UNIVERSITÀ DEGLI STUDI DI BRESCIA
Dipartimento di Ingegneria dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale

PROBLEMI DI ROUTING A PIÙ VEICOLI IN PRESENZA DI VINCOLI TEMPORALI

Time Constrained Vehicle Routing Problems

Relatore:
Prof.ssa Renata Mansini

Laureanda:
Alice Raffaele
Matricola 81956

Correlatori:
Prof. Jean-François Côté
Ing. Daniele Manerba

Anno Accademico 2014/2015

Ringraziamenti

C'è un tempo che non dovrebbe essere dato per scontato, quello che mi è stato dedicato da molte persone durante questi anni di studi. Può essere stato solo un attimo per scrivere un messaggio, oppure ore al telefono, o lunghe serate a parlare: non importa quanto sia durato, è tempo che merita di essere ammirato.

Vorrei ringraziare innanzitutto la Prof.ssa Renata Mansini, relatrice di questo lavoro di tesi, non solo per essere stata sempre gentile e disponibile, bensì soprattutto per l'appoggio e la comprensione mostrati nei momenti più impegnativi.

Un ringraziamento doveroso al Prof. Jean-François Côté dell'Université Laval, per aver reso possibile svolgere la tesi all'estero e per avere finanziato il progetto.

Sono stata fortunata, durante i mesi in Québec, di aver conosciuto alcune persone che, grazie ai momenti passati insieme, mi hanno aiutato a superare il tempo lontana. In particolare, *merci beaucoup*, Louise e Pierre, per come mi avete accolta fin dal primo giorno, per gli scambi di parole in francese e italiano, per i pezzetti di cioccolato fondente e i bicchierini di sciroppo d'acero.

Ho passato molto tempo con altri ragazzi come me, lontani da casa magari per un periodo temporaneo o definitivamente.

Gracias, Roberto, per le lunghe chiacchierate, prima di persona e poi attraverso due monitor, però piacevoli allo stesso modo.

Per le mille gite ed esperienze organizzate assieme, e per i confronti su libri, film e ricette, *bedankt*, Lieke.

Mamnoon, Maryam e Parnian, per aver fatto subito gruppo, per i consigli su come affrontare tutto e per i *plan b*.

Per aver trascorso assieme le giornate in laboratorio, *obrigado*, Luciana.

Grazie, Fabio, per aver condiviso l'esperienza in Canada.

Per le nostre serate multilingue, per avermi parlato delle abitudini, dei piatti e delle canzoni locali, *merci*, Marianic; mi hai fatto entrare di più in contatto con la meravigliosa cultura del Québec.

Grazie, Riccardo, per l'enorme sostegno morale sia lì sia qui, per l'aiuto che mi hai dato e per avermi ricordato che le impressioni non sempre sono corrette ed è bello cambiare idea.

Nel periodo via da casa il supporto è stato grande anche da chi era in Italia. Grazie infinite, Zia Gabri, Zio Primo e Simona, per avermi scritto sempre frasi di incoraggiamento.

A Natalina e Sergio, per come mi fate sentire di essere parte di una seconda famiglia; a voi, a Giada e Osvi, grazie di cuore per come mi avete incitato e di come mi siete stati vicini.

Grazie alle pendolari del treno, in particolare a Ida e Desirée, con cui ho spesso cominciato o terminato le mie giornate in questi anni.

Grazie al supporto di Sandra, del Compare e della sua famiglia, di Luisa e Antonello, di Antonietta e Antonio, di Lorenzo.

Agli amici di sempre, quelli per cui non conta quanto tempo passa ed è sempre il momento di trovarsi una sera a bere qualcosa e a raccontarsi, oppure, se non si riesce dal vivo, allora ci si mette d'accordo per una videochiamata. Grazie specialmente a Paola, per i nostri giri in bici e per aver tentato in tutti i modi di vedermi e sentirmi, riuscendoci alla fine. Grazie, Michele e Davide, per le divertenti chiamate Skype in tre parti diverse del mondo. Grazie in particolare, Sara, per i giorni che abbiamo trascorso assieme là e per aver fatto da messaggera. E poi grazie a Ilaria, Simone, Ele, Emi, Aly, Fra, Gio, Maffi e Ricky.

Grazie mille a Chiara e Rossana, compagne di tutti questi anni di università, per aver vissuto assieme non solo lezioni, pause e progetti, ma altrettanti momenti fuori da Via Branze, e inoltre grazie per aver condiviso le stesse speranze e timori, dandoci supporto a vicenda.

Un grazie speciale ad Alba, *luce mela dei miei occhi*, perché essere lontane ci ha soltanto fatto sentire più di quanto già facessimo prima, a partire dalle lettere scritte a mano e arrivando ai messaggi audio: ricevere minuti e minuti di messaggi in cui ascoltarti e poi risponderti era stupendo e lo è tutt'ora.

Alla mia famiglia, Mamma, Papà e Andrea, per avermi permesso di realizzare tanti dei miei sogni, per aver sempre creduto in me e avermelo trasmesso, attraverso abbracci, messaggi, sorprese e chiamate quotidiane; alla Nonna Alceste, per le sue preziose parole e perché è quasi diventata tecnologica pur di vedermi. Grazie di cuore, perché mi date tutto l'amore e la forza necessari, non solo quando sono via ma ogni giorno.

Infine, grazie di cuore, William, perché non riesco a contare tutti i momenti in cui mi hai supportato e incitato, in mille modi possibili, fin dall'inizio, pur sapendo che sarebbe stata dura per entrambi essere distanti migliaia di chilometri. Il tempo insieme a te, amore, ha un valore incommensurabile.

Contents

Sommario

"One always has time enough,
if one will apply it well."
Johann Wolfgang Von Goethe

Quanto vale un secondo, un minuto o un'ora? Quanto incide il *tempo* in generale sull'organizzazione di tutti i giorni? É stato oggetto di dibattiti, studi e teorie da secoli e continua attualmente a esserlo, ma la sua importanza non è solo filosofica. Si può pensare al tempo pure dal punto di vista finanziario, assegnandogli un valore monetario: nel settore economico dei trasporti esiste una grandezza matematica chiamata *Valore del Tempo*, che rappresenta il costo del tempo impiegato da un individuo a spostarsi da un luogo a un altro, corrispondente a quanto egli sarebbe disposto a pagare per risparmiare tempo. A ogni ora si può attribuire un prezzo, definito in base all'identità di chi si sposta e al motivo per cui lo sta facendo.

Il tempo può essere visto come un indicatore e anche come una risorsa da sfruttare, come succede nella logistica industriale e distributiva, dove l'obiettivo consiste proprio nell'organizzare ogni fase del processo industriale in maniera efficiente e nei tempi programmati. Oltre che nella produzione, il tempo è fondamentale specialmente nell'ultima parte della catena, quella in cui merci o servizi sono forniti ai consumatori finali. L'azienda cerca di soddisfare le richieste della clientela e, contemporaneamente, di ottimizzare la consegna in termini di costo, gestendola nel migliore dei modi e senza sprecare tempo o denaro.

Il servizio di consegna può essere organizzato con un *piano* giornaliero, settimanale o mensile, a seconda delle esigenze, con l'obiettivo di impiegare al meglio le risorse della società, quali per esempio fattorini e mezzi di trasporto. Tale piano, per poter essere ottimale secondo i tempi, deve essere progettato avendo a disposizione più informazioni possibili, come il numero di clienti, i loro recapiti e l'entità degli ordini. Si consideri, per esempio, che due o tre clienti abbiano richiesto una consegna e si supponga di sapere che essi vivono nella stessa via o quartiere: conoscere tale dettaglio permette all'azienda di pianificare di servirli tutti in un solo intervallo di tempo, evitando di tornare nello stesso luogo nuovamente. Ciò può essere programmato solo se le informazioni sui clienti sono disponibili a priori.

Dal lato del cliente, la consegna ovviamente avviene all'indirizzo da egli specificato, che può spesso coincidere con quello di residenza, per esempio in caso di ordini di elettrodomestici o comunque oggetti da consegnare e installare.

La spedizione a casa è sfruttata soprattutto nell'*e-commerce* e il numero di acquisti online è in continua crescita negli ultimi anni. Il vantaggio principale per il cliente è di non doversi spostare per ottenere il servizio richiesto, che si verifica nel giorno voluto o comunque concordando la data di consegna con il venditore o l'azienda.

L'orizzonte temporale per effettuare la consegna può essere gestito analizzando pure il territorio, che potrebbe estendersi su una regione o una provincia intera, oppure interessare solo una città, suddivisa in quartieri e strade. Uno schema simile potrebbe essere sfruttato da un'azienda per organizzare le proprie spedizioni: si potrebbe dividere la superficie interessata in zone più piccole, decidendo di fornire il servizio nelle varie aree durante alcuni momenti particolari della giornata, detti *finestre temporali*. In base al numero di clienti in una determinata zona e ai loro ordini, si può effettuare una stima del numero di volte in cui sarebbe necessario visitare la zona, coincidente con la quantità di finestre temporali da assegnarle durante il periodo preso in esame. Calcolando tale valore per tutte le zone, si potrebbe progettare, a mano o automaticamente tramite un software apposito, una tabella di marcia dove, per ogni zona e per tutti i suoi clienti, sarebbero riportati i giorni e gli orari in cui fornire il servizio di consegna.

Ciò diventa molto più complicato quando non si hanno a disposizione informazioni certe sui clienti e sulle loro richieste, ma si conoscono solo dati statistici, come i valori medi dell'entità degli ordini o del tempo impiegato per effettuare il servizio; alcune volte persino le identità dei clienti, e di conseguenza i loro indirizzi, non sono noti. Si possono prevedere alcuni casi possibili, ma senza sapere mai con sicurezza assoluta quale di questi si realizzerà in concreto. Non si potrà perciò progettare un piano perfetto, ma l'obiettivo diventerà quello di trovarne uno in grado di adattarsi bene a ogni eventualità.

La questione trattata nello *Stochastic Multi-period Time Windows Assignment Problem* (SMTWAP) è proprio la seguente: non essendo noti i clienti futuri e i loro ordini, ma solo lo storico delle consegne di una società (e.g., attraverso un database), il piano deve essere progettato prevedendo gli scenari realizzabili in base alle informazioni del passato, tentando di soddisfare i clienti e minimizzando i costi o i tempi totali di consegna; il tutto vedendo il tempo come il *vincolo* principale da rispettare e ottimizzare, tramite le finestre temporali assegnate alle zone del territorio considerato.

Il problema di servire un insieme di clienti rispettando dei vincoli temporali e avendo informazioni stocastiche è già stato discusso in letteratura, ma mai considerando *contemporaneamente* la gestione di zone geografiche e il problema di assegnamento delle finestre temporali. Spesso l'insieme di finestre temporali è noto per ipotesi, ed esse non si possono modificare ma soltanto assegnare a ogni cliente; invece l'approccio risolutivo dello SMTWAP si concentra proprio sulla loro *manipolazione* degli intervalli di tempo e su quella delle zone. Le finestre temporali non sono date ma devono essere definite; non rimangono fisse e costanti durante la risoluzione del problema ma vengono modificate (e.g., spostate in giorni diversi) per provare nuove combinazioni, in modo che diversi piani possano essere testati e valutati. Potrebbe accadere che uno specifico intervallo di tempo non sia adatto per un'area, ma sarebbe ottimale per un'altra. In alcuni casi, è solo il giorno in cui è effettuato il servizio a essere inadeguato e sarebbe meglio provare un periodo diverso. Infine, potrebbe verificarsi la circostanza di dover allargare una finestra temporale al fine di consentire il

servizio di tutti i clienti in un determinato posto nello stesso momento. Anche se sono le zone a essere associate con delle finestre temporali, lo SMTWAP prevede comunque di risolvere un problema di routing dove i nodi sono individuati dai singoli clienti, non dalle zone.

L'obiettivo del problema consiste nel riuscire a determinare il miglior assegnamento possibile di finestre temporali alle varie zone, valutando come esso influisca sul calcolo dei percorsi per le consegne.

Lo SMTWAP può essere visto come un problema di *Programmazione Stocastica* a due stadi (*Two-stage*), dove il primo stadio (*first stage*) è dedicato alla gestione delle zone e alle decisioni da prendere sulle loro finestre temporali, mentre nel secondo stadio (*second stage*), una volta che le informazioni sui clienti e sui loro ordini sono rese note, si risolve il problema di routing collegato, per calcolare i percorsi dei veicoli e i loro costi. Per fare ciò, si sono progettati e sviluppati alcuni algoritmi ad hoc sfruttando delle euristiche, poi confrontate e valutate nell'ultima fase del lavoro di tesi, dove sono stati effettuati dei test sulle istanze generate. Nonostante le euristiche dovrebbero essere perfezionate, in media l'approccio risolutivo migliora le soluzioni iniziale del 60%.

Una delle applicazioni reali più intuitive del SMTWAP può essere la consegna di mobili offerta da un'industria, ma il problema potrebbe essere adattato anche ad altri casi, come il recapito della posta, il calcolo del percorso di scuolabus o mezzi pubblici e la raccolta dei rifiuti urbani. In generale, il problema rappresenta un punto di partenza per tutte le situazioni in cui organizzare e non sprecare il tempo a disposizione è di fondamentale importanza.

La tesi è redatta in lingua inglese e organizzata come segue.

Nel *Capitolo 1* si discutono i più importanti articoli presenti in letteratura e collegati allo SMTWAP. Il problema è definito completamente nel *Capitolo 2*, dove l'approccio risolutivo è ampiamente spiegato e sono forniti alcuni dettagli sull'implementazione. Il *Capitolo 3* è dedicato ai test computazionali e ai risultati ottenuti. Infine, nelle *Conclusioni*, si riportano le valutazioni finali sulle euristiche sviluppate e alcuni suggerimenti per gli sviluppi futuri.

Introduction

"Nothing is a waste of time
if you use the experience wisely."
Auguste Rodin

What is the value of a second, a minute or an hour? How does *time* affect the daily organization? For many and many centuries it has been at the center of debates and studies; a lot of theories have been developed about its meaning, but its relevance is not just a matter of philosophy. Time can have also a financial value: in transport economics there is a mathematical quantity called *Value of Time*, which represents the opportunity cost of the time spent by a person to travel somewhere, which is measured by how much he would pay for saving time. It is possible to assign a price to every hour, according to the traveller's identity and reason to move.

Time can be thought as a useful resource, as in industrial distribution and logistics, where the goal is to efficiently manage every phase of the process, respecting the scheduled times. Beyond production, time is essential above all in the last part of the logistics chain, when goods or services are provided to end users. The company tries to satisfy customers requests and, at the same time, to optimize the shipping in terms of costs, without wasting money or time.

The existence of a *plan* becomes therefore essential, to settle the delivery service with a daily or weekly organization, following the idea of exploiting at most the company resources, such as delivery men and vehicles. To build an optimal timetable, anyway, the company must know exactly how many customers have to be served, where they are located and the amount of their orders. For example, let us consider the case where two or more customers make a request and suppose to know that they all live in the same street or zip code: this allows the company, if practicable, to serve them all in just one interval, avoiding to come back in the same place again. However, this can be done only if details about customers are given in advance.

From the point of view of the customer, usually the delivery occurs at the address he has specified, that is often his residence, specially in case of order of white goods or items to be installed. Home delivery is frequent in *e-commerce* and the number of online purchases has been increasing constantly in the last years. The best advantage for customers is that they do not have to move to obtain the service, that will be accomplished at the time and place established in an agreement with the company.

The time horizon when performing the delivery can be organized focusing on the delivery area, that can be a city or a province, consisting in several towns, composed of zip codes and streets. A company can use this partition to organize its deliveries, deciding to provide the service in a place during particular moments of the day, called *time windows*. These ones should be defined according to the number of customers in that area and their relative requests. The result of making this for every area would be a complete delivery *schedule* for the company. Generally, if all data are available, then a good plan can be constructed, even by hand or, better, automatically through an appropriate software. The problem arises when this information is not known with certainty and only statistics can be used, such as probability distributions or average values of demands or locations. In this case, it is not possible to make a perfect schedule, but the goal turns into finding one which has a good behavior in every eventuality.

This is the question dealt with the *Stochastic Multi-period Time Windows Assignment Problem* (SMTWAP): since no certain information about future customers and orders is available but just some historical data, a schedule has to be built during the considered working period, satisfying customers and, simultaneously, minimizing the total delivery cost; all this has to be done thinking about time as the main *constraint* to satisfy, through time windows assigned to zones of the considered area.

The general problem of serving a set of customers, satisfying time windows and in presence of stochastic information, has been already treated in the past, but never *jointly* considering the geographical aspect of dealing with zones and the time windows assignment problem. Usually the set of time windows is given by assumption, as known data, and they cannot be modified but just assigned to every customer. In the SMTWAP the sets of time windows are not provided at the beginning but they have to be defined and, once built, they do not remain fixed. The solving approach developed in this thesis focuses in particular on the *manipulation* of times intervals and zones. It may happen that a given time interval is not appropriate for a zip code, but it would be optimal for another one; other times, the service day is just wrong and it would be better to try another period. Moreover, it could be the case that a time window should be expanded, to serve all customers in a determined area at the same time. It has to be underlined that in the SMTWAP zones instead of customers are assigned to sets of time windows, but the related routing problem involves nodes that represent customers, not zones.

Given an initial assignment of time windows to zones, some moves are applied in order to find the best practical schedule, corresponding to the best assignment of time windows to zones, evaluating how it affects the solution of the delivery routing problem. The SMTWAP can be treated in the *Stochastic Programming* as a *Two-stage* problem, where the first stage is dedicated to the management of zones and to the decisions to take about their time windows, while in the second stage the related routing problem is solved, once the information about customers and their orders are known. To tackle this, some ad hoc algorithms have been designed and developed using some heuristics, then compared and evaluated in the last part of the thesis, where solutions to generated instances of the problem have been provided, showing both the assignment of time windows to zones and the computed routes with their costs. Even if the heuristics should be more refined, on average initial solutions are improved by 60%.

One of the most intuitive real applications of this problem can be the furniture delivery offered by an industry, but the problem could be also adapted to model many other situations, e.g. postal and mail delivery, school bus routing and waste collection. Generally speaking, the problem can be thought as a starting point for all real cases where it is basic to organize time and not to waste it.

The thesis is organized as follows.

In *Chapter 1*, the most important articles proposed in literature and related to the SMTWAP are discussed. The problem is completely defined in *Chapter 2*, where the solving approach is explained and some details about the implementation of scenarios and solution methods are provided. *Chapter 3* is dedicated to computational test and results. Finally, the chapter *Last Note* concludes the work and reports possible suggestions for future developments.

List of Tables

List of Figures

List of Codes

Chapter 1

Literature Review

"It may be that in the future you will be helped
by remembering the past."
Virgil, Aeneid, Book I, Line 203

1.1 The Vehicle Routing Problem

The *Vehicle Routing Problem* (VRP) was introduced for the first time in 1959 by George Dantzig, creator of the simplex algorithm, and John Ramser [8] with a different name, *The Truck Dispatching Problem*, where the issue was to serve a large number of service stations using some gasoline delivery trucks.

More abstractly, using the terms *customers* and *vehicles* instead of stations and trucks respectively, the problem can be seen as a generalization of the *Travelling Salesman Problem*, wheref the goal is to deliver some goods to a given set of customers exploiting a fleet of drivers.

Every vehicle must start and end its route in a depot, which there can be one or more of in the considered area. The way customers and depots are located can be described with a graph, where every node represents a customer or a depot, while an arcs is the link between two points. The graph can be directed or undirected, and so the arcs. Reaching a point means crossing through the respective arc paying some transportation costs related to its length or to its travel time.

The main goal of the VRP is to find a route for each driver minimizing the total transportation costs, visiting all customers exactly once and satisfying their requests, without violating any constraints. The objective function is often related to the global distance travelled or the global time elapsed, but can be also about other aspects of the problem, for example the number of vehicles needed.

The VRP is a NP-hard problem with multiple applications. Many methods have been developed and implemented to solve it; often, in case of relatively small instances, exact algorithms can be used, but usually heuristics are preferred in practice.

In these fifty years, many variants of the classic VRP have been discussed. Vehicles can be all homogeneous (i.e., they all have the same features of speed, cost and capacity) or the fleet can be heterogenous and so mixed, as described in 1984 by Golden [20] with the *Mixed Fleet VRP* and by Gendreau [19] in 1999.

The vehicle capacity has been used to define another class of problem, the *Capacitated VRP* (CVRP), where vehicles can not load more than a limited quantity of goods.

Some problems allow, if deliveries can not be performed in one visit, to visit customers several times, defining the *Split Delivery* VRP, as illustrated in 1994 by Dror, Laporte and Trudeau [14].

An extension is the VRP with *Deliveries and Pickups*, introduced by Casco, Golden and Wasil in 1988 [5], that can be performed simultaneously or in a different moment, transporting goods also from customers to depots.

Some problem data could be random and in this case problems are called *Stochastic* VRPs: customers' demands, service times and travel times can be modeled through random variables, and also customers can appear requiring a service with a certain probability. According to the distribution functions of the random variables, there can be several realizations.

There exists also a class of VRP problems where customers do not have to be served during just one day but over a time horizon composed of several days or periods, as generalized in 1974 by Beltrami and Bodim [3].

The *Time dependent* VRP was formulated in 1992 by Malandraki and Daskin [26], introducing the concept of a travel time for each arc: the time required to pass through an arc is influenced not only by the speed of vehicle but also by the time of the day and the relative traffic conditions. The travel time and the cost of an arc are not approximated to deterministic constant values depending just by distance, but they can vary along the whole arc due to traffic congestions, weather conditions and accidents. Speed distributions are given at the beginning.

The variant studied in this work is instead the VRP *with Time Windows* (VRPTW): it is about imposing a time window on the visit of each customers, removing some freedom and forcing vehicles to serve customers just during some given intervals.

Time windows can be *hard* or *soft*: the former do not permit to violate the given intervals, instead the latter allow to make deliveries outside the window, before the start or after the end of the window, paying additional costs. According to the type chosen, a solution can be infeasible or not; however, hard time windows can be seen as soft ones, with infinite costs in case of violation.

A vast literature exists on VRPTW providing both exact and heuristic solution approaches. Interested readers are referred to Desrochers *et al.* [11] and to Solomon and Desoriers [33], that both in 1988 wrote surveys about arising routing problems with time window constraints and future perspectives.

Another work was produced in 2001 by Cordeau *et al.* [7], who considered several aspects and extensions of the problem and presented a multi-commodity network flow formulation; Desaulniers *et al.* [10] wrote the latest survey about VRPTW.

1.2 State of the art

In the following, different problems and solution algorithms available in the literature are analyzed. All of them are essential to better understand the problem tackled in this thesis and the approaches implemented to solve it.

1.2.1 The Time Window Assignment VRP

Focusing on the main papers that inspired or are related to the SMTWAP, the most important ones are the *Time Window Assignment Vehicle Routing Problem* (TWAVRP) by Spliet and Gabor [35] and the *Discrete Time Window Assignment Vehicle Routing Problem* (DTWAVRP) by Spliet and Desaulniers [36], published in 2014 and in 2015 respectively.

Both problems are stochastic in terms of customers' demands: there is in fact a finite set of scenarios, where every scenario is a realization of the demand for each customer and is associated with a probability to occur.

The goal of the TWAVRP is assigning a time window to each customer during the period of a day and then constructing a vehicle routing schedule for each scenario, minimizing the expected traveling costs.

A distinction is made between *exogeneous* and *endogeneous* time windows: the former can be imposed by local government (e.g., only daytime hours), while the latter can be chosen by the supplier and the customer, agreeing on a specific time interval.

According to the TWAVRP, time windows are assigned to customers before demand is known, then the VRP is solved, considering the assignment just done in the constraints; feasible routes, identified by the main variables x_r, are selected in each scenario.

Spliet and Gabor [35] model a mixed integer linear program and solve its relaxation that allows nonelementary routes with a Column Generation algorithm:

- the relaxed program corresponds to the *master problem*, where only a subset of variables or routes is considered;
- the subproblem has the purpose to identify new feasible routes to be added to the master problem and solve it again; if no route is found, the current solution is optimal for the master problem.

The objective of the Column Generation is to find lower bounds solving one pricing problem for every scenario; after that, some valid inequalities valid for VRPs are added to the problem. Finally, a Branch-and-price-cut algorithm is proposed to solve the problem.

In the DTWAVRP [36] for each customer there is instead a discrete set of candidate time windows, from which just one has to be selected, and the time horizon is composed of several days.

Another difference with the TWAVRP [35] is that the DTWAVRP is discussed as a *Two-stage* stochastic optimization problem: in the first stage a time

window is assigned to every customer from the set of possible time windows; in the second stage, after demand is known, vehicle routes are computed.

The developed method is still an exact Branch-price-and-cut, from which five Column Generation heuristics are derived to be used for larger instances.

1.2.2 Time Slot Management in Attended Home Delivery

The *Time Slot Management in Attended Home Delivery* (TSMP) tackled by Agatz *et al.* [1] does not consider explicitly time windows but just deals with *time slots*. They focused on the e-grocery business and developed a fully automated approach to produce a time slot schedule for deliveries: time slots are selected to form a set that is offered in each of the zones of a service region. A zone is called *zip code*.

In literature, time slots are typically assumed to be exogenous information; in this case, the company offers to customers a choice of narrow delivery time slots, i.e. a two-hour delivery slot from a set of time slots available in their zip code areas; every time slot has a fee to be paid. The TSMP allocates one time slot for each zip code geographically, prior to actual order intake.

Demand is measured in terms of customer orders and over a time horizon of a week, therefore the expected amount for each zip code is known and independent from the set of offered time slots and it is divided evenly over the set of offered time slots; that means that all time slots are equally popular.

Split delivery is allowed and multiple vehicles can visit the same zip code in the same time slot, but slots can not overlap among different zones.

The purpose is to minimize the expected delivery costs, meeting all service requirements.

Agatz *et al.* [1] use two different approaches to solve the problem:

- *A continuous approximation model*, for estimating the delivery cost of a given time slot schedule assigned to a set of zip codes. Starting from the evaluation of a given time slot schedule, a local search is made to improve the solution found: for each zip code, the time slot allocation with minimum expected delivery cost is computed, keeping time slots assigned to other zip codes fixed. Then the current time slot schedule is adjusted and the procedure is repeated, as long as there is a reduction greater than some threshold or a maximum number of iterations is done.

- *An integer programming model*, using a seed-based scheme, grouping customers of the same zip code into a circle, where the center dot represents the seed. Routing costs for a vehicle are approximated by the sum of the cost of its route through the seeds plus the estimated costs due to the visit of customers inside a circle, considering their distance from the seed.

1.2.3 Heuristics for the Time Slot Management

The TSMP has been also studied developing some appropriated heuristics and metaheuristics, as done in 2014 by Hernandez *et al.* [22].

They model the problem as a *Periodic* VRP (PVRP), considering a period of a week where every day is divided into a number of time slots or *shifts*; each zone has a service frequency corresponding to the number of shifts (i.e., single,

two or three). Two shifts in the same day can not both appear in the same shift schedule. Split delivery is not allowed.

Every customer is associated with a geographical zone (e.g. a zip code area) and every zone is represented with a square, whose side and center depend on the coordinates of the customers' location. The service area is modeled as a complete graph, where every node is a zip code area; demand is known.

The goal is to select a particular shift schedule for each zone, assigning zones that must be served to vehicles on any given day of the time horizon, then sequencing the zones, trying to minimize the total travel cost.

The method presented is a *Unified Tabu Search*, where neighborhood structures are defined using simple moves, such as the removal of a zone from a route in a certain time period and the reinsertion in another route. The idea is the same used in Agatz *et al.* [1]: after evaluating the current solution, a change is performed and the procedure is repeated.

Two heuristics are suggested:

- *Three Phases - A decomposition approach*:
  1. a PVRP is solved over the shifts, assigning them to zones and constructing a set of routes;
  2. exploiting a greedy algorithm, some merge actions are done, connecting the routes that are performed in the same day in contiguous shifts;
  3. every route is optimized with a VRPTW heuristic, solving a VRPTW for each day.
- *Directly*: the problem is seen as a PVRP with time windows (PVRPTW), adapting the shift schedules into full day schedules.

To compare performance results, they generated a new set of benchmark instances.

1.2.4 The Consistent VRP

Another variant of the traditional VRP introduces some restrictions about the service: customers must be visited by the same driver at the same time, every day the client makes a request; the problem is called *Consistent* VRP (ConVRP) and has been defined in 2009 by Groër, Golden and Wasil [21].

The ConVRP considers a time horizon composed of several periods or days and it also includes constraints related to the capacity of drivers, so it can been seen as a Multi-period CVRP.

The objective is to compute routes for each day of the time horizon, being consistent from one day to the next with customers, satisfying all the constraints and minimizing the total traveled time of the homogeneous vehicles over the days.

The problem is formulated as a *Mixed Integer Program* and optimally solved when instances are small; otherwise, in case of large scale ConVRPs, an adaptation of the Record-to-Record travel algorithm by Li, Golden and Wasil [24] is used to create a set of template routes; these are exploited to build feasible routes for each day of the time horizon, applying insertion and removal procedures too.

1.2.5 An ALNS Heuristic for the PDPTW

Ropke and Pisinger [30] tackled the *Pickup and delivery problem with time windows* (PDPTW) designing a very powerful and general framework called *Adaptive Large Neighborhood Search* (ALNS), composed of several competing sub-heuristics, obtained extending the *Large Neighborhood Search* heuristic.

Since it has been decided to implement ALNS as a part of the solution method for the SMTWAP, more details about it are shown in Chapter 2.

Each customer is associated with a couple of time windows: one for delivery, one for pickup operations. Vehicles are allowed to arrive before the beginning of a time window, even if they would have to wait; instead they can not arrive after the end of the time window.

The purpose is to construct feasible routes, minimizing the traveled distance, the elapsed time and the number of unserved requests, that cannot be assigned to any vehicle and ends up in a virtual *request bank*.

The main idea at the core of ALNS is to perform very large moves to search in very large neighborhood explicitly, rearranging up to 30%-40% of all requests in a single iteration.

1.2.6 Stochastic VRP

The problem of dealing with stochastic information and random variables in VRPs raised in the 1980s and was treated in 1983 specially by Stewart and Golden [37], who presented different formulations of the problem and some heuristics. In 1989 Dror, Laporte and Trudeau [12] suggested a Two-stage approach with recourse *a priori* and then with restricted failures in 1993 (see Dror, Laporte and Trudeau [13]).

Gendreau, Laporte and Séguin [18] made a survey in 1996, describing the Two-stage formulation and the main problems related. The Two-stage formulation is examined in depth in Chapter 2.

During the last thirty years, stochastic VRP has been examined on several aspects, according to constraints imposed and techniques implemented.

Herewith a brief selection of recent works that share some features with the SMTWAP is presented.

Taş *et al.* [38] studied the VRP with stochastic travel times, soft time windows and service costs: first they used a meta-heuristic to find good solutions to the problem, implementing a Tabu Search in 2013; then they developed an exact method with Gendreau in 2014, based on a Column Generation and Branch-and-price algorithm (see Taş *et al.* [39]).

In 2014 Gauvin, Desaulniers and Gendreau [17] used a Branch-cut-and-price algorithm for the VRP with stochastic demands, inspired by the work of Christiansen and Lysgaard [6], who had introduced a new exact algorithm for the Capacitated VRP with stochastic demands (CVRPSD), formulating it as a Two-stage stochastic program with fixed recourse and capacity constraints.

In 2015 Archetti, Jabali and Speranza [2] focused on the multi-period aspect of VRPs, considering a fleet of vehicles for each day of the time horizon. They imposed a temporal constraint when customers have to be served, specifying a release and a due date; if it is not possible to guarantee that some customers are served between this interval, additional costs must be paid. A Branch-and-cut algorithm is used to compare different formulations of the problem.

Also Dayarian *et al.* [9] studied the Multi-period VRP, focusing on aspects related to the goods production and considering the seasonal variations in producers' supplies. They designed a mathematical model based on the Two-stage formulation, where in the First-stage a plan is designed over the time horizon and in the Second-stage different periods are evaluated; the problem is solved with a Branch-and-price algorithm.

A recent work about VRP with stochastic demands has been published in January 2016 by Luoa *et al.* [25], that considers also transportation costs proportionated to vehicles weights. They used an a priori optimization approach and suggested a flexible recourse strategy, designing three heuristics for ALNS and comparing them.

Errico *et al.* [15] introduced the problem of VRP with hard time windows and stochastic service or travel times in 2013, solving with a Branch-and-price algorithm and dynamic programming. More recently, the same authors published a work where they formulated the same problem as a Two-stage program (see Errico *et al.* [16]): in the first stage service times are unknown and a priori plan is determined, composed of a set of planned routes; in the second stage, service times becomes known and some recourse actions are applied to modify the plan and make the solution feasible.

Chapter 2

The Stochastic Multi-period Time Windows Assignment Problem

"We are time's subjects,
and time bids be gone."
William Shakespeare, Henry IV,
Part II, Act 2, Scene 1, Line 110

2.1 Introduction

In *Vehicle Routing Problems*, customers are identified by coordinates, indicating their position on a map and are represented as nodes on a graph. Coordinates are also used to calculate the distances among customers, which then are assigned as weight of the arcs connecting the nodes in the graph. In real problems, people's addresses can be used to identify a location, but it has to be considered that in a map there are also houses, streets and districts; taking into account topological constraints makes not so easy supplying customers' orders. It seems reasonable for a company to deliver goods to customers who are in the same streets at the same time, avoiding if possible to come back to the same place in another moment.

A town or a city can be represented as a squared or a rectangular grid, dividing the total area into several parts and identifying some *zones*; consequently customers can be classified according to their addresses or locations, assigning zones to them.

In the city area there can be several warehouses or depots where the company keeps its goods or provisions and where deliveries start from and end to.

A company can have its own fleet of vehicles or it can lean on an external service; in this problem the only relevant fact is that vehicles cost to the company in terms of travel time to deliver goods and money. Usually the fleet consists of homogenous vehicles sharing the same features of size, capacity and speed, but fleets of heterogenous vehicles are also frequent in practical applications.

Sometimes customers explicitly ask for deliveries in particular parts of the day or in a specific day of the week; in e-commerce, usually customers choose among delivery alternatives offered by the company, paying more or less according to the option selected. In other cases they do not care about the moment of delivery and their only requirement is to be served after their orders are done: this is the eventuality considered in the SMTWAP.

Certainly not all customers and areas can be served at the same time; some intervals of time are assigned to every zone, during which customers belonging to that area can be supplied. Zones can have a different number of time windows, but this number can vary among them, depending on customers' demands. The number can be estimated with the historical delivery data hold by the company, i.e. useful statistics about average customer demands, service times and locations.

Following the company and the type of delivery, the service can be performed for example during a single day, a week or a month. The whole period evaluated is called *time horizon*. Considering five working days, the same customers do not have to be served every week, but the set can change according to identities, locations, demands and service times required, week after week.

Time windows are assigned to each zone before knowing customers and their requests to satisfy.

When predicting the requests for the following week, according to historical data, multiple scenarios can happen: all realizable cases should be evaluated, without forgetting also their probabilities to take place in reality.

Every eventuality contains information about the number of customers to serve, their locations on the grid and their demands and service times.

Obviously what matters to the company is trying to minimize the total cost to pay for delivery and to satisfy customers, visiting them in the time windows allocated for their zones. Some penalties have to be paid if a customer is not served during the schedule of its zone.

2.2 Stochastic Programming

Often reality has to deal with uncertainty and it may happen that not all parameters are known for sure but are related to sets of possible cases, mainly derived from historical observations. This aspect is reflected in stochastic problems: differently from deterministic problems, where complete information on data is assumed to be available in any moment, here some variables should be modeled as random.

The approach to use is called *Stochastic Programming* and allows to define appropriate models to include uncertainty, knowing or at most estimating probability distributions associated with random variables.

There are several ways to manage uncertainty, that can affect feasibility and optimality, because a best solution for a particular case would not be good or even practical in other circumstances:

- *Using expected values* - A common method, frequently followed in practice, is that of substituting random variables with their expected values.

Given a deterministic linear program with coefficients of some variables not known with certainty but whose probability distribution are available, their joint distribution is contemplated. This method has some evident drawbacks. Constraints containing these variables can change; moreover, using expected values to replace the coefficients may not provide a solution that is feasible with respect to the random variables.

- *Wait-and-see* - All decisions are delayed until the last possible moment, after all uncertainties have been resolved: the linear programs associated with all possible outcomes of the random quantities ξ are solved.
  This is the so called *Distribution problem*: finding all solutions $x(\xi)$ of the problem for all scenarios and the relative optimal objective values $z(x(\xi), \xi)$; in general none of these solutions are worthwhile or even feasible respect to the outcome. Finally, it is possible to compute the *Expected Value of the Optimal Solution*: $E[\min z(x, \xi)] = E[z(x(\xi), \xi)]$.

Herewith the most common objectives under uncertainty are listed:

- minimization of expected costs, used in large-scale optimization;
- minimization of expected absolute deviations from goals;
- vector optimization, in case of a multi-objective model;
- minimization of maximum costs, when partial or none distributional information is available.

Anyway, it is impossible to find an ideal solution for all possible cases, but decisions have to be balanced among various circumstances or *scenarios*.

A scenario is one of the possible outcome of the random vector that comprehends all the random variables of the problem. The scenario representation is usually adopted when random variables are discrete and correlated; otherwise, if they are independent and continuous, another model is used, using probability density functions.

Usually information about probabilities can be retrieved thanks to historical data, if they are available, otherwise through an accurate estimate, but it is also possible that distributions change with time.

In stochastic programming literature two approaches are studied: future recourse and restriction of the probability of infeasibility (i.e., management of system failures).

2.2.1 Two-stage recourse models

The term *recourse* indicates the corrective actions performed after the realization of a random event: first, there is the modelization of a response for each outcome of the random elements that might be observed, and then the revealed outcomes are adapted, adjusting the solution following recourse rules.

Under uncertainty, it is essential to adopt initial policies that will accommodate alternative outcomes.

Decision variables are explicitly classified according to the moment they are implemented, relatively to the outcome of the random variable observed.

There are two main phases:

- *First stage* is before uncertainty is solved; the variables of this phase are called *proactive* and are associated with planning issues, applied against all realizations, to determine an *a priori* solution;
- in the *Second stage*, after outcomes are revealed, variables are called *reactive* and are associated with operating decisions. This allows to model a response to the observed outcome: if there are some discrepancies, a recourse policy is applied to compensate, imposing a penalty cost but hopefully making solutions still acceptable, even if they cost more. The modifications introduced should be kept in mind in the following steps, while computing again the first stage solution.

Standard formulation

Starting from the formulation of a linear problem,

$$\begin{aligned} &\min c^T x \\ &\text{s.t.} \\ &Ax = b \\ &x \geqslant 0, x \in R^n \end{aligned} \tag{2.1}$$

this is the standard formulation of a stochastic linear problem, as reported in Birge and Louveaux [4]:

$$\begin{aligned} &\min c^T x + E[Q(x, \xi(\omega))] \\ &\text{s.t.} \\ &Ax = b, \\ &x \geqslant 0 \end{aligned} \tag{2.2}$$

ω represents the uncertain data and $Q(x, \xi(\omega))$ is the optimal value of the second stage problem. It is possible to define the *value function* or *recourse function* $Q(x) = E[Q(x, \xi(\omega))] = E_\xi Q(x, \xi)$.

The second stage problem can be modeled as follows:

$$\begin{aligned} &\min q^T y \\ &\text{s.t.} \\ &Tx + Wy = h, \\ &y \geqslant 0 \end{aligned} \tag{2.3}$$

Considering both first stage and second stage problems, constraints can be easily divided into two groups: the ones involving just variables of first stage (i.e., x, called also *immediate*) and the others including random variables.

The rows $Ax = b$ contain only the deterministic parameters, therefore they are for the first stage. Variables used in the rows $Tx + Wy = h$ can be random and are used in the second stage problem.

The goal is to minimize the gap between Tx and h, corresponding to the cost of violations that influence the choice of the first-stage variables x. The vector y represents the feasible set of recourse actions that can be performed to fix the solution after the outcome is revealed, while q the recourse costs. ξ is a vector whose elements are composed from elements of vectors q and h and matrices T and W, that are respectively called *technological* and *recourse* matrices. Therefore, the second stage problem can be considered a penalty for violating $Tx = h$.

Compact form

Using the definition of the value function $Q(x)$, a compact form of a stochastic linear program can be written as follows:

$$\begin{aligned} &\min c^T x + Q(x) \\ &\text{s.t.} \\ &Ax = b, \\ &x \geqslant 0 \end{aligned} \tag{2.4}$$

The Two-stage problem is linear because the objective functions and the constraints are linear.

Deterministic equivalent

Using the scenarios representations in presence of discrete random variables, if the scenarios set is finite, then it is possible to rewrite the standard formulation to obtain the so called *Deterministic Equivalent* (i.e., DE); in particular, if the set of scenarios is K and every scenario $\xi_k = (q_k, W_k, T_k, h_k)$ is associated with a probability p_k to happen, the model becomes the following:

$$\begin{aligned} &\min c^T x + \sum_{k=1}^{K} p_k q_k^T y_k \\ &\text{s.t.} \\ &Ax = b \\ &T_k x + W_k y_k = h_k, k = 1, ..., K \\ &x \geqslant 0, y_k \geqslant 0, k = 1, ..., K \end{aligned} \tag{2.5}$$

As the number of random variables increases, unfortunately also the computational requirements do, because the linear programming problem can not be solved in a reasonable time anymore.

The deterministic equivalent DE is often called the *extensive form*.

Recourse classification and properties

According to the configuration of the recourse matrices W, there are different types of recourse:

- if the recourse matrices are all identical, for every realization $k \in K$, then the recourse is said to be *fixed*;

- when the matrices are the identity ones $[I, -I]$ and the behavior of the model is the same, independently from every k, that is the case of *simple* recourse;
- otherwise, the recourse is called *general*.

The recourse is said to be *complete* if, whatever the choice of first stage decisions variables is, a feasible recourse action is practical for every possible outcome in the set K. There is also a less restrictive property, where recourse are called *relatively complete*, that eliminates the constraint that it should happen for every possible choice of x; instead it is enough to find variables x such that constraints $Ax = b$ are valid.

Nonanticipativity

Another constraint can be added to the formulation of a stochastic linear program, called *implementability* or *nonanticipativity*.

It states that planning decisions must be implemented before any outcome of the random variables is revealed, meaning that first stage variables must be identical in every possible scenario.

The *Wait-and-see* approach is *anticipative* and therefore is not an appropriate decision-making framework for planning, while the *general recourse* is *nonanticipative*.

EVPI and VSS

If the information about the second stage variables are known before choosing the first ones, the situation is called *Perfect Information* and the optimal solution is the mean of all solutions obtained.

The *Expected Value of Perfect Information* (i.e., EVPI) is the maximum amount that has to be paid to obtain in return complete information about the future; therefore, it represents the cost due to the presence of uncertainty. It has to be used when more information might be available, through deeper extensive forecasting, sampling or exploration.

The EVPI can be calculated as the difference between the average of the solutions of every possible scenario in case of perfect information, and the optimal stochastic solution, computed instead using the deterministic equivalent.

Its counterpart is the *Value of Stochastic Solution* (i.e., VSS), which represents the possible gain from solving the stochastic model rather than its deterministic counterpart, because it states the value of knowing and using distributions on future outcomes. It has to be used when no further information about the future is available.

Also the VSS can be computed as a difference, between the optimal stochastic solution and the weighted average performance of the deterministic model, obtained substituting every random parameters with their expectations.

2.3 Problem definition

In this section the definition of the problem studied in this thesis is provided.

Consider a set of zones (e.g., zip codes or small areas) $Z = \{1, \ldots, h\}$, organized in squared or rectangular grid. Let $K = \{1, \ldots, m\}$ be the fleet of available vehicles, all homogeneous and with the same capacity Q. Vehicles provide a service that has to be performed during a time horizon of τ periods, i.e. $T = \{1, \ldots, \tau\}$. All over the grid, there is a single depot, from which all vehicles have to start their routes and to which they have to return at the end, during a particular period. Every driver k is associated with a route r_k with cost c_{r_k}, corresponding to the time it spends outside of the depot. Let $W = \{1, \ldots, u\}$ be the set of time windows available during the time horizon T. Every time window w is defined as an interval $\delta[s, e]$, where s and e are respectively the start and the end time of the window, during the period δ. Every zone z has to be associated with a set of time windows W_z, such that $W_z \subseteq W$ and $W_z = \{w_{1_z}, \ldots, w_{\mu_z}\}$, where μ_z is the number of time windows assigned to z and it can vary among different zones; μ_z can be at most equal to τ, because for each period there can be just one time window assigned to the zone z.

Let $V = \{0, 1, \ldots, n\}$ be the set of customers, where node 0 represents the depot. At the beginning of every week, when time windows are assigned to each zone, customers to be served are unknown. V is not a fixed set of customers but could change week by week, in terms of locations, demands and service times; this means that the number of customers to serve and the total amount of demand are unknown until orders are done. A customer v is associated with a unique zone z and in particular with a location (identified by the couple of coordinates (x_v, y_v)); it is then characterized by a demand d_v and a service time s_v. None of the customers specifies a time window during which it would like to be reached; it is enough for them to be served during the time horizon.

Let Ω be a set of scenarios, where each scenario ω represents a realization of the set of customers (their locations, demands and service times and also the zones which they belong to), with a probability p_ω to occur in reality; suppose that scenarios are built using historical information. Always according to historical information, for each zone z a minimum number of visiting times is established, i.e. a minimum number of time windows to assign to z during the time horizon. For every scenario ω, an appropriate set of vehicles $K_\omega = \{1, \ldots, m_\omega\}$ is given and it is possible to define a complete graph $G_\omega = (V_\omega, A_\omega)$, where:

- $V_\omega = \{0, 1, \ldots, n_\omega\}$ is the set of customers to serve in scenario ω plus the depot, identified by the node 0, whose location (x_0, y_0) is fixed, independently from scenarios, and its demand d_0 and service time s_0 are zero. Demand and service time of customer v in scenario ω are indicated with the terms d_v^ω and s_v^ω, respectively.

- A_ω is the set of arcs (i, j) such that both i and j belong to V_ω.
  Every arc is characterized by two values: its travel time t_{ij} and its travel cost c_{ij}, representing respectively the time required to go through the arc and the cost to pay; both these quantities satisfy the triangle inequality.

The main objective of the SMTWAP is to build a schedule φ, assigning a set of time windows to each zone, with the purpose of minimizing the expected travel cost during the time horizon, considering the total traveled time and distance. For each scenario $\omega \in \Omega$, the following constraints have to be satisfied:

- every zone z is visited at least a number of times, decided according to historical information or to the scenario ω;
- every customer v of scenario ω has to be visited within one of the time windows assigned to its zone z, otherwise a penalty will be paid.

The schedule φ is associated with a cost C, corresponding to the weighted average of the costs of single scenarios, identified with c_ω:

$$C_\varphi = \sum_{\omega \in \Omega} p_\omega c_\omega. \tag{2.6}$$

2.4 Two-stage SMTWAP

The SMTWAP can be treated in the context of Stochastic Programming because the set of customers, their demands and service times are random variables, depending on the scenario taken into consideration.

It is possible to adopt the formulation of a Two-stage problem with general recourse, where the first stage deals with the assignment and management of time windows to the zones in the given grid; whereas the second stage problem consists in a *VRP with Time Windows* for each scenario in the set given as input; every VRPTW has to be solved respecting all constraints, specially the fact that a customer has to be served during a time window assigned to his zone.

According to the routes computed during second stage, some changes are applied to the list of zones in first stage and the VRPTW is solved again, obtaining a new current solution.

A solution for the SMTWAP is called *Schedule* and is composed of:

- the list of zones in the grid, with their corresponding set of time windows;
- the average cost of the given set of scenarios;
- for every scenario, information about its probability to happen, the total cost and, for every period δ of the time horizon, a full description of the computed routes of vehicles.

More details about implementation are shown at the end of this Chapter, while SMTWAP instances and results are presented in Chapter 3.

2.4.1 First stage

Hereafter the main algorithms utilized during the first stage are described: the approach developed is an appropriate version of the *Variable Neighborhood Search* (VNS), whose main purpose is to move and modify the time windows of the zones, obtaining lists of zones with different assignments.

VNS The meta-heuristic method called *Variable Neighborhood Search* was invented in 1997 by Mladenović and Hansen [27].

Consider a minimum problem; VNS starts with an initial solution and tries to improve it visiting a sequence of *neighborhoods*. A neighborhood of a solution x is the set of all solutions obtained applying a particular move to the current solution; a finite set of these structures must be determined, to be used during the search.

Neighborhoods $N_1, \ldots, N_{k_{max}}$ are systematically changed during the search: there is not a direct trajectory but their exploration proceeds gradually towards more distant solutions. The new solution found is accepted and substitutes the current one if an improvement is made. To find local optima within neighborhoods, local search methods are used.

First, VNS defines a small neighborhood of the current solution, with dimension $k = 1$. Within this structure, perturbing the current solution it finds another feasible one; then it explores, with a local search, the neighborhood of the new solution found:

- if no improvement is made, the method comes back to the previous solution and it increases the size of the neighborhood, with $k = 2$;
- otherwise, if a local optimum is found, better than the best solution, then it is accepted; the search restarts from it, with $k = 1$.

During the local search, the neighborhood can be examined completely (i.e. applying an highest descent heuristic and looking for the best possible solution) or it can be analyzed partially, just stopping at the first improvement found, in comparison to the initial solution (i.e., first descent heuristic).

The pseudocode of the classic VNS is reported below where it implements a random descent with a first improvement heuristic in the *Local Search*.

Code 2.1: Pseudocode of the classic VNS

```
1  Repeat until the stopping rule is satisfied:
2  {
3    k = 1;
4    Repeat until k = k_max:
5    {
6      a) SHAKING: generate a random x_first in the Nk of x;
7      b) LOCAL SEARCH: starting from x_first, look for a local optimum
           x_second using a local search method;
8      c) MOVE OR NOT:
9        if the local optimum found is better than current best solution
10       {
11         move from x in x_second (x = x_second);
12         restart searching with N1(x) (k = 1);
13       }
14       else
15         k++;
16   }
17 }
```

If no better solution is found, then the method restarts from the best solution and considers the following neighborhood in the sequence; otherwise, if an improvement is made, the new best solution is saved and the research comes back to visit the first neighborhood N_1.

The stopping criteria are the number of iterations done and the computational time elapsed.

Features of the ZonesVNS

VNS was chosen for first stage because of its simplicity and ability to be adapted to several problems; moreover, it is easy computing and visiting neighborhood solutions. The variant developed on purpose for this problem is called *Zones Variable Neighborhood Search* (ZonesVNS), since it is focused on zones and their sets of time windows.

Possible moves Several actions can be performed on zones and time windows to modify them and obtain a different list of zones.

Starting from the current schedule, applying a move allows to define a neighborhood of solutions to evaluate.

Herewith the main moves are listed:

- *Exchange Time Windows Slots* - It chooses randomly two time windows of two random different zones in the list and exchanges their opening and closing times, without modifying their days;
- *Exchange Time Windows Slots Of Selected Zones* - It does the same operation of the previous move, but just one zone is random while the other selected is the worst one;
- *Exchange Time Windows* - It swaps completely two random time windows of two random different zones, exchanging their days and opening times;
- *Exchange Time Windows Of Selected Zones* - It does the same operation of the previous move, but zones are not random;
- *Expand Random Time Window* - It increases the duration of a random time window, without violating the depot opening and closing time constraints;
- *Expand Selected Time Window* - It does the same operation of the previous move, but the time window is not random;
- *Expand Start Time* and *Expand End Time* - One of them is called after *Expand Random Time Window* or *Expand Selected Time Window*; they allow to add one hour to the considered time window, respectively opening sixty minutes before the current start time or closing an hour later after the current end time;
- *Move Random Time Window* and *Move K Random Time Windows* - The former permits to move a random time window of a random zone to a different day; the latter performs the same, but it does multiple times, relocating K time windows;

- *Move Random Slot* and *Move K Random Slots* - Similarly to the previous ones, the former acts just on a single random time window, not changing the day but moving the opening and closing times to a different slot; the latter works on K time windows;
- *Move Worst Time Window* - It moves the worst time window to another day, keeping its opening and closing times fixed.
- *Move Worst Slot* - Similarly to the previous one, this move operates on the worst time window, relocating it to a different time slot during its same day;
- *Reduce Random Time Window* - It is the opposite move of *Expand Random Time Window*, in fact it decreases the duration of a random time window;
- *Reduce Start Time* and *Reduce End Time* are the correspondent moves of *Expand Start Time* and *Expand End Time*, i.e. the former postpones the start of the time window of an hour, while the latter anticipates is end time.

Initial solution and Setup The ZonesVNS starts with an initial schedule completely random: according to historical data, for every zone the minimum number of time windows to serve all customers is computed; therefore, a set of time windows with the proper cardinality is assigned to each zone.

Since the built schedule can be very bad in terms of quality, a *Setup* phase has been added before launching the real ZonesVNS, trying to improve the initial solution. It consists in applying some moves sequentially for a certain number of times, moving from the best solution to the current found only if an improvement is made.

Three different moves can be applied to the current list of zones and they are chosen randomly among:

- moving a time window to another day (*Move Random Time Window* or *Move Worst Time Window*);
- relocating a time window to another slot, without changing its day (*Move Random Slot* or *Move Worst Slot*);
- expanding a time window (*Expand Random Time Window* or *Expand Selected Time Window*).

The set of neighborhoods The neighborhood structure is based on the *Exchange Time Window* move: applying this once or several times to the current solution, the sets of time windows of a couple or more of zones are modified, obtaining x' in the k-neighborhood of the current solution x.

The maximum number of exchanges that can be done corresponds to the number of zones divided by two, since a single swap involves two zones and therefore there can be just as many couples as the half number of zones.

Neighborhoods are visited increasing the number of exchanges by one at each iteration, if no improvement is found.

Shaking The aim of this phase is to get a random solution in the considered neighborhood; no time window is transferred to another day or swapped again with another one. The move executed to obtain a close solution is *Move K Random Slots*: K time windows are just shifted to different opening and closing times, keeping their days fixed.

Local search The ZonesVNS local search is not based on just one move, but three in the list of moves are used: expanding the duration of a time window, exchanging intervals between two time windows of different zones and relocating a time window to another day. Starting from the current solution, all three local searches are launched and each of them returns its own first improvement of *x'*; the schedule *x"* will be the best among the three results.

Move or not The comparison between the result of local search and the *best* solution found is done checking if the cost of second stage of *best* is higher than the one of *x"*: if so, then *x"* is accepted and the sequence of neighborhoods are visited again from the first one; otherwise, if the *best* cost is still lower, then the research visits the following neighborhood in the set.

Stopping criteria The implemented rules to stop the ZonesVNS are the maximum number of iterations or the maximum number of swaps reached, and the maximum computational time elapsed.

Pseudocode of the ZonesVNS

Hereafter, following the same style of the classic VNS, the pseudocode of the ZonesVNS is provided.

Code 2.2: Pseudocode of the ZonesVNS

```
1  Repeat until the stopping rule is satisfied:
2  {
3    1) SETUP PHASE: obtain the current best solution;
4    k = 1;
5    Repeat until k = k_max:
6    {
7      2a) MOVE TO NEIGHBORHOOD: generate x in the Nk of current;
8      2b) SHAKING: generate a random x_first from x;
9      2c) LOCAL SEARCH: starting from x_first, for each kind of
           neighborhood where doing a local search, look for the first
           improvement and then keep the best result found as x_second;
10     2d) MOVE OR NOT:
11       if x_second is better than the current best solution
12       {
13         move from best in x_second (best = x_second);
14         restart searching with N1(x) (k = 1);
15       }
16       else
17         move to the next neighborhood in the sequence Nk (k++).
18   }
19 }
20 return best;
```

ZonesVNS Moves Summary

This part summarizes how the ZonesVNS moves are classified, dividing them according to the method they are used in.

Table 2.1: Classification of ZonesVNS moves

ZonesVNS Method	**Moves**
Setup	*Move Random Time Window*, *Move Worst Time Window*, *Move Random Slot*, *Move Worst Slot*, *Expand Random Time Window*, *Expand Selected Time Window*, *Reduce Random Time Window*, *Reduce Selected Time Window*
Set of neighborhoods	*Exchange Time Windows*
Shaking	*Move K Random Slots*
Local search	*Move Random Time Window*, *Move Worst Time Window*, *Expand Random Time Window*, *Expand Selected Time Window*, *Exchange Time Windows Slots*, *Exchange Time Windows Slots Of Selected Zones*

2.4.2 Second stage

Similarly to what has been done for the first stage, in the following parts the main algorithms and techniques of the second stage are fully explained. As anticipated before, the framework exploited to solve the second stage problem is *ALNS*, herewith thoroughly analyzed.

The ALNS Framework

If the core of the first stage is the ZonesVNS, its correspondent in the second stage is the heuristic developed by Ropke and Pisinger [30] for the *Pickup and Delivery Problem with Time Windows* (PDPTW) and then adopted also for other problems, as described in [28].

Given a solution composed of several routes, the way to improve it usually involves moving a single request from the current vehicle to another one, or the move can be relative to two requests assigned to different drivers, to be exchanged. Every new solution found in such cases is not so dissimilar from the previous one, because just a small change happened.

The main idea in the *Adaptive Large Neighborhood Search* is the opposite: applying very large moves to visit bigger neighborhoods, in order to stir more the current solution in just a single iteration of the framework. Ropke and Pisinger [30] estimated to change the 30%-40% of requests in one step.

Paying something in terms of computational time, the evaluation of not standard moves allows to get higher diversification and to find very good quality solutions.

ALNS is the extended version of the *Large Neighborhood Search* (LNS), heuristic developed by Shaw [32] in 1997 where, starting from a given initial solution built with a simple construction heuristic, a certain number q of requests are removed and then reinserted in different positions, using several insertion heuristics. ALNS differs from LNS because it does not use near-optimal methods for removing and reinserting requests but it implements a set of competing sub-heuristics composed of several techniques for removal and insertion, to use during the same search.

At each iteration, just one removal heuristic and just one inserting heuristic are chosen among those available; the way selection is done depends on their *behavior* during the past iterations. The behavior of every sub-heuristic is indicated by a value called *weight*. When a sub-heuristic is selected during an iteration, it gains some *score* depending on the quality of the new solution obtained; after a certain number of iterations called *segment*, weights are updated according to scores, following the *Adaptive Weight Adjustment* procedure; then scores are reset to zero.

Roulette wheel selection principle The *roulette wheel selection principle* is used to pick heuristics to apply: if the total number of sub-heuristics is k and the weight of the sub-heuristic j is w_j, then it is chosen with probability

$$\frac{w_j}{\sum_{i=1}^{k} w_i}. \tag{2.7}$$

The set of sub-heuristics ALNS is *adaptive* because the dimension q of the neighborhood is not fixed and also because of the set of sub-heuristics to select. In the version of Ropke and Pisinger [30], three removal sub-heuristics are implemented:

- *Shaw Removal*, which takes off first a random request and computes the *relatedness measure*, i.e. the similarity among other requests and the random extracted one, then it removes the most similar requests;
- *Random Removal*, that selects requests randomly;
- *Worst Removal*, which removes the worst requests in terms of the cost required to serve them.

Two instead are the insertion sub-heuristics:

- *Basic Greedy*, that inserts requests at their minimum cost positions;
- *Regret*, the look-ahead version of *Basic Greedy*, which considers the difference between the cost of inserting the request in its best route or in another position. This sub-heuristic follows the scheme suggested by Potvin and Rousseau [29]: every customer's request is evaluated calculating its insertion costs in k routes and finding the best ideal route, i.e. the one with the lowest impact on the objective function. The integer k represents the number of routes considered in the evaluation and the *regret value* is defined as the sum of the differences between the best minimum cost and every other cost. Every request is associated with a heuristic value; the one with the largest regret is then inserted in its best route because, otherwise, if the related customer is assigned to a different driver, then the cost to pay would be very high. This is the meaning of *regret*: the higher it is, the more expensive the consequence of not allocating the request in its best computed route would be.

Stopping and acceptance criteria The procedure stops when a certain number of iterations have been completed, while the *Simulated Annealing* is used as accepting criterion: given the current solution s with *f(s)* as objective function and the new obtained solution *s'* with *f(s')*, *s'* is accepted with probability $e^{-(f(s')-f(s))/T}$. T is the *temperature*, i.e. a global time-variant value, set high at the beginning and decreased at each iteration. *Simulated Annealing* allows to accept solutions that are worse than the current one, avoiding to be stuck in local minima.

Sub-heuristics for the SMTWAP

For solving the SMTWAP, both the *Basic Greedy* sequential insertion sub-heuristic and the *Regret* are exploited; in particular, not a single *Regret* sub-heuristic is used, but more with several values of k, i.e. considering different numbers of routes where to insert the request.

About the removal sub-heuristics, beyond the *Random Removal*, a variant of the *Shaw Removal* and another one have been designed for the SMTWAP especially: *Remove Related Zones* and *Remove Smart*.

Remove Related Zones It is a variant of the *Shaw Removal* sub-heuristic that removes requests of customers located in the same zone.

Between two customers i and j, a new *relatedness measure R(i, j)* is defined as follows, considering that nodes in the same zone are closer in comparison to others in different areas.

$$R(i,j) = \begin{cases} \text{distance between } i \text{ and } j, & \text{if } i,j \in \text{the same zone } z; \\ \text{distance between } i \text{ and } j + highvalue, & \text{otherwise.} \end{cases} \tag{2.8}$$

If i and j belong to different zones, then an *highvalue* is added to the relatedness measure to discriminate the request j.

When this sub-heuristic is selected to remove q customers from the current solution, the first request i to remove is selected randomly; for every other request j, its relatedness measure *R(i, j)* is computed. According to these results, requests are then sort and only the $(q-1)$ ones with lower values (i.e., the $(q-1)$ closest to i) are removed.

Remove Smart This sub-heuristic provides an improved version of *Worst Removal*, because it takes into account the time constraints of this problem. Given a route of a driver, *Remove Smart* sorts its requests according to the *temporal gain* they would make the driver earn if removed. Considering the request i, the *temporal gain* associated with the removal of i includes:

- the time required to insert i in the route and reach it;
- possible waiting time before the opening of its time window, to serve it.

After sorting all requests according to this value in a decreasing order, various choices can be done:

- removing the first request in the ordered set would mean taking off the best request, i.e. the one which requires less time, and it could be done to diversificate;
- instead, picking the last request would imply removing the one that takes more time to the vehicle;
- also removing a random request can be done, providing an analogous version of *Random Removal*.

Procedure

To solve the VRPTW defined in the second stage of the SMTWAP, at the beginning an initial solution is obtained, using the *Regret* sub-heuristic, to try to assign a driver to every customer, and ALNS with a high temperature to diversificate and get a good solution.

After computing the initial routes for all vehicles during the considered time horizon, ALNS is launched, setting the temperature to a very low value to be almost sure to accept a solution, if its objective function is better than the current *best* one.

Starting from the current solution, at each iteration the number q of requests to remove is computed, keeping into consideration how many customers are assigned to drivers and therefore how many of them can be removed at most.

Then, among the sub-heuristics implemented in ALNS and using the *Roulette wheel selection principle*, a removal operator and an insertion operator are selected and applied, to take off q requests from the routes and re-insert them in different positions, obtaining the new *current* solution and its related cost.

Depending on the quality of the new solution, the selected sub-heuristics earns some *scores* that will be needed to update their weights every N iterations.

Finally, it is possible to compare the cost of the *current* solution produced with the one of the *best* solution: if it is better, the method replaces the best solution with the one just found and restarts from it at the following iteration.

Pseudocode of ALNS

Hereafter the pseudocode of ALNS is provided, inspired by the pseudocode of the LNS heuristic described in [30].

Code 2.3: Pseudocode of ALNS

```
1  Function ALNS(solution s)
2  {
3    best = s;
4    Repeat until the stopping rule is satisfied:
5    {
6      current = s;
7      1a) DECIDE the number q of requests to remove/insert;
8      1b) SELECT a couple of sub-heuristics to remove/insert requests;
9      1c) REMOVE q requests from current with removal sub-heuristic;
10     1d) INSERT removed requests into current with insertion
           sub-heuristic;
11     1e) MOVE OR NOT:
12     if ( f(current) < f(best) )
13       best = current;
14     1f) ACCEPTANCE:
15     if ( accept(current, s) )
16       s = current;
17   }
18   return best;
19 }
```

Cost of the second stage

To evaluate the quality of a schedule, the cost of the second stage has to be computed, because it permits to compare several solutions, obtained assigning different sets of time windows to the zones in the grid.

Cost of a route Every scenario has a fleet K of drivers; the cost of one of them is the cost of its route, performed in a particular day, and corresponds to the time it spends outside of the depot: service times of requests are not

included because unavoidable, while the waiting times for openings of time windows and the minutes required to move from a location to another are counted.

If the driver k leaves the depot at a certain time go_time and comes back at arr_time, after serving every customer i in its route r_k, then its cost c_{r_k} can be computed as follows:

$$c_{r_k} = (arr\_time - go\_time) - \sum_{i \in r_k} s_i. \tag{2.9}$$

Cost of a scenario The cost c_ω of scenario ω is directly calculated, taking into consideration every driver k of the set K_ω and their routes (i.e., all drivers assigned to every period δ in the time horizon T):

$$c_\omega = \sum_{k \in K_\omega} c_{r_k}. \tag{2.10}$$

Total cost The most important result to compute is the weighted average among the objective functions of scenarios ω, considering their probabilities p_ω to happen. C_φ represents the cost of the second stage.

$$C_\varphi = \sum_{\omega \in \Omega} p_\omega c_\omega. \tag{2.11}$$

2.5 Implementation

Hereafter in this section a full description is provided, about implementation choices for methods and algorithms.

It must be observed that the code has not been totally implemented from zero, but a package for solving a simple Capacitated VRP was available, with an implementation of the ALNS framework as described by Ropke and Pisinger [30] and Pisinger and Ropke [28].

The package did not include any temporal constraints and, above all, it was designed for just one deterministic problem. Since the programming language used was C++, practical for being object-oriented and permitting the use of templates, the same language has been also chosen for solving the SMTWAP. It has been used especially for setting up the second stage of the SMTWAP, developing an appropriate version of the code and implementing all the needed classes.

The starting code was made available by Prof. Jean-François Côté, professor in the *Operations and Decision Systems Department* at Université Laval (Québec, Canada) and member of CIRRELT (*Centre Interuniversitaire de Recherche sur les Réseaux d'Entreprise, la Logistique et le Transport*).

2.5.1 Main objects

In this subsection the most important objects are introduced, to define them and to give an idea to readers, before explaining in details the implementation of the two stages of the SMTWAP.

ScenarioSet

The class *ScenarioSet* represents literally the set of all scenarios revealed at the beginning of the second-stage, all of them associated with the same vector of *ZoneSMTWAP*, i.e. the zones in the grid.

Every scenario is an instance of the class *Prob<Node, Driver>*, i.e. the problem definition, and it is linked to an *Optimizer*, a VRPTW solver based on *ALNS*, which contains also the *Solution* of the problem, with the list of routes necessary to satisfy all customers' requests.

Code 2.4: Attributes of ScenarioSet

```
1  std::vector<ZoneSMTWAP> & list;
2  std::vector< Prob<Node, Driver>* > scenarios;
3  std::vector< Optimizer* > optimizers;
4  std::vector<double> results;
5  std::vector<int> nb_drivers;
6  std::vector<int> empty_routes;
7  std::vector<int> customers_not_served;
8  double depot_opening;
9  double depot_closing;
```

Other attributes are:

- *results*, a set of costs of all scenarios, i.e. the objective functions values obtained solving the VRPTWs;
- *nb_ drivers*, the number of drivers in every scenario;
- *empty_ routes*, the number of drivers not used (i.e., their routes are empty and do not contain any customer to serve) in every scenario;
- *customers_ not_ served*, the number of customers not served, hopefully a vector of zeros.

The class *ScenarioSet* contains all needed operations to handle the object.

Prob<Node, Driver>

This class was already in the original version of the code.

Code 2.5: Problem template class definition

```
1  template <class NodeT, class DriverT>
2  class Prob
3  {
4    // Problem methods and attributes
5  }
```

A template class is chosen because the aim is to implement the most possible abstract VRP, with generic customers and vehicles, in fact entities *NodeT* and *DriverT* are not specific for a particular variant of a VRP; the use of a template allows not to indicate the exact datatype of the class but anyway to define its behavior, such that classes can be more exploited.

Code 2.6: Attributes of Prob

```
1  std::vector<NodeT> _nodes;
2  std::vector<DriverT> _drivers;
3  double ** _distances;
4  double ** _times;
5  int _dimension;
6  double _probability;
```

Prob is defined by the following list of attributes:

- the vector *_nodes* is the set of customers and depots (*depots* is in plural form because for every driver two instances of the same depot are created);
- the vector *_drivers* contains all required vehicles;
- *_distances* is the distances matrix, where the element $\_distances[i][j]$ represents the Euclidean distance between nodes i and j;
- similarly to *_distances*, *_times* is the times matrix, where the element $\_times[i][j]$ indicates the minutes required to travel the distance between nodes i and j;

- the integer *_dimension* represents the number of rows and columns in the matrices; in the scenario ω, it corresponds to $n_\omega + 1$, i.e. the number of customers plus the depot;
- the double *_probability* is the chance to happen for the scenario.

NodeSMTWAP

This is the implementation of the template class *NodeT*, defined appropriately for the SMTWAP.

Code 2.7: Attributes of NodeSMTWAP

```
1  // Node IDs
2  int id;
3  int origin_id;
4  int dist_id;
5
6  // Node data
7  char type;
8  int demand;
9  int serv_time;
10 int x;
11 int y;
12 int zone;
13 double slack;
14 double arr_time;
15 double arr_est;
16
17 // Just for depots
18 double opening_time;
19 double closing_time;
```

A SMTWAP node can be a customer or a depot; according to the type, it can have or not different attributes:

- *id* is the unique identifier of the node; if the node is a customer, it is a value from the $[0, n_\omega - 1]$, otherwise it is from $[n_\omega + 2 \cdot m_\omega]$, where n_ω and m_ω are, as defined in Section 2.3, the numbers of customers and vehicles in scenario ω;
- *origin_id* is the original number associated with the node, read in the instance or in a database; only customers can have it;
- *dist_id* indicates the index position of the node in the symmetric *_times* and *_distances* matrices (e.g., $\_distances[dist\_id_i][dist\_id_j]$ returns the distance between nodes i and j);
- to distinguish which kind the node is among a customer, a start depot or a end depot, the char *type* is used;
- *demand* and *serv_time* contain information about the customer's order;
- the couple (x, y) indicates the customer's location, its coordinates on the grid;

- the integer *zone* is the id_z of the zone z where the customer belongs;
- *arr_est* and *arr_time* correspond respectively to the time the driver reaches the node and when it is ready to serve the customer, considering the opening time of its time window; both of them do not include the service time;
- just for the depot nodes, there are *opening_time* and *closing_time* which represent the opening times of the depot.
- the amount *slack* indicates how long in minutes the delivery to the current node can be delayed, such that none of the time windows of the following customers in the route will be missed, including the depot closing time; the concept of *forward time slack* was introduced by Savelsbergh [31].

  Every time the cost of a route is computed, slacks of all nodes assigned to that vehicle are updated, because they are exploited to evaluate if a request can be inserted into a route or not. Considering the node i and the time window $[s, e]$ when it is served, its is computed as follows:

$$\begin{aligned} slack_i = max(0,\, s - arr\_time_i) + \\ min[(e - max(s,\, arr\_time_i) - s_i),\, slack_j] \end{aligned} \tag{2.12}$$

  where arr_time_i is the arrival time to the node i, s_i is its service time and $slack_j$ is the slack of the following node j. Slacks are computed backward, starting from the last customer in the route.

DriverSMTWAP

Also a class for the specific implementation of *DriverT* is provided.

Code 2.8: Attributes of DriverSTMWAP

```
1  int id;
2  int day;
3  int startnode_id;
4  int endnode_id;
5  int capacity;
6  int sum_demand;
7  double cur_distance;
8  double cur_duration;
9  bool is_feasible;
10 std::vector<int> used_tw;
```

- *id* represents the unique identifier for vehicle k, that belongs to the set $[0,\, m_\omega - 1]$;
- *day* indicates when the driver works; it can assume the values from 0 to 4 (i.e., from Monday to Friday);
- *startnode_id* and *endnode_id* are the identifiers of the start and end depots of the vehicle;
- the amount *capacity* indicates how much demand the vehicle can carry;

- *sum_ demand* instead corresponds to the current demand carried by the vehicle;
- *cur_ distance* and *cur_ duration* are the space the driver has traveled until the moment and how long;
- the boolean *is_ feasible* indicates if the route can be performed or not;
- during its work day, the driver follows a route to serve some customers, traveling through several zones in their appropriate time windows: the vector of integers *used_ tw* is the set of the identifiers of the time windows visited in the vehicle route.

SolutionVRPTW

Also the class *SolutionVRPTW* was in the original package and it is based on the same template classes of *Prob*, i.e. *NodeT* and *DriverT*.

A solution is basically composed of the list of routes, one for each driver, plus the list of the unassigned customers (obviously at the beginning, before solving the VRPTW for the first time, all customers are marked as unassigned).

Code 2.9: Attributes of SolutionVRPTW

```
1  std::vector<NodeT*> Next;
2  std::vector<NodeT*> Prev;
3  std::vector<DriverT*> AssignTo;
4  std::vector<int> RoutesLength;
5
6  Prob<NodeT, DriverT> * _prob;
7  int unassigned_count;
8  std::vector<NodeT*> unassigneds;
9  std::vector<int> unassigned_index;
10 CostFunction<NodeT,DriverT> * _cost_func;
11 double _last_cost;
12 bool _is_feasible;
```

An instance of *SolutionVRPTW* has the attributes described as follows:

- vectors *Next* and *Prev* are used to know the next and the previous nodes of a node in a route; for example, if the *id* of a node is e, it can be used as index to find $Next[e]$, i.e. the node who follows e in the route;
- *AssignTo* contains the pointers to the drivers which nodes are assigned to; for example, given the index e, the element $AssignTo[e]$ returns a pointer to the driver which serves the node with id e;
- for each route, the number of customers is stored in the vector of integers *RoutesLength*;
- the pointer *_prob* is the reference to the scenario or problem associated with the instance of *SolutionVRPTW*;
- *unassigned_ count* is the number of customers currently not assigned to any driver, while *unassigned_ index* contains all their identifiers;

- *_cost_func* is an instance of the class *CostFunctionSMTWAP*, which implements some useful methods to compute the current cost of the *SolutionVRPTW* and update drivers' routes and attributes;
- *_last_cost* corresponds to the last value computed by the *CostFunctionSMTWAP* instance;
- as for drivers, the boolean value *is_feasible* indicates if the solution is practical or not.

All required methods to handle the pointers in the lists are implemented.

ZoneList

Code 2.10: Attributes of ZoneList

```
1  std::vector< ZoneSMTWAP > zones;
2  int last_id_tw;
```

The class *ZoneList* represents a manager for the zones in the grid of the SMTWAP. Its only attributes are:

- *zones*, a vector of *ZoneSMTWAP* instances;
- the integer *last_id_tw*, i.e. the last unique identifier used for time windows.

This class contains all necessary methods to handle the zones, especially the moves exploited in the ZonesVNS.

ZoneSMTWAP

It implements a zone in the grid and hereafter follows the main features:

Code 2.11: Attributes of ZoneSMTWAP

```
1  int id;
2  int origin_id;
3  TimeWindowSet * set_tw;
4  double score;
5
6  // Historical data
7  int historic_nb_cust;
8  int historic_demand;
9  int historic_service;
```

- id_z is the unique identifier for the zone z, whose value can be $\{0, \ldots, h-1\}$;
- the original number or code identifier associated with the zone is stored in the field *origin_id*;
- the attribute *set_tw* points to an instance of the class *TimeWindowSet*, manager of the set of time windows assigned to the zone;

- the double *score* is an indicator of the quality of the set of time windows assigned to the zone: the higher it is, more the time windows of the zone are exploited and therefore good;
- integers *historic_ nb_ cust*, *historic_ demand* and *historic_ service* are the historical data relative to the zone, indicating respectively the average number of customers $\bar{n_z}$ belonging to the zone who made an order in the past, their average demand $\bar{d_z}$ and average service time $\bar{s_z}$ required to deliver their goods.

TimeWindowSet

Similarly to the *ZoneList* class, this represents the manager for a set of time windows, with all required methods; its only attribute is the vector *windows*, where each element is a pointer to an instance of the class *TimeWindowSMTWAP*.

Code 2.12: Attribute of TimeWindowSet

```
1  std::vector< TimeWindowSMTWAP * > windows;
```

TimeWindowSMTWAP

This class implements the SMTWAP definition of a time window, i.e. $\delta[s,\ e]$, as defined in Section 2.3, where:

- δ corresponds to the attribute *day* and it indicates the period of the time horizon when the time window is allocated;
- s and e are the *start_ time* and *end_ time*, i.e. the lower and upper bounds when the time window is opened and closed.

Code 2.13: Attributes of TimeWindowSMTWAP

```
1  int id;
2  int day;
3  int start_time;
4  int end_time;
5  double is_used;
6  bool is_changed;
```

Other useful attributes are:

- *id*, the unique identifier for the time window, whose value is from the set $\{0, \ldots, u\}$;
- the value *is_ used* indicates how much the time window is exploited in the different scenarios (see Subsection 2.5.4 to understand how it is computed);
- the boolean *is_ changed* is to understand if the time window has been modified or not after the first stage (see Subsection 2.5.4 for details about its use).

ScheduleSMTWAP

It represents the solution of the SMTWAP; it has just two attributes:

- the list of zones in the grid *zone_list*;
- the set of solved scenarios, with inside their corresponding instances of the class *SolutionVRPTW*, their costs and the average value, i.e. the value of the SMTWAP objective function.

Code 2.14: Attributes of ScheduleSMTWAP

```
1  ZoneList zone_list;
2  ScenarioSet set;
```

2.5.2 Loading phase

Hereafter follows the description of the main parts of the first stage: the loading of the zones from an input file, the computation and assignment of a set of time windows to each zone and the management of the whole list of zones.

Loading the zones

The first thing to do is knowing the configuration of the grid, i.e. the exact number h of the zones and how they are disposed in the geographical area.

All useful information of the grid are stored in a file given as input to the program.

At the beginning, a list of zones with size h is built and then every element is fulfilled with its own historical information, using the method *LoadZoneHistoric*, that receives the list of zones and the grid file as parameters.

Code 2.15: Pseudocode of LoadZonesHistoric

```
1  void LoadSMTWAP::LoadZonesHistoric(ZoneList &list, const char * filename)
2  {
3    Starting reading filename;
4    count_zones = 0;
5
6    while there exists a line to read in filename
7    {
8      Take the zone z = list[count_zones];
9      Read the following parameters:
10       origin_id, historic_cust, historic_dem, historic_service;
11
12     Save these information in the zone z;
13
14     count_zones++;
15   }
16 }
```

Loading the set of scenarios

Right after the list of zones has been loaded, the method *LoadScenarios* is called, passing as parameters the empty set of scenarios and another input file, which contains the scenarios information. The pseudocode of the method is shown below, where:

- lines [6-13] are dedicated to load generic data common to all scenarios, such as the number of scenarios, the vehicles capacity, depot opening times and coordinates, and the number of zones in the grid;
- for each scenario, all data about the set of customers to serve during the week and their orders are handled in lines [19-25]; instead the creation and initialization of drivers are in lines [26-34].

Code 2.16: Pseudocode of LoadScenarios

```
1  void LoadSMTWAP::LoadScenarios(ScenarioSet & set, const char * filecust)
2  {
3    Starting reading filecust
4    while there exists a line to read in filecust
5    {
6      if line contains "# SCENARIOS"
7        Allocate space to the set of scenarios;
8      else if line contains "VEHICLE"
9        Save capacity of vehicles;
10     else if line contains "DEPOT"
11       Save depot coordinates, opening time, closing time;
12     else if line contains "# ZONES"
13       Save number of zones;
14     else if line contains "SCENARIO #"
15     {
16       Start loading the scenario;
17       Associate the scenario to a problem to solve;
18       Save scenario probability;
19       for each customer:
20       {
21         Create a node;
22         Read the following information:
23         {origin_id, zone_cust, x_cust, y_cust, demand_cust,
               service_cust};
24         Save these information in the node just created;
25       }
26       Create just the sufficient number of drivers
27         in the first day of the time horizon;
28       for each driver:
29       {
30         Create the nodes (start_depot, end_depot);
31         Save these information:
32           {id, capacity, day, start_depot, end_depot}
33         Set sum_demand = 0;
34       }
35       Add customers to the problem to solve;
36       Add drivers to the problem to solve;
```

```
37        Initialize the distances matrix of the problem;
38        Initialize the times matrix of the problem;
39      }
40    }
41  }
```

2.5.3 Initial solution

When the list of zones and the set of scenarios have been loaded, an instance of the class *FirstStage* is created to build an initial solution for the SMTWAP and start the ZonesVNS, core of first stage.

Code 2.17: Constructor of FirstStage

```
1   FirstStage(ZoneList &zones, ScenarioSet & scen):
2     zone_list(zones),
3     set(scen),
4     best_sched( CreateInitialSolution() ) {}
```

To create a new *FirstStage* object, both the list of zones and the set of scenarios must be passed to the constructor, which also calls the *CreateInitialSolution* method.

Code 2.18: Pseudocode of CreateInitialSolution

```
1   ScheduleSMTWAP FirstStage::CreateInitialSolution()
2   {
3     InitializeAverageScenario;
4     BuildInitialSolution;
5     Solve the set of scenarios;
6     return ScheduleSMTWAP( zone_list, set );
7   }
```

InitializeAverageScenario computes the average number of customers in the set of scenarios and calls the method *SetHistoricVisitingNumber*.

Code 2.19: Pseudocode of SetHistoricVisitingNumber

```
1   void FirstStage::SetHistoricVisitingNumber()
2   {
3     historical_minimum_tw = new double(nb_zones);
4     historical_total_times = new double(nb_zones);
5     historic_total_customers = 0;
6
7     for each zone z in zone_list:
8     {
9       historical_total_times[z] = 0;
10      historic_total_customers += z->GetHistoricCustomersCount();
11    }
12
13    for each zone z in zone_list:
14    {
15      historical_total_times[z] +=
16        ( ( average_customers * z->GetHistoricCustomersCount() ) /
```

```
17        historic_total_customers ) * z->GetHistoricServiceTime();
18
19      historical_minimum_tw[z] =
20        historical_total_times[z] / STANDARD_TW_DURATION;
21    }
22  }
```

The purpose of *SetHistoricVisitingNumber* is to compute the minimum number of times every zone should be visited, according to historical information:

- in lines [3-11] two double arrays are initialized with the number of zones in the grid as size; the integer value *historic_ total_ customers* contains the sum of all historical number of customers in the several zones;
- for each zone z, instead, in lines [15-20] the minimum number of times of visiting corresponds to the estimated time needed to serve customers of z, divided by the standard duration of a time window in minutes; this value is then rounded to the higher closer integer to find the number of time window to assign to zone z.

After these computations, the method *BuildInitialSolution* is invoked where, for every zone z, enough space is allocated for all time windows it needs (line 8). Then random data about the day and opening times are assigned to every time window, uniquely identified by the number *id_ tw* that is incremented at each iteration; its last value is stored in the *zone_ list*, just in case new time windows will be added in the future.

Code 2.20: Pseudocode of BuildInitialSolution

```
1   void FirstStage::BuildInitialSolution()
2   {
3     id_tw = 0;
4     for each zone z in zone_list
5     {
6       for tw = 1 to historical_minimum_tw[z]
7       {
8         Reserve space to tw;
9         Assign id_tw to tw;
10        Set tw->is_changed = false;
11        Set tw->is_used = 0;
12        Assign a day not already used to tw;
13        Extract a random value for tw->start_time;
14        Set tw->end_time = tw->start_time + STANDARD_TW_DURATION;
15        id_tw++;
16      }
17      Set the last id_tw used in the zone_list;
18    }
19    Assign zone_list to the set of scenarios;
20  }
```

In the end, at line 19, the list of zones, completed with their set of time windows, is assigned to the *set* of scenarios and finally it can be solved, starting the second stage for the first time (see Subsection 2.5.5). The resulting instance of *ScheduleSMTWAP* is then stored and used to launch the ZonesVNS.

2.5.4 First stage: management of the list of zones

The list of zones and their time windows are handled by the class *ZoneList*, which contains the implementation of the moves used in the ZonesVNS.

It has to be observed that when a move is performed on one or several time windows, their boolean attribute *is_changed* is set to *true*; instead the method *ResetStatus* sets them to *false*. This trick permits to easy individuate which time windows have been modified.

Beyond moves, also some statistics have been implemented, useful to compute some indicators after the second stage is solved. Within this group, the following methods can be found:

- *GetWorstTW* - Its purpose is selecting the worst time windows in the whole zone list, evaluating the attribute *is_used*, i.e. returning the zone with the lowest value;
- *GetMostUsedTW* - Otherwise, in this case the most used time windows is returned, i.e. the one with the highest *is_used* value;
- *GetWorstZone* - This method instead returns the worst zone, according to the value of *score*.

The way *is_used* and *score* attributes are computed depends on the results of the second stage.

For every time window the value *used_tw* is computed as follows, with the method *ComputeUsedTW* in the *ScenarioSet* class:

Code 2.21: Pseudocode of ComputeUsedTW

```
1  void ScenarioSet::ComputeUsedTW()
2  {
3    for each scenario s in scenarios_set
4      for each zone z in zone_list
5        for each time window w
6        {
7          used_tw = 0;
8          count = 0;
9
10         for each driver that uses w in its route
11           count++;
12
13         used_tw += count * probability of s;
14       }
15 }
```

For each scenario ω, *count* indicates the number of times a time window is *used* by drivers, i.e. how many drivers utilize it to serve customers in the corresponding zone (see lines 10 and 11). Since the same time windows and zones appears in all problems and every scenario has its own probability p_ω to happen, the attribute *used_tw* is computed as the weighted sum of the several *count* at line 13.

It represents an indicator of how time windows are exploited.

The value *used_tw* of a time window w can be:

$$\begin{cases} 0 < used\_tw < 1, & \text{if } w \text{ is not used in every scenario;} \\ used\_tw = 1, & \text{if } w \text{ is used exactly once in every scenario;} \\ used\_tw > 1, & \text{if } w \text{ is used once or more in every scenario.} \end{cases}$$

If it is less than 1, then it can mean that the time window ω might be moved to another day or to a different slot, or even removed because not necessary.

These *used_tw* values are needed to calculate the *score* of a zone, i.e. an index of how its set of time windows is good; it corresponds to the sum of *used_tw* of every time window in the set assigned to the zone z, divided by the cardinality of the set.

Code 2.22: Pseudocode of ComputeScoreZones

```
1  void ScenarioSet::ComputeScoreZones()
2  {
3    for each zone z in zone_list
4    {
5      Set score to 0;
6
7      for each time window w assigned to z
8        score += used_tw of w;
9
10     score = score / nb of time windows of z;
11   }
12 }
```

Every time the second stage is performed, these statistics are reset to zero, in order to be computed again.

ZonesVNS procedure

The main function of the procedure is called *Optimize*, and it is composed of two parts: a *Setup* phase and the *Neighborhood Search*.

First, in the *Setup* phase, the ZonesVNS applies to the list of zones some moves for a certain number of iterations, performing a simple local search and saving the best schedule found; then, the core part of the ZonesVNS starts.

Optimizing the schedule The function *Optimize* works on the initial solution build at the beginning of *FirstStage*.

Two versions were implemented: *OptimizeRandom*, where zones and time windows are always chosen randomly, and *OptimizeOnZones*, that instead selects them according to their behavior in the current solution, which is defined by the attributes *is_used* for time windows and *score* for zones.

Hereafter the pseudocode of the latter is described: since the *Setup* method is almost the same for both versions, only one implementation has been made, distinguishing the two types with a char: '*r*' for the random one and '*z*' for the other, as can be seen at line 4. The *MoveToNeighborhood* and *Shaking* methods, respectively at lines 21 and 24, are the same for both versions, while *LocalSearchOnZones* is proper to *OptimizeOnZones*, at line 27.

Code 2.23: Pseudocode of OptimizeOnZones

```
1  void ZonesVNS::OptimizeOnZones()
2  {
3    // 1. SETUP PHASE
4    ScheduleSTMWAP best = Setup('z');
5
6    // 2. NEIGHBORHOOD SEARCH
7    iter = 0;
8    nk = 0;
9    start_time = clock();
10   time = 0;
11
12   while(iter <= VNS_MAX_ITER)
13   {
14     ScheduleSMTWAP current = best;
15     nk = 1;
16     int VNS_MAX_NB_EXCHANGES = nb_zones/2;
17     while(k <= VNS_MAX_NB_EXCHANGES && iter <= VNS_MAX_ITER
18           && time <= VNS_MAX_TIME)
19     {
20         // 2a) EXCHANGE TW BETWEEN ZONES nk times
21         ScheduleSMTWAP x = MoveToNeighborhood(nk, current);
22
23         // 2b) SHAKING
24         ScheduleSMTWAP x_first = Shaking(x);
25
26         // 2c) LOCAL SEARCH
27         ScheduleSMTWAP x_second = LocalSearchOnZones(x_first);
28
29         // 2d) MOVE OR NOT
30         if(x_second.GetCost2Stage() < best.GetCost2Stage())
31         {
32           best = x_second; // Better solution found
33           nk = 1;
34         }
35         else
36           nk++; // No improvement;
37
38         iter++;
39         time = clock() - start_time;
40     }
41     iter++;
42   }
43 }
```

Integers *iter* and *nk* are used respectively to count the total number of iterations done and to know how many swaps have to be performed to move to the next neighborhood from the current solution.

The maximum number of exchanges *VNS_MAX_NB_EXCHANGES* is defined in line 14.

At lines [30-36] the *MoveOrNot* comparison is done only between *x_second* and the best schedule found, because *x_second* is already the improved schedule

of *x_first* or, if no better solution was found in *LocalSearchOnZones*, *x_second* corresponds exactly to *x_first*. If the cost of second stage of the *best* schedule is higher than the cost of *x_second*, then the list of zones and the set of scenarios of *x_second* are saved into *best*, becoming the new best solution found; then *nk* is reset to 1. Otherwise, if the best cost is still lower, then *nk* is increased.

As can be observed at lines 17 and 18, the whole procedure stops when one or more of the following stopping criteria are satisfied: the maximum number of iterations is reached, or the maximum number of exchanges or the maximum computational time elapsed.

Setup Phase As already described in Subsection 2.4.1, three different moves can be applied randomly extracting a number, as can be seen in lines [12-34]:

- if *setup_move* is 0, then a time window is moved to a different day (see lines [15-19];
- instead, if it is 1, a time window is relocated to another slot, without changing its day (see lines [21-25];
- otherwise, if *setup_move* is 2, there is the expansion of a time window (see lines [27-31]).

As said before, according to the type of the *Optimize* method, moves can work on random zones and time windows or on the *worst* ones.

Code 2.24: Pseudocode of Setup

```
1  void ZonesVNS::Setup(char vns_type)
2  {
3    current = best;
4    for VNS_SETUP_NB_ITER
5    {
6      From current:
7        Get the current_list of zones;
8        Get the current_set of scenarios;
9
10     ResetStatus of current_list;
11
12     setup_move = random number in [0, ..., VNS_SETUP_NB_TYPE_MOVES];
13     switch(setup_move)
14     {
15       case 0: if(vns_type == 'r')
16             Apply Move Random Time Window to current_list;
17           else
18             Apply Move Worst Time Window to current_list;
19           break;
20
21       case 1: if(vns_type == 'r')
22             Apply Move Random Slot to current_list;
23           else
24             Apply Move Worst Slot to current_list;
25           break;
26
```

```
27          case 2: if(vns_type == 'r')
28                Apply Expand Random Time Window to current_list;
29              else
30                Apply Expand Selected Time Window to current_list;
31              break;
32
33          default: break;
34        }
35
36        Set current_list to current_set;
37        ApplyChanges to current_set;
38        Optimize current_set;
39
40        if(current.cost < best.cost)
41          best = current; // Better solution found
42        else
43          current = best; // No improvement
44      }
45    }
```

At line 37 the method *ApplyChanges* is invoked: for each scenario in *current_set*, it examines which drivers or routes are *affected* by the changes done on time windows in *current_list*. A driver is considered affected if, within its route, it travels through one or more zones influenced by the moves just applied to time windows; since the time windows have been modified, probably its route has become unfeasible. Therefore, the method removes all the customers assigned to the driver and marks them as *unassigneds*.

Then, at line 38, *current_set* has to be optimized again, providing a new current schedule to compare with the best one (see lines [40-43]).

Move to neighborhood The move chosen to make the ZonesVNS visit the sequence of neighborhoods N_k is *Exchange Time Windows* which, when $k = 1$, operates one swap between two random time windows of two zones; generally, considering the k-neighborhood of the current solution, it performs k swaps among k couples of zones.

The method receives as parameters the number of exchanges to do, that corresponds to a particular neighborhood of the schedule x, also passed in input.

The return value is another schedule *cur*, element of the neighborhood $N_{nb\_exchanges}$, that is not so far from x: a percentage is computed about how distant *cur* is from x, and it is stored in the variable *gap*; if this is not less than the threshold *VNS_NEIGHBORHOOD_GAP*, the procedure is repeated.

This is done to obtain a different solution from x but still good, not too much more expensive.

Code 2.25: Pseudocode of MoveToNeighborhood

```
1   ScheduleSMTWAP ZonesVNS::MoveToNeighborhood(int nb_exchanges,
        ScheduleSMTWAP & x)
2   {
3     ScheduleSMTWAP cur = x;
```

```
4     gap = 0;
5
6     do{
7       cur = x;
8       From cur:
9         Get the current_list of zones;
10        Get the current_set of scenarios;
11      zone_changed = [ ];
12      ResetStatus of current_list;
13
14      for i = 1 to nb_exchanges
15      {
16        Apply Exchange Time Windows to current_list,
17          without touching the zones
18          whose identifiers are in zone_changed;
19
20        Insert in zone_changed the identifiers of the zones affected;
21      }
22      Set current_list to current_set;
23      ApplyChanges to current_set;
24      Optimize current_set;
25      gap = (current.cost - x.cost)/x.cost;
26
27    } while(gap > VNS_NEIGHBORHOOD_GAP);
28
29    return cur;
30  }
```

Exchange Time Windows acts just on zones which it has not already affected: swaps can happen only between two zones not exchanged earlier during the same execution of the method. To do that, identifiers of touched zones are stored in the list *zone_ changed* (line 20) and not used in successive swaps (see lines [16-18]).

Shaking The approach used is similar to the one implemented in *Move-ToNeighborhood*: some moves are applied until a still good solution is found, otherwise the operation is repeated.

Code 2.26: Pseudocode of Shaking

```
1   ScheduleSMTWAP ZonesVNS::Shaking(ScheduleSMTWAP & x)
2   {
3     ScheduleSMTWAP cur = x;
4     gap = 0;
5     do{
6       cur = x;
7
8       From cur:
9         Get the current_list of zones;
10        Get the current_set of scenarios;
11      ResetStatus of current_list;
12
```

```
13        Apply Move K Random Slots to current_list,
14          where K = VNS_SHAKING_NB_MOVES;
15        Set current_list to current_set;
16        ApplyChanges to current_set;
17        Optimize current_set;
18        gap = (current.cost - x.cost)/x.cost;
19
20      } while(gap > VNS_SHAKING_GAP);
21      return cur;
22    }
23  }
```

VNS_SHAKING_GAP is used, at line 20, to define the value of the threshold, as done with *VNS_NEIGHBORHOOD_GAP* in *MoveToNeighborhood*, to evaluate the quality of the new schedule computed.

Local search Three different simple local searches are performed, all with first improvement heuristics; the return value is the best schedule obtained among the three resulting ones.

Code 2.27: Pseudocode of LocalSearchOnZones

```
1   ScheduleSMTWAP ZonesVNS::LocalSearchOnZones(ScheduleSMTWAP & x_first)
2   {
3     ScheduleSTMWAP expand =
4       FirstImprovementOnZones(x_first, 0);
5
6     ScheduleSMTWAP exchange_timings =
7       FirstImprovementOnZones(x_first, 1);
8
9     ScheduleSMTWAP move_worst =
10      FirstImprovementOnZones(x_first, 2);
11
12    return the best among expand, exchange_timings, move_worst
13  }
```

To distinguish the kind of moves to apply in the method *FirstImprovement*, integers from 0 to 2 are used where:

- 0 corresponds to execute a local search with *Expand Selected Time Window* in case of *OptimizeOnZones*, or *Expand Random Time Window* when *OptimizeRandom* is called;
- 1 stands for *Exchange Time Windows Slots of Selected Zones* or *Exchange Time Windows Slots*;
- 2 is for *Move Worst Time Window* or *Move Random Time Window*.

Here follows the pseudocode of *FirstImprovementOnZones*.

Code 2.28: Pseudocode of FirstImprovementOnZones

```
ScheduleSMTWAP ZonesVNS::FirstImprovementOnZones(ScheduleSMTWAP &
    x_first, int move_type)
{
  ScheduleSMTWAP current = x_first;
  iter = 0;
  time = 0;

  while( iter <= VNS_FIRST_MAX_ITER && time <= VNS_FIRST_MAX_TIME)
  {
    From current:
      Get the current_list of zones;
      Get the current_set of scenarios;

    ResetStatus of current_list;

    switch(move_type)
    {
      case 0: Apply Expand Selected Time Window on current_list;
              break;

      case 1: Apply Exchange Time Windows Slots Of Selected Zones on
          current_list;
              break;

      case 2: Apply Move Worst Time Window on current_list;
              break;

      default: break;
    }

    Set current_list to current_set;
    ApplyChanges to current_set;
    Optimize current_set;

    if(current.cost < x_first.cost)
      return current; // First improvement found
    else
      current = x_first; // No improvement

    iter++;
    time = clock();
  }

  return current;
}
```

2.5.5 Second stage: solving scenarios

In the current Subsection, details are provided about the initialization of scenarios after loading, how the related VRPTWs are solved and the way the cost of a schedule is computed.

During the creation of the initial schedule solution in the first stage, after assigning a set of time windows to every zone with the procedure *BuildInitialSolution*, the method *Solve* of the class *ScenarioSet* is invoked on the set of scenarios.

Code 2.29: Pseudocode of Solve

```
1  void ScenarioSet::Solve()
2  {
3    InitializeInternalFields;
4    Optimize;
5  }
```

Solve calls first the method *InitializeInternalFields* and then *Optimize*, both always defined in the class *ScenarioSet*.

InitializeInternalFields The class *ScenarioSet* has an attribute, among others, a vector of pointers to some instances of *Optimizer*.

An optimizer is in charge of solving the VRPTW of the scenario which is linked to, constructing and working on a solution. It is the object which initializes and defines ALNS operators; then it exploits ALNS to find good routes to serve all customers in the problem.

Code 2.30: Pseudocode of InitializeInternalFields

```
1  void ScenarioSet::InitializeInternalFields()
2  {
3    optimizers = [ ];
4    for each scenario s in scenarios_set
5    {
6      Create a new Optimizer opt;
7      Insert opt in optimizers;
8      Initialize opt;
9    }
10 }
```

At line 8, initializing the optimizer means calling the method *Initialize* of the class *Optimizer*: at the beginning, none of the customers is served by any driver and they are all marked as *unassigned*.

The pseudocode of *Initialize* is reported in the following page.

Before launching ALNS, at line 3 the *RegretInsertion* heuristic is used to initially assign customers to drivers' routes, as described in Subsection 2.5.5, talking about sub-heuristics of ALNS. *RegretInsertion* is also used at line 16 during the while loop, after adding a new driver, needed because not all customers could not be assigned to a driver during ALNS execution at line 10.

Code 2.31: Pseudocode of Initialize

```
1  void Optimizer::Initialize()
2  {
3    Use RegretInsertion on the solution sol;
4    Set the initial temperature of ALNS;
5    Set the maximum number of iteration of ALNS;
6    Set the minimum temperature of ALNS;
7
8    while(# unassigned customers in sol != 0)
9    {
10     Use ALNS to optimize sol;
11     if(# unassigned customers in sol == 0)
12       break;
13     Add a new driver to sol,
14       in a random day or when it is more needed;
15     Update sol;
16     Use RegretInsertion on sol;
17   }
18 }
```

At the end of *InitializeInternalFields*, every scenario is connected to an optimizer with an initial solution and enough vehicles to serve all requests, all assigned to one of them; routes have just to be optimized.

Optimize In lines [3-7] of the pseudocode in the following page, all statistics about the use of time windows and scores of zones are set to zero, before optimizing the current solutions of scenarios in lines [10-14].

Code 2.32: Pseudocode of Optimize

```
1  void ScenarioSet::Optimize()
2  {
3    for each zone z in zone_list
4    {
5      Reset used_tw to 0;
6      Reset score to 0;
7    }
8
9    results = [ ];
10   for each scenario s in scenarios_set
11   {
12     Optimize s;
13     Insert the cost of s into results;
14   }
15
16   for each zone z in zone_list
17   {
18     for each time windows tw assigned to z
19       Compute used_tw of tw;
20
21     Compute the score of z;
22   }
23 }
```

Optimization rearranges customers' assignment to drivers during the different days of the time horizon, in order to get a lower total cost for all routes needed to the service.

Get the cost of second stage This value can be retrieved with the method *GetWeightedAverageResults* of the *ScenarioSet* class, whose pseudocode is reported below.

Code 2.33: Pseudocode of GetWeightedAverageResults

```
1  double ScenarioSet::GetWeightedAverageResults()
2  {
3    average = 0.0;
4    for each scenario s in scenarios_set
5      average += cost of s * probability of s;
6
7    return average;
8  }
```

Other statistics After finding better solutions, at the end of second stage, for every time window the value *used_tw* is computed with the method *ComputeUsedTW* and the *score* of every zone is consequently calculated, calling the method *ComputeScore*; both methods are reported in the previous Subsection.

Chapter 3

Computational Results

"Time past and time future
What might have been and what has been
Point to one end, which is always present."
T. S. Eliot, Four Quartets, Burnt Norton, I

This chapter is devoted to the description of the testing environment including the instances generation, the choice of parameters done for the ZonesVNS and ALNS and the tests performed. Computational results are then reported, showing that the adopted solution approach can effectively improve the starting solution.

3.1 Generation of instances

When solving the VRP and its variants, researchers make use of well-known benchmark instances (see [40]) to test the validity of the developed methods. Among these, the most famous instances for the VRP with Time Windows are the ones designed in 1983 by Solomon with 100 customers, and then extended to include a larger number of requests to serve. In the literature other sets of instances for the VRPTW can be found, also different from the Solomon's ones. Nevertheless, none of them can be used for the SMTWAP, because of many reasons:

- customers are never assigned to zones but only their coordinates are given to identify their locations;
- in every set of instances, zones are never considered and therefore their information is always missing;
- as for the zones, no scenarios are defined;
- in the PVRP instances designed by Cordeau, every customer specifies days when they can be served, but this does not respect the definition of the SMTWAP, where customers do not indicate their preferred service days.

It has been consequently decided to build a new set of instances, to consider zones information, scenarios and their probabilities. For each instance, two files are provided that represent the grid of zone and the related set of scenarios. They are illustrated in Subsections 3.1.1 and 3.1.2, respectively.

3.1.1 An instance of a grid of zones

Suppose there is an historic database of all customers' past orders, specifying in each row the identity of the client, which zip code or zone it belongs to, the date the order was made, the requested demand, the occurred service time and the relative cost payed. From this table, a sort of map or *grid* of the different zones can be built, deriving statistics information. For each zip code z, the following data are computable:

- the average number of customers $\overline{n_z}$;
- the average demand per customer $\overline{d_z}$;
- the average service time per customer $\overline{s_z}$.

Since no real data from companies were available, some instances of possible grids of zones have been randomly constructed, using a program specifically developed in *Ruby*, called *zonegrid_maker.rb* . The default grid contains 4 zones, but the scripts allows also to specify in input the number of rows r and columns c wanted in the grid, to obtain a list of $h = r \cdot c$ zones.

It has been considered that zones are differently populated; in particular, the population density can be *high*, *normal* or *low*. In an instance there must be a certain percentage of highly and lowly populated zones. The average number of customers in a zone is obtained from a uniform distribution; hereafter the different ranges used are reported, related to the medium number of customers wanted per zone.

Table 3.1: Ranges of number of customers per zone

Average Nb Customers	**Low Range**	**Normal Range**	**High Range**
25	[1 - 10]	[10 - 20]	[20 - 25]
50	[1 - 15]	[15 - 35]	[35 - 50]
100	[1 - 20]	[20 - 70]	[70 - 100]
150	[1 - 25]	[25 - 125]	[125 - 150]
250	[1 - 50]	[50 - 150]	[150 - 250]
500	[1 - 100]	[100 - 350]	[350 - 500]

Also the average demand and service time per customer in a zone are randomly extracted from a uniform distribution with given minimum and maximum values. The considered range for the average demand is [5, 50] (without a defined measure unity) whereas the one for the average service time is [5, 15] (in minutes).

The resulting file appears as follows, where each zone z is identified by the quadruple ($zipcode_z$, $\overline{n_z}$, $\overline{d_z}$, $\overline{s_z}$).

```
(  1, 100,  35,  12)    (  2,  39,  15,   9)    (  3,  43,  47,  15)
(  4,  18,  41,  10)    (  5,  13,  50,  13)    (  6,   3,  33,  11)
(  7,  17,  21,   8)    (  8,  10,  43,   9)    (  9,  10,  24,  11)
( 10,  62,  19,   8)    ( 11,  75,  20,  12)    ( 12,  50,  17,  10)
( 13,  34,  32,   5)    ( 14,  37,  42,  14)    ( 15,  81,  37,   8)
```

Figure 3.1: Example of a grid of zones

3.1.2 An instance of a set of scenarios

An instance of the grid of zones can be linked to several instances of sets of scenarios, because in the same area different cases can happen.

The script used to create an instance of a set of scenarios is *scenarios_ maker.rb*, still expressly written in *Ruby*; every output file contains these data:

- the number of scenarios;
- the vehicles capacity Q;
- the depot coordinates (x_0, y_0);
- the depot opening and closing times;
- the number of zones h;
- for each scenario, its probability to happen, the number of customers, their coordinates, demands and service times.

The depot is localized in a random zone, such that it is enough far from a zone with a low number of customers or demands.

Every zone is represented as a rectangle, with height and width equal to 700 and 1000 units, respectively. Using the meter as measure unit, then a grid composed of 5 rows and 3 columns, corresponding to 15 zones, is extended over an area of about 10 km^2 and can be used to model some quarters of a big city or a small town. Smaller grids can outline a district, whereas larger ones may be suitable for metropolis.

The number of customers of each scenario is obtained through a normal distribution, given the desired average number and the standard deviation.

Table 3.2: Normal distributions for the number of customers

Average Nb Customers	**Standard Deviation σ**
25	5
50	10
100	15
150	20
250	25
500	30

According to the statistic information about the grid, customers are distributed into different zones and the list of orders is derived, still using normal distributions also for demands and service times.

An instance of the set of scenarios is represented as follows:

```
# SCENARIOS = 4
VEHICLE CAPACITY = 500
DEPOT COORDINATES (X,Y) = (4361, 407)   TIME [8-20]
# ZONES = 10

SCENARIO #1     # CLIENTS = 18  PROBABILITY = 0.141
CLIENT  ZONE    X       Y       DEMAND  S_TIME
  1      10      4470    1098    35       7
  2      10      4566    1036    10       5
  3      10      4279     786     8       5
  4       7      1977    1295     7      12
  5       7      1349    1239     1      12
  6       7      1285    1213    37      13
  7       8      2855    1248    42      15
  8       6       693    1394    33       5
  9       4      3233     635    47      11
 10       7      1043    1306     3      12
 11      10      4402    1391    11       5
 12       4      3192     251    18      11
 13       4      3512     190    34       7
 14       4      3320     335     1      11
 15      10      4607     921    58       5
 16       7      1606    1306    21      15
 17       4      3903     473    42       8
 18       7      1985    1368    46      15


SCENARIO #2     # CLIENTS = 22  PROBABILITY = 0.314
CLIENT  ZONE    X       Y       DEMAND  S_TIME
  1      10      4586     764    15       7
  2       7      1267    1327     7      15
  3       7      1173     972     1      15
  4       6        39    1389    62       6
  5       4      3363     635    11       9
  6      10      4647    1042     1       5
  7       8      2394     840    35      14
  8       7      1965    1273     6      13
  9       7      1057     954     3      14
 10       7      1854     748    23      14
 11      10      4103    1007    38       5
 12       8      2044    1298    34      15
```

Figure 3.2: Example of a set of scenarios

Three instances of grids of zones have been generated, composed of 10, 15 and 20 zones respectively; then sets of scenarios have been produced, linked to grids, varying the number of customers among 25, 50, 100, 150, 250 and 500. For each couple (*number of zones*, *number of customers*), three instances have been created, obtaining in the end 54 instances with 4 scenarios each.

The instance name follows the pattern *zones_ customers_ scenarios_ setnumber* (e.g., *10_ 25_ 4_ 1* indicates that there are 10 zones in the grid, with 25 customers to serve, and 4 scenarios to solve).

3.1.3 The solution files

At the end of the execution of the ZonesVNS algorithm, two output files are produced to provide the solution of the instance given in input.

The first file is textual and is organized into two parts:

1. the first one describes the best schedule found, reporting for each zone its set of time windows;

2. the second one contains the cost of the second stage, corresponding to the weighted average of the costs of single scenarios. Beyond the objective function value, for each scenario there are some data about drivers, organized in the several periods of the time horizon; data include their costs, the lists of customers served and their timetable.

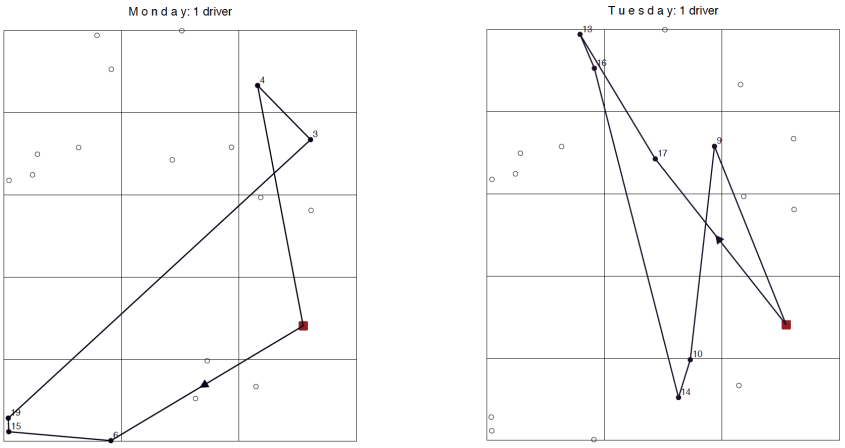
There is also the information about computational times required to obtain the initial and the final solutions. Fig. 3.3 presents a part of the solution for the instance *15_25_4_2*.

```
Total Elapsed Time: 26.41 s
Elapsed Initial solution:  0.68 s
Elapsed ZonesVNS: 25.74 s

ZONE LIST:
ZONE    TIME WINDOWS
 1( 0): { TW 7 - 0[15 - 17] }
 2( 1): { TW 4 - 1[ 9 - 12] }
 3( 2): { TW 2 - 3[12 - 14] }
 4( 3): { TW 14 - 0[17 - 19] }
 5( 4): { TW 1 - 2[12 - 14] }
 6( 5): { TW 6 - 1[17 - 19] }
 7( 6): { TW 13 - 2[16 - 19] }
 8( 7): { TW 8 - 2[10 - 12] }
 9( 8): { TW 9 - 3[13 - 15] }
10( 9): { TW 0 - 2[16 - 18] }
11(10): { TW 10 - 1[11 - 15] }
12(11): { TW 5 - 0[14 - 16] }
13(12): { TW 12 - 1[11 - 13] }
14(13): { TW 11 - 4[12 - 14] }
15(14): { TW 3 - 0[16 - 18] }


SCENARIOS SET:
RESULTS - WEIGHTED AVERAGE = 82.83
Scenario #1:    Cost = 116.07;  Not served = 0  Num routes (empty) = 7(2)       Probability = 0.156
Scenario #2:    Cost =  79.92;  Not served = 0  Num routes (empty) = 8(3)       Probability = 0.296
Scenario #3:    Cost =  71.67;  Not served = 0  Num routes (empty) = 7(1)       Probability = 0.233
Scenario #4:    Cost =  77.35;  Not served = 0  Num routes (empty) = 9(2)       Probability = 0.315


# SCENARIOS = 4
DEPOT OPENING TIME: [ 9-20]

SCENARIO #1:     COST: 116.07   PROBABILITY: 0.156
#EMPTY ROUTES = 2
---------- DAY 1 (EMPTY = 0) ----------
DRIVER # 0      CURR.DEMAND:  206/ 500; COST: 22.98;    LENGTH: 5;      DURAT: 73;
Start(9:0) - 6[Z:1] (15:0) - 15[Z:1] (15:13) - 19[Z:1] (15:26) - 3[Z:12] (15:42) - 4[Z:15] (16:0) - End(16:10)
```

Figure 3.3: Example of a textual solution file

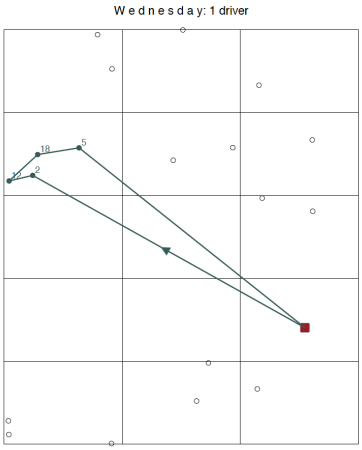
The second output file is a PDF and shows a graphic representation of the solution in the grid of zones, describing every period of the time horizon.

Fig. 3.4 illustrates the solution of the first scenario of the instance *15_25_4_2*, showing each day of the considered week; for each driver, its route, the direction of travel and customers visited are reported.

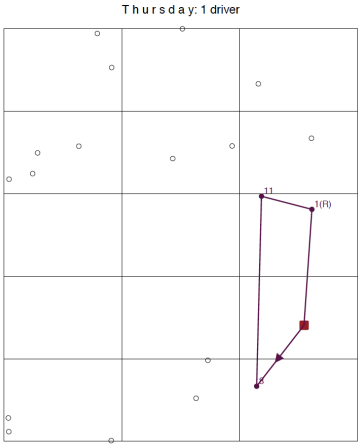


(a) **Monday**

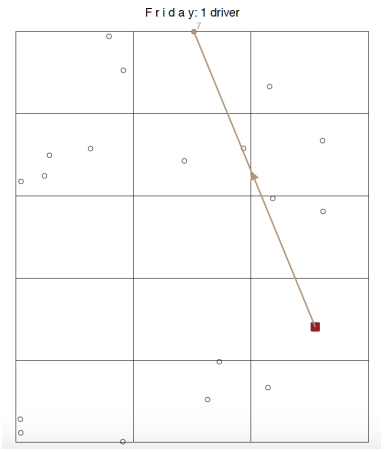
(b) **Tuesday**



(c) **Wednesday**



(d) **Thursday**



(e) **Friday**

Figure 3.4: Example of a graphic solution file

3.2 Computational Results

This section is dedicated to the analysis of computational results, discussing the solutions quality according to objective function values, computational times and other solution features.

3.2.1 Improvement on the objective function

In this subsection an analysis is performed on the objective function values of the final solutions, discussing the percentage improvements using the two ZonesVNS variants and average results.

Percentage improvements

The average percentage improvement obtained with the ZonesVNS is about 60%, as can be seen in Table 3.3.

Table 3.3: Percentage improvement on the objective function

Instances	% Opt	% Random	% Average
10_25_4	72,88%	85,89%	79,38%
10_50_4	79,59%	77,57%	78,58%
10_100_4	68,05%	67,71%	67,88%
10_150_4	65,78%	74,12%	69,95%
10_250_4	45,63%	45,54%	45,58%
10_500_4	42,52%	54,75%	48,64%
15_25_4	70,92%	81,00%	75,96%
15_50_4	75,58%	83,26%	79,42%
15_100_4	65,97%	64,54%	65,25%
15_150_4	57,88%	64,60%	61,24%
15_250_4	51,17%	52,19%	51,68%
15_500_4	46,15%	39,20%	42,67%
20_25_4	55,27%	69,78%	62,53%
20_50_4	59,18%	67,44%	63,31%
20_100_4	61,34%	55,17%	58,26%
20_150_4	53,09%	58,20%	55,64%
20_250_4	31,05%	46,29%	38,67%
20_500_4	27,08%	47,06%	37,07%
	57,17%	**63,02%**	**60,09%**

Also in Fig. 3.5 percentage improvements are shown, computed comparing the random initial solution generated at the beginning of the first stage and the best solution found by the ZonesVNS. The two ZonesVNS variants, *Optimize On Zones* and *Optimize Random*, work on the same instance but, since the first assignment of time windows to zones is purely random, they start from two different initial solutions.

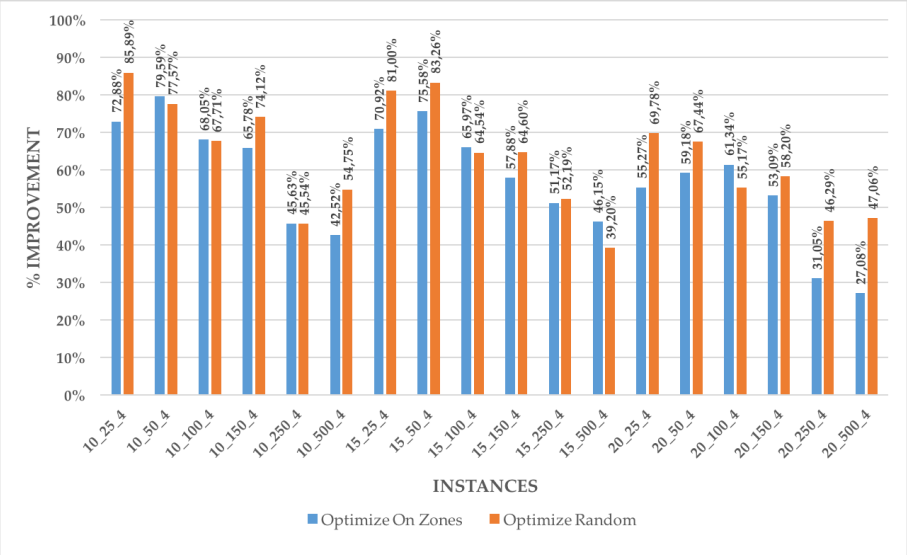


Figure 3.5: Percentage improvement on the objective function

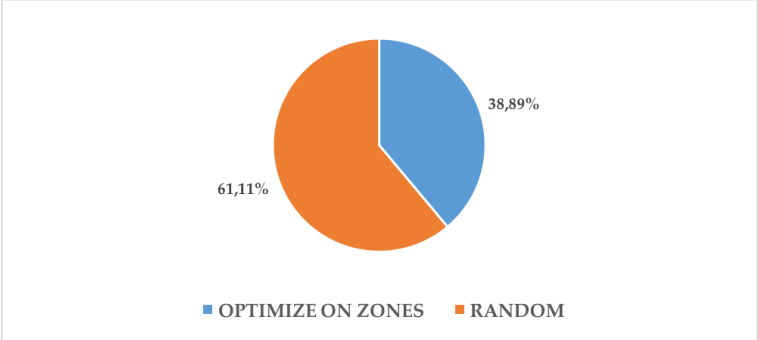


Figure 3.6: Percentage of better solutions found

Using *Optimize On Zones*, the best improvement occurs for the instance *15_100_4_2* (91,00%), whereas the worst one is the instance *15_500_4_3* (13,20%). *Optimize Random* has similar results: the best improvement is obtained with *10_50_4_3* (89,14%), while the worst one with *15_500_4_1* (25,62%).

Even if in about 60% of instances *Optimize On Zones* starts from better initial solutions than *Optimize Random*, the latter finds better solutions in the 61% of the cases (see Fig. 3.6). Since the number of instances is not low but

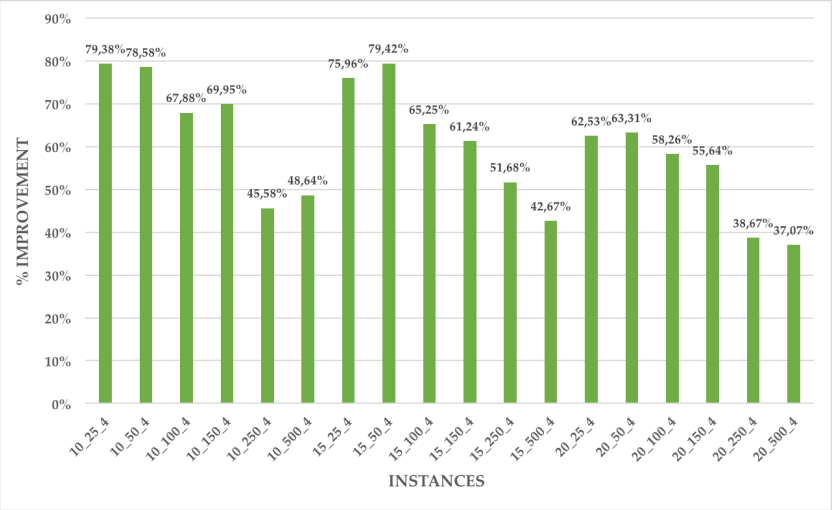


Figure 3.7: Average percentage improvement on the objective function

neither high, this behavior might be due to a lucky case during the optimization, but probably it indicates that the values used to select worst zones and time windows should be adapted and presumably enhanced.

Evaluating together the results of the two ZonesVNS variants, average improvements and values of objective functions have been calculated, which are illustrated in Fig. 3.7 and Fig. 3.8, respectively. The results are also reported in Table 3.4. Larger instances with hundreds of customers present the greatest values, as predictable, involving higher costs.

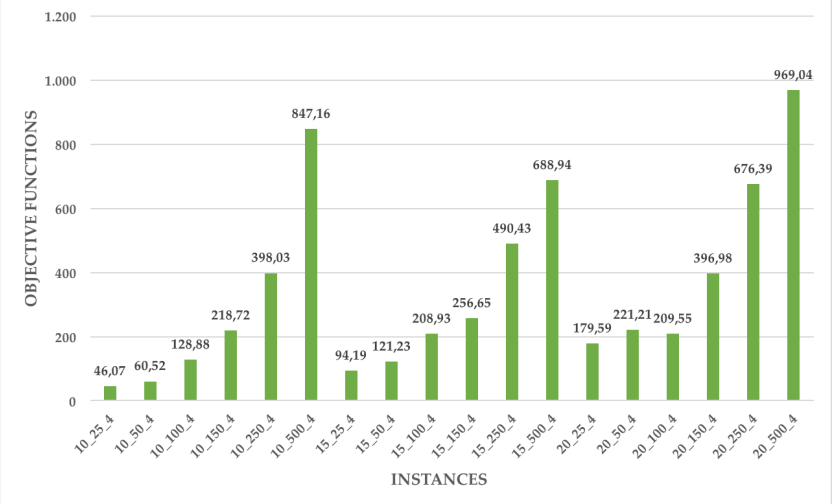


Figure 3.8: Average objective function values

Table 3.4: Average objective function values

Instances	**Opt (min)**	**Random (min)**	**Average (min)**
10_25_4	50,65	41,49	46,07
10_50_4	64,36	56,68	60,52
10_100_4	112,98	144,79	128,88
10_150_4	276,07	161,37	218,72
10_250_4	398,57	397,50	398,03
10_500_4	1010,52	683,80	847,16
15_25_4	126,62	61,76	94,19
15_50_4	130,95	111,51	121,23
15_100_4	223,26	194,60	208,93
15_150_4	281,92	231,37	256,65
15_250_4	397,07	583,79	490,43
15_500_4	587,62	790,25	688,94
20_25_4	206,66	152,51	179,59
20_50_4	245,06	197,35	221,21
20_100_4	183,76	235,34	209,55
20_150_4	405,26	388,69	396,98
20_250_4	783,11	569,66	676,39
20_500_4	1057,74	880,34	969,04

Percentage improvement related to the number of zones

Table 3.5: Percentage improvement related to the number of zones

# Zones	Average % Opt	Average % Random	Average %
10	62,41%	68,85%	65,63%
15	61,28%	66,96%	64,12%
20	47,83%	57,32%	52,58%
	57,17%	**64,38%**	**60,78%**

The number of zones can influence the ZonesVNS because a higher number of zones, and therefore time windows, requires more computational time (see Subsection 3.2.2) for reasoning, in the first stage, about which moves to apply and, in the second stage, about which requests insert or remove with the ALNS heuristics. This can explain why instances based on the grid with 20 zones are improved less than others (see Table 3.5 and Fig. 3.9).

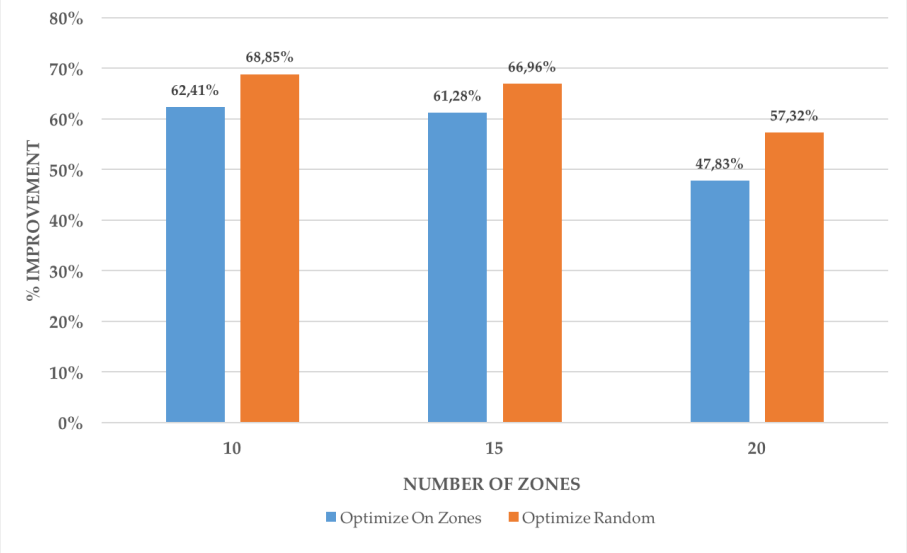


Figure 3.9: Percentage improvement related to the number of zones

Percentage improvement related to the number of customers

The impact on the improvement due to the number of customers is more intuitive to understand and results satisfy expectations about it.

Best improvements are obtained with an average of 25 or 50 customers, independently from the number of zones, decreasing the value of the initial solution by about 75%. The worst instances are the ones with 500 average customers, with an improvement of about 40%.

Results are shown in Table 3.6 and Fig. 3.10.

Table 3.6: Percentage improvement related to the number of customers

# Customers	Average % Opt	Average % Random	Average %.
25	66,36%	78,89%	72,62%
50	71,45%	76,09%	73,77%
100	65,12%	62,47%	63,80%
150	58,91%	65,64%	62,28%
250	42,62%	48,00%	45,31%
500	38,58%	47,01%	42,79%
	57,17%	**63,02%**	**60,09%**

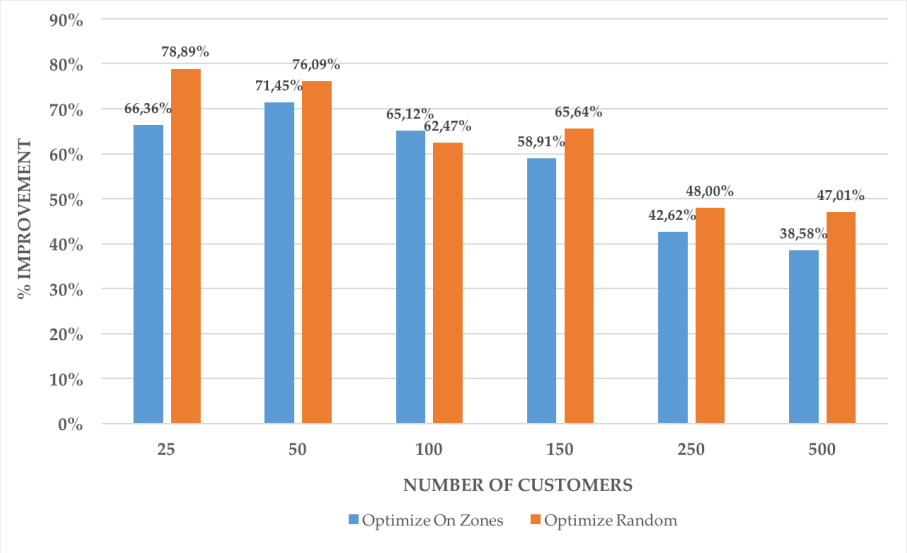


Figure 3.10: Percentage improvement related to the number of customers

3.2.2 Computational times

This subsection is focused instead on the evaluation of solutions according to the computational times required for executing the solving method.

The two main parts of the SMTWAP are the construction of the random initial solution and the following optimization; therefore they both are evaluated considering the time they take, beyond the total computational time.

Table 3.7, Table 3.8 and Table 3.9 show the time elapsed during the computation of the initial solution, during the optimization and the total time, respectively. Both *Optimize On Zones* and *Optimize Random* are considered (see Fig. 3.11) and also average results (see Fig. 3.12).

Table 3.7: Initial solution computational times

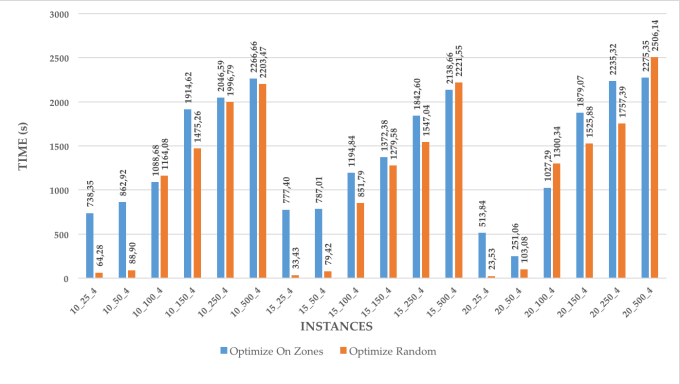
Instances	**Init. Opt (s)**	**Init. Random (s)**	**Average Init. (s)**
10_25_4	0,79	0,68	0,74
10_50_4	2,19	2,35	2,27
10_100_4	14,08	12,35	13,22
10_150_4	29,66	35,08	32,37
10_250_4	119,54	135,63	127,59
10_500_4	665,36	731,96	698,66
15_25_4	0,63	0,69	0,66
15_50_4	1,99	2,67	2,33
15_100_4	9,31	9,74	9,53
15_150_4	28,52	28,59	28,56
15_250_4	114,98	113,14	114,06
15_500_4	739,78	764,69	752,24
20_25_4	0,94	0,99	0,97
20_50_4	2,69	2,72	2,71
20_100_4	8,08	8,64	8,36
20_150_4	27,70	30,42	29,06
20_250_4	98,88	101,34	100,11
20_500_4	550,48	935,68	743,08

Table 3.8: ZonesVNS computational times

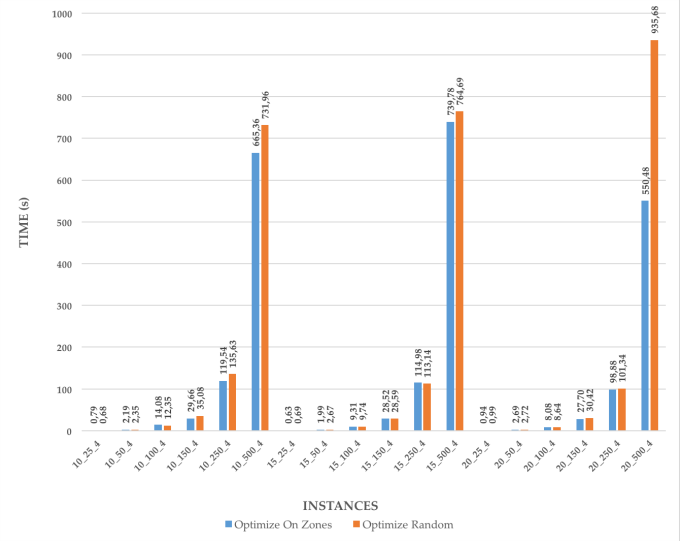
Instances	**VNS Opt (s)**	**VNS Random (s)**	**Average VNS (s)**
10_25_4	737,56	63,59	400,58
10_50_4	860,73	86,55	473,64
10_100_4	1074,60	1151,73	1113,17
10_150_4	1884,96	1440,18	1662,57
10_250_4	1927,05	1861,16	1894,11
10_500_4	1601,29	1471,51	1536,40
15_25_4	776,76	32,74	404,75
15_50_4	785,02	76,75	430,88
15_100_4	1185,53	842,05	1013,79
15_150_4	1343,85	1250,99	1297,42
15_250_4	1727,62	1433,89	1580,76
15_500_4	1398,88	1456,86	1427,87
20_25_4	512,91	22,54	267,72
20_50_4	248,37	100,36	174,36
20_100_4	1019,21	1291,70	1155,46
20_150_4	1851,37	1495,46	1673,42
20_250_4	2136,44	1656,05	1896,25
20_500_4	1724,87	1570,46	1647,66

Table 3.9: Total computational times

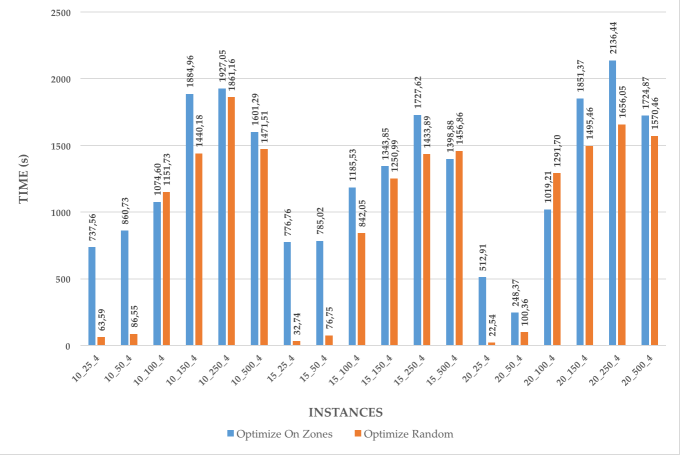
Instances	**Total Opt (s)**	**Total Random (s)**	**Average Total (s)**
10_25_4	738,35	64,28	401,32
10_50_4	862,92	88,90	475,91
10_100_4	1088,68	1164,08	1126,38
10_150_4	1914,62	1475,26	1694,94
10_250_4	2046,59	1996,79	2021,69
10_500_4	2266,66	2203,47	2235,06
15_25_4	777,40	33,43	405,41
15_50_4	787,01	79,42	433,21
15_100_4	1194,84	851,79	1023,32
15_150_4	1372,38	1279,58	1325,98
15_250_4	1842,60	1547,04	1694,82
15_500_4	2138,66	2221,55	2180,10
20_25_4	513,84	23,53	268,69
20_50_4	251,06	103,08	177,07
20_100_4	1027,29	1300,34	1163,82
20_150_4	1879,07	1525,88	1702,48
20_250_4	2235,32	1757,39	1996,35
20_500_4	2275,35	2506,14	2390,75



(a) `Total`

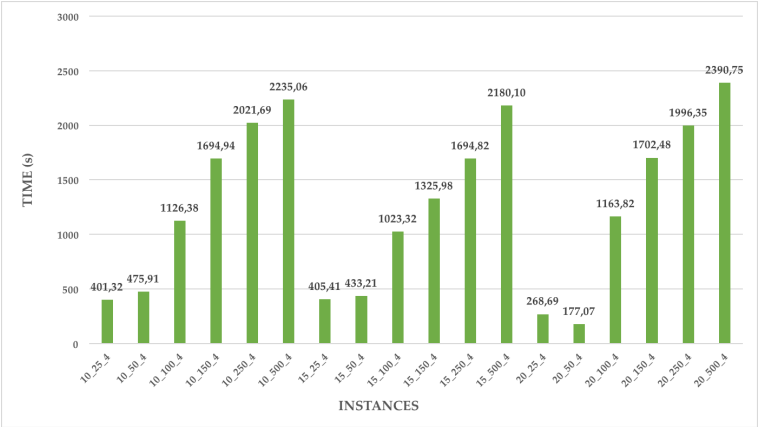


(b) `Initial`

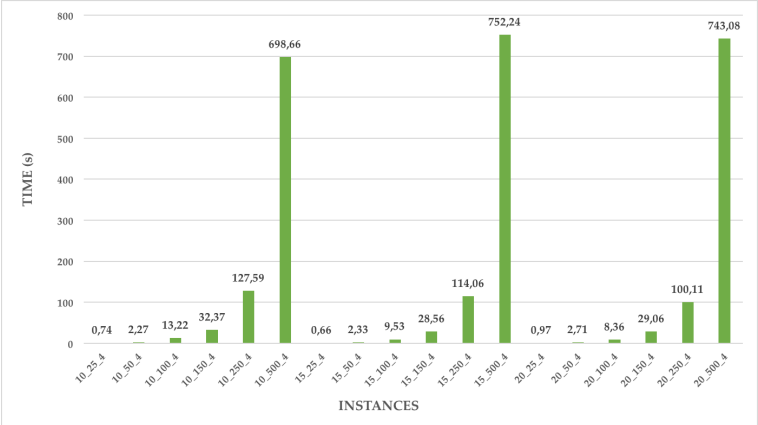


(c) `ZonesVNS`

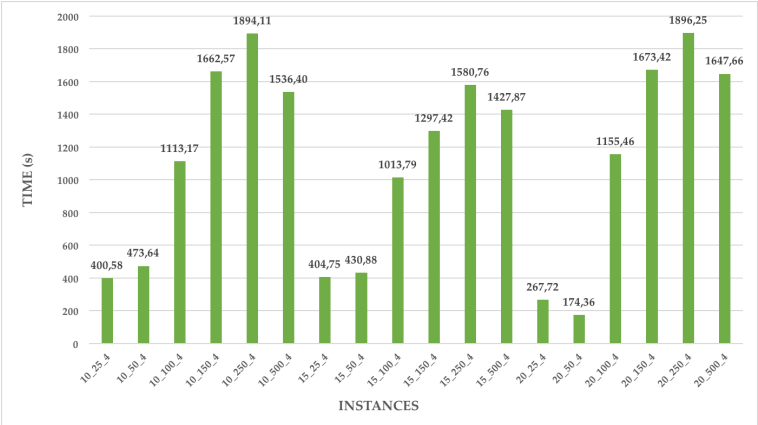
Figure 3.11: Computational times



(a) `Total`



(b) `Initial`



(c) `ZonesVNS`

Figure 3.12: Average computational times

The average time required to compute the initial solution is 148,14 seconds, while the optimization through the ZonesVNS takes 1113,93 seconds. As said in Subsection 2.5.4, one of the stopping criteria of the ZonesVNS is the computational time and in these tests 1200 seconds were set as the limit; the check is done at the beginning of each iteration of the ZonesVNS. This means that the execution may be longer, if the limit is exceeded during the phases within an iteration. In fact, the average total time is 1262,07 seconds, and the worst instances in terms of duration are the 10_150_4_1 (3120,63 s) and 20_100_4_3 (3449,38 s) for *Optimize On Zones* and for *Optimize Random*, respectively. The shortest ones are instead *20_50_4_3* (52,04 s) and *20_25_4_1* (18,36 s).

Computational times related to the number of zones

Observing Table 3.10, it is possible to notice that the average total time generally does not depend on the number of zones. The available time for computation is fully exploited almost equally with 10, 15 or 20 zones, meaning that ZonesVNS iterations need it all; many tests finished their execution because the time limit was reached. It would be interesting to concede larger intervals of time and see if results could be improved more.

Table 3.10: Average computational times related to the number of zones

# Zones	Average Init(s)	Average VNS(s)	Average Total(s)
10	145,81	1180,08	1325,88
15	151,23	1025,91	1177,14
20	147,38	1135,81	1283,19

Also focusing just on the computational time of the initial solution shows that there is not a proper relation only to the number of zones, but the number of customers should be considered above all.

The number of time windows and the initial solution

Rather than zones, the number of time windows can be related to the computational time of the initial solution generation. Obviously increasing the number of zones and customers makes also the number of time windows augment. Moreover, as defined in Section 2.3, each zone can have at most as many time windows as the number of periods of the time horizon. During the *BuildInitialSolution* method (see Subsection 2.5.3) the new time window to be defined for a zone is associated with a random period not already used for the current zone itself; if the other time windows, already assigned to the zone, are many, then finding a free period to assign may be not fast.

Fig. 3.13 shows how the computational time of the initial solution raises exponentially, increasing the number of time windows.

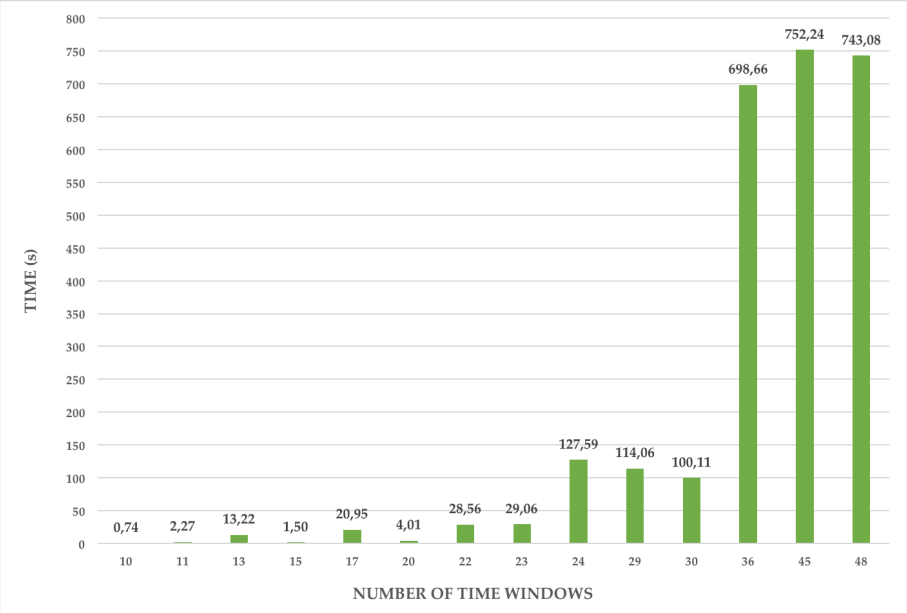


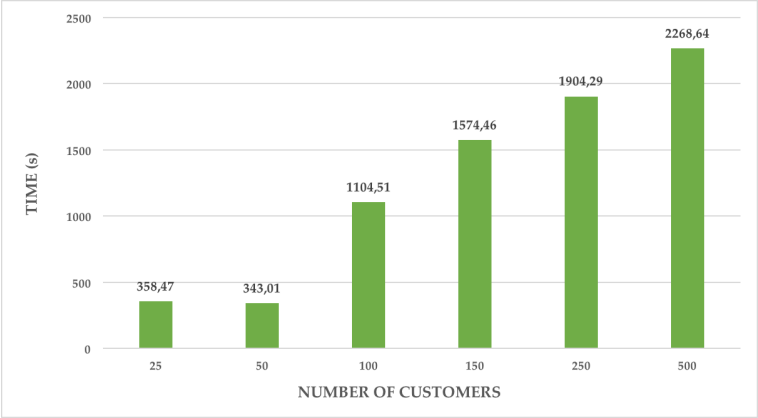
Figure 3.13: Average initial solution time related to the number of time windows

Computational times related to the number of customers

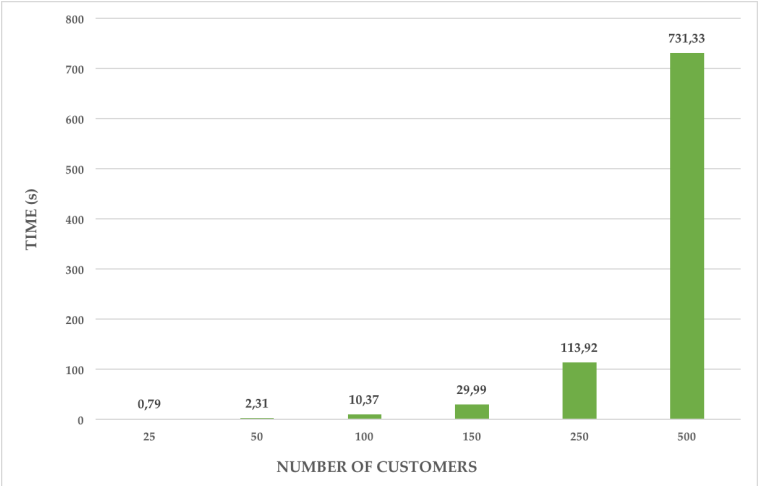
In Table 3.11 and Fig. 3.14 it is presented how computational times depend on the number of customers. Smallest instances with an average of 25 or 50 customers are very fast during both the computation of the initial solution and the optimization, requiring about 350 seconds totally. Largest ones instead need more time to generate and improve the solution, about 2000 seconds on average.

Table 3.11: Average computational times related to the number of customers

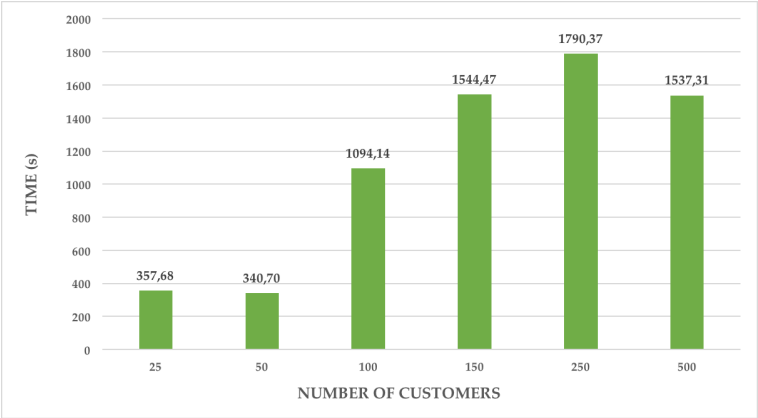
# Customers	Average Init(s)	Average VNS(s)	Average Total(s)
25	0,79	357,68	358,47
50	2,31	340,70	343,01
100	10,37	1094,14	1104,51
150	29,99	1544,47	1574,46
250	113,92	1790,37	1904,29
500	731,33	1537,31	2268,64



(a) `Total`



(b) `Initial`



(c) `ZonesVNS`

Figure 3.14: Computational times related to the number of customers

3.2.3 Drivers

From the results some observations can be done about the number of drivers used to satisfy all requests.

Average number of drivers

Applying both *Optimize On Zones* and *Optimize Random*, the average number of drivers required is 13. Looking at Table 3.12, it can be noticed that the maximum number of drivers exploited is 32, for serving customers of instances *15_500_4*, whereas the minimum value is about 4, for instances *10_25_4*.

Table 3.12: Number of drivers used

Instances	**Average # Opt**	**Average # Random**
10_25_4	4,50	4,42
10_50_4	5,58	5,50
10_100_4	8,67	8,17
10_150_4	9,58	10,25
10_250_4	15,00	15,33
10_500_4	28,33	26,58
15_25_4	5,00	5,25
15_50_4	7,67	7,67
15_100_4	9,92	9,33
15_150_4	10,83	11,50
15_250_4	17,00	17,08
15_500_4	32,00	31,83
20_25_4	6,17	6,33
20_50_4	8,33	8,42
20_100_4	9,33	9,58
20_150_4	12,67	12,25
20_250_4	15,25	16,58
20_500_4	27,83	28,25
	12,98	**13,02**

The trend is showed in Fig. 3.15 where it can be observed that the number raises in about the same way for instances of different grids.

How the number of drivers is instead related to the number of customers is represented in Fig. 3.16.

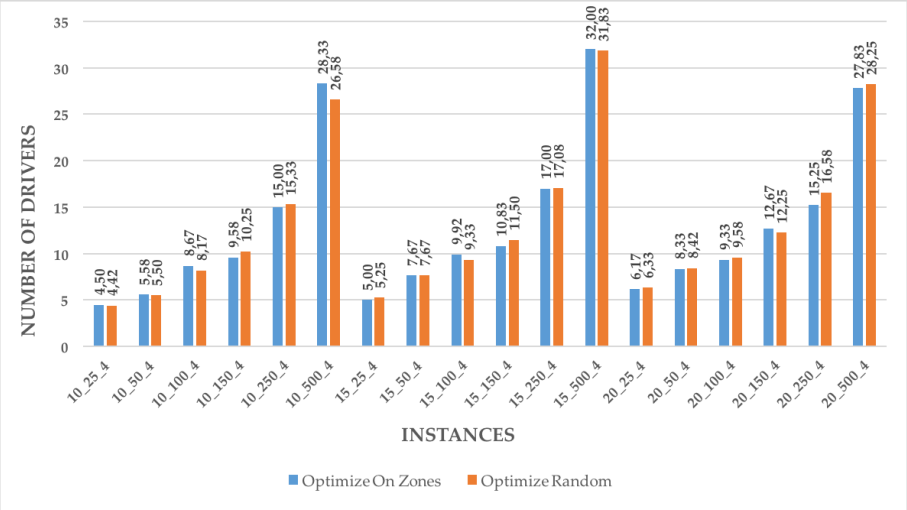


Figure 3.15: Number of drivers

Figure 3.16: Number of drivers related to the number of customers

Last Note

"Time will discover everything to posterity;
it is a babbler, and speaks even when no question is put."
Euripides

The *Stochastic Multi-period Time Window Assignment Problem* proposes a different and innovative point of view for studying the *Vehicle Routing Problem*, focusing specially on the concept of *time window* and, generally, discussing how time can be a constraint when providing a service to customers. The delivery area is composed of zones, which are associated with sets of time intervals for delivery. The objective of the problem was to find a good way to compute a *schedule*, programming each delivery day, trying to minimize the total cost for the company.

A literature review has been done, to discuss the state of art about VRP; the analysis has been concentrated on variants including time windows and zones. The SMTWAP has been modeled as a *Two-stage* problem. For the first stage, a variant of the *Variable Neighborhood Search* algorithm has been designed, for handling and managing zones and time windows. In the second stage, the heuristics of ALNS have been used to solve a VRP with time windows. Changing and modifying the time windows assigned to the zones allows to obtain several timetables, to evaluate in the possible scenarios.

Benchmark instances have been generated, with a different number of zones (from 10 to 20) and customers (from 25 up to about 500). They all have been solved, providing textual and graphic solutions. The developed solving approach improves initial solutions by 60% on average.

Future improvements can involve several aspects.
As far as algorithm's implementation is concerned, a greedy construction algorithm should be developed to compute the initial assignment of time windows to zones, substituting the current method which is completely random. The two ZonesVNS variants, *Optimize On Zones* and *Optimize Random*, should be enhanced in order to exploit better zones and time windows statistics, finding new criteria to apply moves in the first stage. Other moves, such as removing a not used time window, may be added to the current list.

Focusing on computational tests, it would be interesting to define and solve the *Deterministic Equivalent* of the SMTWAP and see how the solution would mutate when changing problem data. Moreover, given this formulation, it would

be possible to perform an analysis varying the number of scenarios, studying the trend of the error computed between the *Deterministic Equivalent* solution and the *Two-Stage* one. More time could be given to the ZonesVNS in order to visit a larger number of solutions and, therefore, to hopefully find better schedules. New sets of instances could be used to evaluate performances, trying new combinations between the number of zones and the number of customers, maybe changing the time horizon. It would be fascinating also using data coming from companies, applying the problem to real cases.

Finally, variants of the SMTWAP may be discussed and studied, for example allowing split delivery, using a heterogenous fleet of vehicles or adding more features to time windows, such as incentives or penalties.

References

[1] N. Agatz, A. Campbell, M. Fleischmann, M. Savelsbergh (2011), *Time Slot Management in Attended Home Delivery*, Transportation Science Vol. 45 (3), pp. 435-449, http://dx.doi.org/10.1287/trsc.1100.0346.

[2] C. Archetti, O. Jabali, M.G. Speranza (2015), *Multi-period Vehicle Routing Problem with Due dates*, Computers & Operations Research, Vol. 61, pp. 122–134, http://dx.doi.org/10.1016/j.cor.2015.03.014.

[3] E.J. Beltrami, L.D. Bodin (1974), *Networks and vehicle routing for municipal waste collection*, Networks, Vol. 4 (1), pp. 65-94, http://dx.doi.org/10.1002/net.3230040106.

[4] J.R. Birge, F. Louveaux (2011), *Introduction to Stochastic Programming*, Second Edition, Springer Series in Operations Research and Financial Engineering, Springer, http://dx.doi.org/10.1007/978-1-4614-0237-4.

[5] D.O. Casco, B.L. Golden, E.A. Wasil (1988), *Vehicle Routing with Backhauls: Models, Algorithms and Case Studies* in *Vehicle Routing: Methods and Studies*, B.L. Golden, A.A. Assad (eds.), Elsevier Science Publishers, Vol. 16, pp. 127-147.

[6] C.H. Christiansen, J. Lysgaard (2007), *A branch-and-price algorithm for the capacitated vehicle routing problem with stochastic demands*, Operations Research Letters Vol. 35 (6), pp. 773–781, http://dx.doi.org/10.1016/j.orl.2006.12.009.

[7] J.F. Cordeau, G. Desaulniers, J. Desrosiers, M. M. Solomon, F. Soumis (2002), *The VRP with Time Windows*, in *The Vehicle Routing Problem*, P. Toth, D. Vigo (eds.), SIAM Monographs on Discrete Mathematics and Applications, pp. 157-194.

[8] G.B. Dantzig, J.H. Ramser (1959), *The Truck Dispatching Problem*, Management Science, Vol. 6 (1), pp. 80-91, INFORMS, http://www.jstor.org/stable/2627477.

[9] I. Dayarian, T.G. Crainic, M. Gendreau, W. Rei (2015), *A branch-and-price approach for a multi-period vehicle routing problem*, Computers & Operations Research, Vol. 55, pp. 167–184, http://dx.doi.org/10.1016/j.cor.2014.06.004.

[10] G. Desaulniers, S. Ropke, O. Madsen (2014), *The vehicle routing problem with time windows* in *Vehicle routing: Problems, methods, and applications*,

P. Toth, D. Vigo (eds.), SIAM Monographs on Discrete Mathematics and Applications, pp. 119-159.

[11] M. Desrochers, J.K. Lenstra, M.W.P. Savelsbergh, F. Soumis (1988), *Vehicle Routing with Time Windows: Optimization and Approximation*, in *Vehicle Routing: Methods and Studies*, B. Golden and A. Assad (eds.), Elsevier Science Publishers, pp. 65–84.

[12] M. Dror, G. Laporte, P. Trudeau (1989), *Vehicle routing with stochastic demands: Properties and solution frameworks* Transportation Science, Vol. 23 (3), pp. 166-176, http://dx.doi.org/10.1287/trsc.23.3.166.

[13] M. Dror, G. Laporte, F.V. Louveaux (1993), *Vehicle routing with stochastic demands and restricted failures*, Zeitschrift für Operations Research, Vol. 37 (3), pp. 273-283, http://dx.doi.org/10.1007/BF01415995.

[14] M. Dror, G. Laporte, P. Trudeau (1994), *Vehicle routing with split deliveries*, Discrete Applied Mathematics, Vol. 50 (3), pp. 239–254, http://dx.doi.org/10.1016/0166-218X(92)00172-I.

[15] F. Errico, G. Desaulniers, M. Gendreau, W. Rei, L.M. Rousseau (2013), *Vehicle routing problem with hard time windows and stochastic service time*, Cahier du GERAD, G-2013-45, http://www.isci.cl/tristan/data/Routing/TRISTAN8_paper_111.pdf.

[16] F. Errico, G. Desaulniers, M. Gendreau, W. Rei, L.M. Rousseau (2016), *A priori optimization with recourse for the vehicle routing problem with hard time windows and stochastic service times*, European Journal of Operational Research, Vol. 249 (1), pp. 55–66, http://dx.doi.org/10.1016/j.ejor.2015.07.027.

[17] C. Gauvin, Guy Desaulniers, M. Gendreau (2014), *A branch-cut-and-price algorithm for the vehicle routing problem with stochastic demands* Computers & Operations Research, Vol. 50, pp. 141–153, http://dx.doi.org/10.1016/j.cor.2014.03.028.

[18] M. Gendreau, G. Laporte, R. Séguin (1996), *Stochastic vehicle routing* European Journal of Operational Research, Vol. 88 (1), pp. 3-12, http://dx.doi.org/10.1016/0377-2217(95)00050-X.

[19] M. Gendreau, G. Laporte, C. Musaraganyi, E.D. Taillard (1999), *A tabu search heuristic for the heterogeneous fleet vehicle routing problem*, Computers & Operations Research, Vol. 26 (12), pp. 1153–1173, http://dx.doi.org/10.1016/S0305-0548(98)00100-2.

[20] B.L. Golden, A.A. Assad, L. Levy, F.G. Gheysens (1984), *The fleet size and mix vehicle routing problem*, Computers & OR, Vol. 11 (1), pp. 49-66, http://dx.doi.org/10.1016/0305-0548(84)90007-8.

[21] C. Groër, B. Golden, E. Wasil (2009), *The Consistent Vehicle Routing Problem*, Manufacturing & Service Operations Management, INFORMS Vol. 11 (4), pp. 630–643, http://dx.doi.org/10.1287/msom.1080.0243.

[22] F. Hernandez, M. Gendreau, J-Y. Potvin (2014), *Heuristics for Time Slot Management: A Periodic Vehicle Routing Problem View*, CIRRELT, https://www.cirrelt.ca/DocumentsTravail/CIRRELT-2014-59.pdf.

[23] G. Laporte (2009), *Fifty Years of Vehicle Routing*, Transportation Science, Vol. 43 (4), pp. 408–416, http://dx.doi.org/10.1287/trsc.1090.0301.

[24] F. Li, B. Golden, E. Wasil (2005), *Very large-scale vehicle routing: New test problems, algorithms, and results*, Computers & Operations Research, Vol. 32 (5), pp. 1165-1179, http://dx.doi.org/10.1016/j.cor.2003.10.002.

[25] Z. Luoa, H. Qinb, D. Zhangc, A. Lima (2016), *Adaptive large neighborhood search heuristics for the vehicle routing problem with stochastic demands and weight-related cost*, Transportation Research Part E: Logistics and Transportation Review, Vol. 85, pp. 69–89, http://dx.doi.org/10.1016/j.tre.2015.11.004.

[26] C. Malandraki, M.S. Daskin, (1992), *Time-Dependent Vehicle Routing Problems - Formulations, Properties and Heuristic Algorithms*, Transportation Science, Vol. 26 (3), pp. 185-200, http://dx.doi.org/10.1287/trsc.26.3.185.

[27] N. Mladenović, P. Hansen (1997), *Variable neighborhood search*, Computers & Operations Research, Volume 24, pp. 1097–1100, http://sci2s.ugr.es/sites/default/files/files/Teaching/GraduatesCourses/Metaheuristicas/Bibliography/HansenVNS.pdf.

[28] D. Pisinger, S. Ropke (2007) *A general heuristic for vehicle routing problems*, Computers & Operations Research, Vol. 34 (8), pp. 2403-2435, http://dx.doi.org/10.1016/j.cor.2005.09.012.

[29] J-Y. Potvin, J-M. Rousseau (1993), *A parallel route building algorithm for the vehicle routing and scheduling problem with time windows*, European Journal of Operational Research, Vol. 66, pp. 331–340, http://dx.doi.org/10.1016/0377-2217(93)90221-8.

[30] S. Ropke, D. Pisinger (2006) *An Adaptive Large Neighborhood Search Heuristic for the Pickup and Delivery Problem with Time Windows*, Transportation Science, Vol. 40 (4), pp. 455–472, http://dx.doi.org/10.1287/trsc.1050.0135.

[31] M.W.P. Savelsbergh (1992), *The vehicle routing problem with time windows: Minimizing route duration*, ORSA Journal on Computing, Vol. 4 (2), pp. 146–154, http://dx.doi.org/10.1287/ijoc.4.2.146.

[32] P. Shaw (1997); *A new local search algorithm providing high quality solutions to vehicle routing problems*, Technical report, Department of Computer Science, University of Strathclyde, Scotland, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.51.1273&rep=rep1&type=pdf.

[33] M.M. Solomon, J. Desrosiers (1988), *Time Window Constrained Routing and Scheduling Problems*, Transportation Science, Vol. 22 (1), pp. 1-13, http://dx.doi.org/10.1287/trsc.22.1.1.

[34] R. Spliet (2013), *Vehicle Routing with Uncertain Demand*, ERIM Ph.D. Series Research in Management, Erasmus University Rotterdam, http://hdl.handle.net/1765/41513.

[35] R. Spliet, A.F. Gabor (2014), *The Time Window Assignment Vehicle Routing Problem*, Transportation Science, Vol. 49 (4), pp. 721-731, http://dx.doi.org/10.1287/trsc.2013.0510.

[36] R. Spliet, G. Desaulniers (2015), *The Discrete Time Window Assignment Vehicle Routing Problem*, European Journal of Operational Research, Vol. 244 (2), pp. 379-391, http://dx.doi.org/10.1016/j.ejor.2015.01.020.

[37] W.R. Stewart Jr., B. Golden (1983), *Stochastic vehicle routing: A comprehensive approach*, European Journal of Operational Research, Vol. 14 (4), pp. 371-385, http://dx.doi.org/10.1016/0377-2217(83)90237-0.

[38] D. Taş, N. Dellaert, T. van Woensel, A.G. de Kok (2013), *Vehicle routing problem with stochastic travel times including soft time windows and service costs* European Journal of Operational Research, Vol. 40 (1), pp. 214–224, http://cms.ieis.tue.nl/Beta/Files/WorkingPapers/wp_364.pdf.

[39] D. Taş, M. Gendreau, N. Dellaert, T. van Woensel, A.G. de Kok (2014), *Vehicle routing with soft time windows and stochastic travel times: A column generation and branch-and-price solution approach*, European Journal of Operational Research, Vol. 236 (3), pp. 789–799, http://dx.doi.org/10.1016/j.ejor.2013.05.024.

[40] A. van Breedam, J-F. Cordeau, J. Homberger, R.A. Russell, M.M. Solomon, VRPTW instances, http://neo.lcc.uma.es/vrp/vrp-instances/capacitated-vrp-with-time-windows-instances/.