ERASMUS UNIVERSITY ROTTERDAM

Erasmus School of Economics

Master Thesis - Econometrics & Management Science

Solving the Technician Routing and Scheduling Problem using Adaptive Large Neighborhood Search

Rick Kaptein - 474650

Supervisor: dr. R. Spliet

Second assessor: dr. S. Sharif Azadeh

October 19, 2020

Abstract

This thesis concerns the solving of Technician Routing and Scheduling Problems for multiple days. Taken into account are multiple aspects deemed relevant in practical applications, such as the rescheduling of tasks that are already communicated to clients, adding overtime, scheduling tasks with (high) priority preferably as soon as possible, and ensuring some sort of work balance between technicians. Therefore, a weighted objective function is used. This objective function minimizes the amount of rescheduled tasks and tours with overtime before minimizing the amount of work time, weighted delay of priority tasks, and the value for work balance. The experiments conducted in this research concern the determination of good parameters for the proposed model, testing the scalability of the model, and simulating the scheduling process during a full working week. The latter includes adding new tasks to an existing schedule and using interventions such as adding overtime or scheduling some fixed tasks. Results show that the model as proposed in this research, is capable of solving problem instances of around 100 tasks. Furthermore, it is shown that the model performs well in a setting of creating a schedule for a week, where tasks are added on a daily basis.

Contents

Chapter 1

Introduction

This research is conducted in cooperation with Finext, with the aim of efficiently solving real life routing and scheduling problems. The current research concerns such problems at a maintenance company, which provides maintenance and repairs, and builds new installations. The current research will be relevant to the maintenance company as they want to improve their scheduling process in multiple ways. However, this research can also be applied to maintenance companies in general. The company used as an example in this study wants to reduce the amount of travel time between appointments, while meeting all customers requirements and preferences. For example, tasks with priority should be scheduled within a certain time window, but preferably as soon as possible. Reducing the travel time between appointments allows the company to fulfill more tasks with the same workforce. Furthermore, they want to be able to accommodate changes to their initial schedule at a later moment in time. Besides making better schedules and better rescheduling, they want to automate the scheduling and rescheduling processes, as these processes currently require many manual steps, which take a lot of time and ultimately results in inefficiencies.

The problem we will optimize in this research is twofold. Currently, the company in question first creates a schedule with all known appointments for one week ahead. In this schedule, all known tasks will be assigned to technicians at a certain day. We will refer to this problem as the initial scheduling problem. Then, during the week, changes occur on a daily basis. These changes consist of new repair requests, cancellations and tasks that need to be rescheduled at request of clients. This means that the initial schedule needs to be rescheduled to incorporate all changes and new requests. A difficulty in this problem is that the scheduled tasks in the initial schedule are already communicated to clients, which means that this part of the schedule already has been fixed.

A fixed task is only fixed to a certain day, as the exact time is not communicated to clients. If it is not possible to add all changes and new tasks to the initial schedule, it is allowed to make minimal changes to the fixed part of the initial schedule, or to add overtime to some tours of technicians. It should be noted that changing already communicated appointments can lead to customer dissatisfaction.

There are different types of tasks that need to be scheduled. One part of these tasks are maintenance tasks and the tasks related to the construction of new installations. These tasks are usually known at

least a month in advance. Other tasks, like repairs or malfunctions of installations, are usually known not long in advance. Repairs and malfunctions are classified using two different priority levels: high and medium. High priority tasks need to be scheduled within two working days, while medium priority tasks need to be scheduled within five working days. Besides different types of tasks, there are different characteristics for technicians as well. Each technician has a unique home address, which will be used as start and end of a tour. Besides that, each technician has a certain skill set, which will be used to determine which tasks can be fulfilled by that technician. Lastly, technicians have availability time windows. Availability time windows encompass the ability to restrict tours to a maximum shift length. Using availability time windows allows for example a shift with a specific time window of four hours on Monday from 9:00-13:00h. However, time windows can also be more general.

Single problem The problem of creating an initial schedule for a week and the problem of rescheduling on a daily basis can be regarded as the same type of problem. In case of the rescheduling problem, all fixed tasks can be regarded as tasks with a very specific time window. On the other hand, the week schedule can be regarded as a special case of the rescheduling problem, where no tasks are fixed yet. In this research, we will focus on creating a model that is able to solve both problems. We will solve the problems using the Adaptive Large Neighborhood Search (ALNS) framework, as it has been shown that using ALNS for these kind of problems generates high quality solutions and has very good performance (Ropke and Pisinger (2006); Pillac, Gueret, and Medaglia (2013)).

Chapter 2

Problem description

The problem addressed in this research can be described using several elements. In this chapter we provide a detailed explanation of all problem elements, the corresponding restrictions and the objective function. The following paragraphs describe the different characteristics of the problem. Table 2.1 and Table 2.2 give a brief overview of the notation used to describe the TRSP, which will be explained in detail in the next paragraphs.

Table 2.1: Constants and sets of the Technician Routing and Scheduling Problem

Notation	Description
$r \in R$	Set of all tasks
$R^+ \subseteq R, R^- \subseteq R$	Set of all currently scheduled and unscheduled tasks, respectively
$k \in K$	Set of technicians
$d \in D$	Set of days
$S_k \subseteq S$	Set of skills of technician k with S as set of all possible skills
$S_r \subseteq S$	Set of required skills for fulfilling task r
$w_{k,d}^{open}$	Opening of availability time window of technician k on day d
$w_{k,d}^{close}$	End of availability time window of technician k on day d
$w_{r,d}^{open}$	Opening of time window of task k on day d
$w_{r,d}^{close}$	End of time window of task k on day d
$d_{u,v}$	Travel duration between location u and v
μ_r	Service duration of task r
s_r	Request datetime of task r
p_r	Priority of task r
Δ	Maximum added duration of overtime

Table 2.2: Variables of the Technician Routing and Scheduling Problem

Notation	Description
$t_{k,d} \in T$	Tour of technician k on day d. Each tour consists of a sequence of actions a.
a_i^t	The i^{th} action of tour t
$w_{a_i^t}^{open}, w_{a_i^t}^{close}$	Earliest start time and latest ending time of action i in tour t
$\tau_{a_i^t}$	The scheduled start time of the i^{th} action in tour t
o_t	Binary variable with value 1 if tour t is a tour with overtime and 0 otherwise

Technicians A technician is an employee that can fulfil tasks. There is a heterogeneous group K of technicians available. Each technician $k \in K$ has a home location, a discrete set of skills and a series of availability time windows. The skill set of technician k is denoted by $S_k \subseteq S$, where S denotes the set of all possible skills. An availability time window describes the start and end of a working day at a specified date. This time window excludes travel time to their first task and from their last task, e.g. a technician may start a task at the beginning of the working day, even though the technician needs to travel to the location of that task first. Consequently, a series of time windows describes the availability of a technician during the complete planning horizon. The availability time window of technician k on a given day $d \in D$ is specified by the interval $[w_{k,d}^{open}, w_{k,d}^{close}]$.

Clients A client is a business-to-business customer. Clients have relevant properties such as a location and a series of time-windows during which the client allows technicians to fulfil tasks at the client's location. As we can assign these properties to tasks, it is not necessary to include clients as variables in this problem.

Tasks A task is a request from a client that needs to be fulfilled. There is a set of R tasks. Each task $r \in R$ has the following properties:

- A discrete priority level. A task with a priority higher than the lowest priority will be referred to as a priority task.
- The service time μ_r. This time is deterministic and independent of the technician fulfilling this task.
- Required skills $S_r \subseteq S$. The required skills determine which technicians are allowed to fulfil this task.
- Specified time windows per day $d \in D$, which are denoted by the interval $[w_{r,d}^{open}, w_{r,d}^{close}]$. These time window series describe the moments during which task r can be fulfilled. This set of time windows is a subset of the time windows from the corresponding company.
- Request date and time s_r. The date and time of the moment a task is requested by a client. This is relevant for priority tasks, because scheduling priority tasks closer to the request date and time results, all other things being equal, in a better objective value.

A fixed task is a task of which the date is already communicated to the client. This means that the task should be fulfilled on that day. Therefore, the time windows of fixed tasks are set to a single day. We formulate this as set $R_f \subset R$ where R_f is allowed to be an empty set $R_f = \emptyset$. The goal is to keep fixed tasks scheduled on their communicated day, but in case the model cannot find a feasible solution, it is possible to reschedule one or more fixed tasks.

Tour A tour is described as the day schedule of a technician. A tour starts at the home location of a technician, then follows along the assigned tasks for that day and finally ends at the home location again. This implies that all tours start and end at technician's home location. The tour of technician k on day d is denoted by $t_{k,d}$. A tour consists of an ordered sequence of actions a^t. Each tour has a length $|t| \geq 2$. The first action a^t_1 and last action $a^t_{|t|}$ of a tour mark the start and end of a tour. All actions in between are assigned tasks. Each action in a tour has a time window on the interval $[w^{open}_{a^t_i}, w^{close}_{a^t_i}]$, which denotes the earliest start time of that action and the latest ending time of that action.

Overtime In case no feasible solution can be found, it is possible to add a fixed amount of overtime to a tour. In case tour t has overtime, the variable o_t is set to $o_t = 1$. All tours $t \in T$ are initialized without overtime, therefore $o_t = 0 \; \forall \; t \in T$.

Work Time Work Time is defined as the time measured since the start of a tour until the end of a tour, which means it consists of travel time, service time, and possibly waiting time if appointments have restrictive time windows and a technician needs to wait until it can start an appointment. Total Work Time is defined as the scheduled start time of the last action, minus the start time of the first action of a trip, so $\tau_{a^t_{|t|}} - \tau_{a^t_1}$.

Weighted Delay Weighted Delay is defined as the elapsed time since the request date and time of a task until the start moment of a scheduled task, multiplied by the priority level. Delay is only taken into account for priority tasks. For tasks with higher priority it is provided that a larger delay results in worse customer satisfaction, e.g. when a client has a broken machine, the client is more satisfied when the machine is repaired as soon as possible. The total Weighted Delay of all priority tasks can be calculated as $\sum_{r \in R} p_r(\tau_r - s_r)$, where p_r equals 0 for non-priority tasks and takes positive discrete values for priority tasks.

Work Balance Work Balance, also referred to as load balancing, is defined as the act of balancing the amount of work each technician has to do on each day. If Work Balance is not taken into account, it is possible that schedules are created with large differences in workload per technician. As large differences in workload between technicians are not preferred, load balancing is taken into account. In this problem case, we will try to minimize the difference between the technician that has the least Work Time in total, compared to the technician that has most Work Time in total. This difference is calculated by using Equation 2.1, where T_k denotes the set of tours belonging to technician k.

$$\max_{k \in K} \Big(\sum_{t \in T_k} \tau_{a^t_{|t|}} - \tau_{a^t_1} \Big) - \min_{k \in K} \Big(\sum_{t \in T_k} \tau_{a^t_{|t|}} - \tau_{a^t_1} \Big) \tag{2.1}$$

2.1 Objective of schedule

The goal of this research is to improve the scheduling process by creating better schedules with and without fixed tasks. A better schedule is characterized by a better weighted objective value. The main objective is to create a feasible schedule with a minimal amount of overtime and rescheduled fixed tasks. Then, the objective is to minimize the total Work Time. The next objective is to minimize the delay of priority tasks. Minimizing this delay is necessary, as it improves customer satisfaction. Lastly, Work Balance should be incorporated in the objective function. Consequently, the objective can be described as:

Minimize rescheduling of fixed tasks and adding overtime, total Work Time, Weighted Delay of priority tasks, and minimizing the difference in total work load between technicians.

We describe this objective using 2.2 as a weighted objective function with weights π_i. For ease of readability, we make use of $f_1(\Pi)$ to $f_2(\Pi)$.

$$z(\Pi) = \min \sum_{i=1}^{5} \pi_i f_i(\Pi) \tag{2.2}$$

$$f_1(\Pi) = \sum_{t \in T} o_t + \sum_{r \in R_f} 1_{\{r \in R_f\}} \tag{2.3}$$

$$f_2(\Pi) = \sum_{t \in T} \tau_{a^t_{|t|}} - \tau_{a^t_0} \tag{2.4}$$

$$f_3(\Pi) = \sum_{r \in R} p_r(\tau_r - s_r) \tag{2.5}$$

$$f_4(\Pi) = \max_{k \in K} \Big(\sum_{t \in T_k} \tau_{a^t_{|t|}} - \tau_{a^t_1} \Big) - \min_{k \in K} \Big(\sum_{t \in T_k} \tau_{a^t_{|t|}} - \tau_{a^t_1} \Big) \tag{2.6}$$

The function $f_1(\Pi)$ simply counts the number of tours that have overtime and the number of rescheduled tasks. It is given that the number of shifts with overtime should be minimal and fixed tasks should remain fixed, if possible. As the function value of f_1 cannot be a large number, it makes sense to set weight π_1 to a very large number. For implementation, it is also possible to regard this part, Equation 2.3, using a lexicographic ordering. In that case, every solution satisfies $z(\Pi) < z(\Pi')$, if and only if $f_1(\Pi) \leq f_1(\Pi')$. The function $f_2(\Pi)$ calculates the total Work Time of solution Π. Then, the function $f_3(\Pi)$ calculates the time between the request date and time of a task and the scheduled moment, multiplied by the priority value of that task. As the priority value of non priority tasks is zero, this function only calculates Weighted Delay for priority tasks. Finally, the function $f_4(\Pi)$ yields the difference between the technician with most total working and the technician with least total Work Time.

Chapter 3

Literature

3.1 Problem Class

The scheduling problem and all corresponding constraints, as described in Chapter 2, are well described in literature. The described problem has a lot of similarities to the Vehicle Routing Problem (VRP), as there is a set of locations belonging to tasks, that must be visited, and there is a set of vehicles, or technicians in our case, that can make tours to visit those locations. However, the described problem has some more requirements, which make the problem different to the VRP. First of all, each technician has a maximum shift time for each day, which can be regarded as a Resource Constraint. Secondly, each task requires a certain skill, which means that only technicians with that type of skill can fulfill that task. Furthermore, tasks can be restricted to certain days or time windows, and employees can have varying time windows per day as well, in which they are available for fulfilling tasks.

Paraskevopoulos, Laporte, Repoussis, and Tarantilis (2017) provide a recent survey about various types Resource Constrained Vehicle Routing Problems and an overview of different approaches to solve these problems. In their paper, they explain how technician scheduling problems can be described as the Technician Routing and Scheduling Problem (TRSP). TRSP can be seen as a generalization of the VRP with Time Windows (VRPTW), where all technicians have a certain skillset and each task requires a certain skill. Compared to the VRP, technicians can be regarded as vehicles and tasks as clients (Pillac et al., 2013).

In our problem case, the TRSP deviates in one regard, namely rescheduling of fixed tasks. The TRSP allows tasks to be fixed, as this can be accomplished by setting a very restrictive time window, but rescheduling a fixed task means that the time window of tasks needs to be relaxed or extended. Adding the possibility of extending time windows of tasks is a generalization of the TRSP. Solving the TRSP is $\mathcal{NP}$-hard, as it is a generalization of the VRP. Therefore, the described problem with fixing and relaxing time windows of tasks is also $\mathcal{NP}$-hard. Pillac et al. (2013) propose using metaheuristic Adaptive Large Neighborhood Search (ALNS) in order to solve this problem. This method shows great flexibility, fast solving times and small optimality gaps, when tested using the Solomon VRPTW benchmark instances.

The model from Pillac et al. (2013) is directed at a single period instance of the TRSP. The TRSP

can be extended to a multi-period setting, where each resource can be regarded as a technician-day pair. Tricoire, Bostel, Dejax, and Guez (2013) provide a clear description for this multi-period problem and propose a column generation approach to solve this. Their approach allows for both time windows for tasks and day dependent time windows per technician.

3.2 Adaptive Large Neighborhood Search

In our problem case, we need to be able to construct efficient schedules, while part of the solution can be fixed. Furthermore, the objective is non linear due to the minimum and maximum values used for calculating work balance. As the ALNS framework is very flexible in constructing solutions and is proven to be fast (Pillac et al., 2013), we will use this framework for our problem case. The ALNS framework is also able to handle non-linear objective functions. ALNS is first introduced by Ropke and Pisinger (2006) as an extenstion to Large Neighborhood Search (LNS). ALNS encompasses LNS, but makes use of multiple large neighborhoods. The ALNS framework makes use of several repair and destroy methods and adaptively switches between these methods using weights, based on their previous succes. Destroy methods remove a certain amount of tasks from solutions, while repair methods reinsert these tasks again in order to create new solutions. The following destroy and repair methods will be based on Ropke and Pisinger (2006). ALNS is also applied to solve problems similar to the TRSP. Majidi, Hosseini-Motlagh, and Ignatius (2018) and Li, Chen, and Prins (2016) showed good performance and results on two different Pickup and Delivery problems.

Destroy operators Both Ropke and Pisinger (2006) and Pillac et al. (2013) make use of three destroy operators, namely random destroy, critical or worst removal, and related or Shaw removal. In addition to these destroy methods, Majidi et al. (2018) make use of several special cases of the Shaw removal operator, e.g. distance based removal and time (window) based removal. Besides that, they use worst time removal, which calculates the waiting time of a technician before fulfilling a task, e.g. when the time window of that task is not open yet. According to their performance analysis on these destroy operators, most operators add at least some value to the quality of the solution. The best performing operators for their problem are Shaw removal and worst distance removal. In their analysis, it appears that the quality of solutions does not deteriorate when time-based removal and worst time removal are not included in the ALNS framework. It is also stated that a combination of multiple operators ultimately works best, because it allows for diversification of the solution space. Li et al. (2016) also use random removal, worst removal and Shaw removal, together with two price related operators, which are not applicable to our problem case.

Repair operators Common repair operators are greedy insertion and regret-k insertion (Ropke and Pisinger (2006); Pillac et al. (2013)). In addition, Majidi et al. (2018) proposes random insertion and best time insertion, where best-time insertion is based on the absolute deviation from the start of the time window of a task and the start time of a task. According to their analysis, random insertion does

not add much value to the solution quality. When best time insertion is left out as an operator, the objective values of solutions worsen by around 10% for their problem instances.

Diversification In order to allow more diversification of the solution space, Simulated Annealing can be used to allow worse solutions to be accepted (Pillac et al. (2013); Li et al. (2016)).

Further improvements Sometimes solutions from the ALNS can be further improved using local search heuristics (Li et al. (2016)), e.g. by swapping one or two tasks within one tour or swapping tasks between tours. Pillac et al. (2013) take a different approach in order to improve solutions from the ALNS. As a post optimization step, they store all promising tours as columns and calculate the optimal solution based on this subset of possible tours using a Set Covering formulation.

3.3 Load balancing

As described in Chapter 2, load balancing needs to be incorporated in the objective as well. Schwarze and Voß (2013) provide several approaches in order to improve load balance. Two relatively simple approaches are either minimizing the maximum indicator value, or minimizing the difference between the maximum and minimum indicator value. In our problem, the indicator value can be regarded as shift length or total shift length per technician. Both approaches can be added to the objective function and additional costs can be associated to this value, depending on how important work balancing is. They describe additional approaches as well, like minimizing the variance of the indicator value or relative indicator value differences. Schwarze and Voß (2013) conclude that minimizing the maximum indicator value yields the best load balancing, but results in higher routing costs.

Chapter 4

Methodology

In this chapter, we will describe the methodology for solving the earlier described problem in detail. Section 4.1 describes the general structure of the ALNS method. In Section 4.2, various destroy and repair operators are explained in more detail. Then, Section 4.3 explains how schedules are created and how variables are updated. Thereafter, Section 4.4 explains how the objective value of a tour can be calculated and how the optimal start times of tasks within a tour can be determined. Next, Section 4.5 explains how insertion costs and removal costs are calculated. Section 4.6 describes an approach on rescheduling fixed tasks, in case no feasible solution can be found without rescheduling. Section 4.7 describes how initial solutions can be created. Finally, Section 4.8 describes several approaches to improve performance.

4.1 Adaptive Large Neighborhood Search

In this section we will explain the general idea of (Large) Neighborhood Search and the parallel implementation of the ALNS algorithm. Let us define Π as a feasible solution to the TRSP. Then, $\mathcal{N}(\Pi)$ can be described as a neighborhood of solutions similar to solution Π, e.g. 2 tasks are removed from Π and reinserted again to obtain solution Π'. The idea of the ALNS method is to use multiple different neighborhoods and explore them in different ways. In ALNS, we create different neighborhoods by making use of multiple destroy and repair operators. A destroy operator is a method of removing tasks from a current solution, while a repair operator is a method of reinserting these tasks to create a new solution. In the current study, we will apply the general structure of the (parallel) ALNS algorithm from Pillac et al. (2013), resulting in Algorithm 1. Algorithm 1 will be explained in detail in the following subsections.

Algorithm 1: Parallel ALNS algorithm

input : Ω, initial solutions; Θ^-/Θ^+, set of destroy/repair operators

output: Π^*

$\Pi^* = \text{argmin}_{\Pi \in \Omega} \{z(\Pi)\}$

while stopping criterion is not met **do**

  $\Omega' \leftarrow \text{selectSubset}(\Omega, K)$

  parallel forall $\Pi \in \Omega'$ **do**

    for I^p iterations **do**

      $destroy \leftarrow \text{select}(\Theta^-)$

      $repair \leftarrow \text{select}(\Theta^+)$

      $\Pi' \leftarrow repair\big(destroy(\Pi^p)\big)$

      if $\text{accept}(\Pi', \Pi^p)$ **then**

        $\Pi^p \leftarrow \Pi'$

      end

      if $z(\Pi') < z(\Pi^*)$ **then**

        $\Pi^* \leftarrow \Pi'$

      end

      $\text{updateWeights}(destroy, repair, \Pi')$

    end

    $\Omega \leftarrow \Omega \cup \{\Pi^p\}$

  end

end

The algorithm starts by using a set of initial solutions. Later on, Section 4.7 will explain how we can build initial solutions. First, the algorithm sets Π^* as the global best solution, so far. Then, it does I^m master iterations, where K solutions are selected based on their objective value. Within a master operation, K processes are started in parallel. Note that it is not a requirement to do these K processes in parallel, but as they can search the different neighborhoods for different solutions independently of each other, a significant performance improvement can be achieved by doing this in parallel. Each parallel process executes I^p local iterations in which it consecutively destroys and repairs solutions.

In each local iteration, one destroy operator is selected and removes q tasks from the current solution Π, where $1 \leq q \leq |T|$. The removed tasks are added to set R^-, which can be seen as the set with the remaining tasks to schedule. Then, one repair operator is selected and inserts tasks from the remaining set R^- iteratively until set R^- is empty, or until it is not able to insert a task. The different destroy and repair operators are explained in detail in Section 4.2. If not all tasks can be reinserted, the solution is infeasible. In case of a feasible solution, we will accept the solution based on the objective value. In case of infeasible solutions, a value of M multiplied by the amount of unscheduled tasks will be added to the objective value, where M is a sufficiently large number to make sure that all objective values of solutions with less unscheduled tasks, are better than solutions with more unscheduled tasks. This way, infeasible solutions can be accepted if and only if the previous solution has the same amount of unscheduled tasks, or more unscheduled tasks. This allows the algorithm to improve infeasible solutions

in multiple iterations, in order to find feasible solutions.

Selecting destroy and repair operators The selection of destroy and repair operators is based on a Roulette Wheel with weights. Let ρ_j^- and ρ_j^+ represent the weights of the j^{th} destroy and repair operators, respectively. We define Θ^- and Θ^+ as the sets of destroy and repair operators we will use in our implementation of the ALNS algorithm. In each local iteration of the ALNS algorithm, one destroy operator and one repair operator are selected. This is based on the roulette wheel principle, where we define p_j^- and p_j^+ as the probabilities of selecting the j^{th} destroy operator and repair operator, respectively. Equation 4.1 defines the probability of selecting repair operator j:

$$p_j^- = \frac{\rho_j^-}{\sum_{\rho_k \in \Theta^-} \rho_k} \tag{4.1}$$

The calculation of probabilities for selecting destroy operators follows in a similar way, which is given in Equation 4.2:

$$p_j^- = \frac{\rho_j^-}{\sum_{\rho_k \in \Theta^-} \rho_k} \tag{4.2}$$

Acceptance method A new solution Π' is always accepted when $z(\Pi') < z(\Pi)$. Besides that, we will use Simulated Annealing to allow worse solutions to be accepted as well. Simulated Annealing allows worse solutions based on how close they are to the current best solution, and on a temperature parameter, which decreases after every iteration. A worse solution is accepted with probability $e^{\frac{z(\Pi)-z(\Pi')}{C}}$, where C represents the current temperature. C is initialized by value C_0 and decreases by a factor on the interval $(0, 1)$ in every iteration. It makes sense to choose a factor close to 1, as this results in a slowly decreasing probability of accepting worse solutions.

Updating weights ALNS has an adaptive layer, which means that it iteratively adjusts some parameters based on the performance of the algorithm at that moment. This means that the probabilities of selecting different destroy and repair operators are adjusted in each local iteration, in order to reward good performing operators and penalize bad performing operators. Let Ψ be a score function which outputs values based on the success of the new solution. The score function is given in 4.3, where the values are chosen such that $\omega_1 \geq \omega_2 \geq \omega_3 \geq \omega_4$ in order to reward operators that lead to better solutions.

$$\Psi(s') = \begin{cases} \omega_1, \text{ if } s' \text{ is new global best solution} \\ \omega_2, \text{ if } s' \text{ is an improved solution} \\ \omega_3, \text{ if } s' \text{ is an accepted solution} \\ \omega_4, \text{ if } s' \text{ is not accepted as new solution} \end{cases} \tag{4.3}$$

Then, after each local iteration, we update the weights of the used destroy and repair operators using the score function Ψ and a parameter λ which determines by how much the weights can change within one iteration. The functions we use to update the weights for the destroy and repair operators are as follows:

$$\rho_j^- = \lambda \rho_j^- + (1 - \lambda)\Psi$$
$$\rho_j^+ = \lambda \rho_j^+ + (1 - \lambda)\Psi$$

We choose $\lambda \in [0, 1]$, where a value of $\lambda = 1$ ignores the performance of an operator and keeps the weights of all operators the same throughout the execution of the ALNS algorithm. On the other hand, a value of $\lambda = 0$ completely ignores the initial weights and sets weights only based on their last performance.

4.2 Destroy and repair operators

In each iteration of the ALNS algorithm, q tasks are removed from the current solution using a destroy operator and then reinserted using a repair operator. The following paragraphs describe various destroy and repair operators that are being used by the ALNS algorithm. In this section, we refer to $c^+(r, t, i)$ and $c^-(a_i^t)$, which respectively describe the cost functions of inserting and removing a task. These functions will be explained in detail in Section 4.5.

4.2.1 Destroy operators

Random destroy In order to diversify the search space, a Random Removal of tasks can be helpful. This destroy operator draws uniformly at random q tasks from the current solution and adds them to the remaining set R^-. This destroy operator is less useful when very good solutions are already found, but it can help to escape local optima.

Worst Removal Another destroy method is Worst Removal, where q most costly tasks are removed from the solution. These costs are based on the part of the objective value that can be assigned to a single task. The worst removal is given by Equation 4.4:

$$\underset{t \in T,\ 2 \leq i < |t|}{\operatorname{argmax}} \quad c^-(a_i^t) \tag{4.4}$$

After each removal, the costs of each task in that tour need to be recalculated, before the costs of the next removal can be calculated. Besides using the total costs of a removal $c^-(a_i^t)$, it is also possible to make this destroy operator specific for the different partial objectives. The partial objective $f_1(\Pi)$, as given in Equation 2.3, is only used for overtime or rescheduling, which will be discussed in Section 4.6. The other partial objectives, $f_2(\Pi)$, $f_3(\Pi)$ and $f_4(\Pi)$ can be used to create partial objective specific Worst Removal operators. Let us define $c_2^-(a_i^t)$, $c_3^-(a_i^t)$ and $c_4^-(a_i^t)$ as the cost functions for the operators Worst Work Time Removal, Worst Weighted Delay Removal and Worst Work Balance Removal, respectively.

Quick Worst Removal A computationally faster method of calculating costs is based on calculating reduced travel time only. Quick Worst Removal is a destroy operator similar to Worst Removal, but costs in this operator are solely based on the reduced travel time of removing a task.

4.2.2 Repair operators

Greedy Insertion A relatively straightforward repair method is Greedy Insertion. This heuristic calculates the minimum cost feasible insertions for all tasks that need to be added and then inserts the

task with the minimum of this to the partial solution at the best position. The best task and best position are yielded using Equation 4.5. This is repeated until all tasks are added to the solution.

$$\underset{r \in R^-,\ t \in T,\ 2 \leq i < |t|}{\operatorname{argmin}} \quad c^+(r, t, i) \tag{4.5}$$

Similar to the partial objective specific Worst Removal operators, we can make the Greedy Insertion repair operator specific for the partial objectives f_2, f_3 and f_4 as well.

Regret-k Insertion Another repair method is called Regret-k Insertion. This heuristic is based on the idea that not choosing an insertion now can result in much higher costs later. Therefore, this heuristic looks at the k cheapest insertions for a task and calculates the difference between the cheapest insertion and the sum of the $(k-1)$ other insertions, e.g. a Regret-2 heuristic looks at the two cheapest insertion positions and calculates the difference between them. It then inserts the task where this regret value is maximized at the location with lowest insertion costs. Note that k in this heuristic is different from technician k, which is used elsewhere in this thesis.

Quick Greedy Insertion and Quick Regret-k Similar to costs in Quick Worst Removal, insertion costs can be based on added travel time. This can be used for both Greedy Insertion and Regret-k Insertion. Let us define Quick Greedy Insertion and Quick Regret-k as repair operators where costs are only based on added travel time.

4.3 Creating feasible schedules

As described in chapter 2, a tour t consists of a sequence of actions a_i^t. In order to determine the feasibility of inserting task r in tour $t_{k,d}$ after the i^{th} action in that tour, two constraints need to be checked. First, technician k must have the relevant skills to fulfill task r, which is only when $S_r \subseteq S_k$. Next, there should be enough time between action a_i and action a_{i+1}. For ease of understanding, let us define $\delta_r^0(t,i)$ (resp. $\delta_r^1(t,i)$) as the earliest (latest) start moment of task r in tour t when inserting after the i^{th} action. Then, Equation 4.6 yields the earliest start moment and Equation 4.7 yields the latest start moment of task r.

$$\delta_r^0(t,i) = \max \begin{cases} w_{a_i}^{open} + \mu_{a_i} + d_{a_i,r} \\ w_r^{open} \end{cases} \tag{4.6}$$

$$\delta_r^1(t,i) = \min \begin{cases} w_{a_{i+1}}^{close} - \mu_{a_{i+1}} - d_{r,a_{i+1}} - \mu_r \\ w_r^{close} - \mu_r \end{cases} \tag{4.7}$$

Using Equation 4.6, $\delta_r^0(t,i)$ equals either the sum of the earliest start moment of the previous action, the service duration of the previous action and the travel time from the previous action to action r, or the start of the time window of task r. Equation 4.7 is similar to Equation 4.6, but calculated backwards relative to the latest start time of the next action. The service duration μ_r is subtracted as well, as it calculates the latest start moment of task r. Task r can only be inserted when $\delta_r^0 \leq \delta_r^1$, as this means there is enough time between the other two actions.

4.3.1 Updating time windows

Similar to determining the earliest and latest start moments of inserting a task, we can update the time windows of actions. These time windows describe the intervals in which a task can be executed. These time windows will be used to determine start times τ, as $w^{open}_{a^t_i} \leq \tau_i \leq w^{close}_{a^t_i} - \mu_{a^t_i} \quad \forall \quad 1 \leq i \leq |t|, t \in T$. Equation 4.8 describes how $w^{open}_{a_i}$ can be updated recursively for all tasks in tour t, so for $1 < i < |t|$, by starting at $i = 2$, because the first and last action only describe the start and end of a tour instead of tasks. We initialize $w^{open}_{a^t_1} = w^{open}_t$, which is the start of tour t. Then, after updating the opening time windows of all tasks in tour t, the opening time window of the last action $i = |t|$ is updated. As mentioned, the last action is not a task, so the opening time window is set equal to the opening time window of the previous action, plus the duration of the previous action, plus the necessary travel time. Similarly, we can recursively update $w^{close}_{a_i}$, by starting at $w^{open}_{a^t_{|t|}}$ which is the closing time window of tour t, and iterate backwards from $i = |t| - 1$ to $i = 2$. We initialize $w^{open}_{a^t_{|t|}} = w^{close}_t$, which is the end of tour t. This approach is formalized in Algorithm 2. Note that both the opening time window $w^{open}_{a^t_1}$ and the closing time window $w^{close}_{a^t_{|t|}}$ are never updated, as these mark the earliest and latest start moments of tour t.

$$w^{open}_{a_i} = \max \begin{cases} w^{open}_{a_{i-1}} + \mu_{a_{i-1}} + d_{a_{i-1},a_i} \\ w^{open}_{r_{a_i}} \end{cases} \tag{4.8}$$

$$w^{close}_{a_i} = \min \begin{cases} w^{close}_{a_{i+1}} - \mu_{a_{i+1}} - d_{a_i,a_{i+1}} \\ w^{close}_{r_{a_i}} \end{cases} \tag{4.9}$$

Algorithm 2: Update time windows of actions

for $i = 2$ **to** $|t| - 1$ **do**
  $w^{open}_{a_i} \leftarrow \max\left\{w^{open}_{a_{i-1}} + \mu_{a_{i-1}} + d_{a_{i-1},a_i}, \quad w^{open}_{r_{a_i}}\right\}$
end
$w^{open}_{a_|t|} \leftarrow w^{open}_{a_{|t|-1}} + \mu_{a_{|t|-1}} + d_{a_{|t|-1},a_{|t|}}$
for $i = |t| - 1$ **to** 2 **do**
  $w^{close}_{a_i} \leftarrow \min\left\{w^{close}_{a_{i+1}} - \mu_{a_{i+1}} - d_{a_i,a_{i+1}}, \quad w^{close}_{r_{a_i}}\right\}$
end
$w^{close}_{a_|t|} \leftarrow w^{close}_{a_1} - \mu_{a_1} - d_{a_0,a_1}$

4.4 Objective value of a tour

In order to determine the total costs of a tour, it is necessary to provide an approach to determine the optimal start times within a tour, given a certain order of actions. As mentioned in Section 2.1, the objective consists of minimizing the number of rescheduled tasks, the total Work Time, the Weighted Delay of priority tasks and minimizing the difference between the longest total Work Time of technicians

and the shortest total Work Time of technicians. When determining the optimal start times of actions within a tour, we only need to minimize total Work Time and Weighted Delay corresponding to that tour. This is because the number of rescheduled tasks does not change within a tour and the objective should favor tours with less Work Time. Note that Equation 2.6, which describes the Work Balance factor in the objective function, may prefer a tour with longer Work Time in case the corresponding technician has the shortest total Work Time. This can only be done by introducing more waiting time by postponing some start times in a tour, as the order of actions is fixed. However, according to practice, this behavior is not preferred. Therefore, we exclude Work Balance when calculating the optimal start times. On the other hand, due to the fact that we choose a relatively small weight π_4 for Equation 2.6, the model would usually not prefer a solution where unnecessary waiting time is added in order to achieve a higher value for Work Balance. Note that this decision may result in bad performance for problems where Work Balance has a higher importance. In the next paragraph, we will first determine an algorithm that finds optimal start times for a tour, when only considering total Work Time. Then, in the second paragraph, we will adjust this tour to minimize delay as well.

4.4.1 Minimizing total Work Time

Total Work Time depends on the start time of the earliest task and the end time of the last task in a tour. Alternatively, total Work Time is defined as the sum of travel time, service time and waiting time. In a given tour t with a sequence of actions, it should be noted that travel times between actions and service times of actions are independent of the start times. Therefore, we only want to minimize waiting time between actions. Let us define γ_i as the minimal waiting time of the i^{th} action, given the start time τ_{i-1} of the previous action. Then, the waiting time of the i^{th} action is equal to the opening of the time window of the i^{th} action, minus the ending time of the previous action and travel time from the previous action to action i, in case this value is positive. In case the opening of the time window is earlier, there is no need for waiting time and therefore, $\gamma_i = 0$. Equation 4.10 describes this. We initialize $\gamma_1 = 0$ for all tours.

$$\gamma_i = \max \begin{cases} w_i^{open} - (\tau_{i-1} + \mu_{i-1} + d_{i-1,i}) \\ 0 \end{cases} \tag{4.10}$$

In order to determine the optimal start times when minimizing total Work Time, we will use Figure 4.1 as an example tour. In this figure, the time windows of actions $w_{a_i^t}^{open}$ and $w_{a_i^t}^{close}$ are shown. To increase readability, we will simply use i instead of a_i^t as the index of the i^{th} action. The service times and travel durations are shown on the right side of Figure 4.1. Furthermore, the earliest and latest possible tours are visualized. Note that the actions start at $i = 2$, because the first action, which describes the moment a technician starts the day, can take place before time zero and has no service duration.

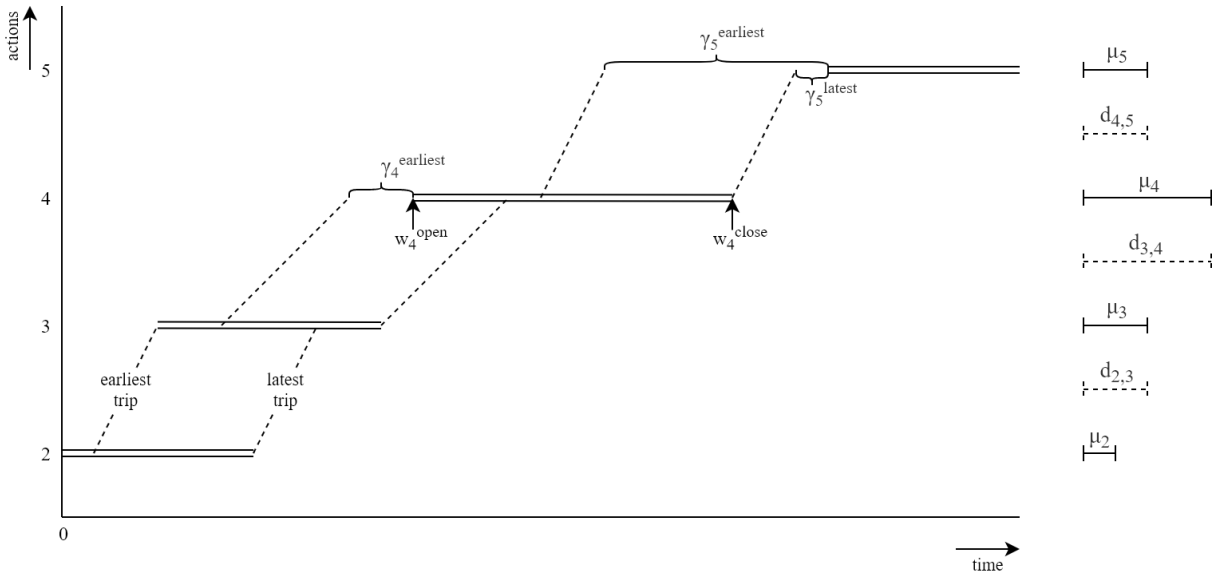


Figure 4.1: Example tour with visual explanation of variables

Let us first explore the earliest possible tour in more detail. In Figure 4.2, it can be seen that there waiting time occurs before the 4^{th} and 5^{th} action. This waiting time occurs before the opening of the time windows w_4^{open} and w_5^{open}. The waiting time γ_4 can be reduced by moving start times τ_2 and τ_3 to a later moment. Therefore, the earliest possible tour is not the optimal tour in this example.

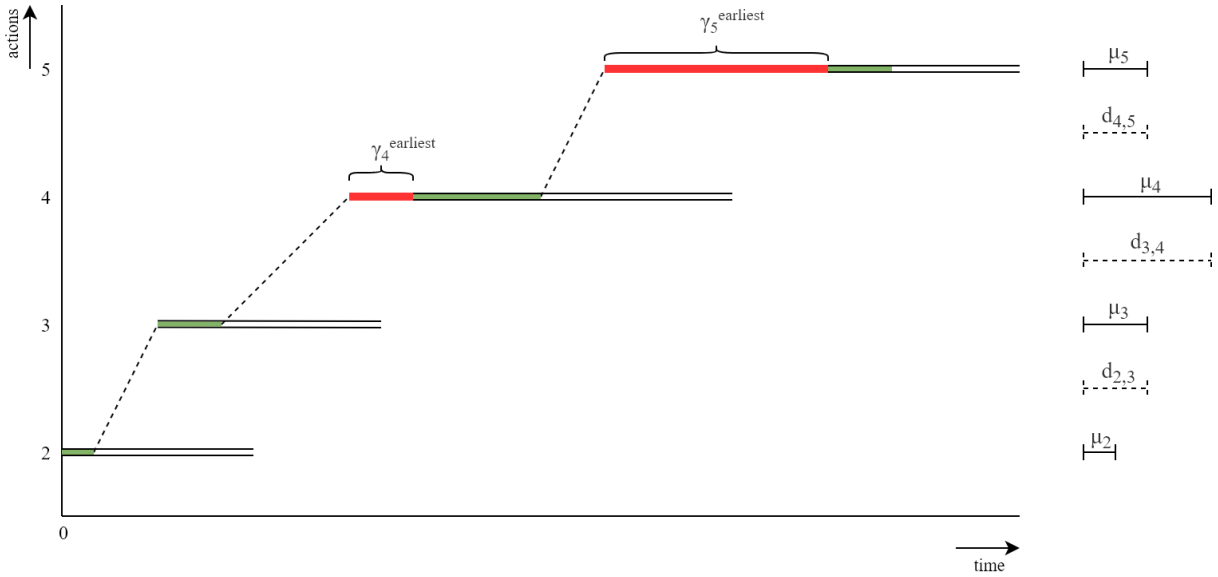


Figure 4.2: Earliest tour

On the other hand, a tour can start at the latest possible moment. Figure 4.3 shows the latest possible tour, where each action starts at $\tau_i = w_i^{close} - \mu_i$. In this tour, some waiting time occurs within the interval of time windows. This is not in line with Equation 4.10, where it is stated that waiting time is equal to zero in case the opening of the time window takes place earlier than the ending time of the last action plus travel time from the last action to this action. Therefore, we can make an improvement when

we remove waiting time within the interval of time windows.

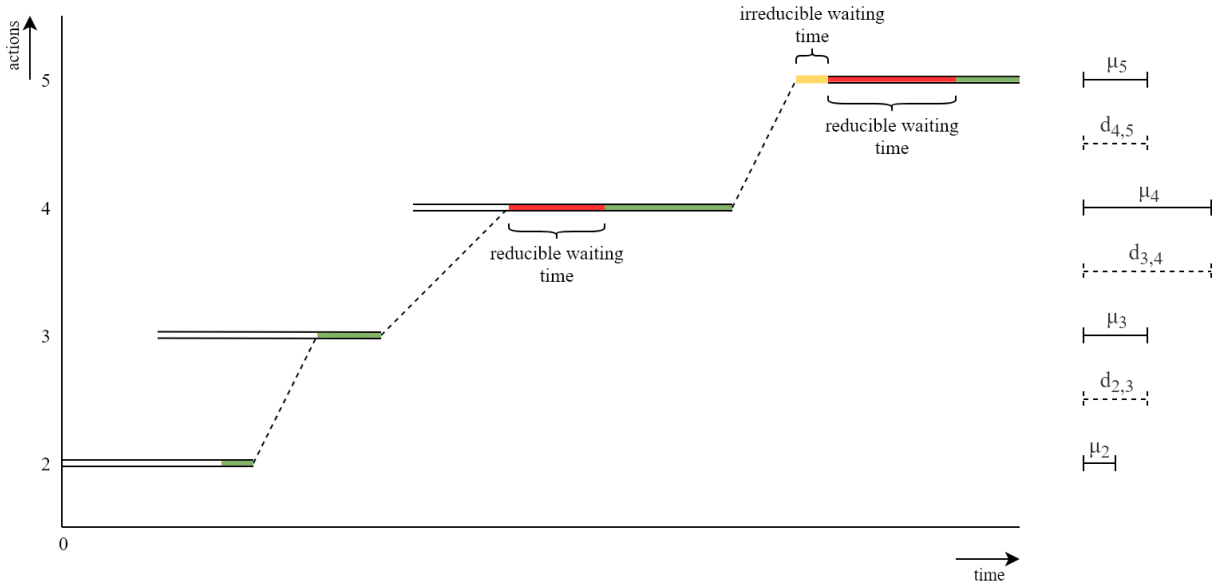


Figure 4.3: Latest tour

We optimize this by iteratively recalculating the start times using $\tau_i = \tau_{i-1} + d_{i-1,i} + \gamma_i$. This equation ensures that, given a fixed start time of the previous action τ_{i-1}, no unnecessary waiting time is added. We know, by using the way $w^{open}_{a^t_i}$ and $w^{close}_{a^t_i}$ are constructed, that $\tau_1 = w^{close}_1 - \mu_1$ is the latest possible moment a feasible tour can start. Besides that, $\gamma_1 = 0$ by definition. Total waiting time is defined as $\sum_{i=1}^{|t|} \gamma_i$. Each next action has either a waiting time of $\gamma_i = 0$ when the time window of the next action is already open upon arriving, or a waiting time of $\gamma_i = w^{open}_i - (\tau_{i-1} + \mu_{i-1} + d_{i-1,i})$. As γ_i depends on the start time of the previous task and γ_i is a decreasing function in τ_{i-1}, we know that increasing τ_{i-1} either decreases waiting time γ_i, or γ_i remains zero. Therefore, as Equation 4.10 yields minimal waiting time given the start time of the previous action, we know that $\tau_i = \tau_{i-1} + \mu_{i-1} + d_{i-1,i} + \gamma_i | \tau_{i-1}$ yields the optimal start time for τ_i. If we start with determining the optimal value of τ_1, we know the optimal value of τ_2, which allows us to determine the optimal value of τ_3, which we continue until we reach the start time of the last action of tour t, $\tau_{|t|}$. As the waiting time of the first action $\gamma_1 = 0$ by definition, and increasing τ_1 will never lead to a decrease in total waiting time, choosing the last possible start time t_1 guarantees that an optimal solution will be found. This leads us to Algorithm 3 to determine optimal start times, with respect to minimizing total Work Time.

Algorithm 3: Update start times

$\tau_1 = w^{close}_1 - \mu_1$

for $i = 2$ *to* $|t|$ **do**

  $\tau_i = \tau_{i-1} + \mu_{i-1} + d_{i-1,i} + \gamma_i$

end

4.4.2 Minimize delay of priority tasks

Algorithm 3 provides an approach to find a tour with start times that minimize total Work Time. These start times are optimal with respect to minimizing Work Time, but not necessarily optimal with respect to Weighted Delay of priority tasks. In exceptional cases and depending on objective weights π_2 and π_3, it might be possible that adding waiting time can reduce Weighted Delay and the total objective value. However, when optimizing the schedule of one tour, this is not preferred. Therefore, when optimizing Weighted Delay, the tour will be optimized in a lexicographic way, which in this case means that it is not allowed to increase total Work Time. As stated in Equation 2.5, Weighted Delay is defined as $f_3(\Pi) = \sum_{r \in R} p_r(\tau_r - s_r)$. When optimizing start times within a tour, it is not allowed to switch the order. Therefore, as Weighted Delay is reduced when values of τ_i are reduced, it is logical to shift the start times optimized using Algorithm 3 backwards as much as possible, as long as total Work Time, or total waiting time equivalently, does not increase. Note that, when waiting time occurs, so when $\sum_{i=1}^{|t|} \gamma_i > 0$, it is not possible to shift the tour backwards without increasing total waiting time. Therefore, we only need to consider the case when total waiting time equals zero. Figure 4.4 shows an example tour which can be created using Algorithm 3, but is not optimal with respect to Weighted Delay. It can be seen that within the time windows of actions a_3, a_4 and a_5, there is margin to shift the tour backwards.

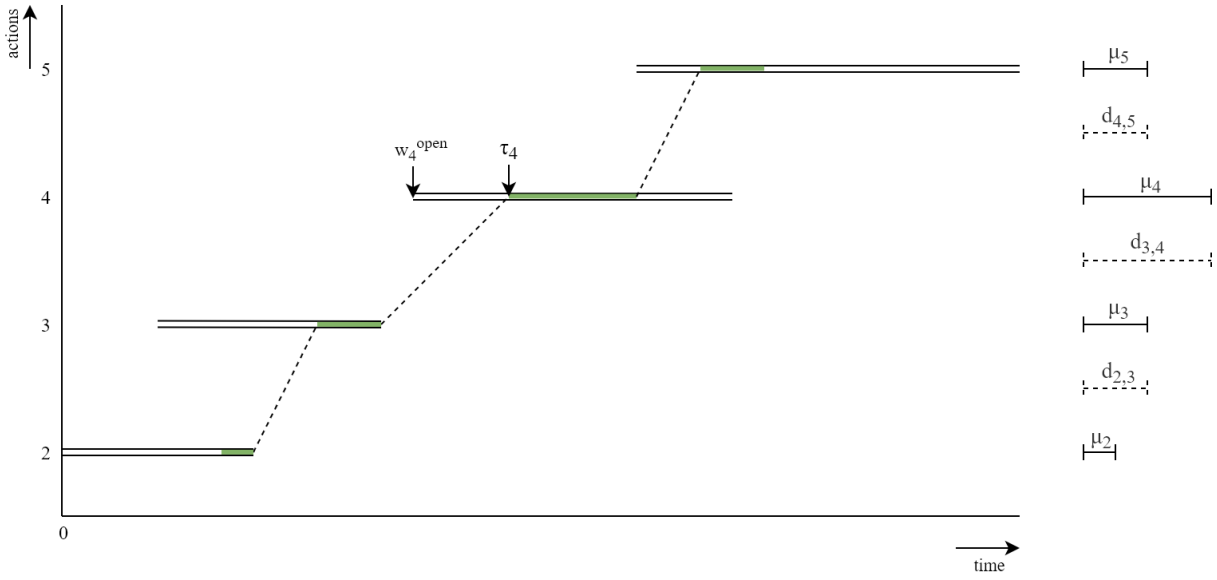


Figure 4.4: Optimal tour w.r.t. waiting time, not optimal w.r.t. Weighted Delay

The maximal shift can be calculated by determining the minimum margin within each time window. This margin is calculated by taking τ_i and subtracting w_i^{open}. This is given in Equation 4.11. After the maximum shift value is calculated, the start times can be moved backward by subtracting this value, so $\tau_i = \tau_i - \text{SHIFT}$. Then, the tour visualized in Figure 4.4 will be shifted backwards, which is shown in Figure 4.5

$$\text{SHIFT} = \min_{1 \leq i \leq |t|} \left\{ \tau_i - w_i^{open} \right\} \tag{4.11}$$

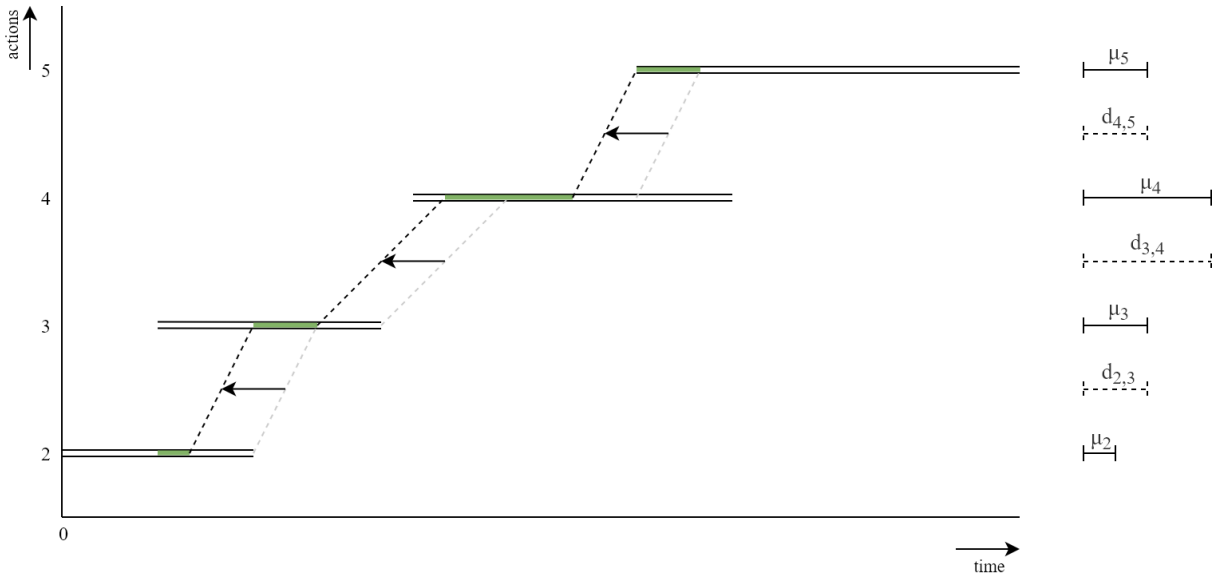


Figure 4.5: Optimal tour w.r.t. waiting time and Weighted Delay

Combining both Algorithm 3 and Equation 4.11 results in Algorithm 4:

Algorithm 4: Create best schedule from actions

$\tau_1 = w_1^{close} - \mu_1$

for $i = 2$ *to* $|t|$ **do**
  $\tau_i = \tau_{i-1} + \mu_{i-1} + d_{i-1,i} + \gamma_i$

end

if $\sum_{i=1}^{|t|} \gamma_i = 0$ **then**
  $\text{SHIFT} = \min_{1 \leq i \leq |t|} \{\tau_i - w_i^{open}\}$
  for $i = 1$ *to* $|t|$ **do**
    $\tau_i = \tau_i - \text{SHIFT}$
  end

end

4.4.3 Work Balance

Work Balance depends on the technician that has least total Work Time and the technician that has most total Work Time, as described in Equation 2.6. As described earlier, total Work Time is not taken into account when determining the optimal start times. However, it is calculated after the optimal start times are determined. The Work Balance equation can be decomposed per tour, where the value depends on whether technician k has most total Work Time, least total Work Time, or something in between. Let us define $c_4(t)$ as the contribution of tour t to the partial objective value of Work Balance $f_4(\Pi)$ in Equation 2.6. If technician k has the most amount of total Work Time, the value of $c_4(t)$ equals the Work Time of tour t. The value of $c_4(t)$ equals the Work Time of tour t made negative in case technician k has the least amount of total Work Time. In all other cases, tour t does not contribute to the partial objective value of Work Balance, which results in a value of zero. This is also formalized in Equation 4.12.

$$c_4(t) = \begin{cases} \tau_{a^t_{|t|}} - \tau_{a^t_1}, & \text{if } k = \text{argmax}_{k \in K} \left(\sum_{t \in T_k} \tau_{a^t_{|t|}} - \tau_{a^t_1} \right) \\ -(\tau_{a^t_{|t|}} - \tau_{a^t_1}), & \text{if } k = \text{argmin}_{k \in K} \left(\sum_{t \in T_k} \tau_{a^t_{|t|}} - \tau_{a^t_1} \right) \\ 0, & \text{otherwise} \end{cases} \tag{4.12}$$

4.4.4 Total costs of a tour

The total costs we assign to a tour consist of total Work Time, Weighted Delay of tasks in that tour, and its contribution to the Work Balance objective value as described in Equation 4.12. The costs of overtime and rescheduling tasks can be assigned to individual tours, however, this is not preferred for implementation. This is because overtime and rescheduling only occur in case no feasible solution can be found and using specialized operators, not within the ALNS framework and standard destroy and repair operators. Therefore, the total costs of a tour consist of the sum of Work Time in that tour, Weighted Delay and the value assigned to Work Balance in that tour, which is formulated in Equation 4.13.

$$c(t) = \pi_2(\tau_{a^t_{|t|}} - \tau_{a^t_1}) + \pi_3 \sum_{r \in t} \left(p_r(\tau_r - s_r) \right) + \pi_4 c_4(t) \tag{4.13}$$

4.5 Insertion and removal costs

In this section we explain how to calculate the costs of inserting or removing a task from a tour and how to incorporate these functions in the various destroy and repair operators.

4.5.1 Calculating full insertion costs

The insertion costs $c^+(r, t, i)$ of inserting task r as action after the i^{th} position in tour t can be calculated by using the total costs of a tour as described in Section 4.4.4. Let us define two tours, tour t and t' that belong to solutions Π and Π', where tour t excludes task r as one of its actions and tour t' includes task r. Besides this, tours t and t' are identical and the start times are optimized as described in Section 4.4 and all other tours in solutions Π and Π' are also equal. Then, we can calculate the costs of inserting task r by simply comparing the costs of both tours. This is formulated in Equation 4.14.

$$\begin{aligned} c^+(r, t, i) &= c(t') - c(t) \\ &= \pi_2(\tau_{a^{t'}_{|t'|}} - \tau_{a^{t'}_1}) + \pi_3 \sum_{r \in t'} \left(p_r(\tau_r - s_r) \right) + \pi_4 c_4(t') - \left(\pi_2(\tau_{a^t_{|t|}} - \tau_{a^t_1}) + \pi_3 \sum_{r \in t} \left(p_r(\tau_r - s_r) \right) + \pi_4 c_4(t) \right) \end{aligned} \tag{4.14}$$

This approach of calculating $c^+(r, t, i)$ is used for both the Greedy Insertion operator and the Regret-k Insertion operator. Furthermore, the cost function in Equation 4.14 can be split into 3 parts, in order to create partial objective cost functions for Work Time, Weighted Delay and Work Balance. Then, the insertion costs for the Greedy Work Time Insertion operator are given by Equation 4.15:

$$c_2^+(r,t,i) = c_2(t') - c_2(t) = \pi_2\big(\tau_{a^{t'}_{|t'|}} - \tau_{a^{t'}_1} - (\tau_{a^t_{|t|}} - \tau_{a^t_1})\big) \tag{4.15}$$

Similarly, cost functions c_3^+ and c_4^+ can be formulated for the partial objective cost functions of Weighted Delay and Work Balance, respectively.

4.5.2 Insertion costs based on travel time

The calculation of insertion costs described above involves multiples steps and thus requires more processing time. Therefore, it can be beneficial for the ALNS algorithm to include an easier approach for calculating insertion costs as well. Besides minimizing overtime and rescheduling, the most important partial objective is minimizing Work Time. When we only focus on minimizing Work Time, the ALNS algorithm will most probably also improve the total objective value. Work Time consists of travel time, service time and usually a small amount of waiting time. When we want to compare different positions for inserting a task, the service time of the task to be inserted remains equal. Therefore, we propose an approach where we calculate insertion costs solely based on the added travel time. Although minimizing travel time does not always result in less Work Time, it can also help in creating more flexible schedules. This will be explained using the following example:

> A tour consists of one task at the start of a day, and one task at the end of the day. These tasks cannot be scheduled closer to each other, due to restrictive time windows. Then, let us consider two feasible non-priority tasks to be inserted in between the other tasks: one with an added travel time of 15 minutes and one with an added travel time of two hours. Inserting either of both tasks results in a tour with exactly the same Work Time as before, which means that the insertion costs are zero, assuming this tour does not increase or reduce Work Balance. However, inserting the task with an added travel time of 15 minutes results in a tour with more space to add other tasks. In other words, the tour is more flexible regarding the insertion of more tasks.

Added travel time can be calculated in a straightforward way. The insertion costs based on travel time of adding task r in tour t after the i^{th} position, are calculated by adding the travel time from the previous task in tour t to task r, and from task r to the next task of tour t, minus travel time $d_{i,i+1}$. This is formulated in 4.16.

$$c^+_{\text{travel}}(r,t,i) = d_{i,r} + d_{r,i+1} - d_{i,i+1} \tag{4.16}$$

These insertion costs based solely on added travel time can be used to create faster performing repair operators to be used in the ALNS algorithm. As described in Section 4.2, both Quick Greedy Insertion and Quick Regret-k Insertion use $c^+_{\text{travel}}(r,t,i)$ as function to determining insertion costs.

4.5.3 Calculating full removal costs

The full removal costs of removing task r from tour t at the i^{th} position, or removing action a_i^t, can be calculated equally to the insertion costs in the previous section. Let us define the same two tours, tour t and t' that belong to solutions Π and Π', where tour t excludes task r as one of its actions and tour t' includes task r. Then, the full removal costs of removing action a_i^t are calculated by using $c^-(a_i^t) = c(t') - c(t)$. Similar to the calculation of insertion costs based on partial objective functions, the removal cost function $c^-(a_i^t)$ can be split up into c_2^-, c_3^- and c_4^-, which are used for the destroy operators Worst Work Time removal, Worst Weighted Delay removal and Worst Work Balance removal, respectively.

4.5.4 Removal costs based on travel time

Following the same reasoning as in Section 4.5.2, we want to include an approach that is computationally faster in calculating the removal costs of removing a task. Therefore, we propose to use Equation 4.17 in order to calculate the removal costs based solely on travel time.

$$c^-_{\text{travel}}(a_i^t) = d_{i-1,i+1} - d_{i-1,i} - d_{i,i+1} \tag{4.17}$$

The removal cost function given in Equation 4.17 is used in the Quick Worst Removal destroy operator, in order to find the action with highest cost based on travel time.

4.6 Infeasible solutions and interventions

In case not all tasks can be inserted, two approaches can be considered. The first approach is by assigning a large value to each unscheduled task. This value should be large enough to ensure that all feasible schedules have a better objective value than schedules with one or more unscheduled tasks. Similarly, every schedule with n unscheduled tasks should be better then every schedule with $n+1$ unscheduled tasks. Then, before the various destroy operators are used, the currently unscheduled tasks are added to the remaining set R^-. Subsequently, a destroy operator removes other tasks until a total of q tasks are added to the remaining set R^-. Using this approach of assigning high costs to unscheduled tasks, Algorithm 1 can be used without modifications. If this approach does not yield feasible solutions after a few iteration, a second approach will be used. The second approach can add overtime to one or more tours, or relax the time window of one or more fixed tasks, or any combination of these two measures. Let us define an intervention as either a relaxation of a fixed task, or a tour with overtime. The next subsection describes this second approach where special operators are used to apply one or more interventions to the current solution.

4.6.1 Interventions

In case no feasible schedule can be found, it is possible to make use of some special operators. These operators are applied as an extra step, prior to applying the different destroy and repair operators. We

propose some special operators that represent realistic changes, from a practical point of view, to the current schedule. The current schedule is defined by the tasks that are fixed to a specific moment in time. According to practice, the following approaches are used in order to create a feasible schedule. The availability time window of one or more technicians is enlarged, or one or more fixed tasks are rescheduled, or any combination of these two. Rescheduling means that the time window of a fixed task, which is set to a single day, will be relaxed. The new time window is set equal to the initial time window of that task, prior to fixing that time window. This allows the algorithm to reschedule the fixed task to another day, which might make it possible to schedule previously unscheduled tasks on this day. We will consider the following operators:

- Enlarging the time window of one tour of a technician on a given day;
- Enlarging the time windows of two technicians on a given day;
- Relaxing the time window of one or more fixed tasks.

The objective function contains the partial objective function $f_1(\Pi) = \sum_{t \in T} o_t + \sum_{r \in R_f} 1_{\{r \in R_f\}}$, which simply counts the number of tours with overtime and the number of rescheduled tasks. This means that the function $f_1(\Pi)$ can be regarded as the total number of interventions. Note that, as an extension, it is also possible to split the partial objective function $f_1(\Pi)$ such that it allows different weights to be assigned to overtime and relaxing a fixed task. This way, different orderings of interventions can be made, depending on the requirements of the maintenance company.

4.6.2 Enlarging the time window of one tour

When no feasible solution can be found, this means that the set $R^- \subset R$ is not empty. For the special operator where we enlarge the time window of a tour $t_{k,d}$ of one technician k on a certain day d, it makes sense to only check for tours that can contain at least one task $r \in R^-$ based on time windows, and the corresponding technician should meet the required skills for at least one task in $r \in R^-$. Let us define $T_{R^-} \subseteq T$ as the set of tours that satisfy both restrictions. Then, Algorithm 5 describes an approach to determine this set.

Algorithm 5: Determine set of promising tours for overtime

```
T_{R^-} ← ∅
forall t_{k,d} ∈ T do
    forall r ∈ R^- do
        if S_r ⊆ S_k and w^{open}_{r,d} ≠ ∅ then
            T_{R^-} ← T_{R^-} ∪ {t}
        end
    end
end
```

For this set T_{R^-}, we want to evaluate if it is possible to create a feasible schedule when allowing one tour t to have overtime. Adding overtime to tour t means that we enlarge the time window with amount

Δ, so the time window for tour $t_{k,d}$ becomes the interval $[w_{k,d}^{open}, w_{k,d}^{close} + \Delta]$. Then, after expanding the time window, there are multiple approaches to try find a feasible solution. The easiest approach is to use the (Quick) Greedy Insertion operator on the set R^-. We propose using Greedy Insertion instead of Quick Greedy Insertion, because Quick Greedy Insertion ignores Work Balance, which is important when assigning overtime to a technician. For example, it is not preferable to assign overtime to the technician that has most total Work Time. If Greedy Insertion fails, we propose to remove all unfixed tasks from tour t, then use the Quick Worst Removal operator to remove a certain percentage of other tasks, and finally use the Regret-k Insertion operator to repair the solution. If this results in a feasible solution, we keep track of this solution and its corresponding objective value. If this still results in an infeasible solution, we retry this approach for all $t \in T_{k,d}$. Algorithm 6 describes this in detail. It should be noted that adjusting the time window of a tour should only have effect in that particular iteration, therefore, we make a copy of the solution before changing the time window. After this algorithm finished and yields multiple feasible solutions, we use these solutions as input for the ALNS algorithm, described in Algorithm 1.

Algorithm 6: Scheduling with overtime of a single tour

input : Π, best infeasible solution so far; T_{R^-}, set of suitable tours for overtime

output: solutions

solutions $\leftarrow \varnothing$

forall $t \in T_{R^-}$ **do**

  $\Pi^c \leftarrow$ `copy`(Π)

  $w_{k,d}^{close} \leftarrow w_{k,d}^{close} + \Delta$ *; adjusts time window of copied solution*

  $\Pi' \leftarrow$ `greedyRepair`(Π^c)

  if `feasible` (Π') **then**

    solutions $\leftarrow$ solutions $\cup \{\Pi'\}$

  else

    $\Pi'' \leftarrow$ `worstDestroy`(t, Π)

    $\Pi''' \leftarrow$ `regretRepair`(Π'')

    if `feasible` (Π''') **then**

      solutions $\leftarrow$ solutions $\cup \{\Pi'''\}$

end

4.6.3 Enlarging the time window of two tours

When allowing two tours to have overtime, the number of possible tours to explore grows quadratically compared to a single tour with overtime. Just like in Algorithm 6, using the Regret-k Insertion repair operator has a higher probability of finding a feasible solution, but is computationally more intensive compared to Greedy Insertion. This is not a problem when exploring the possibilities for a single tour with overtime, but as the number of combinations for two tours with overtime is much larger, it makes sense to adjust the algorithm in some way. Therefore, instead of exploring all tours $t \in T_{R^-}$ and then returning all feasible solutions, we propose to terminate the algorithm immediately after finding a fixed

number of feasible solutions, e.g. after one feasible solution.

4.6.4 Relaxing the time windows of fixed tasks

When a task is fixed, the time windows of that task are reduced to a single day. Relaxing the time window of a fixed task means that the initial time windows of that task are restored. For example, a task with time windows from Monday to Friday which is fixed to Wednesday, can be relaxed such that the time windows from Monday to Friday are available again for scheduling. After relaxing the time windows of a single task, the task will be added to the remaining set $R^- \subseteq R$ in order to reschedule it possibly to another day. Following a similar reasoning as for overtime, it makes sense to reduce the set of fixed tasks we want to explore for relaxation. This can be done by first checking on which days the time windows of tasks $r \in R^-$ occur, then checking which fixed tasks $r \in R_f$ are fixed on one of those days. Let us define this set of fixed tasks we want to explore for relaxation $R_f^- \subseteq R_f$. Then, set R_f^- can be determined using Algorithm 7.

Algorithm 7: Determine set of promising fixed tasks for rescheduling

```
D⁻ ← ∅
R_f⁻ ← ∅
forall d ∈ D do
    forall r ∈ R⁻ do
        if w_{r,d}^{open} ≠ ∅ then
            D⁻ ← D⁻ ∪ {d}
            break
        end
    end
end
forall r ∈ R_f do
    forall d ∈ D⁻ do
        if w_{r,d}^{open} ≠ ∅ then
            R_f⁻ ← R_f⁻ ∪ {r}
            break
        end
    end
end
```

As relaxing the time window of a fixed task $r \in R_f^-$ has the purpose of rescheduling that task, it is a logical choice that task r is added to set R^-. Then, the same destroy and repair operators as described in Section 4.2 can be used to find new solutions. In order to explore different solutions efficiently until a feasible solution is found, we propose to use an adjusted version of the ALNS algorithm, Algorithm 8, that is more focused on finding a feasible solution rather than improving a current solution. The algorithm is described in a way that it allows for relaxing a single fixed task, or relaxing multiple fixed tasks. Note

that the algorithm is not computed in parallel like in Algorithm 1, as it is more complex to terminate all subprocesses correctly after a feasible solution has been found. Besides that, the algorithm omits simulated annealing, as we are only interested in finding a feasible solution. Also, this algorithm either outputs a feasible solution or returns nothing, in case no feasible solution is found. Finally, in order to limit the amount evaluations, it is possible to evaluate only a limited number of random combinations, instead of all combinations. Although this decreases the change of finding a feasible solution, this keeps the total runtime reasonable.

Algorithm 8: Relaxing fixed task(s)

input : Π, initial solution; Θ^-/Θ^+, set of destroy/repair operators; k, number of fixed tasks that will be relaxed

output: Π^*

$\Pi^* = \varnothing$

forall combination $\in$ `takeCombinations`(R_f^-, k) **do**

  $\Pi' \leftarrow$ relax$(\Pi,$ combination$)$

  $\Pi' \leftarrow$ `addToRemainingSet`$(\Pi',$ combination$)$

  for I^{relax} iterations **do**

    $destroy \leftarrow$ select(Θ^-)

    $repair \leftarrow$ select(Θ^+)

    $\Pi'' \leftarrow repair\big(destroy(\Pi')\big)$

    if `feasible`(Π'') **then**

      $\Pi^* \leftarrow \Pi''$

      break forall

    end

  end

end

4.7 Initialization

In order to create one or more initial solutions, we can reuse the repair heuristics of the ALNS framework. First, we need to initialize some variables. We create tours $t \in T$ for all technicians $k \in K$ on all days $d \in D$. Then, each tour t on day d is initialized with only two actions, namely a_1^t and a_2^t. This implies that $a_2^t = a_{|t|}^t$. The time windows belonging to a_1^t and a_2^t are initialized for all tours by setting $w_{a_1^t}^{open}$ and $w_{a_2^t}^{open}$ equal to the start of the availability window of the technician k on day d belonging to $t_{k,d}$, minus 60 minutes. This allows technicians to travel at most 60 minutes to their first task, outside of their availability time windows. This means that the first task can take place at the start of the availability time window of a technician, if travel time is less than 60 minutes. Therefore, the intervals $[w_{a_1^t}^{open}, w_{a_1^t}^{close}]$ and $[w_{a_2^t}^{open}, w_{a_2^t}^{close}]$ are set equal to $[w_{k,d}^{open} - 60, w_{k,d}^{close}]$. Unlike tasks, the first and last action do not have a service duration, so we initialize the service durations of these actions as $\mu_{a_1} = \mu_{a_{|t|}} = 0$. Then, in order to create an initial solution, we add all tasks to the remaining set R^- and either use the greedy

repair operator or the Regret-k repair operator to add all tasks sequentially to a solution. This approach might not yield a feasible solution, but as described in Section 4.6, this is not necessarily a problem. An infeasible initial solution can still be used as input for the ALNS algorithm, as this infeasible solution is usually fixed within a few iterations.

4.8 Performance improvements

In this section, we propose several approaches to improve performance when solving large datasets.

4.8.1 Stepwise initialization

In case the set of tasks R is large, the initialization can take a long time. Therefore, it might be beneficial to build a solution stepwise. For example, a solution for an instance with 200 tasks can be split in subsets of 20 tasks. This way, in each iteration only 20 tasks are compared and added. Clearly, this approach usually finds inferior solutions compared to adding all tasks at once, but as this solution is only used as initial solution for the ALNS algorithm, it can be beneficial to reduce the total running time. This approach is formally stated in Algorithm 9. In this algorithm, the set $R^-_{step} \subseteq R^-$ is defined as the set of tasks that is taken in one step of size η. Note that the size of set R^-_{step} is smaller in the last iteration, in case the total number of tasks is not exactly a multiple of η.

Algorithm 9: Stepwise initialization

input : Π^0, Empty solution without tasks added; R^-, set of unscheduled tasks; η, step size

output: Π^0

while $R^- \neq \varnothing$ **do**

  $R^-_{step} \leftarrow$ `getNextTasks`(R^-, η)

  $R^- \leftarrow R^- \cap \overline{R^-_{step}}$; *Remove* R^-_{step} *from set* R^-

  $\Pi^0 \leftarrow$ `greedyRepair`(Π^0, R^-_{step})

end

4.8.2 Stepwise repair operators

Similar to the process of building an initial solution, stated in 9, repairing solutions in the ALNS algorithm from Algorithm 1 can take relatively long for large datasets. The amount of tasks to be reinserted using repair operators depends on the total amount of tasks R and on the destroy factor. It might improve the performance of the ALNS algorithm on datasets with many tasks, as this reduces runtime, which allows more iterations to be executed compared to evaluating all tasks at once. If stepwise repair operators are used, this will be denotedas the repair stepsize, which defines the amount of tasks that are evaluated in a single step.

Chapter 5

Experiments

In this chapter we will test the performance of the ALNS algorithm, described in Algorithm 1, and the ability to find feasible schedules when a part of the tasks is fixed. First, a brief description of data generation will be given in Section 5.1. Next, we will describe how weights and parameter values are chosen and tuned using two experiments in Section 5.2. Then, Section 5.3 describes an experiment to test the performance and scalability of our implementation of the model, where no fixed tasks are included. Lastly, Section 5.4 includes a conducted experiment concerning the construction of an initial schedule and rescheduling on a daily basis. In such an experiment, interventions such as rescheduling of fixed tasks or adding overtime will be needed. All experiments will be conducted on randomly generated data, as there is no high quality data from practice made available for this research.

5.1 Generating problem instances

This section explains how clients, tasks and technicians are generated for problem instances in order to be used in different experiments. Besides that, the weights of the objective function are described.

Clients In the next experiments, clients are generated with unique addresses. Subsequently, all tasks are randomly assigned to one of these clients. The addresses will be uniformly random generated coordinates on the interval $[0, 1]$. Then, all travel times will be calculated by using the Euclidean distance and multiplying this value by $60\sqrt{2}$. This makes the maximum possible travel time between two clients at most 2 hours. Besides that, clients will be assigned availability windows per day. For the experiments in this research, these availability time windows will the same for all clients.

Tasks Each task will be randomly assigned to one of the generated clients. Next, each task will have a randomly generated service duration. In accordance with practice, the service durations in minutes will be on the interval $[15, 240]$, where only multiples of 15 minutes are chosen. Besides that, each task has an assigned required skill. In each experiment, the amount or fraction of tasks that require certain skills will be predetermined.

Technicians Each technician will be generated with a unique home address. Similar to the locations of clients, the home addresses are uniformly random generated coordinates on the interval $[0, 1]$. Next, each technician will be assigned a certain skillset, where the amount of technicians that have a certain skillset will be predetermined. Otherwise, the unlucky scenario could occur where no technicians with a certain skillset are generated, which would lead to infeasible instances. Finally, each technician will be assigned a series of availability time windows. All technicians will be assigned to full time schedules, which means that each technician will have availability time windows on the interval $[0, 540]$ on all working days. These time windows are equivalent to a working day of 9 hours. In real life applications, different time windows can be assigned to each day, but, as the problem does not necessarily become more difficult to solve, the experiments will use the same time windows for all days.

5.1.1 Objective function weights

The weights π_1 to π_4 are determined before the parameters are tuned and the experiments are conducted. As discussed earlier, weight π_1 will be set to such a high number, that every solution satisfies $z(\Pi) < z(\Pi')$, if and only if $f_1(\Pi) \leq f_1(\Pi')$. In our implementation, this weight is not used, as we use a lexicographic ordering when using interventions. The other partial objectives f_2, f_3 and f_4 are all measured in time, which makes them easy to compare. It should be noted that the partial objective of Weighted Delay, f_3, is actually measured in weighted time. This means that time is multiplied by the priority value, which, for our experiments, will be a discrete value of 0, 1 or 2. The experiments will resemble a realistic case from practice. It is stated that, after minimizing the number of interventions, minimizing the total amount of Work time is most important. Therefore, we assign $\pi_2 = 2$. Furthermore, as Weighted Delay already has priority values of 1 or 2 and can take much larger values than the value for work balance, we assign the same weight to Weighted Delay as to work balance, which is $\pi_3 = \pi_4 = 1$. This translates to the following interpretations of weights:

- Reducing Work time by 30 minutes is equally good as improving work balance by 1 hour. This follows from $30\pi_2 = 60\pi_4 \Leftrightarrow 30 \cdot 2 = 60 \cdot 1$
- Improving Weighted Delay by moving a task r with priority 2 ($p_r = 2$) earlier in time by 1 hour, at the expense of increased total Work time, is only beneficial if the added Work time is smaller than 1 hour. This follows from $60p_r\pi_3 = 60\pi_2 \Leftrightarrow 60 \cdot 2 \cdot 1 = 60 \cdot 2$

5.2 Parameter tuning experiments

In this section two smaller experiments are executed in order to find a good set of parameters for the ALNS algorithm. The parameters used in the ALNS algorithm are:

- Stopping criterion
- I^m, the number of master iterations

- K, the number of solutions that are used for further exploration within one master iteration. This parameter is the sum of the following two parameters
  - K_{best}, which describes number of times the best found solution is used in every master iteration
  - K_{other}, which describes the number of other unique solutions that are used in every master iteration
- I^p, the number of local iterations
- ρ^+, ρ^-, the initial weights of the destroy and repair operators
- Destroy factor, the fraction of tasks that is removed in every iteration
- Destroy decay factor, the factor that reduces the destroy factor in every local iteration
- C_0, the initial temperature of the Simulated Annealing acceptance method.
- Cooling factor on the interval $(0, 1)$, which determines how quick the temperature C decreases in the Simulated Annealing function
- Score function weights ω_1 to ω_4, which control how the destroy and repair operators should be updated, based on their performance
- λ, which controls how much the weights can be adjusted in one iteration
- Repair stepsize, which controls how many tasks the repair operators evaluate at once

The parameter K is split into two parameters, as testing showed that it is beneficial to explore the neighborhoods of the best solution multiple times in every master iteration, instead of exploring the neighborhoods of the K-best solutions only once.

Stopping criterion The ALNS algorithm needs to have a stopping criterion in order to stop. One stopping criterion is to stop after a certain amount of master iterations. However, this stopping criterion does not use any information of the performance of the algorithm, such as how much improvement there is in the last iterations. The following stopping criteria are considered:

- Setting a maximum runtime
- Stop when the algorithm finds no improved best solution for a certain amount of iterations
- Stop when the algorithm improves only $(\cdot)\%$ in a certain amount of iterations, e.g. at most 1% in 10 iterations

These stopping criteria have different advantages compared to each other. For example, stopping after a certain amount of time makes it easier to compare the performance of different parameters, while stopping after the solution does not improve after a certain amount of iterations usually yields better objective values. Therefore, the next experiments use different stopping criteria.

Data For the parameter tuning experiments, all information is known when making a schedule. There are no tasks that are added to the schedule at a later moment in time. Table 5.1 describes the data that will be used. Each experiment will be carried out using 5 different datasets, where client locations, technician locations, task-to-client assignments and task durations are randomly generated.

Table 5.1: Data description of parameter tuning experiments

Object	Value
# days	2
# clients	30
# technicians	12
# technicians with skillset $[a]$	6
# technicians with skillset $[a, b]$	3
# technicians with skillset $[a, c]$	3
Technician availability time windows (hours)	(0,9) hours for all days
# tasks	80
# tasks that require skill a	60
# tasks that require skill b	10
# tasks that require skill c	10
# tasks with priority 0	50
# tasks with priority 1	20
# tasks with priority 2	10
Time windows priority 0 tasks	(0,9) hours for all days
Time windows priority 1 tasks	(0,9) hours for all days
Time windows priority 2 tasks	(0,9) hours for all days

Operators In this experiment, we will make use of all destroy operators described earlier. The destroy operators are given in Table 5.2:

Table 5.2: Destroy operators to be used in parameter tuning experiments

	Operator	Abbreviation
1	Random Destroy	RD
2	Quick Worst Removal	QW
3	Worst Removal	W
4	Worst Work Time Removal	W-WT
5	Worst Weighted Delay Removal	W-WD
6	Worst Work Balance Removal	W-WB

Preliminary testing on this test data showed that the standard Regret-k Insertion operator largely increases runtime. Although this operator can help finding better solutions, it is excluded for this experiment. On the other hand, the operator Quick Regret-k insertion is much faster than the standard

Regret-k operator and is therefore included in these experiments. The repair operators to be used in this experiment are given in Table 5.3:

Table 5.3: Repair operators to be used in parameter tuning experiments

	Operator	Abbreviation
1	Quick Greedy Insertion	QG
2	Quick Regret-3 Insertion	QR3
3	Greedy Insertion	G
4	Greedy Work Time Insertion	G-WT
5	Greedy Weighted Delay Insertion	G-WD
6	Greedy Work Balance Insertion	G-WB

5.2.1 Experiment 1: Destroy factor, local iterations and repair stepsize

The first experiment will be focused on finding good values for the destroy factor, the number of local iterations and the repair stepsize. In this experiment, we start by choosing some parameters, as it is not tractable to optimize all combinations of parameters at once. Table 5.4 shows the values that will be fixed.

Table 5.4: Set parameter values for Experiment 1

Parameter	Value
C_0(Initial SA temperature)	$\frac{1}{10}$ of currently best objective value from last master iteration
SA Cooling factor	0.9
ρ^+	1 for all repair operators
ρ^-	1 for all destroy operators
λ	0.9
ω_1 (Best solution)	8
ω_2 (Improved solution)	4
ω_3 (Accepted solution)	2
ω_4 (Rejected solution)	1
K_{best}	8
K_{other}	8

Then, Table 5.5 lists the values for the different parameters to be tuned using this experiment. All combinations between these values will be tested. As each combination will be tested on six different datasets, this means that there will be a total of $2 \times 3 \times 2 \times 6 = 72$ scenarios executed. All scenarios use a maximum runtime of 30 minutes as stopping criterion, as this makes it easy to compare the performance of different parameters.

Table 5.5: Tuning values for Experiment 1

Parameter	Values
Destroy factor	0.2, 0.3
Local iterations	10, 25, 50
Repair step size	10, all tasks

5.2.2 Experiment 1: Results

To make the differences more clear between the various tested scenarios, all objective values will be indexed based on the best found solution for each dataset. Table 5.6 shows the mean indexed objectvies of the six datasets for all scenarios. This shows that the best found parameters are a destroy factor of 0.2, 50 local iterations and using stepwise repair operators with a stepsize of 10. It also shows that using the parameters 0.2, 50 and no stepwise repair operators performs only slightly worse than the best set of parameters.

Table 5.6: Indexed mean objectives of Experiment 1

Destroy factor	Local iterations	Repair stepsize	Objective
0.2	10	10	1.045
		all tasks	1.043
	25	10	1.039
		all tasks	1.052
	50	10	**1.015**
		all tasks	1.024
0.3	10	10	1.064
		all tasks	1.070
	25	10	1.090
		all tasks	1.076
	50	10	1.028
		all tasks	1.033

Figure 5.1 shows how the indexed mean objective improves over time, when using the best found parameters. Note that the initial horizontal line is caused by infeasible solutions at the start of the execution. The first solutions where all tasks are scheduled for all 6 datasets are found after around 2 minutes. Furthermore, we see that the indexed mean objective first improves quickly, but improves only very little in the last minutes. The line is not completely horizontal in the last part, which suggests that solutions can possibly be improved more by extending the maximum runtime, or by using a different stopping criterion. Note that the runtime usually increases when using the stopping criteria of stopping after either the best solution is not improved, or after stopping when the improvement is less than a

predetermined percentage after a certain amount of iterations.

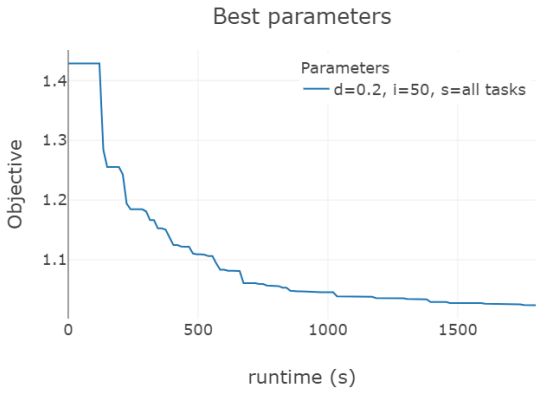


Figure 5.1: Indexed mean objective over time using best found parameters

Other insights of the performance of different parameter values can be obtained by grouping experiments based on the value of one specific parameter. Figure 5.2 shows the performance per value of the three different parameters.

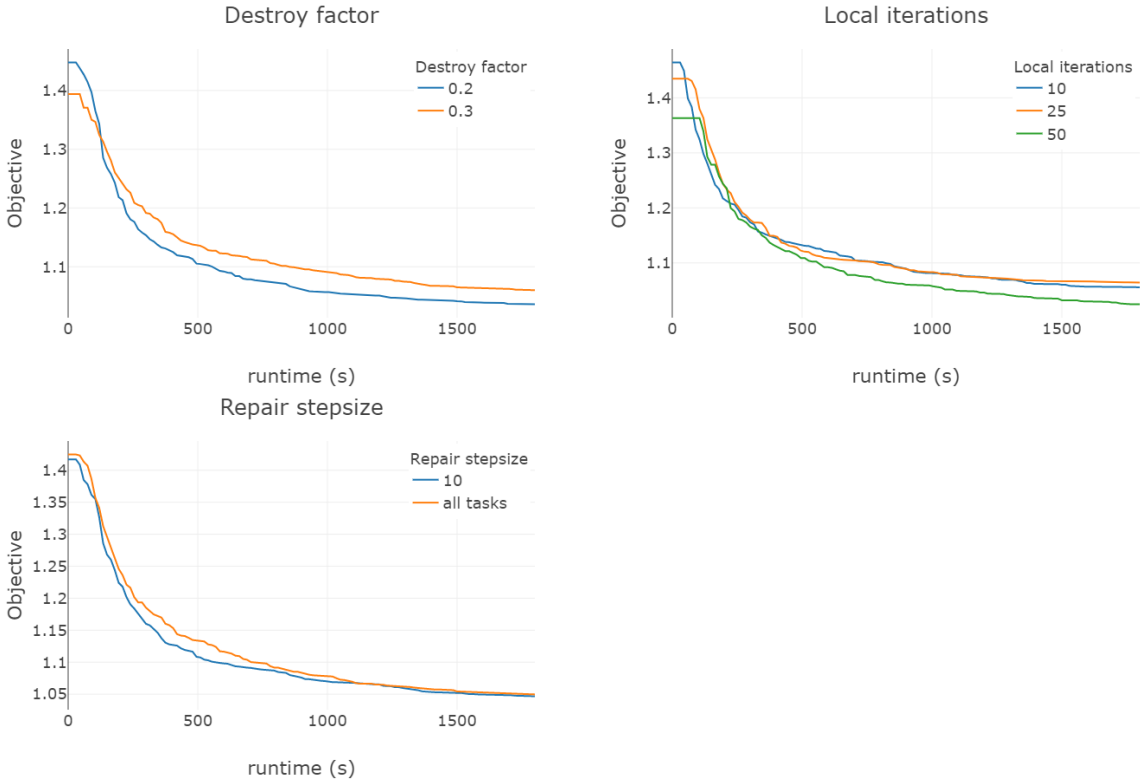


Figure 5.2: Performance over time clustered per different parameter values

Firstly, this shows that using a destroy factor value of 0.2 leads, on average, to better results. It also shows that it can be beneficial to use a larger destroy factor at the start of the execution, as this yields slightly better initial solutions. Furthermore, this may suggest that using a smaller destroy factor than 0.2 can be beneficial. However, some testing showed that decreasing the destroy factor can lead to inferior solutions, as the algorithm gets easier stuck in local optima. As a possible improvement for further

research, it might be interesting to use a variable destroy factor, for example, by using a random destroy size on a given interval. This is also suggested by Avci and Avci (2019). Another possible improvement can be made by decreasing the destroy factor over time, until no new improvements can be found. Then, in order to escape local minima, the destroy factor can be increased again. In the remaining experiments, a fixed destroy factor will be used.

Secondly, the figure clearly shows that using 50 local iterations yields better solutions then using either 10 or 25 local iterations. This may suggest that the performance of the algorithm can be further improved by increasing the amount of local iterations. However, after testing the algorithm on differently sized instances, it appears that the best number of local iterations depends on the size of an instance. For example, using more than 50 local iterations on a larger problem instance leads to worse results than using less than 50 iterations.

Finally, we see that the repair stepsize does not have much effect on the performance of the algorithm, although the best performing set of parameters uses no stepwise repair operators. Note that using no stepsize is the same as using a stepsize of all tasks. This can possibly be explained by the destroy factor used. When using a destroy factor of 0.2 or 0.3, only 16 or 24 are removed in every iteration, respectively. Reinserting 24 tasks by evaluating them all at once does not take much time, thus the potential benefit of using a repair stepsize is not large in this case. However, when using larger datasets, there can be a benefit of using stepwise repair operators.

5.2.3 Experiment 2: Simulated Annealing and lambda

The parameters we want to optimize in this experiment are the temperature used for Simulated Annealing (SA), the cooling factor used for SA, and the value for λ, which determines how much the weights are adjusted in every iteration. A value of $\lambda = 0.9$ means that 90% of the previous weight will be used and 10% of the score function will be used, given in Equation 4.3. In the first experiment, a temperature of $\frac{1}{10}$ of the currently best found solution is used, as early testing showed that this value resulted in good performance. Preliminary testing also showed that using values much larger than $\frac{1}{5}$ of the objective value, for example by using the full objective value, results in accepting many inferior solutions. Consequently, not many new best solutions are found, which leads to worse overall performance. The initial temperature of SA is updated at the start of every master iteration, and subsequently decreases in every local iteration with a cooling factor of 0.9. In this experiment, we compare different values for the initial SA temperature. The values we will consider for this experiment are given in Table 5.7:

Table 5.7: Tuning values for Experiment 2

Parameter	Values
λ	0.9, 0.75, 0.6
C_0 (Initial SA temperature)	$\frac{1}{10}$, $\frac{1}{5}$
Cooling factor	0.9, 0.95

The other parameters used in this experiment will be equal to the parameters in Experiment 1, which

are stated in Table 5.4. Additionally, the best found parameters from Experiment 1 will be used. These parameters are stated in Table 5.8:

Table 5.8: Best found parameters from Experiment 1 to be used in Experiment 2

Parameter	Values
Destroy factor	0.2
Local iterations	50
Repair step size	10

This experiment also uses the same stopping criterion as Experiment 1, which is a runtime of 30 minutes. The total number of scenarios to be solved for this experiment is $3 \times 2 \times 2 \times 6 = 72$.

5.2.4 Experiment 2: Results

Table 5.9 shows the mean indexed objective values for the different values of λ, SA temperature and the cooling factor. It shows that on average the best results are obtained using $\lambda = 0.9$, SA temperature of 0.1 and a cooling factor of 0.9.

Table 5.9: Indexed mean objectives of Experiment 2

Lambda	SA temperature	Cooling factor	Objective
0.60	0.1	0.90	1.027
		0.95	1.042
	0.2	0.90	1.046
		0.95	1.026
0.75	0.1	0.90	1.020
		0.95	1.038
	0.2	0.90	1.049
		0.95	1.040
0.90	0.1	0.90	**1.017**
		0.95	1.025
	0.2	0.90	1.044
		0.95	1.050

Similarly to the Experiment 1, the results of the experiments are also grouped based on the value of one specific parameter. These results are shown in Figure 5.3.

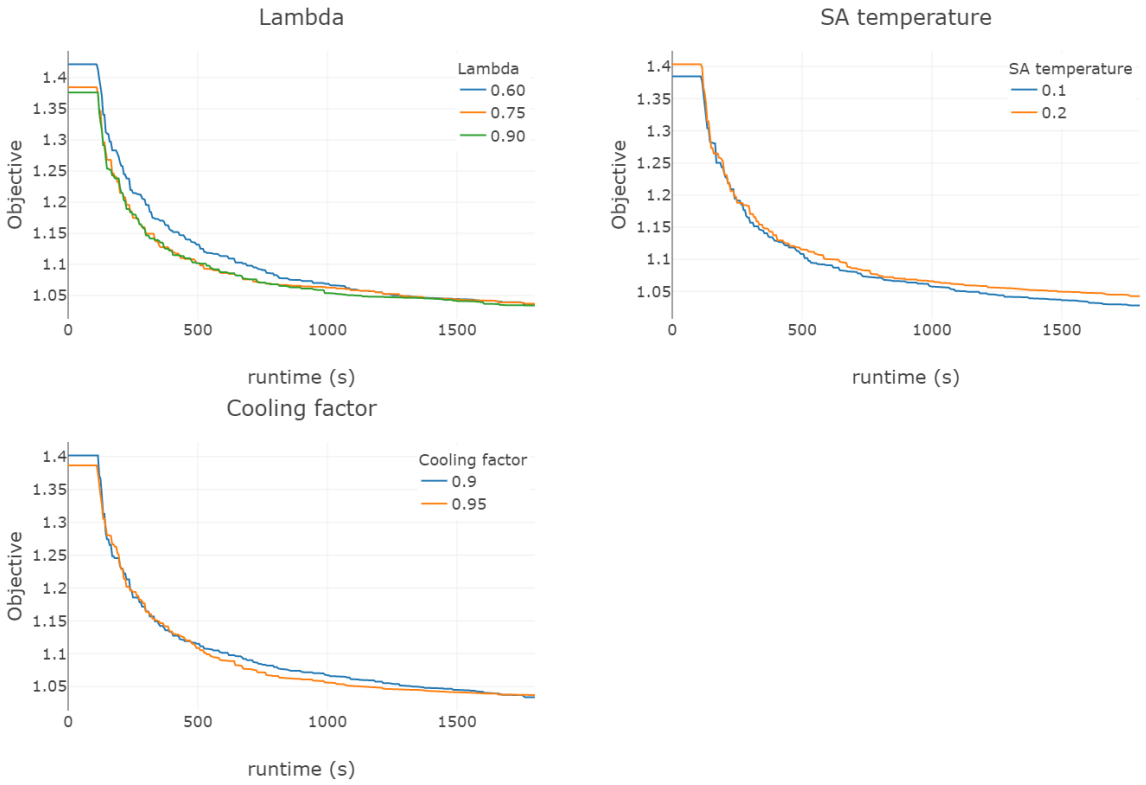


Figure 5.3: Performance over time clustered per different parameter values

Firstly, it can be seen that the different values for λ do not have a large effect on the final objective values, although by using $\lambda = 0.6$ the performance of the algorithm is worse during the first 15 minutes. Secondly, it can be seen that using a SA temperature of $\frac{1}{10}$ of the last best objective value yields to an overall better performance. It may be possible that using smaller fractions than $\frac{1}{10}$ can yield even better solutions, because this means that less inferior solutions will be accepted. However, this also reduces the possibility of escaping local minima. Finally, it can be seen that the two values for the cooling factor perform equally good during the first 8 minutes. Then, using a cooling factor of 0.95 performs slightly better for a while, except for the final iterations. Therefore, we conclude that the isolated effects of λ and the cooling factor are small, but tuning them in combination with other parameters contributes to an overall better performance.

Operators Now, let us analyze the performance of the different repair and destroy operators, based on the experiment with best found parameters. Table 5.10 and Table 5.11 give detailed results of the performance of the different destroy and repair operators, respectively. Both tables show how often each operator is relatively used, and how often each operator yielded a new best, improved, equal, accepted or rejected solution, relatively. Note that Accepted and Rejected are solutions that are worse than the current local solution. On the contrary, Best describes solutions which are better than the currently best found solution.

Table 5.10: Relative usage count and successes of destroy operators

Destroy operator	Best	Improved	Equal	Accepted	Rejected	**Total**
Random Destroy (RD)	0.55%	1.84%	0.04%	1.50%	9.29%	**13.21%**
Quick Worst (QW)	0.13%	2.99%	1.07%	3.22%	9.83%	**17.25%**
Worst (W)	0.15%	3.12%	1.00%	3.22%	9.66%	**17.15%**
Worst Work Time (W-WT)	0.18%	3.07%	1.09%	3.33%	9.78%	**17.45%**
Worst Weighted Delay (W-WD)	0.19%	3.18%	1.07%	3.28%	9.72%	**17.44%**
Worst Work Balance (W-WB)	0.18%	3.12%	1.03%	3.35%	9.82%	**17.50%**
All destroy operators	1.38%	17.32%	5.30%	17.90%	58.10%	**100%**

When looking at the destroy operators in Table 5.10, we see that all operators perform about equal, except the random destroy operator. The destroy operator is used around 25% less often, yields worse results with respect to improved or accepted solutions, but finds around 3 times more often a new best solution, compared to the other operators. A possible explanation for this, is that random destroy helps with escaping local optima.

Table 5.11: Relative usage count and successes of repair operators

Repair operator	Best	Improved	Equal	Accepted	Rejected	**Total**
Quick Greedy (QG)	0.16%	1.54%	0.53%	2.36%	10.07%	**14.66%**
Quick Regret-3 (QR3)	0.15%	1.67%	0.70%	2.41%	10.01%	**14.94%**
Greedy (G)	0.27%	3.58%	1.00%	3.44%	9.53%	**17.81%**
Greedy Work Time (G-WT)	0.29%	3.53%	0.99%	3.23%	9.45%	**17.48%**
Greedy Weighted Delay (G-WD)	0.27%	3.59%	1.08%	3.36%	9.59%	**17.87%**
Greedy Work Balance (G-WB)	0.23%	3.41%	1.00%	3.11%	9.48%	**17.23%**
All repair operators	1.38%	17.32%	5.30%	17.90%	58.10%	**100%**

When we look at the performance of the repair operators, we see that the two quick operators perform in general worse than the other operators. Besides that, we see that the other operators all perform equally well. Next, Table 5.12 and Table 5.13 show the amount of time of each destroy or repair operator, relative to the total time spent destroying or reparing solutions, respectively.

Operator	Relative runtime
Quick Worst	1.2%
Random	0.1%
Worst	24.8%
Worst Work Time	24.6%
Worst Weighted Delay	24.6%
Worst Work Balance	24.7%
Total of destroy operators	100%

Table 5.12: Relative time of each destroy operator spent destroying solutions

When we look at the relative time spent of each destroy operator in Table 5.12, we see that the relative runtime of the Random destroy destroy operator is lowest, which makes sense as no cost calculations are necessary. Next, we see that the Quick Worst operator is around 20 times faster than the standard Worst operators, which is as expected as stated in Section 4.5.4. Interestingly, this operator performs about equal in terms of finding good solutions, compared to the standard Worst operators, as can be seen in Table 5.10. Moreover, we see that all four standard Worst operators have roughly the same total runtime. This can be explained by the fact that the same calculation is done in the implementation of the algorithm.

Operator	Relative runtime
Quick Greedy	1.7%
Quick Regret-3	19.5%
Greedy	20.5%
Greedy Work Balance	19.0%
Greedy Weighted Delay	20.0%
Greedy Work Time	19.4%
Total of repair operators	100%

Table 5.13: Relative time of each repair operator spent repairing solutions

Using Table 5.13, it can be seen that the Quick Greedy Operator is over 10 times faster than the other greedy operators. However, we see that the Quick Regret-3 repair operator has roughly the same runtime as the standard Greedy operators, while the Quick Regret-3 operator has about the same performance as the Quick Greedy operator, according to Table 5.11. Because the performance of the Quick Regret-3 operator is worse than the standard Greedy operators, while it has the same runtime, it may be beneficial to exclude this operator from the algorithm, in order to improve performance. However, more research should be done in order to determine if the Quick Regret-3 operator has added value to the solution quality. For example, it should be checked whether the best found objective value worsens when this operator is excluded. The same check can be done on the various kinds of Greedy repair operators used

in this research, as they all appear to have the same performance.

5.3 No fixed tasks and differently sized datasets

The following experiment shows how the ALNS algorithm performs on differently sized data sets, where we test the performance based on runtime and objective value. The objective value can be split in 4 parts, as described in 2.2, which allows us to report on the progress of the different partial objective values, e.g. total Work time and Weighted Delay of priority tasks. The goal of this experiment is to determine the problem sizes that can be solved adequately using our implementation of the ALNS algorithm. In this experiment we will use the best found parameters from Experiment 1 and Experiment 2. These values are shown in Table 5.14. There are two changes. Firstly, the Quick Regret-k Insertion operator is excluded, because preliminary testing showed that this operator performs much slower on larger instances. Secondly, the Quick Greedy repair operator and the Quick destroy operator are given a larger initial weight, which helps speeding up the first master iteration. This is done in order to find good feasible initial solutions quickly, which are subsequently improved using all operators in the next master iterations.

Table 5.14: Set parameter values for Experiment 3

Parameter	Value
# local iterations	50
Destroy factor	0.2
Repair stepsize	10
C_0(Initial SA temperature)	$\frac{1}{10}$ of currently best objective value from last master iteration
SA Cooling factor	0.9
Repair operators and ρ^+	QG=100, G=1, G-WT=1, G-WD=1, G-WB=1
Destroy operators and ρ^-	RD=1, QW=100, W=1, W-WT=1, W-WD=1, W-WB=1
λ	0.9
ω_1 (Best solution)	8
ω_2 (Improved solution)	4
ω_3 (Accepted solution)	2
ω_4 (Rejected solution)	1
K_{best}	8
K_{other}	8

A description of the data used in this experiment is given in Table 5.15, Table 5.16 and Table 5.17. Table 5.15 shows the fraction of technicians with certain skillsets, the fraction of tasks with certain skill requirements, and the fraction of tasks with a certain priority level. We will conduct this experiment on 4 differently sized datasets, or instances. The sizes of these instances are given in Table 5.16. Finally, tasks with a priority of 1 or 2 have a randomly generated request date on one of the available days. This is used to determine the task time windows, which are shown in Table 5.17.

Table 5.15: Distribution of technicians and tasks of Experiment 3

Object	Fraction
Technicians with skillset $[a]$	$1/2$
Technicians with skillset $[a, b]$	$1/4$
Technicians with skillset $[a, c]$	$1/4$
Tasks that require skill a	$3/4$
Tasks that require skill b	$1/8$
Tasks that require skill c	$1/8$
Tasks with priority 0	$5/8$
Tasks with priority 1	$1/4$
Tasks with priority 2	$1/8$

Table 5.16: Data description of Experiment 3

Object	Instance 1	Instance 2	Instance 3	Instance 4
# days	2	3	4	6
# clients	30	45	60	90
# technicians	12	12	12	12
# Tasks	72	104	144	216

Table 5.17: Time windows of tasks in Experiment 3

Object	Value
Time windows priority 0 tasks	(0,9) for all days
Time windows priority 1 tasks	(0,9) for 5 days since request day
Time windows priority 2 tasks	(0,9) for 2 days since request day

In this experiment, we will use all three different stopping criteria. This means that the algorithm is stopped after one of the stopping criteria is met. The first stopping criterion is to stop when no improved solution can be found for 10 master iterations. The second stopping criterion is to stop when the algorithm improves less then 1% in 15 iterations. The last stopping criterion is to stop after a total runtime of 3 hours. Each data instance is randomly generated for 4 times. When either the first or the second stopping criterion is used, we consider the problem instance adequately solved.

5.3.1 Experiment 3: Results

Table 5.18 shows the average runtime in minutes and the average number of master iterations and the used stopping criteria. It can be seen that the smallest data instance is solved in all 4 cases in 50 minutes on average. The second problem instance is stopped 3 times by the percentage stopping criterion and 1

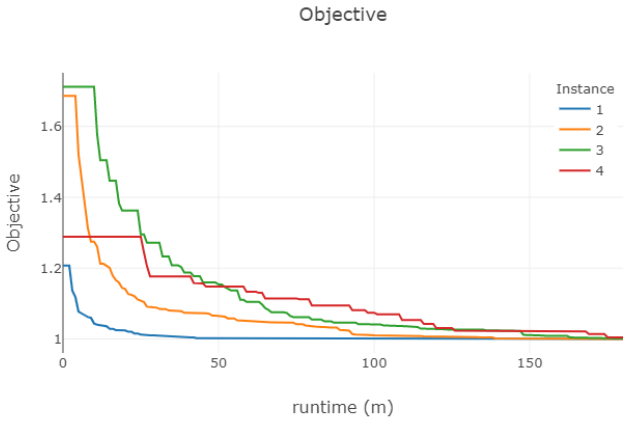
time by reaching the time limit of 3 hours. Then, we see that both instance 3 and instance 4 are not solved adequately and therefore terminated after 3 hours. Furthermore, we see that the stopping criterion of terminating after 10 iterations of no improvement is never reached.

Table 5.18: Average runtime, average number of iterations and stopping criteria

Instance	Runtime (m)	Iterations	No improvement	Percentage	Time limit
1	50.4	29.0	0	4	0
2	138.4	38.3	0	3	1
3	180.0	23.0	0	0	4
4	180.0	9.0	0	0	4

To give a more detailed insight in the performance of the algorithm on differently sized instances, we use Figure 5.4. It is expected that the algorithm finds large improvements initially, but slowly converges when near the optimal solution. The curve of this behavior can be seen at the blue line of instance 1. Note that this algorithm is stopped after around 50 minutes due to an improvement of less than 1% in 15 iterations. When we look at the curve of instance 2, we see that it decreases at a slower rate compared to instance 1, but it also improves relatively more compared to instance 1. This can be explained by the increased size of the instance, as it is easier to find good initial solutions on small problem sizes than on larger problem sizes. Next, when we look at instance 3, we see that the initial solution is around 70% higher than the best found solution, which is about the same as problem instance 2. However, as the instance size is larger than instance 2, we also expect the initial solution to be relatively worse compared to the initial solution of instance 2. Additionally, the algorithm does not yet seem to converge to a horizontal line, which suggests that the algorithm may further improve the best found solution after 180 minutes. Therefore, we conclude that problem sizes of instance 3 cannot be adequately solved with this algorithm and implementation within 180 minutes. Finally, when we look at instance 4, we do not see a curve similar to the curves of instance 1 and 2. Besides that, the algorithm executed less than 10 master iterations on average, which leads us to think that a lot more improvements can be found by increasing the runtime. Thus, the current model is not able to solve instance 4 adequately either.

Figure 5.4: Indexed mean objective values for different problem instances



Partial objectives Besides looking at the overall performance based on the objective value, we now have a look at the development of the different partial objective values. As this experiment does not incorporate rescheduling of fixed tasks and adding overtime, we only focus on the partial objective values of total Work time, total Weighted Delay, and work balance. Table 5.19 shows how each partial objective on average contributes to the total best found objective value. It shows that work balance only has a small impact on the total objective value, which is in line with importance assigned to it. Besides that, it shows that around 80% of the total objective value consists of the partial objective Work time in the best found solutions, except for instance 4. This is another indication that the best found solutions for instance 4 are far from optimal solutions.

Table 5.19: Relative partial objectives for best found solutions

Instance	Work time	Weighted delay	Work balance
1	86%	12%	2%
2	79%	19%	1%
3	76%	21%	3%
4	62%	37%	1%

In order to examine the progress of the different partial objectives, we analyze the development of the average (partial) objective values over time. Figure 5.5 shows the average partial objective values per instance size as a percentage of the objective value of the best found solution.

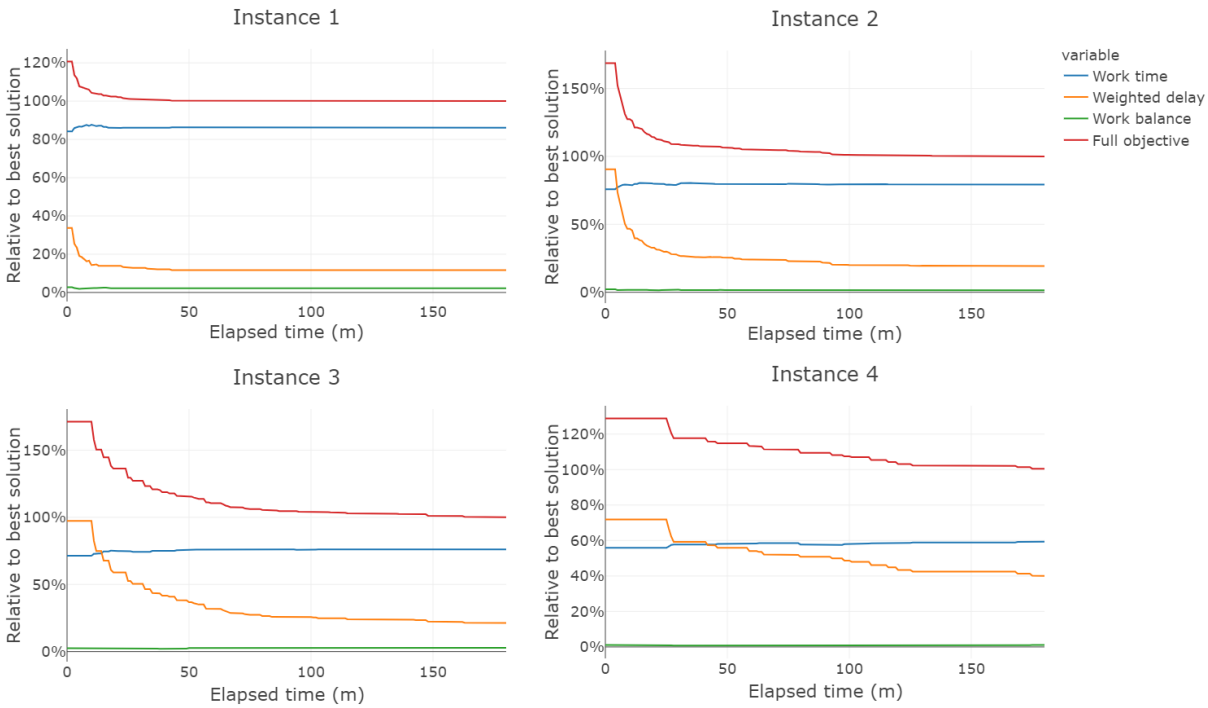


Figure 5.5: Partial objectives over time relative to best found objectives per instance

For example, it can be seen that Weighted Delay in Instance 1 initially has a value equal to 37% of

the objective value of the best found solution. Furthermore, it can be seen that the mean objective value of the initial solutions of Instance 1 is around 20% worse than the best found objective value.

The figure shows that in all instances the partial objective Work time is best in the initial solution and thereafter increases. This can be explained, because we initialized all instances by using the Quick Greedy Insertion operator, and subsequently assigned a large initial weight to this operator. This operator is only aimed at minimizing travel time, which is equal to minimizing Work time in this experiment. This is due to the fact that all task time windows are set equal to the availability time windows of technicians, which causes waiting time never to be added, as explained in Section 4.4.1. Next, we see that in all instances the partial objective Weighted Delay is large initially. As Weighted Delay is a large part of the total objective value with a lot of improvement potential, we see that the progress of the total objective is aligned with the progress of Weighted Delay, at the expense of a small increase in Work time. Weighted delay can be such a large value, because it counts the time between the request moment and the scheduled moment. This value can be much larger than the sum of the added travel time, the service time, and the waiting time of a task (which is zero in this experiment). Finally, while difficult to see in Figure 5.5, the progress of Work Balance behaves quite different compared to Work time and Weighted Delay. It increases and decreases frequently and rapidly, which suggests that the impact of Work Balance is too small. However, we have seen that Work Balance is often increased right after finding a big increase with respect to other partial objectives, which improves the full objective value slightly. Besides that, Work Balance can help to break a tie between two equally good solutions with respect to Work Time and Weighted Delay, but a different impact on work balance.

5.4 Fixed tasks and interventions

In the following and last experiment, we test the ability of handling difficult instances, where a large part of the tasks is fixed. This represents a real life example of adding high priority tasks to a schedule where most tasks are already communicated with clients. We will simulate one working week of 5 days. We provide instances that are not, or do not appear to be, solvable without interventions, such as adding overtime or rescheduling of fixed tasks. The general outline of this experiment is as follows:

1. Create a 5-day schedule with all known non-priority tasks, then fix these tasks to their respective days
   (a) Create an initial solution using Quick Regret-3 operator
   (b) Perform 1 master iteration of the ALNS algorithm
2. Generate new tasks with priority 1 or 2, which need to be scheduled within 2 or 5 days, respectively
3. Add tasks to the current schedule
   (a) Perform 1 master iteration of the ALNS algorithm
   (b) If no feasible solution is found, perform 1 or more interventions

4. Go to the next day, remove all tasks from previous day, and repeat steps (2) to (4) until 5 days have passed or no feasible solution can be found

For this experiment, we will use the same parameters as in Experiment 3, which are given in Table 5.14. Furthermore, the same distribution of technicians and tasks of Table 5.15 will be used.

5.4.1 Data

We want to explore the usefulness of the different interventions. Therefore, the problem instances in this experiment need to have such an amount of tasks, that it becomes often necessary to use interventions in order to create feasible schedules. The problem size and amount of new tasks are based on testing. Using a smaller amount of tasks resulted in little need for interventions, while using a larger amount of tasks resulted in no feasible schedules, as the interventions were not able schedule all unscheduled tasks.

In this experiment, data will be generated in separate phases. First, a problem instance will be generated with technicians, clients, and non-priority tasks for 5 days, with details stated in Table 5.20.

Table 5.20: Initial data description of Experiment 3

Object	Value
# days	5
# clients	90
# technicians	12
# Tasks	168

Note that some generated instances have more dense schedules than others. A reason for this, is that task durations are randomly generated, which means that some instances have longer task durations than other instances. This, in turn, makes schedules more dense, which makes it more difficult to add tasks on a daily basis. As the focus in this experiment is on adding new tasks and using interventions, not all generated instances will be accepted. An instance is accepted and used only when an initial solution can be created using the Quick Regret-3 operator without unscheduled tasks, because this implies that the schedules are not too dense to add more tasks to an existing fixed schedule.

Next, the number of priority tasks that are added on each day are given in Table 5.21. Note that the number of tasks is not the same for each day. The reason for this is, that priority 1 and priority 2 tasks need to be scheduled within either 5 or 2 days, respectively. This means that a priority 1 task that is generated on the fifth day, can normally also be scheduled on days 6 to 9. As we only focus on a time horizon of 5 days in this experiment, we mimic this behavior by decreasing the number of tasks that are added when coming closer to the time horizon.

Table 5.21: Daily data description of Experiment 3

	Day 1	Day 2	Day 3	Day 4	Day 5
# priority 1 tasks	15	12	9	6	3
# priority 2 tasks	14	14	14	14	7

When at a certain day no feasible solution can be found, the following interventions will be used in order, until a feasible solution is found:

1. Relax time windows of one task for rescheduling
2. Add two hours overtime to one tour
3. Relax time windows of tow tasks for rescheduling
4. Add two hours overtime to two tours
5. Relax time windows of three tasks for rescheduling

Adding overtime to a tour is only possible if the corresponding technician does not have another tour with overtime yet. Furthermore, we restrict intervention 3 and 5, which use rescheduling of multiple tasks, to a maximum of 100 random combinations. This is done in order to reduce the total runtime. Moreover, it should be noted that relaxing time windows of tasks at the last day of the time horizon, does not have any effect, as tasks are only fixed to a single day. Therefore, interventions 1, 3 and 5 are skipped when infeasible tasks occur on the last day. This experiment will be conducted on 250 different instances. Note that the runtime of this experiment is much lower than the total runtime of the large instance (instance 4) of Experiment 3, which has the same amount of days and roughly the same amount of tasks in total. This is because all tasks are fixed to a single day after initialization, which reduces the amount of evaluations for all insertions by a great amount. Furthermore, after a day has passed in this simulation experiment, all tasks from the previous day are removed from the problem, which further reduces the required runtime. Lastly, the amount of master iterations is smaller in this experiment, as the focus of this experiment is evaluating the performance of interventions, not finding the best solution.

5.4.2 Experiment 4: Results

After running this experiment 250 times, we see that in total only 35 instances are completely feasible on every day. Let us define failure as the moment that all interventions are unable to schedule all tasks. This means that in the 215 other cases, the interventions failed to schedule all tasks at some day to create a feasible schedule. When we look at the day at which no feasible schedule can be found, we see that most instances fail at the 4^{th}

Day	1	2	3	4	5
Failure count	33	11	11	101	59

Table 5.22: Number of instances where no feasible solution can be found per day

or 5^{th} day, as can be seen in Table 5.22. This can partially be explained due to the fact that rescheduling does not help on the 5^{th} day, because it is the end of the time horizon. Besides that, rescheduling of tasks of the 4^{th} day means that tasks can only be moved to the next day in order to create space for other tasks. Due to the fact that new tasks need to be scheduled within either 2 or 5 days, this means that rescheduling at the 4^{th} day does not help for new tasks, and helps only to some degree for priority tasks that were scheduled earlier. Furthermore, when rescheduling is used on earlier days, this increases the amount of scheduled tasks at the next days.

Intervention successes As we want to look at the usefulness of the various interventions, we now have a look at the amount of times each intervention is successfully used. Let us define a success as an occurrence where an intervention is used and successfully adds all unscheduled tasks to the schedule. For example:

> A schedule on the second day has two unscheduled tasks after adding new tasks and performing one master iteration. Because the schedule is infeasible, the five interventions will be used in order. Intervention 1 is not able to schedule all unscheduled tasks, but intervention 2 is able to create a feasible schedule where all tasks are scheduled. Then, this counts as one success for intervention 2.

Table 5.23 shows the number of successes of each intervention, together with the average count and duration of unscheduled tasks. Furthermore, it shows how often all 5 interventions failed to find a feasible solution.

Intervention		Unscheduled tasks		Successes/Failures
#	Name	Mean count	Mean duration (m)	Count
1	One task rescheduled	1.0	126	28
2	One tour with overtime	1.1	157	61
3	Two tasks rescheduled	1.0	65	3
4	Two tours with overtime	1.9	256	24
5	Three tasks rescheduled	1.0	150	1
-	Failed	4.2	593	215

Table 5.23: Average count and duration of unscheduled tasks and total count of used interventions

First of all, the table shows that all interventions are successfully used at least one time. Furthermore, it shows that rescheduling two or three tasks is only rarely used. A possible explanation can be that the order of interventions does not match the order of the performance of different interventions. For example, rescheduling two or three tasks performs worse than adding overtime to one or two tours, respectively. Another thing to notice in Table 5.23, is the single time that intervention 5 is used. In this situation, there is only one unscheduled task with a duration of 150 minutes. In a tight schedule, adding 120 minutes of overtime to one or two tours can still result in an infeasible schedule. In this specific situation, it is apparent that this intervention of rescheduling three tasks can be useful. However, as this intervention

is not successfully used in all other instances, this intervention is probably worse than adding overtime to two tours.

In order to get more insight in the occasions that these interventions have a success, we group the results per intervention and per amount of unscheduled tasks. The result of this is shown in Table 5.24.

Intervention		Unscheduled tasks		Successes
#	Name	Count	Mean duration (m)	Count
1	1 task rescheduled	1	126	28
2	1 tour with overtime	1	149	57
		2	273	4
3	2 tasks rescheduled	1	65	3
4	2 tours with overtime	1	159	8
		2	256	11
		3	348	4
		4	660	1
5	3 tasks rescheduled	1	150	1

Table 5.24: Average duration of unscheduled tasks and total count of used interventions per amount of unscheduled tasks

In Table 5.24, it can be seen that the different interventions are all useful in order to make infeasible schedules feasible again, as all interventions have at least one success. It should be noted that the five interventions are only capable of making infeasible schedules feasible, when the amount of unscheduled tasks is four at most. However, in most cases it only succeeded with either one or two unscheduled tasks.

The table shows that schedules with more than two unscheduled tasks are only successfully solved on five occasions. In most cases, it would therefore be necessary to increase the amount of rescheduled tasks or tours with overtime, when the amount of unscheduled tasks exceeds two. Additionally, combinations of both rescheduling and adding overtime could be useful for more difficult infeasible schedules, due to the fact that rescheduling of tasks adds a lot of flexibility.

In general, it can be said that the model is very useful from a practical point of view, because the same model can be used for both creating initial week schedules, and rescheduling on a daily basis, when new tasks are added. More research should be done on creating interventions that are able to handle more than 2 unscheduled tasks, as the current interventions are not able to create feasible schedules in those cases.

Chapter 6

Discussion

Key findings The main goal of this research is to create an algorithm that is able to efficiently solve TRSP instances, with the addition of rescheduling on a daily basis in case of changes. The objective is split in four parts, namely the amount of rescheduled fixed tasks or tours with overtime added, the amount of Work Time, the value of Weighted Delay and the value of Work Balance. The research demonstrates that the proposed algorithm is able to optimize problems with respect to all partial objectives, although it should be noted that interventions, which consist of rescheduling fixed tasks and adding overtime to tours, are only used in case no feasible schedule can be found without interventions. Therefore, as interventions are not used in the first three experiments, the first partial objective function is not optimized in this regard. The algorithm is able to quickly find good solutions based on Work Time and subsequently improves Weighted Delay, such that the total objective value improves. The value of Work Balance, in combination with the relatively small assigned weight, seemed to have little impact on the solutions. However, it showed that this value is often slightly improved after making large improvements with respect to working time or weighted delay.

Implications This research showed that the same model can be used for daily (or live) rescheduling, as well as creating initial week schedules. This can be very useful for companies that receive a lot of new requests or change requests at the last moment, but also work with some sort of fixed schedule. Furthermore, this research shows that rescheduling one, two or three fixed tasks, or adding overtime to one or two tours, often helps in making infeasible schedules feasible. Rescheduling of fixed tasks helps with creating feasible schedules and is an intervention that is often used in practice. This is especially useful for companies that need to deal with adding multiple tasks to a current schedule on a daily or weekly basis. However, the amount of rescheduled tasks or the amount of tours with overtime should be increased in order to cope with more changes, especially when many tasks need to be added last minute to an existing schedule.

Limitations and further research This research showed that the current implementation of this algorithm is not able to adequately solve instances with over 144 tasks, as problem instances of this size or larger were stopped after reaching a time limit, instead of stopping after no improvements are found in

ten iterations, or only small improvements of less than 1% in 15 master iterations. The problem instances also showed large improvements at the last few iterations, which is an indication that more improvements can be found.

Furthermore, the experiment that incorporated rescheduling on daily basis, Experiment 4, showed that the proposed interventions are mostly successful when either one or two tasks are infeasible. When the amount of tasks that cannot be scheduled is larger, these interventions are not able to make feasible schedules. As this research only looked at either rescheduling or adding overtime, it might be interesting to investigate the performance of an intervention where both tasks are rescheduled and overtime is added to tours. This could especially be useful for problem instances that are more difficult to solve. Besides this, more research could be done on how to efficiently incorporate the ALNS framework into the interventions, or vice versa. The reason for this, is that the ALNS framework has proven to be good in finding feasible solutions and finding high quality solutions, which could be helpful to use within the interventions. Additionally, the amount of rescheduled tasks or tours with overtime can be increased, possibly with new operators to efficiently search the larger neighborhoods corresponding to these interventions.

Moreover, the performance of the ALNS algorithm can possibly be improved, by implementing more specific destroy and repair operators, such as different variants of the Shaw removal operator, as suggested by Majidi et al. (2018). Besides this, some research could be done on the added value relative to the execution time of the different destroy and repair operators. Furthermore, this research only used a fixed value as destroy factor. It may be beneficial to use a variable destroy factor, which decreases over time, but increases when no improvements can be found, or by using a random destroy factor on a given interval (Avci & Avci, 2019). Moreover, future research could add local search methods, in order to improve current solutions. Finally, some parameters in this research are simply chosen and not optimized, as tuning all parameters is not tractable within the scope of this research, so more research on parameter tuning could be done in order to improve performance.

References

Avci, M. G., & Avci, M. (2019). An adaptive large neighborhood search approach for multiple traveling repairman problem with profits. *Computers & Operations Research*, *111*, 367–385.

Li, Y., Chen, H., & Prins, C. (2016). Adaptive large neighborhood search for the pickup and delivery problem with time windows, profits, and reserved requests. *European Journal of Operational Research*, *252*(1), 27–38.

Majidi, S., Hosseini-Motlagh, S.-M., & Ignatius, J. (2018). Adaptive large neighborhood search heuristic for pollution-routing problem with simultaneous pickup and delivery. *Soft Computing*, *22*(9), 2851–2865.

Paraskevopoulos, D. C., Laporte, G., Repoussis, P. P., & Tarantilis, C. D. (2017). Resource constrained routing and scheduling: Review and research prospects. *European Journal of Operational Research*, *263*(3), 737–754.

Pillac, V., Gueret, C., & Medaglia, A. L. (2013). A parallel matheuristic for the technician routing and scheduling problem. *Optimization Letters*, *7*(7), 1525–1535.

Ropke, S., & Pisinger, D. (2006). An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation science*, *40*(4), 455–472.

Schwarze, S., & Voß, S. (2013). Improved load balancing and resource utilization for the skill vehicle routing problem. *Optimization Letters*, *7*(8), 1805–1823.

Tricoire, F., Bostel, N., Dejax, P., & Guez, P. (2013). Exact and hybrid methods for the multiperiod field service routing problem. *Central European Journal of Operations Research*, *21*(2), 359–377.