UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

epl

Dynamic and Stochastic Vehicle Routing Problem with Time Windows

Comparison between state-of-the-art methods

Dissertation presented by
Niels SAYEZ

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

Supervisor(s)
Yves DEVILLE, Michael SAINT-GUILLAIN

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

Academic year 2016-2017

Abstract

Vehicle routing is a common challenge in our everyday life that aims at optimizing the displacement of a given fleet of vehicles, in order to serve a given set of requests arriving from customers. The scientific community proposed numerous models for this problem as well as approaches to provide solutions respecting the needs and desires of the users. The methods proposed in the literature evolved over time such that the most recent techniques use representations that are different from older ones. The goal of this master thesis is to create a framework using a unique representation for some state-of-the-art techniques proposed by Russel Bent and Pascal Van Hentenryck [BVH04d, BVH04a, BVH07] tackling the dynamic vehicle routing problem with time windows. This variant of the vehicle routing problem has a set of requests that grows over time as customers are allowed to request service during the journey of the fleet, which brings the difficulty to determine if arriving requests may be included in the existing solution(s) during the operations. This variant also contains limitations in the time each request is allowed to be serviced because each of them is associated with a service time window. Our framework implements two major solution techniques that try to maximize the number of served customers for this variant of the vehicle routing problem: the Multiple Plan Approach and its extension using stochastic information, the Multiple Scenario Approach [BVH04d]. In the framework we developed, we give to both of these techniques the possibility to use the *Consensus* or the *Regrets* approaches to take decisions during the *dynamic* phase of the problem [BVH04a]. We also give the possibility to *Multiple Scenario Approach* to apply two improving strategies that are expected to provide better results by increasing the number of serviced requests per instance. These strategies, named *Waiting* and *Relocation* strategies, aim at increasing the number of served customers further by taking advantage of stochastic information some more[BVH07]. To describe the design we developed, we start from the original algorithms given in the literature and then expose the modifications we bring to them to reach our final product. Finally, we validate our framework by reproducing experiments done in the literature to verify the consistency of our results with those found in previous works, and point out parts of our framework that could be improved in further work.

Acknowledgments

After a long period of intensive work, I take the opportunity to thank everyone that helped and supported me for this master thesis. I would like to first express my gratitude to my supervisor Professor Yves Deville and to the assistant Michael Saint-Guillain without whom this work could not be achieved. I would like to thank them for their supervision during this project, for the large amount of time they spent rereading parts of this work, and for the very helpful comments and advices they provided for the redaction of this dissertation.

Special and most sincere thanks are due to Michael Saint-Guillain whose time I certainly abused, for supporting and guiding me to through this project as this work could not be achieved without his availability and engagement.

Finally, I would like to address warmest words of thanks to the other persons who always believed in me, relatives and friends, for their everyday support that led me to success.

Contents

List of Figures

List of Tables

Nomenclature

Abbreviations

CVRP	Capacitated Vehicle Routing Problem, page 3
D-VRPTW	Dynamic Vehicle Routing Problem with Time Windows, page 1
DOD	Degree of Dynamism, page 45
DS-VRP	Dynamic and Stochastic Vehicle Routing Problem, page 12
DS-VRPTW	Dynamic and Stochastic Vehicle Routing Problem with Time Windows, page 1
MPA	Multiple Plan Approach, page 1
MSA	Multiple Scenario Approach, page 1
S-VRP	Stochastic Vehicle Routing Problem, page 11
VRP	Vehicle Routing Problem, page 1
VRPTW	Vehicle Routing Problem with Time Windows, page 1

Functions

$\sigma(r)$	Time a vehicle departs request r, page 8
$cust(sol)/cust(\rho)$	Set of customers visited by the solution sol / route ρ, page 4
$d(i,j)$	Distance between vertices i and $j \in S$ in graph G, page 4
$d(sol)$	Total travel distance of solution sol. Equal to the sum of distances of arcs used in sol, page 4
$e(r), l(r)$	Earliest/latest arrival times (e/l) of request r, page 7
$f(r)$	Score of request r for the approach chosen to implement ChooseRequest, page 17
$p(r), q(r)$	Service time (p) and demand of request r, page 7
$pred(r,\rho)$ / r^-	Predecessor of request r in route ρ, page 7
$q(i)$	Demand of vertex i in graph G, page 4
$req(\gamma)$	Set of request composing roting plan γ, page 8
$req(\rho)$	Set of request composing route ρ, page 8
$score(r,v,\Gamma)$	Number of plans in Γ where request r is the next request on route v, page 43

$succ(r, \rho)$ / r^+	Successor of request r in route ρ, page 7
$w(\gamma)$	Value of objective function for plan γ, page 9
$w(r)$	Value/Gain of servicing request r, page 16

Symbols

(γ^*, σ^*)	Routing plan and Departure times composing the best solution in Γ, page 22
Γ	Pool of plans maintained by MPA and MSA, page 13
γ	Routing plan, composed of a tuple of routes: $\gamma =< \rho_1, ..., \rho_m >$, page 8
$\gamma^-/sol^-, \gamma^+/sol^+$	Elapsed/Remaining part of the plan γ,solution sol, page 11
$\mathcal{O}$	Offline solver that generates solutions $sol = (\gamma, \sigma)$, page 11
$\perp$	Waiting action added to possible actions a vehicle may do when becoming idle, page 27
ρ	Route in a solution of a VRP, page 4
σ	Departure times of a solution, composed of a tuple of departure times of each route: $\sigma =< \sigma_1(\rho_1), ..., \sigma_m(\rho_m) >$, page 8
A	Set of arcs in graph G, page 3
A_t	Set of requests that were accepted before or at time t, page 11
C	Set of customer locations in a VRP, page 6
G	Graph on which the VRP is applied: $G = (S, A)$, page 3
H_e, H_l	Beginning and end of the horizon, page 7
m	Number of vehicles in a VRP, page 4
M_t	Consensus matrix, page 32
$n_{\mathcal{O}}$	Number of optimizations generated per time step, page 16
o	Vertex representing the depot in graph G: $o \in S$, page 4
Q	Load capacity of a vehicle, page 4
R	Set of requests in a VRPTW, page 7
R_t	Requests known at time t, page 10
S	Set of vertices in graph G, page 3
v	A vehicle, page 27
r,r^*	An actual request, a sampled request, page 29

Chapter 1

Introduction

Transportation and delivery are core components of the everyday life of our society, and with the growing need to optimize routing techniques, research in this field developed in the scientific community. This research domain known as the *Vehicle Routing Problem* (VRP) first aimed at producing routing plans that visit a given set of customers using a given set of vehicles such that an objective function is optimized (often, the total travel distance is minimized).

This research field was very active in the last fifty years to model extensions of this problem that handle complicating real-world constraints, and provide methods that solve them efficiently whilst caring the needs and desires of the users. Some models that come directly from our highly dynamic society use limitations on delivery windows (*Vehicle Routing Problem with Time Windows, VRPTW*). Other models have the possibility to update the set of requests to include in the solutions during computation, to allow late requests and increase the benefits of the delivery company. The sub-domain of VRP including these two possibilities, named *Dynamic Vehicle Routing Problem with Time Windows* (D-VRPTW) was addressed by Russel Bent and Pascal Van Hentenryck (we will refer to them as *Bent et al.* from now) in the early 2000's. They designed several techniques and approaches to solve D-VRPTW with the objective to maximize the number of served customers. One of these techniques is the *Multiple Plan Approach* (MPA) using methods named *consensus* or *regrets*. This technique maintains a pool of solutions during the operations that is updated regularly, either by filtering out some solutions, or by adding newly generated ones. These plans in the pool are used together by the *consensus* approach or the *regrets* approach to take decisions about requests arriving during the working hours. Later, *Bent et al.* generalized MPA to an other variant of VRP, which is the *Dynamic and Stochastic Vehicle Routing Problem with Time Windows* (DS-VRPTW), with the *Multiple Scenario Approach* (MSA). MSA extends MPA by allowing usage of prior knowledge that provides stochastic information about the customers. This stochastic information is then used in decision processes such that the total number of serviced customers is expected to increase. Finally, in 2007, *Bent et al.* also designed two enhancements to MSA named *waiting strategy* and *relocation strategy* to improve the results further.

In this master thesis we design and implement a complete solution framework to both D-VRPTW and DS-VRPTW. This framework adapts the different publications of *Bent et al.* in such a way that combinations of techniques (MPA or MSA), with approaches (consensus or regrets) and improving strategies (waiting or relocation), if any, can be done easily and the impact of these combinations on the number of served customers can be measured. Beyond implementing existing techniques so that they are compatible, the framework we propose brings additional features such as the possibility for a customer to request service more than once, whereas the work of *Bent et al.* considers each customer once. We also added the possibility to parallelize one important part of the computation, which is the generation of solutions. This

allows a user having a system with a high number of CPU cores to easily configure the application in such a way that the tradeoff between the quality of a plan (i.e. the time it is optimized) and the number of plans generated for a given time interval is less restrictive.

This dissertation is organized as follows. Chapter 2 first defines formally the class of problems we tackle, the *Dynamic and Stochastic Vehicle Routing Problem with Time Windows*. To do so, we start from the generic definition of VRP and we progressively introduce extensions until DS-VRPTW is fully described. Then, Chapter 3 introduces in sections 3.1 and 3.2 some algorithms proposed by *Bent et al.* to solve DS-VRPTW with a single vehicle. It is then followed by a generalization to fleets with several vehicles in section 3.3. After this, the principles of the two state-of-the-art methods MPA and its stochastic extension MSA are described in sections 3.4 and 3.5, respectively. The last section of Chapter 3 is dedicated to introduce the *waiting* and *relocation* improving strategies, by describing their functioning and illustrating them. We present then the framework we designed in Chapter 4. It starts with the theoretical adaptations we brought to the work of *Bent et al.* in section 4.1, followed by an overview of the structure of the framework itself in section 4.2. Sections 4.3 and 4.4 focus on the functioning of the main components of the architecture: the *Method* component and the *Offline Solver* component. To present the latter, we introduce the library we used in our framework to generate new solutions. The last section of this chapter, section 4.5, gives implementation details about the most important parts of our application. Chapter 5 presents some experiments we made and the results we obtained with our software. Finally, we complete this master thesis by a conclusion on the work achieved so far and on some elements that can be interesting to study or improve in a further work.

Chapter 2

Problem Description

This chapter first exhibits in Section 2.1 the generic expression of the `Vehicle Routing Problem` (VRP) that first appeared in 1959 [DR59] and evolved over time. This section also introduce different variants to the VRP that handle additional complicating constraints, including the variant we are interested in: the `Vehicle Routing Problem with Time Windows` (VRPTW). Section 2.2 provides a detailed description of the VRPTW, depicting in the first place the information it takes as input, then the construction of a solution from this data and finally the objective function we will optimize in the next chapters. Finally, section 2.3 introduces the extensions of VRPTW we will use later in this work, namely the `Dynamic-or Online- Vehicle Routing Problem with Time Windows` (D-VRPTW) and the `Dynamic and Stochastic Vehicle Routing Problem with Time Windows` (DS-VRPTW).

2.1 The Vehicle Routing Problem

In this section, we first expose the most generic formulation of the VRP. This is followed by a brief and non-exhaustive list of variants brought over time in the literature, each accompanied with a small description.

2.1.1 Generic expression of VRP

The most generic *Vehicle Routing Problem* is the *Capacitated Vehicle Routing Problem* (CVRP). As it is the basis for all the variants presented in the next chapters, it will also be referred to as *VRP*.

CVRP address the problem of visiting and servicing a set of customers, associated with demands, by using a given number of vehicles with limited capacity. These vehicles are kept at a unique site named `depot` when they are not on their respective route. The vehicles start their route at the depot location and must return to the same depot once they finished servicing the customers on their route. The goal of CVRP is finding a routing plan (i.e. one route for each of the vehicles in the fleet) such that all the customers of the given set are visited, without exceeding the capacity of the vehicles at any time, and minimizing the total travel distance of the fleet.

We illustrate an instance of the Vehicle Routing Problem and one of its solutions on Figure 2.1.

The VRP applies on a graph $G = (S, A)$ representing a road network, with $S = \{o, 1, ..., n\}$ the vertices of the graph representing the locations of the problem and A is the set of arcs of the road network. In S, o denotes the vertex representing the depot, while vertices $i = 1, ..., n$ denote the customers of the problem. This graph is weighted, simple and complete. The weight of an arc (i, j) between two vertices i *and* $j \in S$ is denoted as $d(i, j)$. It represents the distance

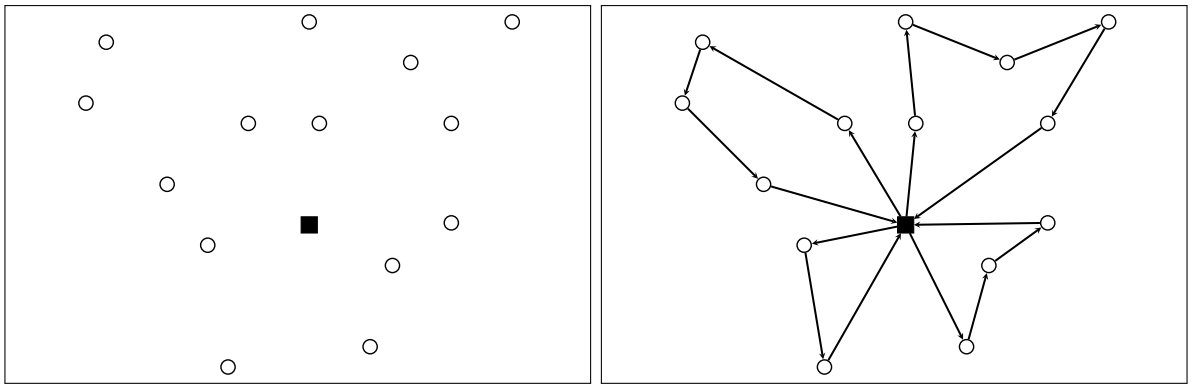

Figure 2.1: Left: Map representing the depot (filled square) and a set of customers requesting service (empty circles). Right: A solution of the Vehicle Routing Problem deploying four vehicle to serve the requests.

on the road network between the locations of those vertices. The distance between a pair of vertices is supposed *symmetric i.e.* $\forall i, j \in S : d(i, j) = d(j, i)$. Each customer vertex $i \in S \backslash \{o\}$ is associated with a demand $q(i)$.

There is additional information given in a CVRP. This information is a set of vehicles available at the depot to solve the problem. This set has a fixed size m and is composed of identical vehicles that can carry a maximal load denoted by Q. Each vehicle can process at most one route (denoted ρ) inside a solution. Each route starts and finishes with the depot location.

The objective of CVRP is thus to find a set of routes (one for each of the k vehicles actually used) denoted $sol =< \rho_1, ..., \rho_k >$ such that:

1. The size of the fleet inside the solution does not exceed the number of available vehicles, i.e.

$$k \leq m$$

2. Each customer is visited exactly once in the solution. So if we denote $cust(sol)$ the set of customers visited in sol and $cust(\rho_i)$ the set of customers visited by route i, then the condition is expressed as

$$cust(sol) = S \backslash \{o\} \ and \ \cap_{i=1}^{k} cust(\rho_i) = \emptyset$$

3. The sum of demands of the customers on a single route does not exceed the capacity of a single vehicle.

$$\forall \rho \in sol \sum_{j \in cust(\rho)} q(j) \leq Q$$

4. The total distance is minimized

$$min \ d(sol) = min \sum_{\rho_i \in sol} d(\rho_i) = min \sum_{\rho_i \in sol} (\sum_{j=0}^{length(\rho_i)-1} d(j, j+1))$$

Note that a solution respecting the constraints 1 to 3 given above is called *feasible*, whereas it is called *infeasible* if any of these constraints is violated. The fourth constraint defines the quality of a solution, it is used when two solutions are compared.

2.1.2 Literature survey

Over time, the scientific literature proposed many variants to the CVRP, each of which adds new rules to bring the initial problem closer to real situations[KP12, ILC11]. As pointed out in [Mas13], three types of variations of the initial formulation can results in new problems:

1. modification(s) to the structure of the routes
2. modification(s) to the objective function
3. additional constraints to be respected by the routes.

We now illustrate these types with actual variants and give reference to literature that address them:

Modification(s) to the structure of the routes

Multi-depot Vehicle Routing Problem In this variant, the depot is not unique. There is a new set of depots O with $|O| > 1$ such that each $o_i \in O$ has its own vehicles. Note that a vehicle starting from a depot must return to the same depot at the end of its trip. For more information, see [SS11][Fer13].

Vehicle Routing Problem With Multiple Trips Here, there can be more routes than the number of vehicle as each one of them may execute more than one route. An unified view of this variant can be found in [CAF16][Fer13].

Modification(s) to the objective function

Vehicle Routing Problem with Minimization of the vehicle fleet A common variant of the objective function is to make it lexicographic in the following fashion: set the minimization of the travel distance as secondary, and put the minimization of the number of vehicles necessary to execute all the service as primary. See e.g. [BVH04b].

Vehicle Routing Problem with Maximization of the serviced customers In this case, the objective function is also lexicographic with the minimization of the travel distance as secondary. But this time, the primary objective is to maximize the number of served customers. See e.g. [BVH04d].

Additional constraints to be respected by the routes.

Vehicle Routing Problem with Time Windows Time windows are associated with the depot and the customers. The service at each customer must start inside the customer's time window. The service at each customer uses up a predefined amount of time and vehicles must return to the depot by the end of its time window. For more information, you can find the expression of this problem in the work of *Bent et al.* [BVH04b][BVH04d], on which is based the definition of VRPTW given in Section 2.2.

Vehicle Routing with Pick-up and Delivery Each customer is associated with a pick-up location and a delivery location. The goods are gathered from the first and then conducted to the second, respecting the capacity constraint of the vehicle. This variant also introduces precedence constraints between pick-up and delivery locations. Further details can be found in

[Fer13] or [PDH08].

Note that it is possible to construct problems resulting from some combination of the variants of any type, in order to get closer and closer to reality.

2.2 The Vehicle Routing Problem with Time Windows

We now present the problem we will address in the next chapters, the **the Vehicle Routing Problem with Time Windows** (VRPTW). Our expression is mainly based on the work of *Bent et al.* in [BVH04d],[BVH03] and [BVH07] with some slight modifications, in order to standardize the ideas and notations that changed over time. This section is organized as follows: we first introduce the parameters and the input data given in a problem, then we explain how a solution is structured and constructed, and finally we express our objective function in a formal manner.

2.2.1 Parameters and Input Data

A vehicle routing problem with time windows has lots of similarities with the original VRP expressed above: it is specified by a fleet of vehicles that must travel between a number of customers in order to process the requests received. The vehicles of the fleet have a maximal capacity that can not be exceeded at any time of the day. Each vehicle starts at a depot when the problem begins, and must return at the same location by the deadline of the day. Each request coming from a customer is associated with a value that represents the capacity taken from the vehicle, but now it is accompanied with a time window representing the period of the day the customer wants to be serviced in.
Each of these components of the VRPTW will now be explained in further details.

Sites of the Problem As in the generic vehicle routing problem, two types of sites must be distinguished:

Depots Every *vehicle routing problem* contains a depot, denoted o, characterized by its location (i.e. its position on a map), a fleet of vehicles and a time interval named `Horizon` representing the period of activity of its vehicles during the problem.
The whole fleet departs from the depot at the start of the day and the total fleet of the instance is expressed as a scalar m.

Customer Locations As in the generic VRP, each problem has also a set C composed of n customers, characterized only by their respective position on a map.

The set S of sites in a VRPTW is composed by the union of theses locations: $S = C \cup \{o\}$. The information about the travel times is contained inside a matrix d, usually symmetric. The travel time between two sites i and j is simply denoted by $d(i, j) = d(j, i)$. A common relationship in matrix d is that these travel times respect the triangle inequality:

$$\forall i, j, k \in S : \ d(i, k) \leq d(i, j) + d(j, k)$$

Vehicles The fleet of vehicles available in a VRPTW is said *homogeneous*, which means that all the vehicles are identical, characterized by a capacity Q. Each vehicle departs from the depot o at the beginning of its route and returns to the depot at the end of its journey. Depending on the algorithm that implements the problem, the whole fleet can be deployed or a smaller part can be found sufficient to serve the known requests. In this case, the non-deployed vehicles can

either stay at their depot until the end of the problem (like in the *Static*-or *Offline*- version of VRPTW) or can be deployed some time later, if the size of the fleet currently used becomes insufficient over time (which can happen in *Dynamic* VRPTW's, see section 2.3).

Requests Each request r in the set of requests R of a VRPTW is associated with the following elements: a customer $c \in C$, a demand $q(r) \geq 0$ representing the load that it will take on the maximal capacity of a vehicle, a service time $p(r) \geq 0$ which is the time taken to serve the customer once the vehicle is on site, and a service time window.

Time Windows Each request $r \in R$ has a time window, denoted by a time interval $[e(r), l(r)]$ (satisfying the relation $e(r) \leq l(r)$) which represent the earliest and latest times a vehicle can arrive at the customer location to process the request r. The depot also has a time window, its bounds represent the start time of the day and the latest return time of the vehicles.

Also, note that it is possible for a vehicle to arrive on a site that sent request r at a time $t < e(r)$ but it will have to wait for the beginning of the time window $e(r)$ to begin the service.

In the later sections, the notation $[H_e, H_l]$ will be used to represents the time window of the depot, with H_e as the earliest departure time of the depot (corresponding to the beginning of the day) and H_l the latest return time of the vehicles (i.e. the end of the day).

2.2.2 Solution of a VRPTW

According to *Bent et al.*, the output of optimization algorithms solving a vehicle routing problem with time windows is a routing plan that contains the customers visited, the order in which those customers are serviced by each one the vehicles of the fleet, and the times at which each vehicle departs their locations. The succession of sites visited by a single vehicle is named a *route*, the collection of routes of the vehicles in a fleet is named a *routing plan* and finally, the leaving times of the vehicles are naturally called *departure times*.

Those structures form the basis of the the construction of a sub-optimal solution. In addition, these elements are also the decision variables on which the choice of a *better* solution among some group relies. This is why it seems important to introduce them in details.

Note that in our work, instead of being composed of a succession of customers, a route is a succession of requests. This was designed to allow a single customer to be visited more than once in a solution. This implies that the time windows and demands are not associated with customers (as in the work of *Bent et al.*), but with the requests.

Decision Variables

- **Routes** A vehicle route is a sequence of requests that represent the locations visited by a vehicle during the day. Such a sequence always starts and terminates with the depot o. More formally, a route is denoted by

$$\rho =< o, r_1, r_2, r_3, ..., r_n, o >$$

  Inside a route ρ, we denote $succ(r, \rho)$ or r^+ the successor of the request r in the route, and $pred(r, \rho)$ or r^- its predecessor.

  The demand of a route is denoted $q(\rho) = \sum_{i=1}^{n} q(r_i)$. The travel cost of a route is the cost/travel time of servicing all its requests, it is denoted by

$$d(\rho) = d(o, r_1) + d(r_1, r_2) + d(r_2, r_3) + ... + d(r_n, o).$$

Also, the notation $cust(r)$ is used to denote the customer associated to request $r \in R$ and $cust(\rho) = \cup_{i=1}^{n} cust(r_i)$ denotes the set of customers composing a route. We also use $req(\rho) = \cup_{i=1}^{n} r_i$ to denote the set of requests composing a route.

- **Routing Plan** A routing plan is a tuple of routes denoted as follows: $\gamma =< \rho_1, ..., \rho_m >$ such that reach request $r \in R$ appears exactly once inside the plan.
  It contains one route per vehicle in the fleet. Inside a plan γ, we denote $succ(r, \gamma)$ or r^+ the successor of the request r in the plan γ, and $pred(r, \gamma)$ or r^- its predecessor [1].

  The travel time of a plan $\gamma =< \rho_1, ..., \rho_m >$ is denoted by $d(\gamma)$ and is the sum of the travel times of its routes. More formally,

$$d(\gamma) = \sum_{i=1}^{m} d(\rho_i).$$

  The notations $cust(\gamma)$ and $req(\gamma)$ are used to denote the set of customers of a plan and the set of requests of a plan, respectively. Note that no time information is contained inside γ, only information about requests is visible. The missing time information is contained inside the next decision variable.

- **Departure times** This structure contains the times at which a vehicle departs the customers, once the corresponding requests are serviced. It is a tuple $\sigma =< \sigma_1(\rho_1), ..., \sigma_m(\rho_m) >$ where

$$\sigma_i(\rho_i) =< t_1, ..., t_n >$$

  is a strictly increasing sequence of integers $t_j \in [H_e, H_l]$ representing the time vehicle i starts its travel from the j-th site of ρ_i to its next destination.

Solution A solution of any VRPTW is a pair (γ, σ) optimizing the user-defined objective function and satisfying the following constraints.

Constraints of a VRPTW

The plan and departure times inside a solution can be found by any search algorithm, but these structures must satisfy the following constraints:

1. $\forall j \in \{1...m\} : \ q(\rho_j) \leq Q$
2. $\forall r \in req(\gamma) : \ \sigma(r) - p(r) \leq l(r)$
3. $\forall r \in req(\gamma) : \ \sigma(r) \geq max(e(r), \sigma(r^-) + d(r^-, r)) + p(r)$
4. $\forall r \in req(\gamma) : \ \sigma(r) + d(r, r^+) \leq l(o)$

The first constraint expresses the impossibility to exceed the maximal capacity of a vehicle. The second and third constraints assert that the time window of request r is respected: the second expresses the fact that a vehicle should always arrive before -or at- the latest arrival time of the request; while the third one asserts that a vehicle can only begin servicing a customer once the time window of the request has started. Finally, the fourth constraint ensures that all the vehicles are back to the depot by the end of the horizon.

[1] One can wonder why the notations r^+ and r^- are used in both routes and plans, this is precisely because VRPTW assumes that a request appear only once in a plan.

2.2.3 Objective Function

Now that a solution to the VRPTW is defined, an evaluation function is needed to distinguish a `best` plan among some set of plans. Usually, the goal of the VRPTW is to minimize the travel distance, as in the generic VRP. But in our case, we consider a variant we presented before: we look for a solution that maximizes the number of requests serviced by the fleet during the day. The goal could be roughly sketched by $\gamma^* = argmax(|cust(\gamma)|)$, but a more precise objective function must be expressed in order to decide between plans servicing a maximal number of customers.

This is why, in case of tie in score of this draft of objective, a secondary objective function is used. This secondary function is the aforementioned basic objective function of VRPTW, *i.e.* the minimization of the travel cost (or distance) of the vehicles in the plan. The complete objective function is thus a lexicographic bi-objective function which can be expressed as

$$w(\gamma) = (|cust(\gamma)|, -d(\gamma))$$

and must be maximized.

2.3 Extensions to Vehicle Routing Problem with Time Windows

The standard version of the Vehicle Routing Problem with Time Windows is commonly called *Static-or Offline- Vehicle Routing Problem with Time Windows* [BVH04d]. This version assumes that all the requests of the instance occurred before the launch of the operations and that the computation of the solution is done entirely before the beginning of the horizon of the depot.

A diagram representing the generic structure of an instance of the classic *Vehicle Routing Problem* (with or without time windows) can be found on Figure 2.2.

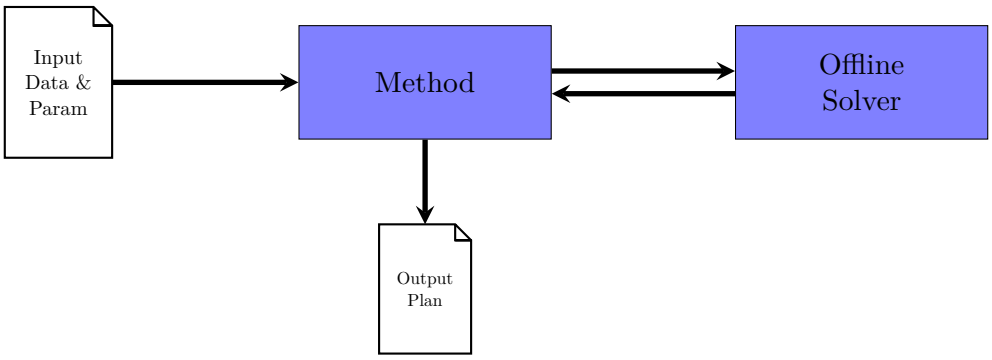

Figure 2.2: Diagram of a classic VRP instance. The rightmost blue box labeled `Offline Solver` represents the optimization algorithm that generates sub-optimal plans/solutions, according to the constraints explained previously, and additional user-defined ones, if needed. The blue box labeled `Method` represents the algorithm implemented by the user. It uses the plans generated by the solver in order to solve the problem the user is confronted to, or to improve the performances of the `solver`. This central box takes *the locations of the sites, the vehicles and their characteristics, and the requests that arrived to the "central office"* as input, and returns a routing plan that was determined as the "best" during the execution.

This section introduces the extensions to the *Vehicle Routing Problem with Time Windows* that will be used later, accompanied by a description of their respective requirements. Section 2.3.1 will depict the *Dynamic* extension of VRPTW while section 2.3.2 will introduce the *Stochastic* extension.

2.3.1 The Dynamic Vehicle Routing Problem with Time Windows

The first extension presented here is the *Dynamic* (or *online*) version of the problem (D-VRPTW). This version is more suited to operational contexts where the whole input data is not known in advance. Indeed, some requests can now arrive **during** the progress of the day. This modification implies an evolution of the plan(s) along time, it is accompanied with some other challenges:

- Some initial plan(s) must be computed with known requests before the start of the operations.
- Requests arriving during the progress of the day must be accepted or rejected **upon reception**, depending on determined criteria (e.g. possibility to insert the new request(s) inside the existing plan(s), or any other user-defined criterion...).
- Any request that was accepted at some time **must** be serviced before the end of the horizon and respect the time window of the request. This obligation is named the **service guarantee** of the accepted requests.
- At each time t, new plans are computed to integrate incoming requests (a plan q resulting of the insertion of a request in a plan p is considered different from p).

The modified diagram representing the dynamic version of the VRP can be seen on Figure 2.3.

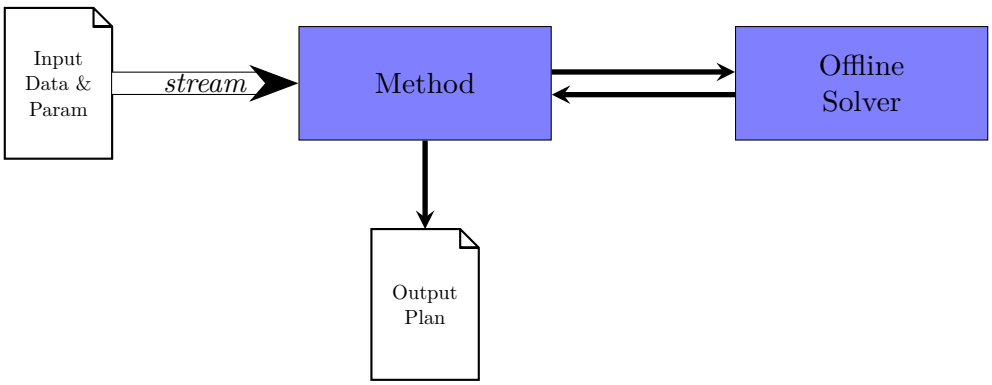

Figure 2.3: Diagram of a Dynamic VRP instance. The `Input Data` is now represented as a stream, as some requests can now be received during the progress of the operations.

New Requirements The characteristic brought by the *dynamic* aspect is the fact that the set R representing the requests of customers that have been sent to the "central office" may now vary over time.

- As a first requirement, this imposes the new notation R_t which represents the set of requests known at time $t \in [H_e, H_l]$, *i.e.* all the requests received before or at time t, **whether they are accepted or not**.
  This new set can be expressed as the following recursion:

$$R_t = \begin{cases} \mathrm{R_{t\text{-}1}} \cup \mathrm{NewRequests} & \text{if } \mathrm{t} \geq 1 \\ \mathrm{OfflineRequests} & \text{otherwise} \end{cases}$$

with R_{t-1} the requests that were part of the solution(s) at time $t-1$, $NewRequests$ the requests received in $[t-1, t]$ and $OfflineRequests$, the requests known before the start of the horizon H_e.

This new notation raised the issue of how to include the content of $NewRequests$ in the solution(s). Indeed, a request received during the dynamic phase may be completely incompatible with the current state of the algorithm. For example, it may be impossible to insert it inside the current solution(s), or the cost of the insertion (travel cost, deploy a new vehicle) may be bigger than what the user is ready to pay for an individual request.

- A method to determine whether the incoming requests can/should be accepted or not is thus required, this is a second requirement imposed by the *dynamic* aspect of the problem.

- The last but not the least requirement -as it is needed for any type of VRP- is the availability of a solver that will produce some solution(s) during the operations. Its functioning is the same as before, only its input will change over time. In the next sections, the words "solver" and "optimization algorithm" will be used interchangeably. The solvers vary according to the version of VRP it is intended to solve both by their internal behavior -which is normal-, but also at their interface -which is less trivial-. This is why a brief description of the generic interface follows.

  The Offline Solver can be written as $\mathcal{O}(state, t, A_t)$. By taking the arguments

  - *state*: which represents the current state of the progress of the day, and the past decisions (e.g. a plan describing the elapsed part of the day, a (γ, σ) pair...)
  - t: which is the time that starts the period to optimize, which finishes with the end of the horizon (i.e. the solver optimizes the time interval $[t, H_l]$). This is also the time that at which the optimization algorithm will set its output (usually, t is set to the current time).
  - A_t: represents the subset of R_t containing the requests that have been **accepted** by the algorithm implementing *Method* and that must be taken into account in order to respect the service guarantee.

  it will return a plan $\gamma = \gamma^- : \gamma^+$, with γ^- the elapsed part of the plan (*i.e.* interval $[0, t-1]$, which did not change from last iteration) and γ^+ the part in the interval $[t, H_l]$. As it was said in the previous sections, the remaining part of the day (i.e. γ^+) serves the requests accepted in $[0, t-1]$ that have not been served yet (and remind it, **must** be serviced before the end of the day because they were accepted by the central office at some point), and the requests that have been accepted in the interval $[t-1, t]$ (i. e. a subset of $NewRequests$).

2.3.2 The Stochastic Vehicle Routing Problem with Time Windows

The next extension concerns *stochastic* problems, where the relevant information is known a priori, but some parts of it are afflicted with a given uncertainty. The *Stochastic* VRP (S-VRP) is basically any VRP where one or more parameters are stochastic, meaning that some events are random variables with a known probability distribution. This extension applies both to *Static*-or Offline- and *Dynamic* versions of VRP (with or without time windows). The aim is to generate solutions enhancing the expected value of the objective function.

In dynamic and deterministic (dynamic and non-stochastic) problems, the available information at the beginning of the planning process is incomplete and there is no information about future events.

In the category of *dynamic and stochastic problems*, relevant information is revealed throughout the planning horizon, and additional stochastic information about the future, commonly collected from historical data, is available. This category is named *Dynamic and Stochastic Vehicle Routing Problems* (DS-VRP).

Several parts of the instance information can be stochastic but the most common (depicted in [RP13] and[RPH16]) are the following:

- **Stochastic Travel times:** The travel times can be associated to some probability distributions in order to provide more authentic values to match the real world in a better fashion. Indeed, many factors can affect the value of the travel time such as traffic accidents, weather, working zones on the road *etc.*
- **Stochastic Demand of the requests:** The actual demand of customers is not known in advance, but instead it is known as a random variable which follows a given probability distribution. This problem mainly arises in practical applications where unknown amount of goods have to be either delivered or collected (e.g. removal services).
- **Stochastic Customers:** The case where the customers are random variables mainly happens in the *Dynamic* version of VRP (D-VRP). In this situation, stochastic information about the expected number of customer requests and the probability of occurrence of these requests is available.
- **Stochastic Service time:** In this configuration, the time needed to process a request once on site in not known in advance and this is associated to random variables.
- **Stochastic Time windows:** The width of the time windows of some customers may not be defined precisely, but historical experience can provide useful information for the computations.

These common sources of uncertainty can of course be combined to form more complex problems, leading to more authentic solutions. In our case, we will focus on problems were the customers are the sole source of uncertainty and deterministic time windows are used. So, the **Stochastic Customers** will be the only case we will consider in the next chapters, which leads to the complete denomination of the problem we will address in this work: the ***Dynamic and Stochastic Vehicle Routing Problem with Time Windows*** (DS-VRPTW).

To deal with the stochastic information, solution methods are either based on sampling approaches ([HLL06, BVH04d]...) where possible future scenarios are included in a larger decision process or considering stochastic information explicitly. Because we follow the work of *Bent et al.*([BVH04d, BVH07, BVH04a]), we follow the first of these approaches.

Chapter 3

State of the art of DS-VRPTW

This chapter presents the different methods introduced in the work of *Bent et al.* in the last decade [HB09][BVH04d][BVH04c][BVH07][BVH04a] [BKVH05]. The corresponding pseudocodes that are presented here are adaptations of the work of *Bent et al.* using a standardized notation, as there are more or less large differences between their papers. The adapted pseudocodes are the basis of the implementation that will produce the results given in Chapter 5. The general algorithm for an *online* ***single*** *vehicle routing problem* will be introduced first in section 3.1, followed by different approaches (in an order of increasing complexity) that determines the requests that are assigned to the vehicle(s) at each time step of the online simulation in section 3.2. We then give the generalization to the ***multiple*** vehicle situation in section 3.3.

After that, we introduce additional algorithms, namely *Multiple Plan Approach* (MPA) and its stochastic widening *Multiple Scenario Approach* (MSA). These major techniques proposed by *Bent et al.* make use of the methods that we present in section 3.2 as a part of a more complex decision process. These additional algorithms are event-driven, they simulate the progress of the day and make their decisions from a pool of plans whose population evolves with the occurrence of events. We discuss the respective hypotheses and behaviors of these algorithms in section 3.4 and 3.5. Finally, section 3.6 of this chapter is dedicated to the description of two extensions to MSA, the *waiting* and *relocation* strategies. These extensions are expected to improve some more the performances of MSA over those of MPA, by extracting more benefits from its *stochastic* aspect.

Remind that from now on, the stochastic problems we tackle are problems where probability distributions for the occurrence of requests from each customer location are available. Because of this, we now distinguish two types of requests: the **actual** requests, which are requests that were explicitly received from the customers, and the **sampled** requests which are requests that are expected to appear shortly, according to the given distributions. As we show in this chapter, these sampled requests are used inside the optimizations to benefit from historical experience and increase the expected results in the objective function (see section 2.2.3).

3.1 Dynamic and Stochastic Single Vehicle Routing Problem

As explained in Chapter 2, the algorithm returns a solution (γ, σ) after some computation implying a pool of plans maintained along the execution of the horizon. The maintained set of plans Γ represents solutions used to make decisions over the course of computation. A complete algorithm of an online solver/simulator can be seen on Algorithm 1. Note that in this algorithm, the pair returned is denoted (ρ, σ). This is because we are in a situation with a single vehicle, so the notation of a *route* (ρ) and the notation of a *plan* (γ) can be used interchangeably.

An important thing that must be introduced is the discretization of the time in **all** the algorithms that we presented in this work. Indeed, we consider that the horizon and the time windows in the problems are discrete intervals with unit time steps. This means that if the horizon of the depot is expressed $[H_e, H_l] = [0, 100]$, then there will be one hundred of online time steps during the online execution (0 is the offline part). With this functioning, time can thus have the values $t = H_e...H_l$.

Algorithm 1 The Generic Online Algorithm with Single Vehicle

1: **function** OnlineAlgorithm($\langle R_0, R_{H_e}, ..., R_{H_l} \rangle$)
2: $\rho \leftarrow \langle\rangle$;
3: $\sigma \leftarrow \sigma_\perp$;
4: $\Gamma \leftarrow$ GenerateSolutions($\rho, \sigma, R_0, 0$);
5: **for** $t \in [H_e, H_l]$ **do**
6: $A_t \leftarrow$ AcceptRequests(R_t, A_t, Γ, t);
7: $\Gamma \leftarrow$ UpdatePlans($\rho, \sigma, A_t, \Gamma$);
8: **if** Idle(ρ, σ, t) **then**
9: $r_t \leftarrow$ ChooseRequest($\rho, \sigma, A_t, \Gamma, t$);
10: $\rho \leftarrow \rho : r_t$;
11: $\sigma \leftarrow \sigma[\text{Last}(\rho) \leftarrow t]$;
12: $\Gamma \leftarrow \{\ \rho_i \in \Gamma \mid \text{First}(t, \rho_i - \rho) = r_t\ \}$;
13: **else**
14: $\rho \leftarrow \rho$;
15: $\sigma \leftarrow \sigma$;
16: $\Gamma \leftarrow \Gamma \cup$ GenerateSolutions(ρ, σ, A_t, t);
17: **return** (ρ, σ);

18: **function** GenerateSolutions(ρ, σ, A_t, t)
19: $\Gamma \leftarrow \emptyset$;
20: **repeat**
21: $Req \leftarrow A_t \cup$ Sample(ρ, σ, t);
22: $\Gamma \leftarrow \Gamma \cup \{\mathcal{O}(\rho, \sigma, Req, t+1)\}$;
23: **until** time $t+1$
24: **return** Γ;

In Algorithm 1, the output pair is constructed incrementally, beginning with empty data structures and growing it at the end of the timesteps, if needed. Lines 2-4 execute the initialization phase: lines 2 and 3 initialize the route and departure times and line 4 generates the initial set of plans (*i.e.* plans including only offline requests, denoted by R_0). The body of the `for` loop simulates the decision making performed during the horizon. It first determines whether to accept requests received during the previous time step (line 6, set A_t is updated), then removes the plans from Γ that are incompatible with an insertion of these new requests (line 7). It should be noted that Γ can never be empty, which means that any request that was not able to be inserted in any plan from Γ is simply rejected. This is because setting Γ empty violates the the necessary `service guarantees` (reminder: the obligation to serve any accepted but not serviced request during at some point of the remaining time).

The `if/else` structure (lines 8 to 15) ensures that the output `plan` (ρ in Algorithm 1) and `departure times` (σ) are updated only when a new customer should be added to the route (i.e. the vehicle is waiting for the next instructions). This corresponds to the fact that a vehicle takes

note of its next destination only if it is *idle*.

The function IDLE(ρ, σ, t) is a function that returns `true` if the vehicle is at the depot or if the time at which the vehicle finished servicing the last customer of the route (in the plan created incrementally) is smaller than current time. More formally, for a growing plan/route $\rho = \langle r_1, ..., r_k \rangle$ and departure time σ:

$$\text{IDLE}(\rho, \sigma, t) = \begin{cases} true & \text{if}(\ (\rho = \langle\rangle) \vee (max(\sigma(r_{k-1}) + d(r_{k-1}, r_k), e(r_k)) + p(r_k) \leq t)\ ) \\ false & \text{otherwise} \end{cases}$$

- If the vehicle is idle at time t, the next request to be serviced is chosen (line 9) according to the future actions described by the pool of plans, and is appended to the output pair, as the next destination of the vehicle (lines 10-11). Finally, all the plans that did not contained this request/customer as the next one on the route are removed from the pool of plans (*i.e.* those plans are removed as they are not compatible anymore with **all** the past decisions).
- Otherwise, the vehicle is either in transit or servicing a customer, which means that no modification should be performed on the output pair, as it can be seen at lines 14-15.

Line 16 adds new plans to the pool in order to maintain a certain population in Γ, as it is filtered during execution. Finally, line 17 returns the solution constructed along the whole horizon, once it is finished.

The second function GENERATESOLUTIONS in Algorithm 1 describes how solutions to be added to the pool are generated. It is quite straightforward to understand its behavior. An empty set of plans is initialized and then, it creates new plans by generating sets of requests *Req* containing the ones in A_t and sampled ones (it creates a scenario), and calling the *optimization algorithm* $\mathcal{O}$. Then, it adds each generated solution to the set Γ. The function repeats these operations while the current time step is not finished, and finally returns the set of plans.

Two called functions still need to be explained in this algorithm: ACCEPTREQUESTS and CHOOSEREQUEST, so we will briefly address them now.

ACCEPTREQUESTS will be tackled quickly, as in all the papers read ([BVH04d][HB09]...), the requests are accepted in a `greedy` fashion. Indeed, in the all the work of *Bent et al.* the requests are accepted as long as there is one plan in Γ where it can be inserted. It is quite easy to understand why they chose this way as any more accurate but more complex method would yield to another complex combinatorial problem. *De facto*, this new problem of finding the optimal combination of requests in *NewRequests* that maximizes the overall scores of the plans in Γ can become more and more complex as the size of the fleet, the number of new requests or the number of plans in the pool grows. Moreover, as other user-defined parameters can influence the acceptation of a specific request, we do not give an actual pseudocode ACCEPTREQUESTS and prefer to follow the idea of *Bent et al.*

The implementation of the second function CHOOSEREQUEST is the only part of the algorithm that determines the approach used. As there are many of these approaches, using more or less complex notions, the following section will be dedicated to describe some of them, namely section 3.2.

After the exposition of these approaches, we present the generalization of Algorithm 1 to the multiple vehicles situation in section 3.3.

3.2 Approaches to Choose the Next Request

The different methods presented in this section were developed in order to make more informed decisions from a limited number (denoted $n_{\mathcal{O}}$) of offline optimizations. This limitation comes from the length of a time step. These approaches assume the availability of: the sampling distributions (seen as black boxes, the way they were obtained is not important) and a procedure to solve or at least approximate the offline problem.

Another thing that must be noticed is that the following pseudocodes are designed to process one request at each time t of the horizon (remind that time is discrete in all the implementations of this work). This is because they were designed to be general enough to be able to tackle several types of online optimization problems such as the *Packet Scheduling Problem* in addition to the *VRP*.

3.2.1 Greedy

This approach is the simplest among the ones we present here. In this configuration, the request that is selected is the request r that maximizes its gain $w(r)$, which brings the "greedy" name of the approach naturally. This score $w(r)$ can be any user defined criterion such as the *income* perceived from the request, the $\frac{income}{capacity}$ ratio, *etc.*
The pseudocode of the greedy implementation of ChooseRequest can be found on Algorithm 2.

Algorithm 2 Implementation of ChooseRequest, **greedy** version.

1: **function** ChooseRequest-G($\rho, \sigma, A_t, \Gamma, t$)
2: $X \leftarrow$ Ready($\rho, \sigma, A_t, \Gamma, t$);
3: **return** $argmax(r \in X)\ w(r)$;

In Algorithm 2, function Ready returns the subset of the known and accepted requests A_t that are serviced at time t in one or more plans of Γ, knowing the ones that are already serviced in the solution (ρ, σ).

The complexity of the *greedy* approach is thus $O(|A_t|)$ as the set X is at most the same as A_t and this set is traversed linearly.

3.2.2 Expectation

This second approach was first proposed by *Chang et al.* in [CGC00] for the *online packet scheduling problem*, and was adapted to the *Dynamic and Stochastic* VRP by *Bent and Van Hentenryck* in [BVH04a].

This method generates future requests via sampling and evaluates each actual request against that sample. It chooses the next request to schedule by approximating the expected gain of each request on the score of generated scenarios.
The pseudocode of ChooseRequest using the *expectation* approach is depicted in Algorithm 3. In this pseudocode, lines 2-4 reach the set of requests that can be served at the current time, and initialize the score of each request inside this set. Then, the number of offline optimizations is distributed among the requests in X: $\lfloor n_{\mathcal{O}}/|X| \rfloor$ scenarios *Req* are created (line 6) and each one is optimized $|X|$ times by calling $\mathcal{O}$ in the nested `for` loop at lines 7-9. The nested loop goes through the set of requests available at the current time and implicitly schedule the current request r as the next one on the route, looks for an offline solution containing all the requests in

Req in the time interval $[t+1, H_l]$.

Algorithm 3 Implementation of ChooseRequest, **expectation** version.

```
1: function ChooseRequest-E(ρ, σ, A_t, Γ, t)
2:     X ← Ready(ρ, σ, A_t, Γ, t);
3:     for r ∈ X do
4:         f(r) ← 0;
5:     for i ← 1...⌊n_O/|X|⌋ do
6:         Req ← A_t ∪ Sample(ρ, σ, t)
7:         for r ∈ X do
8:             sol ← O(ρ, σ, Req \ {r}, t + 1);
9:             f(r) ← f(r) + w(r) + w(sol)
10:    return argmax(r ∈ X) f(r);
```

The point of incrementing $f(r)$ by $w(r) + w(sol)$ instead of $w(sol)$ only is simply to decide between requests whose *sol* would serve the same number of customers. Indeed, in this case, the request with the highest gain will be chosen (i.e. in case of tie, we get back to some greedy approach).

Each of the $\frac{n_\mathcal{O}}{|X|}$ scenarios (*i.e.* each set *Req* computed in the `for` loop in lines 5-9) is evaluated for all request r (second `for` loop, whose complexity is $O(|A_t|)$) . Finally, the request with the highest score is returned to the caller at line 10, the complexity of this step is $O(|A_t|)$. The total complexity of the *expectation* approach is thus

$$O(|A_t|\frac{n_\mathcal{O}}{|A_t|} + |A_t|) = O(n_\mathcal{O} + |A_t|)$$

Observe with the second `for` loop (lines 5 to 9) that the number $n_\mathcal{O}$ of offline optimizations is distributed among the requests in X. This means that if $n_\mathcal{O}$ gets smaller (i.e. the length of a time step shortens), each requests will be evaluated with respect to a lower number of scenarios. This implies that only a small amount of information will be deduced in this case.

This last fact is a restraint in the use of this approach in the *Dynamic VRP* as the optimization in this problem is highly computationally intensive and moreover the number of requests in X can be too large for the time constraints.

One can wonder why we introduce an approach that seems irrelevant to our problem. We do this because it will be useful to understand one of the two next approaches. Indeed, the *regrets* approach is in fact a combination of the expectation approach and the *consensus* approach introduced in the next section.

3.2.3 Consensus

The consensus algorithm was first introduced by *Bent et al.*[BVH04d]. This approach relies on the idea that the available optimizations/solutions are good in term of quality (the optimal solution is close) and thus taking decisions that reduce the divergence between solutions is a good approach. This can be seen as the **consensus principle**. The fundamental difference with the *expectation* algorithm is that in this version, each possible request at time t is not evaluated anymore with respect to each scenario. Instead, *consensus* goes through the offline optimizations in Γ (i.e. on each scenario encountered) and then counts the number of times each request is scheduled at time t (*i.e.* is set as the next one on the route of the vehicle). The request with the

highest count is then selected and returned to the caller to be set as the next one on the route created incrementally.

The pseudocode of the *consensus* is depicted on Algorithm 4.

Algorithm 4 Implementation of ChooseRequest, **consensus** version.

```
1: function ChooseRequest-C(ρ, σ, A_t, Γ, t)
2:     for r ∈ A_t do
3:         f(r) ← 0;
4:     for sol = (ρ_i, σ_i) ∈ Γ do
5:         r ← First(t, Filter(sol − (ρ, σ), A_t));
6:         f(r) ← f(r) + 1;
7:     return argmax(r ∈ A_t) f(r);
```

In this algorithm, lines 2-3 initialize the score of each request $r \in A_t$ (remind, the set of all the known and accepted requests until time t) to 0. Then for each plan available in the pool Γ (*for* loop in lines 4-6), it finds the first request in the $[t+1, H_l]$ interval (which is the result returned by function $\textsc{First}(t, sol)$) and increments its score by 1. Finally, the request with the highest score is returned at line 7.

The complexity of this approach is $O(|\Gamma| + |A_t|)$, as the entire pool of plans is traversed (hence the $|\Gamma|$) to increase the score of the request that is the next on the route, before returning the request maximizing the score function.

It should be emphasized that in the second `for` loop, r can only be part of A_t. Indeed, only the score of actual requests should be computed, as sampled requests should not really be serviced and thus computing their score is irrelevant. This is precisely the reason why the function Filter is called, as there can be sampled requests inside the solutions in Γ. $\textsc{Filter}(sol, R)$ is a function that removes the requests of the given solution *sol* that are not part of the given set of requests R. So, in order to remove all the sampled requests from a route, the set A_t containing all the **accepted** requests, is passed as an argument.

Advantages and Limitations It was shown in [BVH04a] that *consensus* takes advantage of stochastic information as it outperforms simpler approaches like the *greedy* approach and the *Local Optimal* approach, not presented here, in terms of efficacy with respect to the objective function introduced in section 2.2.3. The main advantage of the *consensus* approach is that the available scenarios/optimizations are not distributed anymore to the requests. That is represented by the second **for** loop (lines 4 to 6 in Algorithm 4): the number of iterations is not divided by $|X|$ anymore. In an experimental context, this is a significant advantage in the common situations of D(S)-VRP's we mentioned before: when the number of optimizations per time step is small and/or the number of requests to evaluate is large. The work of *Bent et al.* also showed that the performances of *expectation* and *consensus* are reversed as the number of offline optimization per time steps increases. Besides this, the *consensus* is not perfect and has limitations. The first one is that it does not take the score of the plans into account as it only counts them. Second is the elitism of the approach. Only the best request is given credits for a scenario, while other ones are ignored. This implies that similarity between requests and the fact that a request may never be the best for any sample, but may still be extremely robust overall,

are simply ignored[1].

3.2.4 Regrets

The last approach we present is a combination of the two preceding ones. Presented in [BVH04a], the *regrets* approach rely on the idea that an estimation of the regret of a request at a time t can be computed quickly. The regret of a request r can be represented by the difference in score value between a solution `sol` and the best solution serving r at time t. As computing the actual value of the regret of a request is hard because the optimal solution is not known, it will be approximated with the information contained in the plans inside Γ.

The implementations of the *regrets* approach assume the availability of a function REGRET that returns an approximation of the value of the regret such that its computation time is negligible compared to the computation time of an offline optimization.
In the case of vehicle routing, the regret of serving a customer c next can be evaluated by swapping the next customer on the route with c. To make this more clear, consider the example of choosing which request should be set as the next on the route. Let r_f be the next request planned in a plan and r_c another planned but unserved request. The idea to determine the regret of r_c is to verify if there exists a feasible swap of r_c and r_f, in which case the regret is 0. Otherwise (*i.e.* if such a swap violates the time window constraints), the regret is 1.
More formally, if we denote $sol = (\rho, \sigma)$ a solution in the pool, r_1 the next request on the route and $sol_{r_1 r} = (\rho', \sigma')$ the solution where request r_1 and an other unserviced request r were swapped, the regret of r can be expressed as:

$$\text{REGRET}(sol, t, r) = \begin{cases} 1 & \text{if}(sol_{r_1 r} \text{ is feasible}) \\ 0 & \text{otherwise} \end{cases}$$

A pseudocode of CHOOSEREQUEST using the `regrets` approach can be found in Algorithm 5.

Algorithm 5 Implementation of CHOOSEREQUEST, **regrets** version.

```
1: function CHOOSEREQUEST-R(ρ, σ, A_t, Γ, t)
2:     for r ∈ A_t do
3:         f(r) ← 0;
4:     for sol = (ρ_i, σ_i) ∈ Γ do
5:         r_t ← FIRST(t, sol − (ρ, σ));
6:         f(r_t) ← f(r_t) + w(sol);
7:         for r ∈ READY(ρ, σ, A_t, Γ, t) \{r_t} do
8:             f(r) ← f(r) + (w(sol) − REGRET(sol, A_t, t, r))
9:     return argmax(r ∈ A_t) f(r);
```

In this algorithm, lines 2 and 3 initialize the scores of the accepted requests, as in the previous approach. Then the `for` loop in lines 4-8 iterates on the offline optimizations to increase these scores. Line 5 determines the next destination r_t depicted inside the current solution and line 6 increases the scores of this request by the score of the solution. Then, the nested `for` loop (lines 7-8) will increase the score of all the requests returned by the READY function except

[1] As written in [BVH04a]: *The fundamental issue is thus to determine if it is possible to gather that kind of information* [the overall robustness] *from the sample solutions without solving additional optimization problems.*

r_t, by the score of the current solution minus the regret of swapping r with r_t. Note that the set returned by READY corresponds to the set of all the requests that were returned by FIRST once or more. Finally, the request with the highest score among the requests inside A_t is returned.

The complexity of the *regret* approach is $O(|\Gamma|.|A_t| + |A_t|)$, because for each plan inside the pool, the score of all the requests returned by READY (which is at most identical to A_t) is modified, before returning the request maximizing the score function.

As we emphasized, *regrets* combines *consensus* and *expectation* to improve the performances. Like *consensus* (and unlike *expectation*), it avoids distributing the offline optimizations among the requests. Also, like *expectation* (and unlike *consensus*), each available request in *regrets* is evaluated for every plan in Γ. Moreover, the *elitism* limitation of *consensus* is solved here because the fact that some choices of customers are similar/equivalent is a key feature of the *regrets* approach. Hence, *regrets* preserves the salient features of both approaches and address their respective limitations and because of the removal of these limitations by more complex computations, better results are expected when *regrets* approach is used, than when the two others are used.

3.3 Dynamic and Stochastic Multiple Vehicle Routing Problem

It is now time to show the generalization of the generic online routing algorithm to a fleet with $m > 1$ vehicles. As a solution will contain multiple routes, we get back to use the notations introduced in the previous sections. So a solution will be denoted by $\gamma = (\rho_1, ..., \rho_m)$, where ρ_i is the route of vehicle i.

Using several vehicles implies some modifications to the general algorithm in order to perform well.

- From now, the algorithm is considered *idle* if **any** of the vehicles in the fleet is idle: Formally, IDLE now returns the output of the boolean function

$$\text{IDLE}(\gamma, \sigma, t) \equiv \bigvee_{i=1}^{m} \text{IDLE}(\gamma(i), \sigma, t).$$

- The decision process must take all vehicles into account, as even if only one vehicle is idle, the routing of other vehicles has an impact on the decision.
- The computation of the decisions is more complex as more vehicles are involved: a decision has now the form of a tuple $(r_1, ..., r_m)$ representing the next customer/request each vehicle will serve next, not only the requests of idle vehicles.
- The fact that several vehicles can become *idle* simultaneously make the decision process even more complex.

The modified version of the generic online algorithm is depicted in Algorithm 6. The differences with the original algorithm will now be listed and reviewed in order of occurrence.

First, the structures initialized in line 2-3 correspond to plans instead of routes as announced and the occurrences of ρ have been replaced by γ when needed in the later lines of pseudocode. Second, the output of CHOOSEREQUEST is now a tuple of requests $(r_1, ..., r_m)$, accordingly to what we said beforehand. This is the reason why the r_t in algorithm 1 was replaced by tr_t. Note that vehicle i does not have to depart for customer r_i in the tuple, as it may still being traveling

or servicing its currently assigned request. As we said, there is an entry in the tuple for each vehicle, but only those that are idle will be affected in the `if` structure at lines 13 to 16.
The third modification is the new `for` loop that goes through the vehicles of the fleet to update the routes of the ones currently idle (lines 11-16). The body of the inner `if` structure is the same as what was executed in the algorithm with a single vehicle: it adds a departure time to the routes of idle vehicles (line 14 associates time t to the current last request of route i (obtained by calling $\textsc{Last}(\rho)$ and add it to σ), appends the i-th customer of tr_t to the i-th route and finally prunes the plans that are incompatible with this decision.

Algorithm 6 The Generic Online Algorithm with Multiple Vehicle

1: **function** OnlineAlgorithm($\langle R_0, R_{H_e}..., R_{H_l}\rangle$)
2: $\gamma \leftarrow \langle\rangle$;
3: $\sigma \leftarrow \sigma_\perp$;
4: $\Gamma \leftarrow$ GenerateSolutions($\gamma, \sigma, R_0, 0$);
5: **for** $t \in [H_e, H_l]$ **do**
6: $A_t \leftarrow$ AcceptRequests(R_t, A_{t-1}, Γ, t);
7: $\Gamma \leftarrow$ UpdatePlans($\gamma, \sigma, A_t, \Gamma, t$);
8: **if** Idle(γ, σ, t) **then**
9: $tr_t \leftarrow$ ChooseRequest($\gamma, \sigma, A_t, \Gamma, t$);
10: $\sigma \leftarrow \sigma$;
11: **for** $i \in 1..m$ **do**
12: $\rho_i \leftarrow \gamma(i)$;
13: **if** Idle(ρ_i, σ, t) $\wedge$ $tr_t(i) \neq \perp$ **then**
14: $\sigma \leftarrow \sigma[\textsc{Last}(\rho_i) \leftarrow t]$;
15: $\rho_i \leftarrow \rho_i : tr_t(i)$;
16: $\Gamma \leftarrow \{\ \gamma_j \in \Gamma \mid \textsc{First}(t, \gamma_j - \gamma)(i) = tr_t(i)\ \}$;
17: $\gamma_t \leftarrow \langle \rho_1, ..., \rho_m \rangle$
18: **else**
19: $\gamma \leftarrow \gamma$;
20: $\sigma \leftarrow \sigma$;
21: $\Gamma \leftarrow \Gamma \cup$ GenerateSolutions(γ, σ, A_t, t);
22: **return** (γ, σ);

Algorithm 7 Pointwise Implementation of ChooseRequest with **consensus** for multiple decisions.

1: **function** ChooseRequest-C($\gamma_t, \sigma_t, A_t, \Gamma$,t)
2: **for** $r \in A_t$ **do**
3: $f(r) \leftarrow 0$;
4: **for** $sol = (\gamma_i, \sigma_i) \in \Gamma$ **do**
5: $tr \leftarrow \textsc{First}(t, \gamma_i - \gamma_t)$;
6: **for** $i \in 1..m$ **do**
7: $f(tr(i)) \leftarrow f(tr(i)) + 1$
8: $\gamma^c = argmax(sol = (\gamma_i, \sigma_i) \in \Gamma) \sum_{i=1}^{m} f(\textsc{First}(t, \gamma_i - \gamma_t)(i))$;
9: **return** $\textsc{First}(\gamma^c - \gamma_t)$;

It remains to explain the modifications brought to ChooseRequest so that it behaves

as expected with a fleet composed of several vehicles. The main challenge faced is to make a global decision encapsulated inside a tuple tr_t. The solution to this was presented in [HB09] as a *Pointwise Implementation of Consensus*, whose algorithm can be found on Algorithm 7. This pseudocode is divided into two steps:

- The first step goes from line 2 to line 7. In this step, each decision (*i.e.* each request in the tuple tr containing the next request of each route) is evaluated individually across the plans in Γ.
- The second step (lines 8-9) evaluates each plan γ by the score of the decisions taken and the best plan is selected. The selection of γ^c is made by choosing the plan with the highest sum of the scores associated to the firsts requests of each of its routes. Then, the first next customers of the routes inside the best plan γ^c is returned to the caller.

The fact that the tuples that can be returned by ChooseRequest are limited to the ones that are found inside the plans in Γ ensures the feasibility of the plans after the decision. This is because Γ can only contain plans that satisfy the problem-specific constraints. Moreover, by ranking the plans according to the individual requests they serve, the pointwise decision algorithm maintains feasibility while distinguishing the quality of the plans precisely.

Now that use the approaches to choose the next request(s) are introduced, we present in the next section the two main techniques proposed by *Bent et al.* in [BVH04d] that use these approaches to maximize the number of served customers.

3.4 Multiple Plan Approach

The Multiple Plan Approach (MPA) is a method proposed by *Bent et al.* in 2004 [BVH04d]. This algorithm maintains two structures: a pool of plans (also called solutions) Γ used to take decisions during the online progress, and a solution $sol^* = (\gamma^*, \sigma^*)$, considered as the *best* plan (also referred to as *distinguished* plan), and representing the routing actually followed by the vehicles. The pool of plans varies over time when events occur, either to remove a part of the solutions in Γ, modify their content, or simply add new ones. The solutions inside Γ are ranked regularly with a user-defined function such as the ones introduced in Section 3.2 and the one with the highest ranking is kept as the *best* plan. In their work [BVH04d][BVH07][HB09], *Bent et al.* implemented these and compared their respective efficacy. MPA was proposed in order to measure and highlight the improvements of its generalization, MSA (presented in section 3.5), that is allowed to use stochastic information to use *sampled* requests in order to take more informed decisions and increase the number of served customers.

The rest of this section is organized as follows: section 3.4.1 exposes the hypotheses on which MPA is based, section 3.4.2 describes the aforementioned events, and eventually section 3.4.3 describes the generation of the plans that feeds Γ.

3.4.1 Hypotheses

The technique named MPA relies on a few hypotheses which are the most basic rules that will be respected by any algorithm implementing MPA and its extensions.

1. A vehicle knows only the next destination on its route, it never takes notes of the further customers.
2. A vehicle sees and locks its next destination when becoming idle: It travels to the region that was labeled as its next destination in the last best plan computed **before** it became

idle. This means that before becoming idle (*e.g.* during service at current location), the best plan can change and so the next destination. But once it is idle, the vehicle directly locks the customer in the corresponding route of the best plan as its destination. Also, a vehicle will never be reassigned a new destination during travel, nor when waiting to serve the customer at its current location.

3. All the plans inside the pool must contain the same requests, possibly at different positions in the routes.

4. All the plans in the pool are **feasible** and **consistent** with the past decisions.

These hypotheses are meant to simulate situations that are close to those that occur in the real world. Indeed, it is common for delivery men to know only their next destination. Also, it seems obvious that once they are on the move, delivery men are not assigned to completely different destination because the loss (*e.g.* in travel costs) would be too large.
Hypothesis 3 is more technical as it is in fact the expression of a requirement of the *Dynamic Vehicle Routing Problem with Time Windows*(DS-VRPTW), namely the *service guarantee*. Finally, Hypothesis 4 ensures the consistency of the pool with all the decisions taken during the operations.

3.4.2 Events

As we said, the pool of plans Γ varies over time with events. These events correspond to real actions that may happen during operations, and that may have an impact on the solutions in Γ. Indeed, the occurrence of events may cause violation of the aforementioned hypotheses by some plans. Those plans should thus obviously be removed.

MPA handle four basic types of event: (a) timeouts, (b) generation of new plans, (c) arrival of a new customer request, and (d) vehicle departures from a customer. We describe now these types of event in further details and explain their impact on Γ.

Timeouts These events occur because some plans become invalid over time. This happens when the *best* plan specifies that a vehicle should wait at its current location while some plans recommend that the same vehicle should depart this location. More formally, the pool of plans is updated as follows.

$$\Gamma := \{sol = (\gamma, \sigma) \in \Gamma \mid \textsc{Feasible}(sol, t)\}$$

where $\textsc{Feasible}(sol, t)$ holds if at time t,

$$\forall r \in req(sol^{-}) : \mathrm{LDT}(sol, succ(r, \gamma)) \leq t,$$

$req(sol^{*-})$ denotes the set of customers from which a vehicle departed before or at time t in sol^{*}, and LDT returns the last departure time of request r in solution sol.

The plans that are filtered out have to be removed from Γ because they violate Hypothesis 4 we introduced in the previous section. These plans were not consistent anymore with the past decision of staying at the current location. This decision was made when the solution that requires this wait was set as the *best* plan of the pool.

Finally, as no plan was added in Γ, and because no plan in the pool was modified, the *best* plan remains the same as before.

New Plan Feasible solutions are continuously generated during the *horizon* and added to the pool. The role of this kind of event is straightforward: it does not break any hypothesis but as the events presented here filter out plans from the pool Γ and because the decisions taken during execution are better if the population of the pool is large, it is necessary to maintain a certain number of plans inside Γ. In other words, the new pool can be expressed

$$\Gamma_{t+1} := \Gamma_t \ \cup \ \{sol = (\gamma, \sigma)\}.$$

In this equation, *sol* is a new plan that was just generated and transferred to MPA. As this new solution may be better than the *distinguished* plan, the *best* plan should thus be recomputed from the new Γ set.

$$sol^*_{t+1} := argmax(sol = (\gamma, \sigma) \in \Gamma) \ f_t(sol).$$

New Incoming Requests As explained in the previous sections, in *dynamic* VRP's new requests can arrive during the progress of the day. *New incoming requests* capture these events. The arrival of a new request updates the plans inside Γ to accommodate this request, in order to minimize the request rejection, but remind that it should be avoided to empty the pool completely.

$$\Gamma^{'} := \{\textsc{Insert}(sol, r) |\ sol = (\gamma, \sigma) \in \Gamma,\ \textsc{FeasibleInsert}(sol, r)\}$$

where $\textsc{FeasibleInsert}(sol, r)$ returns true iff there exists a point in *sol* for the request r that does not violates the constraints of the problem. $\textsc{Insert}(sol, r)$ returns a new plan $sol^{'}$ where r has been inserted, minimizing the travel cost and satisfying the constraints. It is important to emphasize the behavior of MPA when FeasibleInsert returns `false` for all $sol \in \Gamma$: Because the service must be guaranteed to all the customers already accepted, Γ can not be empty at any point of the horizon, which means that any request rejected by all the plans in Γ is simply set as **refused** by MPA. Furthermore, the insertion of a new request r inside the solutions affect their respective score, which re-shuffles the cards between the plans were the insertion was possible. This means that the *best* plan must be recomputed among the remaining ones in $\Gamma^{'}$.

This behavior can be expressed more formally as follows:

$$\begin{aligned}
&if\ \Gamma' \neq \emptyset\ : \\
&\qquad \Gamma_{t+1} := \Gamma' \\
&\qquad sol^*_{t+1} := argmax(sol = (\gamma, \sigma) \in \Gamma_{t+1}) \ f_t(sol) \\
&otherwise : \\
&\qquad \Gamma_{t+1} := \Gamma_t \\
&\qquad sol^*_{t+1} := sol^*_t
\end{aligned}$$

As we said, if $\Gamma^{'}$ is empty, then the request is rejected and the pool of plans is left unchanged. Otherwise, $\Gamma^{'}$ is set as the new pool of plans and the new *best* plan from $\Gamma^{'}$ is computed. In the second case, the pool Γ has to be filtered out because Hypothesis 3 is violated. Indeed, there is two subset in Γ: the plans that accepted the request (Γ') and those that could not. As these two subsets serve different sets of requests and because the overall objective function is to maximize the number of served requests, the solutions that could not integrate the new request are removed and Γ' is set as the new pool.

Departures This type of events is the inverse of *Timeouts* and also violates Hypothesis 4. Departures capture the event when the *best* plan specifies that a vehicle should depart from its current location. Then, all the plans in Γ that do not also recommend the departure of the same vehicle from the same location are now considered as incompatible with the past decisions and thus must be filtered out from the pool.

$$\Gamma := \{sol = (\gamma, \sigma) \in \Gamma \mid \textsc{Compatible}(sol, \gamma^*, \sigma^*, t)\}$$

where $\textsc{Compatible}(sol, \gamma^*, \sigma^*, t)$ returns true iff $\forall r \in req(sol^{*-}) : succ(\gamma^*, r) = succ(\gamma, r)$; in other words, if the next location visited by a vehicle departing a customer is the same that in the *best* plan.

3.4.3 Plan generation

The plan generation is divided into steps. The first one consists of the retrieval of the past decisions and the set of accepted requests, which can be done by taking any plan in the pool Γ, as it must contain all this information. From the solution, the unserviced requests are retrieved easily. The second step is a simple call to the offline solver, which optimizes the time interval $[t + 1, H_l]$ with the unserviced requests. The third and final step sends the solution of the optimization to MPA as a *new plan* event.

Algorithm 8 Pseudocode of the plan generation used by `MPA`.

```
1: function GenerateSolutions(sol = (γ, σ), A_t, t)
2:     remaining_requests ← A_t \ req(γ^-)
3:     new_sol ← sol^- : O(sol^-, t, remaining_reguests)
4:     return new_sol;
```

3.5 Multiple Scenario Approach

The Multiple Scenario Approach (MSA) is similar to MPA, except that the plan generation involves *hypothetical* requests in addition to known requests. The requests of this new type, also named *sampled* or *future* requests, appear in solutions when sampled according to some known probability distributions. The plans in MSA are thus called *scenarios* to emphasize their 'imaginary' aspect brought by the sampling. Note that there can be several scenarios among the plans inside Γ, there is no obligation for all plans to contain the same *sampled* requests.

MSA was proposed by *Bent et al* in [BVH04d] alongside MPA to show the benefits of using stochastic information coming from experience, in terms of reduction of refusal of online requests (remind the objective function we defined in section 2.2.3).

New Hypotheses

The fact that the new requests are imaginary requires the addition of an additional hypothesis:

5. Only actual requests can be processed: a vehicle will not travel to a location where there is no ACTUAL request, it can not 'simulate' service.

Also, Hypothesis 3 of MPA must be slightly modified:

3. All the plans inside the pool must contain **at least** the same **actual** requests, possibly at different positions in the routes.

This modification permits MSA to imagine different scenarios (i.e. different sampling), while conserving the **service guarantee** property.

Concerning the events, no major modification is needed, which means that the only changes that should be brought to MPA are located into plan generation.

Plan Generation

The adaptation of the generation of new plans to MSA (depicted on Algorithm 9) can be summarized by the insertion of two steps: one right before the optimization and one right after. Indeed, as scenarios must be created by using *hypothetical* requests, a new step that samples these requests according to their respective probability of occurrence must be added before the optimization (line 3). The second additional step is brought by the new Hypothesis 5 presented in the previous section. As no *sampled* request should be served during the day, they simply are removed from the solution found (line 5) before it is sent to MSA as an event. The solution is said *projected* on the *known* requests.

Algorithm 9 Pseudocode of the plan generation used by `MSA`.

1: **function** GENERATESOLUTIONS($sol = (\gamma, \sigma), A_t, t$)
2: $remaining\_requests \leftarrow A_t \backslash req(\gamma^-)$;
3: $remaining\_requests \leftarrow remaining\_requests \cup$ SAMPLE(γ, σ, t);
4: $new\_sol \leftarrow sol^- : \mathcal{O}(sol^-, t, remaining\_reguests)$;
5: $new\_sol \leftarrow$ FILTER(new_sol, R);
6: **return** new_γ;

3.6 Improving Strategies for MSA: Waiting and Relocation

The performances of the MSA algorithm have presented improvements in [BVH04d] compared to MPA. But *Bent et al.* did not stopped here and proposed two strategies in 2007 in [BVH07]: the *waiting strategy* and the *relocation strategy*. These two strategies are announced to '*improve customer service, especially for problems that are highly dynamic and contain many late requests*'.

It should be noted that the algorithms of *waiting* and *relocation* strategies assume that there is only one vehicle in the fleet (which explains the notation of a plan as ρ) and that the plans inside Γ can contain *sampled* requests, contrary to what was depicted in the previous sections (they are not filtered out anymore before sending a plan to MSA). This implies that some new arrangements (not depicted here) must be done in MSA in order to respect its additional hypothesis.

The next sections present each one of the strategies in details, explaining their respective key idea, illustrate their functioning on a toy example and finally depicting their algorithms, applied on the *consensus* approach. We will also discuss the possibility to combine them.

Note that the improving strategies are decoupled with the approach chosen to implement CHOOSEREQUEST. This means that it is entirely possible to adapt the same strategies to the *regrets* approach or to *expectation*. This leads to a large variety of configurations, such as : **simple expectation**, **regret with relocation strategy**, or **consensus with waiting strategy**.

3.6.1 Waiting Strategy

The key idea behind the name of *waiting strategy* is that it could be valuable for a vehicle to **wait** at it current location for an hypothetical materialization of the sampled request placed

next on its route.

Note that route containing a sampled request as the next destination can be expressed as

$$\rho = \rho^- : \rho^+ \wedge \textsc{First}(\rho^+) \notin A_t.$$

where ρ^- denotes the elapsed part of the route ρ and ρ^+ denotes the remaining part, i.e. in the time interval $[t, H_l]$.

The difficulty with this strategy is being able to determine if the vehicle should wait, considering that Γ contains several solutions for different scenarios.
We will now show the modifications to bring to ChooseRequest with consensus in order to implement this strategy.

Implementation The pseudocode presented by *Bent et al.* in [BVH07] and depicted on Algorithm 10 is a derivative of the *consensus* approach introduced in section 3.2. A new `waiting` action (denoted by $\perp$) is added. Its consensus score is incremented by 1 each time that a plan in Γ contains a `sampled` request as the next destination of the route, regardless of its geographical position. Finally, as in the original *consesus* algorithm, the action with the highest score is decided and returned.

Algorithm 10 Implementation of ChooseRequest, using the **consensus** approach and the **waiting** strategy.

```
1:  function ChooseRequest-CW(ρ_t, σ_t, A_t, Γ)
2:      for r ∈ A_t ∪ {⊥} do
3:          f(r) ← 0;
4:      for ρ ∈ Γ do
5:          r ← First(t, ρ − ρ_t);
6:          if r ∈ A_t then
7:              f(r) ← f(r) + 1;
8:          else
9:              f(⊥) ← f(⊥) + 1;
10:     return argmax(r ∈ A_t ∪ {⊥}) f(r);
```

Illustration We now illustrate the functioning of this strategy in a simple but representative situation. Consider that all the plans in Γ have a sampled request at customer $C2$ as the next destination of vehicle v, when it becomes idle. A waiting vertex (or any representation of the fact that v should wait) at the current location is added to the route between the current request and the next one. After some time, two situations can occur: either a request was received from customer $C2$, or nothing happened until the *time window* of this request is violated and thus the sampled request was removed from the plans. In the first case, the wait was beneficial as the new request was integrated easily into the plans of Γ, increasing the overall score of accepted requests. In the second situation, nothing was lost as the plans in Γ are feasible and contain the same maximal number of accepted requests if no new request appeared.

The application of the waiting strategy is depicted on Figure 3.1.

3.6.2 Relocation Strategy

This improving strategy is the opposite of the previous one. The key idea behind the name of *relocation strategy* is that even if the next request on the route is a sampled request, the vehicle should depart for its location such that if it materializes during travel, its insertion is easy and

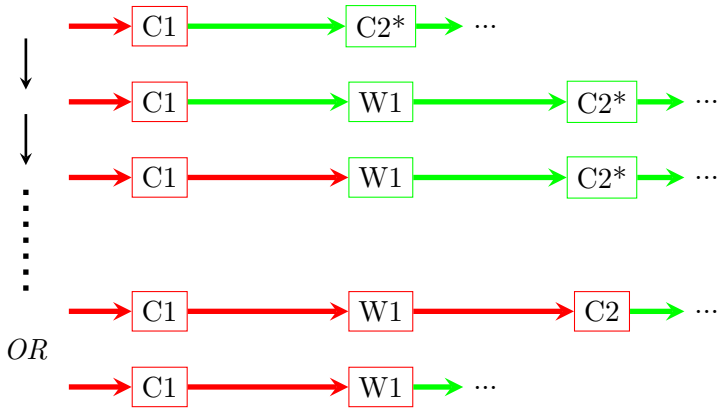

Figure 3.1: Application of the **waiting** strategy. The boxes are the vertices representing requests on a route, $C1$ and $C2$ are two customers of different locations. The 'WX'-labeled boxes are waiting vertices meaning that the vehicle waits at the location of customer X. The red part of a route represents its fixed part (*i.e.* the elapsed part of the day + the locked destination of the vehicle) and the green part represents the free part of the route. The waiting vertex added in the route is located at the current location, expecting the materialization of the following sampled request.

the request is processed quickly. The sampled and actual requests are thus not distinguished anymore, the vehicle moves to the customer location of the request with the best evaluation.

It is straightforward to understand that the *relocation* strategy is not advantageous when the main objective of the problem is to minimize the travel costs/times since the vehicles may move to locations without actual request to serve. But remind that in our case, we try to maximize the number of accepted requests (see section 2.2.3) in priority, so if we allow ourselves this sacrifice, we can benefit from the relocation strategy as it anticipates future requests and positions the vehicles in a way to serve them quickly.

We now show the modifications to bring to ChooseRequest with consensus in order to implement this strategy.

Implementation The pseudocode presented by *Bent et al.* in [BVH07] and depicted on Algorithm 11 is also a derivative of the *consensus*. This time, no new action is added, the only difference with Algorithm 4 is that the sampled requests are not filtered out anymore (compare line 6 on the original with line 5 of algorithm 11). And as in the original *consensus* algorithm, the customer/location (sampled or not) with the highest score is decided and returned.

Algorithm 11 Implementation of ChooseRequest, using the **consensus** approach and the **relocation** strategy.

```
1: function ChooseRequest-CR(ρ_t, A_t, Γ)
2:     for r ∈ Customers do
3:         f(r) ← 0;
4:     for ρ ∈ Γ do
5:         r ← First(t, ρ − ρ_t);
6:         f(r) ← f(r) + 1;
7:     return argmax(r ∈ Customers) f(r);
```

Illustration Consider the same situation as in the example for the *waiting* strategy. A waiting vertex is also added, but this time at the location of the sampled request. After some time, two situations can occur: either a request was received from customer $C2$, or nothing happened and

the vehicle finished traveling to the location of $C2$. In the first case, the travel was beneficial as the new request was anticipated: the request is integrated easily into the plans of Γ, one more request was accepted, and it is served quickly after its arrival (remind that the time window has to be respected). In the second situation, the vehicle becomes immediately *idle* once on site and then locks its next destination. The only loss is that the travel cost increased for nothing, but the plans in Γ are still feasible and contain the same maximal number of accepted request if no new request appeared, or if there was no sampled request at all.

The application of the relocation strategy is depicted on Figure 3.2.

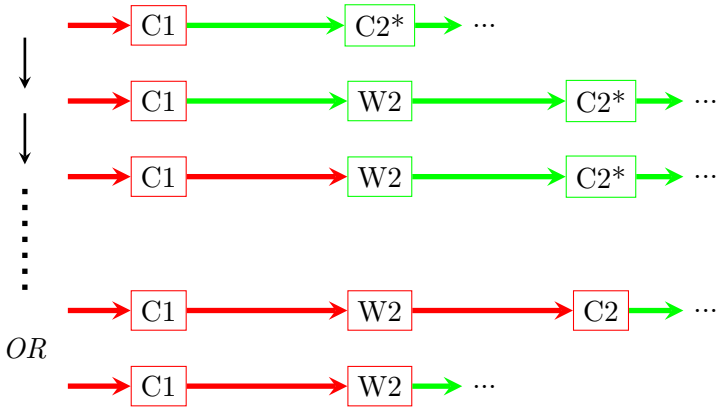

Figure 3.2: Application of the **relocation** strategy. The boxes are the vertices representing requests on a route, $C1$ and $C2$ are two customers with different locations. The 'WX'-labeled boxes are waiting vertices meaning that the vehicle waits at the location of customer X. The red part of a route represents its fixed part (*i.e.* the elapsed part of the day + the locked destination of the vehicle) and the green part represents the free part of the route. The waiting vertex added in the route is located at the position of the sampled request, whereas the *waiting* strategy used a waiting vertex at the current location of the vehicle.

Waiting plus Relocation combination

One can wonder if the two strategies we presented may be combined to profit their respective benefits and increase the performances further. Regarding their functioning, it seems that such a combination would be hard to develop. Indeed, their approach are fundamentally opposed, as the first prefers staying at its current location while the other tends to move permanently. Such a difference makes the simple idea of combining them quite foolish.

In a more technical spirit, as the *waiting* and *relocation* strategies increase the counters of different actions in the case when the next destination is a sampled request, a choice between increasing the counter of the $\perp$ action or increasing the counter of the sampled requests themselves must be done. One can imagine an approach that increases both types of counter but doing that would would be going back to the *waiting* strategy: If we denote $next\_R^*$ the set of sampled requests that are set as the next destination on the route in any plan of the pool, $n_{r_i^*}$ the number of plans where the next destination is a sampled request r_i^* and finally n_{r^*} the number of plans where the next destination is any sampled request, then we have the relation $n_{r_i^*} \leq n_{r^*}$ with equality if $|next\_R^*| = 1$. As this relation is true for all the sampled requests that are the next destination in some plans of the pool, the score of the *waiting* strategy (which is equal to n_{r^*}) will always dominate the multiple scores of the *relocation* strategy.
To conclude, there seems not to be a simple manner to combine *waiting* and *relocation* strategies and as the objective of the improving strategies was to bring small modifications to the known algorithms to increase the performances, we decided to avoid going further in the direction of this problem.

Chapter 4

Solving DS-VRPTW

The goal of this master thesis is to measure the respective efficacy of combinations of algorithms (MPA or MSA) with approaches to select a plan in the maintained pool (consensus/regrets), possibly using improving strategies (waiting/relocation). The purpose is thus to propose to the user an application that returns a sub-optimal solution for some given instance of *DS-VRPTW with stochastic customers*, by using the theory introduced by *Bent et al.* ([BVH04d],[BVH07],[BVH04a]...).

The main contribution of this chapter is to provide an actual framework to compute solutions to DS-VRPTW by using the MPA/MSA methods proposed by *Bent et al.*. This framework follows the descriptions depicted in their work, clearing the grey zones that lacked information and also brings new features as our framework can be used easily and is suited for parallelized computation to generate new plans.

First, section 4.1 presents the adaptations we performed on the state-of-the-art techniques MPA and MSA and that resulted in a single algorithm before building our framework. Second, we present the architecture of the framework itself in section 4.2 with regard to the generic structure of an instance of D-VRPTW described in section 2.3.1. Then, section 4.3 presents the different modules that make up the core component of the framework named the *Method* component. In this section, we also describe the interaction between these modules with regard to the algorithm we adapted in section 4.1 from the work of *Bent et al.* After that, sections 4.4 and 4.5 describe the main tool used as the *Offline Solver* and the implementation details of important parts of our framework, respectively.

4.1 Adaptations

Some adaptations had to be done in order to allow the algorithms proposed by *Bent et al.* along time to work together. This section is dedicated to present the adaptations we brought to make it possible.

First of all, no pseudocode was given in [BVH04d] to implement MPA and MSA, so a new algorithm was designed. It should be emphasized that we designed an algorithm that remains close to the original expression of MPA in this paper where the solution returned is not created incrementally but instead, is an actual solution selected from the pool of plans Γ. This implies that some modifications had to be brought to the state-of-the-art algorithms proposed by *Bent et al.* and presented in the previous chapter, to respect this.

The modifications concern the structures of the functions inside Algorithm 1 and Algorithm6, in order to make them compatible with a *plan selection* instead of a *request selection*. We

concentrate on the differences to bring to the latter algorithm, in order to obtain the final pseudocode we designed. We first highlight the differences between the algorithms, then we explain how we adapted those to the functioning of MPA and MSA.

First, the processing of events during online operations in MPA and MSA does nearly the same as the `for` loop in lines 5-21 of Algorithm 6:

- the *new request* events are captured in lines 6-7.
- the *new plan* events are captured in the single line 21.
- the *departure* events are captured by the `if` structure in lines 8-20 as a next destination is selected for each idle vehicle. This is the main source of divergence between the idea behind MPA/MSA (*plan selection*) and the others algorithms presented in chapter 3 (*request selection* with incremental construction of the final solution). This is the main part that must be modified.
- the *timeout* are simply not explicitly captured in the algorithm.

Thus we have two reasons to design an algorithm different than Algorithm 6: the integration of an explicit processing of the *timeout* events and the adaptation towards *plan selection*.

Integration of timeouts The first problem is easily solved: we simply design a new function *UpdatePlans_timeout* with the same behavior as presented in section 3.4.2.

The second modification is a little more complicated to solve.

Transition to plan selection First, note that the partial plan of Algorithm 6 is modified only when a vehicle becomes idle and chooses a next destination, whereas the *best plan* maintained by MPA/MSA is recomputed when *new plan* or *new request* events occur. We thus designed a function *Best_Plan_From*(Γ,t) that returns, given a time t, the **plan** with the highest score from approaches like *consensus* or *regrets*.

We thus had to adapt the implementations of ChooseRequest (using *consensus* and *regrets*, as those are the two approaches we implemented) to *plan selection*. This was done the following way:

Consensus The idea underlying this new version is the same as before: counting the number of plans in Γ where request r is served next on route of vehicle v. To adapt this to *plan selection*, we need a matrix M_t such that $M_t[v, r]$ denotes the size of the subset of plans in Γ where request r is served next on route of vehicle v.

$$M_t[v, r] = \#\{sol = (\gamma, \sigma) \in \Gamma \mid succ(LDC(v, sol), sol) = r\}$$

Where $LDC(v, sol)$ is the last request serviced and departed by vehicle v in sol and remind, $succ(r, sol)$ denotes the successor of request r in solution sol.

The best solution $sol = (\gamma, \sigma)$ from Γ is then defined as the one that maximizes the consensus function

$$f(sol) = \sum_{v=1}^{m} M_t[v, succ(LDC(v, sol), sol)]$$

i.e. the sum over the routes of scores of each request that is the next destination.

The complexity of this new representation is slightly different from the complexity of the old one (remind, the complexity was $O(|\Gamma| + |A_t|)$), partly because we consider problems with multiple vehicles. Actually, as the computation of the matrix is done by traversing each vehicle from each solution in Γ and the retrieval of the solution with the highest score is done over the plans whereas it was done over the requests before, the complexity hence became $O(m.|\Gamma| + |\Gamma|)$.

Regrets The adaptation of this approach is less straightforward as it requires two passages through the pool Γ. The first passage is used to gather the set of all the requests that could be serviced next on all the routes. We denote this set *Possible_Next*. This permits to retrieve the same set returned by READY$(\rho, \sigma, A_t, \Gamma, t)$ at line 7 of Algorithm 5.

The second passage finds the solution $sol = (\gamma, \sigma)$ that maximizes the following function:

$$f(sol) = \sum_{v=1}^{m} (w(sol) - \text{REGRETSUM}(sol, v, Possible\_Next))$$

where function REGRETSUM(sol,v,A), returns the sum of regrets (as described in section 3.2.4) between the current next destination of route v in *sol* and each request inside set A[1].

The change of complexity between this new representation and the old one for the *regrets* approach is roughly the same than when we modified *consensus*. Remember that the complexity of regrets with a single vehicle was $O(|\Gamma||A_t| + |A_t|)$. Here, REGRETSUM has a complexity of $O(|A_t|)$ as the next request of *sol* is swapped with each request in *Possible_Next* and this last set is at most equal to A_t. Also, as REGRETSUM is called m times (once per vehicle) for each plan in Γ to compute its score, and because the retrieval of the solution with the highest score is done over the plans, the resulting total complexity is $O(|\Gamma|.m|A_t| + |\Gamma|)$.

Once the two problems of *integration of timeouts* and transition to *plan selection*, some other minor changes were also performed to Algorithm 6. For example, we made more clear that computations are done when events occur (mainly for *new plan* and *new request* events) and we separated the processing of each type of events in a more explicit way. To tho this, we created the three functions *UpdatePlans_generation*, *UpdatePlans_request* and *UpdatePlans_departures* in addition to *UpdatePlans_timeout* introduced above. All these function affect Γ as explained in section 3.4.2 and as we said, the function *Best_Plan_From* is called for events types that require to recompute the best plan.

The pseudocode resulting from these modifications is depicted on Algorithm 12. This pseudocode was used as a basis for the program written to run the benchmarks and obtain the results visible in Chapters 5.

Lines 2-3 initialize the pool of plans and determines which of these solutions is the best by calling *Best_Plan_From*(Γ, t). This function applies the chosen approach (*i.e.* one of those presented in 3.2) and return the best solution in Γ at time t according to this approach. This initialization phase can be seen as the execution of an *Offline* VRP, which optimizes the problem knowing only the requests received before the start of the horizon.

Lines 4 to 15 simulate the progress of the day: for each time step of the horizon, the events that occurred in $[t-1, t]$ are processed as explained in section 3.4.2. The order in which the types of events are handled follows the priority described, but not motivated, in [BVH04d]: timeouts (line 5), new plans (lines 6-8), new requests (lines 9-13) and lastly, departures (line 14).

[1]Remind that the regret between two request is 0 if there exists a feasible swap (i.e. a swap that does not violate the time window constraints) between them and 1 otherwise.

Algorithm 12 Proposed pseudocode of the `MPA` algorithm

```
1: function OnlineMPA(⟨R_0, R_{H_e}..., R_{H_l}⟩)
2:     Γ ← Generate_Solutions(R_0)
3:     ⟨γ*, σ*⟩ ← Best_Plan_From(Γ, t)
4:     for t ∈ H do
5:         Γ ← UpdatePlans_timeout(Γ, t)

6:         for γ ∈ PlanGen_t do
7:             Γ ← UpdatePlans_generation(Γ, γ)
8:         ⟨γ*, σ*⟩ ← Best_Plan_From(Γ, t)

9:         for r ∈ R_t do
10:            Γ' ← UpdatePlans_request(Γ, r, t)
11:            if Γ'.size > 0 then
12:                Γ ← Γ'
13:                ⟨γ*, σ*⟩ ← Best_Plan_From(Γ, t)

14:        Γ ← UpdatePlans_departures(γ*_t, σ*_t, Γ, t)
15: return ⟨γ*, σ*⟩
```

The explanation behind this order is quite simple:

New Requests before Departures The reason is that the the arrival of a request r at time t may imply that the new *best* solution contains r as the next one on route v and the corresponding departure time to r is t. This highlights the fact that all the requests known at time t should be considered to process the departures.

New Plans before New Requests The plans received at time t were optimized at time $t-1$ (remind that an optimization at time $t-1$ optimizes the interval $[t, H_l]$) which means that a request r arriving at time t was not taken into account. As a consequence, if the new requests are processed before the new plans, there will be some plans in Γ that do not contain r, which violates the obligation for all plans in Γ to contain **all** the accepted requests (Hypothesis 3 of MPA and MSA).

Timeouts before New Plans The new plans generated at time t (and sent to MPA at $t+1$) take γ_t^* as a basis. This implies that each new plan is compatible with all the past decisions. It follows that none of the new plans would be filtered out when the timeouts are processed. So, some time is spared if these new plans are not processed by *UpdatePlans_timeout*.

These relative priorities leaves us with the aforementioned order. Finally, the last *best* solution (i.e. the best solution at the last time step of the horizon) is returned (line 16).

Now that we introduced the algorithm applying MPA and MSA in details, we can describe the general structure of the program written to obtain the results from Chapter 5, which is the subject of the next section.

4.2 Software Architecture

We based the design of our framework on the generic representation of a DS-VRPTW (depicted in section 2.3) that we slightly modified so that it corresponds our way to exploit stochastic information.
We solve problems implying *stochastic customers*, which allows us to add *sampled* requests to the solutions we generate. As the stochastic information affects the generation of solution only, a single box **Stochastic Generator** can be added to the diagram representing a *Dynamic Vehicle Routing Problem with Time Windows* (depicted in figure 2.3) in order to illustrate an instance using *stochastic customers*. This new box should take as input the probability distributions contained in the problem and interact with the central box when it needs to create new scenarios, before it calls the solver to generate a solution using this scenario.

The modified diagram representing the *Dynamic* and *Dtochastic* version of the VRPTW can be seen on Figure 4.1.

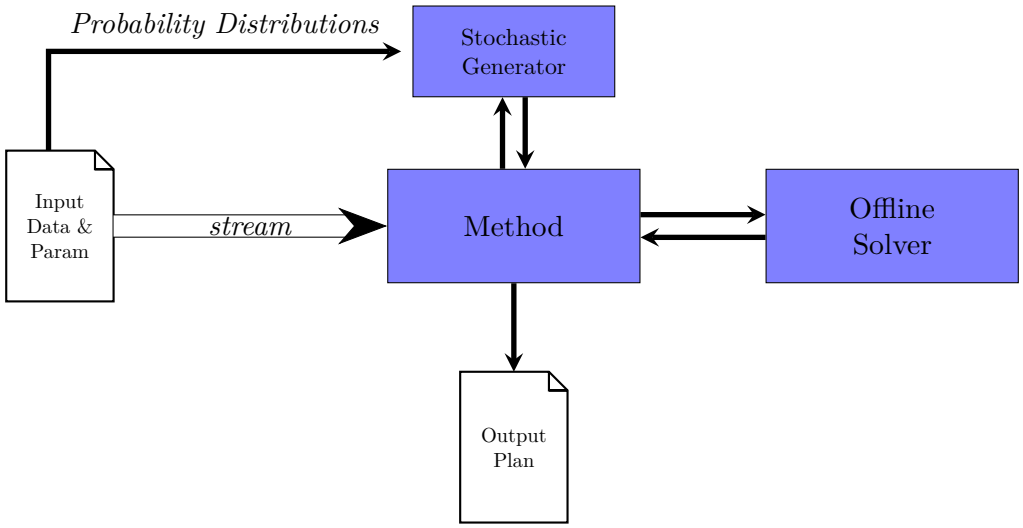

Figure 4.1: Diagram of a Dynamic and Stochastic VRP instance. In this new version, the `Method` box now interacts with the `Stochastic Generator`, that will return hypothetical requests named *sampled requests* during the *online* phase, according to some observed probability distribution of occurrence over time, received as input data at the beginning of the problem. These sampled requests are then integrated to the plans that are optimized by the `Offline Solver`.

The main requirement of this addition is the implementation of the *Stochastic Generator* visible on Figure 4.1. As we said, it is called by the *Method* box to add hypothetical requests to the known ones in order to build scenarios before the execution of the optimization. We now explain the functioning of the new *Stochastic Generator*, when a new scenario is created to be optimized by the *Offline Solver*.

Consider the situation of *Dynamic and Stochastic VRPTW* (DS-VRPTW) where the working hours (i.e. $[H_e, H_l]$) is divided into 3 equally sized *timeslots* (i.e. $ts = 1$ represents the first third of the working hours, $ts = 2$ represents the second third, ad $ts = 3$ represents the last).
The sampling is performed as follows:

- The user/instance gives a table containing the probability distributions over time of the customers at disposal of the *Stochastic Generator*. These distributions represent the

probability that a request from customer $c \in C$ arrives at time t. This table has $|C|$ rows and $\#timeslots$ columns, each entry $table[c, ts]$ containing the probability of occurrence of a request from $c \in C$ in timeslot ts.

- When called, the *Stochastic Generator* samples the probabilities corresponding to all the customers, in each unfinished timeslot.

- The requests that were sampled are returned.

If sampled, an hypothetical request is created and is added to the total set of requests:

$$Req = A_t \cup SampledRequests.$$

where, remind it, A_t represents the set of accepted requests that must be included in any solution produced by the *Offline Solver*, and Req denotes the scenario imagined for plan generation.

Once the scenario constructed, the optimization algorithm is run, taking all the requests inside Req into account. This optimization does not distinguish the sampled requests from the actual ones and try to serve them equally. Because of this, no modification should be brought to the *Offline Solver*, which simplifies the interoperability between implementations or the change of solver and/or stochastic generator.

4.3 The Method Component

We now describe the central component of the diagram we just introduced, the *Method* component that should implement MPA and MSA with the algorithm we proposed in section 4.1. It has to be admitted that the only few information about any framework is given in the work of *Bent et al*[HB09].

As we mentioned before, *Bent et al.* do not provide an actual pseudocode for MPA or MSA. But it is not the only grey zone as nothing is revealed about the generation of plans, the occurrence of the events or the way the new requests are inserted in solutions (we discuss the latter in details in section 4.5.4). We present here the design choices made to implement a program that solves DS-VRPTW, using the **Multiple Plan Approach** (MPA) and **Multiple Scenario Approach** (MSA). We first introduce the different agents implied in the design, followed by a description of their interaction.

The Agents

In our implementation, we designed three types of agents, the `EventHandler`, the `MPA/MSA` agent and the `PlanGenerator` agent:

EventHandler This agent is the liaison agent between the two others, it also links them with the external world. Formally, it contains the following elements:

- `Array data structures`: for each types of events, an array with length equal to the horizon size is created. In the $i-th$ cell of these arrays are contained the events of the corresponding type, occurring at time $t = i$.

- `Best Solution`: It also contains the current best solution, i.e. the routing plan that is currently followed by the vehicles.

Algorithm MPA/MSA This agent applies the chosen algorithm (MPA or MSA), using the pseudocode we presented in Algorithm 12. It receives the events that occurred in the current time step from the `EventHandler` and processes them as depicted in section 3.4.2. An additional step is added to Algorithm 12 in order to work with the other agents: after that all the events of a time step are processed, the best solution of the pool Γ is sent to the `EventHandler`.

PlanGenerator This agent is our implementation of the GenerateSolutions function used before. It calls the *Stochastic Generator* (if needed) and *Offline Solver* boxes in the diagram depicted on figure 4.1. Its functioning is the following: it receives the current best solution from `EventHandler` and generates new plans or scenarios, depending on which algorithm is used, MPA or MSA. Eventually, it returns the set of solutions created to `EventHandler` as *new plan* events.

Parallelization As we emphasized, we designed Algorithm 12 to remain close to the original expression of MPA in [BVH04d], but it has an other property that should be highlighted. This design also allows us to parallelize an important part of MPA/MSA, which is the generation of plans. It is easy to notice that some decision approaches such as *consensus* are expected to provide better results if the size of the pool is large. Indeed, larger and more diversified populations of Γ should imply that the probability distribution of choosing a plan (or a request) instead of an other is more natural, which leads to more informed decisions. So, in order to maintain a certain population in Γ, we ensured the possibility to fill the pool quickly by using multiple *PlanGenerator* agents in different threads to feed the *EventHandler* with lots of *new plan* events in each time step of the horizon.

Interaction during a time step We now explain the interaction between the threads/agents composing the *Method* box during a single time step. A representation of this interaction can be seen on figure 4.2.

In this figure, a new dashed box **Master** is shown. This box is an abstract representation of the central office that inputs the stream of information received during the *dynamic* phase of the problem. It is not an actual agent (which is the reason why it was not introduced before), but it is represented here to make the description clearer. The arrows representing the information passed from (in green) and to (in red) the *EventHandler* are numbered, this was done to emphasize the chronological order in which the actions occur. The progress of a time step t is the following:

1. The central office sends the new online requests received in $[t-1, t]$ to the EventHandler;
2. The EventHandler sends all the events (of all four types) gathered in $[t-1, t]$ to the implemented algorithm (MPA or MSA);
3. The events are processed successively in the order described in the previous chapter: **timeouts**, **new plans**, **new requests** and finally **vehicle departures**. The plan pool is updated accordingly and the last best plan computed from the pool is selected and the algorithm sends it back to the EventHandler;
4. The PlanGenerator instance(s) ask(s) the EventHandler for a basis to generate plans in the next time step $[t, t+1]$. Any plan in the pool could be used by a PlanGenerator, but as EventHandler contains the *best plan*, this is the one that is sent.
5. Each PlanGenerator generates as much plans as possible from the given one during the time step $[t, t+1]$. All the plans generated in this time interval are then sent to the EventHandler as *new plan* events.
6. The EventHandler sends the new best plan to the central office so orders can be given to the fleet.

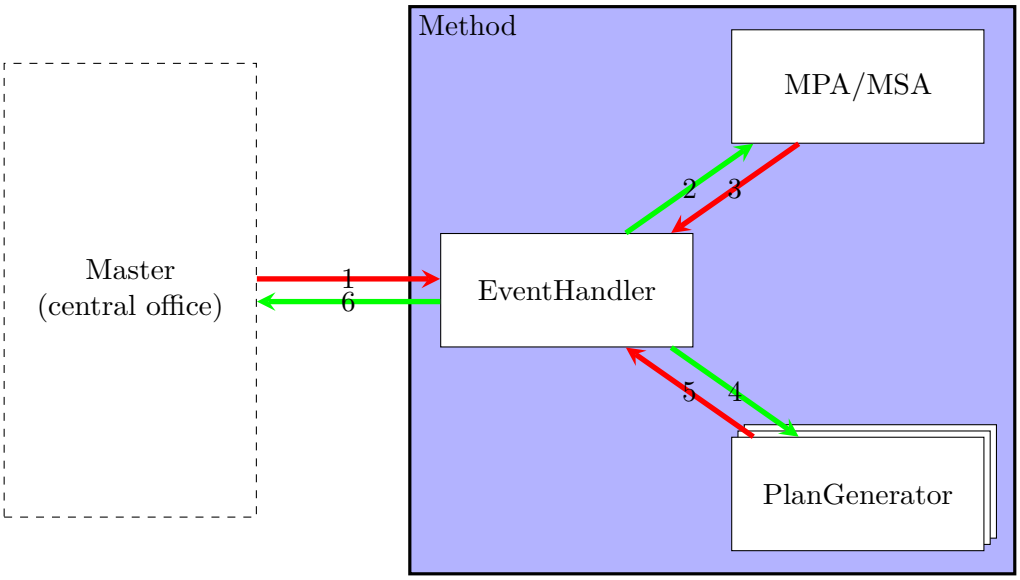

Figure 4.2: Diagram of interactions between the different agents in a single time step. (1)Requests received in $[t-1, t]$; (2)Events that occurred in $[t-1, t]$; (3)Best plan for this time step; (4)Plans that will be the basis for generation; (5)new plans based on the given one; (6)The plan that must be followed.

Once the time step is elapsed and the Master plan received the instructions, the algorithm increments the current time to $t+1$ and the next time step can begin with action 1 on figure 4.2.

We now introduce in the next section the main external tool used as a `Offline Solver` in our framework.

4.4 The Offline Solver

The most important external tool used in our framework concerns the *Offline Solver*, i.e. the implementation of $\mathcal{O}(state, t, R_t)$. We chose the `C++` library developed by Michael Saint-Guillain[SGDS15], assistant at the Université Catholique de Louvain (UCL), as it responds to all our needs. Because it was used as a black box, we do not describe the optimization algorithms in deep details. Instead, we describe the most important features it provides and the way it will be used in our application. A class diagram of this library can be found in Appendix A.

The core type of objects of this library the `VRP_Instance` type. These represent the instances of various types of VRPs such as the generic one, but also variants such as D-VRPs, VRPTW... It contains the properties of the graph G, the properties of the fleet, the horizon and everything that characterize the problem to be solved.

Several methods are provided to parse instances proposed by different actors in the sector of vehicle routing problems. For example, instance files proposed by *Cordeau et al.*[CLM01], or those used by *Bent et al.*, and others can be parsed by the library.

The requests characterizing an instance have a state inside `VRP_Instance` objects. This state is used in Dynamic problems as it determines if the request should be considered as occurred in the instance. A request is labeled as *(un)revealed* if the solutions solving the instance should (not) consider the request as arrived in the central office (whether the solution contains the request or not). This label changes only in dynamic problems as in static ones, all the requests are considered as occurred. We will use this label frequently as any *new request* will be accompanied by flipping

the attribute of the corresponding request in the `VRP_Instance` object from *unrevealed* to *revealed*.

This library uses `VRP_solution` structures to represent a solution object of a basic VRP problem. These structures contain all the information we described in section 2.2.2: they contain the `VRP_Routes` structure (corresponding to the routing plans γ we introduced), themselves containing vertices (`VRP_Vertex`) possibly hosting requests (`VRP_Request`). 'Possibly' is used because it is also possible to add other types of vertices such as *waiting* vertices, as we will use them to implement the improving strategies we presented in section 3.6. The information we need in the σ structure we introduced is also present in `VRP_solution`: the attributes inside the `VRP_Solution` objects contain the information concerning the planned departure times of the corresponding vehicle from the vertices locations.

Extensions to the `VRP_solution` objects are also present inside the library, in order to solve variants of the generic VRP. The extension of those objects we are the most interested in is the `DS-VRP-TW_Solution` objects as it handles *Dynamic* problems using *Time Windows* and *Stochastic Customers*.

- **Time Windows:** The consideration of time windows is handled in `DS-VRPTW_Solutions` objects, with the availability of evaluation and differentiation methods and functions checking the violation of time windows.

- **Dynamic aspect:** The dynamic aspect of such solutions is handled by the availability of different methods allowing time management (e.g. set the solution at a given time to inspect the associated elapsed/remaining parts of the horizon), the insertion and removal of online requests, checking the feasibility of such insertions... The part of a solution that happens before its *current time* is called the **fixed part** of the solution and the remaining is called the **free part**. Note that the fixed part of the routes also contains the request(s) the vehicles are currently traveling to.

- **Stochastic aspect:** `DS-VRPTW_Solution` objects can also contain *sampled* requests (i.e. requests that are not currently set as revealed inside the *VRP_Instance*). Such a request is treated as an actual one, meaning that all the actions allowed on normal requests can be applied to sampled ones.

The `VRP_Solution` objects and their extensions are optimized by a **Local Search** heuristic that can be affected by parameters such as *termination conditions* or *optimization time* (timeout). The `LS_Program` uses a *Neighborhood_Manager* to implement the neighborhood operators needed for intensification/diversification such as *swap*, *cross-exchange*, *relocate* and *inverted 2-Opt* whose descriptions can be found in [KS97] and [TBG+97].

`VRP_Solution` objects have an attribute, named *waiting strategy*, that can take several values such as **DRIVE-FIRST**, **WAIT-FIRST**, **DRIVE-FIRST-DELAY-LAST** or **CUSTOM-WAIT**.

- **DRIVE-FIRST** indicates that the solution is constructed so that a vehicle leaves its current location as soon as it finished servicing the customer.

- **WAIT-FIRST** indicates that the solution is constructed so that a vehicle will wait a maximum of time once the service finished before leaving its location. The maximum time corresponds to to the latest time step in the horizon such that the time window of the next request is not violated. This should not be confused with the improving strategy named *Waiting strategy* from section 3.6.1.

- **DRIVE-FIRST-DELAY-LAST** is a combination of the two preceding. It applies the *DRIVE-FIRST* strategy for all the requests in a route except the last one before the return to the depot. This means that once the last request from a route is serviced, the corresponding vehicle will wait until the last time that permits to get back to the depot by the end of operations before leaving its position, which correspond to the application of the *WAIT-FIRST* strategy.
- **CUSTOM-WAIT** is similar to *DRIVE-FIRST-DELAY-LAST* but also allows controlling the waiting times at each vertex with additional functions.

By looking at these descriptions, one would naturally pick *CUSTOM-WAIT*, but if we remember Hypothesis 5 of MSA introduced in section 3.5 that ensures that no sampled request should be serviced, we realize that the solutions that populate Γ must be treated so that **no sampled request can be in their fixed part**. Using *CUSTOM-WAIT* when implementing the *relocation* strategy would violate this hypothesis. Also, remind Hypothesis 3 from section 3.4.1: it was defined that a vehicle departs for its next destination as soon as it becomes *idle*. So, we decided to use the *DRIVE-FIRST-DELAY-LAST* strategy in our implementation as it is the closest to the algorithms described by *Bent et al.* in their work that allow us to use a single strategy for all configurations of methods (MPA/MSA), approach (consensus/regrets) and improving strategies (waiting/relocation). We chose *DRIVE-FIRST-DELAY-LAST* over *DRIVE-FIRST* to avoid unnecessary travels from and to depot during the horizon.

This *waiting strategy* attribute should not be confused with the improving strategy we presented in section 3.6. It determines the way a solution should be constructed/optimized by the optimization functions of `LS_Program`, whereas the improving strategy *waiting strategy* takes sampled request from the solutions in the pool Γ into account to take more informed decision about choosing the next places that should be visited by the vehicles.

Programming language In order to avoid multiplying programming language and facilitate the interaction of the agents designed with the `C++` library to manipulate the solutions, we decided to write the entire program in the same programming language: `C++`. From the choice of using this library, some additional modifications had to be done to execute the intended behavior. The next section focuses on the description of the most noticeable ones.

4.5 Implementation Details

We now present noticeable parts of the implementation that were adapted to the usage of the library. The first part tackles the implementation of the functions that generate new plans/scenarios, then the second address the implementation of the improving strategies *waiting strategy* and *relocation strategy*. Finally, the two last adaptations concern the consensus matrix M_t as it explains how its definition was modified to be compatible with an implementation using the waiting vertices provided by the library, and the way arriving requests are inserted inside the existing solutions in Γ.

4.5.1 Plan Generation

The solutions are generated during each time interval $[t, t+1]$ corresponding to each time step and are sent to the *EventHandler* as *new plan* events at time $t+1$. Also, we stated that the current best solution inside the pool Γ was used as a basis to create a new solution, which means that plan generation can thus not be performed during *event processing*.
We thus synchronized the MPA/MSA thread with PlanGenerator thread(s) by using an integer

variable `time` containing the current time and a boolean variable `ok2gen` whose value is modified in the implementation of MPA and that is true iff the event processing is done and the MPA-thread is sleeping for a duration equal to a time step. Once the time step is finished, the MPA-thread awakes, and increases the current time in variable `time` by 1.

This way, as long as `ok2gen` is true and the value of `time` is not incremented, PlanGenerators optimize new scenarios and send them to the `EventHandler` as follows:

1. The current best solution is reached and copied.
2. An optimization duration strictly inferior to a time step is generated.
3. If MSA is used (a new **scenario** must be generated, not simply a plan), sampling is performed to include hypothetical requests to the solution to be optimized.
4. The optimization is run for the duration generated at step 2.
5. If the time step did not change during step 4 and the optimized solution is **feasible**, it is sent to the `EventHandler` as a *new plan* event.

The `C++` source code corresponding to these steps can be found in Appendix B.1.

4.5.2 Implementation of the improving strategies

As the improving strategies concern only the moments when some vehicle departs a customer, the natural place to apply them is inside the methods processing *vehicle departure* events, just before executing the behavior described in section 3.4.2. This is the reason why we created the function `ApplyStrategy` that takes the pool of plans and an integer representing the improving strategy to apply as arguments.

The behavior of this function will vary according to the chosen improving strategy, if any. We first describe the behavior of the implementation if none of such strategies is applied (from now, we call this configuration *Simple-MSA*) and then the respective behaviors of the *waiting strategy* and *relocation strategy*.

Simple-MSA

The application of Simple-MSA is may seem quite trivial, as the sampled requests in this configuration should be simply removed from the plans, according to the theory presented in section 3.5. But because our implementation was designed so that a single implementation of the plan generation would work for all the configurations (with and without improving strategies), some slight changes had to be performed.

The main change is that the removal of the sampled requests is executed during the *appyStrategy* phase. This removal concerns all the sampled requests that are following the current locations on a route. This means that the sampled requests after the next actual request on a route are kept in the solution. This behavior is depicted on figure 4.3.

This design choice was done because adding an actual request to a solution containing the corresponding sampled request is faster than adding a node to the routes, as it consists only of flipping the *revealed* attribute of the request. Because of this, we decided to keep a maximal number of sampled requests in our solutions and remove only those that would violate the fifth hypothesis of MSA.

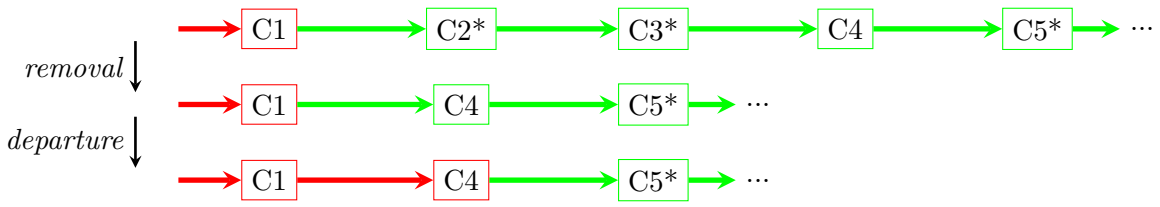

Figure 4.3: Removal of sampled requests next to the current location of a vehicle. The sampled requests in positions further the first next actual request are kept in the route. This way, Hypothesis 5 is respected as a vehicle can only move towards locations of actual requests.

Waiting Strategy

The implementation of this strategy is close to the one depicted on figure 3.1. Indeed, *ApplyStrategy* using waiting strategy adds waiting vertices with duration equal to a time step (provided in the `C++` library we use) as next nodes in the routes where the vehicle should depart to a sampled request. These waiting vertexes are associated with the current location so the vehicle does not start traveling.

Then, as long as the next request is a sampled request, the duration of the *waiting vertices* is increased with an additional time step, which is the behavior intended in [BVH07]. If a solution remains feasible with an additional step of wait, nothing more is done. Otherwise, i.e. if an increase causes *infeasibility* of the solution, the sampled request is removed as its time window is violated. The second case and its resolution are not depicted in [BVH07], but the behavior we implemented seems natural to solve the situation. Indeed, this restores the feasibility of the solution, without affecting actual requests, and thus the **service guarantee** is conserved.

The `C++` code corresponding to this behavior can be found in Appendix B.2.

Relocation Strategy

The implementation of this strategy is also close to the one depicted in section 3.6.2 on figure 3.2. *ApplyStrategy* using relocation strategy adds waiting vertices with duration equal to zero as next nodes in the routes where the vehicle should depart to a sampled request. These waiting vertices are associated with the location of the sampled request so the vehicle starts traveling.

Then once the vehicle arrived at the location of the waiting vertex, two situations may occur. If the sampled request was not concretized during travel, then the node representing this request is removed. Otherwise, nothing is modified. Now, remind what we said in section 3.6.2 about the two situations possible after the removal: either the next node on the route is an actual request or it is a sampled request.

the first case, it is the standard situation, the vehicle will thus depart its current location and travel to the next.
In the second case, if the sampled request serves another customer, it is the situation where a relocation should be performed: a new *waiting vertex* will be added at the location of the next sampled request as the next node and the vehicle travels in its direction. Otherwise, i.e. if for some reason an other sampled request at the same location is the new next node on the route, it is also removed as the vehicle is already on site.

The `C++` code implementing this behavior can be found in Appendix B.3.

4.5.3 Consensus Matrix

The third noticeable adaptation we had to do in order to work with the waiting vertices provided in the library concerns the *consensus matrix* of the next destination we introduced in section 4.1. Indeed, because we use *waiting vertices* to perform *plan selection* instead of the new $\perp$ action introduced by *Bent et al.* in [BVH07], and because we wanted to design a single implementation that would work with all the strategies, the definition of the *consensus matrix* had to change some more.

Remind that the definition we gave in section 4.1 was the following:

$$M_t[v, r] = \#\{sol = (\gamma, \sigma) \in \Gamma \mid succ(LDC(v, sol), sol) = r\}$$

where r is a request in the set A_t containing the accepted requests, v is the index of a vehicle, *sol* is a solution in the pool. Also, $LDC(v, sol)$ is the last request serviced and departed by vehicle v in *sol* and succ(r,γ) denotes the successor of request r in routing plan γ.

The new definition we follow in the implementation of our program does not take a request as the second dimension of matrix M_t. Instead, we use the sites $s \in S$ of the instance, which leads to the next definition for the consensus matrix:

$$M_t[v, s] = \#\{sol = (\gamma, \sigma) \in \Gamma \mid cust(succ(LDC(v, sol), sol)) = s\}$$

Using this definition, the $\perp$ action is not needed anymore as the score that it would have obtained from vehicle v is in fact contained in the entry $M_t[v, cust(LDC(v))]$. This is because this entry increases its score only when a routing plan has a *waiting vertex* associated to the current location of vehicle v, which is the intended score of the $\perp$ action of *Bent et al.*.

Concerning the *relocation strategy*, this definition performs well. To prove this, think that the score of a customer location s for a vehicle v is equal to the sum of the scores of the requests coming from this customer and that are serviced next on the route of vehicle v, sampled or not; i.e. $M_t[v, s] = \sum_{r:cust(r)=s}(score(r, v, \Gamma))$. Also, as having a request at location s, whether it is a sampled request or not, would require a displacement of the corresponding vehicle, it seems natural to combine these scores.

To illustrate how natural it is, imagine the situation where there is one route in the plans and three requests from two customers battle to be chosen as the next. Let's denote r_{11} and r^*_{12} the two requests coming from the same location s_1 with r_{11} an actual request and r^*_{12} a sampled one, and r_{21} an actual request coming from customer location s_2. Say that the scores of these requests are such that

$$score(r_{11}) < score(r_{21}) \bigvee score(r^*_{12}) < score(r_{21}) \bigvee score(r_{11}) + score(r^*_{12}) > score(r_{21})$$

In this situation, one of the plans servicing r_{21} would be chosen as the best plan because the corresponding score is higher, whereas more plans in the pool ask for traveling to s_1. This would break the *consensus principle* we mentioned in section 3.2.3.

4.5.4 Request Insertion

We now address the implementation of a grey zone in the work of *Bent et al.*, which is how the new requests are inserted in the solutions of the pool Γ when they contain *sampled* requests. The first paper ([BVH04d]) did not suffer from any problem as the sampled requests were removed from the solutions when adding it to the pool, which means that new requests were added without problem. But the following publications that involved improving strategies ([BVH07][HB09]) are affected by an issue: is there a hierarchy between sampled requests present in the solutions and

the actual requests arriving in the *Dynamic* phase, when we try to insert a request of the latter type?

This question is hard to answer as preferring actual requests and removing the sampled ones would increase the score of the objective function, which is good, but it implies denying the scenarios we build, and reducing the impact of the improving strategies on the computations. On the other hand, it may seem foolish to refuse actual requests because of a too high confidence in the scenarios containing sampled request corresponding to low probability of occurrence.

Because we found this question interesting, we implemented two different behaviors when it is time to try inserting a new arriving request (upon *new request* events):

- The first behavior acts like both sampled requests and actual requests were strictly equal and thus when adding the *new request* to a solution makes it **infeasible**, it is simply removed to restore feasibility and the next solution is tested.
- The second estimates that actual request have a higher value than the sampled ones and thus if a new arriving request could not be inserted in a solution when using the previous approach, a second try is performed on a copy of the solution where all sampled requests were removed. If this *second test* is successful, i.e. if the new request was inserted in the copy without it became infeasible, then the copy replaces the original in the pool of solutions.

Finally, as we know that keeping solutions containing only actual requests seems contrary to the basic principles of MSA, we ensured that only one of such a solution may exist in Γ. To do so, we ensured that if a solution that successfully passed the **second test** (the new request was inserted only when the sampled requests were removed), it is added to the pool **iff the pool is empty**, otherwise it is dropped .

We added a parameter to our application so that the value passed executes one behavior or the other, determining if the actual requests should be preferred (i.e.second test should be executed). The experiments we will present in the next chapter compare configurations using this parameter (we will refer to as '*actual requests preferred*' or ARP) both enabled and disabled to determine its impact on the objective function.

The `C++` code implementing the insertion of requests can be found in Appendix B.4.

Chapter 5

Experimentations

This chapter is dedicated to some experimentations done on the solution framework we developed for Dynamic and Stochastic Vehicle Routing Problems with Time Windows (DS-VRPTW). The goals of these experimentations are both to measure the impact of the usage of different approaches/improving strategies combinations on the objective function we defined in section 2.2.3, and assess our design by comparing the behavior of our results with those returned by test instances provided by the literature. These experimentations can also be seen as source of suggestions for improvements that can be done in further work.

First, section 5.1 presents the testing environment we used to gather our results by introducing the test instances, the parameters used by the application to perform our tests and the machine on which our applications was executed. Second, sections 5.2 and 5.3 present the experimentations we made, the results obtained and the interpretation of these results.

5.1 Benchmarking Setup

5.1.1 Instances

To get the results we present in this chapter, we executed our program on the same instances as those used in [BVH04d] and [BVH07] to compare the performances of the different approaches and improving strategies in term of refused requests.
These instances contain $n = 100$ customer locations in addition to the depot, distributed on a map as depicted on figure 5.1.

The instances are divided into classes that differ either by their respective **Degree of Dynamism** (DOD) a metrics introduced by *Larsen et al.* in [LMS02], or by the time slot when the *online* requests tend to arrive. The *degree of dynamism* metrics measures the ratio of *dynamic* requests (requests received during the *online* phase of the problem) relative to the total number of requests in the instance. More formally, the *Degree of Dynamism* is defined by:

$$DOD = \frac{\text{Number of dynamic requests}}{\text{Total number of requests}}$$

The value of the average DOD for each class of instance can be found in Table 5.1.

	Class 1:	**Class 2:**	**Class 3:**	**Class 4:**	**Class 5:**
average DOD	44%	44%	44%	59%	81%

Table 5.1: Average degree of dynamism (DOD) of the different instance classes

Figure 5.1: Map of customer region locations of the instances used for benchmarking around the depot. The filled circle ($X = 40, Y = 50$) denotes the location of the depot while the empty circles correspond to customer locations.

On this table, you can see that the class 1 through 3 have the same value of average DOD. These classes are distinguished by the time slot when the *online* requests arrive. Indeed, the durations of the problems (i.e. the time windows of the depots) in the instances provided by *Bent et al.* are divided in four periods named *time slots* or *time bins* corresponding to the known requests (the static part of the instances), morning, early afternoon and late afternoon. Note that in the instances of any class, no requests arrive in the *Late Afternoon* time slot.

Class 1 is characterized by early arriving requests, i.e. arriving during *Morning* time slot and **Class 2** by more late arriving requests (during the *Early Afternoon*). Besides this, *Class 3* is more balanced with an equal number of online requests arriving during *Morning* and *Early Afternoon* but keeping the same DOD as the two preceding classes. Finally, **Class 4** and **Class 5** instances have a higher proportion of late requests in addition to increasing the average DOD further. Note that **Class 5** was created for comparison implying improving strategies, so experiments that do not compare configurations using *waiting* or *relocation* strategies with others will not include measures on instances from this class.

We now describe how an instance is written.

Presentation of an Instance

Every instance we used in our benchmarks is composed of three parts:

1. The first part announces the characteristics of the instance such as the **number of customers**, the **number of available vehicles**, the **capacity of the vehicles** (the fleet is homogeneous), the **number of time bins** and finally, the description of the proper characteristics of each customer. The description of such characteristics is depicted on figure 5.2.

   The characteristics contain the position of the site (**X** and **Y**), the maximal **Demand** of the customer in this instance (distributed among all the requests it sends), the **Start**, **End** and **Service** duration times of the customer's time window. The three last columns

Cust	X	Y	Demand	Start	End	Service	PreBin	0−160	160−320	320−480
D	40	50	0	0	240	0	0	0	0	0
0	25	85	20	145	175	10	50	40	10	0

Figure 5.2: Characteristics of the depot and a customer.

provide the stochastic information we exploit. They give the probability distribution of occurrence of requests from the corresponding customer along the time slots of the instance. For example, customer 0 on figure 5.2 has a probability of 50% to request service during the offline phase (**Prebin**), 40% to request service during the *Morning* phase, 10% to request service during the *Early Afternoon* phase, and 0% to request service during *Late Afternoon*.

2. The second part of an instance file describes the requests known before the beginning of the dynamic phase of the problem, i.e. before the beginning of the time window of the depot. The form of such a known request is depicted on figure 5.3. On this figure, the request is characterized by the customer it arrives from, the time it arrives to the central office (note that the **Arrival** time is −1 because these are the known requests), its time window, its demand and finally the service duration once on site.

Known Requests 57

Cust	Arrival	Start	Deadline	Demand	Service
1	−1	50	80	30	10

Figure 5.3: Characteristics of the known requests.

3. The last part of an instance file describes the **dynamic** requests of the instance. The format of these dynamic requests is the same as the known requests except that the value of **Arrival** will be strictly contained inside the time window of the depot (strictly positive).

Unknown Requests 37

Cust	Arrival	Start	Deadline	Demand	Service
23	10	148	178	10	10

Figure 5.4: Characteristics of the *dynamic* requests.

5.1.2 Parameters of the Application

The application we wrote takes a few arguments in addition to the name of the instance file. These arguments and their roles is described on figure 5.5. The message visible on this figure appears on the user's terminal when the command `./main -help` is run in the `vrplib/` directory.

These parameters are used either to determine the approach to run and the improving strategy if any (parameters `-m,-i,-a`), or are other meta-parameters. These meta-parameters define the size of the pool before the online phase, the number of plan generators that feed the EventHandler with new solutions and the behavior to adopt when inserting a new request in the solutions of the pool (parameter `-arp`, see description in section 4.5.4)... Note that all the parameters must be set to run the program.

The last parameter (-hf) was created because the first experiments we computed returned results (visible in Table 5.3) such that the MPA configuration refuses only a few dynamic requests in average and thus it would be hard to have significantly better scores for configurations that are expected to outperform MPA. We thus decided to make the instances harder by giving the possibility to reduce the number of vehicles available to solve the problem. This can be done by enabling the *Half-Fleet* flag. In the next experiments, we decide to always enable the '-hf'

```
Usage: ./main instance_file METHOD IMPROVING-STRATEGY APPROACH TIME-STEP PG TIME/PLAN INIT
    -POOL-SIZE MAXPLANS ACTUAL-REQUEST-PREFERED HALF-FLEET

METHOD(-m):
        0: MPA
        1: MSA
IMPROVING-STRATEGY(-i):
        0: SIMPLE_MSA
        1: RELOCATION_STRATEGY
        2: WAITING_STRATEGY
APPROACH(-a):
        c: CONSENSUS     r: REGRETS
TIME-STEP (-ts) AND TIME/PLAN(-tp): a double number of seconds
PG (-g): number of plan generators (<5)
INIT-POOL-SIZE(-ip): an int number of plans at initialization
MAXPLANS(-mp): an int number for maximal size of the pool
ACTUAL-REQUEST-PREFERED(-arp):
        0: sampled requests considered equal to actual requests,
        1: actual requests considered more important.
HALF-FLEET(-hf):
        0: if the complete fleet should be used for the instance,
        1: if only one half of the fleet should be used.
```

Figure 5.5: User guide of the application. Gives a list of the parameters to provide to the application, each accompanied with a brief description and listing of the values it can take.

parameter so we expect more separate scores between the different configurations.

Now that we introduced all the parameters and meta-parameters, we introduce the way we denote the different configurations in the tables containing the results of our experiments. In these tables, the title of each column refers to a configuration which is a combination of values of six parameters we introduced before. The way a configuration is constructed is depicted in Table 5.2.

Method	Plan Selection Approach (in subscript)	Improving Strategy (for MSA only)	# Plan Generators	Actual Request Prefered	Half Fleet
MPA	c (consensus)	W (waiting)	1PG	ARP	HF
MSA	r (regrets)	R (relocation)	2PG	_	_
		_	4PG		

Table 5.2: Denotation guide for the tested configurations. The denotation follows the parameters from left to right, choosing one value in each column. A parameter that is not set (_) is not written in the denotation.

In this table, the first row gives the name of the parameters that define a configuration, in the reading order, while the remaining rows define the possible settings for each parameter. For example, the configuration using MPA with *regrets* approach, two plan generators, that prefers the actual requests over the sampled ones, and with a reduced fleet would be denoted by **MPA$_r$-2PG-ARP-HF**, whereas the configuration using MSA with *consensus* approach, the *waiting* improving strategy, one plan generator, that considers actual requests and sampled ones as equal, and with a complete fleet would be denoted by **MSA$_c$-W-1PG**.

Now, if we get back to Table 5.3, one can see that the results provided by the MSA configurations do not provide a stable improvement of the objective function and often provide worse results, which is also the case in the results given in the literature ([BVH04d]), but another possible origin of this observation is addressed in the discussion of the experiment we present in section 5.2.

	MPA_c-1PG	MSA_c-1PG-ARP	MPA_c-1PG
class 1			
0-100-rc101-1	2.40	5.80	5.40
0-100-rc102-1	6.20	8.20	10.80
0-100-rc104-1	2.60	3.20	10.60
class 2			
100-0-rc101-1	2.80	2.40	4.20
100-0-rc102-1	2.40	5.60	4.40
100-0-rc104-1	9.00	9.40	14.80
class 3			
50-50-rc101-1	4.00	3.60	4.60
50-50-rc102-1	4.40	4.60	7.80
50-50-rc104-1	9.80	6.00	11.20
class 4			
20-20-60-rc101-1	1.40	4.20	6.80
20-20-60-rc102-1	1.60	3.40	6.20
20-20-60-rc104-1	18.80	15.60	18.00
class 5			
10-10-80-rc101-1	2.3	2.4	7.5
10-10-80-rc102-1	2	2.5	6.3
10-10-80-rc104-1	21.9	23	29.1

Table 5.3: Average number of requests refused for three instances per class, computed for three configurations using one plan generator and the *consensus* approach: MPA, MSA and MSA preferring actual requests. The entire fleet was used in the solutions.

5.1.3 Testing Machine

The experiments we present in the next sections were performed on a machine running version 5.4.0 of `gcc` on a system with Ubuntu 16.04 LTS, using an Intel(R) Core(TM) i7-3610QM CPU @ 2.30GHz with 4 physical cores and 4 HT logical cores, and 8 GB of RAM.

5.2 Increasing the size of the pool

Description The first experiment we performed implied increasing the number of plans generated per time step to determine the actual influence of the size of the pool Γ on the objective function (reminder: maximize the number of serviced customers, then minimize the travel distance). To do this, we measured the average number of **refused requests** on our test instances with configurations using the *consensus approach*, but differing with the number of **plan generators**. We tested configurations involving 1, 2 and 4 threads generating plans, creating respectively approximatively 6, 15 and 30 solutions per time step of 3.5 seconds. We took our measures on three instances per class, for each class **1** through **4**.

Due to time constraints (a single run of an instance in a given configuration takes more than half an hour to simulate the complete working hours) and because of the highly dynamic aspect of the problem, we decided that instead of having a high number of configurations to compare, each instance would be run a higher number of times (here 10 times). Because of this, only the *consensus* approach will be used.

Parameter values We now give parameters that have common values for all the compared configurations.

- The approach (parameter '-a') is set to *consensus*.
- A time step (parameter '-ts') is set to 3.5 seconds.
- The initial size of the pool (parameter '-ip') is set to 50.

- The maximal size of the pool (parameter '-mp')is set to 400 solutions to avoid using too much memory.
- The fleet is reduced (parameter '-hf' set to 1).
- The optimization time for a single plan (parameter '-tp') is set to 1 second.

The others parameters, i.e. the number of plan generators (parameter '-g'), the choice between MPA/MSA (parameter '-m') and preferences about request types (parameter '-arp'), vary from a configuration to another.

The results of this experiment are shown on Table 5.4.

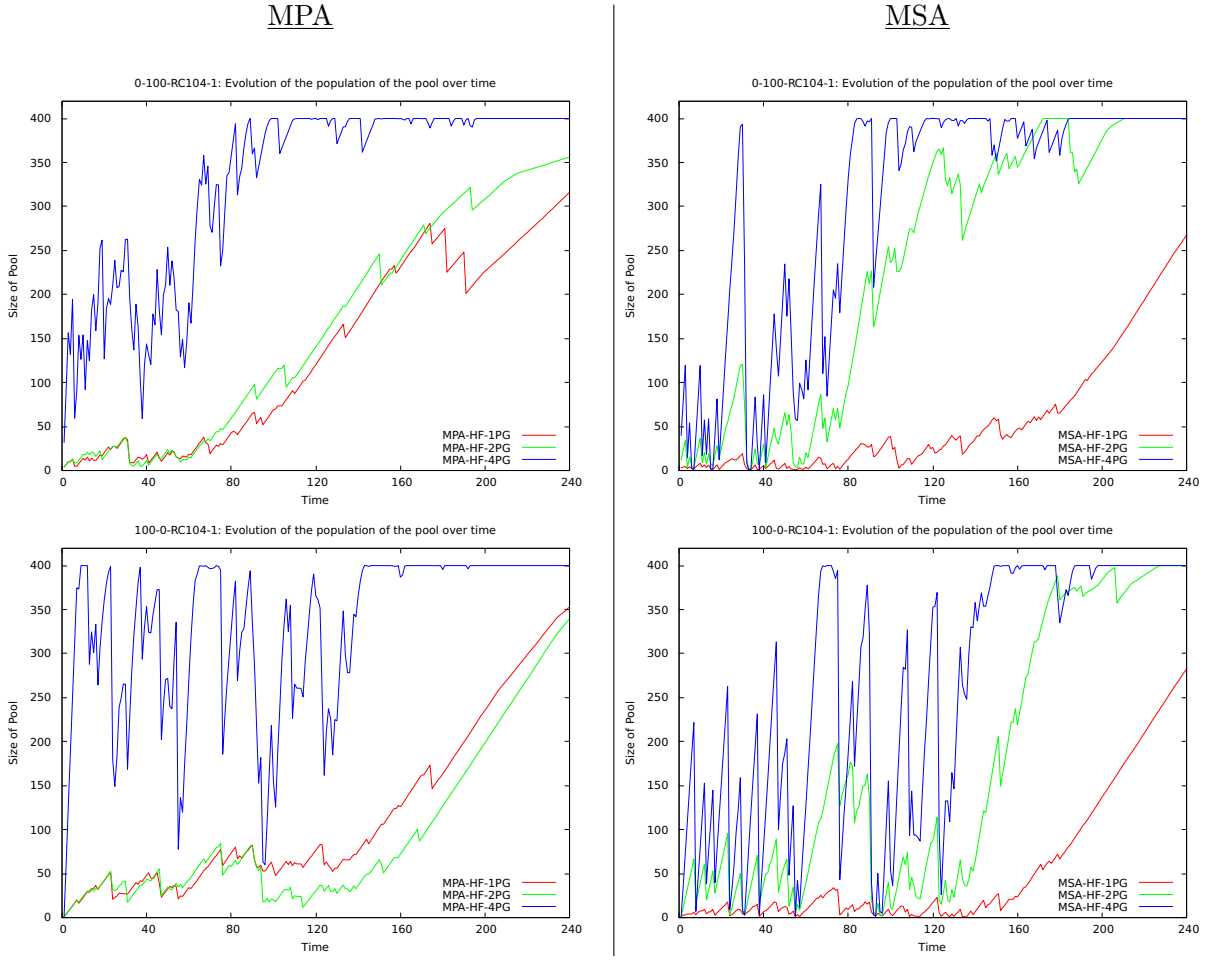

Figure 5.6: Evolution of the population of the pool of plans Γ over the horizon for instances `0-100-RC104-1` of class 1 (top frames) and `100-0-RC104-1` of class 2 (bottom frames), with a maximum of 400 solutions. The frames on the left use **MPA** with *consensus* approach and the right frames use **MSA** with *consensus*. On each figure, the configurations using one *PlanGenerator* thread are represented in red, those using two of them are represented in green, and those using four generators are denoted in blue.

Discussion On this table, the first thing one can observe is that the results of MSA_c configurations tend to be worse than those of MPA_c configurations, as in the experiment where the fleet was not reduced. Because of this, we will discuss on those two families separately.
We first focus on results from the MPA family. We can see on the three leftmost columns that the transition from 1 to 2 then 4 plan generators did not bring significant gain as the results may become either better or worse when increasing the number of *new plan* events, depending on the instance.
The same observation can be done for the MSA family as the the average number of refused

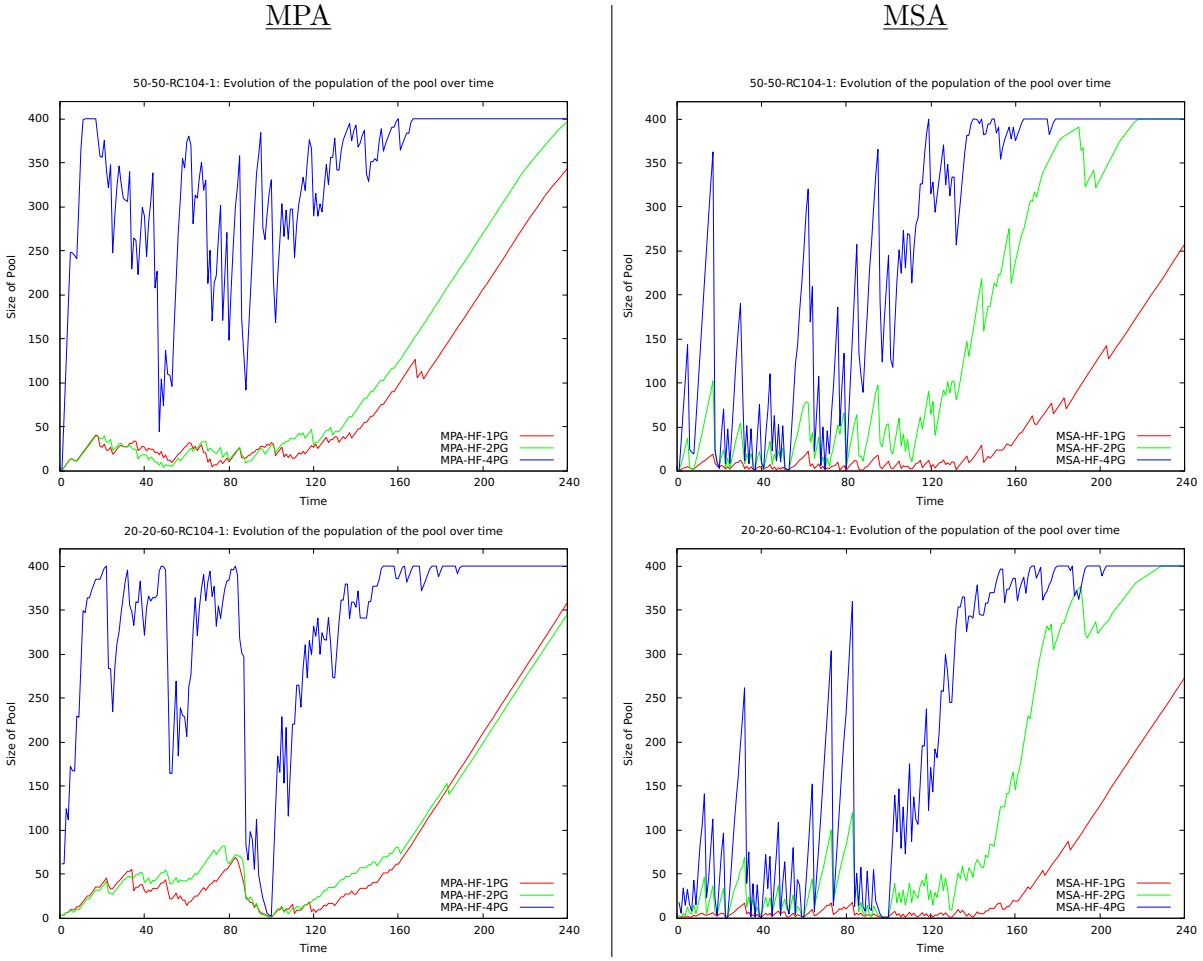

Figure 5.7: Evolution of the population of the pool of plans Γ over the horizon for instances `50-50-RC104-1` of class 3 (top frames) and `20-20-60-RC104-1` of class 4 (bottom frames), with a maximum of 400 solutions. The frames on the left use **MPA** with *consensus* approach and the right frames use **MSA** with *consensus*. On each figure, the configurations using one *PlanGenerator* thread are represented in red, those using two of them are represented in green, and those using four generators are denoted in blue.

requests may also increase or decrease when increasing the number of new plans per time step. Despite this, in the MSA configurations, a bigger number of instances have a decreasing number of refused requests when increasing the number of plan generators. In order to identify the reason for this, we decided to perform an experimentation that would measure the size of the population of the pool of plans Γ along the horizon, for one instance per class. The results of this experiment are represented on figure 5.6 for instances `0-100-RC104-1` from class 1 and `100-0-RC104-1` from class 2, and on figure 5.7 for instances `50-50-RC104-1` from class 3 and `20-20-60-RC104-1` from class 4.

On these figures, a few interesting things have to be observed. First, the population of the pool increases a lot in the last part of the horizon. It is explained by the fact that no requests arrive during the *Late Afternoon* timeslot of the horizon. Also, remember the fact that going from class 1 through 4 increases the degree of dynamism (DOD) and the amount of request that arrive in the late time bins. This implies that the moment when the pool grows quickly should move towards the end of the operations. Indeed, it is clearly visible that the final growing of the pool coincides with the arrival of the last *dynamic* requests. Such an observation highlights the fact that the *new request* events seem to be the main source of solution filtering.
The second thing to notice is the fact that the MSA configurations maintain a much lower number of solutions in Γ, which may explain the overall higher results of MSA compared to

MPA. Indeed, the populations of MSA falls more regularly to small values than those of MPA. These drops may be explained by the second test we perform when using MSA and preferring actual requests ('-arp' parameter set to 1). Remind that when we use this test, a solution where sampled request were removed is kept in Γ only if it is the first in this case. So, if a request could not be integrated to any solution in the pool without removing the sampled requests, only one plan (the first tested) will remain in Γ. Because of this, a large drop may occur and if it happens regularly, population may be maintained small. This situation can be seen for instance `100-0-RC104-1` from class 2 on figure 5.6, where the difference in population between MSA and MPA is clearly visible. This difference implied a performance drop as visible in Table 5.4, where MPA scored 21.1 average of refused requests whereas MSA scored 24.2. This emphasizes the need for methods such as MPA and MSA to have a pool of solutions with a minimal population to perform well.
The third interesting thing is complementary to the second one. We can see on figures 5.6 and 5.7 that having a huge number of solutions in the pool do not bring a significant gain in the objective function. This can be observed by comparing the corresponding scores of configurations using 2 and 4 plans generators in Table 5.4. This implies that a tradeoff must be found between the results targeted, the computational power required to generate enough plans to maintain a certain population, and finally the quality of each of the plans generated.

5.3 Comparing the Efficacy of the Possible Configurations

Description The last experiment we performed was simply the comparison of the respective efficacy of several configurations concerning the objective function. To do this, we measured the average number of **refused requests** on our test instances with various configurations. We first planned to compare all the possible combination of approaches and improving strategies with averages on 10 runs per instance, but due to the same time constraints as for the previous experiment, the configurations using the *consensus* were run 10 times per instance whereas the configurations using the *regrets* approach were run only 6 times per instance.

Parameter values The parameters passed to the application are the same as for the previous experiment:

- The approach (parameter '-a') is set to *consensus* for the first configurations and then to *regrets* for the last ones.
- A time step (parameter '-ts') is set to 3.5 seconds.
- The initial size of the pool (parameter '-ip') is set to 50.
- The maximal size of the pool (parameter '-mp')is set to 400 solutions to avoid too high memory consumption.
- The fleet is reduced (parameter '-hf' set to 1).
- The optimization time for a single plan (parameter '-tp') is set to 1 second.

The other parameters, i.e. the number of plan generators (parameter '-g'), the choice between MPA/MSA (parameter '-m') and the preference about request type (parameter '-arp'), vary from a configuration to another.

The results of this experiment are shown on Table 5.5 for the configurations that used the *consensus* approach and on Table 5.6 for the configurations that used the *regrets* approach.

Discussion On figures 5.5 and 5.6, we can see that we compare MPA (column 1 of each table) using a single plan generator with other configurations that use 2 generators. This was done to measure results of configurations that maintain a similar number of solutions in their pool. By

doing so, we expected averages that would be more comparable, regarding the scores of those configuration in Table 5.4.
The point of this experiment was trying to show the gain brought by the so-called *improving strategies* on our test instances and the impact of the chosen *plan selection* approach. To do so, we computed the average number of *dynamic* requests refused by MSA_c and MSA_r using *waiting* (W-labeled columns) and *relocation* (R-labeled columns) strategies, both with and without using the *actual request* preference (ARP-labeled columns of Tables 5.5 and 5.6 have the second ARP test enabled, whereas it is disabled for the others).

The first global observation we can do is that the configurations using the *regrets* approach of plan selection refuse usually less *dynamic* requests than those using *consensus*, as about two thirds of the values in Table 5.6 are lower than the corresponding numbers in Table 5.5. This is no surprise as *regrets* was expected to perform better than consensus by removing its disadvantages, at the cost of more complex computations. It should also be emphasized that the gap between the *consensus* and *regrets* approaches tends to grow with the DOD at the advantage of the *regrets* approach.

If we compare what is comparable, i.e. the results of MSA configurations preferring actual requests (ARP-labeled columns) on one hand, and the results of the other MSA configurations on the other hand, the expected improvement is quite limited as enhancements are visible on some instances whereas others perform worse when applying improving strategies. This observation is not surprising as it could also be made on the results presented in the work of *Bent et al.*[BVH07].

The most noticeable gain of using improving strategies is visible on the third instance of class 5. For this test instance, both *consensus* and *regrets* tables give very noticeable results. On Table 5.5 (consensus), MPA_c scores 31 refused requests, the MSA_c configurations without ARP score around 29, those using the second test drop this number to around 26 and even to 11 for the one using the relocation strategy. Beside this, on Table 5.6 (regret), similar behavior is shown, as MPA_r scores 29.3 refused requests, and all the MSA_r configurations (with and without ARP) score around mid 20, except the one using ARP and *relocation*, which scores 9.3 refused requests.

On the other hand, the most noticeable loss in performance is visible on the first test instance of class 2. Indeed, MPA_c and MPA_r both scored 8.3 whereas all the MSA configurations (using *consensus* or *regret*, with or without ARP) score around 14 refused requests.

	MPA$_c$-1PG-HF	**MPA$_c$-2PG-HF**	**MPA$_c$-4PG-HF**	**MSA$_c$-1PG-ARP-HF**	**MSA$_c$-2PG-ARP-HF**	**MSA$_c$-4PG-ARP-HF**
class 1						
0-100-rc101-1	13.7	12.2	13.2	16	14	13.9
0-100-rc102-1	21.4	23.2	23.8	25	24.5	26.2
0-100-rc104-1	18.8	21.4	19.2	23.3	22.3	22
class 2						
100-0-rc101-1	14	13.8	14.4	14.1	14.4	13.3
100-0-rc102-1	13.4	14.8	13.9	16	15.7	15
100-0-rc104-1	21.1	20.2	22.9	24.2	23.8	23.6
class 3						
50-50-rc101-1	19	16.4	17	19	18.7	18.4
50-50-rc102-1	16.2	18	16.9	18.3	17.7	18.3
50-50-rc104-1	25.7	24.8	25.1	24.9	25.2	24.9
class 4						
20-20-60-rc101-1	13.3	16.4	13.5	15.7	14.4	13.5
20-20-60-rc102-1	11.6	11	11.5	13.1	13.1	14.1
20-20-60-rc104-1	39.5	40.8	41.2	42.2	39.4	41

Table 5.4: Results obtained on *Bent et al.*'s instances of the four first classes. We exhibit the average number of *dynamic* requests refused by different configurations on 10 run for each *configuration/instance* pair. The compared configurations are from left to right: **MPA$_c$** with a single plan generator and the fleet reduced, **MPA$_c$** with two plan generators and the fleet reduced, **MPA$_c$** with four plan generators and the fleet reduced, **MSA$_c$** preferring actual requests with a single plan generator and the fleet reduced, **MSA$_c$** preferring actual requests with two plan generators and the fleet reduced and finally **MSA$_c$** preferring actual requests with four plan generators and the fleet reduced.

	MPA_c-1PG-HF	MSA_c-2PG-HF	MSA_c-R-2PG-HF	MSA_c-W-2PG-HF	MSA_c-2PG-ARP-HF	MSA_c-R-2PG-ARP-HF	MSA_c-W-2PG-ARP-HF
class 1							
0-100-rc101-1	13.7	15.4	14.2	14.9	13.9	13.3	14.2
0-100-rc102-1	21.4	26	24.9	25.6	21.5	24.1	24.8
0-100-rc104-1	18.8	22.2	21.5	23.3	20.5	21.6	21.6
class 2							
100-0-rc101-1	8.3	9.2	15	14.4	13.8	14.1	14.2
100-0-rc102-1	13.4	10.4	16.3	15.5	15.4	14.7	12.9
100-0-rc104-1	21.1	15.5	22.8	23.5	20.8	20.9	22.1
class 3							
50-50-rc101-1	19	20.2	19.6	19.5	18.3	18.5	18
50-50-rc102-1	16.2	18.6	18.4	17.7	16.9	17	17.5
50-50-rc104-1	25.7	24.1	26.5	25.9	24.5	25.4	26
class 4							
20-20-60-rc101-1	13.3	14.2	13.1	13.5	13.9	13.4	14.1
20-20-60-rc102-1	11.6	13.5	13.4	14.8	12.4	11.1	11.7
20-20-60-rc104-1	39.5	41.4	41	41	34.1	36.3	41
class 5							
10-10-80-rc101-1	13.3	18.7	17.5	19.7	19.5	16.4	14.6
10-10-80-rc102-1	16.8	17.5	16.6	15.7	16.2	17.2	18
10-10-80-rc104-1	31	29.8	29.4	28.5	27.8	11	25.3

Table 5.5: Results obtained on *Bent et al.*'s instances of the five classes. We exhibit the average number of *dynamic* requests refused by different configurations using the *consensus* approach on 10 runs for each *configuration/instance* pair. The compared configurations are from left to right: MPA_c with one plan generator an the fleet reduced (column 1); MSA_c with two plan generators and the fleet reduced (column 2); MSA_c with *relocation strategy*, two plan generators and the fleet reduced(column 3); MSA_c with *waiting strategy*, two plan generators and the fleet reduced (column 4), then the same MSA_c configurations, preferring actual requests (ARP) and performing the *second test* in the last columns (columns 5,6,7).

	MPA_r-1PG-HF	MSA_r-2PG-HF	MSA_r-R-2PG-HF	MSA_r-W-2PG-HF	MSA_r-2PG-ARP-HF	MSA_r-R-2PG-ARP-HF	MSA_r-W-2PG-ARP-HF
class 1							
0-100-rc101-1	11.3	14.3	14	13.6	13	12.3	13.3
0-100-rc102-1	23.3	23.3	25	23.3	21.3	23.3	22.3
0-100-rc104-1	20.3	21.6	21.6	22.6	21.6	20	19.3
class 2							
100-0-rc101-1	8.3	15.3	14	14.6	14	14.3	13.3
100-0-rc102-1	12.6	14.6	14.6	16.3	13	15	15
100-0-rc104-1	22.6	21.3	23.3	23.3	23	17	19.3
class 3							
50-50-rc101-1	18.6	16.6	18.6	18.3	17.3	20.3	18.3
50-50-rc102-1	17	18.3	18.3	16	17	15.3	16
50-50-rc104-1	24.6	23.3	24.6	24.3	24	23.6	24.3
class 4							
20-20-60-rc101-1	13.6	13	13.3	11.3	15.6	13.3	12.3
20-20-60-rc102-1	11	10.6	12.3	14.3	12.3	6.6	10.3
20-20-60-rc104-1	38.6	37.6	38.6	38.3	41	36	37.3
class 5							
10-10-80-rc101-1	15	19	15	18.3	18	13.6	14.3
10-10-80-rc102-1	20	19.3	14.3	17	17	15.6	17
10-10-80-rc104-1	29.3	21	26	26	26.6	9.3	23.6

Table 5.6: Results obtained on *Bent et al.*'s instances of the five classes. We exhibit the average number of *dynamic* requests refused by different configurations using the *regrets* approach on 6 runs for each *configuration/instance* pair. The compared configurations are from left to right: MPA_r with one plan generator an the fleet reduced (column 1); MSA_r with two plan generators and the fleet reduced (column 2); MSA_r with *relocation strategy*, two plan generators and the fleet reduced(column 3); MSA_r with *waiting strategy*, two plan generators and the fleet reduced (column 4), then the same MSA_r configurations preferring actual requests (ARP) and performing the *second test* in the last columns (columns 5,6,7).

Chapter 6

Conclusion

Achievements

During this master thesis, we designed a solution framework for the Dynamic and Stochastic Vehicle Routing Problem with Time Windows following the work of *Bent et al.* [BVH04d, BVH04a, BVH07]. Our design was able to allow all combinations of methods, plan selection approaches and improving strategies proposed by *Bent et al.* to be executed and compared easily thanks to a unique representation in the implementation using a `C++` library currently in development at Université Catholique de Louvain.

First, we exposed the characteristics of the class of problem we tackled in our work, starting with the expression of the most basic VRP. We then expanded it with complicating constraints until the complete DS-VRPTW was reached. We also introduced the techniques developed in the literature that we wanted to implement. Second, we revealed the standardization process we performed on these techniques to allow compatibility between them. It was followed by the presentation of the solution framework we designed, along with the offline solver contained in the library we used as a black box. To do so, we introduced the architecture of our framework and situated it with respect to the generic structure of a VRP instance we presented when we described this combinatorial problem. We then focused on the main components of this architecture by describing the agents we designed and that interact during the *dynamic* phase of the problem. We also highlighted the advantages of our framework, such as the fact that it permits an easy parallelization of the generation of solutions, limiting the problem of making a tradeoff between the time allowed to produce a solution and the length of a time step of the *dynamic* phase of the problem. This was followed by detailed explanations and motivations of some part of our implementation using this solver.

We finally ran some experiments to assess the validity of our design choices and verified if the results we measured were consistent with the expectations of the work of *Bent et al.*. We showed that despite the standardization process we performed on the original algorithms to transit from *request selection*-based approaches to *plan selection*-based approaches, our framework was able to provide results with quality comparable with the expectations from the literature, as long as a sufficient number of solutions are kept in the pools of the *Multiple Plan Approach* and *Multiple Scenario Approach*.

Further Work

Plenty of further work could be done, leading to possible enhancement of our solution framework. One should first try to understand better why the MSA algorithms (using improving strategies or not) does not produce 'dramatically' improved results expected in the literature

[BVH04d][BVH07], when using the same production rate of new solutions. Also, the algorithm is worth to be re-implemented in a more distant version of the expressions given by *Bent et al.* but more effective when using the tools provided by the library we use as an Offline Solver. The algorithm we designed used the *waiting* vertices provided by the library in such a way that we could use a single representation compatible with all the configurations we wanted to test, but it would be worth trying to use distinct representations/implementation to benefit from all the potential of each approaches and improving strategies. For example, we could drop the waiting vertices when using the *waiting* **improving strategy**, and replace the **DRIVE-FIRST-DELAY-LAST** optimization strategy of the solutions by **CUSTOM-WAIT** that allows adding/modifying some waiting times at the customer location of each request in the solutions. Remember that we did not choose this because using it with the *relocation* strategy would have violated the hypothesis we had that prevents *sampled* requests to be in the fixed part of the solutions, and because we wanted a single algorithm that would adapt easily to all the configurations.

In addition, during the processing of the results we presented, some experiments appeared to be worth trying in further work:

- The evaluation of the critical amount of plans needed in the pool Γ, i.e. the minimal number of solutions so that the decisions are sufficiently informed to perform well, would be interesting to be done.
- Evaluate the minimal production rate of plans during a single time step and received as *new plan* events, so that the critical amount of plans is maintained in the pool.
- From the two preceding evaluations, determine the best tradeoff between the number of solutions to produce in a time step, and the quality of these solutions. For a given length of time step, if we reduce the number of plans to produce, we can increase/adjust the duration of optimization for each.
- Another tradeoff worth optimizing is the one between the computational power to invest versus the gain in objective function, i.e. maximizing the number of served requests/minimizing the number of refused requests.

Finally, a compatibility analysis of our framework with different models of VRP (or rather the quantity of adaptations needed, as modifications probably must be done) is also a good orientation for further work. It may be interesting to determine if is easily transposable to new or different common objective functions, such as adding the minimization of needed vehicles to the maximization of served requests, etc...

Bibliography

[BKVH05] Russell Bent, Irit Katriel, and Pascal Van Hentenryck. Sub-optimality approximations. In *International Conference on Principles and Practice of Constraint Programming*, pages 122–136. Springer, 2005.

[BVH03] Russell Bent and Pascal Van Hentenryck. Dynamic vehicle routing with stochastic requests. In *IJCAI*, pages 1362–1363, 2003.

[BVH04a] Russell Bent and Pascal Van Hentenryck. Regrets only! online stochastic optimization under time constraints. In *AAAI*, volume 4, pages 501–506, 2004.

[BVH04b] Russell Bent and Pascal Van Hentenryck. A two-stage hybrid local search for the vehicle routing problem with time windows. *Transportation Science*, 38(4):515–530, 2004.

[BVH04c] Russell Bent and Pascal Van Hentenryck. The value of consensus in online stochastic scheduling. In *ICAPS*, volume 4, pages 219–226, 2004.

[BVH04d] 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.

[BVH07] Russell Bent and Pascal Van Hentenryck. Waiting and relocation strategies in online stochastic vehicle routing. In *IJCAI*, pages 1816–1821, 2007.

[CAF16] Diego Cattaruzza Cattaruzza, Nabil Absi, and Dominique Feillet. Vehicle routing problems with multiple trips. *4OR: A Quarterly Journal of Operations Research*, to appear, 2016.

[CGC00] Hyeong Soo Chang, Robert Givan, and Edwin KP Chong. On-line scheduling via sampling. In *AIPS*, pages 62–71, 2000.

[CLM01] Jean-François Cordeau, Gilbert Laporte, and Anne Mercier. A unified tabu search heuristic for vehicle routing problems with time windows. *Journal of the Operational research society*, 52(8):928–936, 2001.

[DR59] George B Dantzig and John H Ramser. The truck dispatching problem. *Management science*, 6(1):80–91, 1959.

[Fer13] Francesco Ferrucci. *Pro-active Dynamic Vehicle Routing: Real-time Control and Request-forecasting Approaches to Improve Customer Service*. Springer Science & Business Media, 2013.

[HB09] Pascal Van Hentenryck and Russell Bent. *Online stochastic combinatorial optimization*. The MIT Press, 2009.

[HLL06] Lars M Hvattum, Arne Løkketangen, and Gilbert Laporte. Solving a dynamic and stochastic vehicle routing problem with a sample scenario hedging heuristic. *Transportation Science*, 40(4):421–438, 2006.

[ILC11] Sahbi Ben Ismail, François Legras, and Gilles Coppin. *Synthèse du problème de routage de véhicules*. PhD thesis, Dépt. Logique des Usages, Sciences Sociales et de l'Information (Institut Mines-Télécom-Télécom Bretagne-UEB); Laboratoire en sciences et technologies de l'information, de la communication et de la connaissance (UMR CNRS 6285-Télécom Bretagne-Université de Bretagne Occidentale-Université de Bretagne Sud), 2011.

[KP12] Suresh Nanda Kumar and Ramasamy Panneerselvam. A survey on the vehicle routing problem and its variants. *Intelligent Information Management*, 4(3):66, 2012.

[KS97] Gerard AP Kindervater and Martin WP Savelsbergh. Vehicle routing: handling edge exchanges. *Local search in combinatorial optimization*, pages 337–360, 1997.

[LMS02] Allan Larsen, OBGD Madsen, and Marius Solomon. Partially dynamic vehicle routing—models and algorithms. *Journal of the operational research society*, 53(6):637–646, 2002.

[Mas13] Florence Massen. Optimization approachs for the vehicle routing problem with black box feasibility. 2013.

[PDH08] Sophie N Parragh, Karl F Doerner, and Richard F Hartl. A survey on pickup and delivery models part ii: Transportation between pickup and delivery locations. *Journal für Betriebswirtschaft*, 58(2):81–117, 2008.

[RP13] Ulrike Ritzinger and Jakob Puchinger. Hybrid metaheuristics for dynamic and stochastic vehicle routing. In *Hybrid metaheuristics*, pages 77–95. Springer, 2013.

[RPH16] Ulrike Ritzinger, Jakob Puchinger, and Richard F Hartl. A survey on dynamic and stochastic vehicle routing problems. *International Journal of Production Research*, 54(1):215–231, 2016.

[SGDS15] Michael Saint-Guillain, Yves Deville, and Christine Solnon. A multistage stochastic programming approach to the dynamic and stochastic vrptw. In *International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems*, pages 357–374. Springer, 2015.

[SS11] P Surekha and S Sumathi. Solution to multi-depot vehicle routing problem using genetic algorithms. *World Applied Programming*, 1(3):118–131, 2011.

[TBG+97] Éric Taillard, Philippe Badeau, Michel Gendreau, François Guertin, and Jean-Yves Potvin. A tabu search heuristic for the vehicle routing problem with soft time windows. *Transportation science*, 31(2):170–186, 1997.

Appendix A

Class Diagram of the Solver used

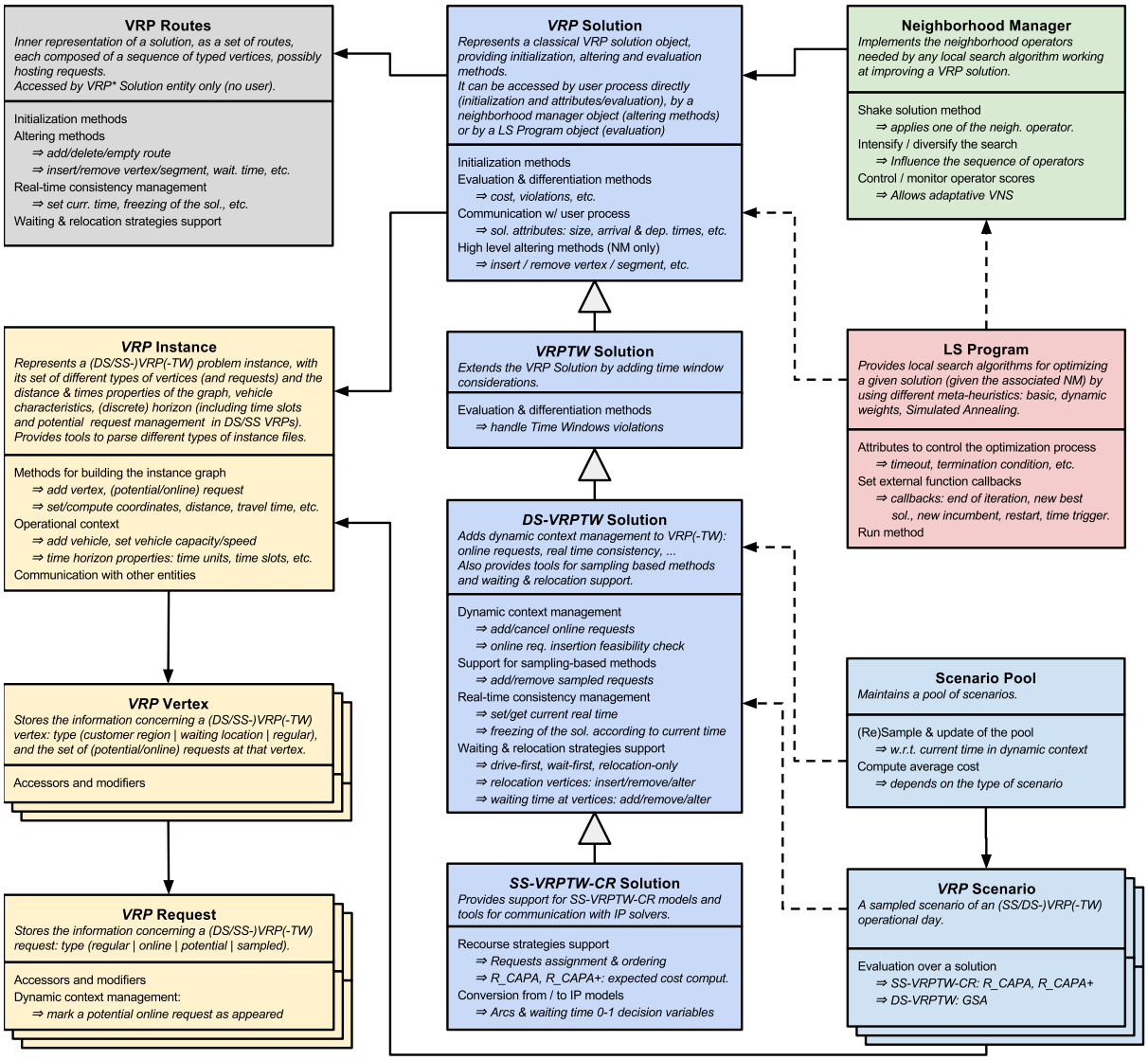

Figure A.1: Class diagram of the C++ library used as solver.

Appendix B

Source Code Parts

B.1 Plan Generation

```
while (!stop) {
    if (ok2gen && next_time < I.getHorizon()) {
        Solution_DS_VRPTW* s = mpa.getBest(false);

        NeighborhoodManager<Solution_DS_VRPTW> nm(*s, DS_VRPTW);
        LS_Program_SimulatedAnnealing<Solution_DS_VRPTW,
                NeighborhoodManager < Solution_DS_VRPTW>> lsprog;

        double duration = dist(mt);

        if (stochastic) {
            s->setCurrentTime(time);
            s->removeAllSampledOnlineRequests();

            s->setCurrentTime(next_time);
            sampleRequests(I, eh, s);
        }
        s->setCurrentTime(next_time);

        lsprog.setTimeOut(duration);
        lsprog.run(*s, nm, numeric_limits<int>::max(), 10.0, 0.95, false);

        if (stochastic) {
            s->setCurrentTime(next_time);
            sampleRequests(I, eh, s);
        }
        if (ok2gen && time < next_time){//still ok to add the new solution
            if (!(s->getNumberViolations() > 0)) {
                mtx.lock();
                eh.addPlanAtTime(s, next_time);
                PG_log << outputMisc::blueExpr(true) <<
                        "Generator: solution added to EventHandler!" <<
                        outputMisc::resetColor() << endl;
                mtx.unlock();
            } else {
                delete s;
            }
        } else {
            PG_log << "too late" << endl;
            delete s;
            next_time = time + 1;
        }
    }
}
```

Figure B.1: Source code generating new solutions and sending them to the *EventHandler*.

B.2 Applying Waiting Strategy

```
double travelTime = I.travelTime(cur_actions[route].pvertex->getRegion(),
        next_actions[route].pvertex->getRegion());

if (cur_actions[route].pvertex->getType() != VRP_vertex::WAITING &&
        (cur_actions[route].action == ActionType::SERVE || cur_actions[route].action ==
            ActionType::WAIT_AT)) {

    route_pos site;
    site.region = cur_actions[route].pvertex->getRegion();
    site.route = route + 1;
    site.pos = next_actions[route].pos;
    sites->push_back(site);

    cout << "a waitingVertex should appear before region "
         << next_actions[route].pvertex->getRegion()
         << " of type " <<next_actions[route].pvertex->getType()
         <<"and revealed is "
         <<next_actions[route].pvertex->getRequest(timeslot).isRevealed()
         <<endl;
} else if (cur_actions[route].pvertex->getType() == VRP_vertex::WAITING) {
    cout << "There is a waiting vertex already" << endl;
    if (cur_actions[route].end_time + travelTime + 1.01 <=
            next_actions[route].pvertex->getRequest(timeslot).e) {
        cout << "we wait some more" << endl;
        plan->increaseWaitingTime(route + 1, cur_actions[route].pos, 1.01);
        cout << "new next: " << next_actions[route].toString() << endl;
    }
    else {
        cout << "Sampled request to remove from route: " << route + 1
             << " at pos: " << cur_actions[route].pos + 1 << endl;
        plan->removeSampledOnlineRequest(route + 1, cur_actions[route].pos + 1);
        next_actions = plan->getNextPlannedActions();
        timeslot = plan->getRequestAtPosition(route + 1, next_actions[route].pos).
            revealTS;
        cout << "new next: " << next_actions[route].toString() << endl;

        if (next_actions[route].action == ActionType::MOVE_TO &&
                next_actions[route].pvertex->getType() == VRP_vertex::REGULAR_ONLINE &&
                !(next_actions[route].pvertex->getRequest(timeslot).isRevealed())) {
            plan->increaseWaitingTime(route + 1, cur_actions[route].pos, 1.01);
            cout << "new next: " << next_actions[route].toString() << endl;
        }
    }
```

Figure B.2: C++ code applying the waiting strategy on a solution.

B.3 Applying Relocation Strategy

```
if(cur_actions[route].end_time - time <= 1 &&
   cur_actions[route].pvertex->getRegion() != next_actions[route].pvertex->getRegion() &&
   next_actions[route].action == ActionType::MOVE_TO &&
   next_actions[route].pvertex->getType() == VRP_vertex::REGULAR_ONLINE &&
   !(next_actions[route].pvertex->getRequest(timeslot).isRevealed())) {

    if ((cur_actions[route].action == ActionType::SERVE ||
        cur_actions[route].action == ActionType::WAIT_AT)) {
        route_pos site;
        site.region = next_actions[route].pvertex->getRegion();
        site.route = route + 1;
        site.pos = next_actions[route].pos;

        sites->push_back(site);
    }
}
else if (cur_actions[route].end_time - time <= 1 &&
        cur_actions[route].pvertex->getType() == VRP_vertex::WAITING &&
        cur_actions[route].action == ActionType::MOVE_TO) {
    //Arriving to a waitingVertex And the  next destination on the route is a smpled
        request at another location
    if (cur_actions[route].end_time - time <= 1 &&
        cur_actions[route].pvertex->getRegion() != plan->getVertexAtPosition(route + 1,
            cur_actions[route].pos + 1).getRegion() &&
        plan->getVertexAtPosition(route + 1, cur_actions[route].pos + 1).getType() ==
            VRP_vertex::REGULAR_ONLINE &&
        !(plan->getRequestAtPosition(route + 1, cur_actions[route].pos + 1).isRevealed())
            ) {
        route_pos site;

        site.region = plan->getVertexAtPosition(route + 1, cur_actions[route].pos + 1).
            getRegion();
        site.route = route + 1;
        site.pos = cur_actions[route].pos + 1;
        sites->push_back(site);
    }

    while (plan->getVertexAtPosition(route + 1, cur_actions[route].pos + 1).getType() ==
        VRP_vertex::REGULAR_ONLINE &&
            cur_actions[route].pvertex->getRegion() == plan->getVertexAtPosition(route +
                1, cur_actions[route].pos + 1).getRegion() &&
            !(plan->getRequestAtPosition(route + 1, cur_actions[route].pos + 1).
                isRevealed())) {
        plan->removeSampledOnlineRequest(route + 1, cur_actions[route].pos + 1);
        next_actions = plan->getNextPlannedActions();
        timeslot = plan->getRequestAtPosition(route + 1, cur_actions[route].pos + 1).
            revealTS;

        if (cur_actions[route].end_time - time <= 1 &&
            cur_actions[route].pvertex->getRegion() != plan->getVertexAtPosition(route +
                1, cur_actions[route].pos + 1).getRegion() &&
            plan->getVertexAtPosition(route + 1, cur_actions[route].pos + 1).getType() ==
                 VRP_vertex::REGULAR_ONLINE &&
            !(plan->getRequestAtPosition(route + 1, cur_actions[route].pos + 1).
                isRevealed())) {
            //cout << "there was 2 sampled requests in a row." << endl;
            route_pos site;
            site.region = plan->getVertexAtPosition(route + 1, cur_actions[route].pos +
                1).getRegion();
            site.route = route + 1;
            site.pos = cur_actions[route].pos + 1;

            sites->push_back(site);
        }
    }
}
```

Figure B.3: C++ code applying the relocation strategy on a solution.

B.4 Request Insertion

```
// We pass through the plans of the pool and try to insert the request
// 'req' inside each solution; depending on the value of 'test2' a single
// try is performed, or a second is executed on a copy of the current plan
// whose sampled requests were removed.
for (auto p : plans) {
    if (!test2) {// if the second test is disabled
        bool inserted = p->tryInsertOnlineRequest(*(req.vertex), req.request->revealTS);

        if (inserted) {
            new_plans->push_back(p);
            accepted_plans[i] = 1; //The request was inserted successfully, the plan is
                accepted as is.
        } else {
            accepted_plans[i] = 0; //The request wasn't inserted, the plan is discarded.
        }
    } else {
        bool inserted = p->tryInsertOnlineRequest(*(req.vertex), req.request->revealTS);

        if (inserted) {
            new_plans->push_back(p);
            accepted_plans[i] = 1;//The request was inserted successfully, the plan is
                accepted as is.
        }
        else {
            Solution_DS_VRPTW* q = new Solution_DS_VRPTW(*p);
            q->removeAllSampledOnlineRequests();

            bool inserted2 = q->tryInsertOnlineRequest(*(req.vertex), req.request->
                revealTS);

            if (inserted2 && new_plans->empty()) {
                new_plans->push_back(q);
                accepted_plans[i] = 2;//The request was inserted successfully in the copy
                    , the copy joins the pool.
            } else {
                delete q;
                accepted_plans[i] = 0;//The request wasn't inserted, the plan is
                    discarded.
            }
        }
    }
    i++;
}
```

Figure B.4: `C++` code applying the expected behavior when trying to insert a new arriving request to the solutions in the pool, depending on the value of variable `test2`.

Appendix C

Benchmark Instance

Customers:	100	Capacity:	200	Time Bins	4	Vehicles	16		
Cust	X	Y	Demand	Start	End	Service	PreBin	0−160	160−320	320−480
D	40	50	0	0	240	0	0	0	0	0
0	25	85	20	145	175	10	50	40	10	0
1	22	75	30	50	80	10	100	0	0	0
2	22	85	10	109	139	10	50	50	0	0
3	20	80	40	141	171	10	50	40	10	0
4	20	85	20	41	71	10	100	0	0	0
5	18	75	20	95	125	10	50	50	0	0
6	15	75	20	79	109	10	50	50	0	0
7	15	80	10	91	121	10	50	50	0	0
8	10	35	20	91	121	10	50	50	0	0
9	10	40	30	119	149	10	50	50	0	0
10	8	40	40	59	89	10	50	50	0	0
11	8	45	20	64	94	10	50	50	0	0
12	5	35	10	142	172	10	50	40	10	0
13	5	45	10	35	65	10	100	0	0	0
14	2	40	20	58	88	10	50	50	0	0
15	0	40	20	72	102	10	50	50	0	0
16	0	45	20	149	179	10	50	40	10	0
17	44	5	20	87	117	10	50	50	0	0
18	42	10	40	72	102	10	50	50	0	0
19	42	15	10	122	152	10	50	50	0	0
20	40	5	10	67	97	10	50	50	0	0
21	40	15	40	92	122	10	50	50	0	0
22	38	5	30	65	95	10	50	50	0	0
23	38	15	10	148	178	10	50	40	10	0
24	35	5	20	154	184	10	50	40	10	0
25	95	30	30	115	145	10	50	50	0	0
26	95	35	20	62	92	10	50	50	0	0
27	92	30	10	62	92	10	50	50	0	0
28	90	35	10	67	97	10	50	50	0	0
29	88	30	10	74	104	10	50	50	0	0
30	88	35	20	61	91	10	50	50	0	0
31	87	30	10	131	161	10	50	40	10	0
32	85	25	10	51	81	10	50	50	0	0
33	85	35	30	111	141	10	50	50	0	0
34	67	85	20	139	169	10	50	40	10	0
35	65	85	40	43	73	10	100	0	0	0
36	65	82	10	124	154	10	50	50	0	0
37	62	80	30	75	105	10	50	50	0	0
38	60	80	10	37	67	10	100	0	0	0
39	60	85	30	85	115	10	50	50	0	0
40	58	75	20	92	122	10	50	50	0	0
41	55	80	10	33	63	10	100	0	0	0
42	55	85	20	128	158	10	50	50	0	0
43	55	82	10	64	94	10	50	50	0	0
44	20	82	10	37	67	10	100	0	0	0
45	18	80	10	113	143	10	50	50	0	0
46	2	45	10	45	75	10	100	0	0	0
47	42	5	10	151	181	10	50	40	10	0
48	42	12	10	104	134	10	50	50	0	0
49	72	35	30	116	146	10	50	50	0	0
50	55	20	19	83	113	10	50	50	0	0
51	25	30	3	52	82	10	50	50	0	0
52	20	50	5	91	121	10	50	50	0	0
53	55	60	16	139	169	10	50	40	10	0
54	30	60	16	140	170	10	50	40	10	0
55	50	35	19	130	160	10	50	50	0	0
56	30	25	23	96	126	10	50	50	0	0
57	15	10	20	152	182	10	50	40	10	0
58	10	20	19	42	72	10	100	0	0	0
59	15	60	17	155	185	10	50	40	10	0
60	45	65	9	66	96	10	50	50	0	0
61	65	35	3	52	82	10	50	50	0	0
62	65	20	6	39	69	10	100	0	0	0
63	45	30	17	53	83	10	50	50	0	0
64	35	40	16	11	41	10	100	0	0	0
65	41	37	16	133	163	10	50	40	10	0
66	64	42	9	70	100	10	50	50	0	0
67	40	60	21	144	174	10	50	40	10	0
68	31	52	27	41	71	10	100	0	0	0
69	35	69	23	180	210	10	50	40	10	0

70	65	55	14	65	95	10	50	50	0	0
71	63	65	8	30	60	10	100	0	0	0
72	2	60	5	77	107	10	50	50	0	0
73	20	20	8	141	171	10	50	40	10	0
74	5	5	16	74	104	10	50	50	0	0
75	60	12	31	75	105	10	50	50	0	0
76	23	3	7	150	180	10	50	40	10	0
77	8	56	27	90	120	10	50	50	0	0
78	6	68	30	89	119	10	50	50	0	0
79	47	47	13	192	222	10	50	40	10	0
80	49	58	10	86	116	10	50	50	0	0
81	27	43	9	42	72	10	100	0	0	0
82	37	31	14	35	65	10	100	0	0	0
83	57	29	18	96	126	10	50	50	0	0
84	63	23	2	87	117	10	50	50	0	0
85	21	24	28	87	117	10	50	50	0	0
86	12	24	13	90	120	10	50	50	0	0
87	24	58	19	67	97	10	50	50	0	0
88	67	5	25	144	174	10	50	40	10	0
89	37	47	6	86	116	10	50	50	0	0
90	49	42	13	167	197	10	50	40	10	0
91	53	43	14	14	44	10	100	0	0	0
92	61	52	3	178	208	10	50	40	10	0
93	57	48	23	95	125	10	50	50	0	0
94	56	37	6	34	64	10	100	0	0	0
95	55	54	26	132	162	10	50	40	10	0
96	4	18	35	120	150	10	50	50	0	0
97	26	52	9	46	76	10	100	0	0	0
98	26	35	15	77	107	10	50	50	0	0
99	31	67	3	180	210	10	50	40	10	0

Known Requests 57

Cust	Arrival	Start	Deadline	Demand	Service
1	−1	50	80	30	10
4	−1	41	71	20	10
10	−1	59	89	40	10
12	−1	142	172	10	10
13	−1	35	65	10	10
14	−1	58	88	20	10
15	−1	72	102	20	10
16	−1	149	179	20	10
17	−1	87	117	20	10
20	−1	67	97	10	10
22	−1	65	95	30	10
23	−1	148	178	10	10
24	−1	154	184	20	10
25	−1	115	145	30	10
26	−1	62	92	20	10
28	−1	67	97	10	10
29	−1	74	104	10	10
30	−1	61	91	20	10
32	−1	51	81	10	10
33	−1	111	141	30	10
34	−1	139	169	20	10
35	−1	43	73	40	10
38	−1	37	67	10	10
41	−1	33	63	10	10
42	−1	128	158	20	10
43	−1	64	94	10	10
44	−1	37	67	10	10
46	−1	45	75	10	10
48	−1	104	134	10	10
53	−1	139	169	16	10
54	−1	140	170	16	10
55	−1	130	160	19	10
56	−1	96	126	23	10
57	−1	152	182	20	10
58	−1	42	72	19	10
62	−1	39	69	6	10
64	−1	11	41	16	10
65	−1	133	163	16	10
68	−1	41	71	27	10
69	−1	180	210	23	10
70	−1	65	95	14	10
71	−1	30	60	8	10
72	−1	77	107	5	10
75	−1	75	105	31	10
78	−1	89	119	30	10
80	−1	86	116	10	10
81	−1	42	72	9	10
82	−1	35	65	14	10
83	−1	96	126	18	10
86	−1	90	120	13	10
88	−1	144	174	25	10
90	−1	167	197	13	10
91	−1	14	44	14	10
92	−1	178	208	3	10
93	−1	95	125	23	10
94	−1	34	64	6	10
97	−1	46	76	9	10

Unknown Requests 37

Cust	Arrival	Start	Deadline	Demand	Service
30	0	61	91	20	10
95	2	132	162	26	10
22	3	65	95	30	10
51	5	52	82	3	10
32	8	51	81	10	10
20	9	67	97	10	10

23	10	148	178	10	10
14	14	58	88	20	10
39	16	85	115	30	10
59	20	155	185	17	10
26	21	62	92	20	10
8	22	91	121	20	10
63	22	53	83	17	10
70	26	65	95	14	10
83	26	96	126	18	10
18	29	72	102	40	10
40	29	92	122	20	10
89	30	86	116	6	10
0	31	145	175	20	10
75	32	75	105	31	10
84	32	87	117	2	10
12	33	142	172	10	10
52	34	91	121	5	10
37	50	75	105	30	10
5	52	95	125	20	10
85	56	87	117	28	10
19	58	122	152	10	10
73	58	141	171	8	10
17	60	87	117	20	10
31	62	131	161	10	10
78	70	89	119	30	10
50	71	83	113	19	10
60	73	73	96	9	10
47	78	151	181	10	10
31	80	131	161	10	10
79	109	192	222	13	10
92	138	178	208	3	10

Appendix D

Quantitative data about the code

The table below presents some quantitative data about the code written for this project. We don't count the library used as an Offline Solver nor the minified version of the files. The entire source code is joined to this master as an archive file and constitutes the last appendix.

Language	Number of files	Total size of files	Total number of lines
C++	13	137Ko	3302
Bash (for test scripts)	9	29Ko	897

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