph of the solomon data set. The depot is represented by the black node, the customers by the white nodes.

6.1.2 Lyon data set

The second Dataset comes from [22]. This is a more realistic data set from measurements made in the city of Lyon, France. This data set describes problems occurring over an 8-hour workday. The time horizon is composed of 480 time units, each corresponding to one minute. However, the online simulation will use a much shorter time to simulate them, in order to obtain results in a reasonable time. The time units are distributed in 96 timeslot, each comprising 5 time units. Unlike the previous dataset, the emission probabilities of requests can therefore be described much more precisely. The instances of this dataset that we use present a large number of potential requests, spread over a small number of locations and each with a low probability of being issued. This is a data set where it is interesting to take into account the probabilities of emission of requests in order to try to anticipate them. We will generate several different scenarios from the same data and retrieve the solutions of the solver for each.

Unfortunately, for this data set, we don't have any coordinates for the different locations involved in the problem. In order to visualize the spatial distribution of the locations, we opened our files in the graphical interface of the simulator which can generate fictitious x,y coordinates from the travel times between the points. Figure 6.2 shows the result obtained in the embedded map of the graphical interface of the simulator.

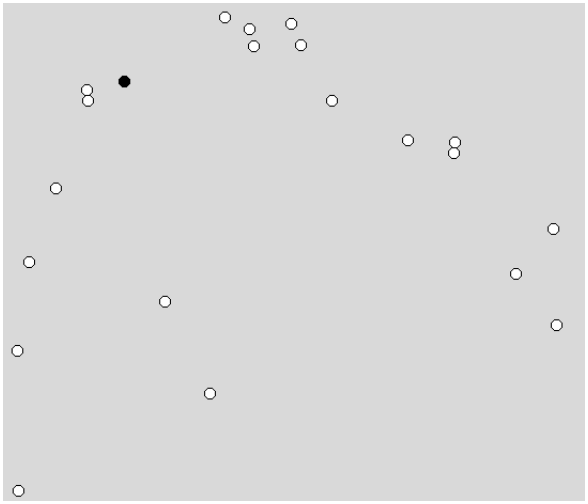


Figure 6.2: Graph of the Lyon data set. The depot is represented by the black node, the customers by the white nodes.

6.1.3 Turin data set

The third dataset we use comes from [17]. Realistic problems taking place in the city of Turin, Italy, are described. In the instance we use, the graph includes a hundred nodes. There are just over a hundred potential requests, and most of them have a high probability of being emitted. On these data, we will proceed so as to be able to study the impact of the number of vehicles available. For this data set, we have two types of vehicles available: vans and bicycles. Each type of vehicle moves differently around the city, and capacity is also different.

For this dataset, we have the coordinates of the different locations. Figure 6.3 shows the different locations involved in the problem. The black marker indicates the location of the depot and the blue markers show all the places where a request is susceptible to be emitted.

6.2 Experiments

In this section, we will present the different case studies we have carried out. With the Solomon's dataset, we perform two case study on the impact of the online and the offline computation time. These are presented in section 6.2.1 and 6.2.1. With the dataset from Lyon, we study in section 6.2.2 how our solver reacts to several scenarios generated from the same stochastic data. Finally, we present in section 6.2.3 a case study on differences resulting from the use of different vehicle types.

In order to perform all these simulations without having to restart the simulator and solver each time, we use the small python script presented in the figure 6.4. In this script, we define a function that takes as input a script file for the simulator (as described in section 4.2) as well as the number of times this script must be executed. This function first starts the solver, then after a few seconds, the simulator is launched with the script indicated. Then the function waits for both processes to end. The function executes this as many times as indicated in argument.

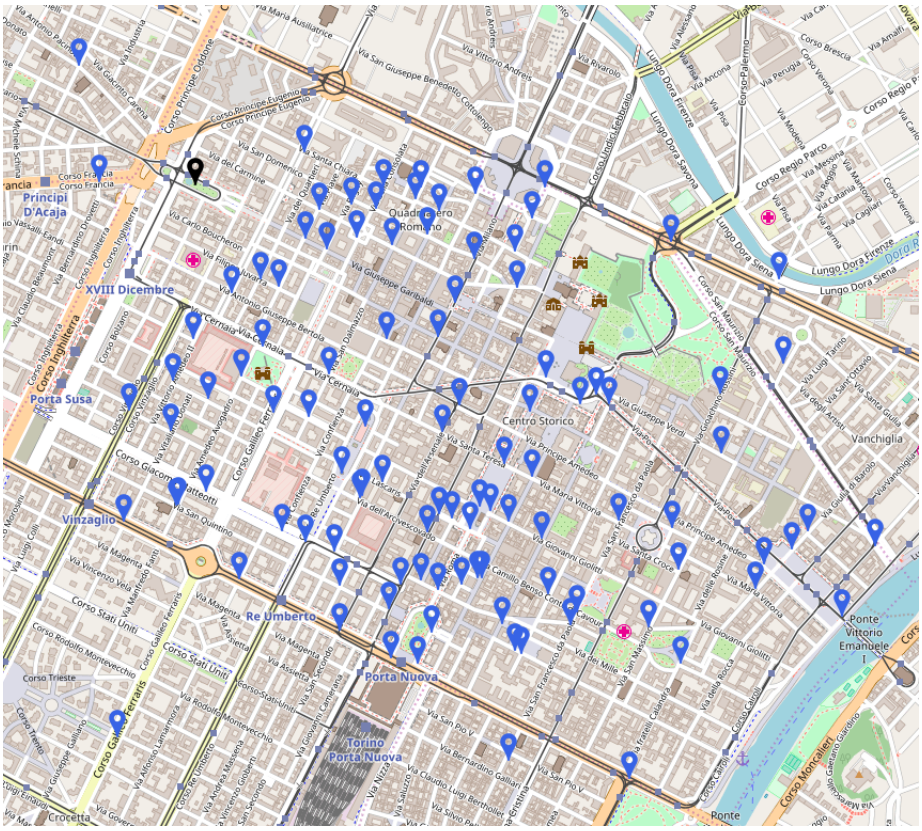


Figure 6.3: Map of the turin data set. The depot is represented by the black marker, the customers by the blue markers.

6.2.1 Solomon case study

Each script is run several times to obtain several solutions from the solver and to better estimate its effectiveness on a given problem. Here we run each script three times, but they can be run more times if necessary. The simulator scripts used can be found in the appendix B.2. In each script, at the end of the online simulation, all the solutions returned by the solver during the whole simulation are saved in a file.

We use instances coming from two different classes of the Solomon's benchmarks. The class 1 describes instances with a lot of offline requests. We will therefore study the impact of offline time variation with instances of this class. Next we will use instances of the class 6 of the Solomon's benchmarks. These instances contain less offline requests and many more online requests. With these instances, we study the impact of the variation of the computation for the online simulation.

Solomon instances of class 1

The first script executed can be found in figure B.1. As for the other scripts, it is executed three times in a row. By performing the same simulation several times, we can compare the solutions returned each time to observe the average solver performance for the same problem.

The second script executed, presented in figure B.2, is similar to the first one. The difference is that it runs an offline simulation twice as long as in the first script. Thus, the only parameter that varies between these two sets of simulations is the offline computation time. This illustrates how it is possible to evaluate the impact of one parameter by retrieving the results of two sets of

```
import subprocess
import time
import os

def runScript(scriptName, runNumber):
    '''
    scriptName : name of the script file to run
    runNumber  : number of times to run the script
    '''

    # redirect the standard output of the solver to not print it in the terminal
    FNULL = open(os.devnull, 'w')

    for i in range(runNumber):
        solverProcess = subprocess.Popen(['./solver', '-G'], cwd='./solver', stdout=FNULL)
        time.sleep(3) # wait 3 seconds before launching the simulator
        simulatorProcess = subprocess.Popen(['./main.py', '--script', scriptName], cwd = './simulator')

        solverProcess.wait()
        simulatorProcess.wait()


runScript('./script_ben_c1_0-100-rc101-1__OT30.txt', 3)
runScript('./script_ben_c1_0-100-rc101-1__OT60.txt', 3)
runScript('./script_ben_c6_0-30-70-rc101-1__CT1.txt', 3)
runScript('./script_ben_c6_0-30-70-rc101-1__CT2.txt', 3)
runScript('./script_lyon_20cw_1.txt', 3)
runScript('./script_turin_10van.txt', 3)
runScript('./script_turin_10bike.txt', 3)
```

Figure 6.4: Python script to launch the simulations and the solver

simulations where only that parameter varies.

Figure 6.5 presents the evolution of the total travel time of the solutions returned during the simulations of the two previous scripts, presented in B.1 and B.2. The x-axis shows the simulation time elapsed in time units. The y-axis shows the total length of the solution, which is computed as the sum of the travel times between the nodes of each route in a solution. The three simulations with an offline time of 30 seconds are labelled with *OT30*. The three simulations with an offline time of 60 seconds are labelled with *OT60*. The pink crosses each indicate when a request was revealed, only the x-axis matters for these crosses.

Figure 6.6 presents the evolution of the total travel time of the solutions returned during the offline part only of the simulations of the two previous scripts, presented in B.1 and B.2. This is like a zoom on the offline part Figure 6.5. The x-axis shows the simulation time elapsed in seconds. The y-axis shows the total length of the solution, which is computed as the sum of the travel times between the nodes of each route in a solution.

Figure 6.7 presents a table with statistics about the solutions returned during the six simulations performed with the script presented in B.1 and B.2. More precisely, we give details concerning the last solution returned during the offline simulation, and the last solution returned during the online simulation. The description of the contents of the table is given in the caption

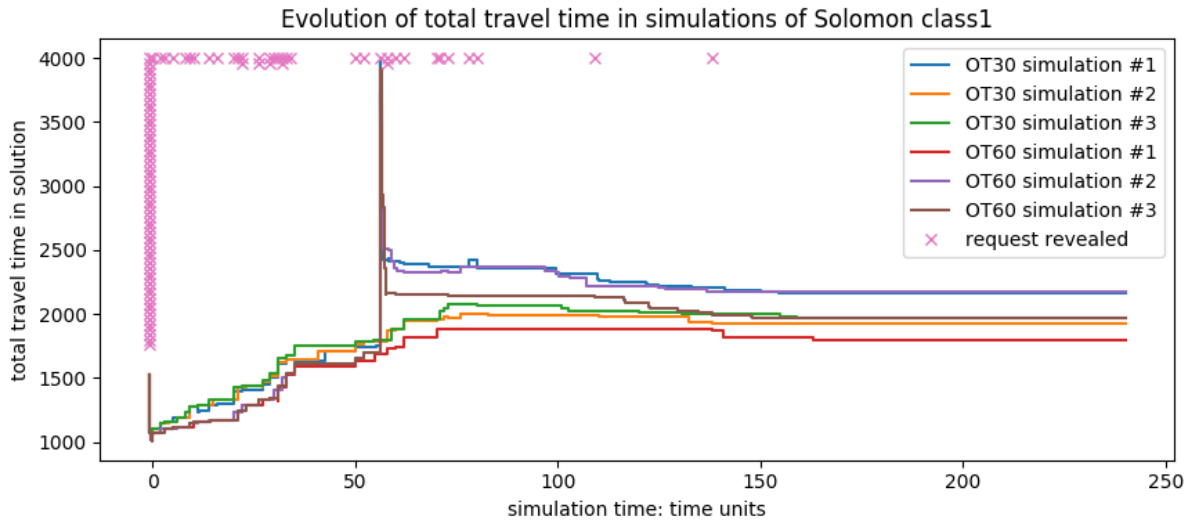


Figure 6.5: Evolution of the total travel times during the simulations with the instances of Solomon, class 1. The three lines labelled *OT30* correspond to the simulations with a 30 second offline time. The three lines labelled *OT60* correspond to the simulations with a 60 second offline time. Each cross indicate the reveal time of a request.

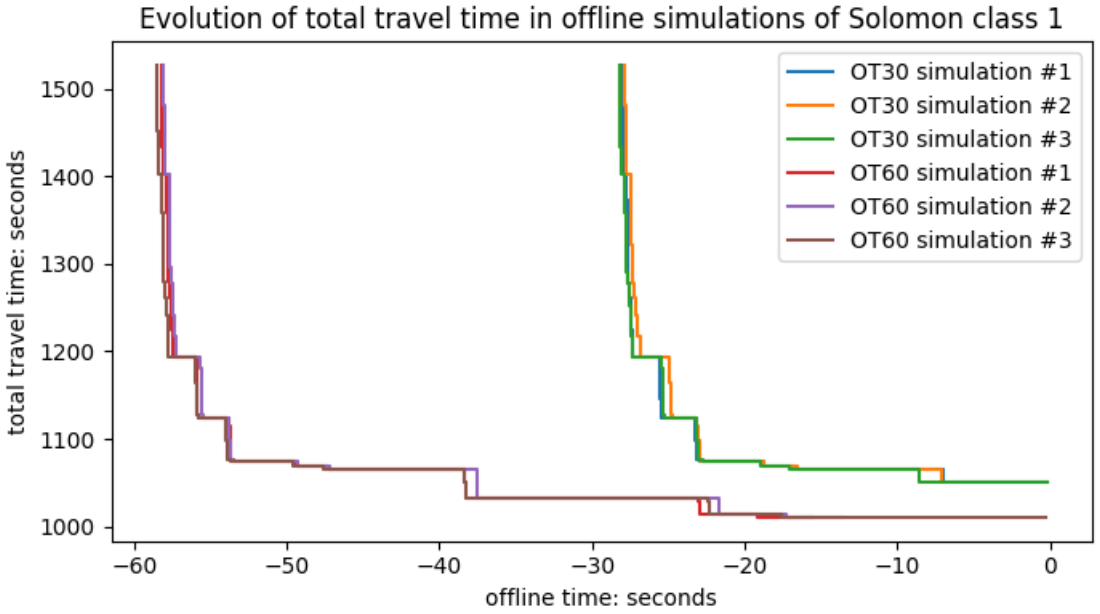


Figure 6.6: Evolution of the total travel times during the offline parts of the simulations with the instances of Solomon, class 1. The three lines labelled *OT30* correspond to the simulations with a 30 second offline time. The three lines labelled *OT60* correspond to the simulations with a 60 second offline time.

of the figure.

First of all, we see that the solver manages to serve all requests. At the end of the online simulation, it cannot be said that simulations with a 60 seconds offline computation time are significantly better or worse than the others. The total travel times of the final solutions of the 30 seconds offline time simulations are all between the worst and the best of the 60 seconds offline time simulation.

With regard only to offline simulation, we can see on figures 6.6 and 6.7 that the extra 30

	Last Offline Solution				Last Online Solution				Numbers Of Solution		
	Revealed	Served	Length	Vehicles Used	Revealed	Served	Length	Vehicles Used	offline	online	total
OT30 S1	57	57	1050,4	8	94	94	2166	15	80	163	243
OT30 S2	57	57	1050,4	8	94	94	1927	11	80	50	130
OT30 S3	57	57	1050,4	8	94	94	1968,5	13	80	61	141
OT60 S1	57	57	1009,6	8	94	94	1798,1	10	86	58	144
OT60 S2	57	57	1009,6	8	94	94	2172,8	15	86	144	230
OT60 S3	57	57	1009,6	8	94	94	1968,5	16	86	141	227

Figure 6.7: Statistics on the simulations of Solomon case study, class 1 instances. For each of the six simulations, data about the last offline solution and the last online solution is given. *Revealed* and *Served* indicate the number of requests revealed and served when the solution is returned. *Length* is the sum of the travel times of the route in the solution. *Vehicles Used* is the number of vehicles used. For each simulation are given the number of offline solutions returned, the number of online solutions returned, and the total number of solutions returned.

seconds for the offline time seems to give a very slightly better solution. Unfortunately, this advantage seems lost during the online simulation. Despite the fact that most of the queries are available during offline simulation, it seems that the solver used does not manage to take advantage of them. The offline calculation time therefore does not seem at this stage to allow the solver used to obtain better results.

Solomon instances of class 6

The next two scripts, which can be found in figure B.3 and B.4, use instances from the same data set as the first two scripts. The difference is that there are many more online requests than for the first two scripts. With these two scripts, we perform two sets of simulations where we vary only one parameter : the computation time for each unit of the online time. First we run three simulations where each time unit is simulated in one second. Then, the second script performs the same simulation but simulates the time units in two seconds.

Figure 6.8 presents the evolution of the total travel time in each solution for the six simulations performed. The three lines labelled *CT1* correspond to the simulations with a computation time of one second per time unit. The three lines labelled *CT2* correspond to the simulations with a computation time of two seconds per time unit. The pink crosses indicates when a request was emitted. Only the x-axis must be taken into account to read the crosses.

The table in Figure 6.9 presents some informations concerning the final solution of each of the six simulations as well as the numbers of solutions returned by the solver during each simulation. For each final solution, the following information is given: *Revealed* and *Served* indicate the number of requests revealed when the solution was returned and the number of request served by the solution, *Length* is the sum of the travel times in each route of the solution, *Vehicles Used* is the number of vehicles used. The number of offline, online and total solutions is also given for each simulation.

Again, we observe that all requests are served. It also seems that using a calculation time of two seconds instead of one has an impact on the quality of solutions. Figure 6.8 shows that the six curves are strongly grouped at the beginning of the simulation, but seem to separate into two groups afterwards, one with the *CT1* curves and the other with the *CT2* corubes. The table in Figure 6.9 points in the same direction. The simulations that used a computation time of two seconds use fewer vehicles and have a smaller total travel time. There are also significantly more solutions returned by the solver. Of course, the computation times used are very short. It would be interesting to see the results for longer times.

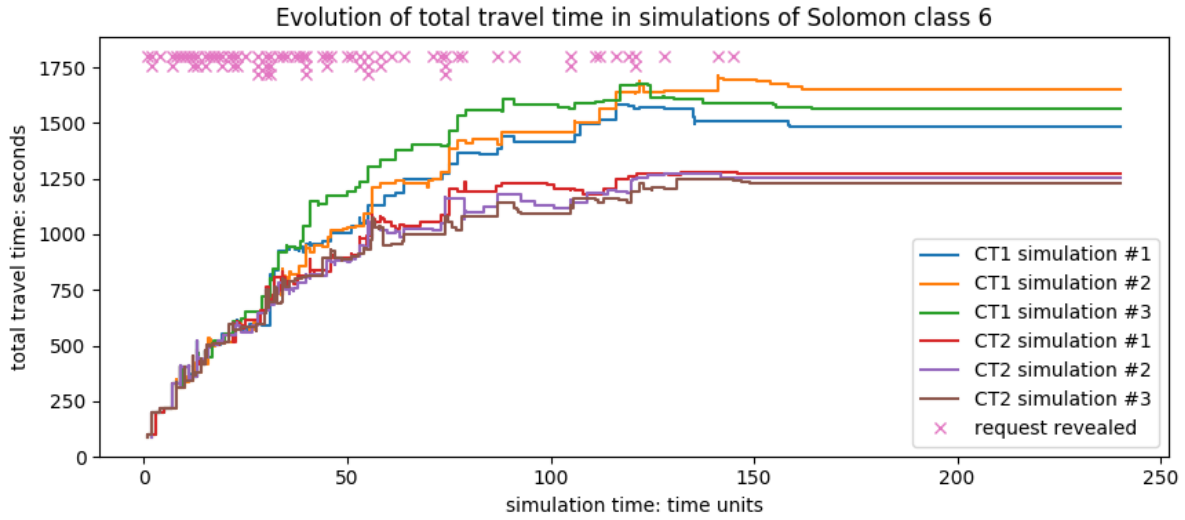


Figure 6.8: Evolution of the total travel times during the simulations with the instances of Solomon, class 6. The three lines labelled *CT1* correspond to the simulations with one second per time unit. The three lines labelled *OT60* correspond to the simulations with a two seconds per time unit. Each cross indicate the reveal time of a request.

	Last Solution				Number of solutions		
	Revealed	Served	Length	Vehicles Used	offline	online	total
CT1 S1	80	80	1485	11	2	181	183
CT1 S2	80	80	1653,9	11	2	174	176
CT1 S3	80	80	1563,4	12	2	166	168
CT2 S1	80	80	1274,9	9	2	282	284
CT2 S2	80	80	1257,4	10	2	278	280
CT2 S3	80	80	1228,9	9	2	276	278

Figure 6.9: Statistics on the simulations of Solomon case study, class 6 instances. For each of the six simulations, data about the last online solution is given. *Revealed* and *Served* indicate the number of requests revealed and served when the solution is returned. *Length* is the sum of the travel times of the route in the solution. *Vehicles Used* is the number of vehicles used. For each simulation are given the number of offline solutions returned, the number of online solutions returned, and the total number of solutions returned.

6.2.2 Lyon case study

The following script, showed in figure B.5, uses instances from the second dataset we mentioned [22]. In these instances, the time units are distributed in a much larger number of timeslot. The emission probabilities of the requests are modeled much more precisely over the time horizon. More precisely, each unit of time corresponds to a duration of one minute in the real world. Each timeslots groups five time units and corresponds to a five-minute period in real world. The emission probabilities of the requests are thus modelled on five-minutes intervals, which is quite fine as modelling.

It can therefore be interesting to see how the solver solves different scenarios generated from the same Customer file. In this case, we generate and save a new scenario each time the script is run. What differs with each execution of the script is therefore the set of requests emitted among the same set of potential requests.

The table in Figure 6.10 gives information about the last solution returned for each of the

three simulation. For each simulation, we can see the number of requests revealed and served by the last solution. In the last row, is indicated the total number of solutions returned.

	Last Solution				Number of solutions
	Revealed	Served	Length [min]	Vehicles Used	
simulation 1	29	1	36,4	1	1
simulation 2	28	2	43,7	1	2
simulation 3	27	10	116,8	2	10

Figure 6.10: Statistics on the simulations of Lyon case study. For each of the three simulations, data about the last online solution is given. *Revealed* and *Served* indicate the number of requests revealed and served when the solution is returned. *Length* is the sum of the travel times of the route in the solution. *Vehicles Used* is the number of vehicles used. The total number of solutions returned is given in the last column.

The solver served almost no request, except for the last simulation where about one third of the requests were served. This is partly because the solver does not anticipate requests and waits for them to be revealed before reacting. The requests here ask to be served very soon after being revealed. As a result, the solver is usually unable to react. We see that there are the same number of solutions returned by the solver as requests revealed in the scenario. This illustrates that the solver was only able to propose solutions on the few occasions when he could immediately react to a request that had just appeared and was not too far from one of the vehicles. The request is then very quickly served, and the solver no longer sees anything to do because it does not anticipate the future requests, and the solution is not modified.

The solver used here therefore does not seem at all suitable, and it would probably be wiser to use a solver trying to anticipate the appearance of future queries to hope for better results.

6.2.3 Turin case study

The last two scripts use instances from the last data set we mentioned [17]. Here we will assess the impact of yet another different parameter: the vehicles in the fleet. In the first script, a fleet of ten vans will be used, and the second will use a fleet of ten bikes. Vans are easier to use because they are generally faster and have a greater capacity. But bikes have the advantage of being more eco-friendly. So we would like to compare what can be done by giving drivers bicycles rather than motorized vehicles. We therefore carry out three simulations with ten vans and three with ten bicycles. The remaining parameters are identical for all simulations.

Figure 6.11 presents the evolution of the total travel time of the solution during the six simulations. On the x-axis is the time of the simulation in time units, and on the y-axis is the total travel time of the solutions in minutes. The simulations where vans were used are labelled *Vans* and the simulations where bikes were used are labelled *bikes*. The pink crosses each indicate when a request was revealed. The same requests were reaveled during each simulation. Only the x-axis matters when reading the crosses.

The table in Figure 6.12 presents some quantitative information about the solutions returned for each simulation. For each last offline solution and last online solution during a simulation, we show the following data. The columns *Revealed* and *Served* give the numbers of requests revealed to the solver and served by the solutions. The columns *Length* give the total travel times of the solutions. The column *Vehicles used* shows the number of vehicles used in each solution. On the right side of the table, the number of offline and online solutions as well as the total number of solutions returned during the simulations are given.

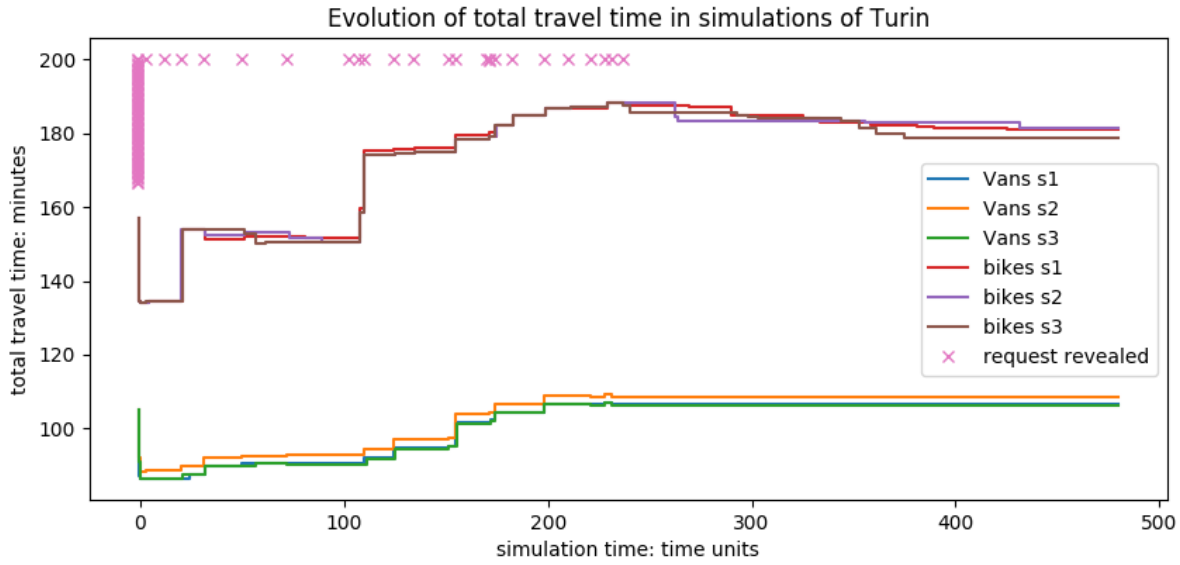


Figure 6.11: Evolution of the total travel times during the simulations with the instances of Turin. The three lines labelled *bikes* correspond to the simulations with ten bikes available. The three lines labelled *Vans* correspond to the simulations with ten vans available. Each cross indicate the reveal time of a request.

	Last Offline Solution				Last Online Solution				Number of Solutions		
	Revealed	Served	Length [min]	Vehicles Used	Revealed	Served	Length [min]	Vehicles Used	offline	online	total
vans s1	68	68	86,4	2	92	92	106,6	2	40	26	66
vans s2	68	68	88,2	2	92	92	108,6	2	37	24	61
vans s3	68	68	86,4	2	92	92	106,4	2	40	26	66
bikes s1	68	68	134,2	8	92	89	180,9	10	58	42	100
bikes s2	68	68	134,2	8	92	89	181,3	10	58	40	98
bikes s3	68	68	134,2	8	92	89	178,8	10	58	38	96

Figure 6.12: Statistics on the simulations of Turin case study. For each of the six simulations, data about the last offline solution and the last online solution is given. *Revealed* and *Served* indicate the number of requests revealed and served when the solution is returned. *Length* is the sum of the travel times of the route in the solution. *Vehicles Used* is the number of vehicles used. For each simulation are given the number of offline solutions returned, the number of online solutions returned, and the total number of solutions returned.

As expected, the total travel time for bikes is significantly higher than for vans. Moreover, only two vans are used each time, whereas all the bicycles had to be used. It also seems that the number of ten bikes is a little too low, because some requests could not be served at the end of the simulation. We also see that the number of solutions emitted during a simulation with bikes is more important. Figure 6.11 also shows that solutions with bicycles still undergo modifications long after the last modification of the solutions with vans.

6.3 Conclusion

With the script in figure 6.4, we executed a total of 21 simulations that took over 2h30 to run. We recovered all the solutions returned by the solver during a simulation, for each of the simulations performed. We illustrate how a large set of scenarios can be simulated to a solver one after the other, and the solutions recovered in order to be analyzed later and to explore how the variation of different parameters can influence the results. More precisely, we explored the effects of the variation of offline and online computation time on instances from Solomon's benchmarks. We

have seen that offline computing time seems insignificant while online computing time can play a more important role. We then tried to solve instances of the Lyon data set, and it turns out that these problems are very difficult for our solver. Finally, we compared the use of bicycles and vans on intances of the Turin dataset.

Chapter 7

Conclusion

Achievements

During this master thesis, we realized a tool allowing on the one hand to simulate the execution of a *Dynamic and Stochastic Vehicle Routing Problem with Time Windows* to a solver and to recover all the solutions returned by this one, and on the other hand to visualize the solutions and their change during the simulation and the execution of the problem. The tool, developed in `Python`, communicates with an API written in `C++` to interact with a solver.

We first explored different variants of the VRPP, starting with the simplest formulation, then adding the different constraints that lead to the formulation of the DS-VRPTW with which we subsequently work. We then presented the tool we developed during this master thesis. To do this, we first described all the features that are offered by our tool and the JSON file format that we defined in order to be able to represent the instances of the problem to simulate as well as the solutions. We decided to divide the description of a VRP into several components, each capturing an aspect of the problem and corresponding to a specific input file for the simulator. These components are the graph, which contains information on the locations of the problem, the carrier which describes the characteristics of the vehicles, the customer which presents the potential requests of the problem, and the scenario which describes the exact course of a simulation. Then, we explained precisely how to use this tool, both to perform simulations and to develop a solver by integrating our tool. After that, we detailed the architecture of the tool and its different components. We explained how these components interact between it and the technologies used. Finally, we illustrated the use of our tool by carrying out some case studies on several datasets of various origins in order to show that this tool is ready to be used and is helpful for the development and analysis of VRP solvers.

Further work

Much work can still be done to extend and improve the tool we offer. Currently, the simulator is in `Python` and communicates with a `C++` API that is used by a solver. it can therefore be interesting to re-implement the API in other languages, such as Python and Java, in order to be able to use these tools with solvers implemented in these languages. The modular architecture of the system makes it possible to consider this option without having to make any modifications to the simulator. To implement these APIs and make them communicate with the simulator, we

are obviously thinking of continuing to use google's remote procedure call framework, gRPC, which has already allowed us to easily communicate between the solver and the simulator which are implemented in different languages.

Another area for improvement that seems obvious to us would be the addition of a new type of problem handled by the simulator. We think in particular of the Dial a Ride Problem (DARP), where customers ask to be transported from one place to another. But even before that, there are a multitude of variants close to the problems already addressed. We believe that the file formats we use to describe instances of problems to be simulated can be extended relatively easily to describe new variants of VRP. The fact that the description of the problem has been divided into several components will also make it possible to extend only certain components in order to be able to describe new variants of the VRP. And using the JSON format makes it easier to use more complex files while keeping existing ones valid.

Currently, simulations performed by the simulator always start with a static execution, where a set of requests is known at the start, and then a dynamic execution, where requests are dynamically revealed during the execution. The simulator and solver can be paused synchronously during both parts. And it is possible to save the result of the static simulation to start the dynamic simulation later. In the future, it may be interesting to be able to suspend a simulation and the simulator at any time during the execution of the static or dynamic simulation, restart the simulator later and resume the execution of a simulation at the moment when it was interrupted. Also, it would be useful if the simulator could launch the desired solver itself and the user no longer had to launch it separately next to it.

In a longer-term vision, we can also consider allowing the simulator to perform several simulations simultaneously for different solvers. Consideration could also be given to allowing the simulator to work with solvers running on remote computers. Once again, the choice we made to use the google framework, gRPC, to implement the communication between the solver and the simulator seems judicious in this perspective. The purpose of this framework being to make easy the communication between processes running on remote machines.

Bibliography

[1] grpc.io. `https://grpc.io/`.

[2] Leaflet. `https://leafletjs.com/`.

[3] Protocol buffers. `https://developers.google.com/protocol-buffers/`.

[4] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.91 edition, May 2015.

[5] Roberto Baldacci, Maria Battarra, and Daniele Vigo. Routing a heterogeneous fleet of vehicles. In *The vehicle routing problem: latest advances and new challenges*, pages 3–27. Springer, 2008.

[6] Roberto Baldacci, Aristide Mingozzi, and Roberto Roberti. Recent exact algorithms for solving the vehicle routing problem under capacity and time window constraints. *European Journal of Operational Research*, 218(1):1–6, 2012.

[7] Russell Bent and Pascal Van Hentenryck. Waiting and relocation strategies in online stochastic vehicle routing. *IJCAI International Joint Conference on Artificial Intelligence*, pages 1816–1821, 2007.

[8] Russell W. Bent and Pascal Van Hentenryck. Scenario-Based Planning for Partially Dynamic Vehicle Routing with Stochastic Customers. *Operations Research*, 52(6):977–987, 2004.

[9] Jürgen Branke, Martin Middendorf, Guntram Noeth, and Maged Dessouky. Waiting Strategies for Dynamic Vehicle Routing. *Transportation Science*, 39(3):298–312, 2005.

[10] T. Bray. The javascript object notation (json) data interchange format. STD 90, RFC Editor, December 2017.

[11] Jean-Francois Cordeau and Québec) Groupe d'études et de recherche en analyse des décisions (Montréal. *The VRP with time windows*. Montréal: Groupe d'études et de recherche en analyse des décisions, 2000.

[12] G. B. Dantzig and J. H. Ramser. The Truck Dispatching Problem. *Management Science*, 6(1):80–91, 1959.

[13] D T U Management Engineering and D T U Transport Nov. Stochastic vehicle routing problems. 7(1992), 2014.

[14] William Ho, George TS Ho, Ping Ji, and Henry CW Lau. A hybrid genetic algorithm for the multi-depot vehicle routing problem. *Engineering Applications of Artificial Intelligence*, 21(4):548–557, 2008.

[15] Adam N Letchford, Jens Lysgaard, and Richard W Eglese. A branch-and-cut algorithm for the capacitated open vehicle routing problem. *Journal of the Operational Research Society*, 58(12):1642–1651, 2007.

[16] Penélope Aguiar Melgarejo, Philippe Laborie, and Christine Solnon. A time-dependent no-overlap constraint: Application to urban delivery problems. In *International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems*, pages 1–17. Springer, 2015.

[17] Guido Perboli. Simulation–optimisation framework for city logistics: an application on multimodal last-mile delivery. *IET Intelligent Transport Systems*, 12:262–269(7), May 2018.

[18] Victor Pillac, Michel Gendreau, Christelle Guéret, and Andrés L. Medaglia. A review of dynamic vehicle routing problems. *European Journal of Operational Research*, 225(1):1–11, 2013.

[19] Harilaos N. Psaraftis, Min Wen, and Christos A. Kontovas. Dynamic vehicle routing problems: Three decades and counting. *Networks*, 67(1):3–31, 2016.

[20] Michael Saint-Guillain, Yves Deville, and Christine Solnon. A Multistage Stochastic Programming Approach to the Dynamic and Stochastic VRPTW. *12th International Conference on Integration of AI and OR Techniques in Constraint Programming (CPAIOR 2015)*, pages 357–374, 2015.

[21] Michael Saint-guillain, Christine Solnon, and Yves Deville. Applications of Evolutionary Computation. 8602:110–127, 2014.

[22] Michael Saint-Guillain, Christine Solnon, and Yves Deville. The static and stochastic vrp with time windows and both random customers and reveal times. In *European Conference on the Applications of Evolutionary Computation*, pages 110–127. Springer, 2017.

[23] Alexander Shapiro, Darinka Dentcheva, and Andrzej Ruszczyński. *Lectures on stochastic programming: modeling and theory*. SIAM, 2009.

[24] Marius M Solomon. Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations research*, 35(2):254–265, 1987.

[25] Éric D Taillard. A heuristic column generation method for the heterogeneous fleet vrp. *RAIRO-Operations Research*, 33(1):1–14, 1999.

[26] Christos D Tarantilis, George Ioannou, Chris T Kiranoudis, and Gregory P Prastacos. Solving the open vehicle routeing problem via a single parameter metaheuristic algorithm. *Journal of the Operational Research Society*, 56(5):588–596, 2005.

[27] Tai-Hsi Wu, Chinyao Low, and Jiunn-Wei Bai. Heuristic solutions to multi-depot location-routing problems. *Computers & Operations Research*, 29(10):1393–1415, 2002.

Appendix A

Graphical User Interface Screenshot

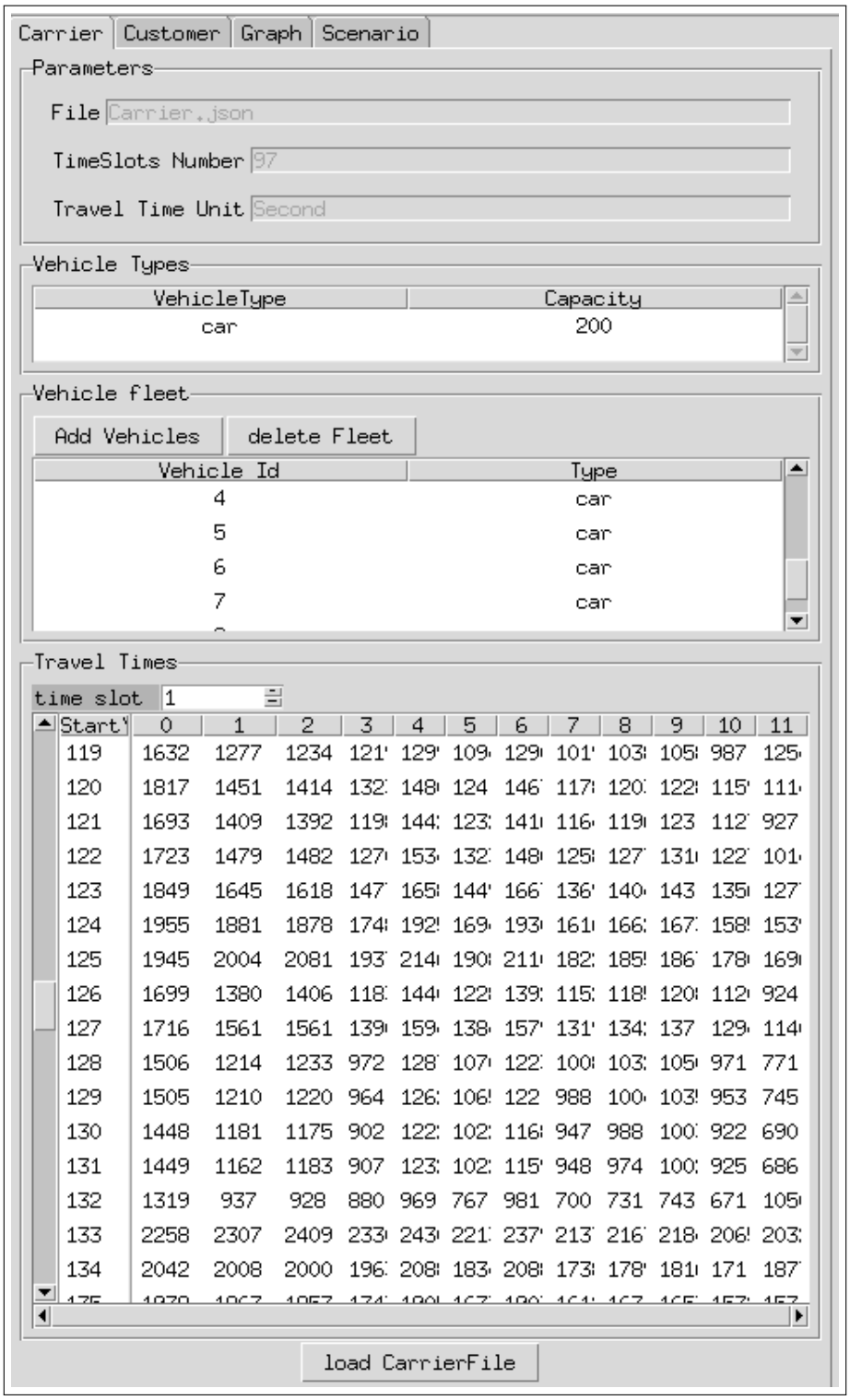


Figure A.1: The Carrier Tab

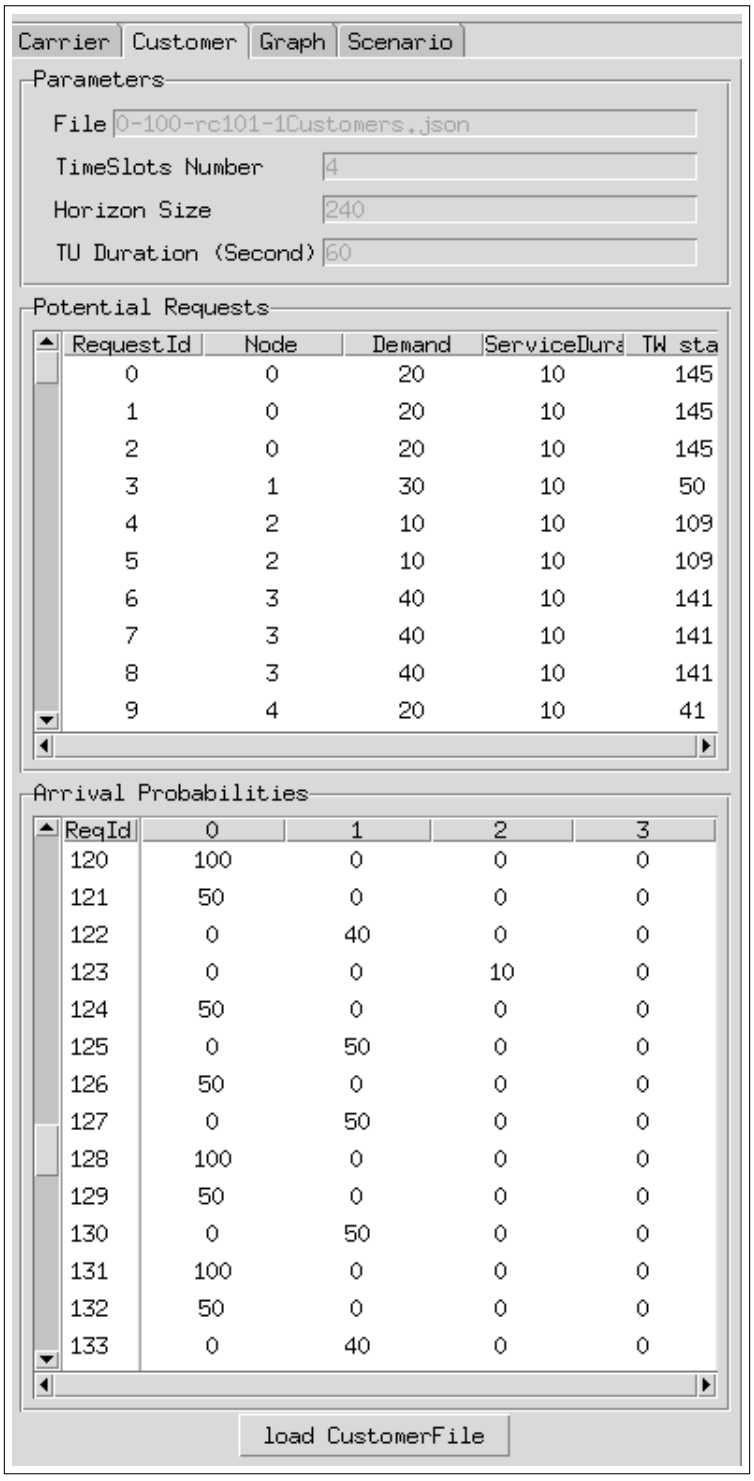


Figure A.2: The Customer Tab

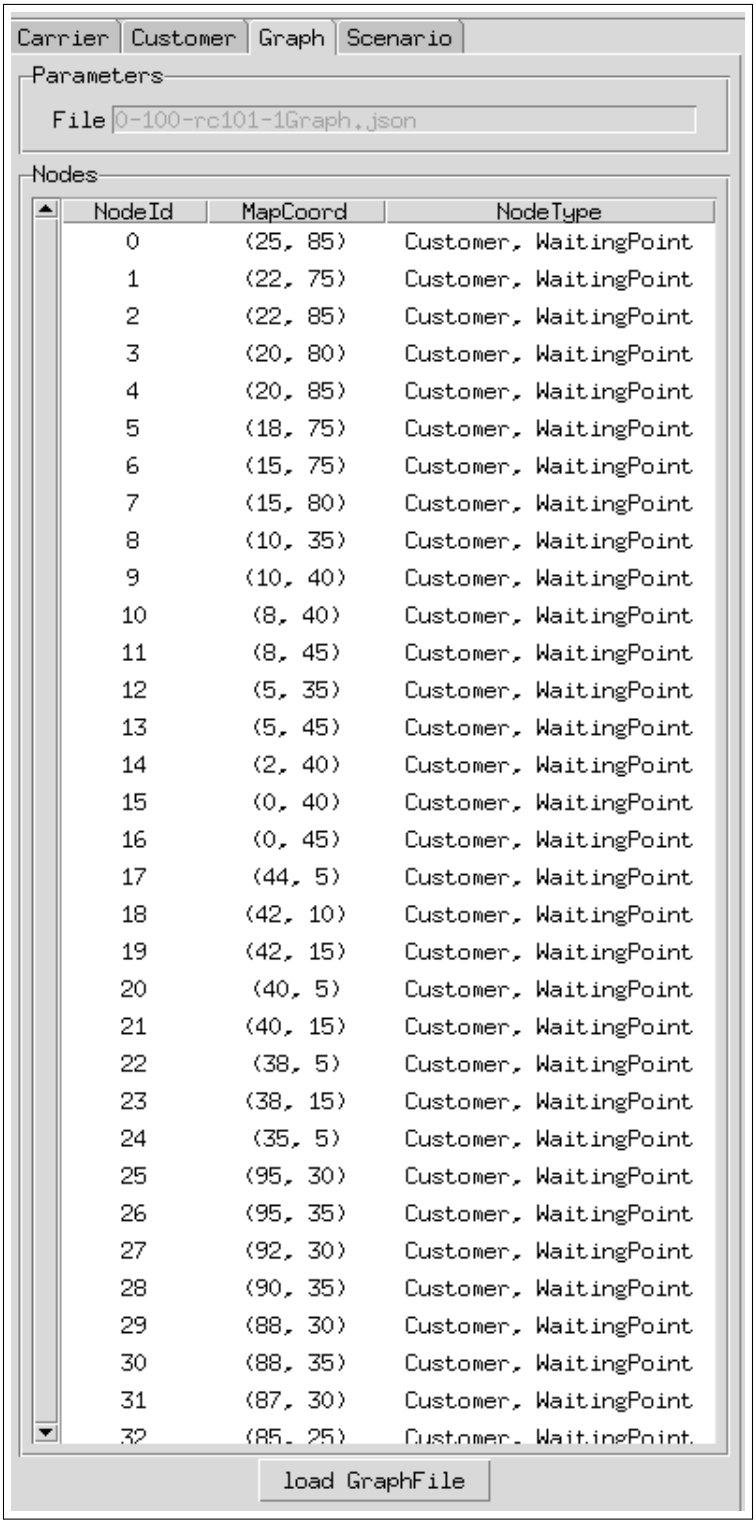


Figure A.3: The Graph Tab

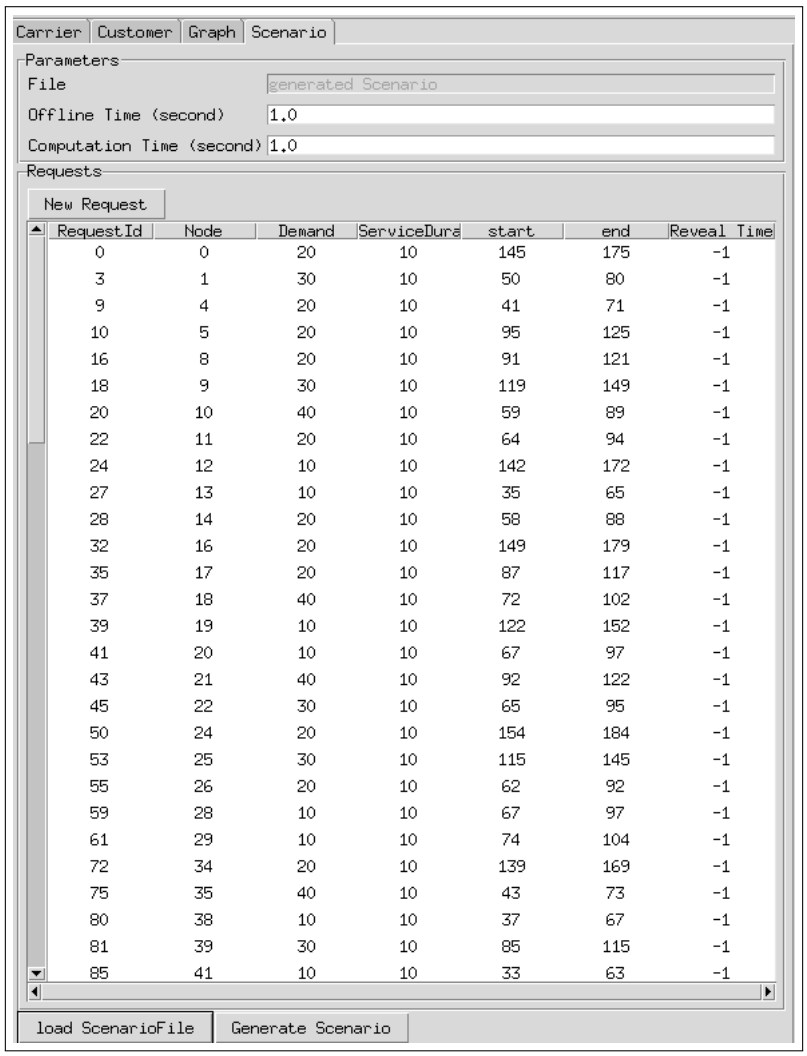
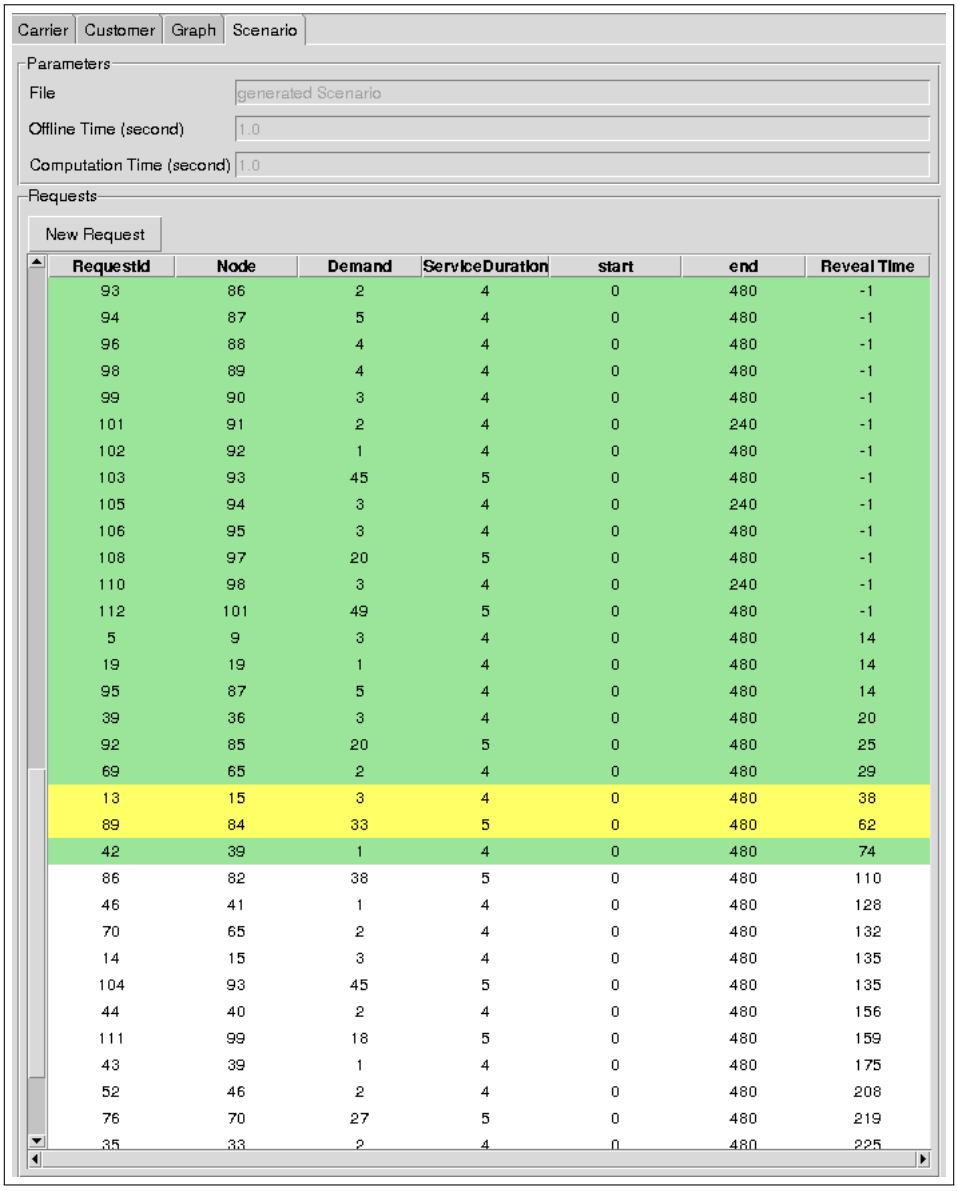


Figure A.4: The Scenario Tab, in the PreSimulation State.



RequestId	Node	Demand	ServiceDuration	start	end	Reveal Time
93	86	2	4	0	480	-1
94	87	5	4	0	480	-1
96	88	4	4	0	480	-1
98	89	4	4	0	480	-1
99	90	3	4	0	480	-1
101	91	2	4	0	240	-1
102	92	1	4	0	480	-1
103	93	45	5	0	480	-1
105	94	3	4	0	240	-1
106	95	3	4	0	480	-1
108	97	20	5	0	480	-1
110	98	3	4	0	240	-1
112	101	49	5	0	480	-1
5	9	3	4	0	480	14
19	19	1	4	0	480	14
95	87	5	4	0	480	14
39	36	3	4	0	480	20
92	85	20	5	0	480	25
69	65	2	4	0	480	29
13	15	3	4	0	480	38
89	84	33	5	0	480	62
42	39	1	4	0	480	74
86	82	38	5	0	480	110
46	41	1	4	0	480	128
70	65	2	4	0	480	132
14	15	3	4	0	480	135
104	93	45	5	0	480	135
44	40	2	4	0	480	156
111	99	18	5	0	480	159
43	39	1	4	0	480	175
52	46	2	4	0	480	208
76	70	27	5	0	480	219
35	33	2	4	0	480	225

Figure A.5: The Scenario Tab during the online simulation. The green requests are served in the solution. The yellow requests are revealed but not yet served in a solution. The white requests are not yet revealed.

Figure A.6: The simulation Frame. Next to each vehicle are presented the different steps of the route assigned to the vehicle. The light green rectangles are the future steps in the route while the dark green are the steps already executed by the vehicle.

Appendix B

Source Code Parts

B.1 Skeleton code to use the API

```
#include "vrpAPI.h"

int main(int argc, char** argv) {

  /*
   *   SECTION 1 : RETRIEVING INITIAL DATA BEFORE OFFLINE SIMULATION
   */

  // instantiate the api object, and start listening the simulator
  vrpApi myVRPAPI;
  myVRPAPI.startListeningSimulator(false);

  while(not myVRPAPI.testConnection()){
    // wait for simulator to be connected
  }

  // retrieve the input data sent by the simulator
  json myVRPdata;
  json mySolution;
  bool dataReceived = false;
  while(not dataReceived){
    if(myVRPAPI.vrpDataChanged()){

      myVRPdata = myVRPAPI.getVrpData();

      if(myVRPdata.find("Carrier")          != myVRPdata.end() and
         myVRPdata.find("Customers")        != myVRPdata.end() and
         myVRPdata.find("Graph")            != myVRPdata.end() and
         myVRPdata.find("OfflineTime")      != myVRPdata.end() and
         myVRPdata.find("AdditionalFileKey") != myVRPdata.end()
         ){
          dataReceived = true;
      }
    }
  }

  // how to access to data after loading them
  json myCarrier        = myVRPdata["Carrier"];
  json myCustomer       = myVRPdata["Customers"];
  json myGraph          = myVRPdata["Graph"];
  json myAdditionalFile = myVRPdata["AdditionalFileKey"]
```

```
42  double OfflineTime    = myVRPdata["OfflineTime"];
43
44  /*
45   * SECTION 2: THE OFFLINE SIMULATION
46   */
47
48  // wait for the simulator to start the offline simulation
49  myVRPAPI.readyForOffline();
50
51  while(myVRPAPI.isNewRequests()){
52    json newRequest = myVRPAPI.popNewRequest();
53    // do something with the new request
54  }
55
56  // while the simulation is in the offline simulation state,
57  // compute solutions for the static problem.
58  while(myVRPAPI.getCurrentSimulationState() == "offlineComputation"){
59    myVRPAPI.pauseIfSimulatorAsks();
60
61    if (myVRPAPI.isNewCurrentSolution()){
62          mySolution = getNewCurrentSolution();
63    }
64
65    currentTu =  myVRPAPI.getCurrentTimeUnit();
66    double timeRemaining = -currentTu/OfflineTime;
67
68    // perform computation, find a solution
69
70    myVRPAPI.updateBestSolution(mySolution);
71  }
72
73  /*
74   *   SECTION 3 : OFFLINE SIMULATION IS FINISH
75   *               RETRIEVE INFORMATION BEFORE ONLINE SIMULATION
76   */
77
78
79  // How to retrieve the computation time
80  dataReceived = false;
81  while(not dataReceived){
82    if(myVRPAPI.vrpDataChanged()){
83
84      myVRPdata = myVRPAPI.getVrpData();
85
86      if(myVRPdata.find("ComputationTime") != myVRPdata.end()){
87          dataReceived = true;
88      }
89    }
90  }
91  double ComputationTime = myVRPdata["ComputationTime"];
92
93  /*
94   *   SECTION 4 : ONLINE SIMULATION
95   */
96
97  // wait for the simulator to start the online simulation
98  myVRPAPI.readyForOnline();
99
100
101 // while the simulation is in the online simulation state,
102 // try to find solutions to the online problem
103 while(myVRPAPI.getCurrentSimulationState() == "onlineComputation"){
104   myVRPAPI.pauseIfSimulatorAsks();
```

```
105
106     if (myVRPAPI.isNewCurrentSolution()){
107           mySolution = getNewCurrentSolution();
108     }
109
110     // retrieve new requests
111     while(myVRPAPI.isNewRequests()){
112       json nR = myVRPAPI.popNewRequest();
113     }
114
115     // get the current time unit.
116     cu = myVRPAPI.getCurrentTimeUnit();
117
118     // perform computation
119   }
120 }
```

B.2 Simulator scripts used in the Proof of Concept

```
1  loadScenario instance_benth_pvh_tfe/class1/0-100-rc101-1Scenario.json
2  loadCarrier  instance_benth_pvh_tfe/class1/0-100-rc101-1Carrier.json
3  loadCustomer instance_benth_pvh_tfe/class1/0-100-rc101-1Customers.json
4  loadGraph    instance_benth_pvh_tfe/class1/0-100-rc101-1Graph.json
5  setComputationTime 1.0
6  setOfflineTime     30
7  sendAll
8  startOfflineSimulation
9  startOnlineSimulation
10 saveSolutions solutionsDir/solution_ben_class1_0-100-rc101-1_OT30.txt
11 stopSimulation
12 close
```

Figure B.1: *script_ben_c1_0-100-rc101-1___OT30.txt* : A simulator script to perform a simulation based on the instances from [8]

```
1  loadScenario instance_benth_pvh_tfe/class1/0-100-rc101-1Scenario.json
2  loadCarrier  instance_benth_pvh_tfe/class1/0-100-rc101-1Carrier.json
3  loadCustomer instance_benth_pvh_tfe/class1/0-100-rc101-1Customers.json
4  loadGraph    instance_benth_pvh_tfe/class1/0-100-rc101-1Graph.json
5  setComputationTime 1.0
6  setOfflineTime     60
7  sendAll
8  startOfflineSimulation
9  startOnlineSimulation
10 saveSolutions solutionsDir/solution_ben_class1_0-100-rc101-1_OT60.txt
11 stopSimulation
12 close
```

Figure B.2: *script_ben_c1_0-100-rc101-1___OT60.txt* : A simulator script to perform a simulation based on the instances from [8]

```
loadScenario instance_benth_pvh_tfe/class6/0-30-70-rc101-1Scenario.json
loadCarrier  instance_benth_pvh_tfe/class6/0-30-70-rc101-1Carrier.json
loadCustomer instance_benth_pvh_tfe/class6/0-30-70-rc101-1Customers.json
loadGraph    instance_benth_pvh_tfe/class6/0-30-70-rc101-1Graph.json
setComputationTime 1.0
setOfflineTime 15.0
sendAll
startOfflineSimulation
startOnlineSimulation
saveSolutions solutionsDir/solution_ben_class6_0-30-70-rc101-1_CT1.txt
stopSimulation
close
```

Figure B.3: *script_ben_c6_0-30-70-rc101-1___CT1.txt* : A simulator script to perform a simulation based on the instances from [8]

```
loadScenario instance_benth_pvh_tfe/class6/0-30-70-rc101-1Scenario.json
loadCarrier  instance_benth_pvh_tfe/class6/0-30-70-rc101-1Carrier.json
loadCustomer instance_benth_pvh_tfe/class6/0-30-70-rc101-1Customers.json
loadGraph    instance_benth_pvh_tfe/class6/0-30-70-rc101-1Graph.json
setComputationTime 2.0
setOfflineTime 15.0
sendAll
startOfflineSimulation
startOnlineSimulation
saveSolutions solutionsDir/solution_ben_class6_0-30-70-rc101-1_CT2.txt
stopSimulation
close
```

Figure B.4: *script_ben_c6_0-30-70-rc101-1___CT2.txt* : A simulator script to perform a simulation based on the instances from [8]

```
loadCarrier   realdata_optimod_lyon/Carrier.json
loadCustomer  realdata_optimod_lyon/20cw/20cw_1Customers.json
loadGraph     realdata_optimod_lyon/20cw/20cw_1Graph.json
generateScenario --ot 20 --ct 1
saveScenario  realdata_optimod_lyon/20cw/20cw_1Scenario.json
sendAll
startOfflineSimulation
startOnlineSimulation
saveSolutions solutionsDir/solutions_lyon_20cw-1.txt
stopSimulation
close
```

Figure B.5: *script_lyon_20cw_1.txt* : A simulator script to perform a simulation based on the instance from [22]

```
loadCarrier   rizzo_stguillain_ds-vrptw/OC-100-70-25%-1/carrier.json
loadCustomer  rizzo_stguillain_ds-vrptw/OC-100-70-25%-1/12-18-0-0/
   lockers-no/Customer.json
loadGraph     rizzo_stguillain_ds-vrptw/OC-100-70-25%-1/Graph.json
loadScenario  rizzo_stguillain_ds-vrptw/OC-100-70-25%-1/12-18-0-0/
   lockers-no/scenario.json
deleteVehicles
addVehicles         10 van
setComputationTime 0.5
setOfflineTime     20.0
sendAll
startOfflineSimulation
startOnlineSimulation
saveSolutions solutionsDir/solution_turin_10van.txt
stopSimulation
close
```

Figure B.6: *script_turin_10van.txt* : A simulator script to perform a simulation based on the instance from [16]

```
loadCarrier   rizzo_stguillain_ds-vrptw/OC-100-70-25%-1/carrier.json
loadCustomer  rizzo_stguillain_ds-vrptw/OC-100-70-25%-1/12-18-0-0/
   lockers-no/Customer.json
loadGraph     rizzo_stguillain_ds-vrptw/OC-100-70-25%-1/Graph.json
loadScenario  rizzo_stguillain_ds-vrptw/OC-100-70-25%-1/12-18-0-0/
   lockers-no/scenario.json
deleteVehicles
addVehicles         10 bike
setComputationTime 0.5
setOfflineTime     20.0
sendAll
startOfflineSimulation
startOnlineSimulation
saveSolutions solutionsDir/solution_turin_15van.txt
stopSimulation
close
```

Figure B.7: *script_turin_15van.txt* : A simulator script to perform a simulation based on the instance from [16]

Appendix C

Quantitative data about the code

The table below presents some quantitative information about the code of the `Python` simulator and the `C++` API for the solver. We do not count the files generated by the gRPC [1] framework we used. All files are available here : `https://github.com/SimonCandaele/simulator_VRP`.

Language	Number of file	Total sizes of files	Total number of lines
Python	18	271 Ko	6426
C++	2	42 Ko	1375
HTML	1	11,6 Ko	414

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve **www.uclouvain.be/epl**