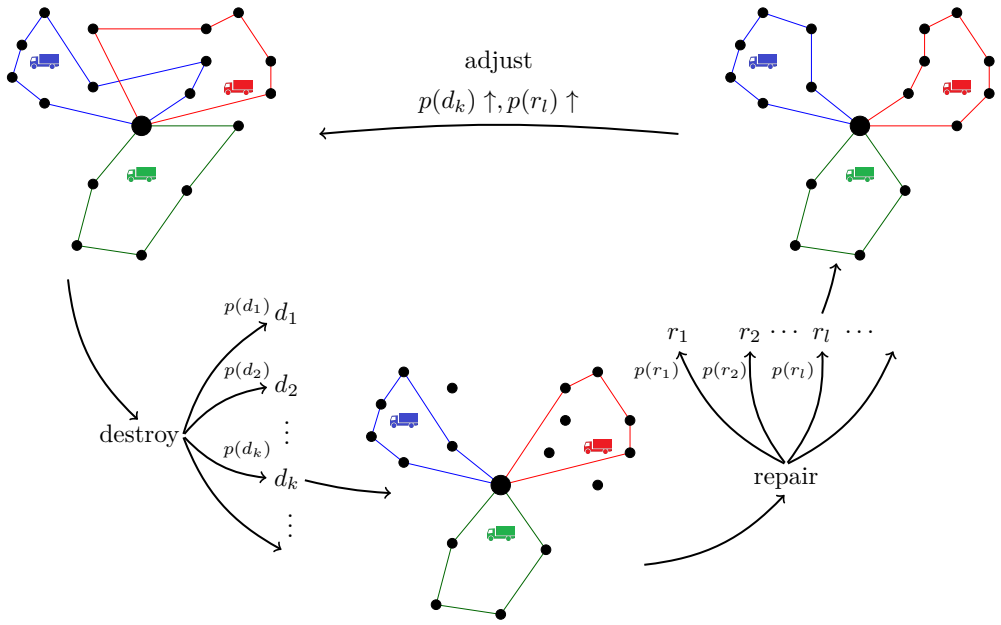
ulm university universität ulm

Ulm University | 89069 Ulm | Germany

Faculty of Engineering and Computer Science
Institute of Theoretical Computer Science

Adaptive Large Neighborhood Search

Bachelor thesis at Ulm University, 15.08.2014



Submitted by:
Roman Lutz
roman.lutz@uni-ulm.de

Reviewer:
Prof. Dr. Uwe Schöning

Supervisor:
Dipl.-Inf. Gunnar Völkel

ABSTRACT

There has already been a lot of research on Local Search Heuristics in Computer Science. Local Search improves an initial solution by manipulating rather small parts of the solution. Large Neighborhood Search (LNS) has a similar approach, but instead of marginal changes, huge parts of the solutions are changed. This is done by the combination of so-called destroy and repair methods. LNS is especially useful for problems with a tightly constrained search space, such as the Rich Pickup and Delivery Problem with Time Windows (RPDPTW). The RPDPTW is a logistic problem, in which a number of pickup and delivery requests have to be served by a fleet of vehicles. Furthermore, certain time, capacity and feasibility constraints have to be met. Adaptive Large Neighborhood Search (ALNS) is an extension of Large Neighborhood Search, that does not commit to one destroy and repair heuristic. Instead, it chooses in every iteration from a pool of heuristics based on past success. Even though it is a general heuristic, ALNS can compete with most specialized heuristics. Therefore, the goal of this thesis is a detailed description of the ALNS heuristic. A closer look at an application of ALNS is provided by the adaption to the Rich Pickup and Delivery Problem with Time Windows. This work involves an implementation of ALNS for the RPDPTW with some extensions, which has been evaluated experimentally on the problem instance set of Ropke and Pisinger. The parameter values for the evaluation have been determined via automatic parameter tuning. The results of the experimental evaluation confirm the promising results obtained by other papers.

ACKNOWLEDGMENTS

This bachelor thesis has benefited greatly from the support of some people I would like to mention here.
First of all, I would like to thank Prof. Dr. Uwe Schöning and Gunnar Völkel for giving me the opportunity to write this thesis. I am really grateful for getting such an interesting and challenging topic with the opportunity to cover both theoretical and practical aspects with my implementation.
Apart from that, Gunnar Völkel deserves a special recognition for all the help and support throughout the last months as my advisor. Whenever questions occurred, he has found time for meetings and his advice and suggestions have always proven helpful.
Furthermore, I would like to thank my fellow students Tobias Baumann and Lukas Schmid for reading my thesis and their remarks about it.
Finally, I want to express my deep gratitude towards my parents. Thank you for the moral assistance and values that you have provided me with not only during the process of creating this thesis but my whole life long.

CONTENTS

LIST OF ABBREVIATIONS

a_i	i-th action
c_i	capacity of the i-th vehicle.
$c(s)$	costs of solution s
d	degree of destruction
d_{ij}	travel distance from action a_i to action a_j
k_i	length of the tour of the i-th vehicle; also k if i is clear
n	number of requests
m	number of vehicles
$Poss(r)$	set of vehicles that can serve request r
$Poss'(v)$	set of requests that can be served by vehicle v
p_u	length of the update period
q_i	quantity or demand of action a_i
r_i	i-th request
$rel(r_i, r_j)$	relatedness of the i-th and j-th request
R	set of requests
R^-	set of currently unassigned requests
s_i	service time of the i-th action
t_{ij}	travel time from action a_i to action a_j
t_i^{opt}	starting time of the service of the i-th action in the optimal schedule
T_i	tour of the i-th vehicle; also T if i is clear
v_i	i-th vehicle; also v if i is clear
V	set of vehicles
w_i^{close}	closing time of the time window of action a_i
w_i^{open}	opening time of the time window of action a_i
α	distance factor in the solution cost calculation
β	duration factor in the solution cost calculation
γ	unserved request penalty in the solution cost calculation
δ	used vehicle penalty in the solution cost calculation
δ_1	reward for finding a new global best solution
δ_2	reward for finding an improving, not global best, solution
δ_3	reward for finding an accepted non-improving solution
η	noise control parameter
π_j^i	index of the j-th action in the tour of the i-th vehicle
ρ	reaction factor for weight adjustments
ϕ	reduction factor of acceptance methods

1 INTRODUCTION

We want to start this thesis by taking a look at distribution problems occurring in our daily lives. They are the foundation for the problem model that will be used throughout the thesis.

1.1 DISTRIBUTION PROBLEMS

Distribution problems arise when goods need to be distributed from supply locations to demand locations. In general, these problems have a large number of solutions, which differ in the order of visited locations for example. The goal is to find the optimal solution that minimizes the costs, which normally consist of the traveled distance, the duration of the distribution and some other factors. Such distribution problems occur almost everywhere in our daily lives: a postman delivering letters, a salesman driving from client to client, a truck driver conveying goods to customers. These are all examples of distribution or transportation problems in real life that companies try to solve. For instance, the postman does not deliver every single letter separately. Instead, he follows a predefined route that has been chosen by a computer program or an experienced planner. The same holds true for the salesman, the truck driver and many others who are employed at companies that have to transport goods or people. The importance of finding optimal or at least very good solutions to distribution problems becomes obvious by taking a look at the implications. For companies, better solutions to transportation problems mean shorter driving distances for their vehicles or less working time for their employees or a combination thereof. Both effects can save a lot of money, which is ultimately the driving motivation of any company. Because of that, companies invest a substantial part of their budget in modern software systems to find such solutions.

1.2 ADDITIONAL PROBLEM CHARACTERISTICS

The problem of finding the shortest tour for a postman or salesman described in the previous section is already very difficult (for details on the complexity see Section 1.3). Still, it is a simplified abstraction of the distribution problems encountered in real life which contain additional constraints. It is important to notice that adding constraints usually makes the problem more complex and therefore more difficult.
First of all, distribution problems often include more than one vehicle. For example, a postman is not able to serve a whole city. Therefore, the different postmen of the local post office have to be taken into account. Apart from that, distribution problems do not necessarily have to include a depot from where all the goods are distributed. Instead, there are companies which only pick up goods at a customers' location and deliver them to a depot and yet others - like

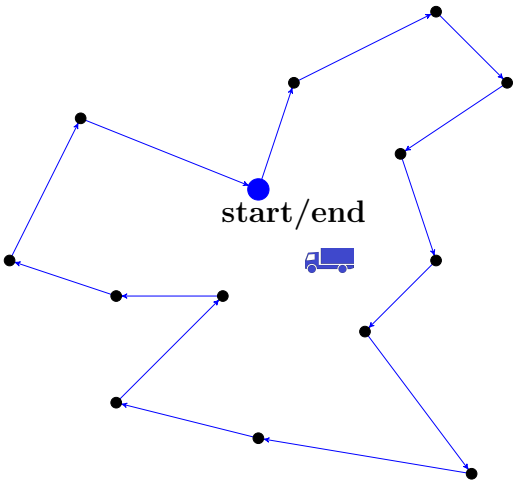


Figure 1: A solution to the TSP consists of a tour that starts at an arbitrary start position, visits all other nodes exactly once and returns to the start position.

a freight forwarder - do both the pickup and the delivery. In most cases, a fleet of vehicles is heterogeneous, i.e. not all vehicles are equal. Again, the postal service is a good example: There are postmen delivering letters and packages with little trucks, vans, bikes and even trolleys. These vehicles are obviously very different in terms of capacity and speed. Besides, some goods might only be transported by special vehicles, e.g. perishable food that needs cooling during the transportation or bulky goods that only fit into the freight hold of some vehicles.
Real life distribution problems are also becoming more and more difficult due to an increasing amount of just-in-time deliveries, i.e. the delivery has to take place at a specific time or during a narrow time slot. If these conditions are not kept, the supplier has to face large penalties. But time windows are not the only kind of time constraints. Due to prevailing legal norms, drivers have to take a break every few hours and are not allowed to work more than a certain number of consecutive hours. Furthermore, drivers might want to work within a relatively stable schedule and working hours similar to their colleagues'.
For all distribution problems, it has to be decided which real life problem characteristics to take into account and which goals to prioritize. Important factors are always the number of vehicles and employees necessary to carry out a distribution, the total traveled distance, the total duration and by that the number of working hours that employees have to be paid for and the number of customer requests that ultimately can be served.

1.3 COMPLEXITY OF DISTRIBUTION PROBLEMS

Distribution problems can be formulated as a decision problem or as an optimization problem. In the decision problem, the task is to decide if there is a solution whose costs are below a given cost maximum k. On the contrary, the goal of the optimization problem is to find the solution with the minimum costs. The Traveling Salesman Decision Problem (TSP-dec) was originally formulated as follows:

Input: a distance matrix $M = (M_{i,j})_{1 \leqslant i,j \leqslant n}, M_{i,j} \in \mathbb{N}$, $k \in \mathbb{N}$.
Question: Is there a permutation π of $\{1, \ldots, n\}$ so that

$$\sum_{i=1}^{n-1} M_{\pi(i),\pi(i+1)} + M_{\pi(n),\pi(1)} \leqslant k \, ?$$

In 1972, Richard M. Karp proved the NP-completeness of the Hamilton Circuit Problem [18]. Based on this knowledge it can be shown that the TSP-dec is NP-complete. It is important to understand the implications of the NP-completeness of the TSP-dec. By definition, NP-complete means that TSP-dec is both NP-hard and in NP. Therefore, we can check in polynomial time if a given TSP-dec tour is a valid solution with costs less or equal to k. At this point we have to distinguish between TSP-dec and the optimization problem TSP-opt. TSP-opt is also NP-hard, but neither has a polynomial verifier been found yet nor has it been proven that TSP-opt $\notin$ NP. Anyway, the fact that both TSP-dec and TSP-opt are NP-hard shows that all problems in NP can be reduced to it. As a consequence, they are at least as difficult complexity-wise as all problems in NP. All of the distribution problems in this thesis including the different Vehicle Routing Problem and Pickup and Delivery Problem variants are generalizations of the TSP-opt. As a direct conclusion, all of them are NP-hard and at least as hard as the TSP-opt. This has enormous implications for solution approaches. It would take even the fastest computers too long to search for the optimal solution of the problems directly because of the superpolynomial runtime (if P $\neq$ NP). Still, finding good solutions is not a hopeless endeavor. With the motivation of the applicability in real life, researchers have focused on solving the individual problems with special heuristics. But while there are plenty of solid heuristics for each of the special problems, general heuristics that can deal with more than one of the problems efficiently are rare.

1.4 ADAPTIVE LARGE NEIGHBORHOOD SEARCH

Ropke and Pisinger [29] have proposed such a general heuristic for the Rich Pickup and Delivery Problem with Time Windows (RPDPTW). It is called Adaptive Large Neighborhood Search (ALNS). The general idea is to repeatedly remove requests from the solution and to reinsert them at a more profitable position. This is done by special destroy and repair heuristics. In contrast to the Large Neighborhood Search (LNS) heuristic proposed by Shaw [37], ALNS uses multiple destroy and repair heuristics instead of just one each. In each iteration, a destroy and a repair heuristic are chosen and applied. The selection is based on the past success of the heuristics. Compared to many local search heuristics that only apply very small changes to a solution, ALNS works with a larger search space, the so-called neighborhood of the current solution. Within one iteration, ALNS can modify up to $30-40\%$ of a solution. This characteristic is especially useful with tightly constrained problems like the RPDPTW. Without the possibility to perform substantial changes, any heuristic is in danger of getting stuck in local optima during the search.

1.5 STRUCTURE OF THE THESIS

At the beginning of the thesis, we formulate the Rich Pickup and Delivery Problem with Time Windows mathematically. The following chapters often refer to the RPDPTW as a prime example for the application of the Adaptive Large Neighborhood Search heuristic. Consecutively, we give an introduction to Large Neighborhood Search, the predecessor of ALNS, to establish the foundation of its more advanced version. Chapter 3 also explains all the adjustments of LNS to ALNS in order to include multiple destroy and repair heuristics. The concrete application of ALNS to the RPDPTW with explicit descriptions of the heuristics is the subject of Chapter 4. In Chapter 5, the implementation of ALNS, its application to the RPDPTW and results of tests are presented. Finally, the thesis ends with a conclusion of the experiments and a summary.

2

RICH PICKUP AND DELIVERY PROBLEM WITH TIME WINDOWS

The Rich Pickup and Delivery Problem with time windows (RPDPTW) offers a very comprehensive model for real world logistic problems. Most of the characteristics mentioned in Section 1.2 are accounted for. We describe the RPDPTW mathematically in Section 2.2. Afterwards, Section 2.3 explains the adjustments that have to be made in order to create one of the more specific problems that are introduced in Section 2.1.

2.1 DISTRIBUTION PROBLEMS IN COMPUTER SCIENCE

Because of the applicability of distribution problems in real life and economic interests, a lot of research has been dedicated to solving distribution problems efficiently. By that, several different abstractions from real life problems have been subject to intensive studies. This section describes the different problems and how they are related to each other. Table 1 offers an overview of the described problems and their characteristics. The information about the problems was taken from different sources: The different kinds of Vehicle Routing Problems were described extensively by Kämpf [17] as well as Toth and Vigo [40]. Parragh, Doerner and Hartl [28] provided an interesting survey on Pickup and Delivery Problems. Finally, Ropke and Pisinger [29] presented the RPDPTW, which is essential for the whole thesis. For a complexity analysis, refer to Section 1.3.

The Traveling Salesman Problem (TSP) is one of the most famous problems in Computer Science. It is based on the daily work of a salesman who has to visit a certain number of customers and afterwards returns to his point of origin, e.g. his home or company. The task is to find a visiting sequence that minimizes the traveled distance while visiting each customer exactly once. Figure 1 shows an example of the TSP.

The Vehicle Routing Problem (VRP) is a generalization of the TSP. As the name suggests, the VRP uses vehicles instead of a salesman. These vehicles all start at a central depot, visit customers in individual routes and then return to the depot. Therefore, the VRP with only one vehicle is exactly the same as the TSP. While the TSP does not include a depot, the start position of the salesman satisfies the same criteria. Sometimes the term VRP is used for the CVRP (see next passage) and the TSP with multiple salesmen is called m-TSP. As a result, the VRP includes not only the problem of finding the optimal minimum distance order within a tour, but also the problem of assigning customers to the different tours of vehicles. The following extensions of the VRP include further details in order to create a model closer to reality.

The idea of the distribution problems with a depot, goods and deliveries is part of the Capacitated Vehicle Routing Problem (CVRP). Instead of just visiting the

problem	capacity	time window	het. fleet	backhaul	pickup & del.
TSP					
VRP					
CVRP	✓				
VRPTW	✓	✓			
SDVRP	✓		✓		
VRPSD	✓				
VRPB(M)	✓			✓	
PDP	✓			✓	✓
RPDPTW	✓	✓	✓	✓	✓

Table 1: The table shows an overview of different distribution problems and some of their characteristics. The characteristics are the inclusion of capacities, time windows, a heterogeneous fleet of vehicles, backhaul actions and combined pickups and deliveries into the problem model. It is important to notice that these are not all problem properties. Still, the RPDPTW subsumes all the other problems as implied by the table.

customers, goods are delivered to them. For that purpose, every customer has a certain demand and every vehicle is assigned a certain capacity. At the depot, the goods are loaded onto the vehicles, so that the capacity of the vehicles is not exceeded.

Moreover, the Vehicle Routing Problem with Time Windows (VRPTW) extends the CVRP by adding time windows. Each customer has such a time window, during which he can be served by a vehicle. Deliveries outside the time window are not allowed. A VRPTW instance and a possible solution are presented in Figure 2.

The Site Dependent Vehicle Routing Problem (SDVRP) includes the aspect of a heterogeneous fleet from the previous section. By that, the vehicles can have different capacities and some customers' requests can only be served by a subset of the vehicles.

In the Vehicle Routing Problem with Split Deliveries (VRPSD), it is allowed to visit a customer multiple times. In the previous problems, each customer was visited exactly once for the delivery or visit. As a consequence, it was not possible to deliver more goods than the maximum capacity of the vehicles. The VRPSD changes that by allowing to split the demanded goods up into several smaller indivisible units. These are then delivered to the customer by possibly multiple vehicles, whereby the customer is visited several times.

The Vehicle Routing Problem with Backhauls (VRPB) introduces a completely new element, the backhauls. These are the opposite of deliveries. Instead of delivering goods to a customer, with a backhaul goods are picked up at a customer's location and transported back to the depot. The VRPB includes both concepts, the delivery and the backhaul. It is important to distinguish between those two types of operations and a third one, the combined pickup and delivery. In the VRPB, it is not possible to pickup goods at a client's location and to deliver them to another client, because this would involve pickup and deliv-

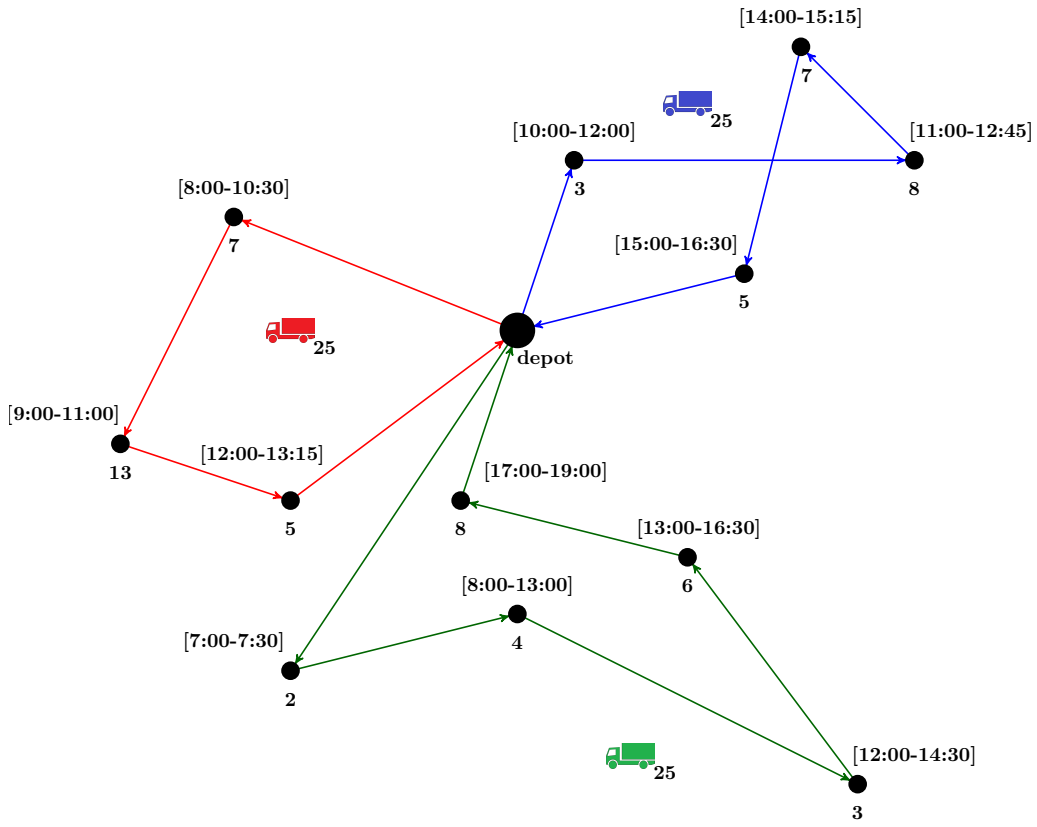


Figure 2: A solution to the VRPTW consists of tours of the different vehicles so that they visit all customers exactly once and then return to the central depot. All vehicles have a capacity of 25. They load the goods at the depot and unload them at the customers' locations. The demand of a customer is written below its node. Additionally each customer has an associated time window in square brackets. The service can only start within that period of time.

ery in the same route and with a sequential order. Furthermore, the deliveries of a route have to take place before the backhauls in order to avoid having to rearrange the goods in the loading space. The Vehicle Routing Problem with Backhauls and Mixed Loads (VRPBM) neglects just that last requirement.
The concept of combining a pickup of goods and the delivery of those goods is realized in the so-called Pickup and Delivery Problem (PDP). The different Vehicle Routing Problems all include a depot from where the goods are distributed. In the PDP, such a depot is not necessary any more, because the goods are picked up during the route and delivered afterwards. Therefore, each vehicle has a start location and an end location instead of a depot. It is important to notice that the vehicle start and end locations can be different but do not necessarily have to be. The same holds true for the start and end location of a vehicle. By that, a depot can be modelled in a PDP with all start and end locations at the depot as well as the pickups. This modification gives the model a lot more flexibility than a fixed depot.
Finally, the Rich Pickup and Delivery Problem with Time Windows (RPDPTW) subsumes all previous problems. It will be properly defined in Chapter 2.
It should be noted that the goal of the presented problems is not always the minimization of the total travel distance. In some problems, the number of

vehicles used should be minimized. Still, other goals are conceivable, e.g. the minimization of the total duration. If the number of pickup and delivery requests is too large to be served by the given fleet of vehicles, an optimization algorithm could also aim at minimizing the number of unserved requests. In the later presented heuristic, all of these alternatives and even a combination of them are possible.

2.2 PROBLEM FORMALIZATION

A Rich Pickup and Delivery Problem with Time Windows contains a set of n requests $R = \{r_i | 1 \leqslant i \leqslant n\}$. Each request r_i consists of a pickup action a_i and a delivery action a_{i+n}. The set of actions is called $A = A_{pickup} \cup A_{delivery} \cup A_{start} \cup A_{end}$, where $A_{pickup} = \{a_i \mid 1 \leqslant i \leqslant n\}$ is the set of pickup actions, $A_{delivery} = \{a_i \mid n+1 \leqslant i \leqslant 2n\}$ is the set of delivery actions, $A_{start} = \{a_i \mid 2n+1 \leqslant i \leqslant 2n+m\}$ is the set of start actions for the vehicles and $A_{end} = \{a_i \mid 2n+m+1 \leqslant i \leqslant 2n+2m\}$ is the set of end actions for the vehicles. Every action has several features, e.g. a certain location. It is important to notice that it is possible that two or more actions in A are equal in terms of their location. Still, they are distinguished, because in this model an action contains more attributes than just the location, e.g. a time window.
Furthermore, there is a set of m vehicles $V = \{v_j \mid 1 \leqslant j \leqslant m\}$. For each request r_i exists a set $Poss(r_i)$ that contains all vehicles that can carry out the pickup and the delivery of the request. Alternatively, $Poss'(v_j)$ could be defined as the set of locations that can be served by vehicle v_j.
With that information, the problem can be represented as a directed graph $G = (A, A \times A)$ with the actions as vertices and edges between them. For each pair of actions (a_i, a_j) the distance and travel time between their locations are denoted as d_{ij} and t_{ij}. In a reasonable model, both are non-negative. Additionally, the travel times satisfy the triangle inequality, so that $\forall i, j, k : t_{ij} \leqslant t_{ik} + t_{kj}$. Moreover, $s_i \geqslant 0$ is the service time for an action a_i that a vehicle spends at the location of a_i for loading and unloading goods. Each action has a time window $[w_i^{open}, w_i^{close}]$. Even the start and end actions of the vehicles have such a time window, so that an upper limit for the total working time can be set for each vehicle driver.
Still, the model lacks a description of the goods. Therefore, let c_j be the capacity of a vehicle v_j. The quantity of goods that is picked up at or delivered to a certain location of an action a_i is q_i. For all pickup actions $a_i \in A_{pickup}$ applies $q_i > 0$, while the delivery actions $a_j \in A_{delivery}$ fulfill $q_j = -q_{j-n}$. This equality corresponds to adding goods to the current load of the vehicle and unloading at the destination.
A solution of the problem contains tours T_j for all vehicles v_j that are represented by a k_j-tuple of actions $(a_{\pi_1^j}, \ldots, a_{\pi_{k_j}^j})$ where $k_j \geqslant 2$ is the number of actions in tour T_j and $a_{\pi_1^j} = a_{j+2n}$, $a_{\pi_{k_j}^j} = a_{j+2n+m}$, i.e. the tour starts with the start action of the vehicle and terminates with its end action. It is worth noting that π_i^j only replaces the actual index of an action. It shows that a tour is a permutation of a subset of A, with j representing the vehicle driving the tour and i being the index within that tour. In order to sustain legibility, we

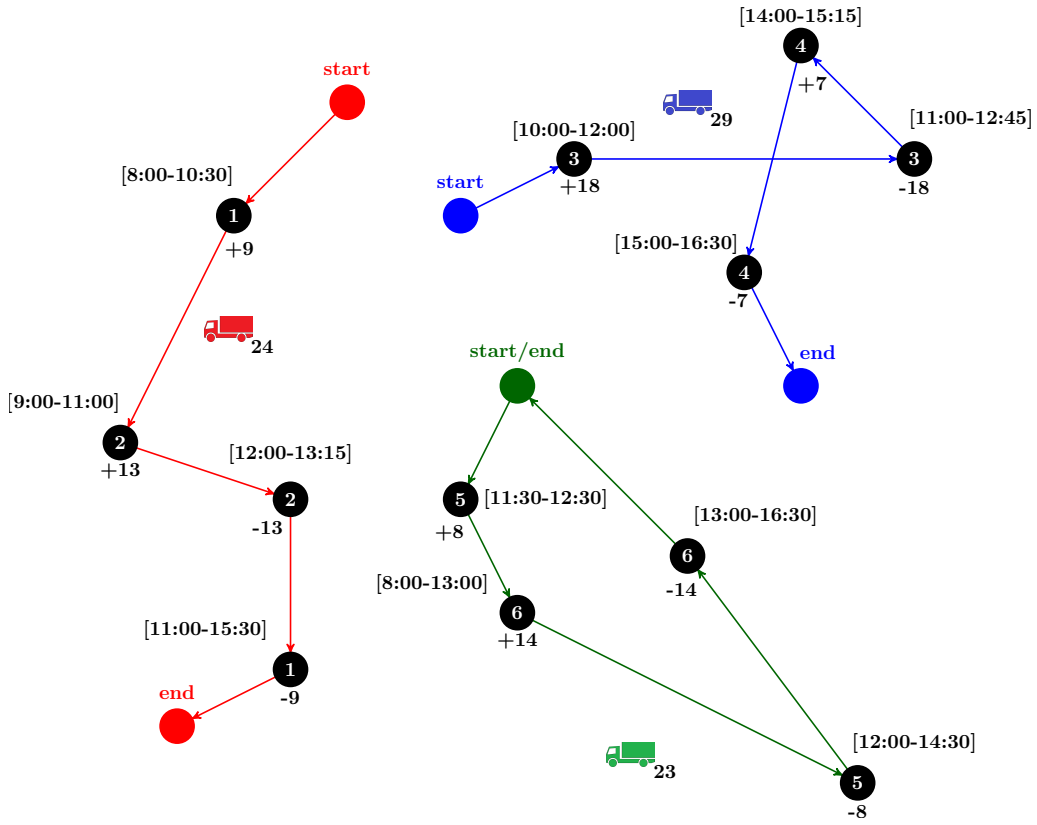


Figure 3: A solution to the RPDPTW consists of tours of the different vehicles so that they peform all pickup and delivery actions exactly once. Each vehicle has a specific start and end position. For the green vehicle, both are in the same place. A customer request consists of a pickup action and a delivery action. Actions that belong to the same request have the same number inside their node and have to be served by the same vehicle. A pickup action has a positive load number next to its node, a delivery action has a negative one. Each vehicle has a capacity written next to the vehicle symbol. They load the goods at the pickup locations and unload them at the delivery locations. Additionally each action has an associated time window in square brackets. The service can only start within that period of time.

assume that the following definitions and rules apply to all vehicles $v_j \in V$ and requests $r_i \in R$ as well as actions $a_i \in A$ and leave out the indices where appropriate. For instance, the tour length k_j is substituted by k whenever it is clear what tour is described.
A tour may only contain actions a from requests than can be executed by the given vehicle. Formally, $\forall v \in V, r \in R$ with an action $a_m \in A, 1 \leqslant m \leqslant 2n$:

$$T = (a_{\pi_1}, \ldots, a_m, \ldots, a_{\pi_k}) \Longrightarrow v \in \text{Poss}(r)$$

Apart from that, both actions of a request have to be served by the same vehicle. Consequently, both the pickup and the delivery action have to be part of the same tour and in the right order, i.e. $\forall v \in V, r \in R$ with an action $a_m \in A$, $1 \leqslant m \leqslant n$:

$$a_m \text{ in tour } T \Longleftrightarrow a_{m+n} \text{ in tour } T$$

$$a_m = a_{\pi_i}, a_{m+n} = a_{\pi_j} \text{ in tour } T \Longrightarrow i < j$$

Furthermore, additional constraints result from the time windows. The arrival time of a vehicle at the location of action a_i along the route is represented as t_i^{arr}. It is not necessarily the same as the starting time of the service, which is named t_i^{st}. First of all, the arrival time satisfies the inequality $t_i^{arr} \leqslant w_i^{close}$. Otherwise, the arriving vehicle would arrive too late to be served. This means that vehicles are allowed to arrive earlier than the opening of the time window, but the actual service does not start before w_i^{open}, i.e. $t_i^{st} = \max\{a_i, w_i^{open}\}$. Obviously, the arrival time at a certain location is dependent on the starting time of the service at the previous location:

$$\begin{aligned} t_{\pi_1}^{arr} &= w_{\pi_1}^{open} && \forall v \in V \\ t_{\pi_{i+1}}^{arr} &= t_{\pi_i}^{st} + s_{\pi_i} + t_{\pi_i \pi_{i+1}} && \forall 1 < i \leqslant k, v \in V \end{aligned}$$

Besides, the capacity constraints of the vehicle must not be violated. The amount of goods stored in the vehicle after action a_i is denoted as l_i. Naturally, the current load may never exceed the capacity c of the vehicle v, i.e.

$$\begin{aligned} c &\geqslant l_{\pi_i} && \forall 1 \leqslant i \leqslant k \\ l_{\pi_1} &= 0 && \\ l_{\pi_{i+1}} &= l_{\pi_i} + q_{\pi_i} && \forall 1 < i \leqslant k \\ l_{\pi_k} &= 0 && \end{aligned}$$

Figure 3 shows an example of a RPDPTW instance including a valid solution with three vehicles.
In order to measure the quality of a solution s, the following formula for the solution costs is introduced:

$$c(s) = \alpha \sum_{j=1}^{m} \text{dist}(T_j) + \beta \sum_{j=1}^{m} \text{dur}(T_j) + \gamma \sum_{i=1}^{n} x_i + \delta \sum_{j=1}^{n} u_j$$

The function dist projects a tour to its total distance, i.e. $\text{dist}(T) = \sum_{i=1}^{k} d_{\pi_i \pi_{i+1}}$. dur calculates the total duration of a whole tour by subtracting the leaving time at the starting position from the arrival time at the destination. In fact, the duration can vary depending on the actual starting time. Often, there is a time interval in which the departure has to take place. In Chapter 4 the optimal tour duration is discussed in greater detail.
The third component of the formula is the binary decision variable x_i with

$$x_i = \begin{cases} 1, & \text{if request } r_i \text{ is not served by a vehicle} \\ 0, & \text{else} \end{cases}$$

Finally, u_j is another binary decision variable that measures if vehicle j has been used to serve a request at all:

$$u_j = \begin{cases} 1, & \text{if } k_j \neq 2, \text{ i.e. if vehicle } v_j \text{ serves at least one request} \\ 0, & \text{else} \end{cases}$$

The parameters α, β, γ and δ can be adjusted according to the user's preferences. In order to minimize one of the four characteristics of the solution, such as the distance for example, one could set $(\alpha, \beta, \gamma, \delta) = (1, 0, 0, 0)$. Alternatively, multiple of the criteria could be selected, possibly with different priorities: $(\alpha, \beta, \gamma, \delta) = (1, 1, N, 0)$ with a large N would make sure that as many requests as possible are served, because every request that is not served causes a penalty of N to the solution costs. N has to be a relatively large number compared to the distances and periods of time in the model. Otherwise, it would probably not affect the costs enough to force the algorithm to serve all requests.

2.3 MAPPING TO SPECIALIZED PROBLEMS

In the introductory Section 2.1 several variations of the Vehicle Routing Problem and other problems were mentioned. They are all more specific than the RPDPTW and can therefore be expressed as a special case of RPDPTW. This section deals with the adaptation of RPDPTW to these problems.

2.3.1 *Traveling Salesman Problem*

The Traveling Salesman Problem (TSP) is very specific compared to RPDPTW. There is only one vehicle, i.e. $m = 1$. The requests can be seen as a pickup at an arbitrary but fixed starting city and a delivery to each of the cities or customers respectively. Alternatively, the situation can be modelled the other way round with pickups at each city or customer and deliveries to the starting point. The start and end position of the vehicle are set as the starting point. Time and load are not important in the TSP. Therefore, all values concerning time and load are set to zero. This also includes the time windows and travel times. By that, only the travel distances are significant for the solution quality. A solution to an example TSP instance is shown in Figure 1.

2.3.2 *Vehicle Routing Problem*

The Vehicle Routing Problem (VRP) uses a similar approach. The differences from the TSP are the given maximum number of vehicles m and the existence of a fixed depot. That depot is the start and end position of all vehicles. As before, every request has its pickup action at the depot and the delivery action is a customer. The basic VRP does not contain capacity constraints or demands, i.e. the customers are only visited and the delivery is not directly included in the model. The following VRP variations are all based on this model. Therefore, we only mention the differences.

2.3.3 *Capacitated Vehicle Routing Problem*

The Capacitated Vehicle Route Problem (CVRP) extends the VRP by adding customer demands larger than zero and capacity constraints to the vehicles. Thus, the load of a vehicle may never exceed its capacity.

2.3.4 *Vehicle Routing Problem with time windows*

We can easily transform the Vehicle Routing Problem with time windows (VRPTW) into a RPDPTW. The approach is the same as for the CVRP. Additionally, the time windows of the delivery locations have to be set according to the VRPTW. The pickup location is always the depot, so we set the time window accordingly. The travel times are very important in this VRP version. An illustration of a VRPTW instance including a solution is presented in Figure 2.

2.3.5 *Site Dependent Vehicle Routing Problem*

In reality, some customers may have special requirements, e.g. in order to deliver a certain kind of furniture an especially spacious truck is needed or the goods require cooling during transport. This aspect is included in the Site Dependent Vehicle Routing Problem (SDVRP). Normally, the set $\mathtt{Poss}(r_i)$ of vehicles that can execute a request r_i includes all vehicles. For the SDVRP, these sets can be restricted. Apart from that, the adaptation of RPDPTW is the same as the one for the CVRP.

2.3.6 *Vehicle Routing Problem with Split Delivery*

The Vehicle Routing Problem with Split Delivery (VRPSD) allows requests to be split up and delivered by multiple vehicles. That can especially be useful when there is more load than the maximum vehicle capacity. For example, a company might want to transport several bulky machines to a new factory building. If that request is modelled, no single vehicle can serve it because the machines do not fit into the loading space. As a result, the request is split up into multiple requests, e.g. one per machine. Unfortunately, it is nigh on impossible to estimate the optimal splits in advance. Therefore, a request that should be split is divided up into as many indivisible parts as possible, each

having the same pickup and delivery locations and possibly different amounts of goods. Then the algorithm can work as usual and has the freedom to serve the different split requests separately or together.

2.3.7 *Vehicle Routing Problem with Backhauls*

The Vehicle Routing Problem with Backhauls (VRPB) introduces a completely new concept. Before, only deliveries from the depot to customers were allowed. Additionally, VRPB includes the possibility of backhauls. A backhaul consists of a pickup of load at the customer's site and its transport to the depot. These are inserted into the model by adding a request for each backhaul. The pickup location is the customer's site, while the delivery location is the depot. The load is adjusted likewise.

3 LARGE NEIGHBORHOOD SEARCH

Neighborhood or Local Search algorithms try to improve an initial solution of an optimization problem by repeatedly applying a local change. The choice of the so-called neighborhood from where the new solution is chosen is obviously crucial. At this point, we need a proper definition of a neighborhood. In the following section, we will give the neighborhood definition by Pisinger and Ropke [30]. The subsequent sections use these neighborhoods in an optimization algorithm as proposed by Shaw [37] and later by Ropke and Pisinger [32].

3.1 NEIGHBORHOODS

Let I be an instance of a combinatorial optimization problem. The set of possible solutions of I which do not violate the constraints of the problem is called $S(I)$. Typically, there is a vast number of solutions which makes it impossible to search through $S(I)$ completely. In order to have quality measures for the solutions, a cost function $c : S(I) \to \mathbb{R}^+$ is introduced. From now on the problem is assumed to be a minimization problem or mathematically speaking:

$$\text{Find } s^* \text{ such that } c(s^*) \leqslant c(s') \quad \forall s' \in S(I).$$

Since the process for a maximization problem is similar, considering the minimization problem only is sufficient. For each solution $s \in S(I)$ we define a neighborhood $N(s) \subseteq S(I)$ with the function $N : S(I) \to \mathcal{P}(S(I))$. In other words, a neighborhood of the solution s is a set of solutions. The way these solutions are selected is specified by N. Of course, there are different ways to set N depending on which part of the current solution s should be changed. The neighborhood contains all the solutions that could be created by changing that part of the solution. The term *neighbor* refers to the similarity between the solution s and its neighbors in $N(s)$. In the search space, they are sort of next to each other. For many problems, a distance measure can even be applied to the solutions in the search space, e.g. the Hamming distance. Solutions in $N(s)$ usually have a comparatively low distance to s.

3.2 NEIGHBORHOOD SEARCH

Neighborhood Search algorithms are based on the use of these neighborhoods. They start with an initial solution and search its neighborhood. Then the procedure is repeated with the best solution found. Algorithm 1 demonstrates Neighborhood Search.

An important factor of Neighborhood Search is the size of such a neighborhood. Although it is more time-consuming to search a larger neighborhood, it might yield better results because the extent of the neighborhood prevents algorithms from getting stuck in local optima. However, for every application the

Algorithm 1: Neighborhood Search

Input: problem instance I
create initial solution $s_{min} \in S(I)$
while stopping criteria not met **do**
  $s' = \arg\min_{s \in N(s_{min})}\{c(s)\}$
  if $c(s') < c(s_{min})$ **then**
    $s_{min} = s'$
return s_{min}

size has to be decided individually in order to get the best results in a reasonable amount of time.

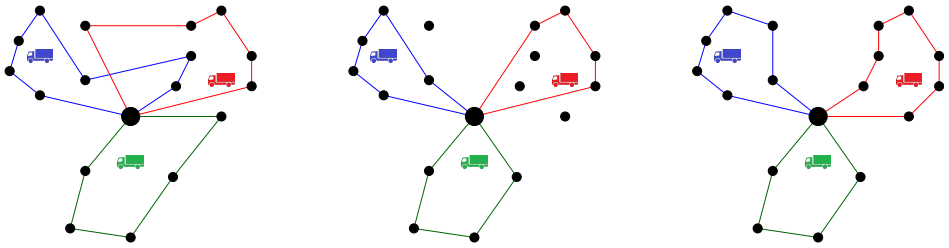
3.3 LARGE NEIGHBORHOOD SEARCH

Large Neighborhood Search (LNS) was originally invented by Shaw [37] and is a heuristic that - as the name implies - uses a large neighborhood approach. LNS belongs to the class of Very Large-Scale Neighborhood Search (VLSN) heuristics. For a profound summary of VLSN heuristics, refer to Pisinger and Ropke [30]. The biggest challenge of such an algorithm is to search the neighborhood efficiently, because the time needed for the search will be the primary contributor to the total runtime of the algorithm.
LNS is slightly different to the Neighborhood Search in Algorithm 1. The schema of LNS is displayed in the following Algorithm 2 similar to the one by Ropke and Pisinger [30].

Algorithm 2: Large Neighborhood Search

Input: problem instance I
create initial solution $s_{min} = s \in S(I)$
while stopping criteria not met **do**
  $s' = r(d(s))$
  if $accept(s, s')$ **then**
    $s = s'$
    if $c(s) < c(s_{min})$ **then**
      $s_{min} = s$
return s_{min}

The algorithm takes the initial solution s and replaces it every time a better solution is found. But the algorithm does not always search in the neighborhood of the currently best solution s_{min}. Instead, it searches the neighborhood of s, which results in the solution s'. While the solutions s and s_{min} are initially equal, differences can occur when a non-improving solution s' is accepted. In this case, s is set to s' and s_{min} remains the same. Notice that the neighborhood function N has been replaced by a combination of the so-called destroy function d and the repair function r. These functions are the most important part of the whole algorithm and will therefore be examined in Section 3.3.2. LNS



(a) initial solution s (b) s′ after removal (c) s′ after reinsertion

Figure 4: An example of the neighborhood concept in the Vehicle Routing Problem. Figure (a) shows an initial suboptimal solution s of the given VRP instance with the depot in the center and three vehicle routes. In Figure (b), solution components (i.e. customers) have been removed from the routes. The neighborhood $N(s)$ contains all solutions that can be created by reinserting the customers into the routes. Figure (c) represents such a solution $s' \in N(s)$.

uses exactly one destroy and one repair function. The use of multiple destroy and repair functions is further examined in Section 3.4. Afterwards, the replacement of the current solution s by the result of the search s' will take place if the accept method triggers it. The accept method is another important part that will be looked into with more detail later in Section 3.3.1. Finally, the best solution found until now is saved in solution s_{min}, which is returned at the end.

Let us, for example, assume the following situation: Figure 4 shows a Vehicle Routing Problem instance with a depot and three vehicles. These vehicles all have tours so that they start at the depot and return there after serving the customers. Obviously, it is not the optimal solution for this VRP instance. With an adequate neighborhood function N, solutions similar to s can be created. The three figures illustrate the destroy and repair process. Figure 4b represents the solution after the removal by the destroy function. The tours remain intact, but some customers are not served any more. Finally, Figure 4c shows the solution after the reinsertion of the previously unserved customers into tours. In this case, the algorithm improved the solution significantly. But it should also be mentioned that the contrary is possible as well.

3.3.1 *Acceptance Method*

The acceptance method of the LNS algorithm has the purpose of deciding whether to continue with the previous solution s or with the newly created one s'. Different ways of implementing the accept method are conceivable. Basically, we could use the acceptance method of every local search framework. What all acceptance methods have in common is the acceptance of improving solutions. They only differ in the acceptance of non-improving solutions. An overview over the concepts as mentioned by Schrimpf, Schneider, Stamm-Wilbrandt and Dueck [8, 35], Ropke and Pisinger [30] and Hu, Kahng and Tsao [14] is presented in Table 2.

method	description
Random Walk (RW)	Every new solution s' is accepted.
Greedy Acceptance (GRE)	The solution s' is only accepted, if it reduces the costs compared to the current solution s. This resembles Algorithm 1.
Simulated Annealing (SA)	Every improving solution s' is accepted. If $c(s') > c(s)$, s' is accepted with probability $exp(\frac{c(s)-c(s')}{T})$ where T is the so-called temperature. The temperature decreases in every iteration by a factor ϕ.
Threshold Accepting (TA)	The solution s' is accepted, if $c(s') - c(s) < T$ with a threshold T. The threshold is decreased in every iteration by a factor ϕ.
Old Bachelor Acceptance (OBA)	The solution s' is accepted, if $c(s') - c(s) < T$ with a threshold T. The threshold is decreased after every acceptance a factor ϕ and increased after every rejection a factor ψ.
Great Deluge Algorithm (GDA)	The solution s' is accepted, if $c(s') < L$ with a level L. The level will decrease by a factor ϕ only if the solution is accepted.

Table 2: The alternatives for the accept method

Greedy Acceptance was the method used by Shaw [37]. It does not accept any worse solutions than the best one found so far. This might therefore limit the search because solutions with a promising neighborhood that seem less desirable at first sight are not accepted due to higher costs. For that reason, other methods have been developed which are often based on a physical process or phenomenon.
Random Walk avoids the problem by simply accepting every solution. It is obvious that the lack of strategy leads to worse results in most cases.
Simulated Annealing - as described by van Laarhoven and Aarts [41] - tries to replicate the physical annealing process. The goal of it is to get the atoms of a piece of metal in a stable state. This is done in two phases: First of all, a metal is heated up until it melts. Afterwards, the metal cools down slowly which enables the atoms to build the solid structure. The second part, i.e. the controlled annealing, is copied in the heuristic. The given initial temperature T decreases in every iteration similar to the annealing metal. For the accept method this means that at the beginning, the probability to accept worse solutions than the current one is high and decreases gradually. More precisely, for an arbitrary but fixed cost difference $c(s) - c(s')$ the probability of acceptance decreases exponentially. Furthermore, for an arbitrary but fixed iteration it decreases exponentially in the cost difference $c(s) - c(s')$. The intention is to let the algorithm search the search space for a while also accepting a slightly worse solution than the current one and to make it focus on more promising solutions later on.
Threshold Acceptance is based on the idea that solutions with higher costs should

be accepted as long as the difference in the costs is below a certain threshold. Similarly to Simulated Annealing the threshold is decreased by a factor ϕ in every iteration. Therefore, the cost difference between the worst solution that would be accepted and the current solution decreases exponentially.
Old Bachelor Acceptance is motivated by the fact that both SA and TA do not take the previous decision into account and rather decrease T monotonously. For example, an algorithm could get stuck in a local optimum at a later point during the execution. Due to their monotonous decrease, SA and TA would probably not allow changes that increase the solution costs even if it were the only way to get away from the local optimum. In contrast, OBA adjusts the threshold by increasing it in case of rejection and by decreasing it in case of acceptance, thus adapting to the success of the algorithm.
The Great Deluge Algorithm makes a compromise between TA and OBA by only decreasing the level in case of acceptance. It is based on a deluge situation in which the water level rises and the fleeing people have to make sure they are above the water level. This description fits a maximization problem which is why the level has to be decreased for a minimization problem.
We will study the success of the different approaches in Chapter 5. We call the factor ϕ reduction factor and ψ increase factor.

3.3.2 *Destroy and Repair*

The repair and destroy functions replace the former neighborhood search function. More accurately, they specify the way in which the search is carried out. The idea is to destroy a part of the solution, i.e. remove it from the solution, and repair it afterwards. Therefore the destroy function has the goal to remove parts that allow the repair function to improve the solution as much as possible. This idea has occured under different names in computer science, from destroy and repair [30], fix and optimize [29], ripup and reroute [6], remove and reinsert [10] to ruin and recreate [35]. Of course, for both functions different heuristics are conceivable. These will be explained in the following sections.

Destroy Heuristics

Before a destroy heuristic can be applied to a solution, we have to determine the degree of destruction d. As the name implies, LNS has a relatively high degree of destruction. This can be useful especially when the search space is tightly constrained because the search space for the repair function is thus expanded. There are different strategies to set the degree of destruction:

- Set a constant degree of destruction.
- Increase the degree of destruction gradually from d_{min} to d_{max}.
- Decrease the degree of destruction gradually from d_{max} to d_{min}.
- Select a random degree of destruction from $[d_{min,max}]$ in every iteration.
- Determine the degree of destruction from $[d_{min,max}]$ based on historical information, i.e. the success that the different values have had so far.

While it is the most important input parameter for every destroy heuristic, it is important to remember that the heuristics work with every possible degree of

destruction.
The following heuristics are rather general. For concrete optimization problems we can apply several more problem-specific heuristics. Chapter 4 for example contains heuristics for the Rich Pickup and Delivery Problem with time windows.

RANDOM REMOVAL Random Removal removes requests uniformly at random. Random Removal might seem to be an unfavorable destroy strategy at first glance because it could remove the already fitting parts of a solution. Still, it has a very positive effect because it provides diversification. As the sole destroy heuristic it may not be productive enough. But especially with the LNS extension ALNS in Section 3.4, which uses multiple heuristics, it gains significance.

WORST REMOVAL The idea of worst removal is to remove the worst parts of the solution, i.e. those that cause the biggest costs, hoping that the repair heuristic is able to eliminate the huge costs. The common way to determine the individual costs of certain parts of a solution is to calculate the difference of the costs of the solution and the costs of the solution without that part.

RELATED REMOVAL Related removal was suggested by Shaw [37] and tries to utilize relations between parts of a solution. We assume that related parts are easy to exchange. The plan is to remove the similar parts so that the repair function has a chance to re-insert them in a more profitable way. So the challenge is to find a reasonable relatedness measure which can be checked very fast.

HISTORY-BASED REMOVAL History-based removal is very problem-specific and will be dealt with in greater detail with respect to the RPDPTW in Chapter 4. Ropke and Pisinger [29, 30] suggested this method that uses historical information on all parts of a solution. This enables it to judge whether a certain part has optimization potential and should therefore be removed or not.

Repair Heuristics

Similarly to the destroy heuristics, there are very simple, but also more complex and therefore time-intensive repair heuristics. The strategies in this section were proposed by Ropke and Pisinger [29]. They are less complex than the Branch-and-Bound heuristic originally proposed by Shaw [37]. While Shaw's method will yield better results in a single iteration, it takes more computing time.

BASIC GREEDY INSERTION This method calculates the minimum insertion cost for every part of the solution that has yet to be inserted. We determine the insertion cost by subtracting the total cost of the solution without the inserted part from the cost of the solution with the inserted part. Afterwards, we insert the one with the minimal cost difference. Therefore, the heuristic calculates

$$\arg \min_{p \in P, i \in IP} c(s_{p,i})$$

where P is the set of the remaining parts, IP the set of insertion possibilities and $s_{p,i}$ represents the solution s where part p is inserted at i. This procedure is repeated until all parts are assigned. Such a greedy strategy is known for often delaying the assignment of costly parts until the end of the process. But at the end there are only few options available so that the costs increase even further. In order to deal with such a situation, we can consider the following regret heuristics.

REGRET HEURISTICS Regret heuristics do not only take the minimum insertion cost into account but also the second cheapest, third cheapest and so on. In general, a Regret-n heuristic calculates the part with the greatest cost difference between the cheapest and the $n-1$ next cheapest insertions. The following formula expresses that by using the symbol $s_i(p)$ for the i-cheapest solution in which part p has already been inserted:

$$\arg\max_{p \in P} \left\{ \sum_{i=2}^{n} (c(s_i(p)) - c(s_1(p))) \right\}$$

The heuristic then inserts the resulting part in the best way possible. Therefore, the inserted part in each step is the one for which a later insertion is assumed to cause the greatest additional cost. Naturally, both the accuracy of the cost prediction and the computing time increase with a larger n. In other words, the higher the selected n, the earlier the heuristic notices that a certain part does not have many advantageous insertion possibilities left.

3.4 ADAPTIVE LARGE NEIGHBORHOOD SEARCH

Adaptive Large Neighborhood Search (ALNS) is an extension of Large Neighborhood Search and was proposed by Ropke and Pisinger. This section is therefore based on their work [32]. Naturally, different problem instances and even different solutions to the same problem are handled by different destroy and repair heuristics with varying success. It may often be difficult to guess which heuristics will be the most advantageous. Therefore, ALNS enables the user to select as many heuristics as he wants. The algorithm will then assign a weight to each heuristic which reflects its success. In the absence of other possibilities, we assume the past success as the best indicator for future success. During the runtime, these weights are adjusted periodically. An update period consists of p_u iterations. The selection of a heuristic in each iteration is then based on these weights. Let $D = \{d_i | i = 1, \ldots, k\}$ be the set of k destroy heuristics and $R = \{r_i | i = 1, \ldots, l\}$ be the set of l repair heuristics. The initially equal weights of the heuristics are denoted by $w(r_i)$ and $w(d_i)$, so that the probabilities to select a heuristic are

$$p(r_i) = \frac{w(r_i)}{\sum_{j=1}^{k} w(r_j)}, \quad p(d_i) = \frac{w(d_i)}{\sum_{j=1}^{l} w(d_j)} \quad \text{respectively.}$$

Apart from the choice of the destroy and repair heuristics and the weight updates every p_u iterations, the basic structure of ALNS is equal to LNS. It is

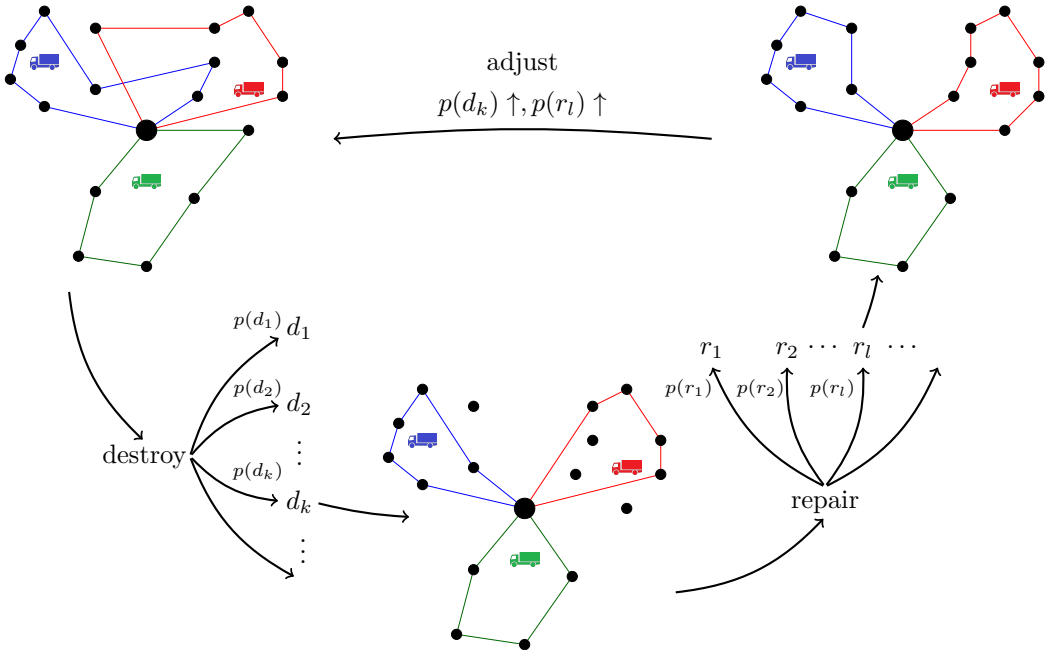


Figure 5: The Schema of Adaptive Large Neighborhood Search. An initial solution goes through the destroy and repair process. Both the destroy and repair heuristic are chosen from a set of heuristics. The selection probabilities are based on their success. After a number of iterations, the probabilities are adjusted according to the success of the heuristics.

presented in Algorithm 3. The schema of ALNS is shown in Figure 5 in the form of a picture.

Algorithm 3: Adaptive Large Neighborhood Search

```
Input: problem instance I
create initial solution s_min = s ∈ S(I)
while stopping criteria not met do
    for i = 1, ..., p_u do
        select r ∈ R, d ∈ D according to probabilities p
        s' = r(d(s))
        if accept(s, s') then
            s = s'
            if c(s) < c(s_min) then
                s_min = s
    adjust the weights w and probabilities p of the heuristics
return s_min
```

The weight adjustment is explained in Section 3.4.1. All heuristics that were suited for LNS are possible for ALNS just as well. Furthermore, Chapter 4 deals with heuristics for a concrete application. At the end of this chapter we give a summary of other possible adjustments and extensions to the ALNS algorithm. Though they are not directly necessary, they can improve the quality of the results dependent on the problem type and instance class.

3.4.1 *Weight Adjustment*

Adjusting the weights of the heuristics is necessary in order to increase the probability that successful heuristics are used more often than less successful heuristics for a specific problem instance. The success of the heuristics may vary between instances of the same problem and even between similar instances. The random removal heuristic is the easiest example of said occurrence: Two executions of random removal paired with the same arbitrary repair heuristic on the same problem instance and initial solution could yield totally different results. Even though most of the other heuristics do not include random decisions - at least in their basic form - slightly different problem instances or current solutions can change their success dramatically. Therefore, the dynamic adjustments are the only way to ensure permanent re-evalution of the heuristics.
In general, there are two approaches. First of all, the weights could be adjusted in every iteration. The advantage of this method is that all weights are up-to-date at all times. On the other hand, it takes more time, because we need to not only adjust the weights, but also calculate the probabilities of the heuristics for the next iteration. The other approach is to execute p_u iterations of the algorithm and adjust the weights afterwards. The success during those iterations has to be saved separately from the current weights. Dependent on the choice of p_u, it can save some computing time.
The following definitions are needed to describe the weight adjustment. As before, $w(h)$ denotes the weight of a heuristic h and p_u is the update period, i.e. the number of iterations that are executed before an adjustment happens. It is worth mentioning that h is either a destroy or a repair heuristic. Both cases are similar, but the different types of heuristics should not be mixed. Furthermore, the number of times the heuristic h has been used during the p_u iterations is called $u(h)$. The success $s(h)$ of h is initialized with zero at the beginning of the period of p_u iterations. After using h in an iteration, $s(h)$ is increased by δ_i if the resulting solution is of the corresponding quality:

δ_1 The new solution is the best one found so far.

δ_2 The new solution improves the current solution.

δ_3 The new solution does not improve the current solution, but is accepted.

For the third case, it is important to keep in mind that - dependent on the chosen accept method - a solution of lower quality than the current one can be accepted. Additionally, it is possible to consider the solution history, i.e. δ_i is only added to $s(h)$ if the solution has not been accepted before. Moreover, for reasonable adjustments we have to ensure the inequality $\delta_1 > \delta_2 > \delta_3$.
Finally, the reaction factor $0 \leqslant \rho \leqslant 1$ controls the influence of the recent success of a heuristic on its weight. Therefore, we calculate the weights for the following iterations by

$$w(h) = \begin{cases} (1-\rho)w(h) + \rho\dfrac{s(h)}{u(h)}, & \text{if } u(h) > 0 \\ (1-\rho)w(h), & \text{if } u(h) = 0 \end{cases}$$

As a result, the parameter ρ is decisive for the weight adjustment. With $\rho = 0$, the weights remain constant on their initial level. In this case, the probabilities

never change. If $\rho = 1$, only the recent success is accounted for. Normally, both the recent success and the heuristic's performance before the last p_u iterations are relevant and should be considered, which means $0 < \rho < 1$.
Furthermore, the reaction factor controls how long the algorithm should work with a set of heuristics instead of just taking the best one in every iteration. The weight of a heuristic h, $w(h)$, is multiplied by $(1-\rho)$ every p_u iterations. This means that the following function is a solid approximation for the weight of rarely used and unsuccessful heuristics:

$$w(h) \approx w_{init}(1-\rho)^{\lfloor k/p_u \rfloor}$$

In the formula, w_{init} denotes the initial weight that is set at the start of the algorithm and k stands for the number of iterations that have been executed. The approximation describes an exponential decrease and a relatively fast convergence to zero. As a consequence, by setting ρ properly the user of the algorithm can regulate approximately after how many iterations most ineffective heuristics should have a weight low enough not to play a significant role any more. For example, a user might want to run the algorithm for 10000 iterations with $p_u = 100$, but after 1000 iterations the weights of the heuristics that have not had success so far should be below 0.01% of the initial weight. Then he could select ρ according to the following inequality:

$$\frac{1}{10000} > (1-\rho)^{\frac{1000}{100}} = (1-\rho)^{10} \Longrightarrow \rho \gtrsim 0.602$$

Naturally, this makes only sense if w_{init} and the rewards δ_1, δ_2 and δ_3 are also selected reasonably and some of the heuristics actually have success. Otherwise, all weights converge to zero regardless of their success. While this does not necessarily reduce the probabilities of the heuristics, it renders the whole concept of the weight adjustments superfluous. As a result, the success should definitely have an effect on the weights.
It is worth mentioning that both the repair and the destroy heuristic are adjusted equally but obviously separately.

3.4.2 *Further Extensions and Adjustments*

Ropke and Pisinger [29] have also extended the basic ALNS approach by adding some new elements which we will discuss in the following.

Penalties for time-intensive Heuristics

Normally, ALNS works with several repair and destroy heuristics. These heuristics can differ greatly in their complexity and thereby in their runtime. More often than not, very time-consuming heuristics have more success than others. As a consequence, the time-consuming heuristics have a better chance of maintaining their weights while the weights of the remaining heuristics diminish. The downside of this effect is that the faster heuristics might execute several iterations in the same amount of time necessary for one iteration of a time-consuming heuristic. In the end, multiple small improvements could be more profitable than a single large improvement. In order to counteract this effect,

penalties can be inflicted on the time-intensive heuristics. For example, we can measure the execution times of the heuristics t_h^{exec} and normalize the success $s(h)$ of a heuristic h according to the following formula:

$$s(h)' = s(h) \cdot \frac{t_{min}^{exec}}{t_h^{exec}}$$

In this formula, t_{min}^{exec} denotes the minimum execution time of all heuristics. Dependent on the kinds of heuristics used, this may be too drastic an influence and it could take away most of the success of the actually better heuristics. As a result, these time-intensive heuristics would be practically eliminated from the algorithm after a certain number of iterations because they would not have any chance to increase their weights substantially. If that is the case, a constant factor instead of the execution time quotient might be the superior alternative. For example, we could consider multiplying 0.5 to the weights of the most time-intensive heuristics.

Noise and Randomization in Heuristics

It has already been mentioned that diversification can be included in ALNS, e.g. with the Random Removal heuristic. In fact, it is one of the biggest strengths of ALNS to include several different heuristics into the search process, some of which directly contribute to diversification like the Random Removal heuristics. Still, these heuristics often do not have much success because their operations are less or even not cost-oriented at all. However, they enable the ALNS algorithm to explore a larger part of the search space and are therefore valuable to the whole process. Unfortunately, the lack of direct success causes diversifying heuristics to be excluded from the search relatively early. Nevertheless, diversification can be preserved - albeit to a lesser extent - by including randomization into heuristics. For example, we can manipulate the cost values of the Basic Greedy Insertion described in Section 3.3.2 by adding a random value, the so-called noise term. Therefore, the currently best insertion is not always executed. This may seem to be suboptimal at first glance, but it can improve the results. The reason for this is that the insertion heuristics are heuristics, i.e. not exact solution methods, which can also make bad choices. The added noise term can alter that choice which can ultimately contribute to the exploration of the second or third best option. By that, we can not only achieve direct improvements compared to the normal operations, but also discover new parts of the search space.
The noise term is chosen randomly from a certain interval. The choice of the interval is obviously very important. A very small interval might not really make a difference while a large interval could produce noise terms that influence the modified costs too much. Therefore, the noise term should be chosen relative to the problem instance. An example is provided in Section 4.5.

special heuristics

In its basic form, ALNS allows every destroy heuristic to be combined with every repair heuristic. It could be extended by assigning a subset of the set of repair heuristics to each destroy heuristic, so that a destroy heuristic can only be

combined with the repair heuristics in its subset. Furthermore, heuristics that do both the destroy and the repair part could be used. This would especially make sense where heuristics that are known to perform well on a certain kind of problem are available, but can not be split into destroy and repair phase. This applies, for instance, to heuristics that only interchange parts of a solution instead of removing and reinserting them. Example for such heuristics are exchange heuristics like 2-Opt and 3-Opt [16].

4

APPLICATION TO THE RPDPTW

This chapter describes the adjustments of and additions to the Adaptive Large Neighborhood Search algorithm due to the application to the Rich Pickup and Delivery Problem with Time Windows. All heuristics are based on or taken from different papers by Ropke and Pisinger [29, 30, 32, 33]. Later in the chapter, different possibilities for the initial solution construction as well as the inclusion of noise and randomization are discussed.

4.1 TIME SCHEDULES

Some of the heuristics in this chapter use the starting times of the service of different time schedules. Their calculation and the ideas behind it are quite extensive. Therefore, this section is dedicated to these time schedules.
Basically, we are interested in determining three different time schedules: the earliest possible with starting times of the service e_i, the lastest possible with l_i and the optimal with t_i^{opt}. While the repair heuristics need the earliest and latest schedule for checking if an insertion is possible, the destroy heuristics use the optimal schedule to compare different requests.
In Figure 6, the three different types are shown. The tour on the left of the figure executes actions as early as possible, i.e. it uses the e_i values. The concepts of t_i^{opt} and l_i produce the same result at action $a_{\pi_{i-1}}$, which is both the optimum in terms of tour duration and the latest possible starting time of the service. The differences start at a_{π_i}: the optimal choice of a starting time of the service is directly after the arrival and marked by $t_{\pi_i}^{opt}$. The latest possible starting time of the service l_{π_i} is later. Actually, it is the latest possible starting time of the service that does not cause the tour to violate the time window constraints for action $a_{\pi_{i+1}}$.

The earliest starting time of the service e_i of an action a_i can be calculated by summing up the traveling distances between the previous actions and possible waiting times: $e_{\pi_i} = \max\{e_{\pi_{i-1}} + s_{\pi_{i-1}} + d_{\pi_{i-1},\pi_i}, w_{\pi_i}^{open}\}$ for $i > 1$. The waiting times occur when the arrival at location happens earlier than the opening time of the time window of the action. As a consequence, the e_i values are calculated by traversing the whole tour. Similarly, the latest starting time of the service l_i is determined by a backward calculation. In general, l_i can be defined as $l_{\pi_i} = \min\{l_{\pi_{i+1}} - d_{\pi_i,\pi_{i+1}} - s_{\pi_i}, w_{\pi_i}^{close}\}$ for $i < k$ and $l_{\pi_k} = w_{\pi_k}^{close}$, i.e. each action is executed as late as possible. Both e_i and l_i have to be updated every time the action a_i is reinserted. During that update, every tour is traversed. At the same time, the waiting times for all actions are determined, i.e. the time that a vehicle waits idly until the time window of the next action opens. Below, these waiting times, denoted by t_i^w, are used to get the starting time of the service of an arbitrary action in constant time. The cumulative waiting time t_i^{cw} of an

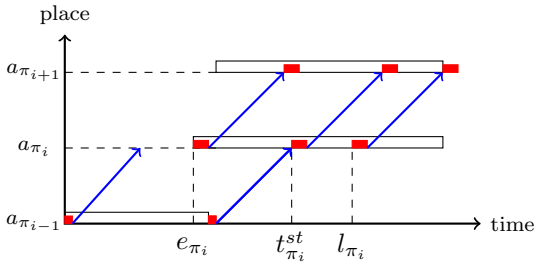


Figure 6: The chart demonstrates the difference between the concepts of time schedules at the example of three differently timed tours at action a_{π_i}. Departing from $a_{\pi_{i-1}}$ as early as possible results in an early arrival. After the waiting time, the earliest possible starting time of the service of a_{π_i}, e_{π_i}, is reached. The optimal and latest possible time schedules are also included. At $a_{\pi_{i-1}}$, the latest possible starting time of the service is clearly at the end of the time window. The differences start at a_{π_i}. $t^{opt}_{\pi_i}$ is the optimal starting time of the service in terms of tour duration. Finally, l_{π_i} is the latest possible starting time of the service. In this case, it includes unnecessary waiting time between l_{π_i} and $t^{opt}_{\pi_i}$.

action a_i is the sum of all waiting times occurring before actions in the same tour that are visited before a_i:

$$t^{cw}_{\pi_i} = \sum_{j=1}^{i} t^{w}_{\pi_j}$$

Naturally, the starting time of the tour influences the waiting times. As a result, the starting time of the optimal tour should be set such that the cumulative waiting time of the whole tour is minimized. The problem with the earliest alternative is that it normally contains unnecessary waiting time, which can be avoided by starting the tour later. It suggests itself to take the latest possible starting time of the tour instead. This approach guarantees that no unnecessary waiting time at any location is included in the tour duration. The reason for this is that unnecessary waiting times can only occur if a vehicle arrives earlier than the time window opening time and the vehicle could have arrived later. Both conditions point to the latest possible starting time of the tour as the optimal choice. Therefore, in this chapter t^{opt}_i is considered to be the optimal starting time of the service of an action a_i after the tour has started at the latest possible starting time. It is important to distinguish between the tour that starts at the latest possible time and the tour that executes actions at the latest possible time, i.e. the latest time schedule. The former is the optimal tour with minimum waiting time whose starting times of the service we need to calculate. The latter is the tour that executes an action a_i at the time l_i, even if that includes waiting unnecessarily after the time window of the action has already opened.
The method to calculate the t^{opt}_i values is to traverse the tour while keeping track of the current time. Interestingly, this approach always requires traversing the whole tour even if the latest starting time of the service of the first action within the tour is wanted, because every later action could cause the whole tour to be shifted backwards. Figure 7 demonstrates that problem. As a result, the time needed to calculate the latest possible starting time of the service for an

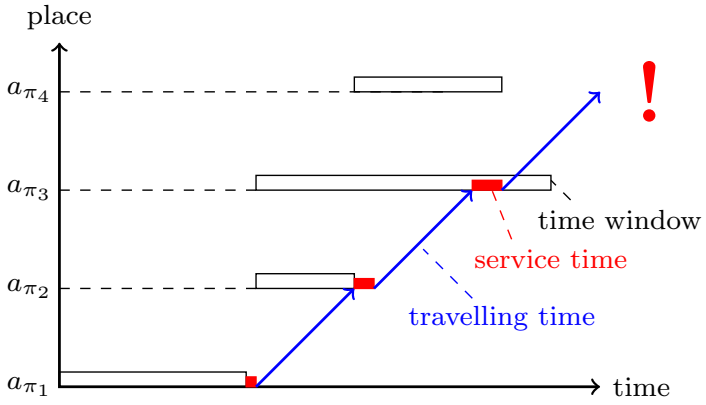
arbitrary action within a tour with this method is linear in the tour length. Still, it only has to be done once for the whole time schedule of a tour.

But there is yet another approach that enables the heuristics to get t_i^{opt} in constant time. It has already been explained that the later a tour starts, the less waiting time is included. The problem is that only the waiting times for the earliest possible tour during the calculation of the e_i values and the waiting times of the latest possible tour during the calculation of the l_i values can be measured without additional effort. The challenge is therefore to find a way to get the optimal starting time of service t_i^{opt} while only having the e_i, l_i and t_i^w values. In the following descriptions, t_i^w is always the waiting time that occurs when taking the earliest possible tour. The waiting time of the whole tour, $t_{\pi_k}^{cw}$ includes both necessary waiting time that even the optimal tour includes and unnecessary waiting time. Unnecessary waiting time occurs when a vehicle arrives at an action too early even though it would have been possible to start the tour later and avoid that waiting time. The service times, traveling times and necessary waiting times of the earliest possible and the optimal tour are equal. Therefore, the only difference between the durations of the tours arises from the unnecessary waiting time, which is illustrated in Figure 8. The marked buffer is the difference in the starting times of the tours. It is calculated by $l_{\pi_1} - e_{\pi_1}$ and will be called Δ in this section. The difference between the optimal and the earliest possible time schedule is initially Δ. From then on, it decreases every time the earliest time schedule includes unnecessary waiting times. The amount by which it decreases is exactly the waiting time t_i^w. This continues until the sum of the waiting times t_i^{cw} is larger than the buffer Δ. This situation happens in the figure at action a_{π_5}. The following waiting time can not be avoided by shifting the starting time of the tour to a later point. As a consequence, the optimal starting time of the service t_i^{opt} can be calculated by

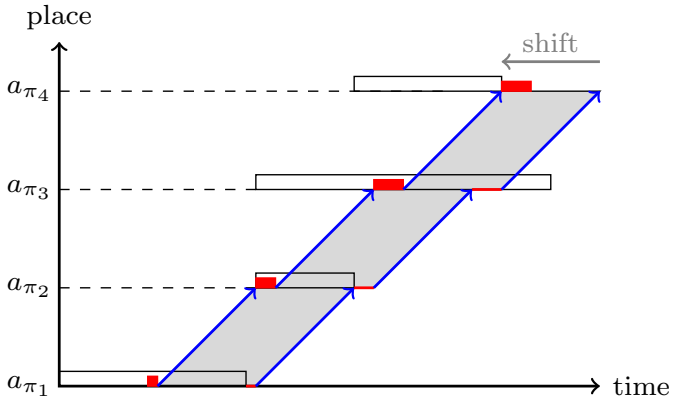
$$t_{\pi_i}^{opt} = \begin{cases} e_{\pi_i} + \Delta - t_{\pi_i}^{cw}, & \text{if } t_{\pi_i}^{cw} \leqslant \Delta \\ e_{\pi_i}, & \text{if } t_{\pi_i}^{cw} > \Delta \end{cases}$$

The first case corresponds to the decreasing difference between optimal and earliest tour while the second case happens when further waiting time can not be avoided. The earliest starting time of the service e_i is then definitely the optimal one. Interestingly, the number of optimal tours is one if $t_{\pi_k}^{cw} \geqslant \Delta$, because every backward shift would result in unnecessary waiting time. On the other hand, if $t_{\pi_k}^{cw} < \Delta$ there is an infinite number of optimal tours with starting times between $[l_{\pi_1} - \Delta + t_{\pi_k}^{cw}, l_{\pi_1}]$.

In summary, the optimal starting time of the service t_i^{opt} of any action a_i within a tour can be calculated in constant time if the waiting times t_i^w and the cumulative waiting times t_i^{cw} are regularly updated with the earliest and latest possible time schedules.



(a) forward calculation with late arrival



(b) backward shift

Figure 7: Figure (a) shows the illustrates the process of the forward calculation when traversing the tour in order to get the latest possible time schedule. The chart shows the position of a vehicle relative to different locations where actions should be executed. Furthermore, the duration of different actions is shown, e.g. the traveling time between locations, the service time for actions. It is worth noting that the time windows are only rectangles in order to make the other lines more easily visible. The trip starts with the first action a_{π_1} at the latest possible starting time of the service, which is at the end of the time window. After the service time, the vehicle continues by traveling to the location of action a_{π_2}. This happens for all actions until a_{π_4} is reached. The time of arrival is obviously outside the time window, so that the action can not be executed. The solution to that problem is shown in Figure (b). The goal is again to get the latest possible starting time of the service. Therefore, the shift amount is exactly the difference between the time window closing time and the current arrival time. The resulting time schedule of the tour is at the left border of the grey area.

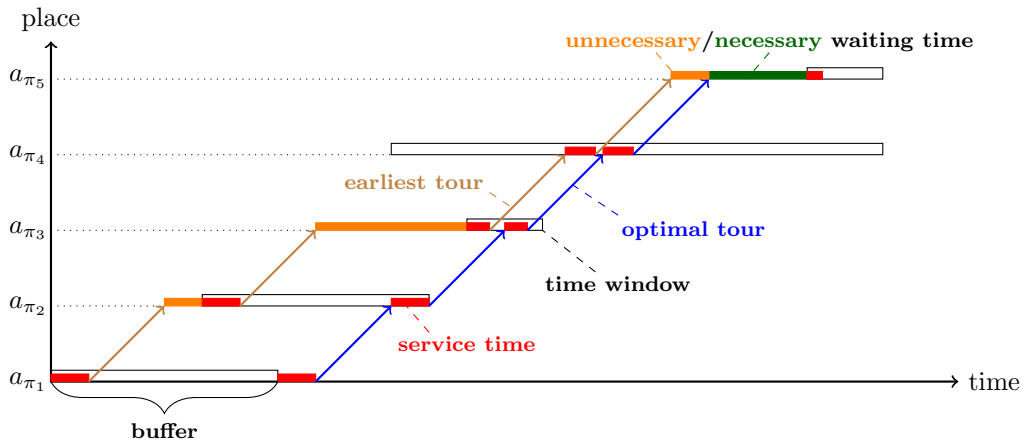


Figure 8: The earliest possible and the optimal tour. The earliest possible tour is drawn in brown, the optimal tour in blue. The difference between them decreases during the tour because of the unnecessary waiting times. At the point where the sum of the waiting times is larger than the buffer, i.e. the difference between the starting times of the tours, the necessary waiting time starts.

4.2 DESTROY HEURISTICS

The destroy heuristics in this section all fit into the categories described in Section 3.3.2. Related Removal and History-based Removal are more specific to the problem and are therefore explained in greater detail.

4.2.1 *Random Removal*

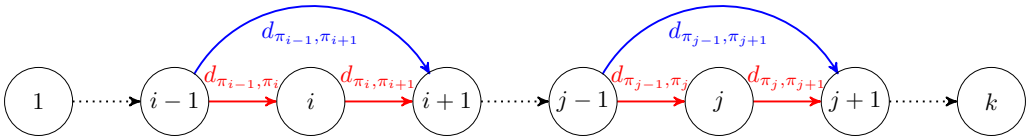
Random Removal is the most simple destroy heuristic. It repeatedly removes requests randomly until the current degree of destruction is reached. It is a very important heuristic for the ALNS algorithm because it can remove any request of the solution regardless of the solution costs. By that, Random Removal creates diversification which is very important when exploring a large search space.

4.2.2 *Worst Removal*

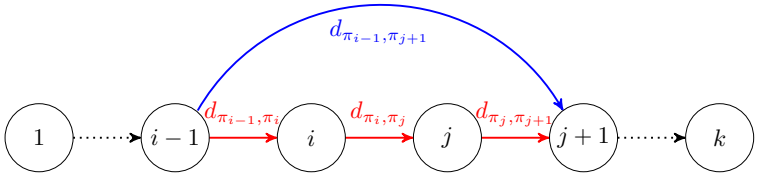
Worst Removal works exactly as described in Section 3.3.2. For all requests $r \in R$, the difference $c(s) - c(s^{-r})$ is calculated. c is the cost function that has been explained in Section 2.2, s stands for the current solution and s^{-r} is the current solution without request r. In a sense, the removal of all requests is simulated. Then, the request that maximizes the cost difference is removed:

$$\arg\min_{r \in R} c(s) - c(s^{-r})$$

This procedure is repeated until enough requests are removed. It is important to notice that the cost differences change after every removal. However, not all cost differences have to be calculated anew, but only the ones of the requests which remain in the tour of the previously removed request. Apart from that,



(a) actions a_i and a_j not next to each other



(b) actions a_i and a_j next to each other

Figure 9: Figures (a) and (b) illustrate the calculation of the distance differences in the simplified Worst Removal heuristic. With these schemata all costs of solutions without an arbitrary request can be calculated in $\mathcal{O}(1)$. In the tours, the request r whose removal is simulated consists of the pickup action a_{π_i} and the delivery action a_{π_j}. Figure (a) shows the situation if pickup and delivery are not directly next to each other in the tour, while in Figure (b) the delivery follows directly after the pickup. If request r is removed, the red edges are removed from the tour and the blue edge is inserted into the tour.

instead of the cost difference only the difference of the tour distances could be measured. With that change, the calculation can be done in $\mathcal{O}(1)$. This is possible, because the previous total tour distance of the tour $T = (a_{\pi_1}, \ldots, a_{\pi_k})$ was determined by the sum of the distances $d_{\pi_i,\pi_{i+1}}$, $i = 1, \ldots, k-1$. The possible removal of a request r would actually cause two removals because a request consists of a pickup and a delivery. Assuming the position of the pickup is i and the position of the delivery is j, the tour distance without r can easily be calculated. There are two cases that have to be distinguished (see Figure 9): If $j = i+1$, i.e. if the delivery action directly follows the pickup action in tour T, then the distance difference is calculated by $d_{\pi_{i-1},\pi_i} + d_{\pi_i,\pi_j} + d_{\pi_j,\pi_{j+1}} - d_{\pi_{i-1},\pi_{j+1}}$. If $j > i+1$, the difference is $d_{\pi_{i-1},\pi_i} + d_{\pi_i,\pi_{i+1}} - d_{\pi_{i-1},\pi_{i+1}} + d_{\pi_{j-1},\pi_j} + d_{\pi_j,\pi_{j+1}} - d_{\pi_{j-1},\pi_{j+1}}$.

4.2.3 *Related Removal*

Related Removal is based on the idea that requests that are in some way related can easily be exchanged. Exchange means that the requests switch route assignments. Naturally, a destroy heuristic only removes requests. But by removing related requests it gives the repair heuristic the chance to explore the possibility of actually executing the switch if it turns out to be profitable. There are basically two approaches to determine a reasonable relatedness measure in the Rich Pickup and Delivery Problem with Time Windows. These are explained in the sections on the Distance-oriented and Service time-oriented Removal. Furthermore, Cluster Removal is very similar to Distance-oriented Removal and is also described in this section. Finally, Shaw Removal combines and extends both approaches.
In general, there are two different execution structures for a Related Removal

heuristic. The first one is the classic Related Removal taken from Ropke and Pisinger [29]. It is presented in Algorithm 4. Only the first removal is decided randomly. The requests removed after that are highly related to at least one already removed request $r_i \in R^-$. In contrast, Randomized Related Removal in Algorithm 5 is a new approach that repeatedly selects a random request from the solution and its most related request for the removal. Both approaches work for all Related Removal heuristics except for the Cluster Removal. This also includes the Historical Request Pair Removal from Section 4.2.4 which can be used like a typical Related Removal heuristic.

Algorithm 4: Related Removal

Input: current solution s, set of unassigned requests R^-, degree of destruction d
determine relatedness values $rel(r_i, r_j) \quad \forall r_i, r_j \in R$
remove a random request r from s
$R^- = R^- \cup \{r\}$
for $j = 1, \dots, d-1$ **do**
  select request $r_i \in R^-$ randomly
  remove request $r' = \underset{r_j \in R \setminus R^-, i \neq j}{\arg\min} \; rel(r_i, r_j)$ from s
  $R^- = R^- \cup \{r'\}$
return s

Algorithm 5: Randomized Related Removal

Input: current solution s, set of unassigned requests R^-, degree of destruction d
determine relatedness values $rel(r_i, r_j) \quad \forall r_i, r_j \in R$
for $j = 1, \dots, \frac{d}{2}$ **do**
  remove a random request r_i from s
  remove request $r' = \underset{r_j \in R \setminus R^-, i \neq j}{\arg\min} \; rel(r_i, r_j)$ from s
  $R^- = R^- \cup \{r_i, r'\}$
return s

Distance-oriented Removal

Distance-oriented Removal uses the distances between the pickup and delivery actions of pairs of requests to measure the relatedness:

$$rel(r_i, r_j) = d_{i,j} + d_{i,j+n} + d_{i+n,j} + d_{i+n,j+n} \qquad \forall r_i, r_j \in R$$

This cross schema is illustrated in Figure 10. All possible edges are taken into account for the calculation apart from the edges between actions of the same request. These would not improve the informative value of the relatedness values, because the distance between pickup and delivery of the same request does not influence the relation to another request.

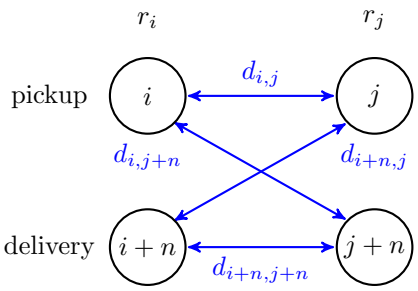


Figure 10: The figure shows the cross schema for the Distance-oriented Removal heuristic. The distances, i.e. the weight of the displayed edges are included in the calculation of the relatedness measure. The vertices are named after their indexes, i.e. request r_i consists of pickup action a_i and delivery action a_{i+n}.

A low relatedness value $rel(r_i, r_j)$ of a pair of requests r_i, r_j means that they are highly related. This definition makes sense, because in that case the sum of the distances is low, i.e. the locations of the actions are relatively close to each other. On the other hand, a high relatedness value indicates that one or more of the distances were high. In that case, the requests are probably difficult to interchange.
If properly implemented, this heuristic is by far the fastest of all Related Removal heuristics. The reason for this is that the distances between the different locations do not change during the execution of the ALNS algorithm and can therefore be calculated in the beginning. Furthermore, rankings based on the relatedness values can be calculated once and used throughout the whole algorithm. The other three Related Removal heuristics and also the History-based Removal heuristics require updates to their data every single time they are called. While this can prove to be time-consuming especially for larger instances, the results are often better, because they do not just look at the problem instance but at the specific solution that is worked on at the moment.

Cluster Removal

Cluster Removal is inspired by the results of observing the behavior of the Distance-oriented Removal and the evolution of the solution. Some RPDPTW instances mainly consist of a few clusters, i.e. many locations of actions are in a small area. This resembles the real life situation of having large cities where several customers are located, albeit not in the same part of the city, but scattered over all parts of the town. Figure 11 shows such a scenario. The tour starts and ends at the central position and visits both clusters. The red path in the figure is the shortest path between the clusters. In order to significantly improve the solution costs, the destroy heuristic should try to remove a whole cluster and assign it to a different tour. By that, the long distance of the red path would be avoided. If the previously presented Distance-oriented Removal heuristic were applied to this example, it would probably remove a few requests of a cluster and then move on to other requests in a neighboring tour. However, in most cases, some requests would be left of the modified cluster. As a consequence, the repair heuristic would reinsert the requests in the same tour with a high probability, because there are already distance-related requests in that tour and

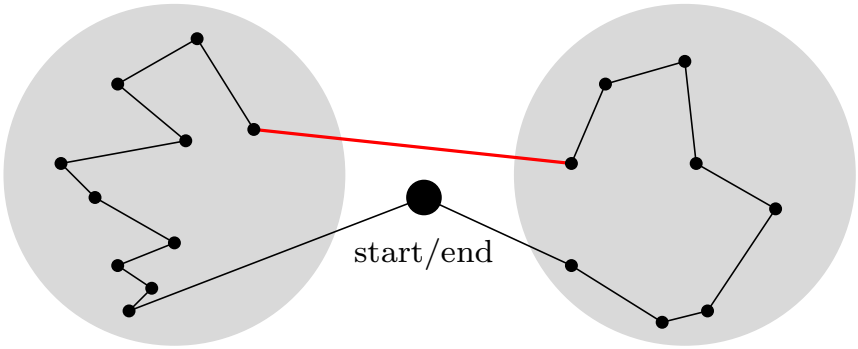


Figure 11: The figure shows the tour of a vehicle that starts and ends at the central position. The main characteristic of the tour is the division into two clusters which are stressed by the grey background. The shortest connection path between the clusters is highlighted in red. It is assumed that both pickup and delivery of any request are in the same cluster.

therefore the additional costs of an insertion are low. This is obviously not the intended result. The intuitional solution of the problem is to remove the whole cluster. This would have two positive effects. First of all, the removed requests can be inserted in any tour in a more favorable way, possibly in an already existing cluster at the same place. Another important implication is that the vehicle from whose tour the cluster is removed gets a lot of free time that has previously been spent on serving the second cluster and traveling between the clusters. With an efficient repair heuristic, this time can be used to serve requests that are approximately in the same area as the remaining cluster.

All these thoughts and hopes are the reason for the invention of Cluster Removal. Still, only the goal and its effects have been named. It can be achieved in several ways, two of which are described in this section.

The first method to remove a cluster from a tour is to apply the Distance-oriented Removal heuristic solely to one tour. The problem here is that the heuristic has to recognize when the whole cluster is removed. If it stops before the whole cluster is removed and continues with another tour, the already described problem of the repeated insertion by the repair heuristic is likely to occur. On the other hand, if it removes more requests than necessary, some probably well-fitting parts are taken away from the solution. The key is to recognize at which point the relatedness values escalate compared to the lower values. In Figure 11, pairs of requests in the same cluster have low relatedness values, while an arbitrary pair of requests from different clusters has comparatively high relatedness values. To recognize that difference can prove difficult for an algorithm, especially if the clusters are not as clearly distinguished as in the figure.

The easier method is to use the minimum spanning tree (MST) of a special graph for the selection of requests. A spanning tree is a tree that connects all nodes. The MST is a spanning tree whose sum of edge weights is minimal. Here, we use the graph consisting of the requests of a single tour. The weights of the edges between the requests are the relatedness values of the requests connected by the edge. It is important to keep in mind that these edges in this special graph have nothing to do with the actual edges in the tour of the vehicle.

The MST can be efficiently calculated with Joseph Kruskal's Algorithm [22]. It sorts the edges in ascending order according to their relatedness values and inserts them one after another in the initially edge-free graph. If the graph with the additional edge contains a cycle, the edge is removed, otherwise it is kept. Kruskal has proven that the resulting graph is the MST. In fact, the algorithm can stop after $q-1$ steps if q requests are involved, because a tree of q vertices always contains $q-1$ edges. In order to determine the two clusters, the last inserted edge is removed. It is automatically the edge with the maximum relatedness value among the edges of the MST because of the ascending order of the insertions. By removing the last inserted edge, two sets of requests are created that have a maximum distance from each other in terms of relatedness. The relatedness resembles the distance relation between requests and therefore these sets correspond to the desired clusters. Finally, we describe the general course of action of Cluster Removal as in Algorithm 6. First of all, a random request is selected. The tour the request is a part of is then divided into two clusters with a suitable method like the modified minimum spanning tree technique. One of the clusters is selected randomly and its requests are removed. If the current degree of destruction is larger than the number of already removed requests, the procedure is repeated, although with a different tour. If the selected cluster is larger than the remaining number of removals, a random subset of the cluster is removed.

Algorithm 6: Cluster Removal

Input: current solution s, set of unassigned requests R^-, degree of destruction d, distance relatedness values $rel(r_i, r_j)$

$n_{removed} = 0$
while $n_{removed} < d$ **do**
  select a tour T randomly from s
  determine two clusters C_1, C_2 from the requests in T
  select one of the clusters $C_{selected}$ randomly
  if $d - n_{removed} \geqslant |C_{selected}|$ **then**
    remove all requests in $C_{selected}$ from s
    $R^- = R^- \cup C_{selected}$
  else
    remove all requests of a randomly chosen subset $C' \subset C_{selected}$ with $|C'| = d - n_{removed}$
    $R^- = R^- \cup C'$
return s

Service time-oriented Removal

Service time-oriented Removal is another version of Related Removal. Instead of the distance-oriented approach, it uses the starting time of the service to determine the relatedness between requests. The idea is again that requests with similar starting times of the service can be swapped easily. Naturally, this only works if the requests are not too far apart. Otherwise, the distance between them can not be traveled fast enough to make use of the similar starting time of the service. Therefore, a random request r is selected and only the set $D(r)$ of d

requests with the lowest distance-oriented relatedness values are considered by the heuristic. As a reminder, d is the current degree of destruction. This ensures that the requests in $D(r)$ are as close as possible to r, thus eliminating the distance between them as the restrictive factor for the exchange of the requests. The actual relatedness values of the Service time-oriented Removal heuristic are calculated by the following formula:

$$rel(r_i, r_j) = \begin{cases} |t_i^{opt} - t_j^{opt}| + |t_{i+n}^{opt} - t_{j+n}^{opt}|, & \text{if } r_j \in D(r_i) \\ \infty, & \text{else} \end{cases}$$

The actions a_i and a_{i+n} are the pickup and the delivery action of the request r_i. That is why both are included in the calculation. If a request r_j is not in $D(r_i)$, the relatedness value is set to infinity thus eliminating it as a removal candidate. The set $D(r)$ of an arbitrary request r initially contains d requests. As a result, when r is selected, the heuristic will never select a request that is not in $D(r)$ for removal, because d is the maximum number of removable requests.

Shaw Removal

Originally, Shaw [36, 37] proposed this heuristic and it was modified and named after its inventor by Ropke and Pisinger [32]. Shaw Removal combines both the service time-oriented and the distance-oriented approach. The idea is that Distance-oriented Removal removes requests whose locations are close to each other but whose time windows do not fit together in order to let the repair heuristic perform an exchange. On the other hand, Service time-oriented Removal might remove requests which are too far apart to be exchanged. Therefore, Shaw Removal tries to get the best of both worlds. In fact, it includes even the vehicle service constraints and the demands of the requests into the relatedness measure. In Chapter 2, $Poss(r)$ was defined as the set of vehicles that can serve the request r. Furthermore, the demand of a request r_i is specified by the quantity q_i. Both relatedness measures of the previous sections are used and have to be distinguished. Let rel_d be the distance-based and rel_t the service time-based relatedness function. Then the relatedness for the Shaw Removal heuristic is defined as

$$\begin{aligned} rel(r_i, r_j) = \lambda \cdot rel_d(r_i, r_j) + \mu \cdot rel_t(r_i, r_j) + \nu \cdot |q_i - q_j| \\ + \xi \cdot \left| 1 - \frac{|Poss(r_i) \cap Poss(r_j)|}{\min\{|Poss(r_i)|, |Poss(r_j)|\}} \right| \qquad \forall r_i, r_j \in R \end{aligned}$$

λ, μ, ν and ξ are fixed parameters that are chosen before the execution of the algorithm. For a discussion of reasonable domains for these parameters, refer to Chapter 5. The functions rel_d and rel_t have already been described in the previous sections. Low relatedness values also signify a high degree of relatedness in the Shaw Removal heuristic. The closer the quantities q_i and q_j are to each other, the lower the absolute value of their difference will be. Similarly, the more matches there are between $Poss(r_i)$ and $Poss(r_j)$, i.e. the more vehicles both requests can be served by, the closer the quotient in the formula will be to one.

Apart from the relatedness values, the procedure is exactly the same as with

Distance-oriented Removal: One request is chosen at random and removed from the solution. From then on, a random request r is chosen from the already removed requests and the request that is the most related to r and is still a part of the solution is removed. This procedure continues until the degree of destruction is reached.

4.2.4 *History-based Removal*

History-based Removal heuristics try to utilize historical information about the success different combinations of requests or actions have had during the current run of the algorithm. It suggests itself that it takes such heuristics several iterations to collect enough information to work properly. Therefore, History-based Removal heuristics should be excluded from the selection of a destroy heuristic at the beginning. The number of iterations after which the restriction should be lifted is dependent on the kind of heuristic and its parameters. Historical Request Pair Removal for example uses information about a certain number of top solutions. As a consequence, Historical Request Pair Removal should not be used before that number or rather several times that number of solutions has been created. Otherwise, the removal is based on a history of bad solutions, which will most likely not improve the solution. While there is no such explicit number of iterations of exemption for Historical Action Pair Removal, the second History-based Removal heuristic, it should definitely be excluded for some time. The easiest approach is to activate it in the same iteration as Historical Request Pair Removal.
In contrast to the previous heuristics, History-based Removal heuristics require updates or at least checks for the necessity of updates in every iteration. This even holds true for iterations in which another destroy heuristic was used. Service time-oriented Removal and Shaw Removal also require updates, but only when they are selected.

Historical Request Pair Removal

Historical Request Pair Removal keeps track of the number of times that each pair of requests has been in the same tour in the top t solutions found so far. This value is denoted by $top(r_i, r_j)\ \forall r_i, r_j \in R$. By that, it infers which combinations of requests in a tour are successful. With that information, Historical Request Pair Removal is a typical Related Removal heuristic. The historical pair success value $top(r_i, r_j)$ serves for the calculation of the relatedness measure:

$$rel(r_i, r_j) = \begin{cases} \dfrac{1}{top(r_i, r_j)}, & \text{if } top(r_i, r_j) > 0 \\ \infty, & \text{else} \end{cases}$$

Compared to the other Related Removal heuristics, the relatedness values of Historical Request Pair Removal are much more likely to be equal because they are integer values. If such a conflict with equal values should arise, the possible removals are taken equally likely.
If t is a large number, many solutions and their associated data structures have to be administered. The operations on the set of solutions include insertions

and removals. But a solution is only removed if it is the one with the highest costs within the solution set and a better solution has been found. In other words, the retrieval operation includes the search for the solution with the maximum costs.
On the other hand, t should not be set too low. Otherwise most of the data will be useless because no meaningful ranking of the request pairs can be set up.
Furthermore, not only the set of solutions but also the data structure containing the top-values has to be updated with every substitution. This includes only simple increment and decrement operations, but every pair of requests in the same tour has to be taken into account. In the worst case all requests are in the same tour, which would take n^2 operations. Normally, this case should not occur, so that the number of operations is well below n^2.

Historical Action Pair Removal

Historical Action Pair Removal tries to identify requests whose actions are in unfavorable positions in their tours by comparing historical success values. For every pair of actions $(a_i, a_j) \in A^2$, $c_{min}(a_i, a_j)$ represents the minimum costs of all found solutions in which a vehicle executed a_j directly after a_i. The c_{min} values are initialized with infinity. Every iteration, the values of all pairs of actions in the current solution have to be updated. By that, for each request r_i a score $c_{sum}(r_i)$ can be determined. The actions of r_i are currently inserted in a tour $T = (a_{\pi_1}, \ldots, a_{\pi_k})$ at the positions p and d (w.l.o.g. $1 < p < d < k$). The relevant c_{min} values for the score are those of the action pairs in T, i.e. $c_{min}(a_{\pi_{p-1}}, a_{\pi_p})$, $c_{min}(a_{\pi_p}, a_{\pi_{p+1}})$, $c_{min}(a_{\pi_{d-1}}, a_{\pi_d})$ and $c_{min}(a_{\pi_d}, a_{\pi_{d+1}})$. All of them are necessarily less than infinity, because there is at least one solution, namely the current solution, that contains these edges, so the c_{min} values have been reset at least once. Then, the score is calculated by

$$\begin{aligned} c_{sum}(r_i) = {} & c_{min}(a_{\pi_{p-1}}, a_{\pi_p}) + c_{min}(a_{\pi_p}, a_{\pi_{p+1}}) \\ & + c_{min}(a_{\pi_{d-1}}, a_{\pi_d}) + c_{min}(a_{\pi_d}, a_{\pi_{d+1}}) \end{aligned}$$

The requests with the largest scores are removed until the degree of destruction is reached. The outline of the heuristic is shown in Algorithm 7.
Still, the plausibility of the heuristic needs to be checked. The removed requests are the ones with the largest scores. A large score is caused by the c_{min} values, so someof them - if not all - are large as well. This means, that the best solution that has yet been found and that contains the same edges adjacent to the actions of the removed request has comparatively large costs. In other words, based on the success history it does not seem favorable to continue with the request inserted at the current position. Solutions that contain these four edges apparently tend to have high costs, so it is reasonable for the Historical Action Pair Removal heuristic to get rid of them.

Algorithm 7: Historical Action Pair Removal

Input: current solution s, set of unassigned requests R^-, degree of destruction d, updated historical success values $c_{sum}(r)$
for $i = 1, \dots, d$ **do**
  remove request $r' = \arg\max_{r \in R \setminus R^-} c_{sum}(r)$
return s

4.3 REPAIR HEURISTICS

The repair heuristics directly correspond to the concepts presented in Section 3.3.2. The explanations are therefore rather short. The general outline of both repair heuristics in presented in Algorithm 8.

Algorithm 8: Repair Heuristic

Input: current solution s, set of unassigned requests R^-
calculate insertion costs of request r into the tour of vehicle v $\forall r \in R^-, v \in V$
while further insertions are possible **do**
  rank request-vehicle pairs (r, v) according to insertion costs
  determine top-ranked request-vehicle pair (r, v)
  insert r into the tour of v in s
  $R^- = R^- \setminus \{r\}$
  update insertion costs of (r', v) $\forall r' \in R^-$
return s

4.3.1 *Basic Greedy Repair*

Basic Greedy Repair determines the cheapest insertion position for all currently unserved requests. The tours of all vehicles are taken into account for the calculation. The ranking in Algorithm 8 is therefore determined by the insertion costs of a request r into the tour of a vehicle v. The lower the insertion costs are, the higher is the pair (r, v) ranked.

4.3.2 *Regret Repair*

Regret Repair takes an approach that is similar to Basic Greedy Repair. But instead of inserting the requests at the cheapest position, a Regret-n heuristic calculates a regret value based on $s_i(r)$, a modification of the current solution s, in which request r has been inserted in the i-cheapest tour. The next insertion candidate is then determined by

$$\arg\max_{r \in R^-} \left\{ \sum_{i=2}^{n} (c(s_i(r)) - c(s_1(r))) \right\}$$

Here, R^- stands for the set of currently unassigned requests. The advantage of Regret Repair heuristics is that they recognize the shortage of suitable insertion possibilities for a request earlier than Basic Greedy Repair and are able

to prepone the insertion. If a request can only be inserted in $j < n$ tours, the regret value can not be calculated. Instead, the request with the lowest number of tours that it can be inserted in is preferred. If multiple requests are tied, the one with the lower regret value based on the j tours is chosen. The described lexicographical order determines the ranking in Algorithm 8.
Every Regret Repair heuristic spends a considerable amount of time checking if insertions are possible. Therefore, that procedure has to be optimized. A possible method is to update the earliest and latest possible starting times of service e_i and l_i whenever a removal or insertion occurs. The computing time for that procedure is linear in the tour length. Still, it saves a lot of time in the insertion checking process. There, for every currently unserved request every possible insertion position in every tour is checked. Without the earliest and latest possible starting times of the service, the tours would always have to be calculated from the start until the end. Most of that can actually be omitted. The insertion of a request into a route consists of the insertion of the pickup and the delivery action. For the pickup action, the insertion check can be done in constant time by using e_i and l_i. In general, the insertion of an action a_{π_i} between $a_{\pi_{i-1}}$ and $a_{\pi_{i+1}}$ is possible, if there is enough time after the service of $a_{\pi_{i-1}}$ to visit a_{π_i} and arrive at $a_{\pi_{i+1}}$ before the latest possible starting time of the service $l_{\pi_{i+1}}$:

$$e_{\pi_{i-1}} + s_{\pi_{i-1}} + d_{\pi_{i-1},\pi_i} + t^w_{\pi_i} + s_{\pi_i} + d_{\pi_i,\pi_{i+1}} < l_{\pi_{i+1}}?$$

t^w_i is the potential waiting time at a_i. Waiting time occurs if $e_{\pi_{i-1}} + s_{\pi_{i-1}} + d_{\pi_{i-1},\pi_i} < e_{\pi_i}$, i.e. if the arrival at a_{π_i} happens before earliest possible starting time of the service. If the insertion of the pickup action is possible, the insertion of the delivery action has to be checked. But it is not sufficient to perform the same check on the insertion position of the delivery, because the earliest possible starting times of the service of the actions following a_{π_i} are possibly changed by the insertion. Therefore, the part of the tour between the insertion positions has to be traversed. If at one point, the arrival time is earlier than the earliest starting time of the service, the calculation can be stopped, because no further changes of the e_i values are possible. Then, the check of the insertion of the delivery action can be executed as explained above. Otherwise, the check is performed using the calculated modified earliest starting time of the service.
Finally, the advantage of the described method compared to the naive calculation should be discussed. The computing time of the naive version is linear in the tour length which is denoted by k. More importantly, it always takes k steps to get the result. The method using e_i and l_i needs at worst $j - i + 1$ steps, if the delivery insertion position is j and the pickup insertion position is i. In some cases, it needs even less, because the modification of the e_i values can be aborted early as explained above. All in all, in the worst case it still takes k steps to check if the insertion is possible. But the number of times the different insertion positions i and j $(i < j)$ occur is also relevant:

difference	occurrences	(i, j)
1	$k-3$	$(2,3),(3,4),\ldots(k-2,k-1)$
2	$k-4$	$(2,4),(3,5),\ldots,(k-3,k-1)$
$\vdots$	$\vdots$	$\vdots$
$k-3$	1	$(2,k-1)$

It is worth mentioning that $2 \leqslant i < j \leqslant k-1$ because the tour T is assumed to be constructed as described in Chapter 2: $T = (a_{\pi_1}, \ldots, a_{\pi_k})$ with vehicle start action a_{π_1} and vehicle end action a_{π_k}. So on average, the difference $j - i$ is

$$\frac{\sum_{q=1}^{k-3} q \cdot (k-2-q)}{\sum_{q=1}^{k-3} q} = \frac{\frac{1}{6}(k-1)(k-2)(k-3)}{\frac{(k-3)(k-2)}{2}} = \frac{1}{3}(k-1)$$

As a result, the optimized version is still linear in the tour length, albeit with a lower constant. Additionally, the updates of the e_i and l_i values have to be calculated once.

4.4 INITIAL SOLUTION CONSTRUCTION

There are several possible methods to create an initial solution. Hosny and Mumford [13] have presented different construction algorithms based on hill climbing for the multiple vehicle pickup and delivery problem with time windows. The basic principle behind it is the exchange of actions inside a route if it improves the solution costs. Unfortunately, the algorithm does not really translate well to the RPDPTW. In our experiments, it did not find initial solutions in which all requests are assigned to tours for most problem instances. On top of that, it was a very time-consuming algorithm compared to the following alternatives.

Ropke and Pisinger [32] have compared different Regret Heuristics that work similar to the Regret Heuristics presented in Section 4.3.2. The difference is that for the initial solution construction all requests have to be inserted instead of inserting a few requests into an already existing solution. For that, the Regret Repair heuristic is repeated until all requests are assigned or no further insertions are possible. Alternatively, Basic Greedy Repair can be used instead of Regret Repair.

The third approach in Algorithm 9 uses a strategy similar to the Related Removal Heuristics. These remove pairs of requests from a solution if they are in some sense related. The relatedness rel of two requests can be based on the distances between their actions or the differences between their time window opening and closing times. The heuristic calculates relatedness values for all pairs (v, r) of a vehicle $v \in V$ and a request $r \in R$ based on the distances, i.e.

$$\mathrm{rel}(v_j, r_i) = d_{2n+j,i} + d_{n+i,2n+m+j}$$

This means that the distance between vehicle start position and pickup location was added to the distance between delivery location and the vehicle end

position. The lower these relatedness values are, the more related are v and r. Afterwards, the vehicles are sequentially assigned their most related request, i.e. arg $\min_{r \in R} rel(v, r)$, that has not yet been assigned to a tour. The order of the actions within a tour is the first feasible order found by checking all options and therefore not necessarily the optimal one. This process continues until all requests are assigned or no more insertions are possible. R^- is the set of currently unassigned requests. R_i is the set of requests that have not yet been tried to insert into the tour of vehicle v_i. The differentiation between the sets of requests R_i and R^- is necessary to avoid an infinite loop. If, for example, the most highly related request of a vehicle can not be inserted into its tour, the vehicle would not be able to serve further requests because it would try to insert the request over and over again. If such a case happens, the Related Construction removes the request from the candidate list R_i of the vehicle. By that, it enables the vehicle to continue with other requests in the next iteration.

Algorithm 9: Related Construction

Input: problem instance I, empty solution s, sets of possible requests for vehicles $Poss'(v)$ $\forall v \in V$
calculate relatedness values $rel(v, r) \quad \forall v \in V, r \in R$
$R_i = Poss'(v_i) \quad \forall i = 1, \ldots, |V|$
$R^- = R$
while further insertions are possible **do**
  for $i = 1, \ldots, |V|$ **do**
    $r_{ins} = \arg\min_{r \in R_i \cap R^-} rel(v_i, r)$
    if insertion is possible **then**
      insert request r_{ins} into the tour T_i of vehicle v_i
      $R^- = R^- \setminus \{r_{ins}\}$
    $R_i = R_i \setminus \{r_{ins}\}$
return s

The related initial solution construction has to my knowledge never been used before in any paper. The success of both the Regret and the Related Construction are compared in Section 5.1.

4.5 RANDOMIZATION AND NOISING

It is worth mentioning that apart from Random Removal all destroy and repair heuristics can additionally include randomization. At some point, the requests that are candidates for a removal or insertion are ranked and the first ranked request is chosen. This process can be randomized with constant probabilities, e.g. the first ranked request is chosen with a probability of 80%, the second with 15% and the third with 5%. Alternatively, Ropke and Pisinger [32] suggest the introduction of a randomization parameter $p \geqslant 1$. For every selection a random number $x \in [0, 1)$ is determined. With the sorted list of l candidates, the $(1 + \lfloor x^p \cdot l \rfloor)$-th request in the ranking is selected. The higher p is set, the more are the selections based on the ranking, because x^p approaches

zero with a large p. Another possibility would be to manipulate the values that lead to the ranking by adding a noise term. This method was also proposed by Ropke and Pisinger [29]. Instead of the removal or insertion costs that were originally compared, a modified cost function f' is used. The original costs are denoted by f here. For a request $r \in R$, f' is calculated by $f'(r) = \max\{0, f(r) + \tau\}$. $\tau \in [-\sigma, \sigma]$ is the randomly chosen noise term. σ has to be chosen with great care. A large σ can influence the function f' and therefore the selection heavily, while a very low σ does not change anything compared to the original selection. As a consequence, σ has to be chosen relatively to f. For example, if f includes the distances in the solution, σ could be chosen as $\sigma = \eta \cdot \max_{i,j}\{d_{i,j}\}$ with η as the noise control parameter. The concept can be applied to most of the heuristics, e.g. Worst Removal with $f(r) = c(s) - c(s^{-r})$ or Greedy Repair with $f(r) = c(s^{+r}) - c(s)$. This method is also used in the implementation, albeit only in the repair heuristics.

Still, it can not be guaranteed that randomization and noising are beneficial to the solution of the algorithm. But in most cases, "a diversification operator at the master level is not sufficient" [29], i.e. it is not enough to use an acceptance method like Simulated Annealing that can also accept worse solutions than the current one. Therefore, randomization or noising should be included in the destroy or repair heuristics.

The use of noise and randomization may not always produce the best immediate results, but there is a chance that it opens up the possibility for the algorithm to explore parts of the search space that have previously not been searched. But the weight adjustments are only influenced by the success of the solutions that were directly created by a heuristic. If a heuristic changes an important part of the solution, thereby increasing the costs, and by that paves the way for substantial improvements, it is not given credit for that. Therefore, it has to be deliberated about whether randomized heuristics should not be put in the pool of heuristics like every other heuristic. Alternatively, the selection of a heuristic can become a two-stage procedure. Firstly, a heuristic is chosen from the non-randomized ones as before. Afterwards, it is decided if the heuristic should be used in the randomized version or in the normal way. For the weight adjustments, nothing changes, because both versions of a heuristic are treated as one sharing each others' success. The decision for or against randomization can also be based on the past success or equiprobable selection. By that, diversification in the algorithm is likely to survive the early stages so that it can contribute later.

In the implementation, we use the two-stage approach for the repair heuristics. The destroy heuristics do not include noise, because there are already the Random Removal and Randomized Related Removal heuristics which include a considerable amount of randomization. In several experiments, we found that some of them perform well enough in order to be used in later iterations. Therefore, the two-stage approach is not necessary.

5 IMPLEMENTATION AND EXPERIMENTAL EVALUATION

This chapter contains details on the implementation and the obtained experimental results. It starts with the construction of the initial solution in Section 5.1. Afterwards, the heuristics that were used in the implementation are specified. Furthermore, the test instances and their properties are examined in Section 5.3. In Section 5.4 the tuning of the input parameters of the ALNS algorithm and its results are described. Finally, Section 5.5 takes a look at the way the experiments were conducted and the experimental results are presented and compared to the results from other sources.

5.1 INITIAL SOLUTION CONSTRUCTION

First of all, the best Greedy or Regret Construction heuristic was determined in order to compare it to the Related Construction afterwards. Therefore, the results of Regret-2, Regret-3, Regret-4, Regret-5 and the Greedy Construction on all instances were compared in Table 15 in the appendix. There is no clear-cut winner on all instances, but Greedy Construction had the lowest solution costs 42 out of 48 times. So we chose Greedy Construction for the comparison to the Related Construction. Greedy Construction was especially successful for the larger instances with 250 or 500 requests.

Both methods were able to find initial solutions in which all requests were served for all problem instances. The comparison of the results of the methods can be found in the appendix in table 14. Figure 12 illustrates the costs of the solutions created by the two construction approaches. In general, the Greedy Construction produced solutions with lower costs while the Related Construction was several times faster. This does not necessarily mean that the Greedy Construction should always be used. While Related Construction needed less than 0.2 seconds on all instances, Greedy Construction took up to 5.8 seconds and around 60 times as much computing time. In that time, the ALNS algorithm could have executed hundreds of iterations, dependent on the number of requests, vehicles and the degree of destruction. The costs of the thereby produced solution might be significantly lower than the costs of the solution of the Greedy Construction.

All in all, both construction approaches are worth using. For the experiments and tuning, the Related Construction has been used because the focus was rather on testing the ALNS algorithm instead of spending time to find a good initial solution.

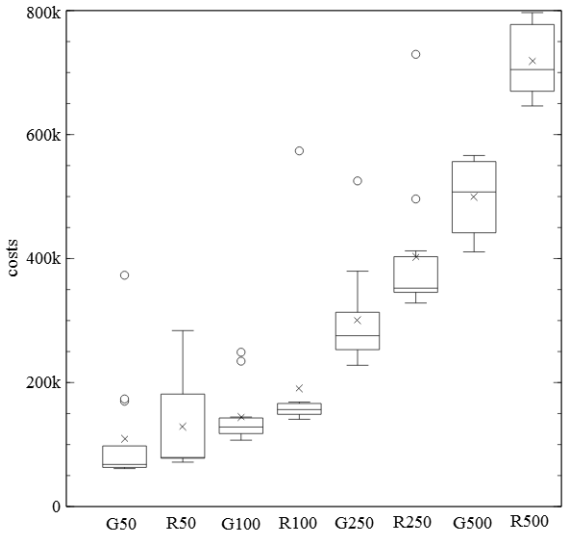


Figure 12: The figure shows the costs of the solutions created by Greedy Construction and Related Construction in a boxplot. It is split into the different instance sizes 50, 100, 250 and 500. The labels correspond to these sizes and the first letter of the used construction method

5.2 HEURISTICS

The previous chapter contained plenty of destroy heuristics and two repair heuristics. As a reminder, Table 3 contains a short list of twelve possible destroy heuristics. The randomized variants of the Related Removal heuristics are explicitly put into the pool of destroy heuristics from which one is selected in every iteration.

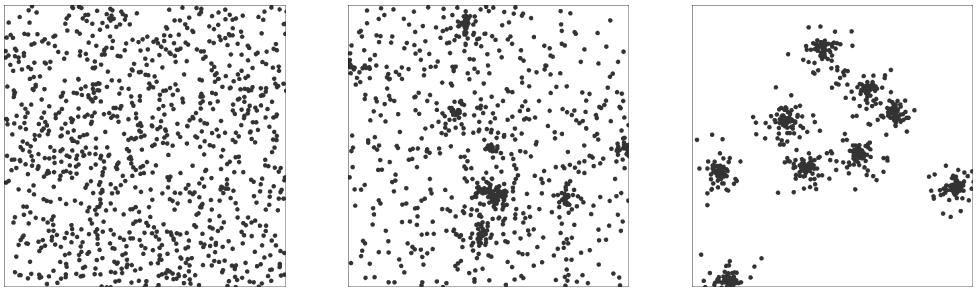
heuristic	normal version	extra randomized
Random Removal	✓	
Worst Removal	✓	
Distance-oriented Removal	✓	✓
Cluster Removal	✓	
Service time-oriented Removal	✓	✓
Shaw Removal	✓	✓
Historical Request Pair Removal	✓	✓
Historical Action Pair Removal	✓	

Table 3: An Overview of the destroy heuristics that were used in the implementation.

For the ALNS algorithm, the sheer number of heuristics might cause problems. If the length of the update period p_u is too short, some of the heuristics might not even get a chance to be executed in the beginning. As a consequence, their weights do not decrease because of bad performance but because of a lack of trials. In order to avoid that, the parameter p_u should be chosen adequately high. Ropke and Pisinger [29] suggest to take $p_u = 100$, but they worked with

a smaller number of heuristics. It might be reasonable to choose p_u dependent on the number of heuristics, especially with a number of heuristics that is considerably larger than 10. In the implementation with just 12 destroy heuristics, we also decided to set the length of the update period to 100 iterations.
The repair heuristics are easier to choose because there are only two different ones: The Basic Greedy Repair and the Regret Repair heuristics. Contrary to the destroy heuristics, there are no extra randomized versions of them. Instead, they use the two-stage approach mentioned in Section 4.5 to decide if noise is used or not. Still, several Regret Repair heuristics with different regret numbers can be included in the ALNS algorithm. A reasonable choice is to select less than five Regret heuristics. With larger regret numbers the heuristic is in danger of ranking the requests that should be inserted by the number of routes in which they can be inserted rather than the cost increase an insertion would bring. The Regret Repair heuristic was originally introduced in order to prepone the insertion of a request that has only very few possible insertion positions. If for example the regret number were chosen as 10, a request with 9 possible insertion positions would be preferred to another request with 10 insertion positions even if the regret value of the second request is by far the highest at the moment. This means that the difference between inserting the request at the best position and the next 9 best positions is large, i.e. it can cause an unnecessary cost increase not to insert the request. On top of that, there is not really a need to insert the request with 9 possible insertion positions immediately because it still has a few further options. This example illustrates that the regret number should not be chosen too high. For the tuning, it was limited to 5 so that 5 repair heuristics including Basic Greedy Repair and 4 Regret Repair heuristics are available.

5.3 INSTANCES



(a) uniform (b) semi-clustered (c) clustered

Figure 13: The three geographical distribution types of the test instances are illustrated by three examples with 500 requests, i.e. up to 1000 different locations. They are based on the data by Ropke [31]. Figure 13a shows the uniform distribution of instance 500A. The semi-clustered distribution of instance 500I is pictured in Figure 13b. Figure 13c contains the clustered distribution of instace 500H.

All the instances were taken from the homepage of Stefan Ropke [31]. Overall, there are four different instance sizes: 50, 100, 250 and 500 requests. For

instance	RT1	RT2	TT1	TT2	GDT1	GDT2	GDT3
A	✓		✓		✓		
B		✓	✓		✓		
C	✓			✓	✓		
D		✓		✓	✓		
E	✓		✓				✓
F		✓	✓				✓
G	✓			✓			✓
H		✓		✓			✓
I	✓		✓			✓	
J		✓	✓			✓	
K	✓			✓		✓	
L		✓		✓		✓	

Table 4: The characteristics of the problem instances. The instances end with one of the letters from A to L which shows their characteristics. The abbreviations RT, TT and GDT stand for request type, tour type and geographical distribution type. The description of the types can be found in Section 5.3.

each of the sizes, there are 12 different instances. Furthermore, the instances can be differentiated because of their characteristics. Ropke and Pisinger [32] describe the generation as follows: each instance was created with one of two tour types (TT), one of two request types (RT) and one of three geographical distribution types (GDT). The 12 instances of each size result from that choice which produces $2 \cdot 2 \cdot 3 = 12$ possibilities. The tours were either constructed so that each vehicle has an identical start and end position (TT1) or each vehicle has a different start and end position (TT2). The requests can either be served by all vehicles (RT1) or 50% of the requests are restricted to $30 - 60\%$ of the vehicles (RT2). The geographical distribution types include a uniform distribution (GDT1), a semi-clustered distribution (GDT2) and a clustered distribution (GDT3). Examples of the geographical distribution types are illustrated in Figure 13.
An overview of the instances and their individual characteristics is presented in Table 4. For the tuning, the instances were considered as representative of RPDPTW instances. Including all instances would have cost a considerable amount of time, so the set of tuning instances consisted only of the instances with 50 and 100 requests. For the experiments, all instances were used.

5.4 PARAMETER TUNING

In order to achieve good results, the input parameters of the algorithm have to be tuned. Doing this by hand would require extensive statistical evaluation and a lot of time. Therefore, we used the irace package which offers these services without additional effort for the user. After a short outline of iterated racing method used by irace, we describe the tuned parameters.

5.4.1 *Iterated racing*

The parameters were tuned with the irace package which can be downloaded from CRAN [24]. The information about irace is based on López-Ibánez, Dubois-Lacoste, Stützle and Birattari[25].
irace is an abbreviation of iterated racing and can be used to tune input parameters of an algorithm. It takes a program, the domains of the parameters and a set of representative instances and tries to find a configuration as close to optimal as possible. The initial budget, i.e. the maximum number of runs, is divided into a number of iterations. Each time an iteration starts, candidate parameter configurations are sampled. Initially, this is done with an uniform distribution. Later, the sampled configurations are chosen dependent on the elite candidates of the previous iteration. The number of candidates decreases with the number of iterations. A larger number of executions per candidate means that the configurations can be tested more extensively. The program is executed with these candidate configurations on a number of instances. Afterwards, irace compares the success with a statistical test. For ALNS, the success is measured by the solution costs. Dependent on the statistical analysis, some of the parameter configurations are discarded. Still, it is not always necessary or reasonable to reject candidates, so irace is not forced to do it after every statistical test. The procedure is repeated with the remaining candidates and instances until the remaining budget exhausted or less than a minimal number of candidates are left.

5.4.2 *Parameters*

The ALNS algorithm has quite a lot of tunable input parameters. Tuning them all together would require a huge number of executions. Therefore, the parameters were grouped and only one of the groups was tuned while the parameter values of the other groups were kept fixed. The groups were formed so that parameters which influence each other strongly are in the same group. The constant values were set similar to the results by Ropke and Pisinger [32] and are shown in Table 5. The tunings had budget of 200000 executions using the 24 instances with 50 and 100 requests respectively. The different groups of parameters are described in the following sections.

repair heuristics	η	r_1	r_2	r_3	ρ	d	d_{min}	$d_{max} - d_{min}$
3	0.15	70	50	70	0.1	random	0.1	0.4

Table 5: The constant values of the first tuning. η is the noise parameter, the r_i values are necessary for the rewards of the weight adjustments and ρ is the reaction factor. The degree of destruction d was randomly selected from the interval $[d_{min}, d_{max}]$.

parameter	accept	tol	ϕ	ψ
range	RW, GA, SA, TA, OBA, GDA	$[0, 0.3]$	$[0.9, 0.99999]$	$[1.00001, 1.1]$
result	TA	0.01	0.9997	-

Table 6: The configuration and results of the first tuning. tol is the acceptance tolerance, ϕ is the reduction factor and ψ is the increase factor. It is only used by OBA and is therefore excluded from the results.

5.4.2.1 *Acceptance Method Parameters*

In Section 3.3.1, six different acceptance methods were presented: Random Walk (RW), Greedy Acceptance (GA), Simulated Annealing (SA), Threshold Acceptance (TA), Old Bachelor Acceptance (OBA) and the Great Deluge Algorithm (GDA). All of them were included in the tuning process. RW and GA do not need any input parameters, but the other four require at least an initial threshold, level or temperature and a reduction factor ϕ. Furthermore, OBA needs an additional increase factor ψ. The reduction and increase factors' domains were set to $[0.9, 0.99999]$ and $[1.00001, 1.1]$ respectively. A lower reduction factor would result in the heuristics working like the Greedy Acceptance, i.e. only accepting improving solutions, after only few iterations. The acceptance of non-improving solutions is very important and a reduction factor close to one makes sure that this acceptance method characteristic is not quickly discarded. For instance, Ropke and Pisinger [32] recommended taking $\phi = 0.99975$. On the other hand, the probability to accept deteriorating solutions when using OBA should not increase too fast if solutions are not accepted. Therefore, the domain of the increase factor was also chosen very close to one. The threshold, level or temperature (from now on called T) should not be chosen as a constant value. It is essential to find a value for T that is adequate for the specific problem instance. Otherwise, T is either too large, which causes the acceptance of solutions with exorbitant costs, or too small, which makes it unlikely that any non-improving solutions are accepted. Both cases are not desirable and should be avoided. Therefore, the tolerance parameter tol is introduced. At this point, it has to be distinguished between the probabilistic SA and the non-probabilistic TA, OBA and GDA.

For Simulated Annealing, the initial T is set so that the probability to accept a solution that is tol percent worse than the initial solution s_{init} is 50%. The probability is normally calculated by $exp(\frac{c(s)-c(s')}{T})$, where s is the current solution and s' is the recently created one. By setting $0.5 = exp(\frac{-c(s_{init})\cdot tol}{T})$ and solving the equation, T is identified as $T = \frac{c(s_{init})\cdot tol}{\log 2}$.

For TA, OBA and GDA, T is set so that the initially worst acceptable solution has tol percent higher costs than the initial solution, i.e. $T = tol \cdot c(s_{init})$.

An overview of the parameter ranges and the results is shown in Table 6.

The results of the tuning were as follows: the best solutions of the tuning were found with Threshold Acceptance, so for all further tunings, TA was used. The best values for the reduction factor ϕ were between 0.9996 and 0.9998, so we choose 0.9997. The increase factor ψ is not used by Threshold Acceptance, so it

did not occur in the results. The tolerance tol varied from 0.0075 to 0.0142. A value around one percent seems appropriate, so tol was fixed to 0.01.
For the tuning, it probably would have been better to define separate tolerance parameters and reduction factors for each of the acceptance methods. Otherwise, the acceptance methods also influence the parameters which are used by the other methods. Still, the differences between the acceptance methods are rather small, so it does not affect the algorithm much.

5.4.2.2 *Weight Adjustment Parameters*

There are also several parameters that influence the weight adjustments. The initial weights can be set to an arbitrary value. They are used for the calculation of the probability to take a specific heuristic. In order to avoid precision problems with floating point numbers close to zero, we set the initial weights to 1000. The choice of the length of the update period has already been discussed in Section 5.2. It is set to 100 in all experiments.
The weight adjustment parameters that are actually tuned are the rewards δ_1, δ_2 and δ_3 and the reaction factor ρ. They have all been described in Section 3.4.1. For the rewards, it is assumed that $\delta_1 \geqslant \delta_2 \geqslant \delta_3$, because the rewards should be merit-based. Finding a new global best solution is obviously more difficult than finding an arbitrary improving solution. A new non-improving, but accepted solution can also be valued in order to encourage diversification. Therefore, the program has the input parameters r_1, r_2 and r_3 which define the actual rewards as follows: $\delta_3 = r_3$, $\delta_2 = r_3 + r_2$, $\delta_1 = r_3 + r_2 + r_1$. This ensures the validity of the inequation above. Ropke and Pisinger suggested using rewards between 9 and 33. But the implementation used for this thesis uses a few more heuristics. Therefore, it is possible that larger rewards should be chosen, because the average number of times a heuristic is used decreases with the total number of heuristics. As a consequence, the total amount of rewards is lower. Assuming the reaction factor ρ is approximately equal to their experiment, the weights decrease faster. This can cause a problem because the rewards are meant to be able to influence the weights considerably. Otherwise, the whole concept of the rewards is useless. Because of this assumption, the input parameters r_1, r_2 and r_3 were given rather large domains from 0 to 100.
The reaction factor ρ decides how much the previous weights and the success in the last update period influence the adjusted weights. In order not to neglect the past success of a heuristic, the domain of ρ for the tuning was chosen as $[0, 0.5]$ so that the factor of the previous weights is greater than or equal to 0.5. The parameter ranges and the tuning results are presented in Table 7.
The tuning results were as follows: $r_1 \in [59, 77]$, $r_2 \in [28, 55]$, $r_3 \in [23, 37]$. Be-

parameter	r_1	r_2	r_3	ρ
range	$[0, 100]$	$[0, 100]$	$[0, 100]$	$[0, 0.5]$
result	65	45	25	0.35

Table 7: The configuration and results of the second tuning. The r_i values contribute to the rewards of the weight adjustments. ρ is the reaction factor.

cause of the broad intervals, the choice was made based on the configuration with the most success, such that $r_1 = 65$, $r_2 = 45$, $r_3 = 25$, i.e. $\delta_1 = 135$, $\delta_2 = 70$ and $\delta_3 = 25$. The reaction factor of the most successful configuration was at 0.35.

5.4.2.3 *Degree of Destruction Parameters*

parameter	d	d_{min}	$d_{max} - d_{min}$
range	random, inc., decr., hist., const.	$[0.05, 0.1]$	$[0.1, 0.4]$
result	random	0.075	0.2

Table 8: The configuration and results of the third tuning. The degree of destruction d is selected from the interval $[d_{min}, d_{max}]$ with one of the given methods.

Different ways of controlling the degree of destruction were explained in Section 3.3.2. All of them were included in the tuning. The options were to have a constant, a decreasing, an increasing, a randomized or a history-based degree of destruction. The constant version requires only one value, but the others need an interval $[d_{min}, d_{max}]$ from which the degree of destruction is taken. Multiple experiments of our implementation in advance have shown that the degree of destruction should not be too large, because the runtime of the repair heuristics increases dramatically. Furthermore, the repair heuristics are not built for constructing a whole solution from scratch because one of them is actually a greedy heuristic and the others are similar to it. Therefore, the degree of destruction should always be well below $0.5 \cdot n$, where n is the number of requests. If n itself is a very large number, e.g. 1000, even $0.3 \cdot n$ can pose a huge problem for the selected repair heuristic. In order to counteract that problem, Ropke and Pisinger [29] suggest taking the degree of destruction from the interval $[\min\{0.1 \cdot n, 30\}, \min\{0.4 \cdot n, 60\}]$. Thereby, the interval is $[0.1 \cdot n, 0.4 \cdot n]$ for instances with a low number of requests and it is $[30, 60]$ for larger instances. For the instances in the tuning, n is either 50 or 100, so the former alternative would be chosen.
For the tuning, the domain of the lower bound of the interval was set to $[0.05, 0.1]$ due to several experiments in advance which favored a rather small degree of destruction. The range of the total size of the interval, i.e. $d_{max} - d_{min}$, varied from 0.1 to 0.4. An overview of the parameter domains and the tuning results is shown in Table 8. The best results were achieved with a randomized degree of destruction in the interval $[0.075 \cdot n, 0.275 \cdot n]$.

5.4.2.4 *Repair Heuristic Parameters*

There are only two input parameters concerning the repair heuristics. First of all, the number of repair heuristics has to be decided. Apart from the Basic Greedy Repair heuristic, there can possibly be an arbitrary number of Regret Repair heuristics. For the tuning it was limited to a maximum of 4, so that 5 repair heuristics are available. The reasons have already been discussed in Section 5.2. It is worth mentioning that all Regret-n Repair heuristics with $n \leqslant$

parameter	repair heuristics	η
range	$\{1, 2, 3, 4, 5\}$	$[0, 0.5]$
result	3	0.43

Table 9: The configuration and results of the fourth tuning. η is the noise parameter.

n_{max} are included in a run if n_{max} is selected as the number of heuristics.
Furthermore, the noise parameter η for the repair heuristics has to be set. The use of η is explained in Section 4.5. Ropke and Pisinger determined a noise value of 0.025. For the tuning, any values from 0 to 0.5 were allowed. The lower η is set, the less noise is used. The noise, i.e. a random number from an interval whose length is influenced by η, is added to the cost increase values in the repair heuristics. In the special case $\eta = 0$, no noise is used at all.
The parameter ranges and the results of the tuning are presented in Table 9.
The tuning resulted in the choice of three repair heuristics, namely Basic Greedy Repair, Regret-2 Repair and Regret-3 Repair. Interestingly, a large amount of noise achieved the best solutions. The three best configurations had a η value of 0.4238, 0.4268 and 0.4700. As a result, η was set to 0.43 for the experiments.

5.4.2.5 *Other Parameters*

Apart from the so far presented parameters, there are a few others that can be but do not necessarily have to be tuned. The parameters α, β and γ for the calculation of the solution costs were already set in the problem instances. The general configuration in all instances was $(\alpha, \beta, \gamma) = (1, 1, 100000)$ in order to make the service of all requests the highest priority.

MINIMIZING THE NUMBER OF VEHICLES First of all, there is the factor δ which influences the solution costs as specified in Chapter 2. The factors α, β and γ are already given for every problem instance. But in some real life scenarios the number of vehicles that are used in a solution also plays a major if not decisive role. Then, δ has to be set properly, i.e. high enough to make the algorithm try to minimize the number of vehicles but low enough to make sure all requests are still served.
The instances include heterogeneous fleets of vehicles, which would make it quite difficult for the algorithm to minimize the number of vehicles. Some of the requests can only be served by a subset of all vehicles. Ropke and Pisinger [29] suggest splitting up the procedure into a two-stage algorithm that minimizes the number of vehicles first by repeatedly removing vehicles until no solution can be found anymore. With the resulting number of vehicles they try to minimize the solution costs based on distance, duration and number of served requests. Unfortunately, they have not applied that approach to their own instances because of the heterogeneity of the fleet of vehicles, so there are no comparable results. Because of that, this thesis does not include special tuning or experiments for the vehicle minimization.

PARAMETERS OF DESTROY HEURISTICS Historical Request Pair Removal has a single parameter that determines the number of top solutions that are memorized. The historical information of the heuristic is then taken from these top solutions. In the implementation, the hundred best solutions are memorized and the heuristic is used for the first time after 500 iterations, i.e. the best 20% of the solutions have contributed to the historical information at that point. All in all, the number of memorized top solutions is only a parameter of minor importance for the success of the whole algorithm and is therefore excluded from the tuning.
The same holds true for the parameters of the Shaw Removal heuristic. The relatedness values are calculated using the parameters λ, μ, ν and ξ to assign more or less importance to distance, time, demand and the set of possibly serving vehicles. The values that were used for the experiments were taken from Ropke and Pisinger [32]. They determined $(\lambda, \mu, \nu, \xi) = (9, 3, 2, 5)$ in their tuning. This is a reasonable choice if we take a look at the goal of Shaw Removal. It tries to identify related requests that can possibly be exchanged.

5.5 COMPUTATIONAL EXPERIMENTS

Some of the tables that are described in this section are too large and were therefore put in the appendix.
The implementation was coded in Java. It was tuned and tested on a computer with two Intel® Xeon® CPU E5-2680 v2 @ 2.80GHz with a total of 20 cores and 128 GB RAM. The test instances are the ones created by Stefan Ropke and available on his homepage [31]. The parameter configuration was exactly as described in Section 5.4. As a result, a general setting was used instead of specifically tuned configurations for every instance. We want to confirm that ALNS is a general heuristic that works good for different kinds of instances. Therefore, the general parameter configuration has to be applied on every instance.
The ALNS program was run 100 times on every instance. The solution costs, distances and durations as well as the runtime were measured for every run. Table 16 in the appendix contains both the best and the average results. Interestingly, the tour durations sometimes exceed three times the tour distance. The durations consist of the distances, the service times and waiting times. This in-

number of requests	runtime in seconds	
	Lutz	Ropke and Pisinger
50	2.9	2.2
100	13.7	8.1
250	156.0	47.8
500	764.1	122.7

Table 10: Comparison of the average runtimes per instance size of both our own and Ropke and Pisinger's experiments. Their runtime was normalized because they used 25000 instead of 10000 iterations.

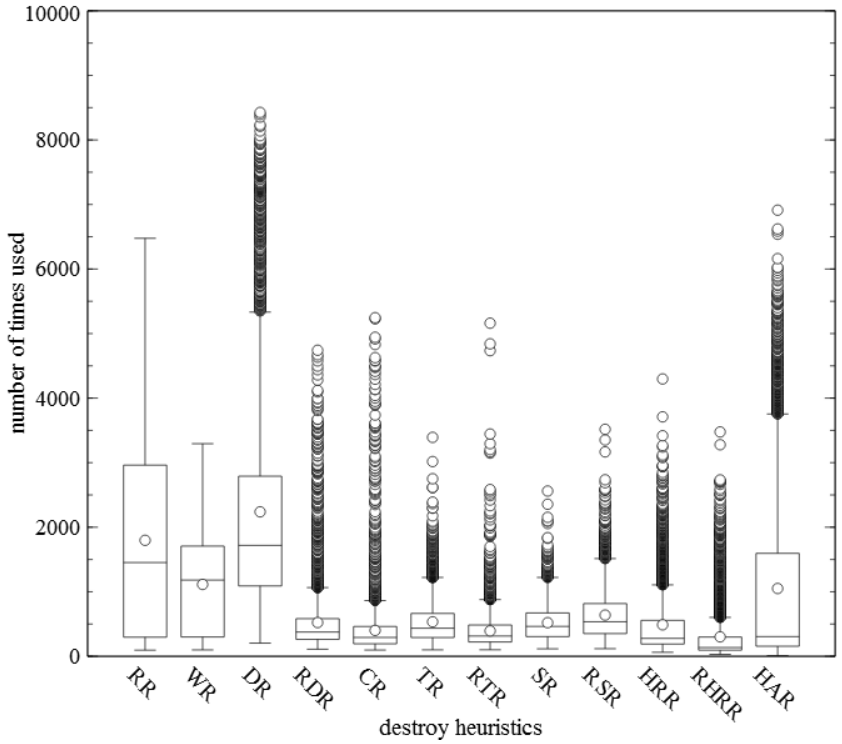


Figure 14: The number of times the destroy heuristics were used. The chart illustrates the data of all 4800 runs on the 48 instances in a boxplot. The heuristics are Random Removal (RR), Worst Removal (WR), Distance-oriented Removal (DR), Randomized DR (RDR), Cluster Removal (CR), Service time-oriented Removal (TR), Randomized TR (RTR), Shaw Removal (SR), Randomized SR (RSR), Historical Request Pair Removal (HRR), Randomized HRR (RHRR) and Historical Action Pair Removal (HAR).

dicates large waiting times caused by early arrivals. If the vehicles wait idly for actions, it is possibly a sign of underutilization.

The average runtimes per instance size of both Ropke and Pisinger's [31] and our own experiments are shown in Table 10. We have written our program in Java whereas Ropke and Pisinger have used C++. The large differences are most likely caused by the considerably larger degrees of destruction that are allowed in our implementation.

Moreover, we have analyzed the success of the heuristics. Figure 14 displayes a part of the data from Table 18 in the appendix in the form of a boxplot. Both show that there were huge differences between the destroy heuristics in the number of times they were called. Interestingly, the on average most frequently used destroy heuristic, Distance-oriented Removal, is not the most successful heuristic. In fact, less than 8% of the executions resulted in an accepted solution compared to the 31.3% of Worst Removal and the 28.6% of Shaw Removal. Furthermore, the number of times some of the heuristics were called was not within a relatively small interval but widespread.

A new feature of our implementation was the use of the Randomized Related Removal heuristics which were introduced in Algorithm 5. They were originally included in order to increase the degree of diversification in the implementation, hopefully even in later iterations of the algorithm. The randomized

versions of Distance-oriented Removal, Service time-oriented Removal and Historical Request Pair Removal had a considerably lower number of executions than their less randomized counterparts. Interestingly, Randomized Shaw Removal was even more successful in terms of the average number of calls than the original Shaw Removal.

6 DISCUSSION

In this final chapter, we want to take a look at ALNS from two different perspectives. First of all, we discuss the results of the experiments in the previous chapter. Thereafter, we recapitulate the success that ALNS has had so far. Finally, the thesis ends with a summary.

6.1 DISCUSSION OF COMPUTATIONAL EXPERIMENTS

We start our discussion with a comparison of our results to Ropke and Pisinger's. Afterwards, we analyze the success of the different destroy and repair heuristics as well as the evolution of their weights during the execution of the ALNS algorithm. At the end, we take a look at a typical execution.

6.1.1 *Comparison to Ropke and Pisinger*

The results in Section 5.5 show that our implementation achieved superior results to those by Ropke and Pisinger [32, 31]. Therefore, we describe the differences and interpret their influence on the success of the algorithm based on the results. An overview of the input parameters is presented in Table 11.

parameter	Lutz	Ropke and Pisinger
iterations	10000	25000
acceptance	Threshold Acceptance	Simulated Annealing
tol	0.01	0.05
ϕ	0.9997	0.99975
δ_1	135	33
δ_2	70	9
δ_3	25	13
ρ	0.35	0.1
d	random	random
d_{min}	$0.075n$	$\min\{30, 0.1n\}$
d_{max}	$0.275n$	$\min\{60, 0.4n\}$
η	0.43	0.025
repair heuristics	3	5

Table 11: Comparison of our parameter configuration with Ropke and Pisinger's [29]. Their previous configuration [32] contained slightly different values for tol, ϕ, d_{min} and d_{max}.

First of all, our tuning resulted in a relatively high reaction factor $\rho = 0.35$, contrary to the tuning by Ropke and Pisinger with $\rho = 0.1$. That difference probably has to do with the lower number of iterations in our implementation. The less iterations the algorithm has, the faster it has to get rid of poorly performing heuristics in order to achieve good results. As a consequence, the reaction factor is set higher. The negative side of it is that heuristics that tend to perform better in later iterations are practically excluded from the algorithm due to the drastic weight decreases caused by a high reaction factor. We will take a closer look at that problem in Section 6.1.2.
Furthermore, the upper limit for the degree of destruction that Ropke and Pisinger used was not installed for our experiments because the degree of destruction for the largest instance is randomly chosen from $[37, 137]$ instead of $[30, 60]$ with the limited version. This enables the destroy and repair heuristics to perform big changes, but also the smaller ones that are executed by the limited version. Still, for larger instances, e.g. with 1000 requests or more, the limit absolutely makes sense. Otherwise, the computation time for a single iteration becomes too large. The original motivation for Ropke and Pisinger for setting a limit was the low acceptance rate of solutions created with a large degree of destruction. In Section 6.1.2, we analyze the accepted solutions in greater detail and suggest a new approach for setting the degree of destruction.
Another parameter with a very different setting was the noise control parameter η. As a reminder, the repair heuristics can add a noise value of up to $\eta \cdot \max_{i,j} d_{ij}$ to their calculated cost increase and regret values respectively. The repair heuristics use a two-stage decision. Firstly, the repair heuristic is selected in the same way the destroy heuristics are selected. Secondly, the algorithm decides whether to include noise randomly based on the past success. The average tour length was approximately $\max_{i,j} d_{ij}$ in our experiments, so it is surprising that the tuning resulted in a noise value of as high as 0.43. This means that the noise can influence the repair heuristics heavily. Several experiments have shown that the noise is indeed used throughout the execution of the algorithm in $40 - 60\%$ of the iterations. Our theory was that the tuning somehow went wrong. To check this assumption, we tested the algorithm with lower noise values $\eta = 0.1$ and $\eta = 0$ ten times on all instances. The solutions with $\eta = 0.1$ were on average 0.07% worse, while the solutions with $\eta = 0$ were on average 0.1% better. Interestingly, the configuration with $\eta = 0.43$ had the best solution relative to the other two variants 43 out of 48 times and the lowest average solution costs only 10 times. While that high number of best solutions is definitely a result of the higher number of executions, a certain tendency is still visible. The average solution costs are lower with one of the low noise variants. Still, we assume that the large amount of noise with $\eta = 0.43$ - even though probably disadvantageous in earlier iterations - provides the algorithm with additional diversification in later iterations when it is more difficult to find accepted solutions. The use of the noise version in $40 - 60\%$ of the iterations shows that it at least did not impede the algorithm too much. All in all, it seems as if the noise control parameter does not influence the algorithm as much as we have originally thought.
Furthermore, we compared the results of the experiments with Ropke and Pisinger's [31] in Table 17 in the appendix. For that, the quality function q from

their paper [32] was used. It should be mentioned that apart from the different parameter configuration, they did not tune their ALNS implementation with the same instances, used 25000 iterations instead of our 10000 and ran their program 10 times on each instance compared to our 100 times. It is not surprising that our implementation had better results on the instances of 50 and 100 requests because it was tuned with them. But even with the larger instances, it has found better solutions on all but two instances. Ropke and Pisinger's average solution costs were lower than ours only four times out of 48. Over all 48 instances, our solutions had on average 14.04% lower costs than their best solution. It was also analyzed how good the two ALNS implementations work on specific kinds of instances in Table 12. Ropke and Pisinger [32] have found that clusters (GDT3), special requests (RT2) and tours with the same start and end position (TT1) make the most difficulties for ALNS. This tendency also holds true for our ALNS program. Still, the differences between the characteristics was far less than in the experiment by Ropke and Pisinger. Our results were on average between 13 and 15 percent better on the different geographical distribution types and tour types. But the request types show a different picture. With RT1, i.e. the requests can be served by all vehicles, the ALNS program of Ropke and Pisinger had only 10.95% higher costs. On the other hand, it had 17.13% higher costs with RT2. As a consequence, our implementation is obviously especially suited for solving problems with special requests. This is probably a result of both the tuning with 50% instances with specialized requests and the inclusion of a second Shaw Removal heuristic which explicitly takes that characteristic into account. Still, the average deviation from the best value was slightly lower in Ropke and Pisinger's experiments. This means that they have rather constant results over several runs.

6.1.2 *Destroy and Repair Heuristic Analysis*

In this section, we analyze the success of the destroy and repair heuristics, their performances with different kinds of problem instances and the developments of their weights.
We have already seen in Table 18 that there are large differences between the average numbers of times the destroy heuristics were used. Additionally, the success percentages of the most frequently used heuristics were not the highest. This effect has to do with the weight adjustments. Distance-oriented Removal tends to have success in the earlier iterations of the algorithm thus increasing the weight and the probability to be selected for further iterations. Even though it has only occasional success later, the weight is already higher than most of the others. As a consequence, Distance-oriented Removal is chosen again and again with little success. While Distance-oriented Removal may be a solid heuristic in the beginning, it should not be selected too often in later iterations because it is based on static data and it does not use information about the current solution. Similarly, Random Removal is meant for diversification in the early phase of the algorithm, but it was obviously used much more than that because of the early success. A method to change that typical sequence could be to explicitly

characteristic	Ropke and Pisinger	Lutz	comparison
GDT1	1.04%	3.19%	14.32%
GDT2	1.23%	3.47%	13.55%
GDT3	1.89%	3.69%	14.26%
TT1	1.59%	3.53%	15.08%
TT2	1.19%	3.38%	13.00%
RT1	1.24%	3.39%	10.95%
RT2	1.54%	3.52%	17.13%

Table 12: The influence of characteristics of the problem instance on the solution quality. The first column contains the different characteristics that were explained in Section 5.3. The second column consists of the data from Ropke and Pisinger [32]. We calculate the quality by $q = \frac{1}{|F|} \sum_{i \in F} \frac{c_{avg}(i) - c_{best}(i)}{c_{best}(i)}$, where F is the set of instances in which the characteristic occurs, $c_{avg}(i)$ are the average and $c_{best}(i)$ the best solution costs on instance i. In the categories, all instances of the specified type were used regardless of their size. We have done the same with our implementation in the third column. The fourth column was calculated similar to the second, but instead of the best solution costs of the same implementation, the average solution costs of our implementation were used. Therefore, the values show how many percent lower the solution costs in our implementation were.

remove mainly diversifying heuristics like Random Removal from the search after a certain number of iterations. Alternatively, the weights could be reset after a proper period of time. In the ensuing iterations, all heuristics would have another chance to be executed. Heuristics like Distance-oriented Removal, which have less success in later stages, would probably not be able to supplant the other heuristics. Still another possibility would be to change the weight adjustments so that a penalty is given to a heuristic if it does not find a solution that is accepted. Maybe it would even suffice to introduce minimum weights for all heuristics to sustain a certain minimum selection probability. This issue results from the assumption that past success indicates future success and will surely be the subject of further studies.

In our implementation we included four new Randomized Related Removal heuristics. Even though 3 of 4 of the heuristics had a lower average number of executions than their Related Removal counterparts, we still think they are an important contributor to the success of our implementation. Their inclusion is one of the major differences to Ropke and Pisinger's implementation which after all has been less productive on their set of instances.

Figure 14 shows that there were huge differences between the number of times the destroy heuristics were used. That does not only apply when comparing the heuristics, but also for a single heuristic on different runs and instances. The Distance-oriented Removal heuristic is a good example of that: it was used between 250 and 8500 times. This characteristic also applies to the other heuristics even though in a less distinctive way. As a consequence, the heuristics had varying success. By that, we can conclude that there is no generally favorable

heuristic for all kinds of instances. It is the main reason the ALNS heuristic is more successful than LNS which has been shown by Ropke and Pisinger [32]. Their idea was that including more heuristics is superior to working with the single best working heuristic only. The large differences in the number of times the heuristics were used confirm that.

Apart from that, Table 18 in the appendix shows that the heuristics did not handle the problem characteristics equally well. The success percentages should be treated with caution. For example, a large number of executions can cause the success rate to decrease because in later iterations it is much more difficult to find accepted solutions. In contrast, the heuristics with lower amounts of calls were mainly used at the beginning of the algorithm. As a result, their task was easier and their success rate is higher. Especially the geographical distribution types uniform (GDT1), semi-clustered (GDT2) and clustered (GDT3) posed problems for some destroy heuristics. For Random Removal, Worst Removal and Distance-oriented Removal, the uniform distribution type was the easiest and the clustered type was the most difficult. That can be concluded from the considerable differences between the average numbers of executions. For instance, Distance-oriented Removal had a third less calls with GDT3 than with GDT1. The problems that Distance-oriented Removal has with clusters arise from the static nature of the heuristic. It only removes the distance-related requests, i.e. some requests in the same cluster, without taking the current tours into account. Then, the repair heuristic probably reinserts the requests immediately in their original tour because the clusters within their tours were not completely removed. For a detailed description of the cluster problem, we refer to Section 4.2.3 where Cluster Removal is explained. All the other heuristics had more executions with the clustered version. Particularly Cluster Removal was successful with an increase of more than 70% more calls on clustered instances compared to the uniformly distributed instances. Naturally, Cluster Removal was built to deal with clusters, so this is not a surprising result. The history-based heuristics were also heavily influenced by the geographical distribution type. Their combined average number of executions was 1444.9 without and 2317.1 with clusters. This serves as a strong indicator that our history-based destroy methods work especially effective on clustered instances. Compared to the differences of the success of the heuristics on different geographical distribution types, the other problem types caused only minor differences. The only distinctive feature is that Distance-oriented Removal is rather suited for RT1, i.e. requests can be served by all vehicles, while Shaw Removal and Randomized Shaw Removal are appropriate for the special requests in RT2. This effect is not surprising as Shaw Removal takes into account by which vehicles a request can be served.

We have made similar observations with the results of the repair heuristics. Table 13 shows an excerpt of Table 19 in the appendix. Again, there is a seeming contradiction with the most frequently used heuristic having the least success. Still, the difference between the repair heuristics is not as large as between the destroy heuristics, so it is probably not necessary to invent countermeasures as described above. Interestingly, there was a striking difference between the average numbers of executions for all repair heuristics with the geographical distribution types. While Basic Greedy Repair had nearly 15% less executions

repair heuristic	r_1	r_2	r_3	total
Basic Greedy Repair	0.5	4.9	9.1	6167.4
Regret-2 Repair	0.5	5.7	12.5	2590.7
Regret-3 Repair	0.9	6.8	15.5	1241.8

Table 13: The success of the repair heuristics over all 4800 experiments. The last column contains the average number of times the heuristic was called over all experiments in which the characteristic occurred. The other three columns show how many percent of these calls resulted in a new global best solution (r_1), an improving (not global best) solution (r_2) and an accepted non-improving solution (r_3).

with clustered instances compared to the uniformly distributed instances, the Regret-3 Repair heuristic had approximately 58% more. All in all, the Basic Greedy Repair heuristic was by far the most frequently used, but it is surely not harmful to include a few other repair heuristics.

Figure 15 shows two examples of very different developments of the selection probabilities of the destroy heuristics. Both describe the probabilities in each of the 100 update periods of specific runs on instance prob500E. In the figure on the left, Worst Removal has a lot of success and prevails, while the weights and therefore the selection probabilities of the other heuristics decrease until there is basically only one heuristic involved in the search. Worst Removal is especially suited for the early iterations of the algorithm, so this development is not desired. The long plateaus in the chart, i.e. the periods where the border between the probabilities of the heuristics is horizontal, indicate that the heuristics do not have any success and their weights decrease evenly such that the probabilities do not change. In the optimal case, some of the heuristics that normally have success in later iterations should still be included in the search. They give the algorithm a better chance of improving the current solution. The figure on the right shows such a situation where after 5000 iterations nine destroy heuristics have probabilities of at least 4%. Even after 9000 iterations, four heuristics, namely Randomized Shaw Removal and the three history-based removal heuristics, are involved. The chart also includes plateaus, albeit significantly shorter ones. Neither of the two charts is representative of the probability development of all instances. Still, they show that the current weight adjustments are far from optimal, because the probabilities differ greatly even with the same instance. This is surely one of the reasons for the deviations presented in Table 12.

6.1.3 *Analysis of a Typical Execution*

The purpose of this section is to analyze a typical execution of our ALNS implementation. We illustrate several characteristics and try to identify areas with potential for improvements.

The experimental results in Table 16 in the appendix have already been presented in the previous chapter. Especially the difference between distance and

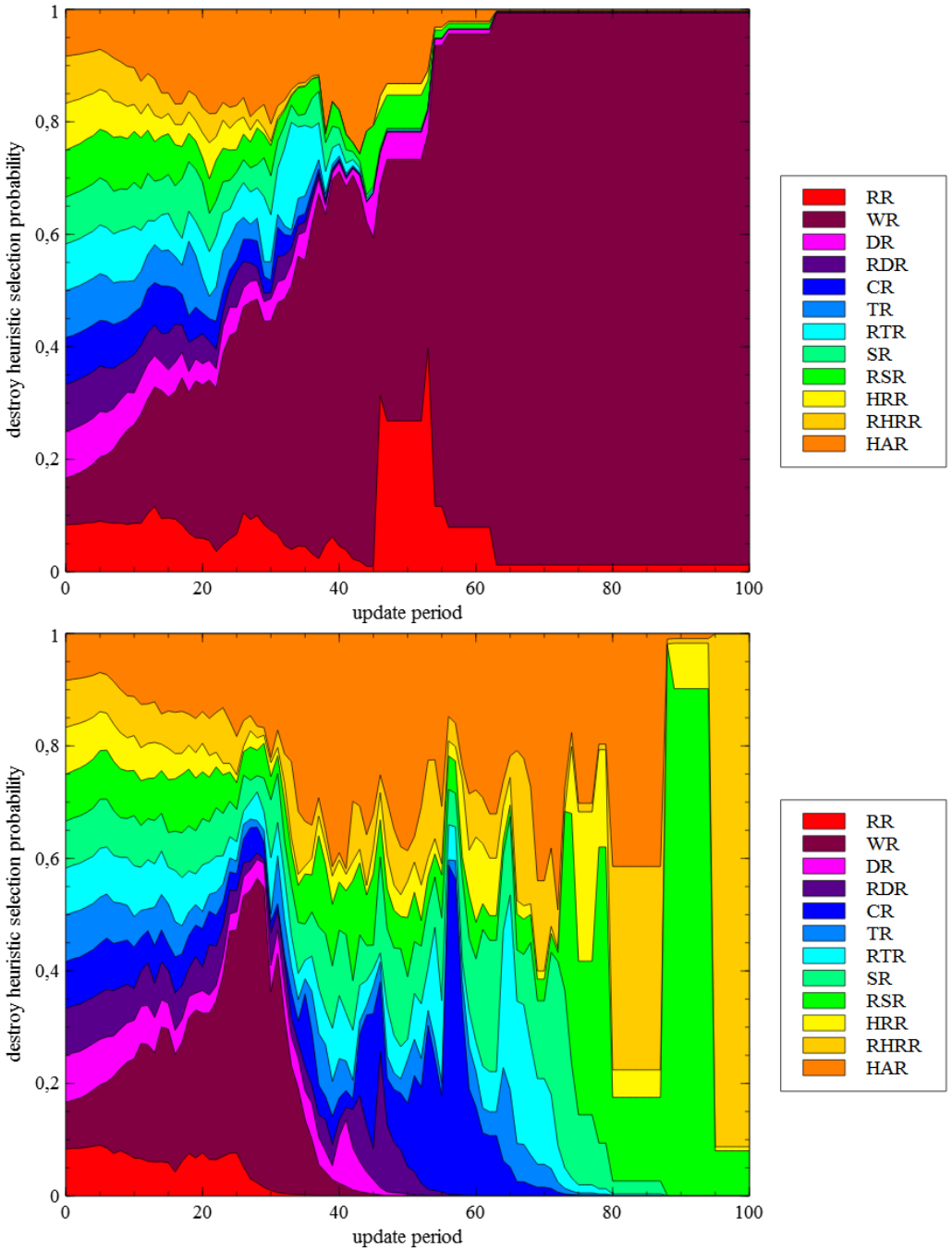


Figure 15: Two examples of the destroy heuristic selection probability development over all update periods in different runs on the instance prob500E. The heuristics are Random Removal (RR), Worst Removal (WR), Distance-oriented Removal (DR), Randomized DR (RDR), Cluster Removal (CR), Service time-oriented Removal (TR), Randomized TR (RTR), Shaw Removal (SR), Randomized SR (RSR), Historical Request Pair Removal (HRR), Randomized HRR (RHRR) and Historical Action Pair Removal (HAR). Initially, the selection probabilities of all heuristics are equal, because their initial weights are equal. With the weight adjustment after each update period these probabilities change. The more success a heuristic has, the higher is the probability to be selected in the ensuing update periods. In the figure on the left, all but two heuristics are practically eliminated from the selection after slightly more than 5000 iterations. On the right, the number of heuristics with success decreases slowly and even at the end there are multiple heuristics involved.

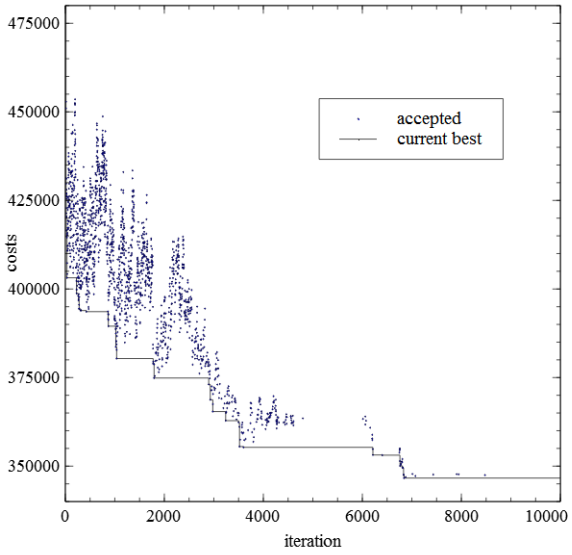


Figure 16: Comparison of the costs of all accepted solutions and the currently best solution. The chart shows the evolution of the solution costs of the run with minimal costs on instance prob500E. The continuous line demonstrates the costs of the currently best known solution. The dots represent the costs of accepted solutions. The Threshold Acceptance rather allows non-improving solutions at the beginning than later, so the average difference between the dots and the line decreases with the number of iterations.

duration was noticeable. It points to underutilization of the vehicles, i.e. the vehicles have to wait idly for the next action. Naturally, such a situation is not favorable in a real world scenario, because the vehicle driver would have to be paid for waiting. Solutions to that problem could be to encourage the algorithm to reduce the total duration or the number of vehicles. Both measures are controlled by input parameters, namely β and δ. In the experiment configuration, both α and β were set to one. This means that distance and duration are equally important to the algorithm. Increasing the β value could solve the problem. On the other hand, setting δ would force the algorithm to take the number of used vehicles into account. By that, maybe less vehicles would be used and the average workload would increase. That problem is certainly interesting for further studies on ALNS.

The cost difference between accepted solutions and the currently best known solutions is shown in Figure 16. The Threshold Acceptance method that was used allows non-improving solutions to be accepted if the cost difference to the current, i.e. the last accepted, solution is below its threshold. Meanwhile, the threshold decreases with an increasing number of iterations. This trend is also visible in Figure 16: At the beginning, the maximum cost difference of the accepted solutions goes up to 15% and steadily decreases from thereon. Furthermore, the number of accepted solutions decreases with the number of iterations. This is the logical consequence of the Threshold Acceptance allowing less non-improving solutions to be accepted and the decreasing solution costs.

In the following, we compare solutions based on their Hamming distance. The Hamming distance of two solutions is calculated by checking all actions of one

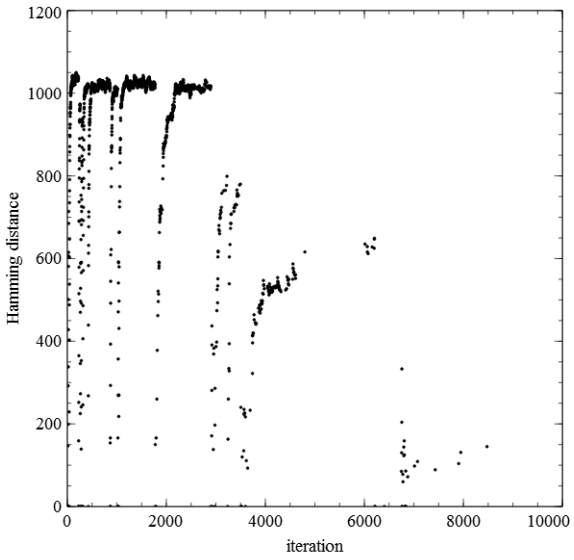


Figure 17: The Hamming distance of all accepted solutions to the currently best solutions. We took the data from the most successful run on the instance prob500E.

solution if the other solution contains the outgoing directed edge that connects it to the next action. For every edge that is not contained, the Hamming distance is incremented. As a consequence, the maximum Hamming distance of two solutions is $2n + 2m$ where n is the number of requests and m is the number of vehicles.

Figure 17 shows the Hamming distance of accepted solutions and the currently best solutions of the best run on instance prob500E. Especially in the early iterations, the distance increases relatively often to around 1000 and then jumps to zero. These jumps correspond to the discovery of a new global best solution. Interestingly, the algorithm does not hesitate to move away from the currently best known solution. Ropke and Pisinger [29] also noticed this effect: "This behavior is contrary to some of the ideas behind the Variable Neighborhood metaheuristics and the Noising Method, where one tries to stick around the currently best known solution or return to it if the current search direction seems fruitless". Figure 18 shows a different picture of the same run. Instead of the distance to the currently best known solution, the distance to the global best solution found during the run is illustrated. Contrary to Figure 17, the distance remains almost continuous and without huge jumps. Despite the multitude of changes to the solutions in the early iterations, the Hamming distance to the global best solution is nearly constant.

Furthermore, we examined the Hamming distance of all accepted solutions and the previously accepted solutions of the same run. The result is shown in Figure 19. At the beginning, different distances from 100 to 400 are included. But with an increasing number of iterations, the Hamming distance decreases almost linearly. Ropke and Pisinger [29] have had similar results. Obviously, the algorithm only accepts solutions with a lower Hamming distance in later iterations. Figure 20 confirms this observation. The Hamming distance of the created solutions to the last accepted solution remains roughly in the same di-

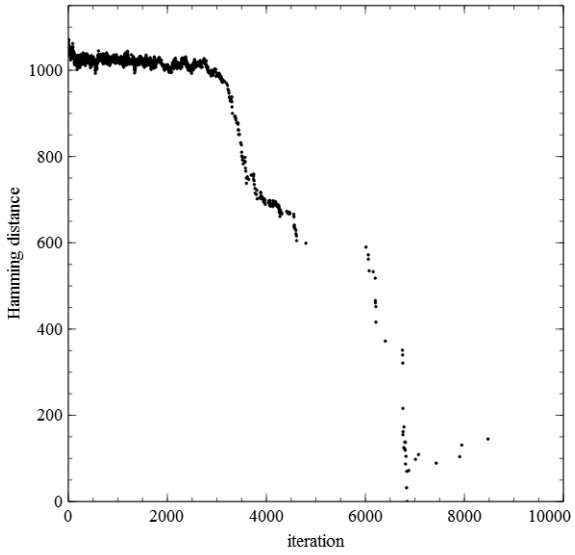


Figure 18: The Hamming distance of all accepted solutions to the overall best solution found. We took the data from the most successful run on the instance prob500E.

mension throughout the run. The only difference is that the created solutions with big changes, i.e. large Hamming distances, are not accepted in later stages. Therefore Ropke and Pisinger recommended using a decreasing degree of destruction. This alternative was rejected by our tuning. Maybe it would work better if not the degree of destruction itself were decreasing but the interval boundaries of the interval from which the degree of destruction is taken. This could be an interesting approach for future work on ALNS.
Apart from that, Ropke and Pisinger [29] remarked that the Hamming distance between the last accepted and newly created solutions was often zero, i.e. the iteration did not change any part of the solution. That problem has only occurred on some of the smaller instances with 50 requests. Apart from them, the Hamming distance was always considerably higher than zero in our experiments. For instance, the Hamming distance in Figure 20 with 500 requests was never below 50. As real life instances normally contain significantly more than 50 requests, the problem does not have to be counteracted in our implementation.

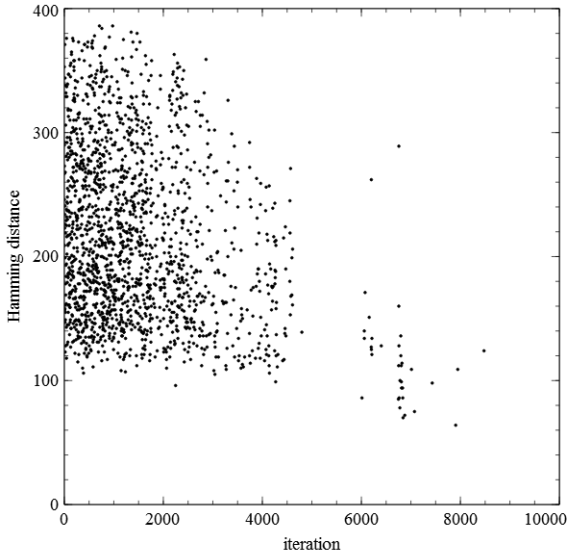


Figure 19: The Hamming distance of all accepted solutions to the previously accepted solution. We took the data from the most successful run on the instance prob500E.

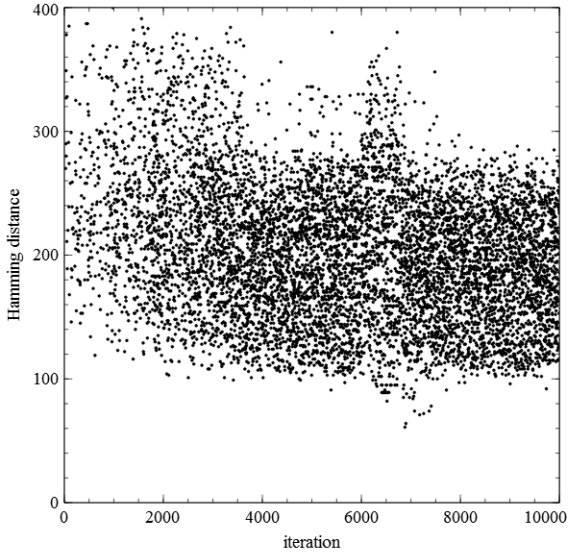


Figure 20: The Hamming distance of all created solutions to the last accepted solutions. We took the data from the most successful run on the instance prob500E.

6.2 THE SUCCESS SO FAR

Even though ALNS is a relatively new heuristic, it enjoys great popularity. Since Ropke and Pisinger [32] proposed it in 2006, several other researchers have adopted, refined and extended ALNS according to the requirements of different problems. Different examples of papers on ALNS include Mattos Ribeiro and Laporte's "An Adaptive Large Neighborhood Search Heuristic for the Cumulative Capacitated Vehicle Routing Problem" [27], Hemmelmayr, Cordeau and Crainic's "An adaptive large neighborhood search heuristic for Two-Echelon Vehicle Routing Problems arising in city logistics" [12], Azi, Gendreau and Potvin's "An Adaptive Large Neighborhood Search for the Vehicle Routing Problem with Multiple Trips" [3], Kovacs, Parragh, Doerner and Hartl's "Adaptive large neighborhood search for service technician routing and scheduling problems" [20] and Masson, Lehuédé and Péton's "An Adaptive Large Neighborhood Search for the Pickup and Delivery Problem with Transfers" [26]. As Ropke and Pisinger pointed out, the ALNS framework can possibly be used for different kinds of optimization problems, not just logistic distribution problems like the RPDPTW. For example, ALNS has been applied to the generalized High School Timetabling Problem by Sørensen, Kristiansen and Stidsen [38] and the Production Routing Problem by Adulyasak, Cordeau and Jans [1].
The reason for the acceptance and further research on the topic is the success that ALNS has had. As a reminder, ALNS was intended to be a general heuristic that can solve different kinds of Vehicle Routing Problems and Pickup and Delivery Problems efficiently without having to adjust parameter configurations to the specific problem. Ropke and Pisinger [29] have tested their ALNS implementation with a general parameter setting on different Vehicle Routing Problems. The results were especially promising because ALNS was not only capable of competing with most of the specialized heuristics, but it even found new best solutions for several problem instances. Additionally, it turned out that ALNS was the best heuristic for the minimization of the number of vehicles on instances of the Vehicle Routing Problem with Time Windows. These encouraging results have definitely contributed to an increased general interest in the ALNS framework.

6.3 CONCLUSION

In this thesis, we have presented the general Adaptive Large Neighbood Search framework with a step by step introduction including the neighborhood concept and the Large Neighborhood Search heuristic. ALNS repeatedly uses different heuristics as operators to destroy and repair components of a solution. An acceptance method then decides if the changes are adopted or not. The application to the Rich Pickup and Delivery Problem with Time Windows was our second goal. Apart from the detailed description, we implemented the ALNS algorithm with a few additions. We used four new destroy heuristics and a new initial solution construction method. Additionally, we have chosen Threshold Acceptance from a pool of acceptance methods including Random Walk, Greedy Acceptance, Simulated Annealing, Old Bachelor Acceptance and the

Great Deluge Algorithm. Finally, we tuned the parameters and tested the program on 48 instances. Even though we used a general parameter configuration, ALNS handled different problem instances well. The results were very positive: On 46 of 48 instances, new best solutions were found and our solutions had on average 14.04% lower costs than Ropke and Pisinger's. Therefore, we can confirm the results of Ropke and Pisinger [29]. Still, there are several parts of the algorithm with room for improvement. First of all, the weight adjustments are based on the assumption that past success indicates future success. In pratice, this approach is suboptimal, because some heuristics are rather suited for the earlier iterations and some perform better during the later iterations. But the algorithm prefers those which have success at the beginning throughout the run. There is definitely potential for improvements, maybe even with one of the ideas proposed above. Secondly, the tuning resulted in a degree of destruction which is selected randomly from an interval in every iteration. But the results have shown that large-scale changes are not accepted in later stages of the algorithm. A steadily decreasing degree of destruction has been rejected by the tuning, so maybe a randomized choice from an interval with steadily decreasing boundaries could improve the algorithm. These were the major problems that stood out during the analysis of the results.
All in all, Adaptive Large Neighborhood Search shows great promise for complex large-scale optimization problems. By using the large neighborhood approach, it avoids getting stuck in local optima within tightly constrained search spaces. Contrary to many heuristics for logistic routing problems, ALNS is not limited to a certain problem class and can be applied to very general problems. With real-world logistic problems containing heterogeneous customers, requests and vehicles that often do not fit into a special Vehicle Routing Problem or Pickup and Delivery Problem category, it is a heuristic with good prospects for the future.

BIBLIOGRAPHY

[1] Yossiri Adulyasak, Jean-François Cordeau, and Raf Jans. Optimization-based adaptive large neighborhood search for the production routing problem. *Transportation Science*, 48(1):20–45, 2012.

[2] Ravindra K. Ahuja, Özlem Ergun, James B. Orlin, and Abraham P. Punnen. A survey of very large-scale neighborhood search techniques. *Discrete Applied Mathematics*, 123(1):75–102, 2002.

[3] Nabila Azi, Michel Gendreau, and Jean-Yves Potvin. *An adaptive large neighborhood search for a vehicle routing problem with multiple trips*. CIRRELT, 2010.

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

[5] Russell Bent and Pascal Van Hentenryck. A two-stage hybrid algorithm for pickup and delivery vehicle routing problems with time windows. *Computers & Operations Research*, 33(4):875–893, 2006.

[6] William A. Dees Jr and Patrick G. Karger. Automated rip-up and reroute techniques. In *Proceedings of the 19th Design Automation Conference*, pages 432–439. IEEE Press, 1982.

[7] Moshe Dror, Gilbert Laporte, and Pierre Trudeau. Vehicle routing with split deliveries. *Discrete Applied Mathematics*, 50(3):239–254, 1994.

[8] Gunter Dueck. New optimization heuristics: the great deluge algorithm and the record-to-record travel. *Journal of Computational physics*, 104(1):86–92, 1993.

[9] Gunter Dueck and Tobias Scheuer. Threshold accepting: a general purpose optimization algorithm appearing superior to simulated annealing. *Journal of computational physics*, 90(1):161–175, 1990.

[10] George F. D. Duff. Differences, derivatives and decreasing rearrangements. *Canad. J. Math*, 19:1153–1178, 1967.

[11] Yvan Dumas, Jacques Desrosiers, and Francois Soumis. The pickup and delivery problem with time windows. *European Journal of Operational Research*, 54(1):7–22, 1991.

[12] Vera C. Hemmelmayr, Jean-François Cordeau, and Teodor Gabriel Crainic. An adaptive large neighborhood search heuristic for two-echelon vehicle routing problems arising in city logistics. *Computers & operations research*, 39(12):3215–3228, 2012.

[13] Manar I. Hosny and Christine L. Mumford. Constructing initial solutions for the multiple vehicle pickup and delivery problem with time windows. *Journal of King Saud University-Computer and Information Sciences*, 24(1):59–69, 2012.

[14] Te C. Hu, Andrew B. Kahng, and Chung-Wen Albert Tsao. Old bachelor acceptance: A new class of non-monotone threshold accepting methods. *ORSA Journal on Computing*, 7(4):417–425, 1995.

[15] Charlotte Diane Jacobs-Blecha and Marc Goetscalckx. *The Vechicle Routing Problem with Backhauls: Properties and Solution Algorithms*. Material Handling Research Center, Georgia Institute of Technology, 1992.

[16] David S. Johnson and Lyle A. McGeoch. The traveling salesman problem: A case study in local optimization. *Local search in combinatorial optimization*, 1:215–310, 1997.

[17] Michael Kämpf. *Probleme der Tourenbildung*. TU Chemnitz, Fak. für Informatik, 2006.

[18] Richard M. Karp. *Reducibility among combinatorial problems*. Springer, 1972.

[19] Philip Kilby, Patrick Prosser, and Paul Shaw. A comparison of traditional and constraint-based heuristic methods on vehicle routing problems with side constraints. *Constraints*, 5(4):389–414, 2000.

[20] Attila A. Kovacs, Sophie N. Parragh, Karl F. Doerner, and Richard F. Hartl. Adaptive large neighborhood search for service technician routing and scheduling problems. *Journal of scheduling*, 15(5):579–600, 2012.

[21] Attila Andras Kovacs. *Heuristics for service technician routing and scheduling Problems*. PhD thesis, Uni Wien, 2009.

[22] Joseph B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50, 1956.

[23] Gilbert Laporte and Ibrahim H. Osman. Routing problems: A bibliography. *Annals of Operations Research*, 61(1):227–262, 1995.

[24] Manuel López-Ibánez and Jérémie Dubois-Lacoste. The irace package download and information page, July 2014. URL `http://iridia.ulb.ac.be/irace/`.

[25] Manuel López-Ibánez, Jérémie Dubois-Lacoste, Thomas Stützle, and Mauro Birattari. The irace package, iterated race for automatic algorithm configuration. *IRIDIA, Université Libre de Bruxelles, Belgium, Tech. Rep. TR/IRIDIA/2011-004*, 2011.

[26] Renaud Masson, Fabien Lehuédé, and Olivier Péton. An adaptive large neighborhood search for the pickup and delivery problem with transfers. *Transportation Science*, 47(3):344–355, 2013.

[27] Glaydston Mattos Ribeiro and Gilbert Laporte. An adaptive large neighborhood search heuristic for the cumulative capacitated vehicle routing problem. *Computers & Operations Research*, 39(3):728–735, 2012.

[28] Sophie N. Parragh, Karl F. Doerner, and Richard F. Hartl. A survey on pickup and delivery problems. *Journal für Betriebswirtschaft*, 58(1):21–51, 2008.

[29] David Pisinger and Stefan Ropke. A general heuristic for vehicle routing problems. *Computers & operations research*, 34(8):2403–2435, 2007.

[30] David Pisinger and Stefan Ropke. Large neighborhood search. In *Handbook of metaheuristics*, pages 399–419. Springer, 2010.

[31] Stefan Ropke. Stefan Ropke's homepage including 48 RPDPTW instances, June 2014. URL http://www.diku.dk/~sropke/.

[32] Stefan Ropke and David Pisinger. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation science*, 40(4):455–472, 2006.

[33] Stefan Ropke and David Pisinger. A unified heuristic for a large class of vehicle routing problems with backhauls. *European Journal of Operational Research*, 171(3):750–775, 2006.

[34] Martin W. P. Savelsbergh and Marc Sol. The general pickup and delivery problem. *Transportation science*, 29(1):17–29, 1995.

[35] Gerhard Schrimpf, Johannes Schneider, Hermann Stamm-Wilbrandt, and Gunter Dueck. Record breaking optimization results using the ruin and recreate principle. *Journal of Computational Physics*, 159(2):139–171, 2000.

[36] Paul Shaw. A new local search algorithm providing high quality solutions to vehicle routing problems, 1997.

[37] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *Principles and Practice of Constraint Programming-CP98*, pages 417–431. Springer, 1998.

[38] Matias Sørensen, Simon Kristiansen, and Thomas Riis Stidsen. International timetabling competition 2011: An adaptive large neighborhood search algorithm. In *9th International Conference on the Practice and Theory of Automated Timetabling (PATAT 2012)*, pages 489–492, 2012.

[39] Paolo Toth and Daniele Vigo. An exact algorithm for the vehicle routing problem with backhauls. *Transportation science*, 31(4):372–385, 1997.

[40] Paolo Toth and Daniele Vigo. An overview of vehicle routing problems. *The vehicle routing problem*, 9:1–26, 2002.

[41] Peter J. M. Van Laarhoven and Emile H. L. Aarts. *Simulated annealing*. Springer, 1987.

[42] Thibaut Vidal, Teodor Gabriel Crainic, Michel Gendreau, and Christian Prins. A unified solution framework for multi-attribute vehicle routing problems. *European Journal of Operational Research*, 2013.

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography *"The Elements of Typographic Style"*. `classicthesis` is available for both LaTeX and LyX:

http://code.google.com/p/classicthesis/

Table 14: The table shows the results of the initial solution construction of both the Greedy and the Related variant, including the execution time and the costs of the created solutions. The last two columns compare the values of the previous columns. The time factor is calculated by dividing the execution time in seconds of the Greedy Construction by the execution time of the Related Construction. The cost factor has been created likewise with the solution costs instead of the execution time.

	Greedy		Related		comparison	
instance	time	costs	time	costs	time factor	cost factor
prob50A	0.029	72207.3	0.010	83329.7	2.9	0.867
prob50B	0.008	373103.9	0.008	184734.9	1.0	2.020
prob50C	0.004	173538.7	0.005	78475.4	0.8	2.211
prob50D	0.005	169890.3	0.006	78518.9	0.8	2.164
prob50E	0.005	61421.1	0.007	279390.0	0.7	0.220
prob50F	0.006	61267.3	0.005	76655.1	1.2	0.799
prob50G	0.004	62183.8	0.002	71631.2	2.0	0.868
prob50H	0.005	64710.8	0.002	78726.5	2.5	0.822
prob50I	0.005	63495.8	0.005	180318.3	1.0	0.352
prob50J	0.005	73742.5	0.009	283811.8	0.6	0.260
prob50K	0.004	67813.2	0.003	72085.2	1.3	0.941
prob50L	0.004	68071.6	0.013	79466.3	0.3	0.857
prob100A	0.031	248944.7	0.041	573706.3	0.8	0.434
prob100B	0.033	129287.7	0.008	159391.5	4.1	0.811
prob100C	0.031	127256.1	0.006	153443.1	5.2	0.829
prob100D	0.032	234516.1	0.012	153037.8	2.7	1.532
prob100E	0.032	111218.9	0.007	168533.5	4.6	0.660
prob100F	0.035	132080.4	0.006	163219.2	5.8	0.809
prob100G	0.030	112668.2	0.008	148995.3	3.8	0.756
prob100H	0.035	107046.6	0.005	143844.7	7.0	0.744
prob100I	0.030	142428.8	0.018	165762.5	1.7	0.859
prob100J	0.033	144390.6	0.010	167782.9	3.3	0.861
prob100K	0.029	123660.0	0.008	140779.1	3.6	0.878
prob100L	0.032	119364.2	0.009	147998.3	3.6	0.807
prob250A	0.548	525263.7	0.110	729449.1	5.0	0.720
prob250B	0.555	306253.1	0.062	394709.0	9.0	0.776

	Greedy		Related		comparison	
instance	time	costs	time	costs	time factor	cost factor
prob250C	0.552	271764.2	0.026	328344.3	21.2	0.828
prob250D	0.570	279402.8	0.0240	350892.5	23.8	0.796
prob250E	0.543	232138.3	0.024	330269.2	22.6	0.703
prob250F	0.566	334714.0	0.081	399960.8	7.0	0.837
prob250G	0.589	230113.1	0.045	346255.3	13.1	0.665
prob250H	0.591	227880.0	0.033	353651.1	17.9	0.644
prob250I	0.574	379665.0	0.203	496140.1	2.8	0.7650
prob250J	0.555	297749.0	0.039	412392.5	14.2	0.722
prob250K	0.561	261510.8	0.024	344294.4	23.4	0.760
prob250L	0.581	260054.2	0.027	345923.2	21.5	0.752
prob500A	5.536	564094.6	0.141	776971.1	39.3	0.726
prob500B	5.451	566217.4	0.184	796746.8	29.6	0.711
prob500C	5.745	517183.3	0.089	654426.1	64.6	0.790
prob500D	5.821	532681.8	0.126	686968.5	46.2	0.775
prob500E	5.495	440833.0	0.102	754116.4	53.9	0.585
prob500F	5.580	411618.0	0.131	716074.8	42.6	0.575
prob500G	5.728	441825.2	0.102	693562.0	56.2	0.637
prob500H	5.851	410782.2	0.088	646168.1	66.5	0.636
prob500I	5.367	553929.1	0.115	779303.9	46.7	0.711
prob500J	5.426	564592.9	0.121	786505.9	44.8	0.718
prob500K	5.663	497616.2	0.116	660999.6	48.8	0.753
prob500L	5.801	493362.2	0.097	672870.9	59.8	0.733

Table 15: The table shows the resulting solution costs of the initial solution construction of both the Greedy and the different Regret-n Construction heuristics. The construction methods are deterministic and therefore only had to be executed once.

	Greedy	Regret-2	Regret-3	Regret-4	Regret-5
prob50A	72207.3	86610.5	84866.6	79442.3	77655.4
prob50B	373103.9	84067.7	94768.8	89376.7	84258.0
prob50C	173538.7	193046.8	82183.2	89986.0	79074.3
prob50D	169890.3	82809.2	81451.6	189081.5	82073.6
prob50E	61421.1	75240.8	69940.1	74374.5	73535.1
prob50F	61267.3	68814.3	69449.6	68131.1	64857.3
prob50G	62183.8	69530.9	71327.3	72673.5	72138.4
prob50H	64710.8	74364.7	68845.0	73662.7	79251.7
prob50I	63495.8	77260.4	71993.1	77517.8	73125.2
prob50J	73742.5	85263.9	285113.3	85407.2	87032.1
prob50K	67813.2	80949.5	80882.1	78244.8	77803.0
prob50L	68071.6	81940.9	78617.6	77819.3	86868.3
prob100A	248944.7	281823.9	185321.0	178208.3	190731.1
prob100B	129287.7	157581.1	170986.7	165442.9	164306.0
prob100C	127256.1	156461.0	163631.1	164618.1	151377.7
prob100D	234516.1	160320.3	162872.8	160639.8	162021.3
prob100E	111218.9	135382.0	151265.4	157800.4	157286.4
prob100F	132080.4	143166.6	140543.4	143057.2	141267.7
prob100G	112668.2	141311.2	150933.0	146283.1	145939.7
prob100H	107046.6	137486.2	140289.0	147795.6	137526.7
prob100I	142428.8	154732.2	158785.4	156905.7	165629.0
prob100J	144390.6	163130.5	175806.7	163782.6	171209.5
prob100K	123660.0	132789.5	151350.5	150815.7	151951.0
prob100L	119364.2	145018.5	155787.4	141627.0	152974.7
prob250A	525263.7	407837.3	441822.4	438825.9	430645.8
prob250B	306253.1	360054.3	392386.8	383471.5	393170.0
prob250C	271764.2	337340.8	356722.6	344371.8	357336.6
prob250D	279402.8	343202.3	351598.4	359496.8	351505.0
prob250E	232138.3	290494.1	281659.7	289883.6	290510.2
prob250F	334714.0	343669.6	335127.6	351412.1	340589.2
prob250G	230113.1	309595.1	319688.3	304385.7	311531.4
prob250H	227880.0	284932.0	310152.8	283948.7	301486.8
prob250I	379665.0	361841.7	367737.3	374757.5	371293.8
prob250J	297749.0	353045.5	388195.5	364584.6	377721.6

	Greedy	Regret-2	Regret-3	Regret-4	Regret-5
prob250K	261510.8	352024.2	361615.6	363918.3	344471.7
prob250L	260054.2	340852.5	335097.6	339147.3	345469.6
prob500A	564094.6	675744.1	710243.0	730167.1	729506.3
prob500B	566217.4	699444.1	707830.1	726268.3	747439.4
prob500C	517183.3	653811.0	644819.3	665781.1	638722.0
prob500D	532681.8	667307.8	672065.1	646233.3	649236.0
prob500E	440833.0	550175.5	585291.8	598186.4	604201.3
prob500F	411618.0	505025.8	561893.0	558240.5	535928.2
prob500G	441825.2	608541.3	605719.2	589113.8	596094.7
prob500H	410782.2	540096.5	574476.0	559457.6	563546.6
prob500I	553929.1	681062.1	700947.7	731912.0	729570.4
prob500J	564592.9	717360.0	701420.9	744094.5	727537.8
prob500K	497616.2	599019.4	634776.5	634940.7	632743.4
prob500L	493362.2	623111.0	612476.1	644198.1	641792.8

Table 16: The results of the experiments of our ALNS implementation on the test instances by Ropke [31]. The number in the instance name corresponds to the number of requests in the instance. Furthermore, the average execution time is given in seconds. In the columns *cost*, *distance* and *duration* both the average and the best value are shown. There were 100 executions per instance. The row *best* contains the values of the solution that had the minimum costs among the 100 solutions. Therefore, the distance and duration values are not necessarily the minimum values of all solutions.

instance	avg exec. time (s)		costs	distance	duration
prob50A	2.3	best	52186.5	17258.3	34928.2
		avg	52985.4	17665.3	35320.0
prob50B	2.9	best	56427.1	20288.6	36138.4
		avg	57234.2	20571.5	36662.7
prob50C	2.5	best	52435.9	17736.3	34699.5
		avg	53831.4	18232.5	35598.9
prob50D	3.0	best	54338.8	18558.9	35779.8
		avg	55393.9	18881.1	36512.8
prob50E	2.8	best	43112.7	13305.2	29807.6
		avg	43802.6	13567.4	30235.2
prob50F	3.5	best	38217.1	10029.4	28187.7
		avg	39514.7	10766.2	28748.5
prob50G	2.9	best	41179.7	11636.4	29543.2
		avg	42335.3	12233.3	30102.1
prob50H	3.3	best	38991.8	10339.9	28651.9
		avg	40159.8	10968.8	29191.0
prob50I	3.1	best	46992.3	15788.1	31204.3
		avg	47877.3	16001.1	31876.3
prob50J	2.8	best	49872.2	16355.4	33516.9
		avg	50413.9	16695.5	33718.4
prob50K	2.7	best	48118.5	15687.7	32430.8
		avg	49301.2	16219.7	33081.5
prob50L	2.7	best	49416.2	16365.3	33050.9
		avg	50776.3	17253.5	33522.8
prob100A	11.2	best	106407.6	36695.0	69712.6
		avg	108817.9	37702.5	71115.4
prob100B	14.0	best	94150.6	30994.2	63156.4
		avg	97051.4	31696.8	65354.6
prob100C	11.8	best	92497.3	29222.0	63275.3
		avg	96152.1	30862.7	65289.4

instance	avg exec. time (s)		costs	distance	duration
prob100D	13.7	best	96480.3	30874.4	65605.9
		avg	99153.8	31945.1	67208.7
prob100E	14.2	best	70365.2	18244.2	52121.0
		avg	72900.1	18679.0	54221.1
prob100F	15.9	best	75347.3	20188.5	55158.8
		avg	77353.0	20928.2	56424.8
prob100G	14.1	best	70651.3	17239.2	53412.2
		avg	73792.5	19286.4	54506.1
prob100H	16.1	best	67091.0	16527.8	50563.3
		avg	70637.3	17911.3	52726.0
prob100I	11.7	best	91313.1	28825.2	62487.9
		avg	93871.0	29728.5	64142.5
prob100J	14.1	best	95318.9	29823.6	65495.3
		avg	97391.4	31152.4	66239.0
prob100K	13.4	best	84474.9	25536.2	58938.7
		avg	87252.8	26968.2	60284.6
prob100L	14.1	best	84778.5	24327.2	60451.3
		avg	87570.0	25835.3	61734.7
prob250A	178.6	best	245033.0	79862.6	165170.4
		avg	253482.8	83606.5	169876.3
prob250B	119.1	best	222486.1	70998.6	151487.5
		avg	229404.3	73775.7	155628.6
prob250C	107.4	best	214712.7	65203.8	149508.9
		avg	223917.0	69196.4	154720.6
prob250D	178.3	best	219725.3	66371.5	153353.8
		avg	228722.4	70074.2	158648.2
prob250E	156.8	best	167748.7	39508.3	128240.5
		avg	174867.1	43022.3	131844.8
prob250F	144.7	best	180032.9	44185.9	135847.0
		avg	185495.6	47889.2	137606.4
prob250G	179.3	best	168574.0	40051.9	128522.1
		avg	175502.4	43307.7	132194.8
prob250H	190.1	best	167771.8	38063.9	129707.9
		avg	175094.9	41478.4	133616.5
prob250I	174.9	best	211451.5	63381.0	148070.5
		avg	220678.6	67055.6	153623.1
prob250J	176.9	best	224181.3	69487.5	154693.8
		avg	232035.1	71877.9	160157.3

instance	avg exec. time (s)		costs	distance	duration
prob250K	103.3	best	196287.9	55442.5	140845.5
		avg	205183.9	59247.9	145936.0
prob250L	162.9	best	207442.5	61616.7	145825.7
		avg	219471.9	65628.7	153843.1
prob500A	623.5	best	443110.8	134584.4	308526.4
		avg	453549.3	139218.1	314331.2
prob500B	699.0	best	448197.6	135673.6	312524.0
		avg	463010.2	143419.1	319591.2
prob500C	669.7	best	428429.4	123434.2	304995.2
		avg	448542.8	133801.2	314741.6
prob500D	715.7	best	425033.9	123909.6	301124.3
		avg	451152.9	135896.4	315256.5
prob500E	917.2	best	346631.7	86406.1	260225.6
		avg	357540.7	89920.3	267620.4
prob500F	930.3	best	340523.2	79573.3	260949.8
		avg	351071.6	85094.7	265976.9
prob500G	725.8	best	356460.8	86863.3	269597.5
		avg	372291.8	95710.1	276581.7
prob500H	929.6	best	340971.2	81554.2	259416.9
		avg	360883.3	89676.9	271206.4
prob500I	663.6	best	431345.9	124932.3	306413.5
		avg	444731.0	131947.6	312783.4
prob500J	887.5	best	437456.8	129946.5	307510.3
		avg	455701.3	137176.5	318524.9
prob500K	677.5	best	409477.7	114120.5	295357.2
		avg	431177.6	124581.5	306596.1
prob500L	729.2	best	411741.2	115899.8	295841.3
		avg	432517.3	125462.2	307055.1

Table 17: The results of the experiments of our implementation are compared to Ropke's [31]. For all instances, the average and best solution costs of both implementations are shown.

instance	implementation	best costs	avg costs
prob50A	Lutz	52186.5	52985.4
	Ropke	63414.8	63551.3
prob50B	Lutz	56427.1	57234.2
	Ropke	71294.2	72568.8
prob50C	Lutz	52435.9	53831.4
	Ropke	63584.6	64324.0
prob50D	Lutz	54338.8	55393.9
	Ropke	74484.3	74936.8
prob50E	Lutz	43112.7	43802.6
	Ropke	50266.0	51332.5
prob50F	Lutz	38217.1	39514.7
	Ropke	51870.8	52414.2
prob50G	Lutz	41179.7	42335.3
	Ropke	51827.6	52243.2
prob50H	Lutz	38991.8	40159.8
	Ropke	56865.1	57776.5
prob50I	Lutz	46992.3	47877.3
	Ropke	54532.0	54673.7
prob50J	Lutz	49872.2	50413.9
	Ropke	62990.5	63801.5
prob50K	Lutz	48118.5	49301.2
	Ropke	57847.3	58559.8
prob50L	Lutz	49416.2	50776.3
	Ropke	64952.8	65478.2
prob100A	Lutz	106407.6	108817.9
	Ropke	119608.1	122114.3
prob100B	Lutz	94150.6	97051.4
	Ropke	107525.6	109104.8
prob100C	Lutz	92497.3	96152.1
	Ropke	114019.5	115210.0
prob100D	Lutz	96480.3	99153.8
	Ropke	120750.6	122372.0
prob100E	Lutz	70365.2	72900.1
	Ropke	79085.2	80907.3

instance	implementation	best costs	avg costs
prob100F	Lutz	75347.3	77353.0
	Ropke	87054.9	88039.0
prob100G	Lutz	70651.3	73792.5
	Ropke	88317.2	89176.6
prob100H	Lutz	67091.0	70637.3
	Ropke	91806.1	93042.2
prob100I	Lutz	91313.1	93871.0
	Ropke	105939.1	106789.4
prob100J	Lutz	95318.9	97391.4
	Ropke	107510.5	108251.1
prob100K	Lutz	84474.9	87252.8
	Ropke	101073.7	102186.4
prob100L	Lutz	84778.5	87570.0
	Ropke	115627.0	116419.2
prob250A	Lutz	245033.0	253482.8
	Ropke	258627.9	260643.9
prob250B	Lutz	222486.1	229404.3
	Ropke	245064.6	247857.9
prob250C	Lutz	214712.7	223917.0
	Ropke	248793.1	250624.6
prob250D	Lutz	219725.3	228722.4
	Ropke	263146.2	265083.6
prob250E	Lutz	167748.7	174867.1
	Ropke	172979.7	174147.5
prob250F	Lutz	180032.9	185495.6
	Ropke	190662.0	192151.1
prob250G	Lutz	168574.0	175502.4
	Ropke	190370.5	191844.5
prob250H	Lutz	167771.8	175094.9
	Ropke	199596.2	201505.9
prob250I	Lutz	211451.5	220678.6
	Ropke	226048.9	228099.5
prob250J	Lutz	224181.3	232035.1
	Ropke	240868.6	245589.6
prob250K	Lutz	196287.9	205183.9
	Ropke	231432.5	234942.4
prob250L	Lutz	207442.5	219471.9
	Ropke	248030.0	251344.5

instance	implementation	best costs	avg costs
prob500A	Lutz	443110.8	453549.3
	Ropke	466780.9	469805.7
prob500B	Lutz	448197.6	463010.2
	Ropke	471652.8	476944.1
prob500C	Lutz	428429.4	448542.8
	Ropke	471017.4	474111.3
prob500D	Lutz	425033.9	451152.9
	Ropke	487647.3	490795.3
prob500E	Lutz	346631.7	357540.7
	Ropke	343950.6	345685.8
prob500F	Lutz	340523.2	351071.6
	Ropke	337737.0	342741.8
prob500G	Lutz	356460.8	372291.8
	Ropke	378502.2	382154.1
prob500H	Lutz	340971.2	360883.3
	Ropke	389949.6	393676.2
prob500I	Lutz	431345.9	444731.0
	Ropke	434295.3	438489.4
prob500J	Lutz	437456.8	455701.3
	Ropke	455011.9	458567.8
prob500K	Lutz	409477.7	431177.6
	Ropke	452319.9	458774.4
prob500L	Lutz	411741.2	432517.3
	Ropke	470132.0	473542.0

Table 18: The success of the destroy heuristics with different problem instance characteristics. The characteristics are the ones explained in Section 5.3. The last column contains the average number of times the heuristic was called over all experiments in which the characteristic occurred. The other three columns show how many percent of these calls resulted in a new global best solution (r_1), an improving (not global best) solution (r_2) and an accepted non-improving solution (r_3). The rows with *all* in the characteristics column show the results over all instances.

destroy heuristic	characteristic	r_1	r_2	r_3	total
Random Removal	GDT1	0.5	4.4	11.6	2009.9
	GDT2	0.4	4.3	11.2	1881.0
	GDT3	0.6	5.6	11.1	1500.0
	TT1	0.5	4.9	11.6	1783.5
	TT2	0.5	4.5	11.1	1810.4
	RT1	0.5	4.6	11.2	1809.8
	RT2	0.5	4.7	11.4	1784.1
	all	0.5	4.7	11.3	1797.0
Worst Removal	GDT1	1.8	11.5	16.8	1229.9
	GDT2	1.9	12.3	17.6	1119.4
	GDT3	1.9	13.9	16.1	991.5
	TT1	1.9	12.2	18.2	1117.2
	TT2	1.9	12.8	15.6	1110.1
	RT1	1.9	12.5	17.4	1129.3
	RT2	1.9	12.5	16.4	1098.0
	all	1.9	12.5	16.9	1113.6
Distance-oriented Removal	GDT1	0.1	1.1	4.4	2613.6
	GDT2	0.1	1.5	5.2	2363.3
	GDT3	0.2	3.4	9.0	1737.8
	TT1	0.1	1.9	5.8	2229.4
	TT2	0.1	1.8	5.9	2247.0
	RT1	0.1	1.9	5.8	2316.8
	RT2	0.1	1.8	6.0	2159.6
	all	0.1	1.9	5.9	2238.2

destroy heuristic	characteristic	r_1	r_2	r_3	total
Randomized Distance-oriented Removal	GDT1	0.4	2.6	7.9	486.7
	GDT2	0.5	2.9	8.1	508.3
	GDT3	0.6	5.0	10.7	573.8
	TT1	0.5	3.5	8.8	517.8
	TT2	0.5	3.6	9.1	528.0
	RT1	0.5	3.4	8.6	537.0
	RT2	0.5	3.7	9.4	508.8
	all	0.5	3.5	9.0	522.9
Cluster Removal	GDT1	0.8	7.8	20.9	315.7
	GDT2	0.8	8.0	18.4	345.8
	GDT3	0.6	6.7	13.8	541.3
	TT1	0.7	7.0	16.1	413.5
	TT2	0.7	7.8	17.9	388.3
	RT1	0.7	7.4	16.5	391.7
	RT2	0.7	7.4	17.5	410.1
	all	0.7	7.4	17.0	400.9
Service time-oriented Removal	GDT1	0.4	3.8	12.4	503.1
	GDT2	0.4	3.9	11.5	529.4
	GDT3	0.6	5.4	12.9	570.3
	TT1	0.5	4.2	12.0	532.0
	TT2	0.5	4.5	12.5	536.5
	RT1	0.5	4.4	12.0	516.7
	RT2	0.5	4.3	12.5	551.9
	all	0.5	4.4	12.3	534.3
Randomized Service time-oriented Removal	GDT1	0.8	6.8	18.1	338.4
	GDT2	0.8	7.2	17.9	352.9
	GDT3	0.7	7.5	15.3	489.4
	TT1	0.8	7.4	16.9	382.7
	TT2	0.8	7.0	16.9	404.4
	RT1	0.8	7.2	16.5	393.4
	RT2	0.8	7.2	17.2	393.7
	all	0.8	7.2	16.9	393.6

destroy heuristic	characteristic	r_1	r_2	r_3	total
Shaw Removal	GDT1	0.8	8.7	21.8	457.8
	GDT2	0.8	8.5	19.5	499.7
	GDT3	0.7	8.8	17.1	605.0
	TT1	0.7	8.6	18.9	524.0
	TT2	0.7	8.7	19.5	517.7
	RT1	0.8	8.7	18.6	510.3
	RT2	0.7	8.7	19.8	531.4
	all	0.7	8.7	19.2	520.8
Randomized Shaw Removal	GDT1	0.5	5.5	14.9	600.0
	GDT2	0.5	5.5	13.7	642.8
	GDT3	0.6	6.9	14.1	673.7
	TT1	0.6	6.0	14.1	640.3
	TT2	0.6	6.0	14.3	637.5
	RT1	0.6	6.1	14.3	602.3
	RT2	0.6	5.9	14.1	675.5
	all	0.6	6.0	14.2	638.9
Historical Request Pair Removal	GDT1	0.1	1.8	5.9	416.0
	GDT2	0.1	1.8	5.4	485.0
	GDT3	0.1	3.0	7.4	563.0
	TT1	0.1	2.2	6.3	482.7
	TT2	0.1	2.3	6.4	493.3
	RT1	0.1	2.3	6.2	476.1
	RT2	0.1	2.2	6.4	499.9
	all	0.1	2.2	6.3	488.0
Randomized Historical Request Pair Removal	GDT1	0.1	3.7	10.2	211.3
	GDT2	0.1	3.4	8.9	257.9
	GDT3	0.2	3.6	7.8	434.8
	TT1	0.1	3.4	8.5	300.1
	TT2	0.1	3.7	8.9	302.6
	RT1	0.2	3.5	8.5	295.7
	RT2	0.1	3.6	8.8	307.0
	all	0.1	3.6	8.7	301.3

destroy heuristic	characteristic	r_1	r_2	r_3	total
Historical Action Pair Removal	GDT1	0.2	5.6	6.1	817.6
	GDT2	0.2	5.9	6.2	1014.6
	GDT3	0.2	6.4	6.0	1319.3
	TT1	0.2	5.8	5.9	1076.9
	TT2	0.2	6.3	6.2	1024.1
	RT1	0.2	6.1	6.1	1020.8
	RT2	0.2	6.0	6.0	1080.1
	all	0.2	6.0	6.1	1050.5

Table 19: The success of the repair heuristics with different problem instance characteristics. The characteristics are the ones explained in Section 5.3. The last column contains the average number of times the heuristic was called over all experiments in which the characteristic occurred. The other three columns show how many percent of these calls resulted in a new global best solution (r_1), an improving (not global best) solution (r_2) and an accepted non-improving solution (r_3). The rows with *all* in the characteristics column show the results over all instances.

repair heuristic	characteristic	r_1	r_2	r_3	total
Basic Greedy Repair	GDT1	0.5	4.5	9.1	6591.2
	GDT2	0.5	4.6	8.9	6263.0
	GDT3	0.5	5.7	9.5	5648.1
	TT1	0.5	4.9	9.3	6208.9
	TT2	0.5	4.9	9.0	6126.0
	RT1	0.5	4.9	9.2	6180.6
	RT2	0.5	4.9	9.1	6154.3
	all	0.5	4.9	9.1	6167.4
Regret-2 Repair	GDT1	0.5	4.8	12.3	2432.8
	GDT2	0.5	5.3	12.4	2537.2
	GDT3	0.6	6.8	12.8	2802.2
	TT1	0.5	5.6	12.6	2591.8
	TT2	0.5	5.7	12.4	2589.6
	RT1	0.5	5.6	12.3	2589.6
	RT2	0.5	5.7	12.7	2591.9
	all	0.5	5.7	12.5	2590.7
Regret-3 Repair	GDT1	0.9	6.6	17.2	976.0
	GDT2	0.8	6.2	15.4	1199.8
	GDT3	0.8	7.5	14.4	1549.7
	TT1	0.9	6.8	15.4	1199.3
	TT2	0.8	6.9	15.5	1284.4
	RT1	0.9	6.6	14.7	1229.8
	RT2	0.9	7.0	16.2	1253.9
	all	0.9	6.8	15.5	1241.8

ERKLÄRUNG

Ich erkläre, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, 15.08.2014

Roman Lutz