ERASMUS UNIVERSITY ROTTERDAM
Erasmus School of Economics

Vehicle Routing with Departure Smoothing

Master's Thesis

Econometrics and Management Science

Author:
N.M. van der Zon

Student ID:
387441

Supervisors:
dr. R. Spliet (ESE)
dr. ir. A.L. Kok (ORTEC)

Co-reader:
prof. dr. A.P.M. Wagelmans

November 5, 2017

Abstract

This thesis investigates the vehicle routing problem with limited processing capacity at the depot. In this problem, there can only be a certain number of vehicles loading at the same time, depending on the number of loading docks available. Furthermore, there is also a constraint on the number of goods that can be loaded per time unit. We develop an adaptive large neighborhood search algorithm that, while constructing the vehicle routes, shifts the start times of the routes in a smart way, which we call departure smoothing, such that the processing capacity constraints are met. Computational experiments on both real-world and theoretical problem instances show that the algorithm performs well and is able to find good solutions in reasonable time. Finally, we also compare the method with a two-phase approach, which first constructs the vehicle routes and applies departure smoothing afterwards. Experiments demonstrate that the integrated approach presented in this thesis obtains better solutions in less running time.

Key words: *vehicle routing; ruin-and-recreate; start time optimization; departure smoothing; adaptive large neighborhood search*

Contents

1 Introduction **1**
1.1 ORTEC . . . 1
1.1.1 Departure smoothing at ORTEC . . . 2
1.1.2 Case description . . . 2
1.2 Thesis structure . . . 3

2 Problem Definition **4**
2.1 Vehicle routing problem . . . 4
2.2 Components . . . 4
2.2.1 Vehicle . . . 4
2.2.2 Customer . . . 5
2.2.3 Depot . . . 5
2.3 Objective . . . 8

3 Literature Review **9**
3.1 Inter-route constraints . . . 9
3.2 Multi-trip VRP with time windows . . . 10
3.2.1 Heuristic methods . . . 10
3.2.2 Exact methods . . . 10
3.3 Attributes . . . 11
3.3.1 Multiple trips . . . 11
3.3.2 Multiple time windows . . . 11
3.3.3 Multiple depots . . . 12
3.4 Departure smoothing . . . 12

4 Adaptive Large Neighborhood Search **14**
4.1 Ruin-and-recreate . . . 14
4.2 Algorithm outline . . . 14
4.3 Construction heuristic . . . 16
4.4 Destruction operators . . . 17
4.4.1 Customer level . . . 17
4.4.2 Trip level . . . 18
4.4.3 Route level . . . 19
4.5 Reconstruction operators . . . 19
4.6 Acceptance rule . . . 20
4.7 Weight adaption . . . 20
4.8 Stopping criterion . . . 21

5 Extended ALNS 22
5.1 Attributes . . . 22
5.1.1 Multiple time windows . . . 22
5.1.2 Multiple depots . . . 22
5.1.3 Heterogeneous fleet . . . 23
5.2 Local search phase . . . 23
5.2.1 Local search operators . . . 23
5.3 Parallel computing . . . 24
5.3.1 Synchronization step . . . 24
5.3.2 Process configuration . . . 25

6 Integrated Smoothing 26
6.1 Smoothing phase . . . 26
6.1.1 Smoothing operators . . . 27
6.2 Capacity allocation . . . 29
6.2.1 Computational issues . . . 29
6.2.2 Allocation heuristic . . . 30
6.3 Additional destruction operators . . . 31
6.3.1 Customer level . . . 31
6.3.2 Trip level . . . 32
6.4 Two-phase smoothing . . . 32

7 Computational Results 34
7.1 Algorithm configuration . . . 34
7.2 Practical case . . . 36
7.2.1 Problem instances . . . 36
7.2.2 Constant processing capacity . . . 37
7.2.3 Time-varying processing capacity . . . 41
7.3 Theoretical case . . . 44
7.3.1 Problem instances . . . 44
7.3.2 Results . . . 44

8 Conclusion 48

References 49

A List of Symbols 51

B Additional results 53
B.1 Constant processing capacity . . . 53
B.2 Time-varying processing capacity . . . 55

1 Introduction

Many companies face transportation problems in which vehicle routes have to be planned. It is important for these companies to have routes which are as efficient as possible, which translates to lower costs. Because of this, the *vehicle routing problem* (VRP) has been studied extensively. A lot of possible extensions of the vehicle routing problem have been studied as well, in order to have a model that is closer to practice. For example, we could have a routing problem where certain customers have time windows in which the goods must be delivered. Another example is a problem in which different types of trucks can be used, where each type of truck has its own characteristics. VRPs in which multiple of these extensions are taken into account are often called *rich vehicle routing problems* (RVRPs) (Caceres-Cruz et al., 2015).

Even though many different extensions of the VRP have been studied, one component of vehicle routing problems in practice is rarely taken into account in the literature, which is the limited processing capacity at the depot. There are only a certain number of loading docks available at the depot, which means we can only have a certain number of vehicles loading at the same time. Furthermore, the speed at which vehicles can be loaded depends on the number of workers available. This processing capacity does not have to be constant and can change at any time during the day. If this aspect is not taken into account, the limited capacity at the depot might cause congestion during peak hours, when the start times of trips tend to cluster. Because of this congestion, the vehicles already start their trips with a delay, which might mean that certain customers cannot be served within their respective time window anymore. This delay also affects successor trips of the vehicle, for which the same problem occurs. It is therefore very important to also take this aspect of the problem into account.

In this thesis, we look at a possible solution to this problem, which we call *departure smoothing*. By shifting the start times of the trips in a smart way, i.e., smoothing the trips, we can satisfy the capacity constraints and avoid congestion at the depots. For this, we develop a large neighborhood search algorithm, based on the work of Azi et al. (2014), which also takes the processing capacities and the start times of the trips into account.

1.1 ORTEC

This research takes place at ORTEC, which is one of the largest providers of optimization software and analytics solutions. ORTEC's optimization software can handle many types of problems, including fleet routing and dispatch, pallet and space loading, workforce scheduling and warehouse control. The software is built on years of experience and incorporates the latest optimization techniques, which is necessary to be able to solve the new complex problems that arise.

1.1.1 Departure smoothing at ORTEC

One of those new problems is departure smoothing, which ORTEC had to find a solution for. Departure smoothing is still a relatively new topic at ORTEC and most of the methods have just recently been developed. It might therefore be interesting to briefly describe the relationship between their work and this thesis. Most importantly, there are two main differences.

The first difference is that ORTEC's current departure smoothing method works in two phases. In the first phase, the routes are planned with an additional constraint to ensure that the start times can be smoothed. This constraint makes sure that the used capacity at each depot is within a certain threshold, which can be slightly higher than the actual capacity, such that there are enough possibilities to shift the start times of the trips. Then, in the second phase, a heuristic is used to actually shift the departure times such that the capacity constraint is met at all the depots. In this thesis, we consider an alternative approach. We look at a completely new integrated method, which already applies smoothing while planning the routes. This is then compared to ORTEC's two-phase method.

The second difference is that ORTEC's method can only handle at most one of the two depot capacity constraint that we mentioned, i.e., the method can either take the number of loading docks into account or the speed at which the vehicles can be loaded. In practice, you want a solution method that can take both constraints into account, which our method is able to do.

1.1.2 Case description

In this thesis, we look at a rich vehicle routing problem from one of ORTEC's clients. This client is a large retail company with more than two thousand stores, and therefore faces a large vehicle routing problem. A lot of different characteristics are taken into account in this problem. For example, the company works with a heterogeneous fleet of vehicles, and each vehicle can execute multiple trips on one day. There are multiple depots at which a vehicle can start, and each store can be served from any depot. Each store has to be served by a vehicle, and this must be done within a certain time window. A store can have multiple time windows in which the goods can be delivered. Finally, as already mentioned, there are capacity constraints for the number of loading docks as well as for the number of workers. These capacities are time-varying and can also differ per depot.

The main goal of this thesis is to develop a method that can efficiently find good solutions to large instances of this rich vehicle routing problem. It is especially important to have a method that can handle the two types of processing capacity constraints at the depot, as this is something that is definitely relevant for many vehicle routing problems in practice. In this thesis, the focus will therefore be mostly on the departure smoothing problem.

1.2 Thesis structure

The structure of this thesis is as follows. In Chapter 2, we give a precise definition of the problem. Chapter 3 reviews and discusses relevant literature for our problem. In Chapter 4, we review the adaptive large neighborhood search algorithm (ALNS) as described in Azi et al. (2014), which can be seen as the basis of our departure smoothing method. Chapter 5 provides some further modifications to the ALNS algorithm in order to find better solutions. In Chapter 6, we describe our departure smoothing method. Chapter 7 gives the results of computational experiments. Finally, Chapter 8 gives a conclusion.

2 Problem Definition

In this section, we give a definition of our problem. We start with a general overview of the vehicle routing problem that we consider. After that, we break the problem into several components and examine each of these in more detail.

2.1 Vehicle routing problem

At the base of our vehicle routing problem is a complete directed graph $N = (V, A)$ representing the network. In this graph, V is the set of all vertices and A is the set of all arcs between the vertices. Each arc $(i, j) \in A$ with $i, j \in V$ has a corresponding travel distance d_{ij} and travel time t_{ij}. The set of vertices can be divided into two distinct sets, i.e., $V = C \cup D$, where C is the set of all customer nodes and D is the set of all depot nodes. At each depot $d \in D$, there is a fleet of vehicles K_d available to serve customers, which is done by constructing vehicle routes. A route can consist of multiple trips going from a depot to one or several customers and back to the same depot again. The main goal is to construct efficient routes, i.e., with minimal distance, such that all customers in the network are served using as few vehicles as possible. Our objective function, which we will discuss in detail in Section 2.3, therefore contains both the total distance of all the routes and the number of vehicles used.

2.2 Components

The problem consists of several components, i.e., vehicles, customers and depots. For each component we have several characteristics and constraints that we have to take into account. In this subsection, we take a look at each of the components and provide some further explanation.

2.2.1 Vehicle

As mentioned, at each depot $d \in D$ there is a fleet of vehicles K_d available. The entire fleet of vehicles is denoted by the set $K = \cup_{d \in D} K_d$. There are multiple types of vehicles W. For each type $w \in W$, there is a corresponding vehicle capacity c_w. Each vehicle $k \in K$ has exactly one vehicle type assigned to it. To simplify the notation, we denote w_k as the type, c_k as the capacity and d_k as the depot of vehicle k. Furthermore, each vehicle k drives a certain route R_k, which we have to construct. This route represents the workday of the vehicle and can consist of multiple trips, i.e., $R_k = \{S_k^j : j = 1, \ldots, n_k^s\}$, where n_k^s is the total number of trips done by vehicle k.

All trips are contained in S, which denotes the set of all trips. Each trip $s \in S$ starts and ends at a certain depot $d \in D$, which means we can divide the set S into multiple subsets, i.e, $S = \cup_{d \in D} S_d$, where S_d is the set of trips that start at depot d. Furthermore, for each trip $s \in S$, we have a decision variable τ_s representing the start time at which

the vehicle leaves the depot. This means that the vehicle must be completely loaded *before* τ_s. Once the vehicle is loaded, it visits one or multiple customers, which are contained in the set C_s, and then goes back to the depot. The total volume of the goods of all the customers on a trip may not exceed the vehicle capacity.

Finally, denote σ_s as the time needed to load the vehicle for trip s. This loading time depends on the volume of the goods that have to be loaded, i.e., it takes longer to fully load a truck than only a part of it. Next to that, as we will show, it also depends on the processing capacity of the depot. If many vehicles are loading at the same time, the available workers have to be distributed over more vehicles, and it can take longer for a vehicle to load. This will be explained in detail when we describe the depot characteristics.

2.2.2 Customer

Each customer $i \in C$ has a certain location, represented by a node in the network, and a certain demand quantity q_i. This quantity has to be delivered within one of the customer's time windows, denoted by the set $TW_i = \{[a_j, b_j] : j = 1, \ldots, n_i^{tw}\}$, where a_j and b_j are respectively the times at which time interval j starts and ends, and n_i^{tw} is the total number of time windows customer i has. Next to that, each customer $i \in C$ also has an associated service time t_i^s that represents the time needed to unload and handle the order at customer i. Note that it is not required that the arrival time of the vehicle plus the service time is within a time window. The constraint is satisfied as long as the start time of service at the customer is within one of its time windows.

Finally, some customers can only be served by certain vehicle types. For example, at certain customers there is not enough room for a large truck to unload, which means that the customer can only be served by a smaller truck. Another example is that the customer has goods that need to stay refrigerated during transport, for which a special truck is needed. For all customers $i \in C, W_i \subseteq W$ is the set of vehicle types that can serve customer i.

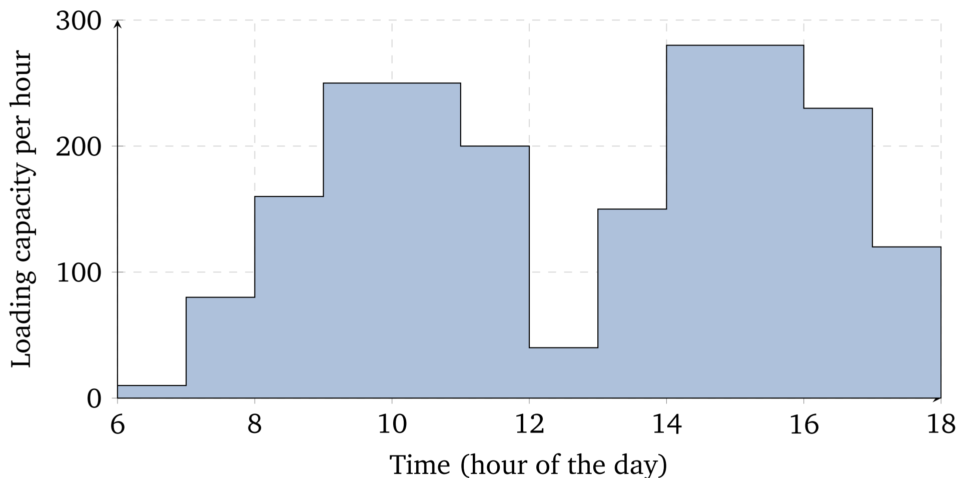
2.2.3 Depot

The main characteristic of the depot is the processing capacity. As mentioned in the introduction, this capacity is very important for our problem. There are two types of capacity constraints that we take into account for each depot $d \in D$. Both capacities can be time-varying. Instead of looking at continuous time, we divide the whole day into multiple time intervals $T = \{t_1, \ldots, t_{n_t}\}$ of equal length, where each t_i for $i = 1, \ldots, n_t$ is an interval in which the processing capacity is constant. It is required that these intervals cover the whole day, such that the capacity is known at any time. Furthermore, the start time of each vehicle trip can only be at the beginning of a time interval $t \in T$. By considering discrete time intervals, we greatly simplify the departure smoothing problem by reducing the number of possible start times we have to consider for each trip. Furthermore, having a finite number of time intervals T also helps us to define the

capacity allocation problem. This problem is about distributing the available capacity at the depot over the vehicles that have to be loaded, and must be solved each time we make a change to a trip or a route. Before we formalize the capacity allocation problem, we first introduce the two types of processing capacity constraints that we consider.

The first constraint is the limited number of loading docks l_{dt} that are available during time interval $t \in T$ at depot $d \in D$. This means that only a certain number of vehicles can load at the same time. The second constraint is the loading capacity. Denote p_{dt} as the total number of goods that can be loaded during time interval t at depot d. This capacity depends on the number of workers that are available at the depot, which might vary a lot during the day. An example of this is shown in Figure 1, which shows the loading capacity (in pallets per hour) during the day. This means that each of the time intervals in the set T has a length of one hour in this case. This figure shows a clear pattern, where most workers are available from 9:00-11:00 and from 14:00-16:00. Furthermore, there is a clear lunch-break between 12:00 and 13:00.

Figure 1: Example of the loading capacity at depot



Unlike the constraint for the limited number of loading docks, the loading capacity is a soft constraint, i.e., there is a certain penalty cost $b \cdot \max(0, p_{dt} - p^*_{dt})$ when the loading capacity is exceeded, where b is the cost per extra unit of loading capacity required and p^*_{dt} is the used loading capacity during time interval t at depot d. In reality, this means that some workers may have to do some overtime or temporary workers are hired.

Capacity allocation problem

Finally, when constructing the routes, we know the loading capacity and the number of vehicles at the loading docks within each time interval. However, we also have to decide how to distribute the loading capacity within this time interval over the vehicles. In other words, we also have a subproblem on how to allocate the workers to the loading docks for any given time interval.

Let $S_{dt} \subseteq S_d$ be the set of trips that can load at depot d in time interval t. Whether a vehicle can load for trip s during a certain time interval depends on several factors. First of all, the vehicle must be loaded before the start time of the trip τ_s, which means that any time interval in which the vehicle is loaded must also be before τ_s. Furthermore, if trip s has any preceding trip, the vehicle can only load in the time intervals after the end time of this trip, i.e., the time at which the vehicle is back at the depot. To ensure that there is at least some time available to load the vehicle in between trips, we require that there are at least T^S intervals of slack between the end time and the start time of two successive trips of a vehicle. Finally, we only allow a vehicle to load if the time interval is within T^L intervals of the start time. This prevents solutions in which, e.g., a vehicle is completely loaded a couple of hours before the start time, which is not desirable in practice.

The capacity allocation problem is defined as follows: for each $d \in D$ and $t \in T$, we have to construct a distribution function $\alpha_{dt}(s)$ which tells us for each trip $s \in S_{dt}$ which part $x \in [0, p_{dt}]$ of the total loading capacity is allocated to trip s. In order for the distribution functions to be feasible, the following two inequalities must hold for each individual distribution function:

1. We cannot allocate more capacity than available within each time interval.

$$\sum_{s \in S_{dt}} \alpha_{dt}(s) \leq p_{dt} \qquad \forall d \in D, \quad t \in T$$

2. The total number of loading docks cannot be exceeded.

$$\sum_{s \in S_{dt}} I_{>0}(\alpha_{dt}(s)) \leq l_{dt} \qquad \forall d \in D, \quad t \in T$$

   where $I_{>0}(x)$ is an indicator equal to 1 if $x > 0$ and 0 otherwise.

Furthermore, it is required that all vehicles are completely loaded on time, i.e., before the start time of the trip. This means we also have a set of constraints that must hold jointly for all distribution functions at each depot $d \in D$:

$$\sum_{t \in T : t \leq T_s^*} \alpha_{dt}(s) = \sum_{i \in C_s} q_i \qquad \forall d \in D, \quad s \in S_d$$

where the right part of the equation represents the total demand of all the customers on the trip. T_s^* is the latest time interval in which a vehicle can get loaded, which is the time interval immediately preceding the start time of the trip τ_s.

The time needed to load the vehicle for every trip $s \in S$, denoted as σ_s, can then easily be derived from the selected distribution functions by taking the difference between the last time interval and the first time interval in which the vehicle is loaded, i.e., the highest and lowest value of t for which $\alpha_{dt}(s) > 0$.

2.3 Objective

Our objective consists of multiple parts, i.e., we want to have routes with minimal distance, satisfying all constraints using as few vehicles as possible. However, in order to easily compare solutions, we want to have a single objective value. This is done by looking at cost minimization, where we convert each part to a corresponding cost value. This allows us to express the objective as a linear combination

$$\text{minimize} \;\; f(D,V,P) = D + V + P$$

where D is the cost of the total distance travelled by all vehicles, V is the cost of the total number of vehicles used, and P is the penalty cost of exceeding the depot loading capacity. Both D and V are relatively straightforward and can be easily calculated by multiplying the total distance and total number of vehicles used by, respectively, the cost per distance unit and cost per vehicle. It is assumed that the cost per distance unit is constant. The cost per vehicle depends on the vehicle type. The penalty cost P of exceeding the depot loading capacity is calculated in the following way:

$$P = \sum_{d \in D} \sum_{t \in T} b \cdot \max(0, p_{dt} - p^*_{dt})$$

where b is the cost per extra unit of loading capacity required and p^*_{dt} is the used loading capacity during time interval t at depot d. This means we have a fixed penalty for each unit of loading capacity exceeded.

3 Literature Review

Vehicle routing problems have been studied extensively in the past decades. The problem is often extended with several characteristics, often referred to as *attributes*, such as vehicle capacities, time-windows, working-hour regulations, etc. Most of these attributes are based on real-life problem instances.

To get an overview of the many possible attributes that are studied already, Vidal et al. (2013) give an extensive survey of 15 different VRP attributes. They distinguish between three main classes of attributes, namely *ASSIGN*, *SEQ* and *EVAL*. ASSIGN attributes deal with the assignment of limited resources, such as multiple depots, heterogeneous fleets, inventory, etc. SEQ attributes impact the structure of the routes, such as multiple trips or truck-and-trailer problems. Finally, EVAL attributes are probably the most common and concern the evaluation and constraint checks when the route is constructed. Examples of EVAL attributes are time windows, loading constraints and working regulations. For each of the attributes in a certain class, the solution approach is relatively similar. In the problem we consider, we have to deal with attributes from all three classes. This means we are required to combine several solution approaches, which makes the problem very challenging.

3.1 Inter-route constraints

We can also make a distinction between *intra-route constraints*, which only apply to a single route, and *inter-route constraints*, which affect multiple routes at the same time. Vehicle capacities and time windows are examples of intra-route constraints, and the limited loading capacities at the depot are inter-route constraints. Because multiple routes are affected, it is clear that inter-route constraints are often more difficult to deal with. Inter-route constraints for VRPs are currently an emerging field in research, mostly because there are many VRPs in practice that benefit from a good solution method to deal with these type of constraints.

Hempsch and Irnich (2008) provide a model that can handle all kinds of inter-route constraints. This model is based on resource-constrained paths and the giant-tour representation, in which all vehicle routes are concatenated to form one long tour. They also show how to use efficient neighborhood search methods when the constraints satisfy some special requirements.

Drexl (2012) gives a survey of synchronization in VRPs, which are also inter-route constraints. An example of a synchronization problem is if two vehicles have to be at the same location at the same time during their route. This affects multiple routes, in this case two, and is therefore an inter-route constraint. However, there are many more types of synchronization. The depot processing capacity in our problem is a case of resource synchronization. As an example of resource synchronization problems, the survey refers to Ebben et al. (2005), who consider a problem with multiple resource constraints, including a limited number of loading docks.

3.2 Multi-trip VRP with time windows

In this thesis, we consider a VRP with multiple time windows, multiple depots and a heterogeneous fleet of vehicles that can perform multiple trips. In addition to this, we also have the loading capacity constraints related to departure smoothing. To the best of our knowledge, this problem as a whole has not been studied yet. However, the problem can be seen as an extension of the *multi-trip vehicle routing problem with time windows* (MTVRPTW), which has been researched before. We will therefore discuss a couple of solution methods for the MTVRPTW that are described in the literature. In the MTVRPTW, there is only one time window for each customer and only one depot. However, the methods for the MTVRPTW that we will discuss could still be used as a starting point for our extended problem.

3.2.1 Heuristic methods

Cattaruzza et al. (2016) provide a hybrid genetic algorithm for the *multi-trip VRP with time windows and release dates* (MTVRPTW-R). The release dates model the (external) arrival of goods at the depot. This means that vehicles can only load after the goods become available at the depot, i.e., after the release date. The MTVRPTW is a special case of this problem, which can be obtained by setting all release dates to zero.

An important class of heuristics for vehicle routing problems use the *ruin-and-recreate* principle (Schrimpf et al., 2000). By iteratively destroying and rebuilding solution parts, it is possible to obtain good solutions to large VRPs. Azi et al. (2014) consider an *adaptive large neighborhood search* (ALNS) for the MTVRPTW. By assigning weights to the various construction and destruction heuristics, the best performing heuristics get a larger weight after several iterations. This way the algorithm can find good solutions for many types of problems, for which it is not known in advance which heuristic performs best.

Another heuristic is described by Brandao and Mercer (1997), where they solve the MTVRPTW with a tabu search algorithm. They also consider both heterogeneous vehicles and driver legislation, however their problem only has single time windows. To assess the performance, the authors simplified their heuristic (Brandao and Mercer, 1998) by removing several characteristics (such as hetereogeneous vehicles and time windows), which allowed for a comparison with the tabu search algorithm of Taillard et al. (1996). Brandao and Mercer (1998) show that although the solution cost for their heuristic is on average slightly higher, the simplified heuristic gives more balanced trips in less computation time.

3.2.2 Exact methods

Several other papers on the MTVRPTW propose exact methods. Hernandez et al. (2014) provide a two-phase branch and price algorithm to solve the MTVRPTW with limited duration. In this problem there is a limit on the duration of each trip. This algorithm

could be used to solve the MTVRPTW, by making the maximum allowed duration arbitrarily large. However, because it relies on full enumeration of the possible trips, this becomes impossible very quickly when there is no limit on the duration. Instead, Hernandez et al. (2016) created two different branch-and-price algorithms for the regular MTVRPTW without limited duration that do not use full enumeration.

Azi et al. (2010) also consider a branch-and-price approach, where the lower bounds are obtained by solving the LP relaxation of a set packing formulation using column generation. The pricing problems are shortest path problems with resource constraints. Furthermore, they also provide computational results for standard benchmark instances.

The main disadvantage of the exact methods is that, in practice, they can only be applied to very small instances of up to 50 customers. In the problem we consider in this thesis, we have to deal with more than 500 customers, which is too many to be able to use any exact method. This is why we will purely look at heuristic methods.

3.3 Attributes

The problem that we consider has many different attributes, such as a heterogeneous fleet, multiple time windows, multiple trips, multiple depots, etc. In this section, we look into three of these attributes in more detail. For some of the attributes we provide some relevant papers with solution approaches and ideas that can be useful for our own solution methods.

3.3.1 Multiple trips

Şen and Bülbül (2008) give a survey on the multi-trip VRP. They also provide computational results to compare several methods. One of the possible approaches to go from a single-trip VRP solution to a multi-trip solution is by solving a bin packing problem. In this problem, each bin corresponds to a vehicle. Single trips have to be allocated to the bins while making sure that the allocation is feasible. Quite a few papers describe methods that use this approach and solve the bin packing problem heuristically (Brandao and Mercer, 1997; Taillard et al., 1996; Petch and Salhi, 2003; Salhi and Petch, 2007).

As an alternative to the bin packing approach, Cattaruzza et al. (2014) look at a hybrid-genetic algorithm to solve the multi-trip VRP. As already mentioned, the same authors extended this algorithm to solve the MTVRPTW (Cattaruzza et al., 2016), which can also handle time windows.

3.3.2 Multiple time windows

Compared to the other attributes, not much research has been done on the VRP with multiple time windows. Favaretto et al. (2007) provide an *ant colony optimization*

(ACO) algorithm that can solve the routing problem with multiple time windows. An alternative heuristic is described by Belhaiza et al. (2014), who consider a hybrid variable neighborhood tabu search heuristic. Some of the computational results are also compared with those of Favaretto et al. (2007). This comparison shows that the heuristic by Belhaiza et al. (2014) clearly performs better, with objective values that are around 20% lower than obtained by the algorithm of Favaretto et al. (2007).

3.3.3 Multiple depots

A survey on the VRP with multiple depots (MDVRP) is given by Montoya-Torres et al. (2015). One of the more simple approaches is to use a two-step procedure. First link customers to the closest depot and then solve the routing problem for each depot separately. Many methods have been developed in the past that are improvements of this approach. Thangiah and Salhi (2001) use a hybrid-genetic algorithm to define clusters of customers, which they call *genetic clustering*. Each cluster is then linked to a certain depot. More recently, Yücenur and Demirel (2011) also look at a genetic clustering algorithm. This method is then compared to the nearest neighbor algorithm, for which they show that the genetic clustering algorithm provides better results.

3.4 Departure smoothing

Even though many different attributes for the VRP have been studied already, the literature on departure smoothing is scarce. Gromicho et al. (2012) consider, next to several other attributes, a constraint for the limited number of loading docks at the depot. This is also the case in our problem, however, we also have an additional constraint for the limited speed of the order pickers.

Hempsch and Irnich (2008) look at the limited capacity for arrivals at the depot instead, which you could call arrival smoothing. They consider both a constraint for the limited number of loading docks and a constraint for limited processing capacities for incoming goods from vehicles that arrive at the depot. This is applicable to VRPs such as letter mail collection from several postboxes in a city. In their case, the processing capacities can be nicely modeled by their general inter-route constraint framework. Because their processing capacities satisfy some special requirements, efficient neighborhood search methods are available. Our constraints do not satisfy these requirements, which makes their framework less suitable for our problem.

Dabia et al. (2014) consider limited loading capacity at the depot. They look at multiple shifts on a day, i.e., a day shift, evening shift and a night shift, and for each shift there is a certain loading capacity. This is a simplified version of our case in which the capacity can change more often during the day. Furthermore, they solve the problem exactly by using column generation, which is too expensive in our case because of the problem size. They looked at instances with a maximum of 100 customers, and we have to deal with instances that are at least five times larger.

It is also possible to use a two-phase approach to solve the problem. In that case, the routes are first planned without taking the processing capacities into account. Then, once the trips are known, the start times are shifted in such a way that capacity constraints are satisfied as well as possible. This second phase then becomes a type of *job shop scheduling problem* that has to be solved for each depot (Beck et al., 2003). In this case, each depot has a certain number of machines, represented by the loading docks, and the trips are the jobs that have to be scheduled. For each job (trip) we have a certain release date and a due date, which represent the interval in which we can shift with the start time. The processing capacities are the side-constraints that have to be satisfied.

4 Adaptive Large Neighborhood Search

As a basis to solve our vehicle routing problem, we use *adaptive large neighborhood search* (ALNS), as described by Azi et al. (2014). This algorithm has been tested on MTVRPTW instances of comparable size, and it can easily handle multiple depots, multiple time windows and a heterogeneous fleet of vehicles as well. In this section, we first explain the ruin-and-recreate principle in general, which is the main idea behind ALNS. After that we show the ALNS algorithm as described in Azi et al. (2014). In the next chapter, we extend the algorithm in order to support the remaining attributes, i.e., multiple depots, multiple time windows and a heterogeneous fleet of vehicles. Finally, in Chapter 6 we extend the ALNS algorithm to allow for departure smoothing.

4.1 Ruin-and-recreate

ALNS is based on the *ruin-and-recreate* principle, which is described by Schrimpf et al. (2000). The basic idea of this principle is to find better solutions by sequentially destroying and reconstructing parts of the current solution. This approach has an important advantage. When destroying a part of the solution, we have a lot of freedom to construct a new solution. This allows us to explore a large part of the solution space and hopefully find a solution that is close to the global optimum.

If we apply this principle to VRPs, we remove a number of customers from the vehicle trips in the solution and then reinsert these customers in the solution again, which can be in a different trip or at a different position in the trip. This is done by destruction and reconstruction operators. In general, there can be many ways to destroy and reconstruct a solution, which means we can also have many operators. Our main focus is on defining suitable operators which can be used to solve our problem.

4.2 Algorithm outline

As mentioned, ALNS is based on the work of Schrimpf et al. (2000), who already apply ruin-and-recreate to VRP problems. The main difference is that ALNS extends this approach by allowing multiple destruction and reconstruction operators in a smart way. At the beginning of each iteration, a pair consisting of one destruction and one reconstruction operator is randomly chosen. This operator pair is then used to destroy and reconstruct the solution.

The selection procedure is then improved by making it adaptive. This means that the probability for an operator to be selected depends on its past performance. If a certain operator has shown good performance in the previous iterations, we want to have a high chance of using this operator again in future iterations. This is done by giving each operator a certain weight, which is contained in the vector ω. These weights are then taken into account when randomly selecting the operators.

Furthermore, Azi et al. (2014) describe multiple levels Z on which the destruction operators can work. The main idea is to start on a high level and destroy larger parts of the solution and then slowly move to lower levels and destroy smaller parts. This means that the algorithm focuses on diversification at the start, such that it explores many regions of the solution space. Once we are at a lower level, the focus switches to intensification, where the algorithm tries to find the (local) optimum in that part of the solution space. Azi et al. (2014) consider three levels, namely $Z = \{\text{route, trip, customer}\}$ as shown in Figure 2. Route is the highest level and customer the lowest. This means that when a destruction operator works on, e.g., route level, it can destroy entire routes. The same holds for the trip and customer levels, where the operator respectively removes trips and customers from the solution.

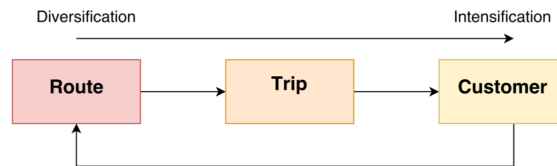


Figure 2: The three levels for the destruction operators

Each level is explored for a certain number of iterations l before moving on to the next level. Summarized, we get the following outline of the ALNS algorithm:

Algorithm 1 ALNS

Step 1. Initialization

Before we can start destroying and reconstructing, we have to construct an initial solution s first. This is done by using a construction heuristic. Because the initial solution is the only solution we have at the moment, we also set it as best found solution s^*. Next to that, the weights ω for the destruction and reconstruction operators have to be initialized. Finally, set the current level z to the highest level.

Step 2. Destruct & reconstruct

While taking the weights ω into account, randomly select a destruction and reconstruction operator. Obtain a new solution by applying the selected destruction and reconstruction operators to the current solution.

Step 3. Acceptance rule

Once we have obtained a new solution s', we compare it with our current solution s. If the new solution is better, then we always accept it and set it as current solution s. Otherwise, we accept the new solution with a certain probability. Finally, we also check if the new solution is better than the best solution s^* found so far and update it if needed.

Step 4. Check stopping criterion

If the stopping criterion is met, the algorithm is done and we return the best found solution s^*. Otherwise, start a new iteration by going back to Step 2. If l iterations have been done on the current level, move down one level and adjust the weights based on the performance of the operators before going back to Step 2. If there is no lower level, go back to the highest level again.

The ALNS algorithm consists of many components that we have to specify in more detail. In the remainder of this section, we look at the construction heuristic, the destruction and reconstruction operators, the acceptance rule, weight adaptation and the stopping criterion as presented in Azi et al. (2014).

4.3 Construction heuristic

In order to construct the initial solution, we use a construction heuristic (Algorithm 2). This heuristic considers each customer in a random order and inserts the customer at the best feasible place, i.e., gives the smallest increase in objective value. This is done by going over all feasible positions in all trips of all vehicles and computing the increase in objective value. In case no feasible insertion place exists, a split procedure (Algorithm 3) is used to try to create feasible insertion places. This procedure goes over all non-empty trips of all vehicles and splits each trip into two sub-trips in every possible way, i.e., between every pair of consecutive customers in the trip. These two sub-trips are then searched for feasible places. If the split procedure does not result in any feasible place, the customer is put in a list of temporarily unserved customers.

Algorithm 2 Construction heuristic

$C' \leftarrow C$
while $C' \neq \emptyset$ **do**
  select customer $c \in C'$ at random;
  for all $k \in K$ **do**
    for all $s \in R_k$ **do**
      compute feasible least insertion cost for customer c in trip s;
    end for
  end for
  if feasible insertion place exists **then**
    insert customer c in trip with least cost;
  else
    apply *split(c)* procedure;
  end if
  $C' \leftarrow C' \setminus \{c\}$;
end while

In order to determine if an insertion place is feasible, we have to check if service at each customer can start within one of its time windows. This means we have to compute the service start time τ_i for each customer i on the trip. This is done by starting the trip at time τ_s and then iteratively going over all customers in the trip (including the customer that we want to insert at the given insertion place) and keeping track of the time passed. This means we consider the travel time when travelling between two customers, the service time when starting the service at a customer, and the waiting time if we have to wait before we can start service (when we arrive before the start of a time window). After computing the start time of service for each customer in the trip, we can easily check if the start time is within one of the time windows for each customer.

Algorithm 3 Split(c) procedure

for every trip s **do**
  for every pair of consecutive customers i and j in trip s **do**
    split trip s into two sub-trips s_1 and s_2 such that i is the last customer in s_1 and j is the first customer in s_2;
    compute feasible least insertion cost for customer c in trips s_1 and s_2;
  end for
end for
if at least one feasible place found **then**
  insert customer c in place with least cost;
else
  put c in a list of temporarily unserved customers;
end if

Azi et al. (2014) also describe an algorithm to reduce trip duration after each customer insertion by letting the trip start at the latest feasible time. In our case, it is better to not include this algorithm. The main reason is that due to the processing capacities, it does not have to be better to start a trip at the latest feasible time. Instead, in some cases, it could be better to start earlier to avoid exceeding the capacity. This is examined in detail in Chapter 6, when we extend the ALNS algorithm to include departure smoothing.

4.4 Destruction operators

The destruction operators destroy parts of the current solution, which can then be reconstructed again. As mentioned, each destruction operator works on a certain level. We therefore define the operators per level, i.e., customer, trip and route.

4.4.1 Customer level

The destruction operators defined at customer level remove n_1 customers from the current solution. Azi et al. (2014) consider two types of destruction operators.

Random customer destruction:
The n_1 customers to be removed are selected completely at random.

Related customer destruction:
This operator (Algorithm 4) starts by randomly selecting a customer for removal. The other $n_1 - 1$ customers are then iteratively selected in such a way that there is a higher chance of removal when a customer is closely related to one of the previous selected customers.

Algorithm 4 Related customer destruction

randomly select a customer $i \in C$ and remove it from the solution;
$L \leftarrow \{i\}$;
while $|L| < n_1$ **do**
  $i \leftarrow$ randomly select a customer $i \in L$;
  for every customer j in the solution **do**
    $z_{ij} \leftarrow \alpha \cdot |\tau_i - \tau_j| + \beta \cdot d_{ij}$;
  end for
  sort z_{ij} values in non-decreasing order and place them in a list B;
  $pos \leftarrow \lceil |B| x^y \rceil$, where x $\in [0, 1]$ is randomly selected;
  select customer j associated with position pos and remove from the solution;
  $L \leftarrow L \cup \{j\}$;
end while

For every pair of customers, the proximity is measured by looking at both the start time of the service and the location, weighted by parameters α and β. This gives a score z_{ij} for every customer pair i, j, where i is one of the previously removed customers and j is one of the customers that is still included in the solution. Based on these scores a customer is selected for removal, i.e., a lower score means a higher probability for a customer to be selected. The intensity of the bias is controlled by parameter y. This means that if y has a high value, the scores are taken into account more strongly, compared to a lower value of y.

4.4.2 Trip level

The operators defined at trip level remove n_2 trips from the current solution. Azi et al. (2014) again consider two types of destruction operators.

Random trip destruction:
The n_2 trips to be removed are selected completely at random.

Related trip destruction:
Similar to related customer destruction, this operator (Algorithm 5) removes trips that are closely related. The pseudocode is therefore almost the same. However, instead of only looking at only one proximity measure, Azi et al. (2014) consider two measures.

The first measure is the distance between the gravity centers of the two trips, where the gravity center is defined as the average location over all customer locations in a trip. The second measure looks at the minimum distance between pairs of customers k, l for every customer k in the trip i and customer l in the trip j.

Algorithm 5 Related trip destruction

randomly select a trip i and remove it from the solution;
$L \leftarrow \{i\}$;
while $|L| < n_2$ **do**
  $i \leftarrow$ randomly select a trip $i \in L$;
  for every trip j in the solution **do**
    compute score z_{ij} using a proximity measure;
  end for
  sort z_{ij} values in non-decreasing order and place them in a list B;
  $pos \leftarrow \lceil |B| x^y \rceil$, where x $\in [0, 1]$ is randomly selected;
  select trip j associated with position pos and remove from the solution;
  $L \leftarrow L \cup \{j\}$;
end while

4.4.3 Route level

The operators defined at route level remove n_3 routes from the current solution. For this level, Azi et al. (2014) look at only one operator.

Random route destruction:
Just as for the customer and trip levels, we also have a random destruction operator on route level where the n_3 routes to be removed from the current solution are selected completely at random.

4.5 Reconstruction operators

After the destruction operator is done, we have to insert the removed customers back in the solution. This is done by a reconstruction operator. Azi et al. (2014) consider two types of reconstruction operators.

Least-cost:
The least-cost reconstruction operator is essentially the construction heuristic (Algorithm 2) applied only to the customers that still have to be inserted. This means that for each customer we choose the (feasible) place with the lowest insertion cost. The order in which the customers are inserted is random.

Regret-based:
Because least-cost reconstruction can be too greedy at times, the regret-based operator provides an alternative that is more balanced. Instead of only looking at the best place, i.e., with the least insertion cost, we can also look at the second-best place or at the third-best place. In general, we can define a *k-regret* operator, which takes the best k places into account. For each customer $i \in C$, we compute the following measure:

$$reg_i = \sum_{j=1}^{k} \left(\Delta d_i^j - \Delta d_i^1 \right)$$

where Δd_i^j is the increase in distance when inserting customer i at the jth best place. The customer with the highest regret measure is then inserted at the place with the least insertion cost. This is continued until all customers are inserted. In the results of Azi et al. (2014), they only used a 2-regret reconstruction operator.

4.6 Acceptance rule

At the end of each destruct-reconstruct iteration we obtain a new solution s'. As mentioned in the outline, we always accept this new solution if it is better than the current solution s. If it is worse than the current solution, we accept it with probability $e^{(f(s')-f(s))/\beta_T}$ as done in simulated annealing (Kirkpatrick et al., 1983), where β_T is the temperature parameter and f is our objective function. Otherwise the solution is rejected, which means we go back to the latest accepted solution.

By accepting worse solutions with a certain probability, we explore a large part of the solution space and therefore decrease the chance of getting stuck in a bad local optimum. Furthermore, we start the algorithm with a high temperature β_T, which means the acceptance probability is high in the beginning. After each iteration, the temperature gets updated by setting $\beta_T \leftarrow c \cdot \beta_T$, with $0 < c < 1$, which leads to a lower probability of accepting a worse solution in later iterations.

4.7 Weight adaption

As mentioned, weights are used during the selection of the destruction and reconstruction operators. If a certain operator performs well, the weight corresponding to the operator will increase and it will be more likely for the operator to be chosen in future iterations. For example, if we have $j = 1, \dots, n$ operators to choose from and ω_j is the weight of operator j, then the probability of an operator i to be selected is:

$$P(\text{operator } i \text{ selected}) = \frac{\omega_i}{\sum_{j=1}^{n} \omega_j}.$$

The weights are adjusted every time the algorithm moves to a different level. At the beginning of a new level, we get the updated weight ω_i' for each operator i in the following way:

$$\omega_i' = \gamma \cdot \omega_i + (1-\gamma) \cdot \pi_i$$

where ω_i is the weight that was used during the previous level and $\gamma \in [0, 1]$ can be used to adjust the influence of the past performance of an operator in the weight update. If we choose γ close to one, the weights will stay relatively stable and only change in the long term. On the other hand, if we choose γ close to zero, the weights are more variable and mostly based on the performance during the previous level, i.e., the short term. Finally, π_i is a score which measures the performance of the operator during the previous level. The score of an operator is incremented at the end of each iteration it is used. The size of the increment depends on how well the operator performed, for which we consider four different possibilities.

1. If the algorithm found a *new best solution*, the scores of the destruction and reconstruction operators used get incremented by σ_1.
2. If the algorithm did not find a new best solution, but the found solution is *better than the current*, then the scores of the used operators get incremented by σ_2.
3. If the solution is *worse than the current one, but accepted*, the scores of the destruction and reconstruction operator get incremented by σ_3.
4. If the solution is *rejected*, the scores of the used operators do not change.

Finally, at the first iteration of a new level, all scores get reset to zero.

4.8 Stopping criterion

As stopping criterion, Azi et al. (2014) use a maximum number of iterations SC^i. In their case, they set SC^i to 24,000, which allowed for convergence even on their largest test instances.

5 Extended ALNS

In the previous chapter, we showed the adaptive large neighborhood search algorithm as described in Azi et al. (2014). In this chapter, we describe some further modifications that we have made to the ALNS algorithm. We first describe how we can extend the ALNS algorithm to handle the relevant attributes for our problem, such as multiple time windows, multiple depots, etc. Afterwards, we describe several modifications that can improve the quality of the found solutions.

5.1 Attributes

The ALNS algorithm as described in Azi et al. (2014) is originally developed for the multi-trip vehicle routing problem with time windows (MTVRPTW). In our case, we have a routing problem with more attributes, which means we have to extend the algorithm. In this section, we go over all relevant attributes for our problem and describe the changes that have to be made to the algorithm.

5.1.1 Multiple time windows

The ALNS algorithm already supports customers with single time windows. The algorithm is however easily extended to allow for multiple time windows per customer. Each time a customer is inserted in a trip using one of the reconstruction operators, we have to check for feasibility. During this feasibility check, we simply check if the service start time τ_i for all customers i is within one of the time windows of the customer. This is done in the same way as described in Section 4.3. When inserting a customer in a trip, we iteratively go over all customers in the trip (including the customer that we want to insert at the given insertion place) and keep track of the time passed. This means we start at τ_s and consider the travel time when travelling between two customers, the service time when starting the service at a customer, and the waiting time if we have to wait before we can start service (when we arrive before the start of a time window). After computing the start time of service for each customer in the trip, we can easily check if the start time is within one of the time windows for each customer.

5.1.2 Multiple depots

ALNS can also easily handle multiple depots if we make some assumptions. We assume that each vehicle is assigned to a certain depot, and cannot start or end any of its trips at a different depot. Furthermore, each customer can be served from any depot. After assigning each vehicle to a depot, we can simply use ALNS as described in the previous chapter.

5.1.3 Heterogeneous fleet

The ALNS algorithm as described in the previous chapter is already able to handle a heterogeneous fleet of vehicles. However, if some customers can only be served by a certain type of vehicle, we have to add this to the feasibility check that is performed for each customer insertion during the reconstruction phase. This means that when we go over all insertion places, i.e., every place in every trip of every route done by each vehicle $k \in K$, we check if the vehicle type w_k is contained in the set W_i of allowed vehicle types that can serve customer i.

5.2 Local search phase

In the ALNS algorithm as described, we try to obtain better solutions by applying the ruin-and-recreate principle, i.e., iteratively destroying and reconstruction parts of the solution. However, we could also try to improve the solution by applying local search methods. These methods can be incorporated in the ALNS algorithm by introducing a local search phase after reconstruction. This means that when all customers have been re-inserted, we select a local search method and apply it to the solution.

5.2.1 Local search operators

Similar to the reconstruction and deconstruction operators, we also define local search operators. At the start of each local search phase, we randomly select one of the operators. This selection procedure is also made adaptive in the same way as described in Section 4.7.

In order to keep the local search phase fast, we look at two relatively simple local search operators. Both operators perform one or multiple swap moves, in which two customers switch positions in the solution.

Random best swap:
This operator randomly selects n customers to swap. For each of these selected customers, we iterate over all other customers in all routes in the solution and compute the change in distance when swapping the positions of the two customers. The feasible swap which results in the highest decrease in distance is then applied to the solution. If all swaps are either infeasible or increase the total distance, the customer is not swapped.

Highest savings swap:
This operator works similar to the random best swap operator, however, in this case we select the n customers with the highest savings. The savings are computed by looking at the reduction in distance that is obtained when removing the customer from the route. Then, for each of these n customers, we perform the same steps as for the random best swap operator.

5.3 Parallel computing

As mentioned in Section 4.1, the ruin-and-recreate principle is used to explore a large part of the solution space. This means we have a higher chance of finding a solution close to the global optimum. We can explore an even larger part of the solution space by making use of parallel computing in a smart way. Figure 3 shows an example of how this is done using 3 processes running in parallel.

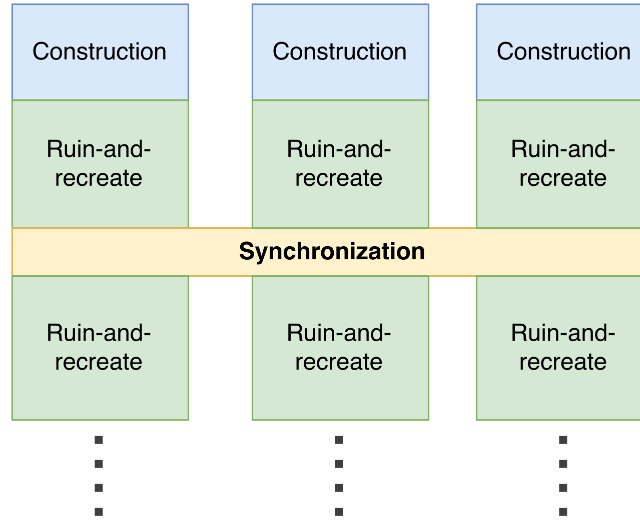


Figure 3: ALNS algorithm with three processes running in parallel

In general, we spawn n different processes which start by constructing n different solutions. Each of these processes then run a certain number of iterations of the ALNS algorithm, i.e., destruction, reconstruction and local search. After these iterations, there is a *synchronization step* in which the processes share their progress.

5.3.1 Synchronization step

The synchronization step consists of two parts. The first part is sending the current best known solution of the process to the other processes. After the solution is sent, the second part is checking the solutions which were sent by the other processes. If the received solution is better than the best known, both the current and best known solution are set to the solution which was received. Otherwise, if the solution is not better than the best known, the process ignores the received solution. Afterwards, each process runs a certain number of iterations of the ALNS algorithm again before it gets to the next synchronization step. This repeats until the stopping criteria are met.

An important aspect of the synchronization step is that it is *non-blocking*. This means that once a process is at the synchronization step, it does not wait for the other processes to be at the synchronization step as well. If some, but not all, processes have sent their solutions, the process will not wait on the remaining processes when checking the received solutions. This also means that if no other processes have sent their best

known solution yet, the synchronization step only consists of sending the best known solution to the other processes.

5.3.2 Process configuration

Another interesting aspect of running multiple processes in parallel is that we can have a separate configuration for each process. This means we are able to have each process use a different set of construction, reconstruction and local search operators for the ALNS algorithm. For example, we could have a process that only focuses on the customer level, and does not use the operators from the trip or route level. This could lead to further diversification and better solutions.

6 Integrated Smoothing

Using the adaptive large neighborhood search algorithm described in the previous two chapters, we are now able to describe the method used to solve the departure smoothing problem. As mentioned in the introduction, ORTEC currently uses a two-phase method in which they first plan the vehicle trips and afterwards shift the trip start times in a second phase. The method we propose is meant as an alternative to ORTEC's two-phase method and is called integrated smoothing. The main idea of integrated smoothing is that we combine the two phases into one single phase, in which we both plan the vehicle trips and shift the start times of the trips. This is done by introducing a smoothing phase in the ALNS algorithm.

Before we describe the smoothing phase, it might be useful to first give an overview of all the phases in the ALNS algorithm we have mentioned so far. This also shows how the smoothing phase fits into the ALNS algorithm. Figure 4 shows the four phases that are part of each ALNS iteration. For each phase, the figure also shows the objective that the phase focuses on.

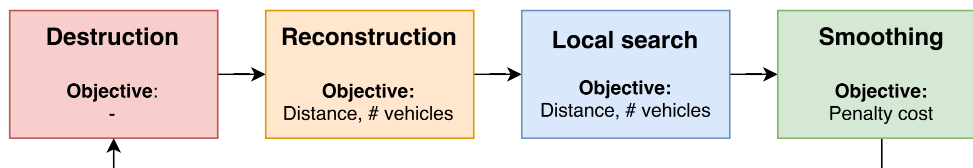


Figure 4: The four phases in each ALNS iteration, including the objective per phase

The ALNS algorithm as described by Azi et al. (2014) consists of only the destruction and reconstruction phases. This was then extended in the previous chapter by also including a local search phase after the solution has been reconstructed. The local search phase, together with the reconstruction phase, focuses purely on minimizing the costs for both the distance and the total number of vehicles used. The processing capacity at the depot are not taken into account at all during these two phases. After the local search phase, the smoothing phase focuses on reducing the penalty costs for exceeding the loading capacity at the depot. This then results into a new solution which is either accepted or rejected using the acceptance rule described in Section 4.6.

6.1 Smoothing phase

The smoothing phase takes place at the end of each ALNS iteration, i.e., after all customers have been re-inserted and the local search operators have been applied. During this smoothing phase, we try to reduce the penalty of exceeding the depot capacity by shifting the start times of the trips using smoothing operators.

6.1.1 Smoothing operators

The smoothing operators work similar to the other types of operators we defined earlier. At the beginning of the smoothing phase, we randomly select one of the available smoothing operators. This selection is also made adaptive by weighing each operator based on the past performance as described in Section 4.7.

The structure of each smoothing operator is very similar. Each smoothing operator selects n_s trips for which the start time should be shifted. Then, for each of these trips that have to be smoothed, the following four steps are applied.

Step 1. Determine lower bound on start time

To obtain a lower bound t_{LB} on the start time of a trip, we first check if the trip has any preceding trips in the route. If this is the case, the end time of the preceding trip gives us the lower bound. If there is no preceding trip, we use 'time zero' as lower bound, which represents the earliest possible time we consider in the problem.

Note that because the start time of the trip has to be at the beginning of a time interval $t \in T$, if the lower bound is not at the beginning of a time interval, we choose the beginning of the first time interval *after* the lower bound as the actual lower bound t_{LB}.

Step 2. Determine upper bound on start time

The upper bound t_{UB} on the start time is computed in a similar way as the lower bound. We first check if the trip s has any succeeding trips in the route. If this is the case, the start time of the succeeding trip gives us the upper bound. Otherwise, we first compute the smallest time difference between the service start time τ_i and the latest possible service start time of each customer $i \in C_s$. This represents the maximum time we can delay the start time compared to the current start time of the trip. The upper bound on the start time of trip s is then obtained by adding this maximum delay to the current start time.

Note again that we must ensure that the start time of the trip is at the beginning of a time interval $t \in T$. This is done by choosing the beginning of the first time interval *before* the upper bound as the actual upper bound t_{UB}.

Step 3. Iterate over possible start times

For each start time between t_{LB} and t_{UB}, we first perform a feasibility check to see if we can service each customer within one of its time windows. This is done in the same way as described in Section 4.3 by computing the start time of service at each customer in the trip. These service start times must be within one of the customer's time windows for each customer in the trip. If this is not the case, the start time is not feasible and we can continue to the next start time. Otherwise, we compute a chosen measure for this start time and continue to the next start time.

Step 4. Shift start time

Once we have iterated over all possible (feasible) start times, we choose the start

time that maximizes the chosen measure and set this time as the start time of trip s.

Each smoothing operator has to define two aspects. The first aspect is how to select the trips which should be smoothed and in which order. In this thesis, we consider two possible ways of selecting trips. The first one is simply selecting n_s trips randomly. The second method chooses n_s trips by looking at the exceeded loading capacity. For each trip s, we iterate over all time intervals t where $\alpha_{dt}(s) > 0$, i.e., in which the vehicle for trip s is being loaded, and compute the total loading capacity exceeded in these intervals. We then select the n_s trips with the highest total loading capacity exceeded.

Secondly, each smoothing operator has to define a suitable measure which can be used to select the best start time for each trip. We consider three different measures.

Weighted start time difference:

The first measure looks at the start time difference to other trips, i.e., we want to let each trip have a start time that is far away from the start times of other trips. This leads to less vehicles loading at the same time, which might be needed to satisfy the constraint for the limited number of loading docks. The measure is computed in the following way, for every trip s which has to be smoothed and possible start time t:

$$m^1_{st} = \sum_{s' \in S_d \setminus \{s\}} \frac{|t - \tau_{s'}|}{\sum_{c \in C_{s'}} q_c}$$

where d is the depot at which the vehicle doing trip s starts. The start time t which gives the highest value for the measure is the best start time for trip s. The start time difference to every other trip s' is weighted by dividing by the total demand quantity of all the customers on trip s'. This means that we prefer the start time to be far away from trips that have to deliver (and load) a larger quantity of goods, compared to trips that only have to deliver a smaller quantity. This can help reduce the penalty costs for exceeding the loading capacity.

Potential loading capacity:

The second measure looks at the potential loading capacity, i.e., we prefer a trip to be loading during the time intervals where the capacity is high. This gives us the following measure:

$$m^2_{st} = \sum_{t' \in T'_{st}} p_{dt'}$$

where T'_{st} are the time intervals in which trip s can load if it starts at time t, and d is the depot at which the vehicle doing trip s starts. The start time t which gives the highest value for the measure is the best start time for trip s. Compared to the weighted start time difference measure, this measure is especially helpful when the loading capacity varies a lot during the day, such as the example that was shown in Figure 1. This measure will recognize the time intervals during which the loading capacity is higher, and will shift the start time towards these intervals.

Actual loading capacity:

The third measure is very similar to the second measure. The main difference is that instead of looking at the potential loading capacity, we now look at the actual available loading capacity. This means we take into account the capacity that is already used by the other trips, which makes the measure more accurate. The measure is computed as follows:

$$m_{st}^3 = \sum_{t' \in T'_{st}} (p_{dt'} - p^*_{dt'})$$

where T'_{st} are again the time intervals in which trip s can load if it starts at time t, and d is the depot at which the vehicle doing trip s starts. The start time t which gives the highest value for the measure is the best start time for trip s. As we will describe in the next section, this measure is computationally more expensive than the second measure because we have to solve the capacity allocation problem in order to obtain the correct value of the used loading capacity $p^*_{dt'}$.

With the two possible ways of selecting trips and the three measures we have defined, we get six different smoothing operators we can choose from in the smoothing phase.

6.2 Capacity allocation

As mentioned in the problem definition, we also have a subproblem at the depot on how to distribute the loading capacity within each time interval over the vehicles. Each time we make a change to a trip, i.e., either a customer is inserted/removed or the start time is shifted, we have to solve the capacity allocation problem. Only after we have obtained a feasible allocation, the penalty cost for exceeding the depot loading capacity can be computed. In order to solve the capacity allocation problem relatively fast, we look at a heuristic method. However, before we describe our allocation heuristic, we first want to describe some computational issues.

6.2.1 Computational issues

Solving the capacity allocation problem for every small change made to any trip is computationally very expensive. We therefore want to reduce the number of times we have to solve the capacity allocation problem. As shown in Figure 4, we do not require the penalty cost during the destruction, reconstruction and local search phases. We therefore also do not need to solve the capacity allocation problem for any changes made to any trip during these three phases. This already greatly reduces the number of times we have to create a capacity allocation.

The smoothing phase does use the penalty costs as an objective, which means we do have to deal with the capacity allocation problem during this phase. However, we can still minimize the number of times we have to run the allocation heuristic. For the weighted start time difference and potential loading capacity measures described in the previous section, we do not use any information from the capacity allocation.

This means that when we are smoothing multiple trips in the smoothing phase, we do not have to solve the capacity allocation problem after shifting the start time of a trip, before being able to smooth the next trip. Only for the third measure, i.e., the actual loading capacity measure, we do have to solve the capacity allocation problem before we can smooth the next trip.

6.2.2 Allocation heuristic

The allocation heuristic (Algorithm 6) is inspired by the earliest due date (EDD) rule used for job scheduling problems. This rule means that priority is given based on which job has the earlier due date. In our case, the due dates correspond to the start times of the trips and we give priority to vehicles that have to be finished loading earlier.

Algorithm 6 Capacity allocation heuristic

Step 1. Initialization

For the given depot, set the earliest time interval $t \in T$ as the current time interval.

Step 2. Determine which vehicles can be loaded

Check for the current time interval which vehicles can be loaded, and place these vehicles in the set K'. Divide this set into multiple sets $K'_0, K'_1, \ldots, K'_{|T|}$, where the set K'_0 contains the vehicles that must be completely loaded at the end of the current time interval. K'_1 contains the remaining vehicles that must be finished loading at the end of the next time interval, etc.

Step 3. Allocate capacity

All vehicles in K'_0 must be completely loaded at the end of the current time interval, so we allocate the capacity needed to load all these vehicles, even if this exceeds the loading capacity p_{dt}. Note that if the number of available loading docks l_{dt} in the current time interval is smaller than $|K'_0|$, the allocation is not feasible and the heuristic stops.

If any capacity is left, we distribute this evenly over all vehicles in K'_1. If there are not enough loading docks available to distribute the capacity evenly, i.e., there are $x < |K'_1|$ loading docks available, we randomly select x vehicles in K'_1 and distribute the capacity evenly over these x vehicles. If there are any loading docks remaining, we then again check if any capacity is left, which we can distribute evenly over all vehicles in K'_2 in the same way. This continues until either no loading capacity is left for the current time interval, there are no more loading docks available in the current time interval, or all vehicles in K' are completely loaded.

Step 4. Move to next time interval

If there is a next time interval, set the current time interval to the next time interval. Otherwise, the heuristic is done.

6.3 Additional destruction operators

In order to obtain better solutions, we define some additional destruction operators to be used alongside the destruction operators defined in the previous chapter. Compared to the more general destruction operators in Section 4.4, these additional operators focus on trips or customers in trips that are not smoothed as much, e.g., trips with a start time close to the start time of other trips. We define one new operator on customer level and two on trip level.

6.3.1 Customer level

Just as the other destruction operators on customer level, this destruction operator removes n_1 customers from the solution.

Unsmoothed customer destruction:

This operator (Algorithm 7) selects customers for removal in such a way that there is a higher chance of removal when a customer has a high demand quantity and is in a trip that is loading during time intervals in which loading capacity is exceeded. This second part is denoted as $PT(s_i)$, and is computed as follows:

$$PT(s_i) = \sum_{t \in T: \alpha_{dt}(s_i) > 0} \max(0, p_{dt} - p^*_{dt})$$

where s_i is the trip containing customer i, d is the depot at which the trip s_i is loading, and p^*_{dt} is the used loading capacity at depot d during time interval t corresponding to allocation α_{dt}. By removing these customers and inserting them in a better place, we hope to reduce the total penalty costs.

Algorithm 7 Unsmoothed customer destruction

```
L ← {};
while |L| < n_1 do
    for every customer i in the solution do
        z_i ← α · q_i + β · PT(s_i);
    end for
    sort z_i values in non-increasing order and place them in a list B;
    pos ← ⌈|B|x^y⌉, where x ∈ [0, 1] is randomly selected;
    select customer i associated with position pos and remove it from the solution;
    L ← L ∪ {i};
end while
```

For every customer i in the solution, we compute a score z_i based on the two measures described above weighted by parameters α and β. We then select a customer for removal in the same way as done in the related customer destruction operator (Algorithm 4).

6.3.2 Trip level

Both destruction operators on trip level also remove n_2 trips from the solution.

Similar trip start time destruction:
This operator is the same as the related trip destruction operator (Algorithm 5), only we use a different proximity measure where we look at the difference in start time $|\tau_i - \tau_j|$ for any pair of trips i, j.

Unsmoothed trip destruction:
This operator (Algorithm 8) is very similar to the unsmoothed customer destruction operator. We again look at a measure consisting of the total demand quantity and the capacity exceeded during the intervals in which the trip is loading.

Algorithm 8 Unsmoothed trip destruction

$L \leftarrow \{\}$;
while $|L| < n_2$ **do**
  for every trip s in the solution **do**
    $z_s \leftarrow \sum_{j \in C_s} q_j + PT(s)$
  end for
  sort z_s values in non-increasing order and place them in a list B;
  $pos \leftarrow \lceil |B| x^y \rceil$, where $x \in [0, 1]$ is randomly selected;
  select trip s associated with position pos and remove it from the solution;
  $L \leftarrow L \cup \{s\}$;
end while

The selection is done in the same way as in the related trip destruction operator (Algorithm 5).

6.4 Two-phase smoothing

The two-phase smoothing method is ORTEC's current approach for solving the departure smoothing problem. In the introduction, we have already provided a brief description of some of the differences between their approach and our integrated smoothing approach. Because we also want to make a more extensive comparison between the two approaches, it is useful to describe the two-phase smoothing method in slightly more detail. In order to make a fair comparison between the integrated and two-phase approach, we have implemented a slightly modified version of ORTEC's two-phase algorithm which uses the same ALNS algorithm as described in chapters 4 and 5. This means that the same destruction, reconstruction and local search operators are used as in the integrated smoothing method.

In the first phase of the algorithm, the vehicle routes are planned. This is done using ALNS with an additional constraint. This constraint checks the 'smoothability' of the solution, i.e., if there are enough possibilities to shift the start times of the trips. This constraint works by determining a capacity allocation based on the earliest and latest

possible start time of each trip. If the allocation is not feasible, i.e., the capacity is exceeded in any of the time intervals, the start times are not flexible enough and the solution is infeasible as well.

The second phase of the algorithm is where the actual smoothing happens. This is done heuristically using a peak shaving approach. In each iteration of this heuristic, we select the trip which has the highest penalty costs for exceeding the loading capacity, go over all possible start times of this trip and select the start time which leads to the highest reduction in total penalty costs. If smoothing the trip with the highest penalty costs does not lead to a reduction in penalty costs, we try to smooth the trip with the second highest penalty costs, etc. This continues until no further reduction in penalty costs can be obtained.

7 Computational Results

The computational experiments are done on two cases containing multiple problem instances. The first case, as described in the introduction, is provided by one of ORTEC's clients. This case consists of several large real-world instances with many different aspects which have to be taken into account. The second case is purely theoretical, and is created from the well-known Solomon benchmark instances, which are adapted to include the depot processing capacities. This also allows us to make a comparison of how our method performs on both real and synthetic data.

Aside from testing our integrated smoothing method on both cases, we also make a comparison with ORTEC's current departure smoothing method, i.e., the two-phase smoothing method. It is especially interesting to see how two completely different approaches compare both in terms of running time and solution quality.

7.1 Algorithm configuration

Both the integrated smoothing as well as our version of the two-phase approach are implemented using the Rust programming language. All results were obtained on a laptop running Windows 10 with an Intel i7-4800MQ processor (2.70GHz) using 4 cores and 16 GB RAM. For both the practical case and theoretical case, we used some base settings for the ALNS algorithm. Because there are many parameters which have to be configured, we split up the configuration into several categories.

General configuration

- We let each level of the ALNS algorithm, i.e., route, trip and customer level, be explored for 200 iterations before moving on to the next level.
- The total objective (Section 2.3) is computed as follows: *total distance* + 1000 · *number of vehicles used* + 5 · *total loading capacity exceeded*. This means that the cost for each used vehicle is 1000, which is selected in such a way that for each instance reducing the number of vehicles has a higher priority than minimizing the distance. Furthermore, the penalty cost per unit of loading capacity exceeded is 5.
- We look at time intervals of 5 minutes, which means a trip can only have a start time that is divisible by 5 minutes.
- A vehicle can only start loading at most $T^L = 6$ time intervals (30 minutes) before the start time.
- The slack between trips has to be at least $T^S = 3$ time intervals (15 minutes), i.e., if a vehicle comes back at the depot after a trip, there should be at least 15 minutes available for loading the vehicle before it starts its next trip.
- We allow at most 3 trips per route.

Weight adaption

- We set $\gamma = 0.5$, which means that the weight update (Section 4.7) depends equally on the previous value of the weight ω_i as well as the value of the score π_i.
- If a new best solution is found, the scores of the operators are increased by $\sigma_1 = 4$
- If the found solution is not a new best solution, but better than the current solution, the scores of the operators are increased by $\sigma_2 = 2$.
- If the solution is worse than the current one, but accepted, the scores of the operators are increased by $\sigma_3 = 1$.

Acceptance rule

- The start temperature β_{T_0} is equal to 1.05 times the objective value of the initial solution obtained by the construction heuristic described in Section 4.3.
- At the end of each iteration, the temperature decreases by a factor $c = 0.99975$.

Reconstruction phase

- Both the least-cost and regret-based reconstruction operator are available for selection in the reconstruction phase. In our results, we only use the 2-regret reconstruction operator.

Destruction phase

- All destruction operators as described in Chapter 4 and 6 are available to be selected at the start of the destruction phase.
- The number of customers, trips or routes to remove during the destruction phase, i.e. n_1, n_2 or n_3, is selected randomly. At the beginning of the destruction phase, we uniformly select a percentage between 0 and 35% of the customers, trips or routes to be removed.

Local search phase

- Both the random best and high savings swap operators as described in Section 5.2.1 are available to be selected at the start of the local search phase.
- In each local search phase, we perform 10 swap moves.

Smoothing phase

- All six smoothing operators that were described in Section 6.1.1 are available to be selected in the smoothing phase.
- In each smoothing phase, we smooth 10 trips. In case there are less than 10 trips in the solution, the operator simply smooths all trips.

7.2 Practical case

The practical case comes from one of ORTEC's clients and was already described briefly in the introduction. This case consists of real-world problem instances.

7.2.1 Problem instances

The problem instances we consider for the practical case are based on five base data sets. Each of these data sets contains a list of stores with their corresponding location, the travel times and distances between the locations, the delivery time windows and the type of store. We consider two types of stores, i.e., a regular store and a special store. There are two types of vehicles, both regular and special vehicles (which have some extra capabilities), where both types have a capacity of 30 units. The regular store can be served by either type of vehicle and the special store can only be served by a special vehicle. In all 5 data sets, there are 80 regular vehicles and 40 special vehicles available per depot. Furthermore, each data set contains two depots, from which all stores can be served. Table 1 shows the number of stores and the percentage of special stores in each data set.

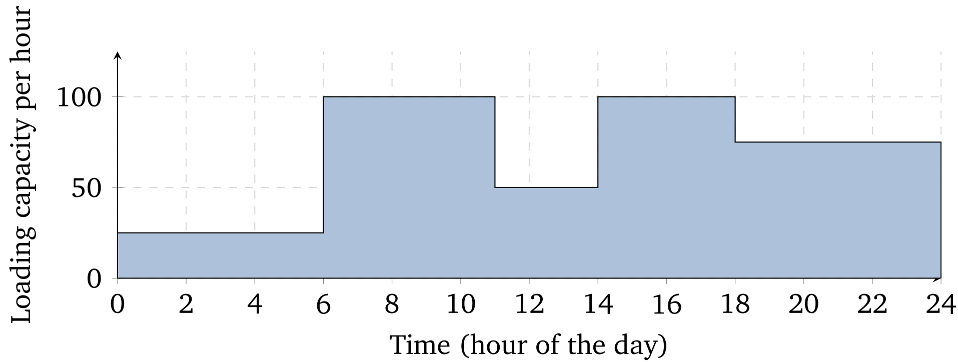
Table 1: The five base data sets

data set	PC1	PC2	PC3	PC4	PC5	average
# stores	687	561	502	606	578	586.8
% sp. stores	25.3	26.2	32.1	26.4	22.0	26.2

Note that these base data sets do not yet contain any processing capacities for the two depots. Because it is difficult for a company to provide an estimation of the processing capacity, we will be adding the processing capacities ourselves by considering multiple scenarios. We look at both time-varying capacities (TVC) and constant capacities (CC). Furthermore, we look at scenarios with high (H), medium (M) or low (L) processing capacities. The processing capacities are the same for both depots in each scenario. By looking at all possible combinations, we get six different scenarios and 30 different instances. Each instance is denoted by combining the data set name with the scenario, e.g., ***PC1_TVC_H*** is the first data set combined with high time-varying capacities.

For the time-varying capacity scenarios, we have decided to only make the loading capacity time-varying and let the number of loading docks remain constant during the day. Figure 5 shows how the loading capacity looks like during the day. The figure clearly shows that the capacity is at its lowest during the night. Furthermore, around noon (11:00-14:00), some workers might have a break, which also lowers the available loading capacity.

Figure 5: Time-varying loading capacity



Note that this figure is only meant to show how the loading capacity changes throughout the day. It does not show the actual loading capacity per hour. The actual loading capacity depends on whether we consider the high, medium or low capacity scenario, however, the ratio stays the same.

7.2.2 Constant processing capacity

We start by looking at the results for the constant processing capacity scenarios. These results are divided in two parts. First we look at the results when running the integrated and two-phase smoothing methods using just one process. Afterwards, we show the results when running the algorithms using parallel computing as described in Section 5.3. This allows us to show the effect of using parallelization on the solutions. Finally, in this section, we only show aggregated results where we take the average objective value over multiple instances. For the results of each separate instance, we refer to Appendix B.

For all results in this section, we run the ALNS algorithm for 5,000 iterations in total.

Sequential

We will first show the results when running the integrated smoothing method using just one process. Table 2 shows the average objective values for the integrated smoothing method for the high, medium and low constant capacity scenarios. This means that, for example, the results for the high capacity scenario show the average values of the 5 instances PCX_CC_H with $X = 1, \ldots, 5$.

Table 2: Average objective values for the three different constant capacity scenarios for the integrated smoothing method

scenario	obj. val	distance	vehicles	penalty	running time
high	66,506.54	26,892.54	39.60	2.80	239.59 sec
medium	66,112.06	26,060.06	40.00	10.4	254.57 sec
low	66,066.87	26,063.87	39.60	40.6	262.16 sec
average	66,228.49	26,338.82	39.73	17.93	252.11 sec

As expected, the average penalty is the highest in the low capacity scenarios and the lowest in the high capacity scenario. We also see that the average running time slightly increases when the capacity gets lower, i.e., the instance becomes more difficult.

In Table 3, the results of ORTEC's two-phase method are given for the high, medium and low constant capacity scenarios.

Table 3: Average objective values for the three different constant capacity scenarios for the two-phase smoothing method

scenario	obj. val	distance	vehicles	penalty	running time
high	67,756.61	27,756.61	40.00	0.00	312.24 sec
medium	67,871.73	28,448.73	39.40	4.60	324.68 sec
low	68,109.39	27,850.39	40.00	51.80	360.00 sec
average	67,912.58	28,018.58	39.80	18.80	332.31 sec

If we compare this to the integrated smoothing results of Table 2, we see that integrated smoothing gives solutions with a lower objective value in less time. Especially the average distance is lower in all three scenarios. The two-phase method is however able to obtain a lower average penalty on the high and medium capacity scenarios. Only for the low capacity scenario does integrated smoothing outperform ORTEC's two-phase method in finding a lower penalty cost. This means that for this case, the integrated smoothing approach works better on more difficult instances, where it is harder to find a solution that satisfies the processing capacity constraints.

It is also interesting to see how much time is spent in the first and in the second phase of the two-phase smoothing method. Table 4 shows the running time of both phases for the high, medium and low processing capacity scenarios.

Table 4: Average running time of the first and second phase for the three constant capacity scenarios for the two-phase smoothing method

scenario	phase 1	phase 2	total
high	293.52 sec	18.72 sec	312.24 sec
medium	299.88 sec	24.80 sec	324.68 sec
low	322.88 sec	37.12 sec	360.00 sec

The running time clearly increases when the processing capacity is lower. An instance with lower processing capacity leads to longer running times in the first phase because it is harder to satisfy the smoothability constraint. In the second phase, the running time increases because we have to reduce a higher penalty and potentially have to smooth more trips. Furthermore, the percentage of time spent in the second phase relative to the total running time also increases. In the high capacity scenario, around 6% of the total running time is spent in the second phase. For the low capacity scenario, this increases to around 10%.

Finally, Table 5 shows the results for the same instances when running sequential ALNS without any departure smoothing.

Table 5: Results for for the high, medium and low constant capacity scenarios without departure smoothing

				high		medium		low	
instance	**distance**	**# vehicles**	**running time**	**obj. val**	**penalty**	**obj. val**	**penalty**	**obj. val**	**penalty**
PC1_CC	35,502.55	45	234.46 sec	86,002.55	1,100	86,702.55	1,240	87,817.55	1,463
PC2_CC	27,798.40	36	141.11 sec	67,883.40	817	68,658.40	972	69,558.40	1,152
PC3_CC	23,217.57	36	106.94 sec	63,787.57	914	64,102.57	977	64,862.57	1,129
PC4_CC	28,360.70	42	173.06 sec	75,320.70	992	75,565.70	1,041	76,810.70	1,290
PC5_CC	22,993.93	35	160.43 sec	61,843.93	770	62,098.93	821	62,943.93	990
average	27,574.63	38.80	163.20 sec	70,967.63	918.60	71,425.63	1,010.20	72,398.63	1,204.80

As expected, the running time is lower if we do not apply departure smoothing. Furthermore, we also see that the distance and total number of vehicles is lower, because the ALNS algorithm without departure smoothing does not have to take the depot processing capacities into account. However, if we look at the total objective and the penalty costs, we see that both the integrated smoothing method as well as ORTEC's two-phase method are able to obtain much better solutions.

Parallel

The following results were obtained by running the ALNS algorithm with four processes in parallel as described in Section 5.3. The synchronization step is after every 600 iterations, i.e., each time the algorithm changes from customer level back to route level again. Each process is configured in the same way using the base configuration described in Section 7.1. Table 6 shows the average objective values for the high, medium and low constant processing capacity scenarios for the integrated smoothing method.

Table 6: Average objective values for the three different constant capacity scenarios for the integrated smoothing method with four processes in parallel

scenario	**obj. val**	**distance**	**vehicles**	**penalty**	**running time**
high	65,069.64	26,669.64	38.40	0.00	328.62 sec
medium	64,373.86	25,915.68	38.40	11.60	338.55 sec
low	64,813.33	25,667.33	39.00	29.20	374.53 sec
average	64,752.28	26,084.22	38.60	13.60	347.23 sec

Compared to the results for the sequential integrated smoothing method, we see that the average objective value for all three scenarios is considerably lower. This shows the benefit of using parallel computing, which allows us to explore a much larger part of the solution space. This is also illustrated by Figure 6 which shows the best found solution at the end of each ALNS iteration for both the parallel and sequential integrated smoothing method. This shows that when using multiple processes in parallel, we are able to find solutions with a lower objective value in less iterations.

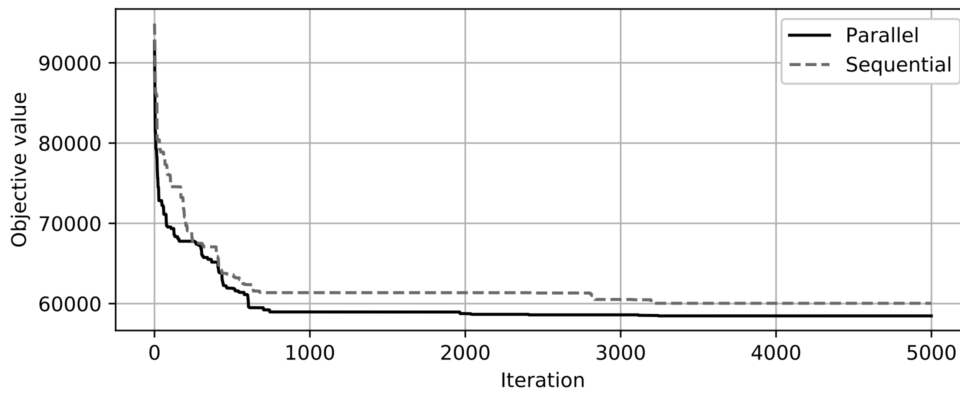


Figure 6: Parallel versus sequential integrated smoothing for the PC3_CC_H instance

Just as for the sequential case, we again make a comparison with ORTEC's two-phase method. Table 7 shows the average objective values for the high, medium and low constant processing capacity scenarios for ORTEC's two-phase method when using four processes in parallel.

Table 7: Average objective values for the three different constant capacity scenarios for the two-phase smoothing method with four processes in parallel

scenario	**obj. val**	**distance**	**vehicles**	**penalty**	**running time**
high	66,552.90	27,352.90	39.20	0.00	434.67 sec
medium	66,373.48	27,166.48	39.20	1.40	451.22 sec
low	66,799.57	27,631.57	39.00	33.60	512.42 sec
average	66,575.32	27,383.65	39.13	11.67	466.10 sec

As expected, we again see that the average objective value is lower for all three scenarios compared to the sequential two-phase smoothing results. Compared to the parallel integrated smoothing results in Table 6, we get a similar conclusion as for the sequential results. The two-phase smoothing method obtains a slightly lower penalty on average, at the cost of a higher distance and number of vehicles. However, when we compare the total objective value, we see that the integrated smoothing method clearly works better on average.

Finally, we also look at the results when applying ALNS in parallel without departure smoothing, which are shown in Table 8. These results provide us with the same conclusion as in the sequential case, with the only difference being that the distance and total number of vehicles is lower than in the sequential case.

Table 8: Results for for the high, medium and low constant capacity scenarios without departure smoothing

				high		medium		low	
instance	**distance**	**# vehicles**	**running time**	**obj. val**	**penalty**	**obj. val**	**penalty**	**obj. val**	**penalty**
PC1_CC	32,085.58	45	326.30 sec	82,825.58	1,148	83,290.58	1,241	84,300.58	1,443
PC2_CC	26,347.12	36	187.72 sec	66,562.12	843	66,812.12	893	67,832.12	1,097
PC3_CC	21,947.87	36	152.25 sec	61,907.87	792	62,292.87	869	63,637.87	1,138
PC4_CC	26,420.30	41	238.20 sec	71,955.30	907	72,810.30	1,078	73,910.30	1,298
PC5_CC	21,569.44	35	317.27 sec	60,644.44	815	60,979.44	882	61,519.44	990
average	25,674.06	38.60	244.35 sec	68,779.06	901.00	69,237.06	992.60	70,240.06	1,193.20

7.2.3 Time-varying processing capacity

In the same way as for the constant processing capacity results, we also divide the results for the time-varying processing capacity instances into two parts. We first look at the sequential results and afterwards we again make the comparison with running the algorithm using multiple processes in parallel.

Sequential

We first show the results when running the algorithm using just one process. Table 9 shows the average objective values for the integrated smoothing method for the high, medium and low time-varying capacity scenarios. Again, this means that, e.g., the results for the high capacity scenario show the average values of the 5 instances PCX_TVC_H with $X = 1, \ldots, 5$.

Table 9: *Average objective values for the three different time-varying capacity scenarios for the integrated smoothing method*

scenario	**obj. val**	**distance**	**vehicles**	**penalty**	**running time**
high	66,639.65	26,659.65	39.80	36.00	244.88 sec
medium	66,282.31	26,301.31	39.80	36.20	243.81 sec
low	67,604.57	27,226.57	39.20	235.60	268.87 sec
average	66,842.18	26,729.18	39.60	102.60	252.52 sec

Compared to the results for the instances with constant processing capacities shown in Table 2, we see that making the capacities time-varying does not impact the running times when using integrated smoothing. In both cases, we have almost the same average running time.

If we compare the integrated smoothing results to the results of ORTEC's two-phase smoothing method, shown in Table 10, we see that the two-phase method is better at finding solutions with a lower penalty cost. However, these solutions have a much higher total distance and therefore a higher objective value. Furthermore, the two-phase method again has a higher average running time than the integrated smoothing method.

Table 10: *Average objective values for the three different time-varying capacity scenarios for the two-phase smoothing method*

scenario	**obj. val**	**distance**	**vehicles**	**penalty**	**running time**
high	67,030.67	27,170.67	39.80	12.00	340.80 sec
medium	68,201.71	28,273.21	39.80	25.70	379.60 sec
low	68,601.10	27,972.10	40.20	85.80	447.54 sec
average	67,944.49	27,805.33	39.93	41.17	389.31 sec

Finally, we see that the running time needed for the two-phase smoothing method is higher in the time-varying capacity case. This has to do with the fact that these instances have a lower average processing capacity than in the constant capacity case, which increases the running time as was shown in Table 4.

Parallel

The following results were obtained by running the ALNS algorithm with four processes in parallel as described in Section 5.3. There is a synchronization step every 600 iterations, and we use the same base configuration for each process, as described in Section 7.1. Table 11 shows the average objective values for the parallel integrated smoothing method for the high, medium and low time-varying capacity scenarios.

Table 11: Average objective values for the three different time-varying capacity scenarios for the integrated smoothing method with four processes in parallel

scenario	obj. val	distance	vehicles	penalty	running time
high	63,905.57	25,658.57	38.20	9.40	352.01 sec
medium	64,712.30	25,474.80	39.00	47.50	355.94 sec
low	65,640.77	25,543.77	39.40	139.40	382.97 sec
average	64,752.88	25,559.05	38.87	65.43	363.64 sec

If we compare these results to the sequential results shown in Table 9, we again see that the average objective value is lower for all three scenarios. Especially the average penalty cost for the low capacity scenario has decreased significantly. On the other hand, the average penalty cost for the medium capacity scenario has increased compared to the sequential case. This can happen because the objective consists of multiple components. In this case, the algorithm found solutions with a lower distance and number of vehicles, at the cost of a slightly higher penalty for exceeding the loading capacity.

Table 12 shows the results for the two-phase method. First of all, the average penalty obtained by parallel integrated smoothing is now closer to the average penalty obtained with the two-phase method.

Table 12: Average objective values for the three different time-varying capacity scenarios for the two-phase smoothing method with four processes in parallel

scenario	obj. val	distance	vehicles	penalty	running time
high	66,483.14	27,190.14	39.20	18.60	447.87 sec
medium	66,174.59	27,295.09	38.80	15.90	489.28 sec
low	67,455.18	27,347.18	39.60	101.60	592.87 sec
average	66,704.30	27,277.47	39.20	45.37	510.01 sec

Compared to the sequential two-phase results, we see that even though the average objective value and distance are lower, the average penalty costs is slightly higher. This can be explained by noticing that only the first phase runs in parallel. In this phase, we mostly focus on reducing the distance and number of vehicles. Finding a solution with a lower distance and number of vehicles does not always have to result in a lower penalty in the end, after the second phase has run.

For completeness, the results for the regular ALNS algorithm without departure smoothing can be found in Appendix B. These results provided the same conclusions as for the constant processing capacity case.

7.3 Theoretical case

Next to the results from the practical case, we also want to test our method using theoretical benchmark instances and see if we obtain similar conclusions.

7.3.1 Problem instances

The problem instances for the theoretical case are obtained by modifying the Solomon benchmark instances, which are often used to test the performance of algorithms for the *vehicle routing problem with time windows* (VRPTW). The Solomon benchmark instances consist of three different types of instances, namely clustered customers (C-type), uniformly distributed customers (R-type) and a mix between clustered and uniformly distributed customers (RC-type). Furthermore, there are instances with either 25, 50 or 100 customers available.

For the theoretical case, we selected six different instances, namely: C201.050, C201.100, R201.050, R201.100, RC201.050 and RC201.100. These instances are modified in two different ways. We first reduce the capacity of each vehicle in each instance to 50 units, which gives us solutions with more trips. Secondly, we add the processing capacities. Because it is difficult to assign realistic time-varying capacities to the Solomon benchmark instances, we only look at constant processing capacities. This is done in the same way as for the practical case, i.e., we look at three scenarios: high, medium and low processing capacity. The processing capacities corresponding to each scenario is shown in Table 13.

Table 13: Three different scenarios for the processing capacity for the theoretical instances

scenario	loading capacity	# loading docks
high	3 unit per minute	5
medium	2 units per minute	5
low	1 units per minute	5

7.3.2 Results

For the theoretical case, we only consider the algorithm with four processes running in parallel. The synchronization step is after every 600 iterations, and we again use the same base configuration (Section 7.1) for each process. We let the algorithm run for 50,000 iterations in total.

Integrated smoothing

We start by looking at results of the integrated smoothing method on all of the instances, which are shown in Table 14. For each of the three capacity scenarios, we show the average objective value, distance, number of vehicles, penalty costs and running time.

Table 14: Integrated smoothing on the theoretical instances for the high, medium and low capacity scenarios

instance	obj. val	distance	vehicles	penalty	running time
C201_050_H	7,184.29	1,184.29	6	0	49.45 sec
C201_100_H	15,642.14	2,642.14	13	0	119.41 sec
R201_050_H	6,470.97	1,410.97	5	12	29.52 sec
R201_100_H	13,208.02	2,208.02	11	0	89.15 sec
RC201_050_H	8,874.87	1,874.87	7	0	30.18 sec
RC201_100_H	15,140.77	3,140.77	12	0	91.28 sec
average	11,086.84	2,076.84	9.00	2.00	68.17 sec
C201_050_M	7,204.69	1,204.69	6	0	52.38 sec
C201_100_M	15,640.74	2,640.74	13	0	116.74 sec
R201_050_M	6,616.13	1,441.13	5	35	40.83 sec
R201_100_M	13,350.80	2,345.80	11	1	87.43 sec
RC201_050_M	8,952.02	1,952.02	7	0	42.54 sec
RC201_100_M	15,458.93	3,258.93	12	40	68.61 sec
average	11,203.89	2,140.55	9.00	12.67	68.09 sec
C201_050_L	7,191.16	1,191.16	6	0	54.52 sec
C201_100_L	15,633.59	2,633.59	13	0	131.93 sec
R201_050_L	6,807.05	1,462.05	5	69	40.49 sec
R201_100_L	13,877.95	2,387.95	11	98	77.29 sec
RC201_050_L	9,002.93	2,002.93	7	0	44.4 sec
RC201_100_L	18,011.02	3,606.02	12	481	54.59 sec
average	11,753.95	2,213.95	9.00	108.00	67.20 sec

For the C-type instances, integrated smoothing was able to find a solution that satisfied the processing capacities without any penalty costs in all scenarios. This is not the case for the R- and RC-type instances, where especially the RC201_100_L instance has a large penalty cost remaining. Finally, the average running time is almost the same for the three scenarios.

Two-phase smoothing

Table 15 shows the results of the two-phase method on the theoretical instances. Again, for each capacity scenario we give the average objective value, distance, number of vehicles, penalty and running time.

Table 15: Two-phase smoothing on the theoretical instances for the high, medium and low capacity scenarios

instance	**obj. val**	**distance**	**vehicles**	**penalty**	**running time**
C201_050_H	7,197.31	1,197.31	6	0	39.15 sec
C201_100_H	15,646.88	2,646.88	13	0	139.94 sec
R201_050_H	6,584.21	1,549.21	5	7	38.04 sec
R201_100_H	13,475.91	2,425.91	11	10	118.18 sec
RC201_050_H	8,908.15	1,883.15	7	5	44.19 sec
RC201_100_H	16,513.95	3,888.95	12	125	157.15 sec
average	11,387.74	2,265.24	9.00	24.50	89.44 sec
C201_050_M	7,200.49	1,200.49	6	0	42.24 sec
C201_100_M	15,672.20	2,672.20	13	0	146.76 sec
R201_050_M	6,608.53	1,456.03	5	30.5	40.81 sec
R201_100_M	14,065.41	2,460.41	11	121	161.03 sec
RC201_050_M	9,255.83	1,905.83	7	70	47.82 sec
RC201_100_M	17,107.03	3,494.53	13	122.5	267.48 sec
average	11,651.58	2,198.25	9.17	57.33	117.69 sec
C201_050_L	7,247.57	1,197.57	6	10	43.93 sec
C201_100_L	15,693.01	2,643.01	13	10	157.83 sec
R201_050_L	6,905.29	1,630.29	5	55	46.72 sec
R201_100_L	15,377.29	2,482.29	12	179	387.94 sec
RC201_050_L	9,575.42	1,925.42	7	130	53.16 sec
RC201_100_L	19,587.56	3,742.56	15	169	637.82 sec
average	12,397.69	2,270.19	9.67	92.17	221.23 sec

The average running time for the two-phase method increases when the capacity is lower, which was also the case for the practical instances. This mostly had to do with the smoothability constraint in the first phase, which is more difficult to satisfy with a lower capacity. Furthermore, compared to the integrated smoothing method, we see that the two-phase method performs a lot worse for the high and medium capacity scenarios. In both scenarios, the two-phase method has a higher average distance as well as much higher average penalty costs.

Table 16: Results for the theoretical instances for the high, medium and low capacity scenarios without departure smoothing

				high		medium		low	
instance	**distance**	**# vehicles**	**running time**	**obj. val**	**penalty**	**obj. val**	**penalty**	**obj. val**	**penalty**
C201_050	1,171.62	6	12.25 sec	8,271.62	220	9,221.62	410	10,271.62	620
C201_100	2,587.74	13	25.72 sec	19,412.74	765	19,737.74	830	21,862.74	1,255
R201_050	1,275.53	5	11.68 sec	7,570.53	259	8,330.53	411	9,105.53	566
R201_100	2,135.38	10	26.12 sec	14,855.38	544	16,480.38	869	17,825.38	1,138
RC201_050	1,753.82	7	14.37 sec	10,553.82	360	11,353.82	520	12,603.82	770
RC201_100	2,996.17	12	23.61 sec	19,046.17	810	20,681.17	1,137	21,966.17	1,394
average	1,986.71	8.83	18.96 sec	13,285.04	493.00	14,300.88	696.17	15,605.88	957.17

Finally, Table 16 shows the results of applying regular ALNS without departure smoothing. As expected, applying regular ALNS without departure smoothing gives us solutions with a lower distance and lower number of vehicles in less time. However, because the processing capacities at the depot are ignored, we get enormous penalty costs for all instances. Even in the high capacity scenario, we exceed the loading capacity by almost 500 units on average. This shows that it is worthwhile to consider using a departure smoothing method in order to reduce this penalty.

8 Conclusion

In this thesis, we looked at a rich vehicle routing problem with many different attributes. One of these attributes is the limited processing capacity at the depot, which is rarely taken into account in the literature. As mentioned in the introduction, ignoring the processing capacity at the depot can lead to congestion, which makes vehicles start their trips with a delay. This delay might mean that certain customers cannot be served within their respective time window anymore, which can be costly. Furthermore, any successor trips of the vehicle can also be affected by the delay, which makes the problem even worse. As a solution, we developed a method that shifts the start times of the trips, which we call departure smoothing, such that we can satisfy the processing capacity constraints.

Recently, ORTEC has developed their own departure smoothing method. This method uses a two-phase approach in which the trips are planned first and smoothed afterwards. Our method uses a different approach in which we combine the two phases into one single phase, in which we both plan and smooth trips. This was done by extending the adaptive large neighborhood search (ALNS) algorithm by Azi et al. (2014) to include a new smoothing phase. We also made some further modifications to the ALNS algorithm to support the other attributes of our problem, such as multiple time windows, a heterogeneous fleet and multiple depots. Finally, we added a local search phase and used parallelization in order to improve the solution quality.

The computational experiments showed that the integrated smoothing method performed well on both the real-world instances as well as the theoretical instances. For almost all instances, integrated smoothing performed better than ORTEC's two-phase approach. Both the running time as well as the objective value was lower for the solutions obtained with the integrated smoothing method. Finally, we also showed the effect of using multiple processes in parallel within the adaptive large neighborhood search algorithm and compared this to the regular sequential approach of using only one process. The parallel method was able to find solutions with a lower objective value in less iterations by being able to explore a larger part of the solution space.

Even though the integrated smoothing method as described in this thesis performs well already, there are still some interesting ideas and extensions left which can be investigated in future research. First of all, we have extended the ALNS algorithm by Azi et al. (2014) to be able to handle several additional attributes, such as multiple time windows, heterogeneous fleet, etc. However, in order to improve the solution quality, it might be interesting to introduce several new reconstruction and destruction operators which specifically take these attributes into account. Secondly, we have mentioned in Section 5.3.2 the possibility of having different process configurations when running the algorithm in parallel. It could be interesting to investigate in future research what kind of combination of configurations works well. Finally, it would be beneficial to have an exact solution method for the vehicle routing problem with depot processing capacities, instead of a heuristic. This could then be used to solve small instances exactly which can be used as a benchmark.

References

Azi, N., Gendreau, M., and Potvin, J.-Y. (2010). An exact algorithm for a vehicle routing problem with time windows and multiple use of vehicles. *European Journal of Operational Research*, 202(3):756–763.

Azi, N., Gendreau, M., and Potvin, J.-Y. (2014). An adaptive large neighborhood search for a vehicle routing problem with multiple routes. *Computers & Operations Research*, 41:167–173.

Beck, J. C., Prosser, P., and Selensky, E. (2003). Vehicle routing and job shop scheduling: what's the difference? In *ICAPS*, pages 267–276.

Belhaiza, S., Hansen, P., and Laporte, G. (2014). A hybrid variable neighborhood tabu search heuristic for the vehicle routing problem with multiple time windows. *Computers & Operations Research*, 52:269–281.

Brandao, J. and Mercer, A. (1997). A tabu search algorithm for the multi-trip vehicle routing and scheduling problem. *European journal of operational research*, 100(1):180–191.

Brandao, J. and Mercer, A. (1998). The multi-trip vehicle routing problem. *Journal of the Operational research society*, pages 799–805.

Caceres-Cruz, J., Arias, P., Guimarans, D., Riera, D., and Juan, A. A. (2015). Rich vehicle routing problem: Survey. *ACM Computing Surveys (CSUR)*, 47(2):32.

Cattaruzza, D., Absi, N., and Feillet, D. (2016). The multi-trip vehicle routing problem with time windows and release dates. *Transportation Science*, 50(2):676–693.

Cattaruzza, D., Absi, N., Feillet, D., and Vidal, T. (2014). A memetic algorithm for the multi trip vehicle routing problem. *European Journal of Operational Research*, 236(3):833–848.

Dabia, S., Ropke, S., and van Woensel, T. (2014). An exact algorithm for the vehicle routing problem with time windows and shifts.

Drexl, M. (2012). Synchronization in vehicle routing—a survey of vrps with multiple synchronization constraints. *Transportation Science*, 46(3):297–316.

Ebben, M. J., Van der Heijden, M., and van Harten, A. (2005). Dynamic transport scheduling under multiple resource constraints. *European Journal of Operational Research*, 167(2):320–335.

Favaretto, D., Moretti, E., and Pellegrini, P. (2007). Ant colony system for a vrp with multiple time windows and multiple visits. *Journal of Interdisciplinary Mathematics*, 10(2):263–284.

Gromicho, J., van Hoorn, J., Kok, A. L., and Schutten, J. (2012). Vehicle routing with restricted loading capacities. Technical report, Beta Working Paper, Eindhoven University of Technology (804).

Hempsch, C. and Irnich, S. (2008). Vehicle routing problems with inter-tour resource constraints. In *The Vehicle Routing Problem: Latest Advances and New Challenges*, pages 421–444. Springer.

Hernandez, F., Feillet, D., Giroudeau, R., and Naud, O. (2014). A new exact algorithm to solve the multi-trip vehicle routing problem with time windows and limited duration. *4OR*, 12(3):235–259.

Hernandez, F., Feillet, D., Giroudeau, R., and Naud, O. (2016). Branch-and-price algorithms for the solution of the multi-trip vehicle routing problem with time windows. *European Journal of Operational Research*, 249(2):551–559.

Kirkpatrick, S., Gelatt, C. D., Vecchi, M. P., et al. (1983). Optimization by simulated annealing. *science*, 220(4598):671–680.

Montoya-Torres, J. R., Franco, J. L., Isaza, S. N., Jiménez, H. F., and Herazo-Padilla, N. (2015). A literature review on the vehicle routing problem with multiple depots. *Computers & Industrial Engineering*, 79:115–129.

Petch, R. J. and Salhi, S. (2003). A multi-phase constructive heuristic for the vehicle routing problem with multiple trips. *Discrete Applied Mathematics*, 133(1):69–92.

Salhi, S. and Petch, R. (2007). A ga based heuristic for the vehicle routing problem with multiple trips. *Journal of Mathematical Modelling and Algorithms*, 6(4):591–613.

Schrimpf, G., Schneider, J., Stamm-Wilbrandt, H., and Dueck, G. (2000). Record breaking optimization results using the ruin and recreate principle. *Journal of Computational Physics*, 159(2):139–171.

Şen, A. and Bülbül, K. (2008). A survey on multi trip vehicle routing problem.

Taillard, É. D., Laporte, G., and Gendreau, M. (1996). Vehicle routeing with multiple use of vehicles. *Journal of the Operational research society*, pages 1065–1070.

Thangiah, S. R. and Salhi, S. (2001). Genetic clustering: an adaptive heuristic for the multidepot vehicle routing problem. *Applied Artificial Intelligence*, 15(4):361–383.

Vidal, T., Crainic, T. G., Gendreau, M., and Prins, C. (2013). Heuristics for multi-attribute vehicle routing problems: A survey and synthesis. *European Journal of Operational Research*, 231(1):1–21.

Yücenur, G. N. and Demirel, N. Ç. (2011). A new geometric shape-based genetic clustering algorithm for the multi-depot vehicle routing problem. *Expert Systems with Applications*, 38(9):11859–11865.

A List of Symbols

For completeness, a comprehensive overview of all used symbols and notation is given here. The overview is divided into several categories, which makes it easier to look up a symbol when the context is known.

Network Graph

N directed graph representing the network.

A set of all arcs in the network graph.

C set of all customers in the network.

D set of all depots in the network.

V set of all vertices in the network graph, i.e., $V = C \cup D$.

d_{ij} travel distance for arc $(i, j) \in A$ with $i, j \in V$.

t_{ij} travel time for arc $(i, j) \in A$ with $i, j \in V$.

Vehicle

K set of all vehicles.

K_d set of vehicles available at depot d.

W set of vehicle types.

c_w capacity of vehicle type $w \in W$.

w_k type of vehicle $k \in K$.

c_k capacity of vehicle $k \in K$.

d_k depot of vehicle $k \in K$.

R_k route of vehicle $k \in K$.

S set of all trips.

s_k^j jth trip in the route R_k of vehicle $k \in K$.

n_k^s total number of trips done by vehicle $k \in K$, i.e., $n_k^s = |R_k|$.

τ_s start time of trip s.

σ_s time needed to load the vehicle used for trip s.

Customer

C set of all customers.

C_s set of all customers on trip $s \in S$.

W_i set of all vehicle types that can serve customer $i \in C$.

q_i demand quantity of customer $i \in C$.

t_i^s service time of customer $i \in C$.

TW_i set of time windows of customer $i \in C$.

n_i^{tw} total number of time windows of customer $i \in C$, i.e. $n_i^{tw} = |TW_i|$.

τ_i start time of the service at customer $i \in C$.

Depot

D set of all depots.

T set of time intervals in which processing capacity is constant.

T^L maximum number of intervals a vehicle can start loading before the start time

T^S minimum number of intervals of slack between two successive trips

l_{dt} number of loading docks available during time interval $t \in T$ at depot $d \in D$.

p_{dt} loading capacity available during time interval $t \in T$ at depot $d \in D$.

p_{dt}^* used loading capacity during time interval $t \in T$ at depot $d \in D$.

α_{dt} distribution function of capacity in time interval $t \in T$ of depot $d \in D$

Adaptive Large Neighborhood Search

ω_i weight of destruction or reconstruction operator i.

γ parameter that influences how the weights get updated.

π_i score which measures the performance of an operator during a level.

σ_1 score increment when the operator is used during an iteration in which a new best solution is found.

σ_2 score increment when the operator is used during an iteration in which a solution is found that is better than the current, but not the new best.

σ_3 score increment when the operator is used during an iteration in which a worse than the current, but accepted, solution is found.

β_T temperature parameter for the simulated annealing acceptance rule.

B Additional results

B.1 Constant processing capacity

Sequential

Table 17: Sequential integrated smoothing results for the constant capacity instances

instance	obj. val	distance	# vehicles	penalty	running time (s)
PC1_CC_H	81,225.03	33,155.03	48	14	362.19
PC1_CC_M	81,202.90	31,202.90	50	0	373.22
PC1_CC_L	78,279.04	31,844.04	46	87	352.99
PC2_CC_H	63,726.68	27,726.68	36	0	210.46
PC2_CC_M	65,437.64	28,287.64	37	30	249.00
PC2_CC_L	62,793.96	26,633.96	36	32	234.49
PC3_CC_H	60,041.52	23,041.52	37	0	150.00
PC3_CC_M	58,795.55	21,750.55	37	9	145.60
PC3_CC_L	59,800.26	21,550.26	38	50	173.14
PC4_CC_H	70,572.82	28,572.82	42	0	226.10
PC4_CC_M	68,051.16	26,036.16	42	3	275.10
PC4_CC_L	70,913.20	27,803.20	42	22	282.36
PC5_CC_H	56,966.67	21,966.67	35	0	249.19
PC5_CC_M	57,073.07	23,023.07	34	10	229.91
PC5_CC_L	58,547.88	22,487.88	36	12	267.83

Table 18: Sequential two-phase smoothing results for the constant capacity instances

instance	obj. val	distance	# vehicles	penalty	running time (s)
PC1_CC_H	81,763.27	34,763.27	47	0	442.22
PC1_CC_M	81,947.43	34,927.43	47	4	455.04
PC1_CC_L	83,216.12	33,716.12	49	100	505.66
PC2_CC_H	63,801.23	26,801.23	37	0	259.89
PC2_CC_M	65,947.35	29,917.35	36	6	293.61
PC2_CC_L	64,338.29	28,338.29	36	0	313.36
PC3_CC_H	61,775.15	23,775.15	38	0	223.45
PC3_CC_M	58,956.11	21,936.11	37	4	227.06
PC3_CC_L	61,219.62	23,739.62	37	96	263.12
PC4_CC_H	70,535.40	28,535.40	42	0	306.32
PC4_CC_M	71,242.49	29,197.49	42	9	344.03
PC4_CC_L	71,622.54	29,432.54	42	38	388.74
PC5_CC_H	60,907.98	24,907.98	36	0	329.30
PC5_CC_M	61,265.25	26,265.25	35	0	303.40
PC5_CC_L	60,150.39	24,025.39	36	25	329.20

Parallel

Table 19: Parallel integrated smoothing results for the constant capacity instances

instance	obj. val	distance	# vehicles	penalty	running time (s)
PC1_CC_H	77,575.64	32,575.64	45	0	520.33
PC1_CC_M	77,892.03	31,767.03	46	25	542.75
PC1_CC_L	77,758.44	31,498.44	46	52	580.56
PC2_CC_H	63,439.63	27,439.63	36	0	265.64
PC2_CC_M	61,278.69	26,278.69	35	0	265.04
PC2_CC_L	63,025.77	26,950.77	36	15	351.14
PC3_CC_H	58,470.59	22,470.59	36	0	199.80
PC3_CC_M	57,351.15	20,351.15	37	0	201.60
PC3_CC_L	58,303.20	21,248.20	37	11	249.41
PC4_CC_H	67,842.58	26,842.58	41	0	302.95
PC4_CC_M	67,415.68	27,330.68	40	17	345.61
PC4_CC_L	68,071.00	26,856.00	41	43	316.36
PC5_CC_H	58,019.76	24,019.76	34	0	354.37
PC5_CC_M	57,931.77	23,851.77	34	16	337.77
PC5_CC_L	56,908.23	21,783.23	35	25	375.17

Table 20: Parallel two-phase smoothing results for the constant capacity instances

instance	obj. val	distance	# vehicles	penalty	running time (s)
PC1_CC_H	79,577.75	33,577.75	46	0	613.50
PC1_CC_M	80,767.76	33,767.76	47	0	647.55
PC1_CC_L	81,605.77	34,485.77	47	24	720.02
PC2_CC_H	63,445.31	27,445.31	36	0	383.64
PC2_CC_M	63,015.20	27,015.20	36	0	391.95
PC2_CC_L	63,128.37	27,128.37	36	0	428.70
PC3_CC_H	59,747.45	22,747.45	37	0	303.87
PC3_CC_M	59,538.05	22,538.05	37	0	332.08
PC3_CC_L	60,305.87	23,960.87	36	69	374.74
PC4_CC_H	69,216.31	27,216.31	42	0	446.05
PC4_CC_M	70,176.77	29,141.77	41	7	470.49
PC4_CC_L	68,560.65	27,425.64	41	27	549.19
PC5_CC_H	60,777.68	25,777.67	35	0	426.29
PC5_CC_M	58,369.61	23,369.61	35	0	414.05
PC5_CC_L	60,397.19	25,157.19	35	48	489.45

B.2 Time-varying processing capacity

Sequential

Table 21: Sequential integrated smoothing results for the time-varying capacity instances

instance	obj. val	distance	# vehicles	penalty	running time (s)
PC1_TVC_H	80,670.13	32,430.13	48	48	361.33
PC1_TVC_M	80,251.10	33,048.60	47	40.5	412.35
PC1_TVC_L	83,382.39	34,342.39	47	408	367.84
PC2_TVC_H	65,344.48	28,114.48	37	46	203.18
PC2_TVC_M	63,109.93	26,079.93	37	6	199.22
PC2_TVC_L	63,209.56	26,574.56	36	127	221.39
PC3_TVC_H	59,872.44	22,822.44	37	10	149.41
PC3_TVC_M	60,040.69	22,933.19	37	21.5	145.13
PC3_TVC_L	61,796.74	23,676.74	37	224	187.51
PC4_TVC_H	68,385.26	26,215.26	42	34	255.1
PC4_TVC_M	70,225.71	26,760.71	43	93	210.39
PC4_TVC_L	69,549.11	27,149.11	41	280	306.27
PC5_TVC_H	58,925.92	23,715.92	35	42	255.38
PC5_TVC_M	57,784.11	22,684.11	35	20	251.94
PC5_TVC_L	60,085.06	24,390.06	35	139	261.32

Table 22: Sequential two-phase smoothing results for the time-varying capacity instances

instance	obj. val	distance	# vehicles	penalty	running time (s)
PC1_TVC_H	80,433.74	32,413.74	48	4	474.5
PC1_TVC_M	82,623.72	34,398.72	48	45	532.3
PC1_TVC_L	84,576.70	34,986.70	49	118	645.74
PC2_TVC_H	63,180.26	27,080.26	36	20	302.44
PC2_TVC_M	66,861.71	30,706.71	36	31	331.76
PC2_TVC_L	65,796.05	28,581.05	37	43	396.24
PC3_TVC_H	60,962.46	23,922.46	37	8	254.48
PC3_TVC_M	59,176.38	22,161.38	37	3	275.85
PC3_TVC_L	60,124.58	22,784.58	37	68	313.58
PC4_TVC_H	71,720.15	29,580.15	42	28	360.73
PC4_TVC_M	71,241.64	29,171.64	42	14	387.23
PC4_TVC_L	72,785.05	30,045.05	42	148	493.62
PC5_TVC_H	58,856.76	22,856.76	36	0	311.86
PC5_TVC_M	61,105.08	24,927.58	36	35.5	370.86
PC5_TVC_L	59,723.10	23,463.10	36	52	388.51

Table 23: Sequential ALNS results for the high, medium and low time-varying capacity scenarios without departure smoothing

				high		medium		low	
instance	**distance**	**# vehicles**	**running time**	**obj. val**	**penalty**	**obj. val**	**penalty**	**obj. val**	**penalty**
PC1_TVC	35,502.55	45	234.46 sec	86,587.55	1,217	88,035.05	1,506.50	89,692.55	1,838
PC2_TVC	27,798.40	36	141.11 sec	68,573.40	955	69,428.40	1,126	70,518.40	1,344
PC3_TVC	23,217.57	36	106.94 sec	63,977.57	952	65,917.57	1,340	66,502.57	1,457
PC4_TVC	28,360.70	42	173.06 sec	74,890.70	906	77,303.20	1,388.50	79,385.70	1,805
PC5_TVC	22,993.93	35	160.43 sec	61,943.93	790	63,681.43	1,137.50	64,933.93	1,388
average	27,574.63	38.80	163.20 sec	71,194.63	964.00	72,873.13	1,299.70	74,206.63	1,566.40

Table 24: Parallel integrated smoothing results for the time-varying capacity instances

instance	obj. val	distance	# vehicles	penalty	running time (s)
PC1_TVC_H	76,258.71	31,258.71	45	0	564.83
PC1_TVC_M	79,144.10	32,794.10	46	70	567.05
PC1_TVC_L	79,205.99	31,775.99	46	286	547.31
PC2_TVC_H	62,380.03	27,240.03	35	28	277.49
PC2_TVC_M	62,009.41	25,926.91	36	16.5	275.44
PC2_TVC_L	63,170.90	25,795.91	37	75	342.45
PC3_TVC_H	56,848.43	20,848.43	36	0	217.24
PC3_TVC_M	57,752.82	20,587.82	37	33	214.45
PC3_TVC_L	58,511.11	20,421.11	38	18	235.64
PC4_TVC_H	67,517.68	26,492.68	41	5	330.01
PC4_TVC_M	67,017.53	25,740.03	41	55.5	385.56
PC4_TVC_L	70,045.83	27,685.83	41	272	406.96
PC5_TVC_H	56,523.00	22,453.00	34	14	370.49
PC5_TVC_M	57,637.64	22,325.14	35	62.5	337.18
PC5_TVC_L	57,270.00	22,040.00	35	46	382.47

Table 25: Parallel two-phase smoothing results for the time-varying capacities instances

instance	obj. val	distance	# vehicles	penalty	running time (s)
PC1_TVC_H	79,694.55	32,694.55	47	0	615.97
PC1_TVC_M	80,350.95	34,240.95	46	22	674.76
PC1_TVC_L	83,488.42	34,458.42	48	206	854.05
PC2_TVC_H	65,172.20	29,112.20	36	12	408.33
PC2_TVC_M	62,850.89	26,713.39	36	27.5	422.53
PC2_TVC_L	63,708.69	27,373.69	36	67	533.75
PC3_TVC_H	59,474.15	23,384.15	36	18	332.21
PC3_TVC_M	59,265.87	23,225.87	36	8	370.08
PC3_TVC_L	60,623.83	23,408.83	37	43	428.59
PC4_TVC_H	69,580.51	27,265.51	42	63	472.53
PC4_TVC_M	69,615.79	28,555.79	41	12	531.07
PC4_TVC_L	71,654.66	29,254.66	42	80	648.19
PC5_TVC_H	58,494.31	23,494.31	35	0	410.29
PC5_TVC_M	58,789.46	23,739.47	35	10	447.95
PC5_TVC_L	57,800.32	22,240.32	35	112	499.78

Table 26: Parallel ALNS results for for the high, medium and low time-varying capacity scenarios without departure smoothing

				high		medium		low	
instance	**distance**	**# vehicles**	**running time**	**obj. val**	**penalty**	**obj. val**	**penalty**	**obj. val**	**penalty**
PC1_TVC	32,085.58	45	326.30 sec	83,120.58	1,207	84,398.08	1,462.50	86,790.58	1,941
PC2_TVC	26,347.12	36	187.72 sec	67,317.12	994	67,957.12	1,122	69,247.12	1,380
PC3_TVC	21,947.87	36	152.25 sec	63,137.87	1,038	63,717.87	1,154	65,627.87	1,536
PC4_TVC	26,420.30	41	238.20 sec	73,645.30	1,245	74,127.80	1,341.50	75,910.30	1,698
PC5_TVC	21,569.44	35	317.27 sec	60,294.44	745	61,434.44	973	62,919.44	1,270
average	25,674.06	38.60	244.35 sec	69,503.06	1,045.80	70,327.06	1,210.60	72,099.06	1,565.00