Solving a real-world rail maintenance scheduling problem

Menno Oudshoorn

Solving a real-world rail maintenance scheduling problem

by

Menno Oudshoorn

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Friday July 5, 2019 at 04:00 PM.

Student number:	4395751	
Project duration:	November 12, 2018 – July 5, 2019	
Thesis committee:	Dr. ir. Neil Yorke-Smith,	TU Delft, supervisor
	Prof. dr. ir. Peter Bosman,	TU Delft
	Dr. ir. Maurício Aniche,	TU Delft
	Timo Koppenberg, M.Sc.	Macomi B.V.

TU Delft

Abstract

The rail network in the Netherlands is one of the busiest in Europe. To ensure a safe and reliable infrastructure, preventive maintenance is of utmost importance. ProRail, the sole maintainer of the railway infrastructure in the Netherlands, spends hundreds of millions of dollars on maintenance each year. Due to the complexity and busyness of the network, maintenance can cause major disruptions leading to longer travel times. Despite these factors, maintenance is currently scheduled mostly manually, leading to suboptimal schedules being created. A pilot study by Macomi, the company which enabled this study through an internship, showed great potential for improvement in scheduling this maintenance. In this thesis, the aim is to further improve the maintenance schedule of ProRail, as well as to find out what kind of solution method is most suitable to improve that schedule. To achieve this goal, various types of algorithms have been implemented and tested on the problem. An evolution strategy algorithm developed by Macomi was successfully improved. Through the use of multi-objective algorithms, the trade-off between maintenance costs and availability of the infrastructure was analyzed. These multi-objective algorithms were found to provide unsatisfactory solutions. Greedy algorithms were also developed to provide a quicker solution method, and the resulting solutions were of surprisingly high quality. Finally, a hybrid algorithm using the evolution strategy and a greedy algorithm was created. It was shown that this hybrid algorithm outperforms all other algorithms, and provides solutions that are better in terms of costs and constraints compared to the manual schedule of ProRail, and the baseline established by the pilot study of Macomi.

Preface

Before you lies the final product of my thesis project for the M.Sc. Computer Science at the faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS), at the Delft University of Technology. This thesis came to be through an internship at Macomi, a company focused on optimization and simulation.

I would like to express my gratitude to a number of people, without whom this project could not have been completed successfully. First of all to my supervisor at the TU Delft, Dr. ir. Neil Yorke-Smith, and my daily supervisor at Macomi, Timo Koppenberg, M.Sc. Both have provided me with valuable ideas and constructive feedback towards the successful completion of my thesis. I would also like to thank everyone at Macomi, for taking me into the team and giving me a great internship experience. I'm looking very much forward to be returning in a few weeks, no longer as an intern, but as your colleague. Finally, I would like to thank my parents, for everything you've done for the past twenty-two years. You both deserve my eternal gratitude.

Menno Oudshoorn
Rotterdam, July 2019

Contents

1 Introduction

The Netherlands contains more than 7000 kilometres of railway track [1]. This track needs to be kept in good condition to minimize unexpected failures and provide a solid infrastructure for travellers and transport companies. To achieve this goal, both preventive maintenance and the construction of new infrastructure is of utmost importance.

All railway tracks in the Netherlands are maintained by ProRail [5]. ProRail analyzes the necessary maintenance and incorporates this into maintenance plans of different granularity; both multi-year plans and single year plans are created. This thesis focuses on the single year planning.

To ensure track workers' safety, ProRail prohibits any trains from running on a track where maintenance is taking place [67]. This means that maintenance will cause disruptions in the train schedule and impediment for society. Furthermore, track maintenance accounts for a large yearly cost. Therefore, creating a solid schedule for track maintenance, which minimizes costs and maximizes availability, is vital for both ProRail and its customers.

1.1. Problem statement and motivation

The problem that needs to be solved is to create a year-long schedule of all maintenance activities which are deemed necessary to be performed. The maintenance *planning*, i.e. determining which maintenance is necessary for the given year, falls outside the scope of this project; the input to the problem is a list of maintenance blocks that need to be scheduled. Each block indicates which parts of the rail network need to be taken out of service, and for how long. The problem is of a substantial size, with more than 600 maintenance blocks having to be scheduled in a year, and over 8000 possible moments to plan each block.

The objective of the problem is to minimize the costs of the schedule. The costs are two-fold. On the one hand, there are actual maintenance costs, which need to be paid by ProRail to contractors. Some of these costs are independent of the planned maintenance time, but others are not. For example, the personnel costs depend on the planned moment, due to the fact that extra fees need to be paid when working at night or during a holiday. On the other hand, the availability of the infrastructure and the degree with which passengers and goods are hindered through a certain maintenance schedule are also expressed as a cost. When performing maintenance, both passengers and goods trains will be hindered and may need to take a detour route or a replacement bus, therefore undergoing extra travel time or other inconveniences. The amount of hinder is highly dependent on the planned time of the maintenance. Scheduling multiple blocks of maintenance that affect the same part of the rail network at the same time may decrease unavailability, as the track only needs to be taken out of service once. Scheduling maintenance at night will mean much fewer passengers are hindered compared to scheduling it during rush hour.

The two types of cost are also conflicting. For example, scheduling maintenance at night will yield lower availability costs, but higher personnel costs, and vice versa. This implicit trade-off is something that needs to be taken into account when considering solution methods.

The problem is also subject to a number of constraints. Some constraints are relatively simple; for example, a maintenance block can have a certain time window in which it must be performed. Other constraints are more complex, and also capture dependencies between different maintenance blocks. For example, certain parts of the railway track may not be taken out of service at the same time. The constraints lead to a reduced (feasible) search space which severely limits the possibilities to find a good solution in terms of costs. The combination of the problem size, the presence of multiple conflicting objectives, and the complex constraints make this a hard, but interesting problem to solve.

There is both societal and scientific motivation for this project. The societal motivation is directly connected to improving the maintenance schedule in terms of costs. Maintenance costs are largely paid for using government money, so a cost reduction leads to possibilities for increased spending in other areas or tax reductions. Furthermore, increasing availability of the rail network means fewer passengers are affected by maintenance, and the overall efficiency and satisfaction will be higher. Another aspect of the societal motivation is the possibility for a large reduction in the time it takes to make a yearly maintenance schedule. When a schedule can be made in the timespan of a day instead of multiple months, this allows for the option to quickly evaluate changes to the schedule and to run with different scenarios. This can help ProRail to support arguments when assigning priorities with all the other stakeholders, such as the ministry, traveler organizations, and train companies. The scientific motivation lies primarily in finding out how algorithms that work well in theory and on small, sometimes artificial test problems, perform on a large size, real-world problem with many complex constraints. There have not been many studies on a problem of this scale, and getting an idea of the effectiveness of certain solution methods is an interesting research area.

1.2. Introduction to ProRail and Macomi

In this section, two companies that are closely related to this project are introduced briefly. The first is ProRail, the company responsible for the rail network in the Netherlands. The second is Macomi, a company focused on optimization and simulation, which enabled this study through an internship.

ProRail is the company which is solely responsible for the rail network in the Netherlands. As an independent party, they divide the available space on the track, regulate all train traffic, build and maintain stations, and build new tracks. Furthermore, all existing infrastructure is maintained by ProRail. ProRail aims to ensure a safe, reliable, punctual and durable rail network with comfortable stations, together with transport companies and other partners [12]. ProRail has three main objectives to ensure the Netherlands is connected via rail, now and in the future, despite the challenge of increasing demand for transport of passengers and goods [6]. These objectives are to

- Connect: developing the capacity to ensure the future's mobility.
- Improve: making mobility via railway as reliable as possible, now and in the future.
- Ensure durability: making mobility via railway as durable as possible.

Macomi is a small company which was founded in 2011 and has its roots at the Delft University of Technology. Macomi focuses on optimization and simulation, and offer a specialized analytics software platform that can be used to develop client specific solutions [4]. It offers two products with specific tool sets for rail management and workforce planning. Furthermore, clients can be provided with custom solutions based on their specific challenges, such as planning and optimization. Macomi has worked, and is still working together with ProRail on such a custom module to gain more insight in, and improve the process of planning and scheduling rail maintenance. This thesis relates directly to this collaboration project, and came to be through an internship at Macomi.

1.3. Research goal and scope

The main goal of this research is to improve upon the solutions to the problem found by the scheduling experts from ProRail as well as a pilot study performed by Macomi. This, however, is a broad statement and it is unclear how this goal would be reached. Therefore, some interesting characteristics of the problem are identified which will be studied in more detail:

- The trade-off between the maintenance costs and the availability of the rail network. Although it is known that such a trade-off exist, it is unclear how solutions would range over the two types of costs. An attempt to improve this will be made by using multi-objective solution methods; methods which produce multiple solutions that span a large range of the front.
- The trade-off between costs and constraint violations. The constraints in the problem limit the search space significantly, and an interesting research topic is to investigate how the costs of the obtained solutions change.

Furthermore, due to the complexity of the problem, it is not directly clear which solution methods will work well on this problem, due to the fact that there are no problems which are an exact match in literature. Therefore, another research goal is to perform a comparison of different types of algorithms. This combines well with the point previously made about the trade-off between the two cost types; the difference in performance between single and multi-objective algorithms can be researched.

1.4. Research questions

The main research objective is to study different solution methods and compare their capability to improve the yearly maintenance schedule of ProRail in terms of costs while respecting the problem constraints. Furthermore, there is an objective to make the trade-off between maintenance and availability costs more explicit. Therefore, the main research question will be

Which solution method is the most suitable to improve the yearly maintenance schedule of ProRail?

To help answer the main research question, several sub-questions are defined:

- How can the problem be defined in terms of a minimization problem with multiple objectives and constraints?
- What scheduling approaches are available and being used in problems with similar characteristics?
- What is the current situation in terms of solution quality as well as runtime for both the manual ProRail planning as well as the algorithm created by Macomi?
- What are the shortcomings of the current algorithm and how may it be improved?
- Which multi-objective algorithms exist, how can they be used to make the trade-off between maintenance costs and availability more explicit, and what is their performance on the problem?
- Can other single-objective algorithms, such as different metaheuristics or a greedy algorithm, outperform Macomi's algorithm?
- Can algorithms be combined to improve the performance?

1.5. Anonymity of the data

During this thesis project, experiments with real ProRail data have been performed. This means the obtained results and conclusions will be valid and useful in a real-life scenario. However, the real maintenance costs of ProRail may not be made public. Therefore, all costs in this thesis have been anonymized by dividing the real costs through a large constant which is not given. This means that the relative differences will remain the same. It does mean that the absolute differences are small; it should be kept in mind that a small relative cost reduction can mean a large absolute cost reduction in real life. The hard constraint violations and the problem itself are not anonymized.

1.6. Structure of this document

The remainder of this thesis is structured as follows: Chapter 2 introduces the necessary background knowledge and provides a detailed description of the problem in both textual and mathematical forms. Chapter 3 provides an overview of existing solution methods in literature, as well as the existing infrastructure and solution method developed by Macomi. In chapter 4 some shortcomings of the existing solution method are identified, and a number of improvements are discussed and tested. Afterward, two multi-objective algorithms are described, implemented and tested in chapter 5. Chapter 6 contains a parameter analysis for the improved evolution strategy algorithm to find optimal parameter settings. In chapter 7, two greedy algorithms are described. Furthermore, a hybrid between a greedy algorithm and the evolution strategy is proposed. In chapter 8, further improvements to the evolution strategy based on the results in previous chapters are described. In chapter 9, the results obtained throughout the different chapters are summarized and discussed. Finally, in chapter 10, the thesis is concluded by answering the research questions and providing an outlook for future research.

2 Background and problem statement

In chapter 1, a brief introduction to the problem and research goals was given. In this chapter, the necessary background knowledge is given and the problem is explained in more detail. First, an introduction to the current process of planning and scheduling maintenance at ProRail is given. Then the specific problem tackled in this thesis is extensively explained, both in textual form and as a mathematical model.

2.1. Maintenance planning and scheduling at ProRail

The Dutch railway network contains more than 7000 kilometres of railway track and is one of the busiest railway networks in Europe [7]. In 2018, a total of 164 million kilometres were driven by passenger trains, and a total of 55 billion tonne-kilometres were driven by goods trains [9]. The number of trains and passengers using the network is growing constantly; the number of passenger train kilometres has increased from 156 million in 2014 to 164 million in 2018, and the tonne-kilometres of goods trains have increased from 51 billion to 55 billion in the same timespan [10]. The demand for transport of passengers and goods is expected to keep increasing until 2040 [12]. Figure 2.1 shows the network usage intensity of the rail network in various European countries, and it can be seen that the Netherlands has the busiest rail network of them all. Keeping the infrastructure in optimal condition is therefore of utmost importance to ensure a safe and durable network, and to minimize the number of unexpected failures, causing major disruptions to the train schedule and high costs for corrective maintenance.

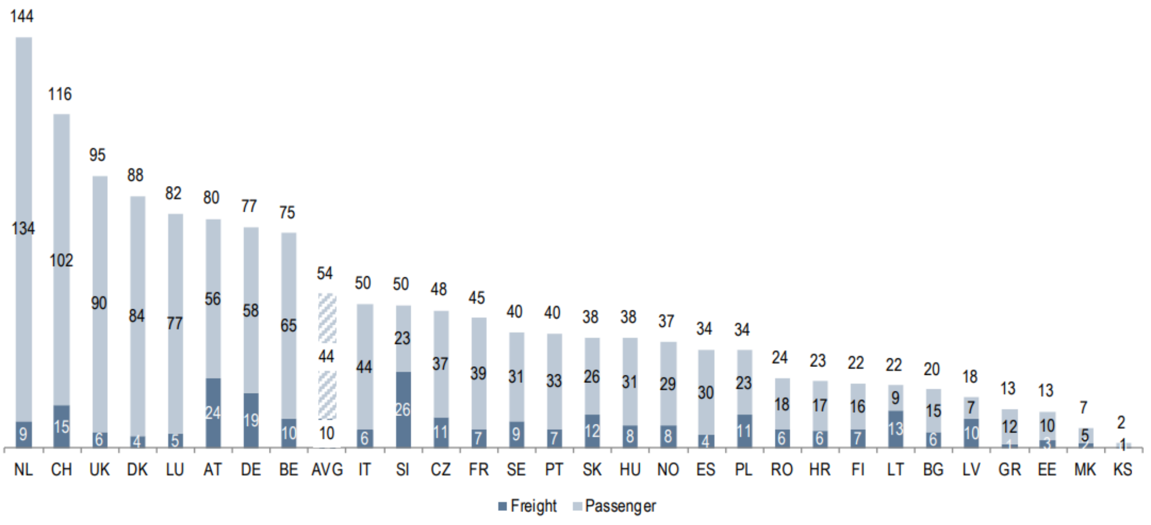


Figure 2.1: Network usage intensity (train-km per route per day) in 2017 in various European countries. Source: IRG-Rail [3]

Maintenance and building new infrastructure accounts for a large part of the yearly costs of ProRail, around 479 million in 2018 [8]. These costs are mostly paid for by the government of the Netherlands, meaning all taxpayers would benefit from lower maintenance costs. Furthermore, performing maintenance causes trains to be hindered which, especially on a rail network as busy as the one in the Netherlands, causes major disruptions and possible capacity problems on detour routes. Passengers will have to travel for a longer time, possibly with more train changes, and may have to take replacement buses which are slower and less comfortable.

ProRail and other stakeholders have recently come up with a vision regarding track maintenance named Future-proof Working On The Tracks (TWAS) [11]. They recognize that the usage of the rail infrastructure will keep increasing and this means that more and more maintenance is needed to keep the infrastructure in a reliable state. Without a new way of working, this increase would mean that passengers and goods trains are increasingly hindered, costs would rise, and possible personnel shortages may arise. The new vision focuses on finding a balance in the triangle of availability, affordability and reliability. Improving the process of planning and scheduling maintenance is one of the key factors in achieving this new vision. This improvement is a large project inside ProRail, and the research performed in this thesis is done in this context.

It is obvious that there is plenty of incentive to create a maintenance schedule in such a way that maintenance costs and unavailability of the track are minimized, while still ensuring reliability by not decreasing the amount of maintenance that is performed. The problem researched in this thesis project focuses on this maintenance schedule. The reliability factor is not relevant in this project, as this is mostly important when deciding what maintenance needs to be done in the year. This is an important issue but lies outside the scope of this thesis. The focus of this project lies on decreasing maintenance costs and unavailability.

In this section, the necessary background information regarding the process of planning and scheduling maintenance at ProRail is provided. First, the necessary terminology regarding the rail infrastructure is explained. Then, the maintenance scheduling process is described.

2.1.1. The Dutch railway network

The Dutch railway network is one of the busiest railway networks in Europe. Figure 2.2 shows an overview of the rail network and indicates its complexity. A disruption in the network due to preventive maintenance can cause many passengers and goods trains to have to take detour routes, not only causing a longer travel time but also a larger number of passengers on other parts of the network, possibly causing overfull trains. Furthermore, if a detour route is not available, passengers have to take replacement buses which are both slower and less comfortable than a train. When scheduling maintenance, this should be taken into account. To make sure all passengers can still get to their destination, there are various rules stating which parts of the network must remain open when other parts are closed down due to maintenance.

Some terminology regarding the rail network is necessary to understand the remainder of this document. The terminology is adopted from ProRail's corridorbook [2].

- **Junction (Knooppunt)**: A junction is an important station or yard for passengers and/or goods trains.
- **Endpoint (Eindpunt)**: An endpoint is either a point where the available infrastructure ends or a border crossing.
- **Section (Baanvak)**: A unique part of the network between two junctions and/or endpoints.
- **Corridor**: A number of subsequent sections with an ongoing transport flow. A section can be part of multiple corridors.
- **Sub-corridor (Deelcorridor)**: One or more subsequent sections lying on a corridor. A sub-corridor can be part of multiple corridors.

2.1.2. Maintenance process

Maintenance planning and scheduling is a large operation consisting of multiple stages. In this section, an overview of this process is given.

The smallest unit of maintenance considered in this thesis is an *asset operation*. An asset operation describes either a piece of maintenance which has to be performed at a certain part of the infrastructure or development of new infrastructure. An example is the renewal of a piece of track and sleepers. The asset operations which need to be performed are determined through inspection and routine work; the exact method is out of the scope of this thesis. The asset operations are set up for multiple years in advance.

Each year, a number of asset operations are chosen to be performed for that year. Which asset operations are chosen is based on the necessity of the maintenance, the available budget, and such that there is not too

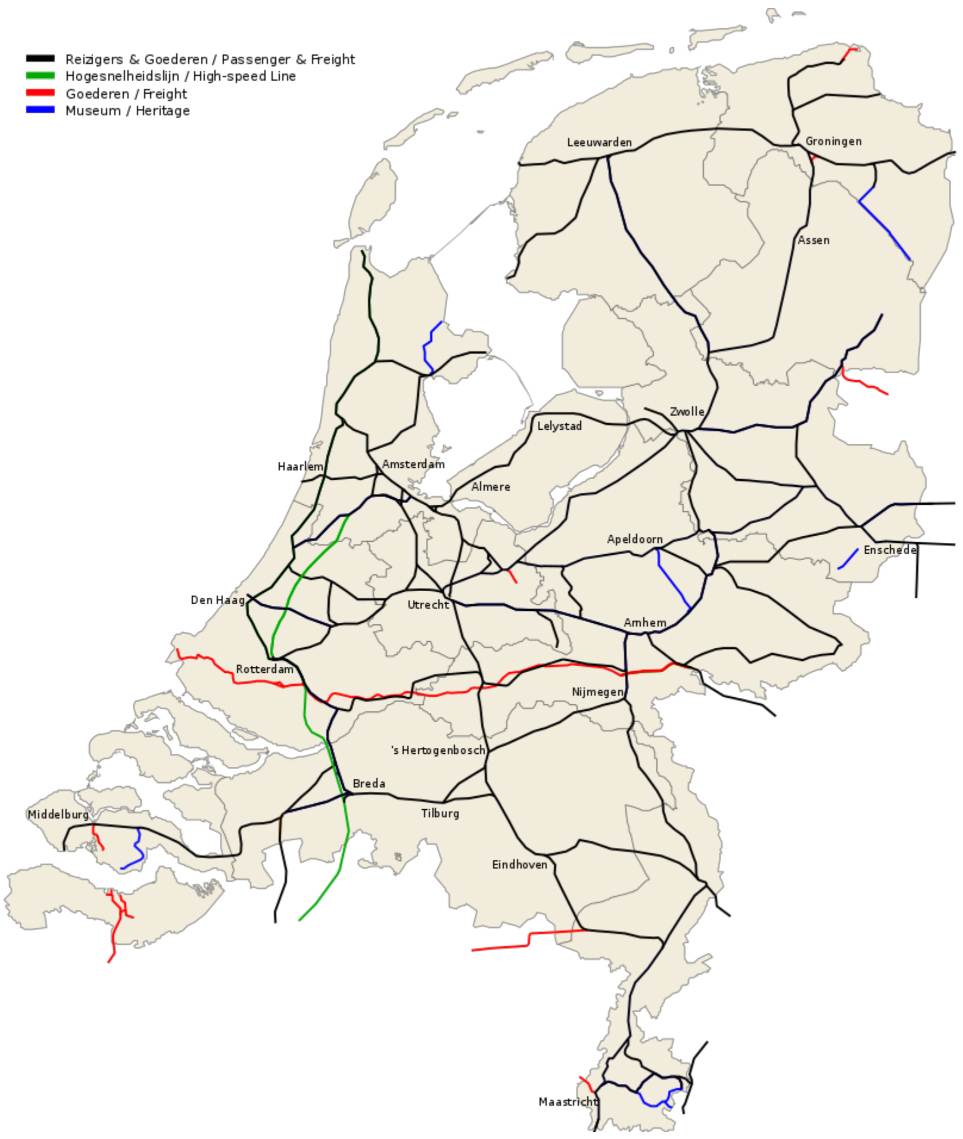


Figure 2.2: Rail network of the Netherlands

much maintenance planned on the same parts of the network in one year. The chosen asset operations are then clustered into *projects*. A project consists of one or more asset operations which are somehow related. The exact way in which asset operations are chosen and clustered into projects is out of the scope of this thesis.

The managers of each project can then do one or more *project requests*. A project request is a request to take a certain part of the rail network out of service, to allow the asset operations associated with the projects to be performed. There are multiple reasons why a project may need more than one period in time in which a part of the infrastructure is taken out of service; the duration of the maintenance may be too long, or the project may affect parts of the railway network which should not be out of service at the same time. The exact way in which project requests are formed out of projects is not in the scope of this thesis. It is also important to note that certain asset operations, and therefore project requests, do not require the track which is worked on to be taken out of service and will therefore not hinder the trains using that piece of the network.

When the project requests necessary for each project are known, they can be scheduled for the given year. This is done manually by scheduling experts from ProRail. The created schedule must adhere to various constraints, which are explained in section 2.2.2.

The fact that the schedule is created manually means it takes a long time to make one. In 2019, approximately one thousand project requests had to be scheduled which can take months, even by experts with many years of experience. Furthermore, the complex constraints in combination with the fact that the necessary data is incomplete and unclean, logic regarding constraints is unmodelled, and other stakeholders are involved, leads to the scheduling experts having a hard time making a schedule which adheres to all the constraints, as well as in seeing which constraints may be broken. The scheduling problem is simply very large and complex and is hard to optimize well by a human. Therefore, a model of the situation and an optimization algorithm can provide valuable insights as well as reductions in costs and unavailability.

2.2. Description of the problem

The specific problem researched in this thesis concerns scheduling a number of project requests in a given time period, usually a year. Given a list of project requests, each of these requests must be assigned a start time, where an hourly granularity is used. This must be done in such a way that the costs are minimized while respecting several constraints. The way the costs are calculated is explained in section 2.2.1 and the constraints are explained in section 2.2.2.

A project request is characterized by the following information:

- A duration (in whole hours).
- A set of sub-corridors it affects and that should be taken out of service for this piece of maintenance.
- A list of maintenance types planned during this project request, possibly empty.
- A set of prerequisite project requests which must be finished before this project request can start.
- An indication of the specialized staff needed during this project request. There are three types of specialized staff: BFI, BVL and THL.
- A time window indicating the period in which this project request must be scheduled.
- Maintenance costs of the project request, subdivided into three categories: personnel costs, security costs, and constant costs.
- An indication of the percentage of passengers and freight trains that will be blocked by this project request on the given subcorridors. This number is possibly zero, meaning the project request does not cause a disruption in the infrastructure.

A sub-corridor is characterized by the following information:

- The set of corridors it is part of.
- The number of passengers estimated to travel through the sub-corridor, for any given hour in the global plan window. The number depends on multiple factors such as the day of the week, the hour, and whether there is a holiday or not.
- The number of extra travel minutes (Extra Reizigers Minuten, ERM) each passenger can expect when the sub-corridor is taken out of service.
- The percentage of travelers taking replacement buses (rather than taking a detour by train) when this sub-corridor is taken out of service.
- The number of freight trains estimated to go through the sub-corridor, for each hour in the global plan window.
- The fine that has to be paid per freight train when the sub-corridor is taken out of service.

2.2.1. Problem objectives

The problem contains two objectives which both need to be minimized: the maintenance costs and the availability costs.

Maintenance costs

The maintenance costs are costs for actually performing the maintenance, which have to be paid by ProRail. There are five types of maintenance costs defined: logistic costs, goods costs, stock costs, personnel costs, and security costs.

Due to the current model being used, the first three types of costs are not optimizable: they always have to be paid fully, regardless of when the project request is planned. These costs do therefore not contribute anything meaningful to the optimization problem, but they are included to give ProRail a complete picture of their predicted costs. These costs make up around 67% of the total maintenance costs, which should be kept in mind when considering the relative improvements being made. For now, the costs for logistics, goods and stock are grouped together and referred to as "constant costs" from this point onwards. The total constant costs are calculated simply by adding up the constant costs of all given project requests.

The next part of the maintenance costs are personnel costs. The personnel costs can, in principle, be calculated for each project request independently. There is one exception, which will be touched upon in a moment. The given personnel costs of a project request are always divided equally over its sub-corridors. For each sub-corridor, the personnel costs per hour are calculated, simply by dividing the total personnel costs by the duration of the project request. Then, for each hour of the period the project request is planned in, the hourly personnel costs are multiplied by a multiplier which is given for the specific sub-corridor, date and hour. This multiplier could, for example, indicate more personnel costs have to be paid due to having to work at night or during a weekend. All these multiplied hours for each sub-corridor are then added together to compute the personnel costs for the project request, and the total personnel costs are computed by adding up all the individual personnel costs of all the requests.

There is one additional aspect which needs to be taken into account when calculating the personnel costs. Because a personnel shift always takes at least eight hours, the duration of a project request under eight hours is extended to eight hours for the calculation of personnel costs. However, the hourly personnel costs are still calculated using the original duration. This means that a project request with a duration of four hours will effectively have double the personnel costs. There is one exception to this rule: at a certain sub-corridor, when two project requests which both take 8 hours or less are planned directly after each other, instead of scaling up the duration of both requests to 8 hours, the combined length is scaled up once. For example, if two requests with length three are planned subsequently at the same sub-corridor, the total duration is six and effectively, the personnel costs will be multiplied by $\frac{8}{6}$ instead of $\frac{8}{3}$.

The final piece of the maintenance costs is the security costs. Again, each project request has a given total security cost and the costs are equally divided over the sub-corridors. The security costs are calculated per *overlapping period* in each sub-corridor. An overlapping period (in a sub-corridor) is a continuous period in time in which at least one project request is planned at that sub-corridor at all times. For each overlapping period, the security costs are defined to be the maximum of the security costs of each project request planned at the sub-corridor during the overlapping period. For example, if at a certain sub-corridor, project request 1 with security costs 4 is planned from time 0 to 4, request 2 with security costs 8 is planned from time 2 to 5, and request 3 with security costs 2 is planned from time 5 to 12, the overlapping period will be from time 0 to 12 and the security costs of this overlapping period will be equal to 8, the maximum of the three requests.

The total maintenance costs are computed by adding up the personnel costs, security costs and constant costs.

Availability costs

The availability costs are the costs associated with the (un)availability of the railway infrastructure and the impediment experienced by passengers and freight trains. Not all the sub-costs in this objective are costs in the sense that someone has to pay them; part of the costs are extra travelling minutes for passengers. However, to provide an easy comparison with for example the maintenance costs, these are expressed in a Euro amount.

The first part of the availability costs are the costs of passenger impediment. These impediment costs are calculated per hour, per sub-corridor. For a specific hour and sub-corridor, the degree of impediment is calculated first. This is done by finding the maximum degree of the impediment of the project requests which affect the given sub-corridor and are planned during that hour. Furthermore, for each date and hour, and for each sub-corridor it is known how many passengers are expected to travel through the sub-corridor and the ERM passengers are expected to experience. Finally, for each month of the year, a static month multiplier is given to indicate busier and less busy months in the year. These can all be multiplied to find the total ERM, which can then be multiplied by the cost per ERM which is given as an input parameter. The costs should be increased by 30% for each passenger expected to travel by replacement bus, as travelling by bus is considered a worse experience than travelling by train and transferring from bus to train takes longer due to (longer) walking times. The computed ERM costs for the sub-corridor and hour are then multiplied by the maximum degree of impediment. The total impediment costs are computed by adding up the impediment costs for each hour and sub-corridor during the global plan window.

A similar calculation is performed for freight train impediment costs. Again, the maximum degree of impediment for freight trains is calculated per hour and sub-corridor. Note that the impediment for freight trains is not necessarily the same as that for passengers. For each sub-corridor, the fine that has to be paid for each hindered freight train is known. Furthermore, the expected number of freight trains is known for each sub-corridor, date and hour in the global plan window. These can be multiplied with the maximum degree of impediment to find the freight train impediment costs for a certain hour and sub-corridor. The total freight train impediment costs are computed by adding up the impediment costs for each hour and sub-corridor during the global plan window.

Finally, some alternative travel costs have to be paid. The alternative travel costs are computed per overlapping period for each sub-corridor and can be computed by a given function of the number of affected passengers to alternative travel costs. The number of affected passengers is computed in the same way as for passenger impediment costs. The total alternative travel costs are computed by adding up the alternative travel costs of all the overlapping periods in all sub-corridors.

The sum of the impediment costs for passengers and freight trains, as well as the alternative travel costs, make up the total availability costs which make up the second objective of this problem.

Trade-off between the objectives

Although the two objectives do not directly contradict each other, there is some negative correlation between the two, meaning there is a trade-off present. A simple example is the trade-off between planning during the day or during the night. When a project request is planned during the day, the personnel costs will be lower because no extra fees will have to be paid for working at night. However, there will be more passengers hindered, as there are more people using the rail network during the day than during the night. Therefore, the maintenance costs will be relatively low, but the availability costs will be relatively high. This works vice-versa for planning the request at night. The way this trade-off is handled by various solution methods will be discussed in due time, but the contradicting nature of the objectives is good to keep in mind.

Until recently, the primary focus of the schedulers at ProRail has been to reduce the availability costs and the impact on the maintenance costs was not really taken into account. This can be seen by the fact that almost all maintenance is usually scheduled during a night, weekend, or holiday. Only recently they have started investigating the effect on costs. This trade-off is relatively unexplored and will be investigated further in this thesis.

2.2.2. Problem constraints

There are a number of constraints which must be satisfied by a solution schedule. Below, all of these constraints are explained. Unless specified otherwise, the constraints are only enforced for project requests which create *hindrance*, meaning its degree of impediment for either passengers or freight trains is larger than zero.

Combination matrix

The Combination matrix constraint specifies which types of maintenance can and cannot be performed simultaneously. There may never be two projects that are planned simultaneously at any sub-corridor while any of there maintenance types are incompatible.

Prerequisite

Certain project requests require other project requests to be completely finished before they can start. Such a relation is called a *prerequisite*. The prerequisite constraint specifies that a project request may only start once all its prerequisites have (completely) finished. This constraints also holds for project requests without hindrance.

Personnel constraints

There are certain types of specialized personnel who are sometimes needed to successfully perform the necessary maintenance in a project request. There is only a limited number of such personnel available in the

Netherlands, but they do work all across the nation. An example is the so-called THL personnel, which are personnel that are able to perform exothermic welding. For each of the three specialized personnel types (BVL, BFI, THL) there is a given maximum that cannot be exceeded at any time. The maximum is global, meaning all project requests planned at a certain time contribute towards the maximum personnel. Project requests that do not cause hindrance also count towards the maximum.

Conflict matrices
There are five different conflict matrices which specify combinations of sub-corridors that may not be taken out of service at the same time. These types are:

- Corridor conflict: Only certain parts of a corridor may be taken out of service simultaneously.
- Passenger detour conflict: Passengers must have an alternative route available so they can still get to their destination.
- Goods detour conflict: Goods trains must have an alternative route available so they can still get to their destination.
- Junction conflict: Certain important junctions must stay reachable.
- Border conflict: Trains must be able to cross the border at at least one location.

This means that there may never be two projects planned at the same time if there is a conflict between any of their sub-corridors. It must be noted that a pair of requests can only create one conflict violation, even if there are multiple conflict matrices that would indicate a conflict.

Dependencies
Dependency constraints specify combinations of time and sub-corridor where planning maintenance is prohibited, for example, because of a national event taking place where more passengers are expected to travel to a certain location, or to prevent performing maintenance simultaneously with Germany at certain border locations. If any project request is planned during the given time and sub-corridor, this will create a constraint violation.

Corridor constraints
The corridor constraints apply to overlapping periods, which are, as explained before, continuous periods in time for a given corridor or sub-corridor in which at least one request is planned at all times. Please note that these constraints only hold for requests that create hindrance, and therefore the overlapping periods are computed using only these requests. Whether the overlapping periods are defined on a sub-corridor or corridor level depends on the specific constraint.

In these constraints, the notion of *TVP* is used. This term is used loosely by ProRail employees for some slightly different concepts, but in this thesis, it will be used to indicate an *overlapping period of at least 24 hours*.

The *max TVP corridor* constraint specifies the maximum number of TVPs that can be present in a corridor, in the complete global plan window. When a TVP takes longer than 168 hours, they should be considered as two TVPs for this constraint.

The *max weekends corridor* and *max weekends sub-corridor* constraints specify the maximum number of weekends that may be affected by TVPs in a corridor and sub-corridor respectively. Whether the TVP only affects a few hours or the complete weekend does not matter.

The *mintime between TVPs corridor* constraint specifies the minimum number of hours that must be between two TVPs within the same corridor.

Required time window
A project request may have a required time window in which the request must be planned. Of course, the request always has to be planned within the global plan window, but the required window may be more restrictive. This constraint is also enforced for requests without hindrance.

Max simultaneous project requests at one location
This constraint specifies the maximum number of project requests which may be planned at the same time in a sub-corridor at any time. Requests without hindrance also count towards this maximum. Going over this maximum will lead to a constraint violation.

Constraint type	Constraint severity	Penalty	Penalty aggregation
Combination matrix	Exclude		
Conflicts - Border	Hard		
Conflicts - Corridor	Hard		
Conflicts - Goods detour	Hard		
Conflicts - Junction	Soft	0.0121	Linear
Conflicts - Passenger Detour	Hard		
MaxTVPCorridor	Soft	0.0242	Linear
MaxWeekendsCorridor	Exclude		
MaxWeekendsSubcorridor	Exclude		
MintimeBetweenTVPsCorridor	Soft	0.0121	Linear
Dependencies - Germany	Hard		
Dependencies - RWS	Hard		
Dependencies - Unwritten	Exclude		
Dependencies - Events 1	Hard		
Dependencies - Events 2	Soft	0.0605	Linear
Dependencies - Events 3	Soft	0.0363	Linear
Dependencies - Events 4	Soft	0.0121	Linear
Dependencies - Events 5	Exclude		
MaxRequestsAtOneLocation	Soft	0.0242	Exponential
Outside required timespan	Hard		
Personnel - BFI	Exclude		
Personnel - BVL	Exclude		
Personnel - THL	Exclude		
Prerequisite	Hard		

Table 2.1: Base constraint scenario

2.2.3. Constraint scenarios

From the initial pilot study regarding this problem by Macomi, it became clear that enforcing all these constraints as hard would lead to a completely infeasible solution space and there would be little room for optimizing costs; a scheduling algorithm would spend all its time on decreasing the hard constraint violations. Therefore, it was decided to loosen the constraints and implement constraint scenarios in which the user can set each constraint to hard, soft, or exclude. A hard constraint must be solved and is infinitely more important than the objective(s). A soft constraint is a constraint for which it is preferred that the solution satisfies it, but breaking it is allowed at the expense of a penalty which the user can set themselves. The penalty is based on the type of constraint that is broken, and the *amount* with which it is broken. The amount of a broken constraint is constraint-specific. For example, for a violation of the maximum number of project requests at one location, the amount would be the difference between the actual number of planned requests and the allowed number of planned requests. The soft constraint penalty can be "one time", meaning a single penalty is given regardless of the amount with which the constraint is broken, "linear" where the penalty increases linearly with the amount of the violation or "exponential" where the penalty increases exponentially with the broken amount. Finally, constraints may also be excluded completely from the problem. Certain constraints may also be split up and used as a different constraint type or with a different penalty. For example, the dependency constraints are split up in a number of categories such as "Germany" to indicate a prohibition to perform maintenance due to maintenance being performed in Germany, and five "Event" categories, one to five, which allow the user to specify a different constraint type or penalty for different types of events.

In consultation with ProRail a "base" scenario was established which indicates their preferences regarding the importance of certain constraints over others. Table 2.1 shows this scenario. Unless otherwise specified, all experiments and results in this thesis are using this constraint scenario.

For the constraints that are set to soft in the base scenario, it is briefly explained how the broken amount can be calculated. In the next section, this will be specified formally for all constraints.

- Conflicts - Junction: Each pair of subcorridors which is taken out of service even though prohibited by the junction conflict matrix is a violation, the violation amount is the total number of times this happened for that subcorridor pair in the complete schedule.

- Max TVP corridor: One corridor can cause a single violation, with violation amount equal to the difference between the actual number of TVPs and the allowed number of TVPs.
- Mintime between TVPs corridor: Each pair of TVPs in a corridor which are too close together is a violation, the violation amount is equal to the minimum number of days that should be between TVPs minus the actual number of days between the TVPs.
- Dependencies - Events 2, 3, 4: Only a single violation can exist, the violation amount is the total number of occurrences of a project request overlapping with a dependency at a subcorridor.
- Max requests at one location: Each subcorridor can cause a single violation, the violation amount is equal to the difference between the maximum occurred number of simultaneously planned project requests for that subcorridor, at any time in the global plan window, and the allowed maximum number of project requests.

2.2.4. Goal of the optimization

The goal of the optimization slightly changes depending on whether the problem is viewed as single or multi-objective. In a single-objective context, the goal is to minimize the sum of both objectives and the soft constraint penalties, while satisfying the hard constraints. In a multi-objective setting, the objective is to minimize both objectives. The soft constraints, by default, are divided equally over the objectives, and the solutions must still satisfy the hard constraints. A second goal in multi-objective optimization is to find solutions with a large diversity in terms of the trade-off between the objectives.

2.3. Mathematical model of the problem

Next, the problem will be presented as a mathematical optimization model. This model gives a complete and formal description of the problem at hand. Please note that the model does not describe an efficient way of calculating the objectives and constraints, but rather focuses on readability. More on the efficient calculation of the objectives and constraints can be found in section 4.1.

The model is defined as follows:

- First, the parameters are defined. These are part of the input data, independent of the actual solution, and never change during the optimization.
- Second, the decision variables are defined. These are the variables that have to be optimized; they are changed during the optimization and any change in the values of the objective functions and constraints can only happen when the decision variables are changed.
- Next, a number of predicates and functions are defined. Predicates are boolean-valued functions and are used to specify constraints. Functions are used primarily for a shorter notation.
- Then, the objective functions are defined. The objective functions measure the quality of a solution. This is a minimization problem, so the smaller the objective functions the better. The problem is currently written with two objectives; these can either be optimized simultaneously using a multi-objective algorithm or combined in any way and be optimized using a single-objective algorithm. The soft constraints are also part of the objective, these are defined in section 2.3.8.
- Finally, the constraints are specified. These are equalities or inequalities that must hold for the solution to be valid. Even though the objective value may be very small, a constraint violation would still render that solution useless. Afterward, the way the soft constraint penalties can be computed is also described.

2.3.1. Parameters

Indices

$i(,j)$	Project request index
c	Corridor index
sc	Sub-corridor index
t	Time index
ty	Maintenance type index
Sets	
P	Set of all projects
C	Set of all corridors
SC	Set of all sub-corridors
SC_i	Set of sub-corridors affected by project request i
Dates and times	
T_{start}	Start date of the global plan period
T_{end}	Number of hours in the global plan window
d_i	Duration of project request i
TWS_i	Time before which project request i may not be started
TWE_i	Time before which project request i must be finished
Numerical parameters	
N:	Number of project requests
bfi	Total allowed BFI staff at any point in time
bvl	Total allowed BVL staff at any point in time
thl	Total allowed THL staff at any point in time
BFI_i	BFI staff used by project request i
BVL_i	BVL staff used by project request i
THL_i	THL staff used by project request i
$maxCor_c$	Maximum number of TVPs on corridor c
$mwsc$	Maximum number of weekends affected by TVPs per sub-corridor
mwc	Maximum number of weekends affected by TVPs per corridor
mpi	Minimum number of hours that should be between two TVPs in a corridor
$maxAtOneLocation$	Maximum number of project requests that may be active at one sub-corridor at any point in time
TR_{sc}^t	Number of passengers estimated to go through sub-corridor sc at time t
$ermCost$	Cost of an extra minute of travel
ERM_{sc}	Number of extra minutes of travel per passenger when sub-corridor sc is taken out of service
BUS_{sc}	Percentage of travellers having to take replacement buses when sub-corridor sc is taken out of service
FT_{sc}^t	Number of freight trains expected to go through sub-corridor sc at time t
ATC_p	Amount of alternative travel costs when p passengers are affected
$FINE_{sc}$	Fine that has to be paid per freight train when sub-corridor sc is taken out of service
COP_i	Personnel costs of project request i
COS_i	Security costs of project request i
COO_i	Constant costs of project request i
PCM_t	Personnel cost multiplier at time t.
MO_t	Month multiplier at time t

2.3.2. Decision variables

$x_i, i = 1...N,\ x_i \in \mathbb{N}, 0 \le x_i < T_{end}$: start time of project request i in number of hours since T_{start}

2.3.3. Predicates

overlap(i, j)	True if project requests i and j overlap in time
overlap(i, t_1, t_2)	True if project request i overlaps with the time window (t_1, t_2)
hinders(i)	True if project request i causes a disruption by hindering (part of) the affected infrastructure
conflict(i, j)	True if project requests i and j would cause a conflict violation when planned in an overlapping manner
dependency(i, d)	True if dependency d is relevant to project request i, i.e. when project request i is planned during the specified time window of d, a dependency violation would arise
prerequisite(i, j)	True if project request i is a prerequisite of project request j
combinationConflict(i, j)	True if project requests i and j would cause a combination matrix conflict when planned in an overlapping manner
requiredWindow(i)	True if project request i has a required window which is tighter than the global plan window on at least one side
requestActive(P, t)	True if any request in the set P has a planned period which overlaps with time t

2.3.4. Functions

overlapping(t)	Returns the set of all project requests whose planned period overlaps with time t
overlapping(t, P)	Returns the set of all project requests in P whose planned period overlaps with time t
affectsCorridor(c)	Returns the set of all project requests affecting corridor c
affectsSubcorridor(sc)	Returns the set of all project requests affecting corridor sc
getTVPs(P)	Returns a set of TVPs, i.e. overlapping periods of at least 24 hours, from the requests in set P.
getHinders()	Returns all project requests i for which hinders(i) is true
weekendsAffected(tvp)	Returns the number of weekends affected by TVP tvp
timeBetween(tvp1, tvp2)	Returns the time between TVPs $tvp1$ and $tvp2$
pbMax(sc, t)	Returns the maximum passenger blockage of project requests affecting subcorridor sc at time t
gbMax(sc, t)	Returns the maximum goods train blockage of project requests affecting subcorridor sc at time t
scale(x, n)	Returns the number x scaled up to the nearest multiple of n
atc(n)	Returns the alternative travel costs for an overlapping period in which n passengers are affected

2.3.5. Objective functions

$$f_1(\mathbf{x}) = \sum_{t=0}^{T_{end}} \sum_{sc \in SC} \text{pbMax}(sc,t) \cdot (TR_{sc}^{t} \cdot MO_t \cdot ERM_{sc} \cdot (1 + 0.3BUS_{sc}) \cdot ermCost) \tag{2.1.1}$$
$$+ \text{atc}(\text{pbMax}(sc,t) \cdot TR_{sc}^{t} \cdot MO_t) + \text{gbMax}(sc,i) \cdot (FT_{sc}^{t} \cdot FINE_{sc})$$

$$f_2(\mathbf{x}) = \sum_{i \in P} \left(COO_i + \sum_{t=x_i}^{x_i+\text{scale}(d_i,8)} PCM_t \cdot \frac{COP_i}{d_i} \right)$$
$$+ \sum_{sc \in SC} \sum_{tvp \in \text{getTVPs}(\text{affectsSubcorridor}(sc))} \max_{i \in tvp} \frac{COS_i}{|SC_i|} \tag{2.1.2}$$

2.3.6. Objective function explanation

Equation 2.1.1 is used to calculate the impediment costs. The costs are computed per hour, per sub-corridor. The first term computes the ERM for passengers, based on the maximum degree of passenger blockage for active project requests, the expected number of travellers, month index, ERM per passenger for the given sub-corridor, the expected number of passengers to travel by replacement bus, and the cost per ERM. The second term computes the alternative travel costs, based on the expected number of affected travellers computed similarly as in the first term. The third and final term computes the impediment costs for freight trains, based on the maximum degree of freight train blockage, the expected number of freight trains and the fine that has to be paid per freight train.

Equation 2.1.2 is used to calculate the maintenance costs. The first part of the maintenance costs is computed per project request. The first term contains the constant costs and is currently independent of the assigned timeslot for the project request. The second term is to calculate the personnel costs, where the duration of the project is scaled to a multiple of 8 hours because that is considered a full shift length. The personnel cost multiplier is also used. Finally, the security costs are calculated per sub-corridor. For each sub-corridor, the TVPs are computed, and for each TVP the maximum security costs of all project requests in that TVP are added to the total security costs.

2.3.7. Constraints

$$\text{combinationConflict}(i,j) \implies \neg\text{overlap}(i,j) \qquad \forall i,j \in P \tag{2.2.1}$$

$$\text{prerequisite}(i,j) \implies x_j + d_j \le x_i \qquad \forall i,j \in P \tag{2.2.2}$$

$$\text{requiredWindow}(i) \implies \max(0, TWS_i) \le x_i \le x_i + d_i \le \min(T_{end}, TWE_i) \qquad \forall i \in P \tag{2.2.3}$$

$$\sum_{i \in \text{overlapping}(t)} BFL_i \le bfl \qquad 1 \le t \le T_{end} \tag{2.2.4}$$

$$\sum_{i \in \text{overlapping}(t)} BVL_i \le bvl \qquad 1 \le t \le T_{end} \tag{2.2.5}$$

$$\sum_{i \in \text{overlapping}(t)} THL_i \le thl \qquad 1 \le t \le T_{end} \tag{2.2.6}$$

$$\text{hinders}(i) \land \text{hinders}(j) \land \text{conflict}(i,j) \implies \neg\text{overlap}(i,j) \qquad \forall i,j \in P \tag{2.2.7}$$

$$\text{hinders}(i) \land \text{dependency}(i,d) \implies \neg\text{overlap}(i,d) \qquad \forall i \in P, d \in D \tag{2.2.8}$$

$$\left|\text{getTVP}(\text{getHinders}() \cap \text{affectsCorridor}(c))\right| \le maxCor_c \qquad \forall c \in C \tag{2.2.9}$$

$$\sum_{tvp \in \text{getTVP}(\text{getHinders}() \cap \text{affectsSubcorridor}(sc)} \text{weekendsAffected}(tvp) \le mwsc \qquad \forall sc \in SC \tag{2.2.10}$$

$$\sum_{tvp \in \text{getTVP}(\text{getHinders}() \cap \text{affectsCorridor}(c)} \text{weekendsAffected}(tvp) \le mwc \qquad \forall c \in C \tag{2.2.11}$$

$$(t1, t2) \in \text{getTVPs}(\text{getHinders}() \cup \text{affectsCorridor}(c)) \implies \text{timeBetween}(t1,t2) \ge mpi \qquad \forall c \in C \tag{2.2.12}$$

$$\left|\text{overlapping}(t) \cap \text{affectsSubcorridor}(sc)\right| \le maxAtOneLocation \qquad \forall sc \in SC, 1 \le t \le T_{end} \tag{2.2.13}$$

Equation 2.2.1 enforces the CombinationMatrix constraint. For each pair of project requests with a potential maintenance type conflict, overlap is not allowed.

Equation 2.2.2 enforces the Prerequisite constraint. If a project request is a prerequisite of another, it should be finished before the other starts.

Equation 2.2.3 enforces the time window constraint. Each project request with a required window must fall within that required window. Furthermore, it may not be planned before or after the global planned window.

Equations 2.2.4, 2.2.5 and 2.2.6 enforce the different personnel constraints. For each possible time, it is checked if the total number of personnel used by projectactive at that time exceeds the maximum for that personnel type.

Equation 2.2.7 enforces the conflict constraints. For each pair of project requests with a potential conflict, overlap is not allowed, if they both cause hinder. Note that in the constraint scenario there are various types of conflict which may all have a different severity and/or penalty. In this model, these are not shown separately for brevity.

Equation 2.2.8 enforces the dependency constraints which indicate a prohibition of working on certain subcorridors in certain time periods. If a dependency and a project request causing hinder have a potential conflict, they may not overlap.

Equation 2.2.9 enforces the constraint which specifies a maximum number of TVPs that can be held in a corridor. For each corridor, it is checked that the number of TVPs does not exceed the maximum.

Equations 2.2.10 and 2.2.11 enforce the constraints which specify the maximum number of weekends that may be effected by TVPs within a single corridor and subcorridor respectively. For each corridor and subcorridor, the number of weekends that are affected by a TVP are counted and it is checked that the maximum is not exceeded.

Equation 2.2.12 enforces the constraint which specifies the minimum time between two TVPs within the same corridors. For each pair of TVPs within a corridor, the time between them must be greater than the specified minimum time.

Equation 2.2.13 enforces the constraint which specifies the maximum number of project requests that may be active simultaneously within a subcorridor at any point in time. For each subcorridor and time, we check that the maximum number of projects is not exceeded.

Finally, it should be mentioned that not all of these constraints may be hard, as explained in section 2.2.3. The calculation of soft constraint penalties is specified in the next section.

2.3.8. Soft constraints

As explained in section 2.2.3, the soft constraint penalties are computed using the number of violations, the amount per violation, the penalty, and the penalty type. In this section, the way the soft constraints are computed is mathematically defined. To do this, some parameters need to be defined for each constraint. To avoid unnecessary repetition, these parameters are defined in general.

Soft constraint parameters

SCP_x	Soft constraint penalty for x
SoC_x	one if x is a soft constraint, zero otherwise
HaC_x	one if x is a hard constraint, zero otherwise
$SCOP_x$	one if x has a one time soft constraint penalty, zero otherwise
$SCLP_x$	one if x has a linear soft constraint penalty, zero otherwise
$SCEP_x$	one if x has an exponential soft constraint penalty, zero otherwise

It is assumed that only one of SoC and HaC is one, and only one of $SCOP$, $SCLP$ and $SCEP$ is one, for each constraint.

The following mapping from constraint names to parameter names is used:

Prerequisite	pr
Required window	rw
BFI, BVL, THL personnel	bfip, bvlp, thlp
Conflicts	co
Dependencies	dp
Max TVP corridor	mTVPc
Max weekends subcorridor	mwsc
Max weekends corridor	mwc
Mintime between TVPs	mtTVP
Maximum at one subcorridor	msc

So, as an example, SCP_{mwc} specifies the soft constraint penalty of the max weekends corridor constraint.

Soft constraint calculation

Next, the formulas for calculating the soft constraint penalty for each constraint are specified. First, some helper functions are declared. Then, the penalty term for each constraint is computed.

Helper functions

$$maxTVPC(c) = \max(0, \left|\text{getTVP}(\text{getHinders}() \cap \text{affectsCorridor}(c))\right| - maxCor_c)$$

$$maxWC(c) = \max\left(0, \sum_{tvp \in \text{getTVP}(\text{getHinders}() \cap \text{affectsCorridor}(c)} \text{weekendsAffected}(tvp) - mwsc\right)$$

$$maxWSC(sc) = \max\left(0, \sum_{tvp \in \text{getTVP}(\text{getHinders}() \cap \text{affectsSubcorridor}(sc)} \text{weekendsAffected}(tvp) - mwsc\right)$$

$$mTV(c, t1, t2) = \max(0, mpi - \text{timeBetween}(t1, t2))$$

$$pRV = |\{(i, j)|\text{prerequisite}(i, j) \wedge x_j + d_j > x_i\}|$$

$$bVLV = \max_{1 \le t \le T_{end}} \left(\sum_{i \in \text{overlapping}(t)} BVL_i\right) - bvl$$

$$bFIV = \max_{1 \le t \le T_{end}} \left(\sum_{i \in \text{overlapping}(t)} BFI_i\right) - bfi$$

$$tHLV = \max_{1 \le t \le T_{end}} \left(\sum_{i \in \text{overlapping}(t)} THL_i\right) - thl$$

$$cOV = \left|\{(i, j)|\text{hinders}(i) \wedge \text{hinders}(j) \wedge \text{conflict}(i, j) \wedge \text{overlap}(i, j)\}\right|$$

$$dPV = \left|\{(i, d)|\text{hinders}(i) \wedge \text{dependency}(i, d) \wedge \text{overlap}(i, d)\}\right|$$

$$mOV(sc) = \max_{1 \le t \le T_{end}} \left(\left|\text{overlapping}(t) \cap \text{affectsSubcorridor}(sc)\right|\right) - maxAtOneLocation$$

Penalty terms

$$\begin{aligned} P_{mTVPc}(\mathbf{x}) = \sum_{c \in C} \Big(& SCOP_{mTVPc} \cdot (\min(1, maxTVPC(c)) \cdot SCP_{mTVPc}) \\ & + SCLP_{mTVPc} \cdot (SCP_{mTVPc} \cdot maxTVPC(c)) \\ & + SCEP_{mTVPc} \cdot (2^{maxTVPC(c)} \cdot SCP_{mTVPc})\Big) \end{aligned} \tag{2.4.1}$$

$$\begin{aligned} P_{mwc}(\mathbf{x}) = \sum_{c \in C} \Big(& SCOP_{mwc} \cdot (\min(1, maxWC(c)) \cdot SCP_{mwc}) \\ & + SCLP_{mwc} \cdot (SCP_{mwc} \cdot maxWC(c)) \\ & + SCEP_{mwc} \cdot (2^{maxWC(c)} \cdot SCP_{mwc})\Big) \end{aligned} \tag{2.4.2}$$

$$\begin{aligned} P_{mwsc}(\mathbf{x}) = \sum_{sc \in SC} \Big(& SCOP_{mwsc} \cdot (\min(1, maxWSC(sc)) \cdot SCP_{mwsc}) \\ & + SCLP_{mwsc} \cdot (SCP_{mwsc} \cdot maxWSC(sc)) \\ & + SCEP_{mwsc} \cdot (2^{maxWSC(sc)} \cdot SCP_{mwsc})\Big) \end{aligned} \tag{2.4.3}$$

$$\begin{aligned} P_{mtTVP}(\mathbf{x}) = \sum_{c \in C} \Bigg(\sum_{(t1, t2) \in (\text{getHinders}() \cap \text{affectsCorridor}(c))} \Big(& SCOP_{mtTVP} \cdot (\min(1, mTV(c, t1, t2)) \cdot SCP_{mtTVP}) \\ & + SCLP_{mtTVP} \cdot (SCP_{mtTVP} \cdot mTV(c, t1, t2)) \\ & + SCEP_{mtTVP} \cdot (2^{mTV(c,t1,t2)} \cdot SCP_{mtTVP}\Big)\Bigg) \end{aligned} \tag{2.4.4}$$

$$\begin{aligned} P_{pr}(\mathbf{x}) = & SCOP_{pr} \cdot \min(1, pRV) \cdot SCP_{pr} + SCLP_{pr} \cdot SCP_{pr} \cdot pRV \\ & + SCEP_{pr} \cdot 2^{pVR} \cdot SCP_{pr} \end{aligned} \tag{2.4.5}$$

$$P_{rw}(\mathbf{x}) = \sum_{\{i|\text{requiredWindow}(i) \wedge (x_i < TWS_i \vee x_i + d_i > TWE_i)\}} SCP_{rw} \tag{2.4.6}$$

$$\begin{aligned} P_{bvlp}(\mathbf{x}) = & SCOP_{bvlp} \cdot \min(1, bVLV) \cdot SCP_{bvlp} + SCLP_{bvlp} \cdot (SCP_{bvlp} \cdot bVLV) \\ & + SCEP_{bvlp} \cdot (2^{bVLV} \cdot SCP_{bvlp}) \end{aligned} \tag{2.4.7}$$

$$P_{bfip}(\mathbf{x}) = SCOP_{bfip} \cdot \min(1, bFIV) \cdot SCP_{bfip} + SCLP_{bfip} \cdot (SCP_{bfip} \cdot bFIV)$$

$$+ SCEP_{bfip} \cdot (2^{bFIV} \cdot SCP_{bfip}) \tag{2.4.8}$$

$$\begin{aligned} P_{thlp}(\mathbf{x}) &= SCOP_{thlp} \cdot \min(1, tHLV) \cdot SCP_{thlp} + SCLP_{thlp} \cdot (SCP_{thlp} \cdot tHLV) \\ &\quad + SCEP_{thlp} \cdot (2^{tHLV} \cdot SCP_{thlp}) \end{aligned} \tag{2.4.9}$$

$$\begin{aligned} P_{co}(\mathbf{x}) &= SCOP_{co} \cdot \min(1, cOV) \cdot SCP_{co} + SCLP_{co} \cdot (SCP_{co} \cdot cOV) \\ &\quad + SCEP_{co} \cdot (2^{cOV} \cdot SCP_{co}) \end{aligned} \tag{2.4.10}$$

$$\begin{aligned} P_{dp}(\mathbf{x}) &= SCOP_{dp} \cdot \min(1, dPV) \cdot SCP_{dp} + SCLP_{dp} \cdot (SCP_{dp} \cdot dPV) \\ &\quad + SCEP_{dp} \cdot (2^{dPV} \cdot SCP_{dp}) \end{aligned} \tag{2.4.11}$$

$$\begin{aligned} P_{msc}(\mathbf{x}) &= \sum_{sc \in SC} \Big(SCOP_{msc} \cdot (\min(1, mOV(sc)) \cdot SCP_{msc}) \\ &\quad + SCLP_{msc} \cdot (SCP_{msc} \cdot mOV(sc)) \\ &\quad + SCEP_{msc} \cdot (2^{mOV(sc)} \cdot SCP_{msc}) \Big) \end{aligned} \tag{2.4.12}$$

Then, the total of soft constraint penalties is defined as

$$\begin{aligned} TotalSC(\mathbf{x}) &= SoC_{mTVPc} \cdot P_{mTVPc}(\mathbf{x}) + SoC_{mwc} \cdot P_{mwc}(\mathbf{x}) + SoC_{mwsc} \cdot P_{mwsc}(\mathbf{x}) + SoC_{mtTVP} \cdot P_{mtTVP}(\mathbf{x}) \\ &\quad + SoC_{pr} \cdot P_{pr}(\mathbf{x}) + SoC_{rw} \cdot P_{rw}(\mathbf{x}) + SoC_{bvlp} \cdot P_{bvlp}(\mathbf{x}) + SoC_{bfip} \cdot P_{bfip}(\mathbf{x}) + SoC_{thlp} \cdot P_{thlp}(\mathbf{x}) \\ &\quad + SoC_{co} \cdot P_{co}(\mathbf{x}) + SoC_{dp} \cdot P_{dp}(\mathbf{x}) + SoC_{msc} \cdot P_{msc}(\mathbf{x}) \end{aligned} \tag{2.5}$$

2.3.9. Goal of the optimization

Now that all objectives and constraints have been specified, the goal of the optimization can be defined. The goal of the optimization is to find $\mathbf{x} = \{x_1, \ldots, x_N\}$ such that $f_1(\mathbf{x}) + f_2(\mathbf{x}) + TotalSC(\mathbf{x})$ is minimized and the hard constraints are satisfied.

2.3.10. Structure of the model

Writing the problem in a linear or quadratic form could enable the use of specified solver techniques for these kinds of structures. However, it is infeasible to write this problem with such structure due to the complexity and non-linearity of the constraints. With a single decision variable for each project request, this non-linearity can, for example, clearly be seen in the dependency constraints. Say there is a single event prohibiting project requests, and a single project request to be planned. The constraint violation will be zero as long as the project request and the event do not overlap, will suddenly jump to one as soon as they do, and will jump back to zero when the request is scheduled after the event. This behaviour is like a step function which is clearly not linear.

The idea of achieving a linear structure was quickly abandoned, but an attempt was made to create a quadratic model for the problem. This was done by increasing the number of decision variables; instead of a single decision variable per project request, indicating the start time, for each (project request, hour) pair, a binary decision variable was introduced to specify whether the project request was active at the given hour. Using this model, most of the constraints could be written linearly or quadratically. However, the corridor constraints operating on TVPs were still too complex to be written in this form. Although it is probably possible to introduce more decision variables to write these constraints quadratically as well, this would introduce a vast increase in complexity. Together with the fact that the problem size is too large for a linear or quadratic solver anyway, this idea was abandoned and the non-quadratic model was kept.

2.4. Problem data

To be able to test solution methods on a real world dataset, ProRail has provided Macomi with the real data for both 2019 and 2020 which was also used to create the real maintenance schedule for those respective years. In this section, some characteristics of these datasets are explored. Furthermore, the differences of between the two datasets are also looked at. Finally, the purpose of the two datasets in this thesis is briefly discussed.

2.4.1. Data characteristics

Table 2.4 summarizes some characteristics of both data sets. These are explained and discussed further below.

Number of corridors and sub-corridors
Both datasets use the same rail network and therefore have the same number of corridors and sub-corridors. There are 155 sub-corridors present, as well as 49 sub-corridors.

Number of project requests
The first part of table 2.4 shows the total number of project requests and the number of project requests with certain properties. Both datasets contain a substantial number of project requests to be planned. The 2020 data contains fewer requests than the 2019 data, but in other characteristics it will become clear that despite having fewer requests, the requests in the 2020 data are generally longer and more impactful. Both datasets have an approximately 50/50 split between project requests which actually hinder the sub-corridor and those that do not. As explained before, projects that do not hinder the sub-corridor do not cause availability costs and are not taken into account in several constraints.

It can be seen that there are a substantial number of project requests that have a required time window which is tighter than the global plan window. Especially the requests in 2020 have such a window relatively often. It must be said that most of the required time windows are quite large and do not limit the possible plan moments for a request by much. On the other hand, however, in both datasets there are a number of requests which have a required window which is too small for the request to actually fit in. This means that, when using the required window constraint as hard constraint which is the case in the base scenario, there will always be a few broken hard constraints.

Neither dataset has any requests that have one or more prerequisite project requests. This is the case because data about which project requests need to be finished before others is currently not readily available at ProRail.

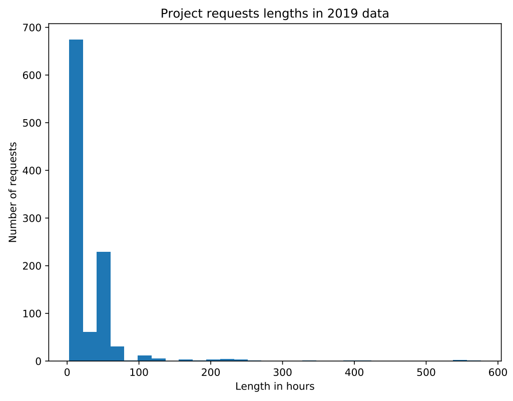
Finally, in both datasets there are a number of project requests that have no maintenance costs. Again, this is due to missing data. Rather than making a possibly inaccurate guess about the maintenance costs of these project requests, it has been decided that they will be set to zero at this point. It should be noted that the number of requests without maintenance costs is relatively higher in the 2019 data than in the 2020 data. The project requests that do not have any maintenance costs do not take part in the trade-off between maintenance and availability and thus limit the range of this trade-off.

Impact of project requests
The next two sections of table 2.4 show the number of affected sub-corridors and the length per request in each of the two datasets. Together, they provide a reliable picture of the (average) impact a request has in terms of availability. As already indicated previously, it is observed that the requests in the 2020 generally have more impact than those in the 2019 data. This is also shown in figure 2.3 which shows histograms for both the number of sub-corridors and the length of a request, for both datasets. It is interesting to see that the 2020 data has fewer requests but the requests have more impact. At this point, it cannot be determined if one is more difficult than the other. However, the fact that these differences are present means both datasets can be used to see if developed solution methods are robust given this kind of change in the input data.

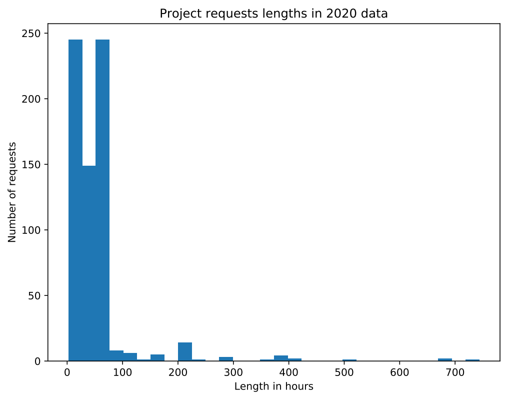
The distribution of these two characteristics should also be noted. In both datasets, although somewhat more in the 2019 data, there are many requests which are relatively small in terms of length and number of affected sub-corridors, and a relatively small number of larger, more impactful requests. This means that a relatively small number of the requests make up a relatively large portion of the availability costs, and that finding an optimal or near-optimal plan moment for these large requests is more important than finding it for the small requests. This insight is used later in chapter 7 to develop a greedy algorithm.

Impact of constraints
The final three characteristics in table 2.4 show the impact of several constraints. The number of affected corridors indicates the difficulty of the corridor constraints. A larger number of affected corridors per request does not only mean it is harder to satisfy the corridor constraints, but also means there is more interdependency between the requests. Changing the plan moment of a request which affects many sub-corridors means it potentially interacts with all other requests in any of those sub-corridors. This increases the complexity of the problem substantially. There are a total of 48 corridors present in the complete rail network. As expected, the higher number of affected sub-corridors in the 2020 data compared to the 2019 data also leads to a higher number of affected corridors, on average.

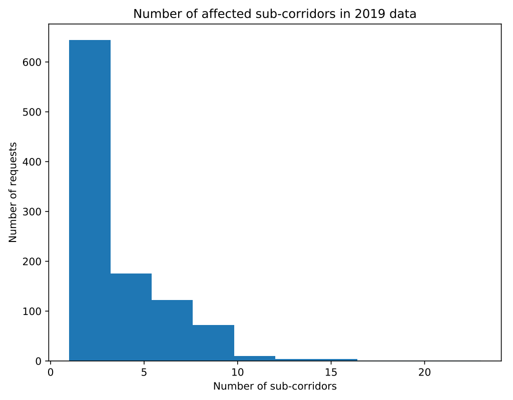
Next, the number of potentially conflicting dependencies indicates the difficulty of the dependency constraint. It can be seen that there are quite some dependencies that have to be taken into account. The difference in number of dependencies between the two datasets is large. Even though the average length of a



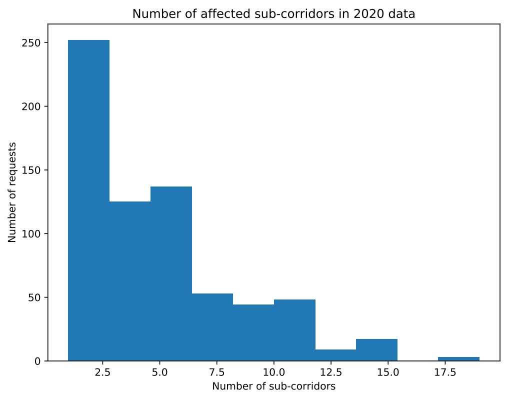
(a) Lengths of project requests, 2019



(b) Lengths of project requests, 2020



(c) Affected sub-corridors per project request, 2019



(d) Affected sub-corridors per project request, 2020

Figure 2.3: Impact of project requests in 2019 and 2020 data

dependency is 122 hours in 2019 versus 81 hours in 2020, requests in 2020 have almost four times as much dependencies to take into account compared to those in 2019. This fact, combined with the larger average length of a request in 2020, could indicate that there are fewer possible plan moments for a request in 2020 compared to 2019.

Finally, the number of potentially conflicting requests shows the difficulty of the conflict constraints. Considering that only the requests which hinder the affected sub-corridors are taken into account, on average, approximately one third of the project requests are potentially conflicting with a certain request in both datasets. This means that planning a request at a certain time period can already prevent many other requests from being planned there. The difficulty of the conflict constraint was already observed in the pilot study done by Macomi as well; when excluding the conflict constraints the algorithm was able to find a solution with a massive cost reduction. Of course, this is not a realistic solution in real life as it would lead to large parts of the network taken out of service simultaneously, but it is an interesting insight.

2.4.2. Purpose of the datasets

During their pilot study, Macomi have mostly tested with the 2019 data. This means that there are some clear baseline results available of their existing algorithm on this data. Therefore, the 2019 data has been used as the primary data set to measure the effects of certain algorithms or other improvements. The 2020 data has been used as a test set to validate that certain results still hold for a different dataset on the same problem. The fact that the 2020 data is quite different from the 2019 data, with having fewer but more impactful blocks, makes this a good dataset to use for this purpose. Obviously, both types of data can occur in real life and robustness over this change is an important characteristic of a good solution method.

During the remainder of this document, the usage of the 2019 data can be implicitly assumed unless otherwise specified.

Characteristic	2019 data	2020 data
Total number of sub-corridors	155	155
Total number of corridors	49	49
Number of project requests	1033	688
with hinder	526	316
with required time window	305	484
that do not fit in their required window	4	9
with prerequisites	0	0
with non-zero maintenance costs	455	530
Number of affected subcorridors per request		
Min	1	1
First quartile	1	2
Median	3	4
Mean	3.47	4.75
Third quartile	5	7
Max	23	19
Length of requests in hours		
Min	3	2.33
First quartile	5	12
Median	6	50
Mean	26.7	49.5
Third quartile	51	52
Max	576	744
Number of affected corridors per request		
Min	1	1
First quartile	1	1
Median	2	2
Mean	2.29	2.73
Third quartile	3	4
Max	8	10
Number of potentially conflicting dependencies per request (with hinder only)		
Min	0	0
First quartile	13	10
Median	24	79
Mean	41.3	159.7
Third quartile	50.75	232
Max	308	1040
Number of potentially conflicting requests per request (with hinder only)		
Min	0	0
First quartile	77	56.25
Median	201	116.5
Mean	181.3	117.3
Third quartile	248	171.25
Max	467	277

Table 2.4: Data characteristics

3 Existing solution methods

Now that the problem has been extensively explained in chapter 2, existing solution methods can be considered. In this chapter, an overview of existing solution methods is provided. First, a literature survey is performed, in which solution methods that have been used in similar problems are identified and described. Afterward, an overview of evolutionary and multi-objective evolutionary algorithms is given. Finally, the existing solution method and platform by Macomi is described.

3.1. Maintenance scheduling problems

First, an overview of research that tackles rail maintenance scheduling is provided, and after that, a brief overview of research in other scheduling problems that have similar characteristics as the rail maintenance scheduling problem is given. These are timetabling, and maintenance scheduling in other network-based structures. The goal is to provide an overview of the similarities and differences between the problems tackled in research and the given real-world problem, as well as to draw conclusions regarding the types of solution methods that are used and how they may or may not be suitable to solve the given problem.

3.1.1. Rail maintenance scheduling

Rail maintenance scheduling is a broad field where many different kinds of problems have been considered. Although they all concern some kind of maintenance on railway infrastructure, there are many differences between the various papers tackling a variant of this problem.

The first characteristic that changes is the granularity that is considered. Some literature considers a low-level model in which individual tracks and switches are taken into account. Often, a fabricated example or only a small part of a real-life rail network is considered in these types of papers. Others look at a bigger picture like a complete tram network of a city. There are clear differences in formulation and constraints between these two types of problems.

Another changing characteristic is the use of a deterioration model. Some researchers do not only take into account when to schedule certain maintenance operations but also what to consider for scheduling at what time based on some kind of deterioration model and a trade-off between an increasing chance of failure and lower maintenance costs. Others assume a list of necessary maintenance tasks is provided, possibly with time window constraints to indicate when specific maintenance is needed.

The final major characteristic that changes is the use of periodic and aperiodic maintenance tasks, the latter also being called "projects" in literature. Both of these types of tasks, as well as a combination of them, are considered in the literature.

When analyzing ProRail's railway maintenance scheduling problem with respect to these three characteristics, it can be seen that it considers a large railway network and uses a high level of granularity. Furthermore, a deterioration model is not present, rather, the complete list of necessary maintenance tasks is known beforehand. Finally, only aperiodic tasks are given; no tasks need to be executed in a cyclic fashion. From the many different characteristics in rail maintenance scheduling literature as well as the presence of some specific, complex constraints, it becomes clear that there is no literature tackling the exact same problem that needs to be solved. This means there is no clear answer as to a specific algorithm that will perform well. However, by considering many problems with similar characteristics it is possible to get a good idea of the types of algorithms that have been successfully used, in order to make an informed decision on the solving method afterwards.

Papers on rail maintenance scheduling

Budai *et al.* [20] introduce the Preventive Maintenance Scheduling Problem (PMSP) where a set of routine tasks, as well as one-time projects, have to be scheduled. Similarities with ProRail's problem are the fact that possession and maintenance costs are being used, as well as similarity in some constraints such as "project a is combinable with project b". However, there are some key differences; the PMSP doesn't consider locations and location conflicts, and all costs are completely independent unlike in the ProRail situation. The problem is initially solved by the CPLEX 7.1 MIP solver, but small (+- 15 tasks) problems already took more than 3 hours sometimes. Therefore, some heuristics have been developed to improve performance. One heuristic individually plans each routine task optimally without looking at their interaction, after which the projects are planned to be as much as possible during routine maintenance work. Other heuristics are similarly greedy-like but the order in which maintenance works is planned is optimized, for example, by planning the most frequent work first, or the most costly work first.

Budai-Balke *et al.* [21] test various genetic and memetic algorithms (GA, MA) on the PMSP. A memetic algorithm is a genetic algorithm with a local search component built in. According to the authors, "GAs [have been] used in many maintenance settings". Various GAs and MAs were tested, with hill climbing [57], simulated annealing [47] and tabu search [41] as local search procedures. Adding the local search improved performance for "almost all of the 80 instances", and GA + simulated annealing was the best combination in this case.

Jenema [43] has written a Master Thesis at TU Delft for ProRail maintenance scheduling. She only uses the "top 10 most determinative maintenance activities" meaning the problem is of much smaller size. A linear model is used, and a branch and bound algorithm is used together with a relaxation method. She considers the problem slightly more low level, where e.g. individual track switches are taken into account.

Andrade and Texeira [15] use two objectives: maintenance costs and train delays. The authors focus more on the deterioration model and deciding when maintenance should be performed at all. Bi-objective simulated annealing is used to solve the problem.

Soh *et al.* [64] review six papers regarding scheduling techniques for preventive maintenance of railway infrastructure. Of these, three used a genetic algorithm, one used local search, one used an ontology-based method and one used strategic gang scheduling. One of the genetic algorithms was multi-objective.

Zhang *et al.* [72] use a deterioration model and consider "importance" of track segments which is similar to the notion of affected passengers for certain sub-corridors. The authors argue that "Such a scheduling problem has been proven to be a non-deterministic polynomial-time. The standard methods of non-linear programming are therefore not suitable, particularly when the number of segments considered is large. The GA-based approach has been proven to be effective in solving NP-hard problems and has been successfully applied to many engineering optimisation problems, including maintenance scheduling. Hence, the GA is used and enhanced to solve this problem.".

Caetano and Texeira [22] use a multi-objective approach where the objectives are unavailability and life-cycle costs of the track. They use NSGA-II [30], a multi-objective genetic algorithm, as their solution method.

Khalouli *et al.* [45] use the PMSP model [20] and test it with a 2-year horizon and approximately 30 projects. Using the CPLEX MIP solver they manage to solve 62% of instances within 3 hours. Afterwards, they use an ant colony optimization [35] (ACO) algorithm which reaches the optimum in 55% of these cases. The average runtime is 200 seconds for ACO and 4808 seconds for CPLEX.

Dao *et al.* [29] use a binary optimization model which is tested by using CPLEX on problems with 5 projects and 12 time steps which can be considered very small.

Kiefer *et al.* [46] formulate their problem as a mixed integer programming model but quickly find out its runtime limitations on a larger test problem. Therefore they develop a metaheuristic based on large neighbourhood search. They test it on the Vienna tram network which is the largest test problem encountered. Using the CPLEX solver, the 24-hour time limit was already exceeded.

Peralta *et al.* [56] use a multi-objective approach with cost and delay as objectives. They use two methods: AMOSA [16], a multi-objective simulated annealing algorithm, and NSGA-II [30]. They also use a non-random initialization; solutions are generated following certain expert heuristic rules. Such rules are also available for the ProRail problem. An interesting result is that AMOSA performed better than NSGA-II, likely because of the non-random initialization.

3.1.2. Maintenance in network-based structures

Next, other maintenance problems in network-based structures have been studied.

First of all, road maintenance has been researched. This problem is similar to rail maintenance as it also affects travellers who have to reroute to get to their destination, and a similar tradeoff between impediment costs and maintenance costs exists. In the field of road maintenance scheduling, there are various solution methods being used, such as genetic algorithms [24, 26, 27], multiobjective evolutionary algorithms [34, 58], tabu search [69] and algorithms based on Markov decision processes [38, 52].

Second, a brief look was taken into power grid maintenance scheduling problems, which also show some similarities with the ProRail problem. Froger *et al.* [37] have performed a review of scheduling in the electricity industry and conclude that "the solution techniques mainly focus on metaheuristics and mathematical programming". The metaheuristics that are being used most are genetic algorithms and particle swarm optimization.

3.1.3. Timetabling

The final type of problem that has been looked into is (university) timetabling. Although there are some major differences between the university timetabling problem and the ProRail problem, there are some similarities as well: similar to assigning lectures in timetabling, project requests need to be assigned to time slots. Furthermore, both problems contain both hard and soft constraints. Gashgari *et al.* [39] have performed a review of exam scheduling techniques, and conclude that the most used methods are mixed integer programming (MIP) and metaheuristics (mostly genetic algorithms, simulated annealing and tabu search). The former is not applicable to the ProRail problem because it can not be formulated linearly. Other recent papers on university timetabling mostly use a wide variety of metaheuristics.

3.1.4. Conclusions

Based on the studied literature, the following can be concluded:

- Many exact solution methods such as linear or quadratic programming have been used in similar problems. However, the test problems used in literature are usually a lot smaller than ProRail's scheduling problem [20, 21, 29, 43, 45]. When using such an exact method, the runtime increases exponentially and for very small instances it can already take multiple hours to find the solution [20, 45]. Furthermore, the given problem cannot be written in a linear or quadratic form. This makes it infeasible for a generic (MIP) solver to solve the problem in reasonable time, and unlikely that the problem is viable to be solved by an exact method.

- The metaheuristic that is most used is the genetic algorithm; both the single-objective and the multi-objective versions have been used [21, 22, 56, 64, 72].

- Using a memetic component improves performance in both single- and multi-objective genetic algorithms [21, 56].

- Using a non-random initialization does not only increase the convergence speed and performance, but also seems to favour local search algorithms rather than genetic algorithms [56].

- Even with metaheuristics, the search space of the given problem might be too large, but this is hard to determine beforehand. Using non-random initialization [56] as well as smart, feasibility-preserving crossover and mutation operators will help [28]. If the search space turns out to be too large anyway, it may be possible to reduce it, for example by decreasing the time scale from hours to days or by partitioning the problem based on locations.

3.2. Evolutionary algorithms

In this section, a brief introdution to evolutionary algorithms is given. It provides the necessary background knowledge for both multi-objective evolutionary algorithms which are explained in the next section, as well as for the existing solution method by Macomi which is an evolutionary algorithm. Evolutionary algorithms are a family of population-based metaheuristic optimization algorithms which use mechanisms inspired by biological evolution, such as reproduction, mutation, recombination, and selection. The underlying idea behind these techniques is the same: given a population of individuals the environmental pressure causes natural selection (survival of the fittest) and this causes a rise in the fitness of the population. Recombination and mutation operators ensure novelty and diversity in the population, while selection pushes the quality of the population. Two evolutionary algorithms are explained in more detail; the genetic algorithm and the evolution strategy algorithm.

3.2.1. Genetic algorithm

The genetic algorithm is the most well-known evolutionary algorithm. Pseudocode for the genetic algorithm can be found in algorithm 3.1. The different components are briefly discussed below.

Algorithm 3.1 General scheme of a genetic algorithm

```
Initialize population with random candidate solutions
Evaluate each candidate
repeat
    Select parents
    Recombine pairs of parents
    Mutate the resulting offspring
    Evaluate new candidates
    Select individuals for the next generation
until Termination condition is satisfied
```

Representation

The representation used in a genetic algorithm is problem-specific, but a binary string representation is used most often. An individual in the population is represented by a binary string that can be translated to an actual solution of the problem that is attempted to be solved.

Fitness function

The fitness function, also called the objective function, determines the quality of a given solution. This quality is used in selection to determine which parents are selected for creating offspring.

Parent selection

From an existing population, parents are selected to create offspring. The parents are selected based on fitness; an individual with higher fitness has a larger chance to be chosen. Usually, all individuals have a non-zero chance to be chosen as a parent in a generation. Various selection methods are available, and they vary in the frequency a relatively bad solution is chosen over a good one. A selection method that is used often is binary tournament selection, in which two random individuals are selected from the population, and the fittest one is selected as a parent.

Recombination and mutation

When the parents are selected, they are used to create new individuals called offspring. In the genetic algorithm, this is done by crossover. A crossover operator takes two parent solutions and combines them such that two offspring are created, both of which contain part of the solution from both parents. A simple crossover used often is a one-point crossover, where the binary string representation of the parents is "cut" at a random point, and the first offspring takes the left part of parent one and the right part of parent two. The second offspring does this vice versa.

After a number of offspring are generated using the crossover method, the offspring are mutated with a certain probability. A mutation is a small change to an individual, and is used to explore new areas of the search space and avoid getting stuck in a local optimum. An example mutation is flipping a single bit in a binary string representation.

In a genetic algorithm, there is no further selection of individuals at this point; all selection happens when the parents are selected.

Termination condition

Various termination conditions can be used. Examples are: a set number of generations or runtime, a lack of improvement over multiple subsequent iterations, or reaching a preset fitness threshold.

3.2.2. Evolution strategy

The evolution strategy is also an evolutionary algorithm, but it has some slight differences compared to the genetic algorithm. Algorithm 3.2 shows pseudocode for an evolution strategy.

Algorithm 3.2 Pseudocode of a $(u/\rho \overset{+}{,} \lambda)$ evolution strategy

```
Initialize parent population $P_\mu$
repeat
    repeat
        Select randomly $\rho$ parents from $P_\mu$
        Recombine the $\rho$ selected parents to form an offspring $r$
    until $\lambda$ offspring are generated
    if comma-selection is used then
        The new parent population is determined from the offspring population
    else
        The new parent population is determined from both the old parent population and the offspring population
until a termination criterion is fulfilled
```

As can be seen there are four parameters that need to be set: the parent size μ, the offspring size λ, the number of parents for recombination ρ and the selection type, either offspring only (comma-selection) or both parents and offspring (plus-selection). Selection of the next parent population is done deterministically. Note that the fitness of individuals is not being used when selecting the parents. If ρ is equal to 1, it can be omitted and the algorithm becomes a $(\mu \overset{+}{,} \lambda)$-ES. In this case, there is no recombination and the algorithm becomes more of a population-based local search algorithm.

3.3. Multi-objective evolutionary algorithms

Multi-objective evolutionary algorithms (MOEAs) are a type of evolutionary algorithm which solve multi-objective optimization problems. First, multi-objective optimization problems are explained and a formal definition is given. Afterwards, an overview of various MOEAs that have been proposed over the years is given, and their similarities and differences are discussed.

3.3.1. Multi-objective optimization problem

A multi-objective optimization problem is an optimization problem with multiple objectives which are in some way conflicting; generally, given an (optimal) solution, improving one of the objectives will worsen another. This means that there is no single optimal solution in the sense that it gives the optimal value for both objectives. Instead, trade-offs need to be made between different objectives and the importance of one objective over the other is not clear by default.

There are multiple variants of a multi-objective optimization problem (MOP) that primarily differ in the type of constraints and objective functions used, for example a linear MOP or a quadratic MOP. However, due to the fact that the given problem cannot be written linearly or quadratically, the focus lies especially on *combinatorial* MOPs. Formally, a combinatorial MOP is defines as follows:

$$\begin{array}{ll} \text{minimize} & F(\mathbf{x}) = \{f_1(\mathbf{x}), \ldots, f_n(\mathbf{x})\} \\ \text{subject to} & \mathbf{x} \in \mathscr{S} \end{array}$$

Here, $\mathbf{x}$ is a vector of decision variables. $\mathscr{S}$ is the search space or feasible space, the set of all possible solutions. $F = \{f_1(\mathbf{x}), \ldots, f_n(\mathbf{x})\}$ is a vector of n objective functions. Without loss of generality it is assumed all objectives are to be minimized; if a objective f_i is to be maximized it is equivalent to minimize $-f_i$.

For the comparison of two solutions s_1 and s_2, there are two methods that are mainly used. The first method is Pareto dominance: s_1 is said to *dominate* s_2 (denoted by $s_1 \succ s_2$) if and only if s_1 is better than or equal to s_2 in all objectives and if there exists at least one objective according to which s_1 is strictly better than s_2:

$$s_1 \succ s_2 \longleftrightarrow \forall f \in F, f(s_1) \leq f(s_2) \wedge \exists f \in F, f(s_1) < f(s_2) \tag{3.1}$$

Two mutually non-dominated solutions are considered incomparable. A solution that is not dominated by any other possible solution is a Pareto optimum. The Pareto optimal set or Pareto optimal frontier is the set of all Pareto optima.

The second method often used to compare two solutions is a scalarizing function which combines all objectives into one using some kind of weighting. Common scalarizing functions include the simple weighted sum ($\sum_{k=1}^{n} \lambda_k f_k(x)$) or the weighted Chebyshev norm ($\max_{1\leq k\leq n}\{\lambda_k|f_k(x) - f_k(z)|\}$ where z is a reference point [17]. Such a scalarizing function is mostly only used to compare intermediate solutions during an optimization run; the final objective is still to create a number of non-dominated solutions.

3.3.2. Multi-objective evolutionary algorithms

A multi-objective evolutionary algorithm is an evolutionary algorithm which solves a multi-objective optimization problem. This type of algorithm is non-exact and feasible to approximate the optimal solution of the problem within a reasonable time (with respect to the problem size), handles many types of constraints well, and optimizes multiple objectives at once. Instead of specifying some kind of weighted composite objective function, MOEAs try to find Pareto-optimal solutions. Evolutionary algorithms are well-suited for this as they already use a pool of solutions which makes it easier to create an output set with multiple solutions. Besides finding Pareto-optimal solutions, another goal of a MOEA is to find solutions that are diverse and span the complete Pareto-optimal frontier.

Using such a MOEA gives the user of the algorithm an overview of possible solutions and the trade-offs that have to be made. Defining a weighting of the different objectives to transform a multi-objective problem into a single-objective one may be hard to do *a priori*. On the other hand, providing a wide range of non-dominated solutions among the Pareto-optimal frontier gives domain experts the chance to make an informed decision of the trade-offs they want to make. Therefore, providing an expert with multiple options is believed to be beneficial.

3.3.3. MOEAs in the literature

In this section, an overview of various MOEAs that have been proposed in literature over the years is provided, and they are classified according to their similarities and differences.

The MOEAs that have been proposed in literature can be broadly categorized into three categories [66], based on the way the fitness of solutions is measured and used in the evolutionary operators: domination-based, indicator-based and decomposition-based. These three categories will be discussed separately.

Domination-based MOEAs

Domination-based MOEAs are characterized by the fact that the assignment of fitness to solutions is based on the Pareto-dominance principle. To maintain a diverse set of solutions and prevent convergence to a single Pareto-optimal solution, an explicit diversity preservation scheme is necessary.

One of the first proposed MOEAs is the Nondominated Sorting Genetic Algorithm (NSGA) [65]. This algorithm is developed based on the idea of nondominated sorting suggested by Goldberg [42]. NSGA only varies from a regular genetic algorithm in the way selection works; the crossover and mutation operators remain as usual. Given a pool of solutions, the population is ranked based on domination: the first rank consists of all non-dominated solutions in the pool, the second rank consists of all non-dominated solutions in the pool after removing the first rank, etc. The solutions are then assigned a dummy fitness value based on their rank only. To preserve diversity, individuals have to share their fitness with neighbours that lie within a certain distance of themselves. The more solutions lie close together, the more fitness will have to be shared and the lower the overall fitness of these solutions will be.

Some years later, an improved version of the NSGA algorithm was proposed, called NSGA-II [30]. Three main issues were found in the NSGA algorithm: the nondominated sorting procedure was slow, no elitism was present, and there was a need for an explicit sharing parameter. NSGA-II addressed all of these issues by proposing a faster nondominated sorting procedure, introducing elitism in the algorithm, and using a different diversity preservation scheme through a *crowded comparison* operator based on density estimation. NSGA-II is probably the most well-known MOEA and has been used and extended in many other papers [25, 32, 62]

Another approach based on the dominance principle which was developed around the same time as NSGA is the Strength Pareto Evolutionary Algorithm (SPEA) [73]. This approach stores non-dominated solutions in an external archive and assigns fitness values to solutions based on Pareto dominance with respect to solutions in the archive only. To preserve diversity, a clustering operator is used.

Similarly to NSGA, some weaknesses in the SPEA algorithm were addressed in the next version of the algorithm: SPEA2 [74]. The identified weaknesses were: a flawed fitness assignment scheme due to considering the solutions in the archive only, not enough use of density estimation to guide the search, and a problem with archive truncation where outer solutions were often lost. SPEA2 addresses all these issues by also considering dominance among individuals in the pool, using a nearest-neighbour density estimation technique, and using a new archive truncation method.

Indicator-based MOEAs

Indicator-based approaches assign fitness to solutions by some kind of performance indicator assessing its contribution to the combined convergence and diversity of a MOEA.

The indicator-based evolutionary algorithm (IBEA) [75] proposes a general framework of including indicators in a MOEA and provides an example indicator based on the hypervolume concept [73]. The indicator is used in the fitness assignment of individuals.

SIBEA [19] improves the IBEA framework for problems with many objectives by proposing objective reduction techniques. PIBEA [18] uses a prospect-based indicator which measures the potential of each individual to reproduce offspring that dominate it and uses an adaptive mutation rate adjustment based on the convergence towards the Pareto front.

Decomposition-based MOEAs

MOEAs based on decomposition use scalarizing functions to transform the MOP into single-objective optimization sub-problems which are then solved simultaneously in a single run.

The most well-known decomposition-based MOEA is MOEA/D [70]. In this paper, three scalarizing functions are proposed: the weighted sum approach, Tchebycheff approach and boundary intersection approach. It should be noted that the boundary intersection approach is made for continuous variables and therefore not applicable to the given problem. Furthermore, in the proposed algorithm, each sub-problem is assigned a neighbourhood consisting of other sub-problems that lie close (in terms of weighting) to that sub-problem. Only the solutions in this neighbourhood are used to optimize a sub-problem. Non-dominated solutions which are found during the search are stored in an external population which represents the (intermediate) solution of the algorithm. The MOEA/D algorithm has been extended in many different studies afterwards [51, 71].

3.3.4. Constraint handling in MOEAs

Many (multi-objective) optimization problems do not only consider objective functions to be minimized but also constraints which a solution must satisfy. The way that these constraints and (in)feasibility should be handled in MOEAs is not trivial. Several constraint handling methods have been proposed which will be summarized in this section.

A simple but effective constraint handling method is proposed by Deb *et al.* in the NSGA-II paper [30]. They consider the following dominance scheme:

- Two feasible solutions (non-)dominate each other based on the usual definition of Pareto-dominance.
- Any feasible solution dominates any infeasible solution.
- An infeasible solution dominates another infeasible solution if it has a smaller overall constraint violation.

Another technique is proposed by Long [50]. They identify three key features of a good solution set for an MOP: closeness (to the Pareto-optimal frontier), diversity and feasibility. They propose a metric for each of these three features to be able to assign to each individual in an intermediate population of a "regular" MOEA. Selection is done by ranking the solutions by a method called the optimal sequence method [33], considering these three objectives. This selection method replaces the regular selection method of, for example, non-dominance or a weighted objective function, and takes into account the three features for a good solution set.

Vieira *et al.* [68] treat constraints as two added objectives to the original problem. The first is based on a penalty function and the second is based on the number of violated constraints. To ensure convergence to a feasible Pareto optimal front, the constrained individuals are eliminated during the elitist process. The method of using a penalty function is also indicated by Deb [31], although he does not consider the number of violations.

These methods are designed to handle *hard* constraints; constraints which may not be broken for a solution to be valid. Often there are also *soft* constraints present; constraints which may be broken but the violations need to be kept as small as possible. Handling soft constraints can be done using similar methods, for example by using a penalty function or treating the soft constraint violation as (one or more) added objective(s).

Finally, it must be kept in mind that for certain problems, finding *any* feasible solution may already be a hard problem, and constraint handling methods alone may not be enough. One would need to consider feasibility-preserving genetic operators or repair methods [50] to try and preserve feasibility throughout the iterations of the evolutionary algorithm.

3.3.5. Memetic MOEAs

To improve the performance and convergence speed of a MOEA, a local search module can be incorporated into the algorithm. Such a combination of a MOEA and a local search component is called a hybrid or memetic MOEA. In this section, an overview of various hybrid extensions of MOEAs is provided.

Sindhya *et al.* (2013) [62] propose a hybrid framework for evolutionary multi-objective optimization. In this framework, local search is used both during the evolutionary algorithm on certain selected solutions, as well as at the end of the evolutionary algorithm to locally improve each member of the final population. Implementations of this framework are provided considering NSGA-II [30], MOEA/D [70] and MOEA/D-DRA [71] as the base algorithms. In the given local search procedure, the selected individual is used as a reference point in order to determine the scalarizing function of the objectives, such that a single-objective local search method can be used. The exact local search method is not given; this can be any appropriate local search method for the problem.

Sindhya *et al.* (2011) [61] propose to use a saw-tooth type probability function to determine whether to perform local search throughout the evolutionary process. This idea is based on the assumption that the probability to perform local search should be increased when the solutions are closer to the Pareto optimal front. However, to achieve global optimality, the algorithm must constantly look for better solutions despite an apparent convergence to a front. This is contradictory with what a local search procedure does, so the probability to perform it should be lowered periodically.

Doush *et al.* [36] add Harmony Search [40] as a local search procedure using a hybrid framework and both a non-dominated sorting approach and a decomposition approach.

Cai *et al.* [23] propose an adaptive memetic framework for multi-objective combinatorial optimization problems. Their novelty lies especially in how solutions are selected to undergo local search. They propose two methods; one based on the solutions' convergence towards the Pareto front, and one which uses the information of an external archive. Simulated annealing is used as a local search procedure.

Michalak [53] propose a new local search method that utilizes the knowledge concerning promising search directions. This local search method is decomposition-based but it does not require the MOEA to be so as well. They test it with NSGA-II [30].

Ke *et al.* [44] propose the memetic algorithm based on decomposition (MOMAD). Their algorithm uses both Pareto Local Search, a multi-objective local search method, as well as a single-objective local search method. Any local search method can be used.

3.3.6. Other multi-objective metaheuristics

Although evolutionary algorithms are especially suitable for multi-objective optimization problems because they already use a pool of solutions, other metaheuristics such as local search methods have also been adapted to be able to deal with multi-objective problems. Some of these methods will be briefly discussed here.

Serafini [59] proposed the multi-objective simulated annealing (MOSA) algorithm. Like in the original single-objective algorithm, a single solution is considered and moved through the search space, but a subsidiary set is now used to store potential Pareto optimal solutions. The rule for accepting new solutions is now based on dominance rules.

An extension called archived multi-objective simulated annealing (AMOSA) is proposed by Bandyopadhyay *et al.* [16]. They introduce a novel concept of amount of dominance in order to determine the acceptance of a new solution. The amount of dominance of two solutions a and b is defined as $\Delta dom_{a,b} = \prod_{i=1, f_i(a)\neq f_i(b)}^{M} (|f_i(a) - f_i(b)|/R_i)$, with M being the number of objectives and R_i being the (estimated) range of objective i. For two objectives, this metric can be seen as the area of the rectangle with a and b as its corner points. Note that this metric does not measure whether two points actually dominate each other; this needs to be considered separately. An archive is used to store non-dominated solutions, and the acceptance probability of a new solution is based on the amount of dominance regarding the current point and the points in the archive.

Another simulated-annealing based algorithm is evolutionary multi-objective simulated annealing (EMOSA), proposed by Li & Landa-Silva [49]. This approach uses a decomposition of the problem using weight vectors similarly to MOEA/D [70], but then uses simulated annealing to optimize the subproblems. Furthermore, it uses adaptive weight vectors and ϵ-dominance [48].

Various other methods have been proposed, for example based on tabu search and variable neighbourhood search [17].

3.4. Existing platform, data and algorithm

Since a pilot study had already been performed by Macomi on this problem before the start of the work on this thesis, some infrastructure and an evolution strategy algorithm were already in place. This section describes the existing elements to give the reader a clearer picture of the work done during this thesis, as well as to provide baseline results to compare newly implemented algorithms and other improvements with.

3.4.1. Macomi platform

The full optimization pipeline from input to output has already been implemented by Macomi. This means the following elements are present:

- Code to read in the input data and process it into various models which are used in the optimization and output.
- Code to determine the quality of a solution; more on this in section 3.4.2
- An evolution strategy algorithm for optimization; more on this in section 3.4.2
- Code to create various output tables and visualizations from the result of an optimization run.
- A front-end which allows the user to insert or change input data, start new optimization runs, and view output tables and visualizations.

As it would be of little use to work on processing input data or working on output, since they already exist and are not the focus of this project, the existing infrastructure for this has been used and the work in this thesis focuses only on the optimization part. This means that the described algorithms and improvements have been implemented directly into the existing Macomi codebase. This codebase is written in C#, and therefore this is the language in which any code written for this thesis is written in.

3.4.2. Existing algorithm

For the pilot studies done by Macomi, the optimization algorithm that was used is an evolution strategy. The evolution strategy used by Macomi is a $(\mu + \lambda) - ES$. This means that a parent population of size μ is maintained. Each generation, λ offspring are generated, each from a single parent, and the parent set of the next generation is determined by a deterministic selection of the parents and the offspring. Pseudocode of the evolution strategy can be found in algorithm 3.2 The default parameters used are $\mu = 20, \lambda = 80$. A genetic algorithm was also implemented and tested on the problem by Macomi, but the results were found to be worse than the results provided by the evolution strategy, so the genetic algorithm was abandoned for now.

Generating starting individuals

To improve the convergence speed of the algorithm, the initial individuals in the parent population are not completely random but are created using a heuristic. Each project request is planned using this heuristic:

- If the length of the project request is 8 hours or less, plan it in a random night.
- If the length of the project request is 56 hours or less, plan it in a random weekend.
- If the length of the project request is less than a week, plan it in any holiday of at least a week that is not the summer holiday.
- Else, plan it in the summer holiday.

The heuristic takes the required time windows of the individual blocks into consideration when planning, but does not consider any other constraints or costs yet.

Mutations
Three mutations are used to create new offspring from a parent:

- Hour mutation: take a randomly selected project request and move it a random number of hours forward or backward. The maximum number of hours is given as a parameter and is set to 8 by default.
- Day mutation: take a randomly selected project request and move it to a random different day. The required time window constraint, if present, is taken into account. The starting *time* will stay the same, only the date changes.
- Loop mutation: perform another mutation multiple times after each other. The loop mutation is used with the day mutation as inner mutation, meaning up to 20 blocks will be moved in a single mutation.

Any time an individual is mutated, one of these mutations is randomly selected using pre-set weights.

Scoring function
Due to the size of the problem and the complexity of the objectives and constraints, calculating the fitness of an individual is not trivial. A scoring function has been implemented by Macomi which takes an individual as input and computes the costs and constraint violations which are used as fitness in the evolution strategy. This scoring function re-computes the full costs and constraints after each (small) change to an individual. The scoring function is relatively slow compared to the other parts of the algorithm and therefore causes the bulk of the runtime.

Baseline results
Macomi have used the evolution strategy to run various business case scenarios. The one that is best to use as a baseline is the one that uses the base constraint scenario as explained in section 2.2.3. On this scenario, the evolution strategy was run for 9000 generations on a machine with a six-core CPU @ 3.6GHz and 64GB of RAM, which took approximately three days of runtime. The results of the algorithm are as follows:

Total costs:	1000.0
Hard constraint violations:	32

These results will be used as a baseline for comparison with other algorithms and improvements in the remainder of this thesis.

4 Improvements to Macomi's existing solution method

Although the existing algorithm by Macomi, as described in section 3.4.2, already provides a great improvement upon the manually created schedule of ProRail, there are some possible areas of further improvement:

- Calculating the objectives and constraint violation takes a long time. Due to the size of the problem and the sophisticated objectives and constraints, an iteration of the evolution strategy takes a relatively long time. This means only a limited number of iterations can be run within a reasonable time, thereby limiting the search capabilities of the algorithm.
- The algorithm takes a long time to optimize to a low number of (hard) constraint violations. Due to the lack of mutations which focus specifically on solving hard constraint violations such as conflicts, these need to be solved by accident when a block is moved to a random new time window.
- Planning (non-conflicting) requests which affect the same sub-corridor(s) at the same time is a powerful tool to decrease costs. The availability costs will decrease because the sub-corridors only have to be taken out of service once, and the maintenance costs will decrease due to a reduction in security costs. However, the algorithm is in no way guided towards creating overlap in requests, and can only find this overlap by "accident" when moving blocks around randomly.
- Most of the costs and (most difficult) constraints are only affected by blocks with hinder. Approximately half of the blocks in the input do not create hinder and are therefore relatively easy to plan. However, the algorithm currently does not make any distinction between blocks with or without hinder in selecting which block to move, for example.

All of these issues are addressed in the sections below. Section 4.1 describes a new way to calculate the objectives and constraints which is a lot faster, especially for local search algorithms. In section 4.2, two new mutations are added to the evolution strategy which are specifically focused on solving conflict and dependency violations. Furthermore, a process called "constraint cooling" is explained, which is an attempt to avoid algorithms quickly converging to a local optimum. Section 4.3 describes the "bucket system", a set of mutations specifically designed to create overlap in requests and minimize availability costs. Finally, section 4.4 explains how separating requests with and without hinder may help the convergence speed and solution quality of the algorithms.

Afterwards, the improvements are tested in section 4.5. In section 4.6, a simulated annealing algorithm is tested on the problem. This is another local search based algorithm. This experiment is designed to see if another metaheuristic might be more suitable to solve the problem.

4.1. Incremental scoring function

The existing scoring function takes a solution to the problem and completely calculates the objective values and constraint violations from scratch. When a small change is made to the solution, all constraint violations are recomputed from scratch, and the costs are updated for each sub-corridor in which at least one project request has been moved. Due to the size of the problem and the complex objectives and constraints, computing the fitness of a solution takes a relatively long time. Because the proposed type of algorithms relies on evaluating many solutions and making a lot of small changes, this directly impacts the convergence speed and quality of the algorithms.

Therefore, an incremental scoring function was designed and implemented. Instead of recomputing the complete fitness after a small change, local changes only lead to local re-computations. Furthermore, the solution and its fitness are now integrated into one class instead of the scoring function being a separate function.

The incremental scoring relies on three basic operations:

- Add block: add a block to the schedule at a given time.
- Remove block: remove a block from the schedule. The block must have been added before.
- Move block: A concatenation of "remove block" and "add block".

While adding and removing blocks, various things are kept track of to allow the incremental scoring to work:

- The currently planned blocks and their planned date.
- The current constraint violations of each type, and the total costs of all types.
- The overlapping periods
  - On a sub-corridor level.
  - On a sub-corridor level with only requests that create hindrance.
  - On a corridor level.

  As briefly explained before, overlapping periods are periods in time in which at least one project request is planned at all times. Overlapping periods can exist on multiple levels. For example, an overlapping period on a certain sub-corridor only takes project requests which affect that sub-corridor into account. Section 4.1.3 explains them in further detail.

  Inside the overlapping periods,

  - The availability costs and maintenance costs caused by the overlapping period.
  - The maximum number of overlapping project requests in the overlapping period.
- The request pairs that are currently causing a conflict violation.
- The requests that are currently causing a dependency violation.
- The requests that are currently outside their required time window.
- The pairs of overlapping periods currently causing a mintime violation.
- The number of weekends currently affected by TVPs in each (sub)corridor.
- Many data structures containing (solution independent) preprocessed data allowing for faster computation.

Next, the add and remove block operations are explained to give an overview of how the incremental scoring works.

4.1.1. Add block operation

When a block is added, various operations have to be performed to update the objective values and constraint violations. These operations are described briefly below.

- The block and its planned period are stored.
- For each sub-corridor affected by the new block, the overlapping periods for that sub-corridor are updated with the new block. The same is done for all corridors affected by the block. While updating the overlapping periods, the associated costs are updated as well. The logic for overlapping periods is explained in more detail in section 4.1.3.
- When the new block creates hindrance, the conflicts and dependencies are updated. For each sub-corridor that potentially conflicts with the newly added block, it is checked which requests overlap with the newly added request. If any are found, a conflict violation is added and the conflicting pair is stored. A similar process is followed for the dependencies. To speed up the search for overlapping blocks, a binary search on the overlapping periods is used, after which all requests in an overlapping period are checked individually. The overlapping periods generally contain few (less than five) requests, which makes this a fast computation.
- It is checked whether the block is planned within its required period, and if not, a violation is added and the block is added to the list of blocks which violate the required window constraint.
- The prerequisites of the block are checked. Because of the incremental scoring, it is not defined whether the block itself or its possible prerequisite is added first. Therefore, when adding a block, both its own prerequisites are checked (if they were added before) and all blocks that have the new block as a prerequisite are checked as well (if they were added before). This way, whichever order the requests are added in, the prerequisite violations are always correctly kept track of.
- The constant costs of the block are added to the total constant costs of the solution.

4.1.2. Remove block operation

The remove block operation closely follows the add block operation but everything is reversed. The remove block operation consists of the following operations:

- The overlapping periods of the sub-corridors and corridors affected by the removed block are recomputed.
- If the block occurs in the lists of currently conflicting blocks, blocks currently causing dependency violations or blocks currently causing time window violations, the block is removed from those lists and the corresponding constraint violations are removed.
- If the block was causing a prerequisite violation, either as a prerequisite of another block or because one of its prerequisites was planned after it started, the violation is removed.
- The constant costs of the request are subtracted from the total constant costs of the solution.

4.1.3. Overlapping periods

The computation of overlapping periods and their properties is probably the most sophisticated part of the incremental scoring. First, the concept of overlapping period is briefly reiterated, and then, some of the logic regarding overlapping periods is explained.

Overlapping periods

Overlapping periods are periods in time in which at least one project request is planned at all times. Overlapping periods can exist on multiple levels. For example, an overlapping period on a sub-corridor only considers project requests affecting that specific sub-corridor, and can be used to compute availability costs. Similarly, an overlapping period can exist on the corridor level, where it can be used to compute possible violations of the corridor constraints. An overlapping period contains one or more project requests. A project request can never be part of more than one overlapping period (on the same level), and two overlapping periods can never overlap; in both cases these overlapping periods would actually be a single overlapping period.

In figure 4.1 some examples of how overlapping periods are formed from project requests are shown. The project requests are shown in white and the resulting overlapping periods are shown in blue. In case 1, there are two overlapping periods formed from the two groups of project requests. It becomes clear that an overlapping period can consist of multiple requests. In case 2, all project requests form a single overlapping period, because during that complete period in time at least one project request is planned at all times. In case 3 it is shown that an overlapping period can also consist of a single project request when it does not overlap with any other project requests at that level.

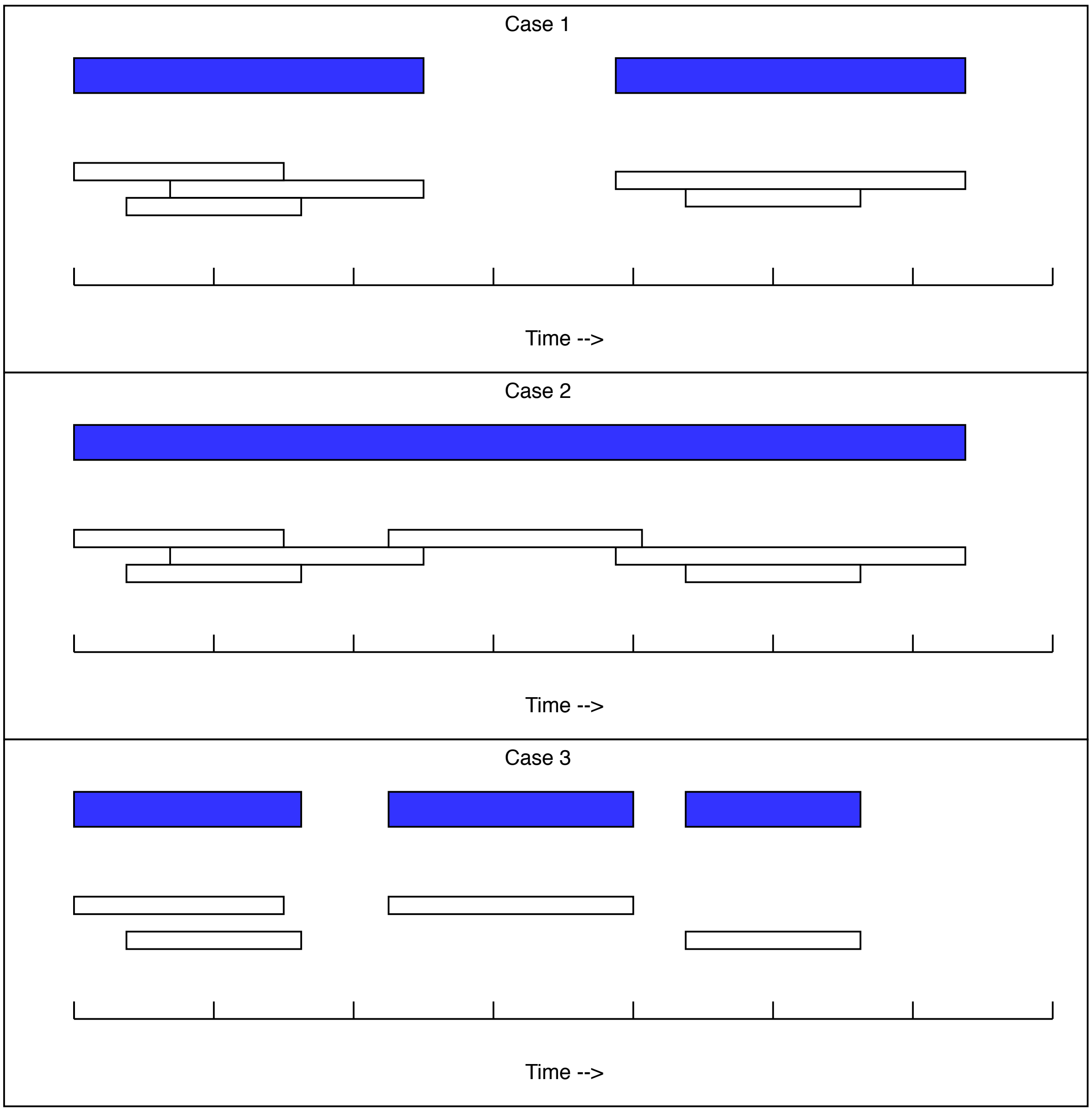


Figure 4.1: Examples of how overlapping periods and project requests relate. Project requests are shown in white, the resulting overlapping periods are shown in blue.

Adding and removing project requests

Whenever a block is added, all existing overlapping periods that overlap with the new block are found using binary search. When no existing overlapping periods are found, a new overlapping period with just the new block is created. When there are one or more overlapping periods overlapping with the new block, these are removed and a new overlapping period consisting of the new block and all blocks in the removed overlapping periods is added. Figure 4.2 shows some cases that can arise when adding a block and recomputing the overlapping periods. In each case, the overlapping periods before adding the block are shown in blue, the new block is shown in white, and the new overlapping periods are shown in red.

In case 1, there are two overlapping periods, and they both overlap (partly) with the new block. This situation results in a single overlapping period, since the new block now bridges the gap between the two existing overlapping periods. In case 2, the added block partly overlaps with a single overlapping period, meaning this overlapping period is extended, but the total number of overlapping periods stays the same. In case 3, the added block does not overlap with any existing overlapping periods, so a new overlapping period is created for the added block. Finally, in case 4, the added block fully overlaps multiple existing overlapping periods. These are all merged to a single overlapping period, similarly to case 1.

When a block is removed, the overlapping period it was part of needs to be checked. First, the existing overlapping period is removed. Then, new overlapping periods using the other requests in the old overlapping period (if any) are computed and added.

Costs

Whenever an overlapping period is created on the sub-corridor level, various relevant costs and constraint violations are computed. When adding an overlapping period on the sub-corridor level, the availability costs, alternative travel costs, security costs and personnel costs of the block are computed and added to the total. Furthermore, the maximum number of project requests planned simultaneously is computed and if this is larger than the allowed maximum a violation is added.

When removing an overlapping period on the sub-corridor level, the costs and constraints are again updated. The availability and maintenance costs are simply removed from the total. It is also checked whether the removed overlapping period was (one of the) periods causing the current violation for the maximum number of simultaneously planned project requests, if it exists. If it was, this maximum is recomputed from the remaining overlapping periods on the sub-corridor. This is a quick computation, because the maxima are stored in each overlapping period and do not need to be recomputed.

Corridor constraints

When an overlapping period on the corridor level is added, the TVP constraints are checked. Remember that these constraints operate on TVPs which are overlapping periods of at least 24 hours long. They are explained in further detail in section 2.2.2.

For the maximum TVPs and the maximum affected weekends, the number of TVPs and weekends of the new overlapping period is added to the total for the corridor. If necessary, the constraint violation is updated. Checking the constraint for the minimum time between two TVPs is slightly more complex. When a new overlapping period is added, it's (potential) left and right neighbours are found. These are then checked for a mintime violation with the newly added block. Furthermore, if there are two neighbours present, it is possible that there was an existing mintime violation between those two neighbours which needs to be removed.

If a violation for the maximum number of project requests at one location existed for the overlapping period this is removed as well. For the corridor level, the number of TVPs and affected weekends are subtracted and the potential violation is updated or removed. For the mintime violations, any existing violations with the block's (old) neighbours are removed, and if two neighbours were present, a potential violation between them may be added.

4.1.4. Locality and speedup

It becomes clear that the new scoring method is local; the overlapping periods and their costs and constraints are only updated locally, conflict and dependency violations are only checked where necessary and are not fully recomputed each time, etc.

A small experiment has been run to determine the speedup of the new scoring function. For both the old and the new scoring, the time it takes to do one hundred mutations has been measured. For the old scoring function, the score is computed after each mutation. For the new scoring function, this is done implicitly. The mutations that were used randomly select and move a single block each time. The runs have been executed twenty times to minimize the effect of randomness and background processes.

The results of the experiment are summarized in table 4.1. All shown values are in milliseconds.

Looking at the mean, the speedup factor is approximately 20.7. Since for this problem, calculating the fitnesses takes more than 90% of the runtime for the various algorithms, this means the algorithms can now run the same number of iterations in about 1/20th of the time. Thus, the same results can be achieved in a

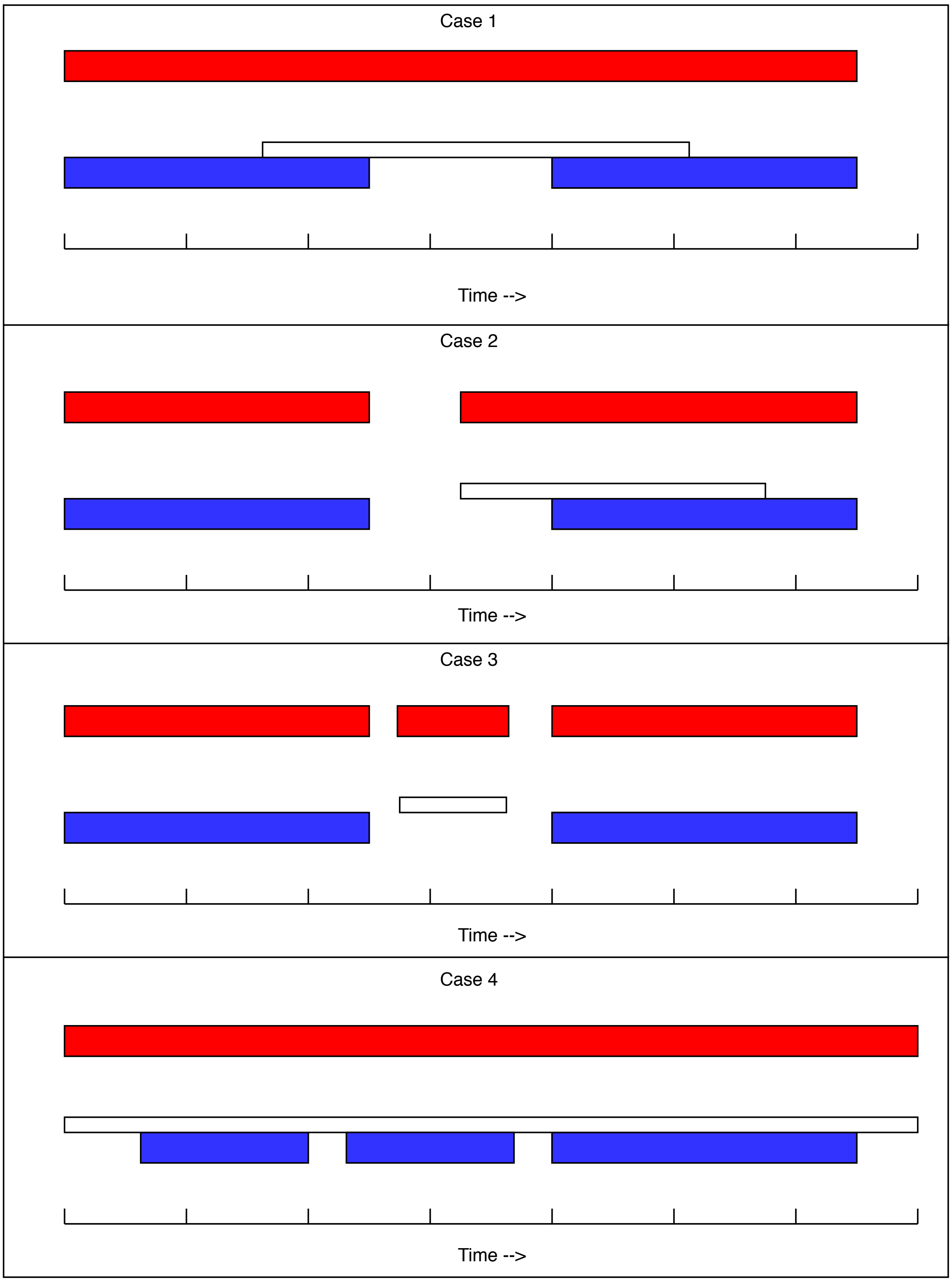


Figure 4.2: Various cases when recomputing overlapping periods after adding a block. The old overlapping periods are shown in blue. The new block is shown in white. The new overlapping periods are shown in red. Case 1: two overlapping periods are merged by the added block. Case 2: A single overlapping period is extended by the added block. Case 3: A new overlapping period is created by the new block. Case 4: Multiple overlapping periods are merged by the added block.

Algorithm 4.1 Pseudocode for handling overlapping periods when adding and removing requests

```
function ADDREQUEST(r)
    Find existing overlapping periods overlapping with r
    Remove all these overlapping periods
    Add an overlapping period with r and all requests of the previously removed overlapping periods

function REMOVEREQUEST(r)
    Remove the overlapping period containing r
    if The removed overlapping period contained any other requests then
        Recompute and add all overlapping periods with the remaining requests

function ADDOVERLAPPINGPERIOD(p)
    if p is on the sub-corridor level then
        Compute the availability costs and maintenance costs of p and add them to the total.
        Compute the maximum number of overlapping requests, and update the violation if needed.
        if p is at least 24 hours then
            Compute the number of affected weekends, add them to the total, and update the violation if needed
    else if p is on the corridor level then
        if p is at least 24 hours then
            Compute the number of TVPs, add them to the total, and update the violation if needed.
            Compute the number of affected weekends, add them to the total, and update the violation if needed
            Check for potential mintime violations with neighbours, and remove a possibly existing mintime viola-
            tions between the neighbours of p

function REMOVEOVERLAPPINGPERIOD(p)
    if p is on the sub-corridor level then
        Remove the availability and maintenance costs of p from the total
        if p was (one of the) blocks causing the current violation for max number of requests then
            Recompute the violation from the remaining overlapping periods
        if p is at least 24 hours then
            Remove the affected weekends of p and update the violation if needed
    else if p is on the corridor level then
        if p is at least 24 hours then
            Remove the number of TVPs of p from the total and update the violation if needed
            Remove the number of affected weekends of p from the total and update the violation if needed
            Remove potential mintime violations with the (old) neighbours of p, and check for a mintime violation
            between the old neighbours of p
```

	Old scoring	New scoring
Min	30802	1313
Mean	31191	1507
Median	31169	1514
Max	31889	1739

Table 4.1: Time comparison for 100 simple mutations for old and new scoring function, 20 repetitions. All values are in milliseconds.

way shorter time frame or more iterations can be run within the same time which gives the algorithm(s) more chances to find better solutions.

It is noted that the (relative) spread is higher for the new scoring function. This is the case because, for the new scoring function, the time it takes to move a block is dependent on the characteristics of that block, such as the length and the number of locations. However, for a large number of mutations, the fact that the spread is larger does not matter much as the average time will even out anyway. Neither scoring function has any large outliers on either side.

4.2. Hard constraint mutations and constraint cooling

The mutations used to create new individuals in the evolution strategy move blocks around seemingly random and do not take any specific constraints into account. This causes the convergence towards a solution with a low number of hard constraint violations to be quite slow. To provide a fast convergence towards an (almost) feasible solution, the following changes have been made:

- In the heuristic to create the initial starting individuals, dependency constraints were already taken into account. If possible, a block will be planned in a time window where it would not cause any dependency violations. This decreased the number of initial constraint violations by approximately one hundred on average. Furthermore, a bug was found in the existing code which caused all blocks to be planned outside their required window instead of within. Fixing this bug brought the initial number of hard constraints down from approximately 4000 to 500 on average.
- A mutation to fix conflict violations was implemented. This mutation randomly selects any block that is currently causing a conflict with another block. Then, it computes possible time windows where the selected blocks would not cause any conflicts. If any of such time windows are available, one is randomly selected and the block is moved there. If none are available, the block is moved to a random other time. If possible, the starting *time* of the block is kept the same (such that a block planned at night will stay planned at night, for example).
- A mutation to fix dependency violations was implemented. This mutation works similar to the mutation that fixes conflicts but instead selects a block that is currently causing a dependency conflict.

Initial tests using these improvements showed good results. The constraint violations went down much faster than before. As expected, this also caused the costs to increase faster in the early generations of the algorithm. Unfortunately, it was observed that, after the bulk of the hard constraint violations were optimized, the improvements in costs in the generations after that were not satisfactory. This is likely due to the fact that the hard constraint violations are leading over the costs when comparing two solutions, and it is found to be hard to decrease costs without causing (temporary) extra hard constraint violations. Essentially, the algorithm converges to a local optimum where it does not find a way to escape. To avoid getting stuck in such a local optimum, a system called constraint cooling was added. In this system, solutions are allowed to break a large number of constraints at the start of the algorithm. Any solution with fewer constraint violations than this number of allowed constraints is considered feasible and is only compared on costs. During the algorithm, the number of allowed constraints is slowly decreased, similar to the cooling process in simulated annealing, hence the name. At two-thirds of the total number of iterations, the number of allowed constraints will be equal to a predefined lower limit and is kept there for the remainder of the iterations. This allows the algorithm to have some time to improve upon a solution with little to no constraint violations.

For the constraint cooling, the initial and final allowed constraints are hyper-parameters which need to be set beforehand. The cooling schedule used is exponential: $T_k = T_0 \cdot \alpha^k$. As the number of iterations is known beforehand, the α is calculated in such a way that the final number of allowed constraints is reached at two-thirds of the number of iterations. Figure 4.3 shows an example of the constraint cooling schedule for a run with 15000 iterations, where 600 constraint violations are allowed at the start and 8 are allowed at the end. These values for the number of allowed constraints at the start and end of the algorithm are also used as default parameters for the tests performed in section 4.5. The value of 600 was chosen to be slightly above the (average) number of constraint violations in a starting solution. These usually have around 450 to 500 constraint violations. The final number of allowed constraints was set to 8 rather then 0, because there are a number of required time window constraints which are unsolvable as explained in section 2.4.1. Furthermore, some small experiments showed that the final solutions usually have some conflict or dependency violations remaining which are not directly solvable. Therefore, since it is impossible to get a feasible solution anyway, setting the allowed violations at the end to 8 causes a somewhat shallower slope in the constraint cooling schedule which leads to the algorithm having a bit more time to optimize the costs while also being forced to optimize hard constraints.

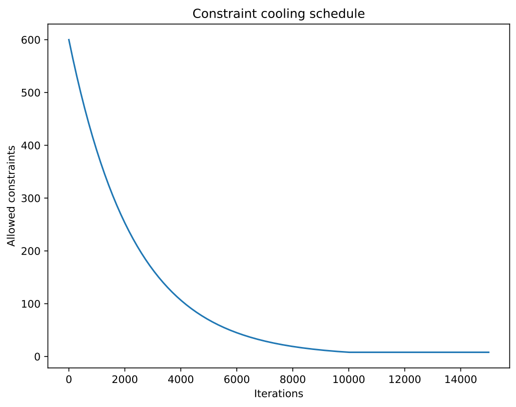


Figure 4.3: Constraint cooling example. Initial allowed constraints is 600, final allowed constraints is 8, number of iterations is 15000.

4.3. Bucket system

One of the best ways to reduce both maintenance and availability costs is by planning requests with one or more overlapping sub-corridors at the same time. By performing multiple maintenance jobs on a sub-corridor at the same time, the time for the track to be taken out of service decreases and the availability costs decrease with it. Furthermore, the security costs will also decrease because of the fact that the workplace only has to be secured once.

Of course, not all project requests which affect one or more of the same sub-corridors can just be planned at the same time because their other locations might be in conflict. Furthermore, there is a maximum number of project requests that can be planned at the same time on one location, and a combination matrix constraint which specifies work types that can and cannot be planned at the same time. However, in the standard scenario, the combination matrix constraint is turned off, meaning the logic for determining when two blocks (with overlapping sub-corridors) can be planned at the same time is quite easy now; only the conflicts and the maximum number of projects need to be checked.

The current evolution strategy algorithm does not guide the search process towards creating this overlap. Simple shift mutations which randomly move around blocks are used, and although overlap can still be found, it will likely take a long time because the algorithm would have to get "lucky" to move a block to an optimal overlapping location. Guiding the algorithm towards creating valid overlap would be beneficial both for the convergence speed and the solution quality of the algorithm.

To do this, the so-called bucket system has been implemented. It will be explained in the sections below.

4.3.1. What is a bucket?

A bucket is a collection of project requests that are clustered together and planned in an overlapping manner with the goal of creating overlap at the sub-corridor level in order to decrease availability costs. In certain mutations used in the metaheuristic algorithms, all requests in a cluster will be moved simultaneously to keep this overlap intact.

All requests in a bucket must be

- Mutually non-conflicting: any pair of requests in the bucket should not create any conflicts when planned in an overlapping manner. This can also be extended, for example with the combination matrix constraint.
- Creating hindrance: Requests without hinder don't cause availability costs and are therefore (relatively) unuseful; they would only save on security costs but this is a very small percentage of the total costs.
- Have sufficient sub-corridor overlap; this will be explained further in section 4.3.2.

For now, the maximum size of a bucket will be set to the maximum number of requests that can simultaneously be executed at one sub-corridor per the relevant constraint.

4.3.2. When can a request be added to a bucket?

The first two conditions for adding a request to a bucket are clear: there must be no conflicts with any existing requests in the bucket, and the request must create hindrance. If these conditions hold, the sub-corridors of the new request and the existing request are considered. At this point, a distinction is made between two situations: creating a new bucket with two requests and adding a request to a bucket with two or more requests already in it.

The idea behind these two cases is similar though: a good bucket has

- Many overlapping sub-corridors, preferably in long requests.
- Few sub-corridors that occur in only a single request in the bucket.

The reason for the first objective is obvious; it is exactly the reason the bucket system is implemented. The reasoning behind the second objective is that a larger number of total distinct sub-corridors in a bucket will decrease the options where the bucket can be planned because it will create more conflicts with other requests and/or buckets. Therefore, sub-corridors that do not contribute to saving costs should be kept to a minimum. The rules for accepting or rejecting a request to a bucket, as explained below, try to optimize these objectives.

Creating a new bucket with two requests

The rule for accepting or rejecting a new bucket with two requests is as follows:

1. Let the "impact" of a request be the number of sub-corridors multiplied by the length of the request, and let request **a** be the request with the highest impact, and request **b** the other request.
2. Accept when the number of overlapping sub-corridors between **a** and **b** is larger than or equal to the number of sub-corridors in **b** that do not overlap with **a**, reject otherwise.

The idea behind this rule is as follows: having more overlapping sub-corridors than non-overlapping sub-corridors is good. However, simply counting the total number of overlapping and non-overlapping sub-corridors does not work well in certain cases. The idea behind using the impact of a request is the fact that, when the request with most impact has a relatively high number of sub-corridors that do not overlap with the other request, this is not so much of a problem because the request would have to be planned anyway, and planning another request with similar but fewer sub-corridors at the same time would have little downside. When the requests with least impact has many non-overlapping sub-corridors, planning it at the same time would likely give relatively few benefits while adding a number of unique sub-corridors to the bucket, making it harder to plan.

Below, two examples are given to illustrate the rule

Example 4.1. Accepted bucket

- Sub-corridors: a, b, c, d, e; Length: 50
- Sub-corridors: a; Length: 150

Example 4.2. Rejected bucket

- Sub-corridors: a, b, c, d, e; Length: 20

- Sub-corridors: a; Length: 150

The first situation would be accepted because the first request is seen as the one with the most impact, whereas the second situation would be rejected because the second request is seen as the one with most impact. Here, the trade-off between overlap length and the number of non-useful locations becomes clear; the overlap of 50 in the first situation is deemed enough of an advantage to allow multiple non-overlapping sub-corridors in the bucket, whereas the overlap length of 20 in the second situation is not enough to warrant the number of non-overlapping sub-corridors.

Adding a request to an existing bucket with two or more requests
Algorithm 4.2 shows the rule which is used to determine whether a request can be added to an existing bucket:

Algorithm 4.2 Rule used to determine whether a request can be added to an existing bucket

```
newBlock ← the new request to be added to the bucket
requests ← the existing requests in the bucket
maxOverlap ← 0, addedLength ← 0
for all sub-corridor sc affected by newBlock do
    selectedRequests ← all project requests in 'requests' that affect sub-corridor sc
    if selectedRequests is empty then
        addedLength ← addedLength + length of newBlock in hours
    else
        maxOverlap ← maxOverlap + min(length of newBlock in hours, max(length of blocks in selectedRequests))
if maxOverlap ≥ addedLength then
    accept
else
    reject
```

The idea behind the rule is as follows: when more overlap can be created (in theory) than non-useful sub-corridors are added to the bucket, the request should be accepted and otherwise it should be rejected. The potential overlap is computed by finding the maximum overlap that can be achieved with existing requests on each sub-corridor. The non-useful sub-corridors are expressed by the number of non-useful sub-corridors times the length of the new block.

An example to illustrate is given below:

Example 4.3. The existing requests are:

- Sub-corridors: a, b; Length: 100
- Sub-corridors: a, b, c; Length: 10

The new request is:
Sub-corridors: b, c, d, e, f; Length: 50

The non-overlapping sub-corridors are d, e and f, so the $addedLocations$ metric is $3 * 50 = 150$. For location b the maximum overlap is 50 because the new block can fully overlap with the first existing block, meaning 50 hours of overlap could potentially be achieved. For location c, the maximum overlap is 10, because there can only be 10 overlapping hours (with the second existing request).

Therefore, the total maximum overlap is 60 and the block will be rejected; adding three extra non-overlapping sub-corridors is not worth it when compared to the potential overlap.

4.3.3. Planning the requests within a bucket

There are multiple ways in which requests within a bucket could be planned relative to each other. The most trivial way is to let each request in a bucket have the same starting time. This always leads to maximum overlap, and does not require any extra computation and thus will not slow down the algorithm. A disadvantage of this method is that the first part of a bucket may become very "dense" in the sense that it will cause conflicts with other requests relatively fast, and the maximum number of project requests may be exceeded more quickly.

Another way is to try and plan smaller blocks within a bucket behind each other such that their total length does not exceed the length of the largest block in the bucket. This will require more computation to find the optimal way to plan the blocks and may lead to suboptimal overlap and thus ERM save. It does mean that the requests are more spread out so it may be easier to plan simultaneously with other requests. However, this is not necessarily true because the second part of a bucket might now cause more conflicts.

To be able to achieve maximum overlap and speed, the first method of having each request in a bucket start at the same time has been chosen. It is believed that these advantages outweigh the possible disadvantages of this method.

4.3.4. Implementation into the algorithm

To make sure the search space is not limited, buckets will not be precomputed and kept static for the remainder of the algorithm. Instead, four mutations have been added which create, remove and move buckets:

- Add bucket mutation: Find two requests that are currently not in a bucket and are eligible to be created into a bucket. Create a bucket from those two requests. The bucket is planned at the original plan moment of one of the two requests with equal chance. Candidate pairs to create a bucket from can be precomputed.
- Expand bucket mutation: Take a random request that is not in a bucket yet and try to find an existing bucket that would accept the request. If no such bucket is found, retry with a different request up to a user-specified number of times. If multiple eligible buckets are found, choose one at random.
- Shrink bucket mutation: Take a random bucket and move one request out of the bucket and move it to a random other starting time. If the bucket only had two requests, this will lead to the bucket being destroyed.
- Move bucket mutations: Move a bucket (i.e. all requests in it) to a random other starting time.

Existing mutations have been adapted to only move around blocks that are not in a bucket. An alternative would have been to also allow buckets of size one and consider each singly planned block as a bucket as well. However, it was decided to use the term buckets only for two or more requests. This provides a bit more customizability with mutations for either buckets or singly planned blocks and the frequency which with they are used.

4.4. Separation of requests with hinder

A closer inspection of the problem and its constraints led to the conclusion that it would be beneficial to split up the problem into two parts: the requests with hindrance and those without. In the data from 2019, this division is approximately 50/50. The idea behind this split is that the requests with hindrance make up most of the difficulty of the problem. Availability costs are only caused by these requests. The most difficult constraints are only enforced for blocks that create hindrance. On the contrary, blocks without hindrance are only relevant for

- Maintenance costs; a large part of the maintenance costs cannot be optimized at all, and the personnel costs for a single request are for 99% completely independent of the planned time of other requests.
- Prerequisite constraints: this constraint is quite easily solved, and does not occur often in practice.
- Required time window constraints: this constraint limits the possibilities for a block to be planned but is in itself not a difficult constraint to solve.
- Maximum project requests at one location constraint: similar to the required time window constraint, this constraint is relatively easy to solve.

Given this information, it was concluded that it may be beneficial to separate the requests without hindrance and run the (metaheuristic) algorithm on the requests with hindrance only. This will definitely increase the convergence speed of the algorithm because fewer mutations are "wasted" on blocks without hindrance. It may also improve the final solutions of the algorithm; the results of a test are provided and discussed in section 4.5.

The requests without hindrance will be added afterwards using a relatively simple heuristic. The heuristic makes use of the information that approximately half of the requests without hindrance in the 2019 data have zero personnel and security costs. In practice, this is due to missing data about those project requests. It is noted that blocks without hindrance and without any (optimizable) maintenance costs can pretty much be planned anywhere as long as it doesn't violate the required time window and the maximum project requests at one location constraints. Therefore, these requests are treated separately from the requests that have no hindrance but do have personnel costs and/or security costs. If in the future, the missing costs are added, the impact on the heuristic and its performance will be minimal, because the requests will simply get planned by the other heuristic. Therefore, this method can be considered as not overfitted on the 2019 data specifically.

The requests with no relevant costs are planned by finding the starting time which leads to the request having the most hours of overlap with existing requests in all of its sub-corridors combined, while not creating any (extra) hard constraint violations and not exceeding the maximum number of overlapping project requests at any of its sub-corridors. This starting time is found by trying each possibility. If no suitable starting time can be found, the request is planned so that the added constraint violation is minimal. However, this almost never happens in practice.

The requests *with* personnel and/or security costs are planned using the following heuristic:

1. For each possible starting time the block could be planned on
   (a) Check if the block does not cause hard constraint violations. If it does, do not consider this starting time as a possibility and continue to the next one.
   (b) Compute the personnel costs the request would cause when planned at the specific starting time. Here, the rule for planning two short requests directly after each other is not taken into account, but this has very little effect on the total costs in practice anyway.
   (c) Compute the security costs to be either zero if there is already a block planned at the time, or the full security costs specified in the request otherwise.
   (d) If the sum of these costs is the smallest found so far, store the starting time as the current best starting time.
2. Plan the block at the current best starting time. If no suitable starting time was found, plan it at a moment where it causes minimal (extra) constraint violation and lowest possible costs for that constraint violation increase.

The requests with personnel and/or security costs will be planned first, followed by the remainder of the requests. The whole process takes approximately five to ten minutes; a negligible time compared to the runtime of the evolution strategy. Even though each possible starting time is tried for each block, the incremental scoring function in combination with the fact that many costs and constraints don't need to be computed for blocks without hindrance makes that this part of the algorithm is still much faster than whatever metaheuristic will be used to plan the blocks with hindrance.

4.5. Testing the improvements

To test the impact of the improvements two tests have been performed. One uses the constraint cooling and bucket system, the other uses the constraint cooling, bucket system and separation of requests with and without hindrance. Furthermore, the "regular" evolution strategy is rerun with the new scoring function. The new scoring function merely provides a speed up which was separately tested and discussed in section 4.1.4. All runs in this comparison are done using the new scoring function. All three experiments are run for 20000 iterations and use the same parameters apart from the difference in implemented improvements.

The results are summarized in table 4.2.

Algorithm	Total costs	Hard constraint violations
ES baseline	1005.6	9
ES + constraint cooling & buckets	1002.7	11
ES + constraint cooling, buckets & separation on hindrance	982.3	11

Table 4.2: Results for various improvements to the evolution strategy algorithm

From these results, it becomes clear that the proposed improvements have indeed improved the solution quality. When adding the constraint cooling and the bucket system, the improvement in costs is minimal and the number of constraint violations is slightly worse. However, when also separating on hindrance the costs decrease significantly. Although the hard constraint violations are still slightly higher compared to the baseline, the substantial cost decrease is believed to outweigh this increase in constraint violations. From these results, it is not 100% clear what would happen if only the separation on hindrance was added as an improvement. However, it is believed that adding this separation is also beneficial for the bucket system, as less "useless" blocks are present and on average, the bucket mutations will be effective more often.

Figure 4.4 shows the costs and constraint violations over time for the evolution strategy baseline and the evolution strategy with constraint cooling and buckets. The run with separation on hindrance is not included, because the intermediate costs throughout the evolution strategy are not comparable due to the fact that only around half the blocks are present at the time.

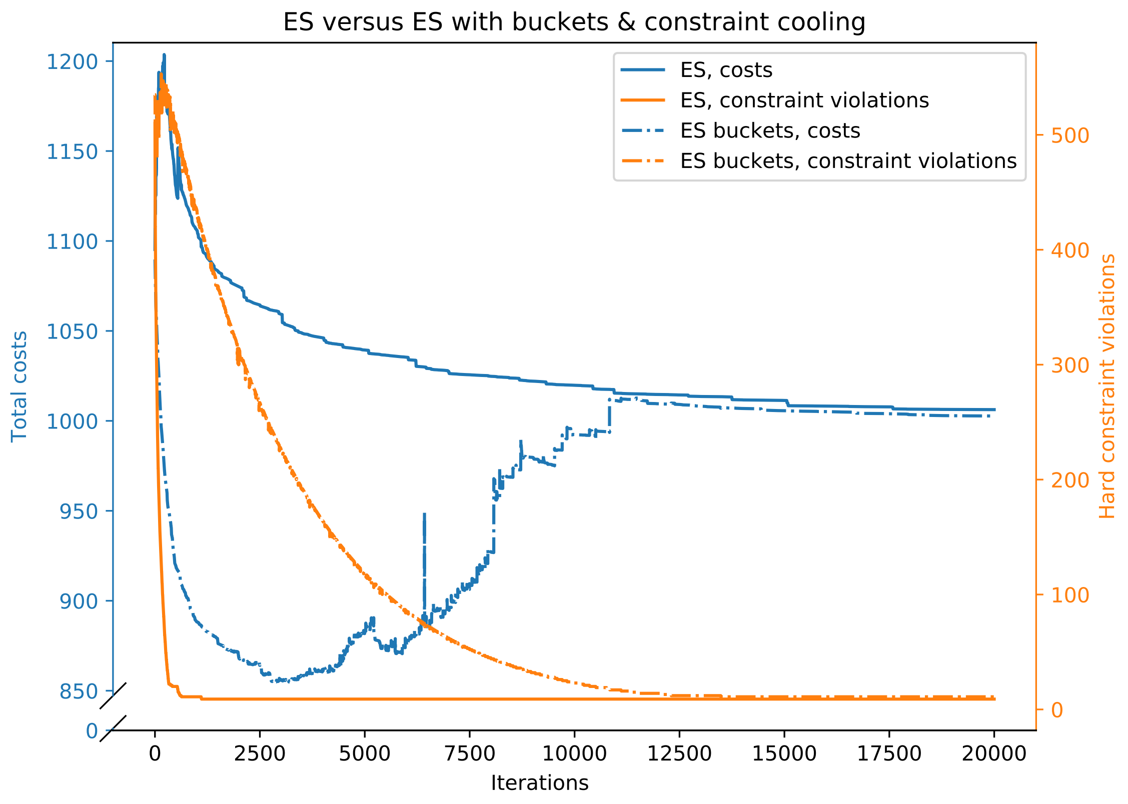


Figure 4.4: Comparison of evolution strategy without constraint cooling and buckets, and evolution strategy with constraint cooling and buckets

The behaviour of both algorithms is as expected. Allowing many constraint violations at the start decreases the costs massively. This also gives a good indication of the trade off between hard constraint violations and costs. From a certain point, the number of allowed violations becomes small enough to make the costs rise again. The costs end up below the baseline costs which was the desired result.

4.6. Simulated annealing

The improvements made to the evolution strategy were effective, and better results were obtained. However, at this point, it is unclear whether an evolution strategy is the correct metaheuristic to use at all. Many metaheuristics exist and it is hard to say anything about the performance of any of them without an actual implementation and experiment.

To get some more insight into the relative performance of the evolution strategy, another local search based algorithm was implemented: Simulated Annealing [47]. The reason for another local search algorithm, rather than a more different type of metaheuristic such as a genetic algorithm is twofold. First, the new scoring function works best for small, local changes to a solution. This is in line with a local search algorithm. A genetic algorithm would take significantly longer per generation and therefore its search capabilities would be limited. Second, in the pilot study of Macomi, an initial implementation of a genetic algorithm was also created, and its results were substantially worse than that of the evolution strategy which is a local search algorithm. The specific choice for simulated annealing was made because it has been used successfully in previous studies on rail maintenance scheduling problems in both single objective [20] and multi-objective forms [15, 56].

Simulated annealing is a local search based metaheuristic, in which a single solution is moved through the solution space by repeatedly applying small changes, also called *mutations*, to the solution. Similar to a hill climbing algorithm, a solution is accepted whenever it is an improvement of the previous solution. However, to avoid converging to a local optimum, a solution worse than its predecessor is accepted with a certain probability, which is based on the difference in solution quality as well as the current *temperature*, which is a parameter controlling the trade-off between exploitation and exploration. Over time, the temperature is decreased and the chance to accept a worse solution with it.

Algorithm 4.3 shows the pseudocode of a simulated annealing algorithm.

Algorithm 4.3 Pseudocode of a simulated annealing algorithm

```
initialize initial solution $s$, starting temperature $T$, cooling rate $\alpha$
currentBestSolution $\leftarrow s$
repeat
    $s_{new} \leftarrow$ mutate($s$)
    if $s_{new} > s$ then
        $s \leftarrow s_{new}$
        if $s_{new} >$ currentBestSolution then
            currentBestSolution $\leftarrow s_{new}$
    else
        $\delta \leftarrow s - s_{new}$
        $s \leftarrow s_{new}$ with probability $e^{-\delta/T}$
    $T \leftarrow \alpha * T$
until A stopping criterion is satisfied
return currentBestSolution
```

4.6.1. Dealing with constraints

The standard implementation of simulated annealing does not handle hard constraint violations well. To calculate the acceptance probability of a solution, the difference in fitness values is used. However, one cannot simply subtract two cost values from each other if the number of hard constraint violations differs. Furthermore, the scale of the cost differences is completely different from the scale in differences of hard constraint violations. Therefore, the rules for accepting or rejecting a solution have been slightly adapted. This new strategy is inspired partly by Singh *et al.* [63]. Three new parameters are required. The first is the *constraint delta factor*, which is a problem specific factor, scaling the difference in constraint violations to be approximately of the same order as the general difference in fitness. Because the acceptance probability is calculated using absolute differences, using the same temperature for differences of a completely different order of magnitude would not yield the expected results. Second, an *initial* and *final acceptance probability* need to be defined. These acceptance probabilities are used in the case where the current solution is feasible and the new solution is infeasible. The probability to accept in this case starts at the provided initial probability and is decreased over time towards the final acceptance probability.

Algorithm 4.4 shows the new acceptance strategy.

When both the current and new solutions are feasible, the regular acceptance strategy is used. When both are infeasible, a regular acceptance strategy is also used, but if applicable the difference in hard constraints is scaled according to the corresponding parameter. When the current solution is feasible and the new solution is infeasible, it is accepted with a certain probability based on the initial and final probabilities as well as the iteration number. Finally, a feasible solution is always accepted over an infeasible solution.

Algorithm 4.4 Simulated annealing acceptance strategy

```
Given a current solution $s$, a new solution $s_{new}$, current temperature $T$, current iteration $i$, total number of iterations $N$, initial acceptance probability $P_i$, final acceptance probability $P_f$ and constraint delta factor $C$.
if both $s$ and $s_{new}$ are feasible then
    Use the regular simulated annealing acceptance strategy
else if both $s$ and $s_{new}$ are infeasible then
    $\delta \leftarrow$ difference in constraint violations between $s$ and $s_{new}$
    if $\delta < 0$ then
        $s \leftarrow s_{new}$
    else if $\delta = 0$ then
        Use the regular simulated annealing acceptance strategy
    else
        $s \leftarrow s_{new}$ with probability $e^{-C\delta/T}$
else if $s$ is feasible, $s_{new}$ is infeasible then
    $s \leftarrow s_{new}$ with probability $P_i \cdot \left(\frac{P_f}{P_i}\right)^{i/N}$
else if $s$ is infeasible, $s_{new}$ is feasible then
    $s \leftarrow s_{new}$
return $s$
```

4.6.2. Operators and parameter values

To keep the comparison between the simulated annealing and evolution strategy algorithms as fair as possible, the mutation(s) used for simulated annealing are the exact same as for the evolution strategy. This means the new mutations for decreasing constraint violations, as well as the bucket system are included. A constraint cooling schedule is also included. The new scoring function is used to evaluate the solutions throughout the algorithm.

Table 4.3 shows the values of the other parameters which were used in the simulated annealing.

Parameter	Description	Value
Repetitions	Number of iterations at a single temperature level	100
α	Cooling factor	0.993
$Tmax$	Starting temperature	$0.00001/\alpha^{30000}$
P_i	Initial acceptance probability for changing from a feasible to an infeasible solution	0.9
P_f	Final acceptance probability for changing from a feasible to an infeasible solution	0.05
C	Constraint delta factor	1.2
restartAfterNoImprovement	Number of iterations without improving the solution after which to restart at the current best solution	100
constraintViolationsAllowedStart	Number of constraint violations allowed at the start of the algorithm	1200
numberConstraintViolationsAllowedEnd	Number of constraint violations allowed at the end of the algorithm	8

Table 4.3: Parameters used for simulated annealing

The number of iterations being able to complete in 24 hours was 3 million. However, with this many iterations it is hard to determine good values for the starting temperature, cooling factor, and final temperature. Therefore, the number of repetitions on each temperature level was set to 100. This way, the number of cooling iterations decreases to 30000. The cooling factor and starting temperature were set in such a way that the final temperature is 10^{-5}, and that the starting temperature is less than the maximum value of a *double* type in C#, which is $1.798 \cdot 10^{308}$. This method was adopted from a paper by Singh *et al.* [63]. In the same paper, the authors mention, regarding the initial and final acceptance probability for changing from a feasible to an infeasible solution, that "we start from a high value of P_i to accept infeasible solutions initially, and reduce it exponentially to a low value P_f in the end." The chosen values for these two parameters reflect these thoughts. The constraint delta factor was set by educated guess and reflects the cost increase which is believed to be similarly bad as an increase of the hard constraint violations by one. The number of iterations after which to restart was set to a relatively low value after seeing that, given the high starting temperature, the "current" solution could drift quite far away from the current best solution in terms of costs and constraints, after which the algorithm would take a long time to get back on a similar level of solution quality. The number of constraint violations at the start of the algorithm was set to a high value to allow for much exploration in terms of cost decrease at the start of the algorithm. Finally, the number of constraint violations at the end of the algorithm was set to the same value as used for the evolution strategy in order to get comparable results.

4.6.3. Results

The simulated annealing was run five times for 24 hours. The improved evolution strategy was also run five times for 24 hours. The results can be seen in table 4.4.

Algorithm and run	Total costs	Constraint violations
Evolution strategy, run 1	980.1	10
Evolution strategy, run 2	993.5	10
Evolution strategy, run 3	988.5	10
Evolution strategy, run 4	978.7	10
Evolution strategy, run 5	981.4	10
Evolution strategy, average	984.4	10
Simulated annealing, run 1	1000.2	10
Simulated annealing, run 2	1034.1	8
Simulated annealing, run 3	1066.9	9
Simulated annealing, run 4	1036.1	8
Simulated annealing, run 5	1059.3	9
Simulated annealing, average	1039.7	8.8

Table 4.4: Results of evolution strategy and simulated annealing

It can be seen that the results of the simulated annealing algorithm have slightly lower constraint violations on average compared to the results of the evolution strategy. However, the costs are much higher, and because there is only a small difference in constraint violations, the results of the evolution strategy are considered of better quality than those of the simulated annealing.

Although the number of runs is relatively small, there is a clear difference in costs between the two algorithms, and with some confidence it can be concluded that the evolution strategy algorithm is better suited to solve this problem than a simulated annealing algorithm. Therefore, the evolution strategy will be used to perform parameter analysis to try and improve the performance of the algorithm further. More on this can be found in chapter 6.

4.6.4. Concluding remarks

In this chapter, a number of improvements to the existing evolution strategy developed by Macomi were proposed. It was shown that these improvements were successful and better solutions were found. A simulated annealing algorithm was also tested on the problem, but its results were significantly worse than those of the evolution strategy. In the next chapter, the problem will be viewed from a slightly different perspective, namely that of a multi-objective problem.

5 Multi-objective algorithms

After successfully improving the existing solution method in the previous chapter, the problem will now be tackled from a different perspective. As explained in section 2.2.1, the given problem contains multiple objectives and there is a trade-off present between them. Therefore, two multi-objective metaheuristics have been implemented and tested on the problem. These multi-objective algorithms provide a range of Pareto-optimal solutions and allow the user to be able to see and choose which trade-offs he wants to make.

5.1. NSGA-II

The first algorithm which was implemented is NSGA-II [30], the successor of the Nondominated Sorting Genetic Algorithm [65]. This algorithm is a domination-based MOEA, meaning it ranks individuals based on Pareto-dominance. The algorithm resembles a regular genetic algorithm, and only varies in the way selection works; the crossover and mutation operators remain as usual. Selection is done by generating *non-dominated fronts*, where the first front contains all non-dominated solutions in the pool, the second front contains all non-dominated solutions after removing the solutions in the first front, etc. The fitness of a solution is based on the front it falls in only. To preserve diversity, a *crowded comparison* operator is used. This operator estimates the density of solutions surrounding a particular solution in the population. When the nondomination rank is equal, the solution with larger crowding distance is preferred over one with a smaller crowding distance.

NSGA-II was chosen as a first multi-objective algorithm for two main reasons. First, it has been used successfully in multiple other studies on rail maintenance scheduling [22]][56]. This leads to the belief that this algorithm could be suitable to use for this problem as well. Second, NSGA-II has shown to be effective in many different problem areas, and works well over a range of different problems. Because no multi-objective algorithm has been used on this problem yet, using a generic and robust algorithm is favoured over using a more specific algorithm at this point, because an insight in the effectiveness of *any* multi-objective algorithm is important, before possibly refining or changing which exact algorithm is being used. This can be done in a next step, if the results are promising.

Pseudocode of the main loop of the NSGA-II algorithm is given in algorithm 5.1. For the full specification, the reader is referred to the original paper [30].

5.1.1. Dealing with constraints

In the original NSGA-II algorithm there is no specification on how to handle soft or hard constraints in the problem. The problem at hand contains both, and below it is described how these are handled.

Hard constraints are incorporated in the same way as suggested by the authors of NSGA-II. Their proposed constraint handling approach is to change the definition of dominance to incorporate feasibility [30]:

Definition 5.1. A solution i is said to constrained-dominate a solution j if any of the following conditions is true:

- Solution i is feasible and solution j is not.
- Solutions i and j are both infeasible, but solution i has a smaller overall constraint violation.

Algorithm 5.1 Pseudocode for the NSGA-II algorithm. Adapted from [30].

```
$R_t = P_t \cup Q_t$    ▷ Combine parent and offspring population
$F$ = non-dominated-sort($R_t$)    ▷ $F = (F_1, F_2, \ldots)$, all nondominated fronts of $R_t$.
$P_{t+1} = \emptyset$ and $i = 1$
repeat
    crowding-distance-assignment($F_i$)    ▷ Calculate crowding distance in $F_i$
    $P_{t+1} = P_{t+1} \cup F_i$    ▷ Include $i$th nondominated front in the parent population
    $i = i + 1$    ▷ Check the next front for inclusion
until $|P_{t+1}| + |F_i| \leq N$    ▷ Until the parent population is filled
Sort($F_i$)    ▷ Sort the first front that does not fit fully in the parent population on crowding distance
$P_{t+1} = P_{t+1} \cup F_i[1 : (N - |P_{t+1}|)]$    ▷ Choose the first $(N - |P_{t+1}|)$ elements of $F_i$.
$Q_{t+1}$ = make-new-pop($P_{t+1}$)    ▷ Use selection, crossover and mutation to create a new population $Q_{t+1}$
$t = t + 1$    ▷ Increment the generation counter
```

- Solutions i and j are feasible and solution i (Pareto-)dominates solution j.

To deal with soft constraints, a simple penalty function approach has been adopted. Half of the total soft constraint penalties are added to both objectives as a penalty. Therefore, they do not affect the trade-off between the objectives, but decreasing the soft constraints will lead to a better solution.

5.1.2. Used parameters and operators

Crossover
As the crossover operator, a simple time-based crossover was implemented, which randomly chooses a time within the global plan window and takes everything planned before that time from parent 1 and everything planned after that time from parent 2 (and vice versa for the second offspring). This will sometimes create a situation where one offspring contains one project request twice and the other offspring does not contain it at all, which is solved by randomly moving one of the occurrences to the other offspring. Pseudocode is provided in algorithm 5.2.

Algorithm 5.2 Pseudocode for the time-based crossover

```
procedure TIME-BASED CROSSOVER(parent1, parent2)
    $t \leftarrow$ a random time within the global plan window
    for all $x \in blocks$ do
        if x is planned in the same half in both parents then
            For offspring 1, take x from parent 1 if x is planned in the first half, from parent 2 otherwise
            For offspring 2, take x from parent 2 if x is planned in the first half, from parent 1 otherwise
        else
            Select x randomly from parent 1 and 2 for offspring 1.
            Select the other parent for offspring 2.
    return both offspring
```

Mutations
The used mutations are the same as used in the evolution strategy. This includes the existing mutations as described in section 3.4.2, as well as the added mutations for fixing conflict and dependency violations from section 4.2, and the mutations from the bucket system which was explained in section 4.3.

Other parameters
Table 5.1 describes the other parameters in the NSGA-II algorithm and the values that were used.

The crossover rate was set to the same value as used in the paper describing NSGA-II [30]. The mutation rate was set somewhat arbitrarily, but is set significantly lower as is usually the case for a genetic algorithm. It was hard to determine a good population size *a priori* so it was set to be equal to the size of a parent and offspring population of the evolution strategy combined. Finally, the tournament size percentage was set to two, because binary tournament selection is used often in genetic algorithms, and using a larger tournament size may lead to too high of a selection pressure and therefore too little exploration of the search space.

Parameter	Description	Used value
CrossoverRate	The chance of using crossover to generate two new offspring	0.9
MutationRate	The chance to mutate a newly created individual	0.1
PopulationSize	The size of the parent population	100
TournamentSizePercentage	The number of individuals used in tournament selection to select a parent	2

Table 5.1: Parameters used in NSGA-II

5.1.3. Results

NSGA-II was run for 2500 iterations which took approximately 48 hours. Unfortunately, due to the fact that the crossover operator requires many blocks to be moved around, the incremental new scoring function which is optimized for small changes is actually slower than the old scoring function for this algorithm. Therefore, the old scoring function has been used in this algorithm. The results can be found in table 5.2.

Name	Costs	Constraints	Maintenance	Availability	Soft constraints
Evolution strategy	1026.8	7	n/a	n/a	n/a
NSGA-II, result 1	1148.6	13	628.4	466.4	53.8
NSGA-II, result 2	1148.2	13	628.0	466.7	53.5
NSGA-II, result 3	1148.4	13	628.0	466.9	53.5
NSGA-II, result 4	1148.5	13	628.0	467.1	53.5
NSGA-II, result 5	1149.3	13	627.8	468.0	53.6
NSGA-II, result 6	1149.5	13	627.8	468.1	53.6
NSGA-II, result 7	1149.6	13	627.8	468.3	53.6
NSGA-II, result 8	1149.6	13	627.8	468.3	53.6
NSGA-II, mean	1149.0	13	627.9	467.5	53.6

Table 5.2: Results of the evolution strategy and NSGA-II. For the evolution strategy, the cost breakdown is not available as an intermediate result was used to create a fair comparison in terms of runtime

.

Figure 5.1 shows the constraint violations and costs over time for both algorithms.
Some notes regarding fair comparison must be made:

- Due to the bugfix described before in section 4.2, the number of hard constraint violations in the starting individuals of NSGA-II was significantly lower than in those used for the evolution strategy.
- The evolution strategy is single-objective while NSGA-II is multi-objective. The lines shown in figure 5.1 for NSGA-II are the best solution in terms of constraint violations and costs as if treated as a single-objective problem. This does not account for the fact that an added benefit of a multi-objective algorithm is the fact that multiple solutions are provided.

Taking these points into account, the two algorithms can be compared. The changes made to decrease hard constraint violations faster were effective; within an hour of runtime, the number of constraint violations is decreased under 50. This also causes, as expected, a fast increase in costs in the early stages of the NSGA-II algorithm. Afterwards, there is a downward slope in the total costs over time. However, this slope is much less steep than that of the evolution strategy, meaning the evolution strategy is more effective in optimizing the costs. This causes the evolution strategy to overtake NSGA-II in terms of costs just before the 20-hour mark and results in significantly fewer costs in the final solution of the evolution strategy when compared to the solution(s) of NSGA-II. In conclusion, the NSGA-II algorithm combined with the changes focused on hard constraint fixes provides a fast way to a (near) feasible solution, but over a longer period of time, the evolution strategy is better at optimizing costs and provides a better final solution.

Figure 5.2 shows the solutions provided by the NSGA-II algorithm and the trade-off between the two objectives.

Multiple conclusions can be drawn from this figure. First, it is noticed that the range of generated solutions is small, and only a small portion of the Pareto front is covered. This means the algorithm has some

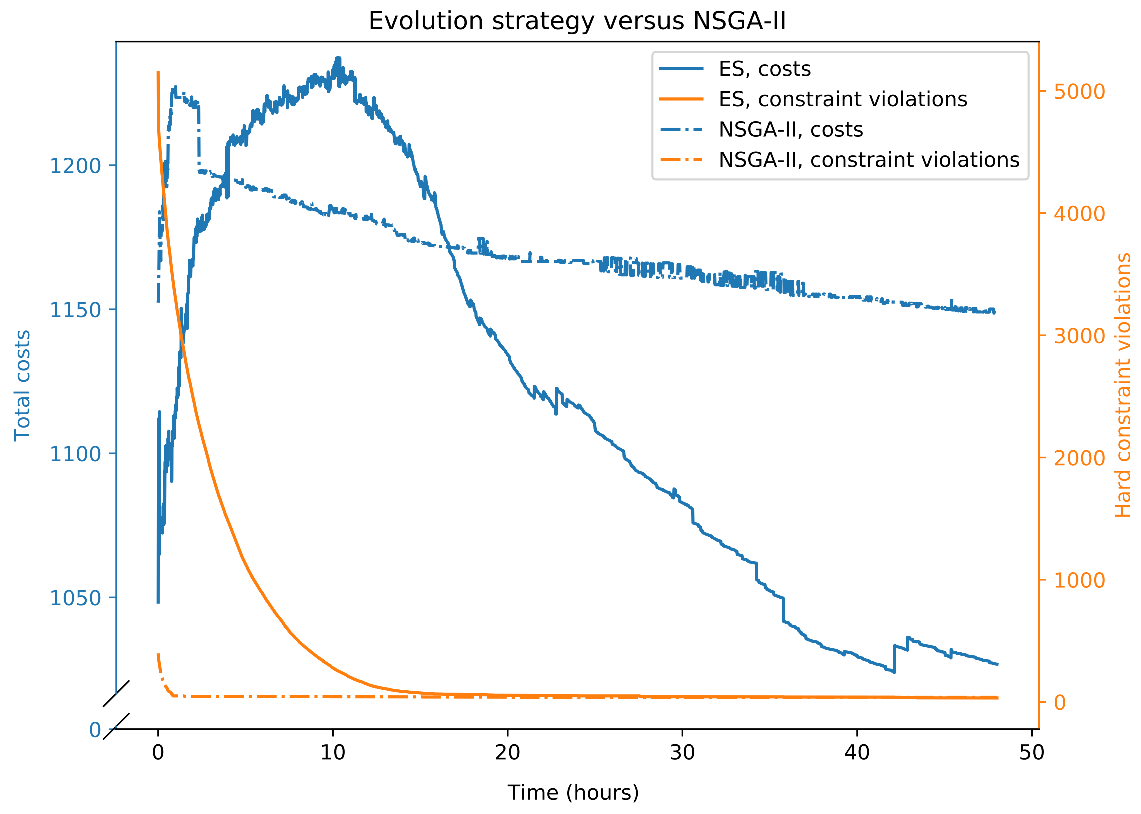


Figure 5.1: NSGA-II versus Evolution strategy

trouble finding diverse solutions. Second, from these results it looks like a decrease in availability costs causes a lesser increase in maintenance costs, and that therefore, when a low total cost is the objective, focusing on availability costs is the way to go. However, because only a small portion of the Pareto front is covered this conclusion is just preliminary; it is entirely possible for another part of the front to not have this property. Further tests with (other) multiobjective algorithms will have to give more insight.

Improving the diversity of solutions

A possible cause for the low diversity in solutions that was observed in the NSGA-II results is the fact that the heuristic for creating starting individuals as explained in section 3.4.2 is primarily focused on decreasing availability costs by planning everything during non-busy periods such as a weekend or holiday. Although the individuals themselves will be quite different, they will all lie primarily on the part of the Pareto front where availability costs are relatively low and maintenance costs are relatively high.

To try and increase the diversity of the final solutions, a second heuristic to create starting individuals was created. It works similarly to the existing heuristic in the sense that it uses time windows of increasing length to plan requests in. However, this heuristic focuses on optimizing maintenance costs, specifically personnel costs since this is the most optimizable part of the maintenance costs. The heuristic to plan a block is as follows:

- If the request takes less than 13 hours, plan it during the daytime on a weekday.
- If the request takes less than 109 hours, plan it during a period from Monday 07:00 until Friday 20:00.
- Else, plan the request anywhere so that it does not touch a holiday (if possible).

The idea behind the heuristic is to try and avoid expensive personnel periods which are nights, weekends and holidays.

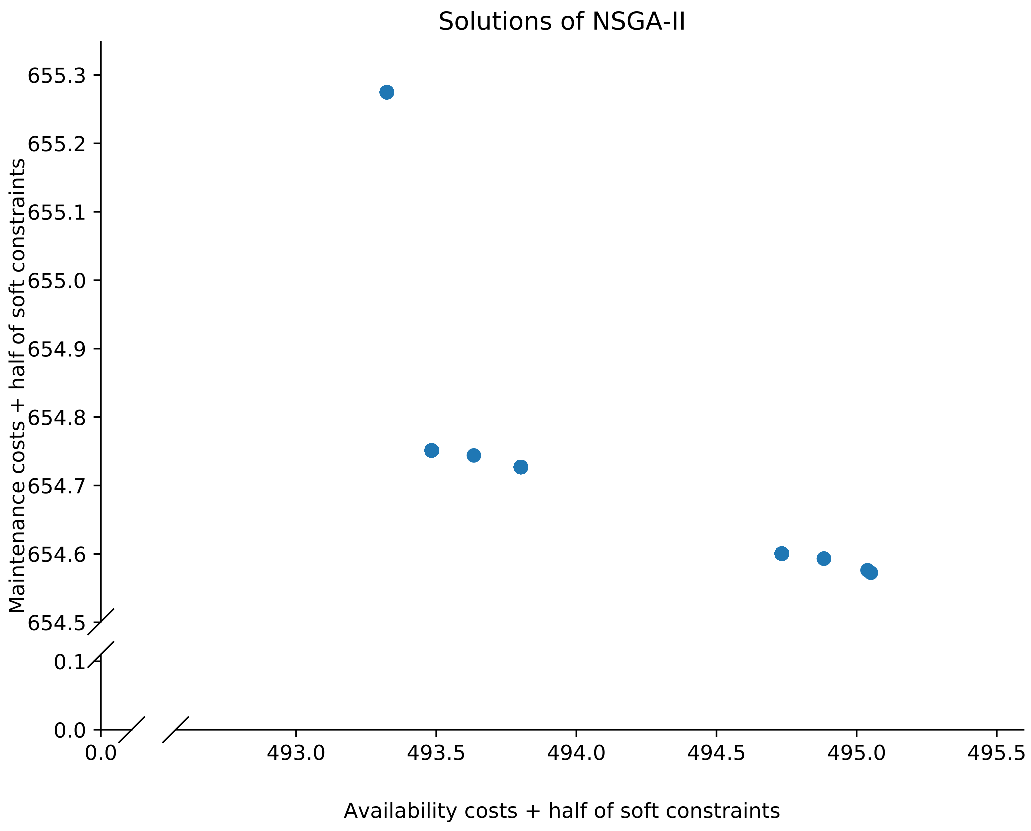
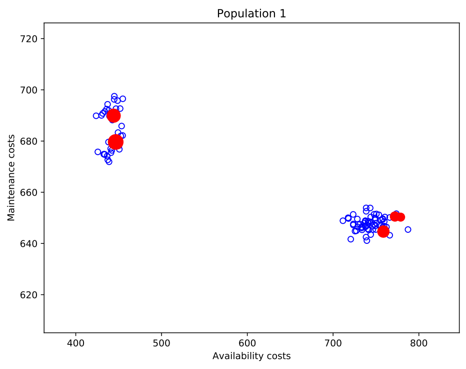


Figure 5.2: Solutions of the NSGA-II algorithm

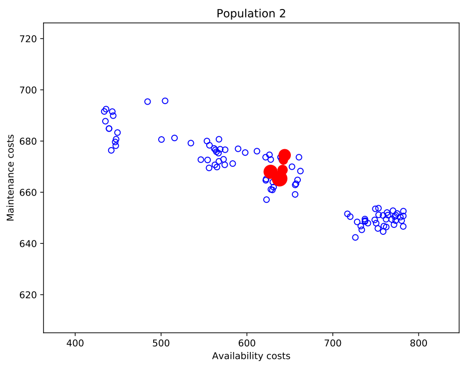
A second run of the NSGA-II algorithm was performed, where half of the starting individuals were created using the original (availability-focused) heuristic, and the other half were created using this new maintenance-focused heuristic. Unfortunately, it was observed that after a few iterations, the diversity in the population was already small again.

Figure 5.3 shows how the population evolves over time and provides a possible explanation for the observed behaviour. In this figure, the blue dots are individuals and the red dots are the five individuals with the smallest hard constraint violation. In figure 5.3a the initial population is shown. Here the separation between the two initialization heuristics is clearly visible; there are two distinct groups, one of which has low maintenance costs and high availability costs, and vice versa. Figure 5.3b shows the population at the second generation. Here, it is clear that crossover has happened; there are multiple individuals that fall in between the two groups. The population is very diverse here which was the result aimed for when creating the second starting individual heuristic. However, what is key here is that all individuals with the lowest constraint violations fall in the middle and are not spanning the whole diversity of the population. The likely cause for that is the fact that if you cross over an individual that is optimized for availability costs, and therefore has everything planned at a night, weekend or holiday, with an individual that is optimized for maintenance costs, and therefore has everything planned during the day and during weekdays, there will be fewer conflict violations on average, as the requests are more spread out. Since the conflict violations make up a large portion of the hard constraints, this will lead to the observed behaviour that these individuals that fall in the middle of the availability-maintenance spectrum have fewer hard constraint violations on average.

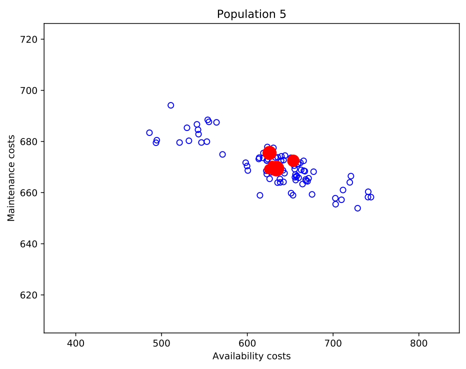
Because hard constraint violations are leading over costs when talking about dominance, these individuals then get chosen more often as parents for crossover and mutation, and this causes the whole population to converge towards the middle. This can be seen in figure 5.3c, where the diversity of the population is already decreasing, and figure 5.3d where the diversity in the population has completely disappeared, after only 15 generations. Afterwards, the algorithm does not escape from this local optimum and the diversity of the population stays unsatisfactory.



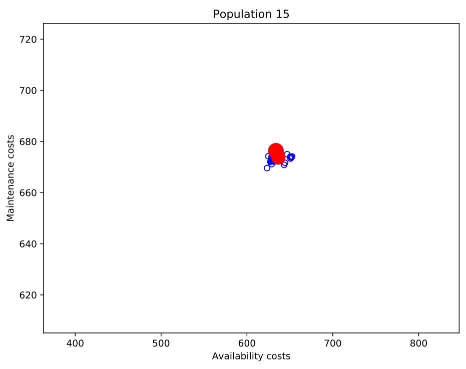
(a) Population at the first generation



(b) Population at the second generation



(c) Population at the fifth generation



(d) Population at the fifteenth generation

Figure 5.3: Various populations during an NSGA-II optimization run. The blue dots indicate individuals, the red dots are the five individuals with the smallest hard constraint violation

5.2. AMOSA

The results provided by the NSGA-II algorithm are clearly unsatisfactory. Both the costs are too high and the diversity is too low. Therefore, a different type of algorithm needs to be tested.

From earlier pilot tests by Macomi, it was concluded that an evolution strategy, which is a type of local search algorithm making small changes to an individual each time, performed better than a genetic algorithm, where large changes are being made to individuals by crossover. NSGA-II is also based on a genetic algorithm, which may partly explain the bad performance. A similar observation was made by Peralta *et al.* [56]; they found that when using a non-random initialization, a local search based algorithm outperformed a genetic algorithm. Since the heuristic for creating starting individuals is clearly not random, this behaviour may also hold for the problem at hand.

Taking this into account, the next algorithm that was implemented is a multi-objective local-search based algorithm, based on simulated annealing: archived multi-objective simulated annealing (AMOSA)[16].

AMOSA is based on simulated annealing [47], which was explained in section 4.6. AMOSA has many similarities with SA; there is also a single solution moved through space by small mutations. There is also a decreasing temperature and a probability to accept worse solutions dependent on the temperature. However, because solutions can be mutually non-dominating, it is not trivial to simply compare the current or best solution found so far with the new solution. AMOSA handles this by keeping track of an *archive*, in which all best solutions found so far are kept. This means that all solutions in the archive will always be non-dominating. Furthermore, there are various cases regarding the domination status of the current point, the new point, and the archive. To keep the size of the archive under control, a hard and soft archive limit (HL, SL) are provided. Once the archive grows to size SL, the archive is clustered to HL clusters and a representative solution is chosen from each cluster. At the end of the algorithm, a maximum of HL solutions will be returned. Algorithm 5.3 shows pseudocode for the AMOSA algorithm [16].

Algorithm 5.3 Pseudocode for the AMOSA algorithm, adopted from [16]

```
Set $Tmax, Tmin, HL, SL, iter, \alpha, temp = Tmax$
Initialize the Archive
current-pt = random(Archive)
while $temp > Tmin$ do
    for $i = 0; i < iter; i++$ do
        new-pt=mutate(current-pt)
        if current-pt dominates new-pt then
            $k \leftarrow$ number of points in the Archive that dominate new-pt ($k \geq 1$).
            $\Delta dom_{avg} \leftarrow \frac{(\sum_{i=1}^{k} \Delta dom_{i,new-pt}) + \Delta dom_{current-pt,new-pt}}{k+1}$
            set new-pt as current-pt with probability $\frac{1}{1+\exp(\Delta dom_{avg} \cdot temp)}$
        else if current-pt and new-pt are non-dominating then
            if new-pt is dominated by $k \geq 1$ points in the Archive then
                $\Delta dom_{avg} \leftarrow \frac{(\sum_{i=1}^{k} \Delta dom_{i,new-pt})}{k}$
                Set new-pt as current-pt with probability $\frac{1}{1+\exp(\Delta dom_{avg} \cdot temp)}$
            else if new-pt is non-dominating w.r.t. all the points in the Archive then
                Set new-pt as current-pt and add new-pt to the Archive
                if Archive size > SL then
                    Cluster Archive to HL clusters
            else if new-pt dominates $k \geq 1$ points in the archive then
                Set new-pt as current-pt and add it to the archive.
                Remove all dominated points from the Archive
        else if new-pt dominates current-pt then
            if new-pt is dominated by $k \geq 1$ points in the Archive then
                $\Delta dom_{min} \leftarrow$ minimum of the difference in domination amounts between the new-pt and the
                $k$ points.
                $prob \leftarrow \frac{1}{1+\exp(-\Delta dom_{min})}$
                Set point of the archive which corresponds to $\Delta dom_{min}$ as current-pt with probability $prob$,
                else set new-pt as current-pt
            else if new-pt is non-dominating w.r.t. the points in the Archive then
                Set new-pt as current-pt and add it to the archive.
                if current-pt is in the archive then
                    Remove it from the archive
                else if Archive size > SL then
                    Cluster Archive to HL clusters
            else if new-pt dominates $k \geq 1$ points in the archive then
                Set new-pt as current-pt and add it to the archive.
                Remove all dominated points from the archive
    $temp \leftarrow \alpha \cdot temp$
if Archive size > HL then
    Cluster archive to HL clusters
return all solutions in the archive
```
.

5.2.1. Dealing with constraints

Neither the simulated annealing nor AMOSA algorithms are designed with a (hard) constrained problem in mind. It is not trivial to determine how constraint violations should appear in the acceptance probability functions. For this problem, a constraint handling strategy from Singh *et al.* [63] has been adopted. The algorithm proposed by these authors is an extension of AMOSA in the sense that the procedure for AMOSA is used when both the current and new points are in the feasible space. When either one or both points are infeasible, other rules regarding acceptance to the current point and archive are used. For this procedure, two extra parameters are present: the so-called initial probability (P_i) and final probability (P_f). These indicate the probability that an infeasible solution is accepted as the current point when the current point is a feasible solution. This probability starts at P_i and is decreased to P_f over the number of iterations. The expanded acceptance rules are summarized in algorithm 5.4. This specific constraint handling strategy was chosen because it is specifically designed to extend AMOSA and is therefore expected to be effective for use in this situation.

Algorithm 5.4 Pseudocode for the constraint handling strategy, adopted from [63].

```
given new-pt, current-pt and archive
if new-pt and current-pt are feasible then
    Follow the AMOSA procedure
else if current-pt is feasible, new-pt is infeasible then
    $prob \leftarrow P_i \cdot \left(\frac{P_f}{P_i}\right)^{i/N}$
    Set current-pt = new-pt with probability $prob$
else if current-pt is infeasible, new-pt is feasible then
    Set current-pt = new-pt
    if new-pt is non-dominated w.r.t. Archive then
        Add new-pt to the archive
        if Archive size > SL then
            Cluster archive to size HL
    else if new-pt dominates points in the Archive then
        Add new-pt to the archive
        Remove all dominated points from the archive
else if new-pt and current-pt are infeasible then
    if Constraint violation of new-pt ≤ constraint violation of current-pt then
        Set current-pt = new-pt
    else
        Set current-pt = new-pt with probability $\exp\left(\frac{-\Delta\text{constraint violation}}{T}\right)$
return current-pt
```

Soft constraints are dealt with in the same manner as before: they are added as a penalty to both objectives equally.

5.2.2. Used parameters and operators

Mutations

The neighbourhood of a solution used in the AMOSA algorithm is implicitly defined by a number of mutations that may be used to perturb a solution. The mutations used are the same as in the NSGA-II algorithm.

Other parameters

Table 5.3 shows the description and used parameter values for a run of 28000 iterations. The parameters were mostly set similarly as for the single-objective simulated annealing algorithm, explained in section 4.6.2. The archive hard limit was set to ten, so that ten solutions are returned by the algorithm in the end, assuming that many non-dominated solutions can be found. The value for the soft archive limit should not be set too high as it increases the computational complexity of the algorithm, but also not too close to the hard limit to avoid clustering too often. It is believed that the value of 30 strikes a good balance between these two objectives.

Parameter	Description	Used value
CoolingIterations	The number of iterations after which the temperature is decreased	28000
TempIterations	The number of iterations at each temperature	100
Tmax	The initial temperature	$1.64 \cdot 10^{117}$
HL	The size of the archive after clustering	10
SL	The size of the archive at which we cluster	30
α	The factor with which to decrease the temperature with	0.99
P_i	The initial probability to accept an infeasible solution to replace a feasible one	0.9
P_f	The final probability to accept an infeasible solution to replace a feasible one	0.05
RestartAfterConsecutiveRejections	The number of consecutive rejections after which to restart from a random point in the archive	100

Table 5.3: Parameters used in AMOSA

5.2.3. Results

The AMOSA algorithm was run for 24 hours. Table 5.4 shows the resulting solutions. The resulting population can be seen in figure 5.4, together with the previously shown population of NSGA-II and the single solution of the improved evolution strategy.

Name	Costs	Constraints	Maintenance	Availability	Soft constraints
Evolution strategy	987.4	10	606.7	342.7	38.0
AMOSA, result 1	1047.8	10	601.7	420.2	26.0
AMOSA, result 2	1046.9	10	601.7	419.1	26.1
AMOSA, result 3	1046.8	10	601.7	418.8	26.3
AMOSA, result 4	1046.9	10	601.7	419.0	26.1
AMOSA, result 5	1046.0	10	602.2	417.6	26.1
AMOSA, result 6	1045.7	10	602.2	417.4	26.1
AMOSA, result 7	1045.0	10	602.2	416.7	26.1
AMOSA, result 8	1042.4	10	602.3	412.0	28.1
AMOSA, result 9	1044.7	10	602.3	415.9	26.5
AMOSA, result 10	1043.1	10	602.3	414.2	26.5
AMOSA, result 11	1040.7	10	602.4	410.1	28.1
AMOSA, result 12	1039.2	10	603.3	403.6	32.2
AMOSA, result 13	1038.5	10	603.4	402.9	32.2
AMOSA, mean	1044.1	10	602.3	414.4	25.6

Table 5.4: Results of evolution strategy and AMOSA

The results of AMOSA are clearly better than those of NSGA-II. The costs are lower and the diversity in solutions is larger. Furthermore, the maintenance costs of the solutions of AMOSA are lower than those of the evolution strategy result. However, the availability costs are much higher, therefore also causing the total costs of the AMOSA solutions to be higher than the costs of the evolution strategy solution. Furthermore, when put in perspective, the range of solutions of AMOSA is still quite small. This is easily seen by the fact that the solution of the evolution strategy would be a valid point in the Pareto-optimal solution set of the AMOSA algorithm. However, this point lies far away from the actual solution set, indicating that only a small portion of the theoretical Pareto front is found.

A useful result which was already seen in the NSGA-II results and is now confirmed by the results for AMOSA is that the availability costs rise faster than the maintenance costs fall. Therefore, when the objective is to minimize the total costs, the best way to go is to decrease the availability costs, even if that means increasing the maintenance costs.

Overall, the results of the AMOSA algorithm are not found to be good enough. The added benefits of

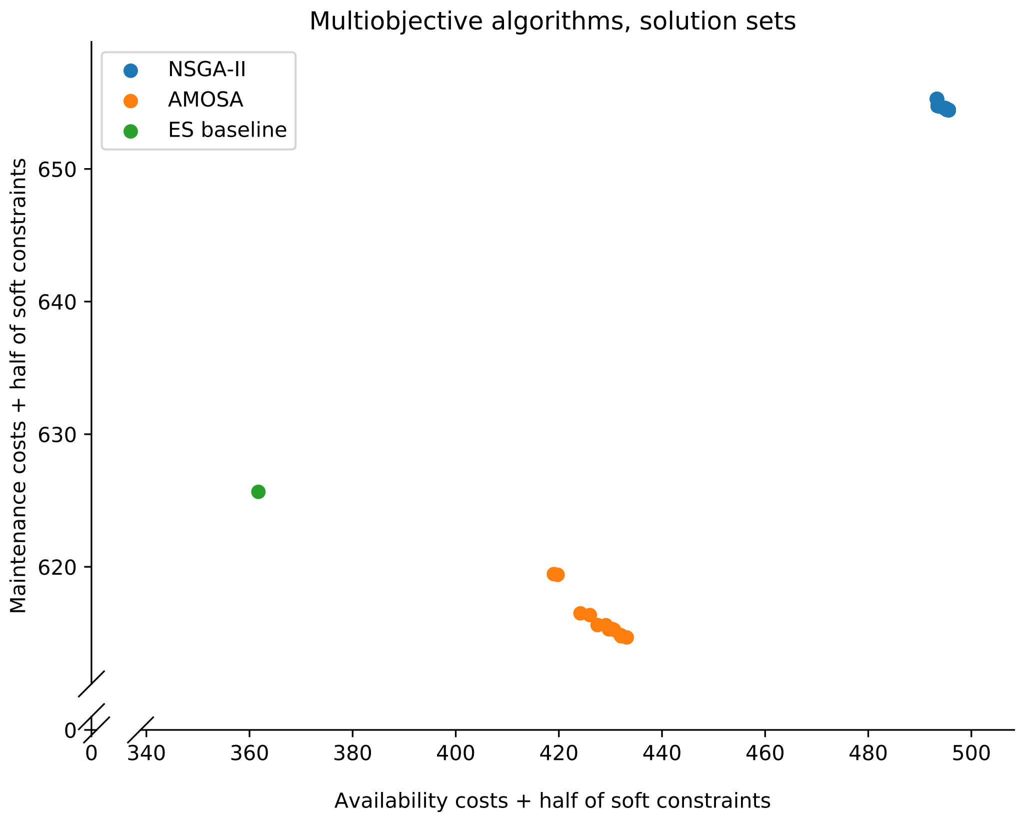


Figure 5.4: Solution sets of AMOSA, NSGA-II, and solution of improved ES, maintenance vs availability

providing the user with multiple solutions are outweighed by the negatives; the diversity in solutions is small, and the total costs are too high. Although further research could probably lead to an improvement of these or other multi-objective algorithms on this problem, the sheer gap in costs between these algorithms and the single-objective evolution strategy leads to the current conclusion that a multi-objective algorithm is not suitable to solve this problem.

5.2.4. Concluding remarks

In this chapter, two multi-objective algorithms were tested on the problem. It was found that the results of these algorithms were of significantly worse quality than the results of the single-objective (improved) evolutions strategy as described in chapter 4. Therefore, the use of a multi-objective algorithm has been discarded. In the next chapter, the evolution strategy will be refined further through a parameter analysis.

6
Parameter analysis for the evolution strategy

From the results seen in chapters 4 and 5, the evolution strategy is the algorithm that works best on the problem so far. Therefore, a parameter analysis was performed for this algorithm. The goal of this analysis is to get an idea of the effect of the different parameters on the solution quality. Due to time limitations, the goal is not to find the perfect parameter settings by iteratively applying small changes. Instead, the performed analysis is a parameter effect *screening* [55], where the goal is to find out which parameters heavily affect the solution quality when changed and which parameters do not effect the solution quality so much. Furthermore, the *direction* of the effect is important to determine, so it becomes clear whether a parameter should be increased or decreased to obtain better solutions. When these goals are reached, a next step would be to take the most significant parameters and iteratively apply small changes to find the optimal parameter settings. However, it is important to screen first so that insignificant parameters can be discarded and fewer runs are needed in the second stage. As indicated before, only the screening is performed in this analysis; fine-tuning the parameters is left as future research.

6.1. Experimental setup

For this parameter analysis, the evolution strategy including all improvements described in chapter 4 is used. This section describes the experimental design of the analysis: which parameters are being optimized and what response variable is being used to analyze the results, what combinations of parameters will be tried, and how will the results be analyzed.

6.1.1. Data

Until now all experiments have been run using the real data for 2019 from ProRail as described in section 2.4. Although using a single dataset suffices to get an idea of the performance of different types of algorithms, for finetuning a single algorithm, using more than one dataset is preferred. Conclusions on the effect of parameters based on the results on a single dataset are prone to be overfitted on that dataset, and the conclusions may not hold for different datasets. Therefore, the parameter analysis has been performed over both the 2019 and 2020 data.

As indicated in section 2.4.1, the data from 2020 is somewhat different from that of 2019. First, it contains fewer project requests: 688, of which 316 create hinder. However, the average project request in 2020 is longer and has more locations than those in 2019. The fact that these differences are present means this dataset is a good one to ensure that the results are stable; it means that both of these types of datasets can appear in real life and the algorithm should be optimized to be able to work with both.

6.1.2. Response variables

The main response variable will be the total costs. The hard constraint violations are expected to be similar for each configuration, so they will initially not be used as a response variable, except when there is a substantial difference observed between the different parameter configurations.

6.1.3. Parameters

The following parameters are present in the algorithm:

- ParentSize: The number of parents used to create offspring in each generation.
- OffspringSize: the number of offspring generated in each generation.
- SelectionType: one of Offspring, ParentPlusOffspring. Decides whether the selection process only selects from the offspring, or from the parents and the offspring.
- Generations: The number of generations to run.
- ConstraintsAllowedStart: The number of constraints allowed at the start of the algorithm, when constraint cooling is used.
- ConstraintsAllowedEnd: The number of constraints allowed at the end of the algorithm, when constraint cooling is used.
- UseConstraintCooling: Whether to gradually decrease the number of allowed constraints over time, as explained in section 4.2.
- A total of eleven mutation weights for the mutations being used: AddBucketMutation, ExpandBucketMutation, ShrinkBucketMutation, DayMutationBucket, HourMutationBucket, FixBucketConflictMutation, DayMutation, HourMutation, MultipleDayMutation, FixConflictMutation, FixDependencyMutation. The weight decides how often a certain mutation is selected.

To allow for a fair comparison, each configuration will be run for the same amount of time rather than generations, because certain parameters, especially the offspring size, affect the average time per generation. Therefore, the Generations parameter will not be set. Each configuration will be run for ten hours.

The ConstraintsAllowedEnd parameter will not be changed, as it will simply yield more hard constraint violations at the end which is not desirable. The UseConstraintCooling parameter will not be changed, as it was shown before that constraint cooling improves the results, and the goal of this experiment is not to test this individual improvement. Furthermore, turning off constraint cooling in some configurations affects the results of the ConstraintsAllowedStart parameter effect, as this parameter becomes obsolete when constraint cooling is turned off.

This leaves fifteen parameters to test. A full factorial design, where each combination of a low and high value for each parameter is tested, would mean 32768 configurations need to be tested which would take approximately 38 years when each run for ten hours. This is obviously not viable, so the number of parameters or the number of tested configurations needs to be decreased.

For starters, the mutations will be grouped into three groups which will all be given a single weight. The distribution of that weight among the mutations within the group will be fixed. The effect of changing the weights within a group is also an interesting research, but it has been omitted in this parameter analysis. The mutation groups are as follows:

- Shift mutations: DayMutation, HourMutation, MultipleDayMutation, DayMutationBucket, HourMutationBucket.
- Bucket mutations: AddBucketMutation, ExpandBucketMutation, ShrinkBucketMutation.
- Fix mutation: FixConflictMutation, FixEventMutation, FixBucketConflictMutation.

This decreases the number of parameters to seven, requiring 128 runs for a full factorial design. This is still too long for the given time frame. Therefore a fractional factorial design will be used, as explained in the next section.

For each parameter, two values will be used, called High and Low. Table 6.1 shows the High and Low values for each parameter, as well as the value that was used in experiments so far. Note that the desired outcome of the parameter analysis is not to directly find the perfect parameter settings, but rather to see which parameter *changes* have a large effect on the outcome. Therefore, using relatively large differences between the Low and High values of a parameter is good, as it will likely give a better picture of the effects of increasing or decreasing a certain parameter. Therefore, values that are relatively far away from the current values have been chosen to test with.

Parameter	Low	High	Current value
ParentSize	10	40	20
OffspringSize	50	170	80
SelectionType	Offspring	ParentPlusOffspring	ParentPlusOffspring
ConstraintsAllowedStart	200	1200	600
Shift mutations	0.2	1.4	0.65
Bucket mutations	0.2	1.4	0.5
Fix mutations	0.2	1.4	0.4

Table 6.1: Parameter values used in the parameter analysis

6.1.4. Fractional factorial design

A fractional factorial design is a subset of a full factorial design. In a full factorial design, if each factor under test takes two different values, all combinations of these values would be tested leading to 2^N configurations, with N being the total number of factors.

A fractional factorial design is used to decrease the number of necessary configurations to be tested while still getting relatively good results. Its successfulness is based on three ideas [54]:

- The sparsity of effects principle: When there are many variables under consideration, it is typical for the system or process to be dominated by main effects and low-order interactions.
- The projective property: A fractional factorial design can be projected into stronger designs in a subset of the significant factors.
- Sequential experimentation: It is possible to combine the runs from two or more fractional factorial designs to sequentially form a larger design that allows estimation of all interaction effects of interest.

When using a fractional factorial design, certain effects will be *aliased* with other (higher-order) interactions. This means that these effects cannot be separated in the results. This is the price to pay for using a fractional factorial design rather than a full one. However, due to the sparsity of effects, it will still be possible to draw solid conclusions about the main effects, and reasonable conclusions about the second order interactions. The quality of a fractional factorial design is determined by the *resolution*. The resolution indicates what kinds of effects are aliased. The three most important resolutions are:

- Type III resolution: No main effect is aliased with any other main effect, but at least one main effect is aliased with a two-factor interaction.
- Type IV resolution: No main effect is aliased with any other main effect or two-factor interaction, but at least one two-factor interaction is aliased with another two-factor interaction.
- Type V resolution: No main effect or two-factor interaction is aliased with any other main effect or two-factor interaction, but at least one two-factor interaction is aliased with a three-factor interaction.

Since higher-order interactions are usually negligible, a resolution IV design should give solid information, especially about the main effects. There are seven factors to test in the evolution strategy, so a 2^{7-3}_{IV} fractional factorial design will be used. This design is of resolution IV and leads to a total of 16 configurations which need to be tested; a manageable number of runs.

A fractional factorial design can be fully specified by its *generators* which indicate how the setting of certain factors is determined. The seven parameters under test are each given a letter in A-G for shorthand notation, as specified in table 6.2. Furthermore, to be able to multiply factors, instead of using "Low" and "High", "-1" and "1" are used to indicate the low and high values of a parameter instead.

Before defining the generator which is being used to analyze the parameters of the evolution strategy, a short example is given. Say there are a total of four parameters that need to be tested; A, B, C, and D. This needs to be done in 8 runs, rather than a full design of 16 runs. Therefore, a full factorial design is created using parameters A, B, and C. Then, the value of D for each run is determined by a generator. An example of such a generator is $I = ABCD$. This means that, in each run, when multiplying the values of A, B, C, and D, the result should always be equal to the identity which is 1. From this, it follows that $D = ABC$, so the value of the parameter D is determined by multiplying the values for A, B, and C. If, for example, a run uses the low

Letter	Parameter
A	ParentSize
B	OffspringSize
C	SelectionType
D	ConstraintsAllowedStart
E	Shift mutations weight
F	Bucket mutations weight
G	Fix mutations weight

Table 6.2: Assignment of parameters to letters in the design

value of A and the high values of B and C, the value for D will be $-1 \cdot 1 \cdot 1 = -1$, so D will use the low value in that run.

In the original problem there are seven parameters which will be tested in sixteen runs. Therefore, a full factorial design is created using parameters A-D, and the values for E, F, and G are determined by the generator.

The generator used is the following:

$$I = ABCE = BCDF = ACDG$$

In practice, this means that the values for E, F, and G are determined as follows:

$$E = ABC$$

$$F = BCD$$

$$G = ACD$$

As an example, if a certain run uses the low value for A, the high values for B and C, and the low value for D, the value for E will be $-1 \cdot 1 \cdot 1 = -1$, so the low value. The value for F will be $1 \cdot 1 \cdot -1 = -1$, so also the low value. The value for G will be $-1 \cdot 1 \cdot -1 = 1$, so the high value.

This leads to the run configuration as specified in table 6.3, where a "-1" indicates the low value of a parameter is used, and a "1" indicates the high value is used.

Run number	A	B	C	D	E	F	G
1	-1	-1	-1	-1	-1	-1	-1
2	-1	-1	-1	1	-1	1	1
3	-1	-1	1	-1	1	1	-1
4	-1	-1	1	1	1	-1	1
5	-1	1	-1	-1	1	1	-1
6	-1	1	-1	1	1	-1	1
7	-1	1	1	-1	-1	-1	1
8	-1	1	1	1	-1	1	-1
9	1	-1	-1	-1	1	-1	1
10	1	-1	-1	1	1	1	-1
11	1	-1	1	-1	-1	1	-1
12	1	-1	1	1	-1	-1	1
13	1	1	-1	-1	-1	1	1
14	1	1	-1	1	-1	-1	-1
15	1	1	1	-1	1	-1	-1
16	1	1	1	1	1	1	1

Table 6.3: 2^{7-3}_{IV} fractional factorial design for parameter analysis. "-1" indicates the low value is used, "1" indicates the high value is used.

From the generators and corresponding design table, the estimation structure can be determined. The estimation structure specifies which effects and/or interactions can be estimated independently from each other. If two effects are aliased with each other, they appear in the same line of the estimation structure. As it is assumed that three-factor interactions and higher are negligible, they will not be shown in the estimation structure. That being said, the estimation structure can be seen in table 6.4.

Effect number	effects
1	A
2	B
3	C
4	D
5	E
6	F
7	G
8	AB + CE + FG
9	AC + BE + DG
10	AD + CG + EF
11	AE + BC + DF
12	AF + DE + BG
13	AG + CD + BF
14	BD + CF + EG

Table 6.4: Estimation structure for the design specified in table 6.3. Each row indicates an effect that can be independently estimated.

As can be seen, the main effects can all be estimated separately from each other. The second-order effects are aliased, due to the design being resolution IV. For example, it is impossible to distinguish between the effects of the AB interaction and the CE interaction.

6.1.5. Other methodological factors

Each experiment will be run on the same machine and will be run for ten hours of *processor* time. By using the processor time rather than the real time, the fluctuations caused by other processes running on the machine is minimized. The runtime of ten hours is slightly low compared to experiments done so far, but it is believed to be long enough to get a satisfactory result for the purpose of this experiment. Due to the fact that a run takes significant time, there will be no replications; each of the 16 configurations will be run once.

6.1.6. Analysis

To analyze the results and draw conclusions about the statistical significance of certain parameters and interactions, Analysis of Variance (ANOVA) will be used. F-tests will be used to determine significance, and a significance level of 0.05 will be used. Manual inspection of the results will also be done and if any anomalies are found they will be corrected in the best possible way.

6.2. Results

Tables 6.5 and 6.6 show the parameter values, costs and hard constraint violations of the parameter analysis experiments for the 2019 and 2020 data respectively.

From these results, some initial conclusions can be drawn:

- Looking at the costs, there are two clear outliers that have significantly higher costs than for the other configuration. This is likely due to low selection pressure. If there are 50 offspring generated each generation and 40 of those are selected as parents for the next generation, and the parents are not considered in the selection process, there is very little selection pressure on the individuals as they will almost always get selected anyway. This leads to a significantly slower convergence which is reflected in the results.

  It must also be concluded that contrary to the assumptions made in the experimental design, this specific three-factor interaction of Parent, Offspring and SelectionType is not negligible. Considering the design and alias structure from the design, the main effect of ShiftMutation is aliased with this interaction, and the results of this main effect will therefore be directly affected by these outliers. This should be kept in mind during the remainder of the analysis.

Parent	Offspring	Selection	ConstraintsAllowed	Shift	Bucket	Fix	Costs	Hard constraints
10	50	Offspring	200	0.2	0.2	0.2	998.9	12
10	50	Offspring	1200	0.2	1.4	1.4	1013.5	11
10	50	ParentPlusOffspring	200	1.4	1.4	0.2	1010.9	10
10	50	ParentPlusOffspring	1200	1.4	0.2	1.4	996.2	16
10	170	Offspring	200	1.4	1.4	0.2	1008.0	10
10	170	Offspring	1200	1.4	0.2	1.4	975.5	19
10	170	ParentPlusOffspring	200	0.2	0.2	1.4	1013.6	10
10	170	ParentPlusOffspring	1200	0.2	1.4	0.2	985.4	18
40	50	Offspring	200	1.4	0.2	1.4	1220.5	17
40	50	Offspring	1200	1.4	1.4	0.2	1216.5	59
40	50	ParentPlusOffspring	200	0.2	1.4	0.2	978.6	17
40	50	ParentPlusOffspring	1200	0.2	0.2	1.4	989.7	17
40	170	Offspring	200	0.2	1.4	1.4	1018.0	10
40	170	Offspring	1200	0.2	0.2	0.2	991.8	15
40	**170**	**ParentPlusOffspring**	**200**	**1.4**	**0.2**	**0.2**	**972.5**	**15**
40	170	ParentPlusOffspring	1200	1.4	1.4	1.4	997.2	10

Table 6.5: Costs and hard constraints for various configurations of parameters. The lowest cost solution is highlighted in **red**

- The hard constraints are, as expected, close together, except for one outlier as already explained in the previous point. However, in previous experiments, it was shown that, at least for the 2019 data, the costs usually rise quite significantly with solving the final few remaining constraint violations, and therefore, the difference between for example 10 and 15 hard constraint violations should not simply be discarded.

Taking these points in mind, to provide an overview that is as complete and fair as possible, three different analyses will be done. The first is done on only the costs as they are; the outliers will be kept as-is and the hard constraints are not considered. The second analysis is done on costs which are adjusted for constraint violations; for each constraint violation, 2.4205 costs will be added. Third, the two outliers will be *winsorized*. This means they are set to the 95th percentile of the rest of the data. Furthermore, the penalty for hard constraint violations will still be added as in the second analysis.

In the following chapters, the most important results are shown. Full ANOVA tables and interaction plots can be found in appendix A.

6.2.1. As-is analysis

The first analysis uses the costs without any adjustments made for outliers or constraint violations. First, a linear regression is fitted on the full model, for both datasets. Keep in mind that only one term of each row from the estimation structure in table 6.4 can be included. For example, if the model contains the ParentSize:OffspringSize term, this is actually this term aliased with two other two-factor interactions.

The resulting models have no significant terms when using a significance threshold of 0.05. To increase the degrees of freedom of the residuals, the most insignificant two-factor interactions will be removed from the model. These are ParentSize:ConstraintsAllowed, ParentSize:BucketMutation, ParentSize:FixMutation and OffspringSize:ConstraintsAllowed, all of which have a p-value larger than 0.40 for both the 2019 and 2020 data.

In these reduced models, significant terms start to appear. The main effects for ParentSize, OffspringSize and SelectionType, as well as the two-factor interactions ParentSize:SelectionType and ParentSize:ShiftMutation are all statistically significant for both datasets. Furthermore, the main effect for ShiftMutation, and the interaction ParentSize:OffspringSize are significant in the 2019 model, where the latter has a p-value of 0.054 in the 2020 model, meaning it's close to being significant there. It must be kept in mind that the ShiftMutation main effect is aliased such that it is directly affected by the outliers caused by low selection pressure.

From this analysis, it is concluded that ParentSize, OffspringSize and SelectionType are the most important parameters. However, this does not say whether a high or low value should be chosen for these parameters. To visualize this, the main effects of the parameters for 2019 and 2020 are plotted in figures 6.1 and 6.2 respectively.

Parent	Offspring	Selection	ConstraintsAllowed	Shift	Bucket	Fix	Costs	Hard constraints
10	50	Offspring	200	0.2	0.2	0.2	1121.1	27
10	50	Offspring	1200	0.2	1.4	1.4	1125.5	29
10	50	ParentPlusOffspring	200	1.4	1.4	0.2	1079.4	24
10	50	ParentPlusOffspring	1200	1.4	0.2	1.4	1114.1	30
10	170	Offspring	200	1.4	1.4	0.2	1121.3	27
10	170	Offspring	1200	1.4	0.2	1.4	1064.6	27
10	170	ParentPlusOffspring	200	0.2	0.2	1.4	1098.3	27
10	170	ParentPlusOffspring	1200	0.2	1.4	0.2	1120.1	21
40	50	Offspring	200	1.4	0.2	1.4	1263.8	74
40	50	Offspring	1200	1.4	1.4	0.2	1273.6	140
40	50	ParentPlusOffspring	200	0.2	1.4	0.2	1101.8	30
40	50	ParentPlusOffspring	1200	0.2	0.2	1.4	1108.8	31
40	170	Offspring	200	0.2	1.4	1.4	1109.9	26
40	170	Offspring	1200	0.2	0.2	0.2	1115.8	25
40	**170**	**ParentPlusOffspring**	**200**	**1.4**	**0.2**	**0.2**	**1055.5**	**24**
40	170	ParentPlusOffspring	1200	1.4	1.4	1.4	1140.1	22

Table 6.6: Costs and hard constraints for various configurations of parameters. The lowest cost solution is highlighted in red

As expected, the significant main effects have a larger slope than the insignificant ones. From the plots, it can be concluded that, according to these experiments, good parameter settings are a low ParentSize, high Offspring size, and SelectionType "ParentPlusOffspring". Furthermore, there seems to be a slight preference for a low bucket mutation, although this effect is small and statistically insignificant. The ShiftMutation main effect clearly points towards a low value for this parameter but is heavily influenced by the outliers. The effects for ConstraintsAllowed and FixMutation are flipped between the 2019 and 2020 models. For the FixMutation, the effect is small in both models, and its value does not influence the results much. For constraints allowed, there is a somewhat larger effect in the 2020 data, although still statistically insignificant. This could indicate that, for the type of dataset with fewer, larger blocks like 2020, allowing many constraint violations at the start could lead to worse results.

An interesting result is the fact that a high offspring size looks to be better than a low one, even though a higher offspring size causes a longer runtime per generation and therefore less generation within a certain time. Apparently, doing fewer iterations where more possibilities are tried per generation is beneficial for the performance of the algorithm.

Finally, special attention is drawn to the interaction between ParentSize and OffspringSize. The main effects plot indicated that a low parent size would be good. However, in the interaction plots included in appendix A it becomes clear that this is only true when the offspring size is also low. This makes sense, as a high parent size combined with a low offspring size would yield too low of a selection pressure. However, when the offspring size is high, a setting that was also indicated to be good by the main effects plot, the setting for parent size is negligible as seen in the interaction plot.

6.2.2. Adjusting for constraint violations

Even though the observed constraint violations are relatively close together, the fact that it was previously seen that solving the final constraint violations usually increases the costs relatively fast, leads to this second analysis being performed. Here, the costs are still used as the response variable, but 2.4205 costs are added as a penalty for each hard constraint violation. This number was not experimentally determined, and it is unlikely that a single penalty for each hard constraint violation can model an appropriate penalty well. However, this parameter analysis is in an early stage where initial conclusions about the effects of parameters are drawn. For that purpose, this penalty function is believed to produce satisfactory results, especially when looking at the differences with the model without the penalty function and seeing how the effect of various parameters may change.

As before, a linear regression is fitted on both data sets. An interesting difference between the 2019 and 2020 data is observed here. The 2019 model contains, similar to the previous analysis, no significant effects in the full model. On the contrary, all but one effect in the 2020 model are statistically significant. The effects

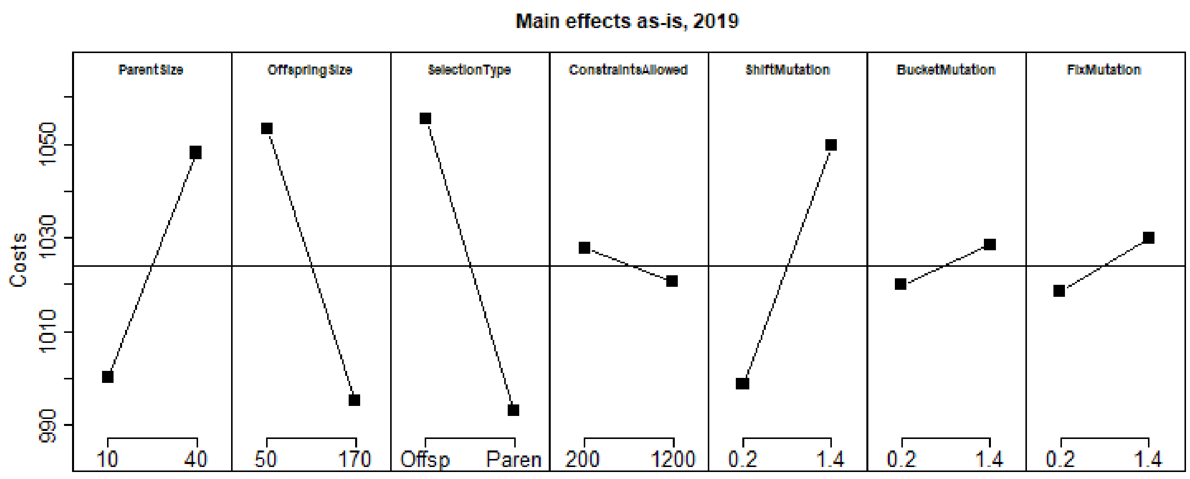


Figure 6.1: Main effect plot of the as-is parameter analysis, 2019 data

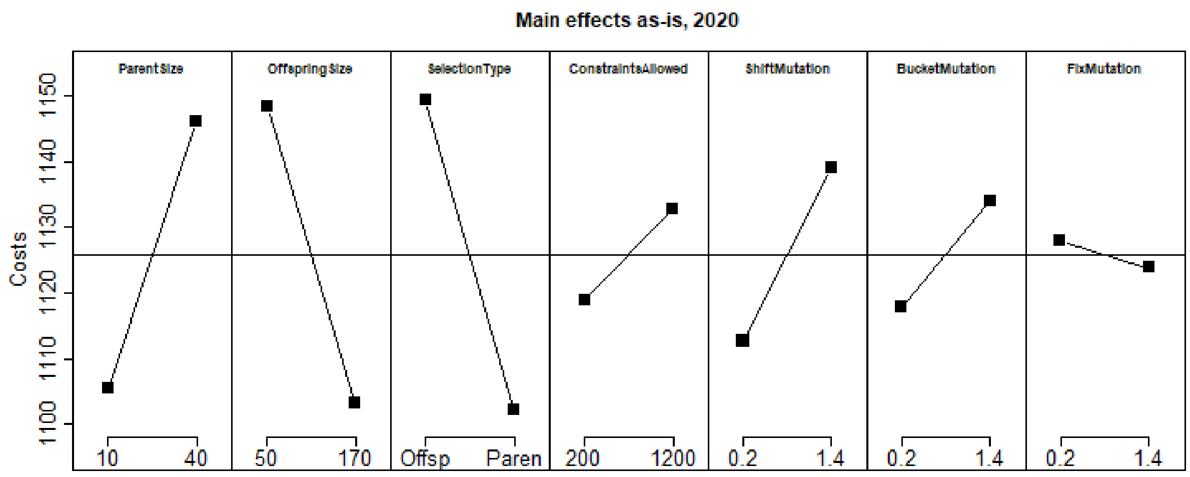


Figure 6.2: Main effect plot of the as-is parameter analysis, 2020 data

of the parameter settings are possibly more important on data sets like the 2020 data. Removing the most insignificant interactions in the 2019 model yields the same significant effects as in the as-is analysis.

The main effects plot for the constraint adjusted models for 2019 and 2020 are shown in figures 6.3 and 6.4 respectively.

First, the constraint adjusted main effects are compared to the as-is main effects. The three most significant parameters; ParentSize, OffspringSize and SelectionType have similar slope and direction after adjusting for constraints. The effect direction of ConstraintsAllowed and FixMutation have flipped in the 2019 model, but the effects are small in both models. A possible explanation for this lies in the fact that a higher number of allowed constraints leads, on average, to a final solution that has slightly lower costs but slightly higher constraint violations, simply because the number of constraints during the optimization process will increase further than when a low number of constraints is allowed. The algorithm may not be able to fully solve all these constraints near the end. Similarly, a low fix mutation weight may cause similar solutions with slightly higher constraint violations and slightly lower costs, simply because there are fewer mutations taking place on average that focus on removing hard constraint violations. This means that, when not adjusting for constraints, the lowest cost (but higher constraint violation) was preferred. However, now that a penalty is applied for constraint violations, it appears that a solution with lower constraint violations and, because of that, slightly higher costs, would be preferred. It must be noted that this trade-off is largely dependent on the

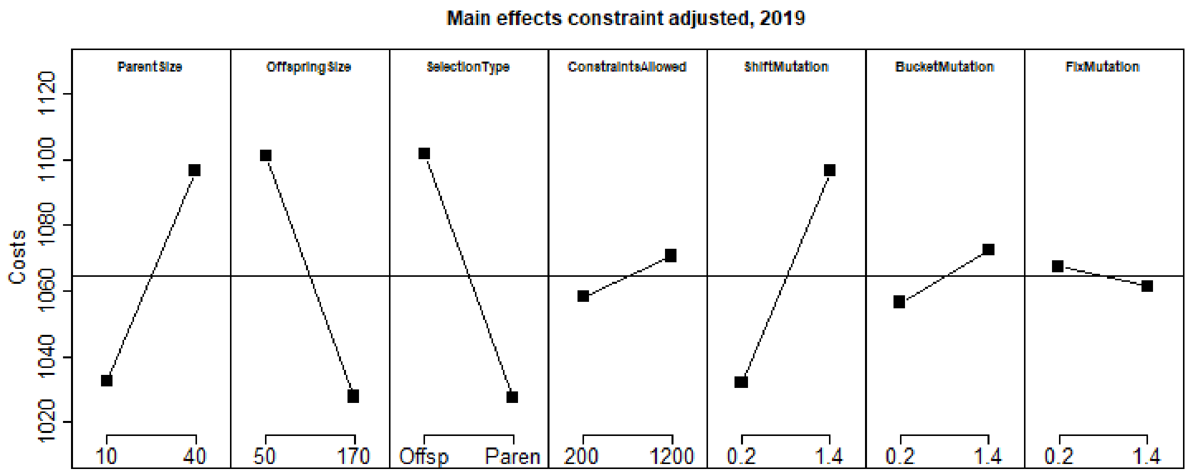


Figure 6.3: Main effect plot of the constraint adjusted parameter analysis, 2019 data

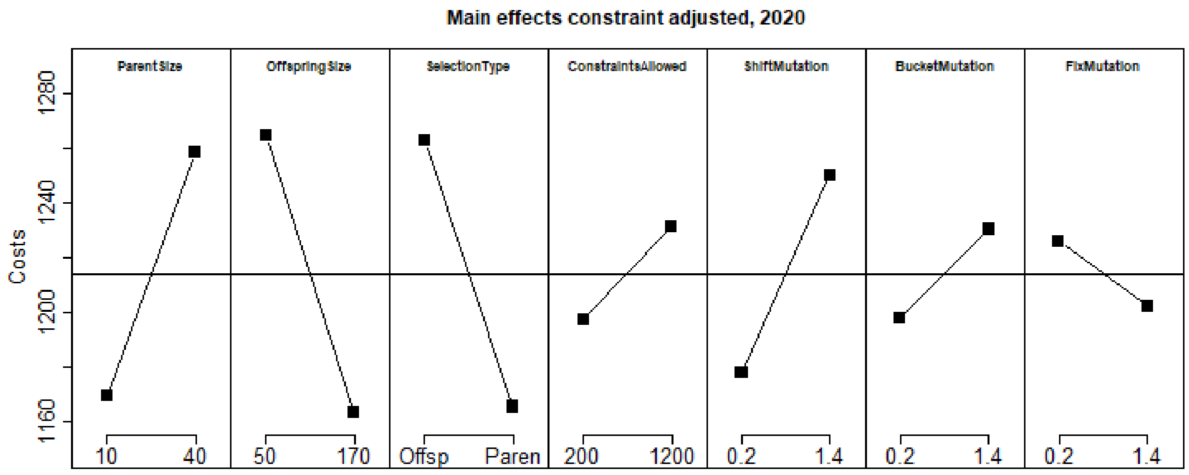


Figure 6.4: Main effect plot of the constraint adjusted parameter analysis, 2020 data

chosen penalty for the hard constraint violations. It is, however, an interesting insight and one that should be kept in mind when deciding whether a solution with some more hard constraint violations is preferred over a solution with some lower costs. The effects for the 2020 data are very similar before and after constraint adjustment.

Second, the two data sets are compared. It is noticed that the slopes of all main effects now have the same direction in both models, meaning the effect of increasing and decreasing a parameters looks to be the same over both data sets. This means the results are more meaningful and are less likely to be overfitted on a single data set. The effects in the 2020 model are somewhat larger, indicated by the steeper slopes. As seen in the as-is analysis, the effect of using a small number of allowed constraints is more beneficial for the 2020 data than for 2019. A similar effect is observed for the bucket and fix mutations.

In the as-is analysis, it was already shown that the preference for a low parent size as indicated by the main effects plot was only important when the offspring size was also low. In this constraint-adjusted analysis, it is even visible that for a high offspring size, a high parent size is slightly preferred. Therefore, it is concluded that from this analysis, it is beneficial to the performance of the algorithm to set a high parent size, high offspring size, and set the selection type to "ParentPlusOffspring". Furthermore, using a low number of allowed constraints is beneficial, especially when the data resembles the 2020 data.

6.2.3. Winsorized outliers

The two outliers caused by the specific configuration of parent size, offspring size and selection type leading to a very low selection pressure impact the results significantly. Although these outliers are not necessarily erroneous in the sense that they are valid results of the experiments, it is deemed interesting to see what happens if they are brought closer to the other values. A possible way to deal with outliers as explained by Aguinis *et al.* [13] is *winsorization*. This is the process of setting outliers to a certain percentile of the remainder of the data. In this case, the two outliers have been set to match the 95th percentile of the rest of the data. The penalty for hard constraint violations is also still in place.

For both fitted models, no significant effects are present. In both models, the most insignificant main effects are removed. Afterwards, for the 2019 data, the main effects for OffspringSize, SelectionType and FixMutation are significant, as well as the interactions for ParentSize:SelectionType and OffspringSize:ConstraintsAllowed. Unexpectedly, there are no significant effects at all in the 2020 data, even in the reduced model. This is especially surprising because there were so many significant effects in the 2020 model for the constraint adjusted analysis described before.

Figures 6.5 and 6.6 show the main effects for 2019 and 2020 respectively.

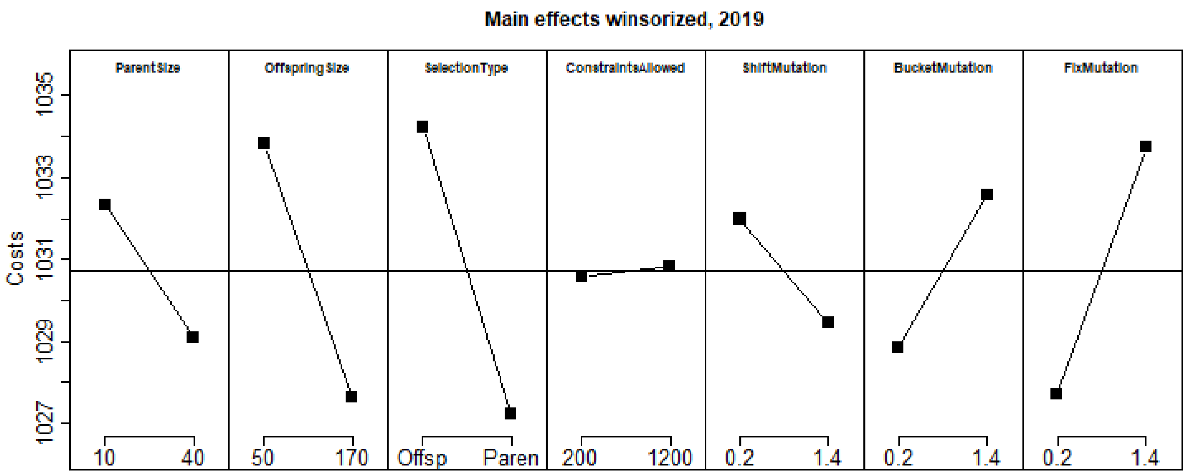


Figure 6.5: Main effect plot of the winsorized parameter analysis, 2019 data

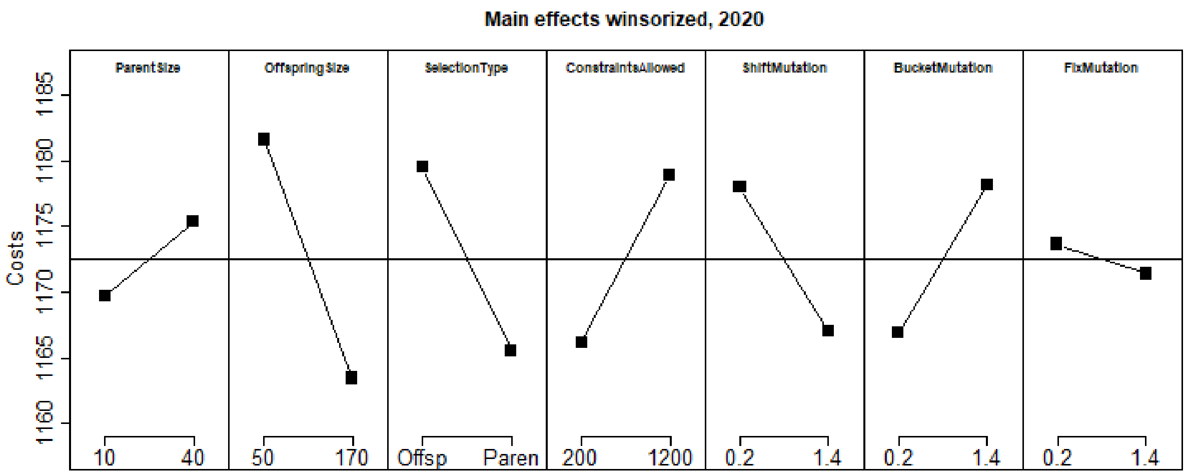


Figure 6.6: Main effect plot of the winsorized parameter analysis, 2020 data

First, the differences and similarities with the previous two analyses are discussed. The effects of OffspringSize and SelectionType are still strong and have the same direction. This has been the case in all analyses and datasets so far, so a more certain conclusion about these effects can be drawn. The effect for ShiftMutation is flipped when compared to the as-is and constraint adjusted analyses. This is as expected, as this effect is aliased with the specific three-factor interaction that has been altered here. The effects of ConstraintsAllowed and BucketMutation show similar behaviour as before. The effect of ParentSize is weaker now; it's no longer significant in any of the two data sets, and the direction of the effect is disagreed upon between the 2019 and 2020 data. Earlier, it already became clear from the interactions that for a high offspring size, the parent size is not that important. This conclusion is supported here again; now that the results for the ParentSize effect are less affected by the outliers, its effect is smaller.

For the bucket mutation, and in lesser form also for the fix mutation, the analysis shows that a lower weight may be better for these mutations. Initially, this may be a counter-intuitive result, as the bucket- and fix mutations are "smarter" mutations than the shift mutations, and they were designed specifically for this problem. However, what may be happening here is that there are too many useless mutations happening when the weights for these mutation groups are high. Especially near the end of the algorithm, almost all conflict and dependency violations are already fixed, and almost all (eligible) requests are already in a bucket. At this point, doing more bucket and/or fix mutations is essentially a waste of a mutation, whereas a shift mutation, although more random, has a higher chance to find an improvement in costs. In section 6.3, a possible improvement regarding these mutations is explained.

6.3. Conclusions and adaptations

Overall, a number of conclusions can be drawn:

- The main effects for OffspringSize and SelectionType were always significant and in the same direction; a high offspring size and selection type "ParentPlusOffspring" should be used.
- The main effect of ConstraintsAllowed is insignificant, but increasing it may cause solutions to have slightly higher hard constraint violations, and slightly lower costs. This could be used if any of these two is favoured over the other, for whatever reason.
- The main effect of ShiftMutation is heavily impacted by the outliers caused by low selection pressure, so it is hard to draw solid conclusions for this parameter.
- The main effect of ParentSize is also affected by the outliers. However, from the interaction plots and the analysis on the winsorized data, it seems a higher parent size is slightly better in case a high offspring size is used. The latter is true, as confirmed by the main effect for offspring size.
- The main effects for BucketMutation and FixMutation indicate that these more sophisticated mutations may need to be used more sparingly, as there are certain situations in which their effect is minimal.

Based on these conclusions, the following adaptations have been made:

- The parameter for offspring size is increased from 80 to 170. Further tests are necessary to see if increasing the offspring size further has an increasingly positive effect. These tests are not performed in this thesis but are left as future work.
- The parent size is increased from 20 for 40, and the selection type is kept at ParentPlusOffspring.
- The weights for the three mutation groups are kept equal for now, but a test has been performed with an *adaptive mutation*, which is explained and tested further in section 8.2.

7 Greedy algorithms

The implemented changes described in chapter 4 have successfully led to an improvement of the existing baseline. However, there is still some room for (further) improvements in some areas:

- The runtime of the algorithm is long. A runtime of at least 10 hours is needed to get satisfactory results, and a runtime of 24 hours or more would be preferred. Although this is not a problem per se, as this scheduling problem spans a whole year and in theory there is plenty of time to run the algorithm for a year ahead, in practice, the schedulers are subject to possible changes in data. Furthermore, they might desire to manually move around and "fix" certain project requests in place, and re-plan around that. Finally, they might want to run a large number of different constraint scenarios. There are also some ideas for a feedback loop with another algorithm which creates project requests from asset operations, which would require a relatively fast result from the project request planner. Although this integration is not in the scope of this thesis, it is kept in mind when deciding on the next step.
- Although the algorithm always produces a somewhat satisfactory result given a decent amount of runtime, there is still quite some variation in the results of the algorithm because of the high amount of randomness. Although this does allow good results to be returned with some probability, there are also some worse results, and especially when run on a new dataset, it may not always be clear whether the results are good or not, and multiple runs would be required to get an idea of the spread of the results. More information on the robustness of the algorithm can be found in section 8.1.
- The lengths and number of locations in the given project requests follow a long-tail distribution as shown in section 2.4.1. This means that a relatively small number of blocks makes up a relatively large portion of the costs and the difficulty or the problem. Therefore, finding a good starting time for the largest blocks is important, and the starting times for the smaller blocks are not *that* important.

These points led to the idea of using a greedy algorithm. Greedy algorithms follow the problem-solving heuristic of making a series of locally optimal choices, in the hope that this will lead to a solution that is (close to) globally optimal. Greedy algorithms are generally quite fast. Their solution quality depends heavily on the problem as well as the specific problem instance; certain structures may cause a greedy algorithm to make bad decisions in the early stages causing a far from optimal solution.

In this chapter, two greedy algorithms that were implemented and tested on the given maintenance scheduling problem are discussed. Furthermore, a randomization component is introduced in the greedy algorithm, and a hybrid greedy and evolution strategy algorithm is designed and tested.

7.1. Time-window based greedy algorithm

The first greedy algorithm was inspired by the heuristic used to create starting individuals for the metaheuristic algorithms, as explained in section 3.4.2. Pseudocode for this algorithm is given in algorithm 7.1.

As before, the project requests with hinder are planned before the ones without. The blocks with hinder are ordered on the length of the block times the number of locations. This is a good indicator for the number of availability costs the request will cause. As indicated before, there are few large blocks and many small blocks. By planning the requests in this order, there will be many optimal or near-optimal plan dates for the

Algorithm 7.1 Time-window based greedy algorithm

```
Separate the project requests in two lists $H$ (with hinder) and $NH$ (without hinder).
Sort $H$ on number of locations * length in hours, descending
Sort $NH$ on length in hours, descending
Construct four types of time windows: night (22:00-06:00), weekend (Fri 22:00 - Mon 06:00), short holiday of at least one week, summer holiday.
for project request $x \in H$ do
    for each time window $t$ that $x$ fully fits in do
        Try planning $x$ at the start of $t$ and keep track of the best result so far
    if the hard constraint violations increased for each tried time window then
        Also try planning in time windows that $x$ does not fully fit in.
    Choose the time window $t$ with the best fitness and plan $x$ at the start of that time window
for project request $x \in NH$ do
    for each day $d$ in the global plan window do    ▷ Blocks without hinder are very fast, so each day can be tried
        Plan $x$ at the start of $d$ and keep track of the best result so far
    Plan $x$ at the best found day.
```

most impactful blocks. This means that there is probably not a lot of space left for the small blocks in the optimal time slots like the summer holiday, but because they are relatively unimportant, this will hopefully not matter that much.

The requests are then planned in each time window where the request fits, and the best possibility in terms of costs and constraints is kept track of. As the hard constraint violations are leading over the costs in terms of comparing solutions, keeping those low is the most important. Therefore, when no fitting time windows are found where the hard constraint violations do not increase, the non-fitting time windows are also tried. It is entirely possible that, after trying these time windows as well, there is still no possible plan date where no extra hard constraint violations are caused. In any case, the best possible time window that was tried will be chosen. To keep the tried number of possible plan dates low and therefore decrease the runtime, only the start of each time window is tried, even though there may be some leeway given the lengths of the time window and the request.

The requests without hinder are planned afterwards. Here, the personnel costs make up the largest portion of the relevant costs. As planning a request without hinder takes much fewer checks and is therefore quite fast, in this section of the algorithm, instead of choosing certain time windows, each day in the global plan window is simply tried and the best result is chosen. As the blocks with hinder are much more important to get right than those without, the focus is on that part and the blocks without hinder can be planned in a relatively simple way.

7.1.1. Results

The given algorithm is deterministic, so running it multiple times will give the exact same result. The runtime of the algorithm is approximately 10 minutes. The achieved solution has total costs of 995.9 with 13 hard constraint violations. This is approximately 23 higher in costs than the lowest-cost result achieved using the evolution strategy, during the parameter analysis, but has two less hard constraint violations. The greedy algorithm is definitely still worse in terms of solution quality than the evolution strategy. However, the runtime is significantly lower. This would allow the user to quickly get an idea of the expected costs for a new dataset, or to see what happens if certain blocks are fixed in place, as the greedy algorithm will easily plan around it. Furthermore, the greedy algorithm is still relatively simple, and because requests are only planed at the start of the given time windows, it is the expectation that especially the holidays are massively underused. This means there is probably room for improvement in the greedy algorithm. An improved algorithm is discussed in the next section.

7.2. Improved greedy algorithm

The greedy algorithm explained in section 7.1 gives a decent result in a short period of time. However, there are two major flaws in the algorithm:

- Project requests are only planned at the start of selected time windows. For large time windows such as the summer holiday, this means they are underutilized. Two blocks that potentially conflict with each other could be planned after each other in the same time window, but are instead planned at the same time and will therefore cause more hard constraint violations. This is especially bad since these large time windows are the best options when it comes to availability costs.
- The project requests are sorted based on number of locations and length. However, certain locations have a lot more passengers travelling through them then others, meaning they also cause much more availability costs. The number of passengers is currently not taken into account. This causes blocks that are long, but affect locations that are not busy to be planned early in the algorithm, possibly claiming a large part of a favourable time window like the summer holiday, while the ERM saved is actually quite low due to the low number of affected passengers. It would be better to plan other blocks that affect busier locations in these time windows.

Based on these problems, an improved greedy algorithm is proposed. Its pseudocode is found in algorithm 7.2.

Algorithm 7.2 Improved greedy algorithm

```
Separate the project requests in two lists $H$ (with hinder) and $NH$ (without hinder)
Sort H descending on the *expected affected passengers*: the average number of affected passengers per hour for each of its locations times the length of the request in hours.
Sort NH descending on length in hours
**for** each project request $x \in H$ **do**
    **if** length of $x < 4$ **then**
        startTime ← 01:00
    **else**
        startTime ← 22:00
    **for** each day $d$ in the global plan window **do**
        Plan $x$ on $d$ at *startTime* and keep track of the best result.
    Plan $x$ on the found best result
**for** each project request $x \in NH$ **do**
    **for** each day $d$ in the global plan window **do**
        Plan $x$ on $d$ at 07:00 and keep track of the best result.
    Plan $x$ on the found best result
```

Instead of sorting on the number of locations times the length of the request, the actual number of passengers that travels through the affected locations is now also taken into account. Furthermore, the notion of time windows is abandoned, and now each day in the global plan window is tried. To minimize availability costs, a favourable start date is chosen; 01:00 for requests of four hours and less and 22:00 for other requests. This leads to maximum usage of the low number of passengers during night and weekend when planned in one. The request is planned on the best possible date at that time. For the requests without hinder, not much changes compared to before, except for the fact that they are now specifically planned at 07:00 to achieve maximum usage of the relatively cheap personnel costs during the daytime of weekdays.

7.2.1. Results

Due to the fact that more possibilities are tried, the runtime of this algorithm slightly increases compared to the previous greedy algorithm and takes approximately fifteen minutes. As before, the algorithm is deterministic, so running it multiple times gives the same result. The solution achieved by this algorithm has a total cost of 980.5 with 10 hard constraint violations. The greedy algorithm works surprisingly well; although it is not a sophisticated algorithm, the results are similar in terms of costs and constraints to the results of the evolution strategy, but the runtime is much lower. This means that this greedy algorithm can be used to give a good solution in a very short time frame, allowing for more interactivity from the user side. As the decrease in costs is significant and the runtime only increases slightly compared to the old greedy algorithm, this improved greedy algorithm is used in the remainder of this chapter and thesis, and will henceforth be referred to as "the greedy algorithm".

Table 7.1 shows an overview of the relevant results of the (improved) evolution strategy and the greedy algorithms.

Algorithm	Total costs	Allowed constraints
Evolution strategy baseline	1005.6	9
Evolution strategy, parameter analysis best result	972.5	10
Initial greedy algorithm	995.9	13
Improved greedy algorithm	980.5	10

Table 7.1: Comparison of evolution strategy and greedy algorithms

7.2.2. Allowing more constraint violations

It may be desirable to allow the algorithm to produce a result with more constraint violations but fewer costs. Therefore, the algorithm is slightly adapted. Instead of always choosing the best option, where hard constraints are leading over costs, an *allowed constraints* parameter is given. As long as the total number of constraint violations is lower than this number, only the costs are considered. This rule works well in practice because the most impactful blocks are planned first, so trading a few hard constraint violations for some overlap, for example, is most effective in these blocks anyway so the allowed constraints will get "used up" by the most favourable situations.

Table 7.2 shows some results of a number of allowed constraints.

(Allowed) constraints	Costs
10	980.5
13	965.4
18	957.7

Table 7.2: Constraints versus costs for greedy algorithm

An interesting result becomes visible here; allowing a small number of extra constraint violations gives a relatively large cost decrease. This indicates that solving the final few constraint violations usually leads to a lot of extra costs, likely because two large, partly overlapping but conflicting requests have to be separated, causing a large amount of availability costs.

7.3. Introducing randomness in the greedy algorithm

The greedy algorithm is currently deterministic causing the exact same solution to be outputted when the algorithm is run multiple times on the same data. Although this means the algorithm is as robust as it will get, it would be good to create some randomness for two reasons:

1. When the requests are planned in a slightly different order, the solution quality may improve. However, this would currently require a whole different metric to order the project requests.

2. In order to hybridize the greedy algorithm together with a metaheuristic like the evolution strategy, for example to create an initial starting population, some variety in solutions is wanted in order to have a diverse starting population, for example.

To address these points, various methods of introducing randomness have been implemented. Randomness is introduced in two different places. First, the order of the requests can be randomized, causing a (slightly) different plan order compared to the deterministic version. Second, instead of always choosing the best plan date, a second or third best option could be chosen with some chance. The three implemented methods of randomized are as follows:

- *Roulette wheel selection*: Given some kind of metric per project request, such as the expected affected passengers used to order the requests in the deterministic greedy algorithm, select the next request to be planned in each iteration by roulette wheel selection on this metric. This means blocks that should be more favoured to be planned first must have a higher metric value. For metrics that are inversely ordered, a simple fix is available by using $\frac{1}{x}$ rather than x. A parameter n indicates that the first n requests should be planned by roulette wheel selection, and the remainder of the requests should be planned deterministically.

- *Randomize next request*: The requests are ordered as if they would be planned deterministically. Each iteration, one of the top n requests is chosen to be planned next, according to a predefined distribution. For example, the top request is chosen 50% of the time, the second request 35% of the time and the third request 15% of the time.
- *Randomize start time*: Instead of just keeping track of the best plan date for a request, an ordered list of solutions is kept. At the end of each iteration, the best n options are chosen with a certain predefined probability.

7.3.1. Distributions of solutions

To test the effect of randomness, five different randomization setups have been made and each has been run for twenty times on the 2019 data and ten times on the 2020 data. The five ways of randomness are as follows:

- Roulette wheel selection, $n = 50$.
- Roulette wheel selection, $n =$ total number of requests.
- Randomize next request, probabilities 50%/35%/15%.
- Randomize start time, probabilities 50%/35%/15%.
- Randomize next request and start time, probabilities for both 50%/35%/15%.

The first configuration uses only a small number of project requests for the roulette wheel selection, but these are the requests with the most impact. It is believed that only randomizing this limited number of requests provides sufficient variation in results, while the chance of inferior results obtained due to a suboptimal plan order is mitigated, compared to the second configuration where all requests are included in the roulette wheel selection. The weights for the last three configurations are chosen in such a way that a (perceived) better option always has the highest chance of being chosen, but the other options have a significant chance as well. Furthermore, the same distribution is used over the three configurations to provide a fairer comparison.

Figures 7.1 and 7.2 shows scatter plots of costs and constraint of all generated solutions by the various methods, on the 2019 and 2020 data respectively. Table 7.3 shows the average and lowest cost solution for each method on both datasets.

Configuration	Costs 2019	Constraint violations 2019	Costs 2020	Constraint violations 2020
Roulette wheel				
average	996.7	11.9	1114.5	21.7
lowest costs	959.7	11	**1064.7**	**26**
Roulette wheel 50				
average	**978.3**	**10.9**	**1105.6**	**21**
lowest costs	956.6	10	1074.7	25
Next request				
average	980.1	11.4	**1110.1**	**15.8**
lowest costs	972.6	11	1089.5	15
Starting time				
average	**960.8**	**12.6**	1139.2	48.7
lowest costs	956.2	12	1126.9	52
Both request and time				
average	966.1	12.4	1137.2	48.3
lowest costs	**947.9**	12	1118.8	50

Table 7.3: Results of various configurations of randomization in the greedy algorithm. Highlighted are the lowest constraint average in **blue**, the lowest cost average in **red** and the overall lowest cost result in **orange**

The results are quite different over the two datasets. On the 2019 data, the "roulette wheel" method gives the worst results, probably caused by the fact that insignificant blocks have a chance to be chosen early and may claim a good time window, thereby prohibiting more significant blocks to be planned there. There is

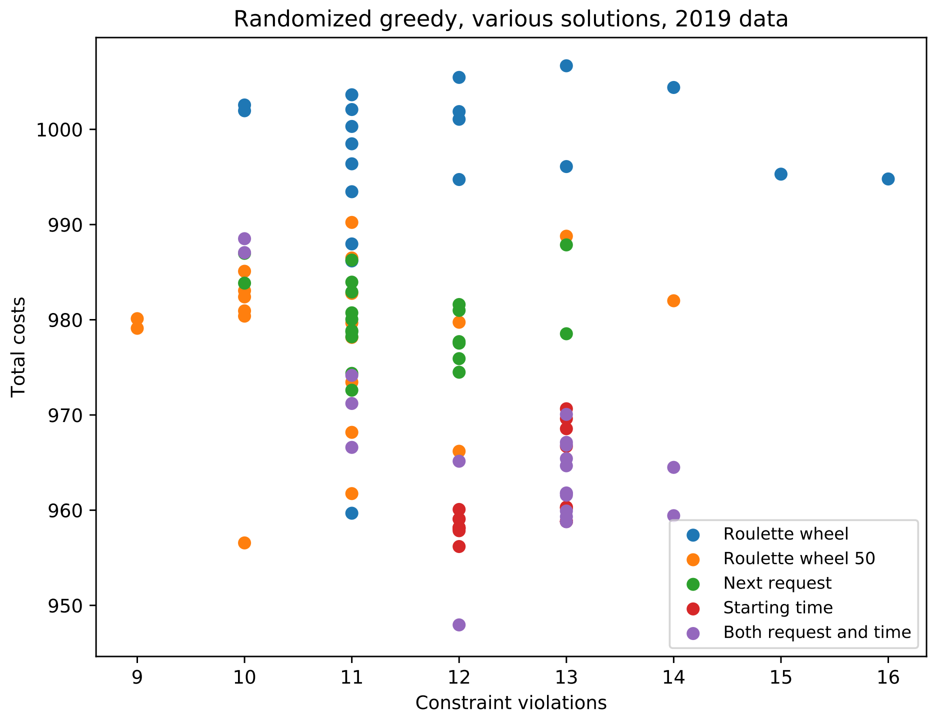


Figure 7.1: Solutions of various randomization methods for the greedy algorithm, 2019 data

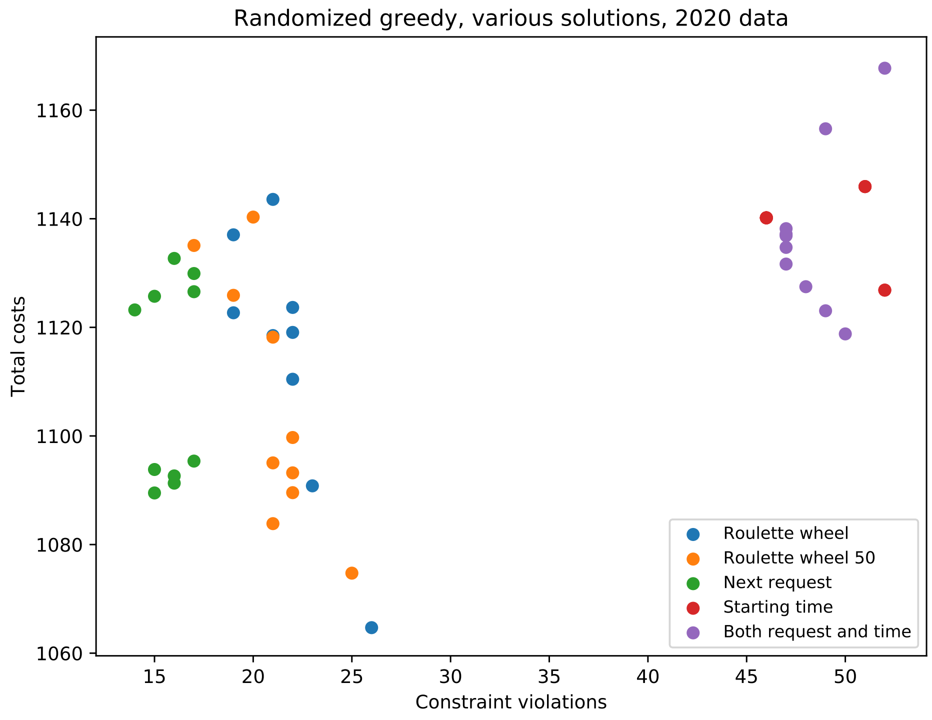


Figure 7.2: Solutions of various randomization methods for the greedy algorithm, 2020 data

essentially too much randomness for the algorithm to still be effective. The solution for randomizing "both" start time and next request give solutions that have low costs on average but have some more hard constraint violations as well. A similar thing holds for randomizing the starting time only. The "roulette wheel 50" and "next request" methods achieve fewer constraint violations, but pay the price for that by having higher costs.

On the 2020 data the solutions groups are much more separated from each other. The two methods that randomize the starting time of the planned requests are clearly much worse in terms of costs and constraints. This is interesting, as these two methods were the ones with the lowest average cost on the 2019 data. The other three methods are quite close in terms of average costs, but do differ in average constraint violations.

Overall, the "next request" randomization gives the best results; on the 2020 data is it the best method, and on the 2019 data it does not achieve the lowest average cost, but strikes a good balance between costs and constraint violations.

Figures 7.3–7.6 show simple kernel density estimates for the costs of the different randomization methods on both datasets. For the 2019 data, figure 7.3 shows the estimated densities for costs only, and figure 7.4 adjusts for hard constraint violations by adding two million costs for each violation; the same number that was used in the constraint-adjusted parameter analysis. Figures 7.5 and 7.6 show similar figures for the 2020 data. The default *gaussian_kde* function of Python was used; this function uses a normal kernel with a bandwidth equal to the "optimal" bandwidth according to Silverman [60]

The density estimates confirm the conclusions drawn above. Furthermore, they show more clearly how wide the distributions are. The distributions for randomizing the starting time is quite narrow, meaning it is robust in its solutions. Conversely, the distributions for roulette wheel is quite wide. This is to be expected due to the fact that there is more randomness when using roulette wheel than when randomizing the starting time, given the used probabilities.

When looking at which method is best, this depends somewhat on the dataset but the "next request" method looks to be the most robust over the two datasets. On the 2019 data it is not the best in terms of costs, but it is not much worse than other methods, especially in the constraint-adjusted version. On the 2020 data, this method is one of three that achieves the lowest costs solutions, and of these three has the lowest average number of constraint violations. The "next request" method will therefore be used in the hybrid algorithm introduced in the next section.

Finally, it is important to notice that the baseline result of the deterministic algorithm on the 2019 data is improved upon in multiple runs; although the average result is slightly worse than the deterministic version, there are random instances in which the result is better. This means that it is favourable to run the greedy algorithm with some randomization for multiple times and keep track of the best result. Since the runtime is short anyway, the greedy algorithm can be run many times in the same timespan that, for example, the evolution strategy algorithm would take.

7.4. Hybrid greedy and evolution strategy

Upon further inspection of the results of the evolution strategy and the (randomized) greedy algorithm, an interesting discovery was made. The results achieved by some runs of the randomized greedy algorithm were better in terms of costs than the results of the evolution strategy. However, the availability costs were usually slightly higher, and almost all of the improvement was made in the soft constraint penalties, specifically the penalties for violating the "minimum time between TVPs" constraint which makes up a large part of the soft constraint penalties in practice.

This behaviour makes sense. Because the greedy algorithm tries each day in the global plan window, it can find the "perfect" spots such that these mintime violations are minimized, especially for relatively small blocks where the actual plan date does not matter that much. The evolution strategy, on the other hand, would have to randomly move blocks such that the mintime violation is decreased or removed, as there are no mutations focused on removing mintime violations. The evolution strategy is however more capable of decreasing availability costs, due to the bucket system and the fact that many different options are tried.

This insight led to the idea to combine the two algorithms, in the hope that this allows both parts to exploit their strengths and cover each other's weaknesses. The two algorithms can be combined in multiple ways, such as, but not limited to

- Using the greedy algorithm to create a starting population for the evolution strategy, then running ES.
- Using a "greedy mutation" which takes a random block and re-plans it greedily.

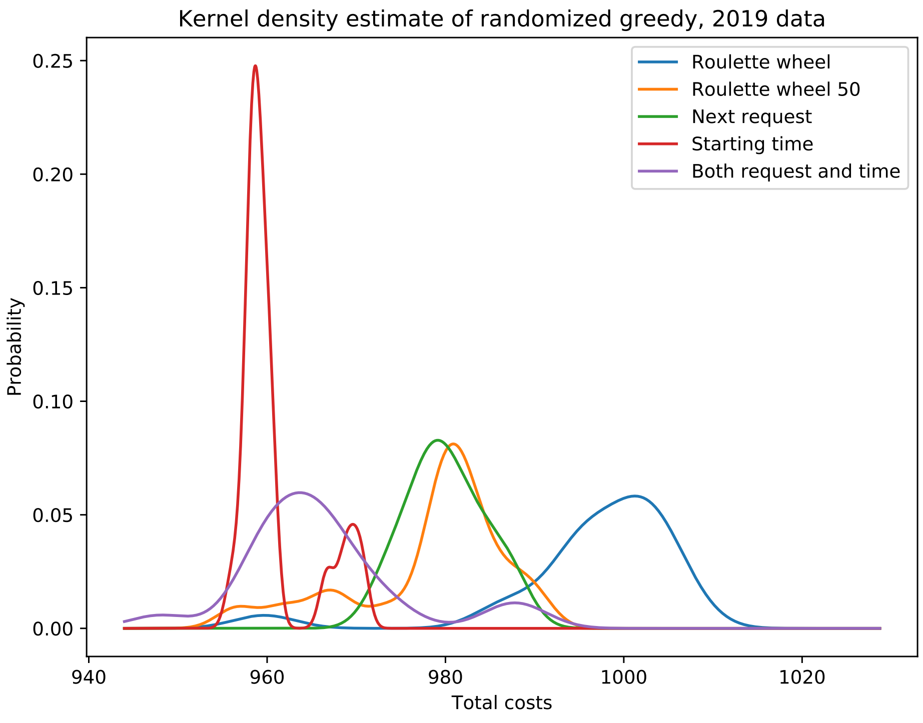


Figure 7.3: Kernel density estimation, costs only, 2019 data

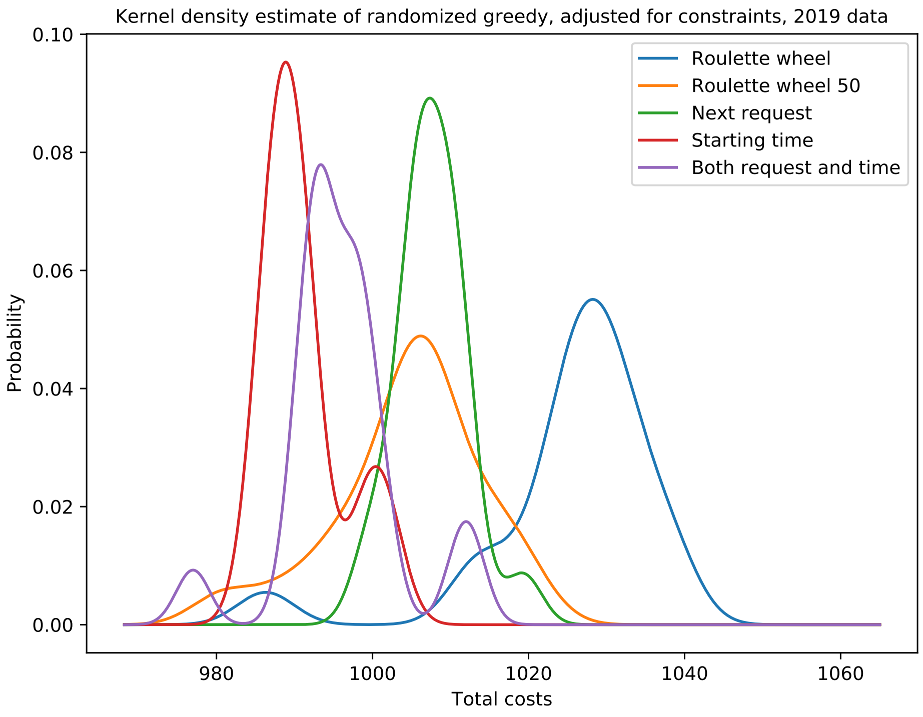


Figure 7.4: Kernel density estimation, two million added for each hard constraint violation, 2019 data.

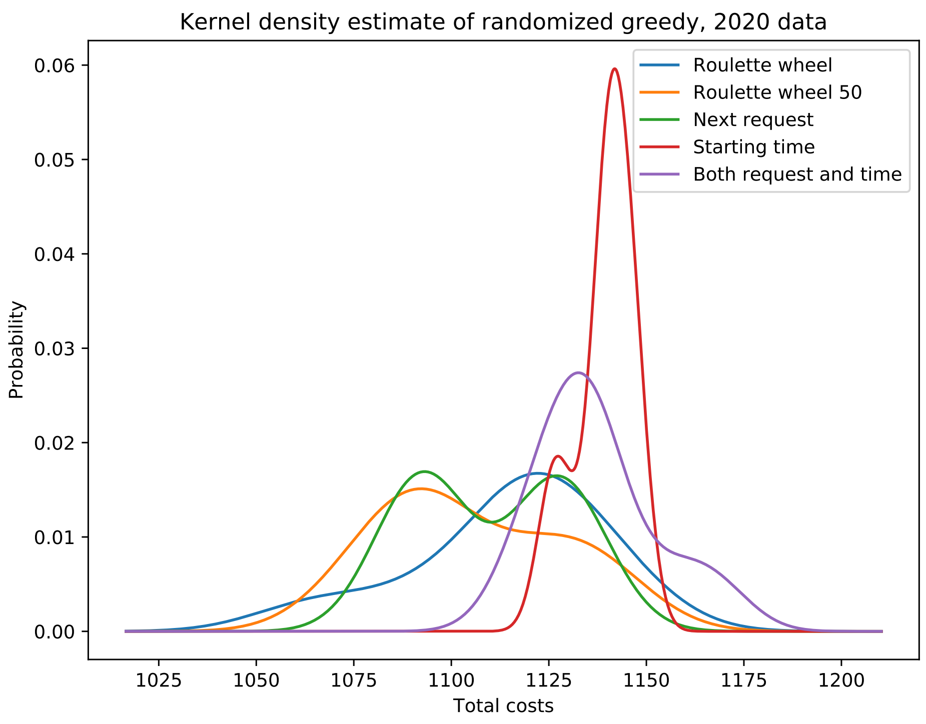


Figure 7.5: Kernel density estimation, costs only, 2020 data

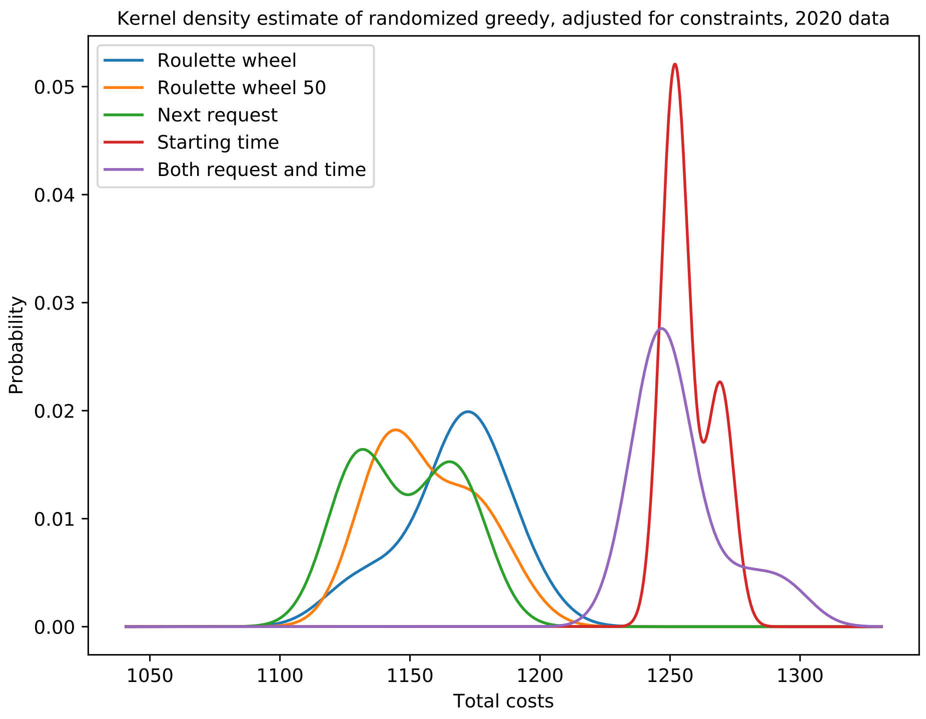


Figure 7.6: Kernel density estimation, two million added for each hard constraint violation, 2020 data.

- Using ES in between the greedy algorithm in an iterative manner; first plan x blocks greedily, run some ES, add some more blocks greedily, run more ES, etc. until all blocks are planned.

The second option was abandoned for now, because the proposed greedy mutation would take significantly longer than existing mutations; in the order of seconds instead of milliseconds. This would slow down the algorithm significantly and, although not tested, it is believed that this would not be outweighed by the possible improvements brought by the mutation.

The first and third options were both implemented and tested, and are discussed below.

7.4.1. Greedy individuals as starting population

The first method of combining the evolution strategy with the greedy algorithm is by creating the starting population for the evolution strategy by using solutions from the (randomized) greedy algorithm, rather than from the starting individual heuristic. A short test was performed using this method; ten individuals were created using the greedy algorithm with randomization in selecting the next request. Then, 1000 iterations of evolution strategy were run using these individuals as the starting population. To avoid getting stuck in a local optimum directly, the constraint cooling was set such that the constraint violations were allowed to rise a little at the start, and were forced back down again towards the end. This resulted in a solution with total costs equal to 959.3 with 10 hard constraint violations. This is approximately 14.5 fewer costs than the best result achieved so far, with the same number of hard constraint violations. As only 1000 iterations were run, the runtime is around 1-1.5 hours which is also a lot shorter than a "regular" evolution strategy run. This means that the envisioned result of combining the two algorithms is achieved.

Some further tests were performed to explore the possibilities of this hybrid method. In the first test, some more hard constraint violations were allowed in the greedy algorithm. This turned out to be significantly worse than the previous method; the costs of the starting individuals were not too much lower than before, and when the evolution strategy started solving the constraint violations the costs rose rapidly. A second test was done where the evolution strategy was swapped for a simulated annealing algorithm. Here, only a single greedy solution is generated and used as the starting point for the search. This method also turned out worse than when using the evolution strategy, confirming the earlier conclusion that the evolution strategy works better than simulated annealing on this problem.

7.4.2. Iterated greedy-ES hybrid

The third option as described above was also implemented. An initial test was performed, where first, ten individuals were greedily created with the fifty largest and most impactful blocks. The number fifty was chosen arbitrarily for this initial test, but is consciously chosen to be a small percentage of the total number of requests. These individuals were then used as a starting population in the evolution strategy which was run for 1000 iterations. After that, the best individual, still only with fifty requests planned, was taken, and the remainder of the requests was added greedily. This new individual was then copied ten times to create the starting population for the second stage of evolution strategy, also run for 1000 iterations. Finally, the best individual was returned as the solution.

This individual had total costs of 957.2 with 10 broken constraints. The runtime was approximately two hours. This is also a good improvement on the results of the evolution strategy and the greedy algorithm separately, and is also a further, albeit smaller, improvement over the results achieved by the hybrid version where the greedy individuals are used as starting individuals for a single stage of evolution strategy.

Due to the promising results of this hybrid algorithm in a relatively short runtime, a generic implementation was created to allow full customization. The number of stages can be specified, as well as the number of requests to be added greedily per stage and the generations or runtime of evolution strategy to use each stage. There is also an option on how to transfer between stages; either the method described above can be used, where the next batch of requests is only added to the best individual, and this individual is then copied to create the next starting population, or the next bath of requests is added to the complete population of the previous stage of ES to create the next starting population. The latter is obviously slower, but also promotes more diversity in the next population.

Using this generic set-up, various configurations were tested. Each of the configurations was run for approximately 24 hours. The runtime cannot be determined exactly beforehand due to the unknown time it takes to plan all requests in greedily. The total specified runtime of the evolution strategy stages was twenty hours in each run. After seeing the difference in results between the first two runs, it was decided to keep the number of stages low, and only vary the number of requests per stage and the runtime somewhat. Table 7.4

specifies the tested configurations, and table 7.5 shows the results.

Run	# stages	Requests per stage	Runtime per stage (hours)	Transfer method
1	2	50/983	12/8	Best individual only
2	10	15/20/30/50/80/100/100/120/250/268	2/2.5/2.5/2.5/2/2/2/1.5/1.5/1.5	All individuals
3	2	90/943	12/8	All individuals
4	2	20/1013	8/12	All individuals
5	3	20/70/943	7/7/6	All individuals
6	2	50/983	16/4	All individuals
7	2	50/983	4/16	All individuals

Table 7.4: Tested configurations of greedy-ES hybrid

Table 7.5 shows the results of the different runs

Run number	Total costs	Hard constraint violations
1	928.1	10
2	967.8	8
3	937.4	10
4	932.8	10
5	934.5	10
6	949.8	10
7	961.2	10

Table 7.5: Results of hybrid greedy-ES runs as specified in table 7.4

From the results, the following conclusions can be drawn:

- Compared to the short test run which already outperformed the previous algorithms, another large cost improvement of approximately 29 was found. This further proves the strength of this hybrid algorithm, and shows that an increased runtime is worth it in terms of results. Table 7.6 shows the best results encountered so far of the improved evolution strategy, the randomized greedy algorithm, and the hybrid greedy-ES algorithm. A substantial cost reduction is made by the hybrid algorithm while not increasing hard constraint violations.

- A configuration with two or three stages works much better than a configuration with ten stages.

- For a configuration with two stages, the number of requests in the first stage does not matter much if it stays within reasonable bounds. The run with 50/983 did perform best, but is closely followed by the runs with 90/943, 20/1013 and 20/70/943. These differences are small enough to be caused by variance in the evolution strategy part of the algorithm.

- Using a near equal runtime for the two or three stages is better than giving one stage much more runtime than the other, as seen by the inferior results of the 16 hour/4 hour and 4 hour/16 hour runs.

Algorithm	Total costs	Constraint violations
Evolution strategy	972.9	10
Greedy	947.9	12
Hybrid greedy-ES	928.1	10

Table 7.6: Best results encountered of ES, greedy, and hybrid greedy-ES

7.4.3. Concluding remarks

In this chapter, two greedy algorithms were designed to solve the problem. These algorithms have a much shorter runtime than the metaheuristic algorithms described in chapter 4, and provide solutions which are of almost the same quality on average. After noticing that the greedy algorithm and the evolution strategy were strong in different areas of the problem, they were combined into a hybrid algorithm. This algorithm proved to be very successful, improving the results substantially compared to both of its counterparts.

8 Further improvements and experiments for the evolution strategy

The improvements described in chapter 4 and the parameter optimization shown in chapter 6 have clearly improved the baseline algorithm. At this point, two possible areas of further improvement have been identified:

- Overuse of mutations that are specifically focused on one specific part of the solution, such as creating buckets or fixing conflicts, can negatively impact the results. Using these mutations can have little to no effect in certain situations, therefore being a waste of runtime.
- The soft constraint penalties consist primarily of penalties for violations of the minimum time between two TVPs in a corridor. (Manual) observation of the results indicated that there were often plenty of such violations that were easy to fix by simply moving one of the TVPs to lie directly subsequent to the other. However, the evolution strategy does not find these cost reductions easily, as it would have to randomly move one or more TVPs to that exact starting time.

Furthermore, it has been observed that the results for the evolution strategy can differ quite substantially, even when using the same parameter values. Therefore, a robustness analysis has been performed, in order to get a better insight into the distribution of solutions of the evolution strategy.

Finally, the trade-offs between maintenance and availability costs, and soft and hard constraints have been explored somewhat using the multi-objective algorithms in section 5. However, the results of these algorithms were not satisfactory and therefore, these trade-offs need to be investigated further using better-performing (single-objective) algorithms.

In this chapter, all of these points are addressed. In section 8.1, a robustness analysis is performed for the evolution strategy algorithm. Section 8.2 describes an adaptive mutation which tries to address the first point of improvement by making smarter choices on which mutation to use at what point. Section 8.3 describes a mutation which is tailored to solve violations of the minimum time between two TVPs, to try and decrease the soft constraint penalties of solutions obtained by the evolution strategy. These two improvements are combined and tested in section 8.4. Validation of these two mutations is done using the 2020 data in section 8.5. In section 8.6 the trade-off between maintenance and availability is investigated further. Finally, in section 8.7 the trade-off between costs and constraints is explored in more detail.

The indicated improvements are tested using the stand-alone evolution strategy algorithm. If proven to be effective, they can also be used in the hybrid algorithm which was described in section 7.4. However, the indicated shortcomings are not as large in the results of the hybrid algorithm, so any improvement is likely to be mitigated when used in the hybrid algorithm. The improvements are not tested on the hybrid algorithm specifically; this is left as future research.

8.1. Robustness analysis

A robustness analysis will provide insight in the distribution of solutions, under the exact same parameter conditions. Obviously, due to the high amount of randomness, the solutions will not be the same in each run. However, it is desired that the solutions lie close and getting a good result does not rely on luck.

Unfortunately, the runtime necessary to get a representative result is quite long, and the remaining time and computing power for experiments were limited. Regardless, a total of 53 runs have been completed, a good enough sample size to make some conclusions about the robustness of the algorithm.

Each run was run for a total of 12000 generations and no changes were made to any parameters in between the runs. The full results can be found in appendix B. Figure 8.1 shows the solutions in terms of total costs and constraint violations.

It can be seen that there is quite some spread in the solutions. In 35 out of the 53 runs, ten hard constraints were broken. However, the relation between hard constraints and total costs is not as expected. One would expect that solutions with higher constraint violations have lower costs, but this is not the case. It seems that sometimes, the algorithm simply converges to a bad local optimum where it is unable to solve the final few constraint violations, but also does not have lower costs.

Figure 8.2 shows a kernel density estimate of the costs of the solutions from the robustness analysis. This gives an idea of the probability distribution of the costs of a run of the evolution strategy algorithm. The average costs are 986.1 with a standard deviation of 8.71. The distribution resembles a normal distribution, which is also confirmed by an Anderson-Darling test for normality [14], which does not reject the null hypothesis that the data is randomly distributed under a significance level of 0.05. This means that approximately 68% of all solutions will fall within one standard deviation of the mean, so in the range [977.38, 994.80], and approximately 95% will fall within two standard deviations or the interval [968.67, 1003.5]. Unfortunately, these ranges are reasonably wide, making it somewhat harder to compare two versions of the evolution strategy with a single run.

Since the spread of solutions is somewhat wide, the possibility of early restarting was investigated. If there is a clear correlation between the costs at an earlier point in the algorithm and the final result, a potentially bad run could be stopped early and a restart could be performed. Figure 8.3 shows the costs over time for all runs in the robustness analysis, with some specifics highlighted. The average, max and min lines show the average, minimum and maximum values at each point in time, meaning the source of the minimum and maximum values are not necessarily from the same run the whole time. The lines for the solutions with the best and worst end result are also highlighted. These runs show that it may be hard to implement such an early restart mechanism. After 2000 iterations, both of these runs have high costs and are close together. Around 7000 iterations which is already quite far into the total runtime, the two runs are both in the middle of the pack and there is still almost no difference between them. Only in the final stages, the quality of the runs separates. This conclusion is supported further by a computation of the correlation between the intermediate costs and final costs at various points in the algorithm. For this, the Pearson correlation coefficient was computed between the vector containing the final costs, and the vector of corresponding intermediate points. Table 8.1 shows these results.

Number of iterations	Pearson correlation coefficient
1000	0.0891
2000	0.1603
3000	0.1134
4000	0.1204
5000	0.1431
6000	0.2365
7000	0.3484
8000	0.5396
9000	0.6374
10000	0.8814
11000	0.9391

Table 8.1: Pearson correlation of intermediate costs versus final costs

It becomes clear that only from 9000 iterations onward, reasonable conclusions about the final result could be drawn. At that point, it is not worth it to restart since three-quarters of the total runtime would have been elapsed already anyway. Overall, it is concluded that it is near impossible to predict the results of an evolution strategy run in the early stages and that the full run would have to be completed to find out the result. If possible, multiple runs should be done to have a higher chance of finding a result with below average costs, and to get a good idea of the distribution when a change is made to the algorithm.

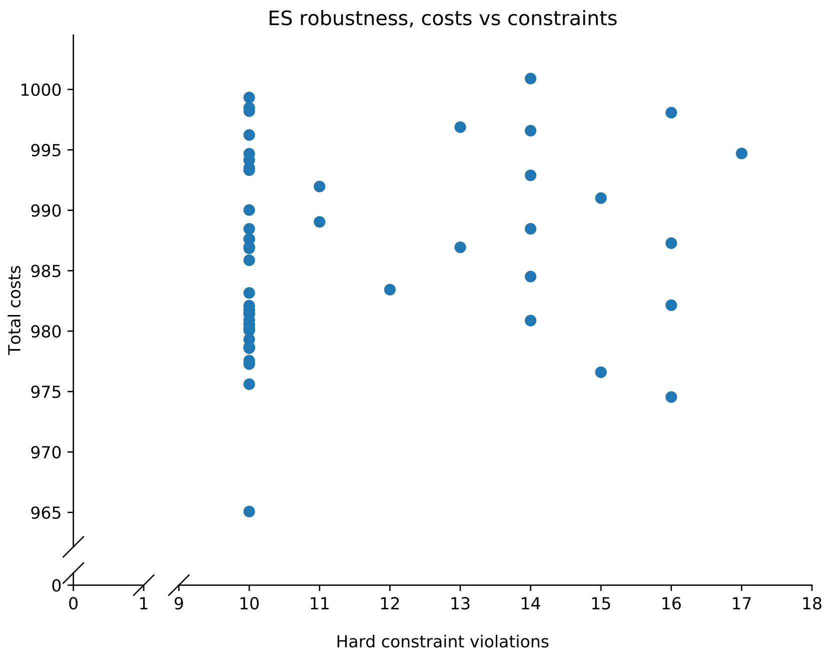


Figure 8.1: Robustness analysis ES, costs versus constraints

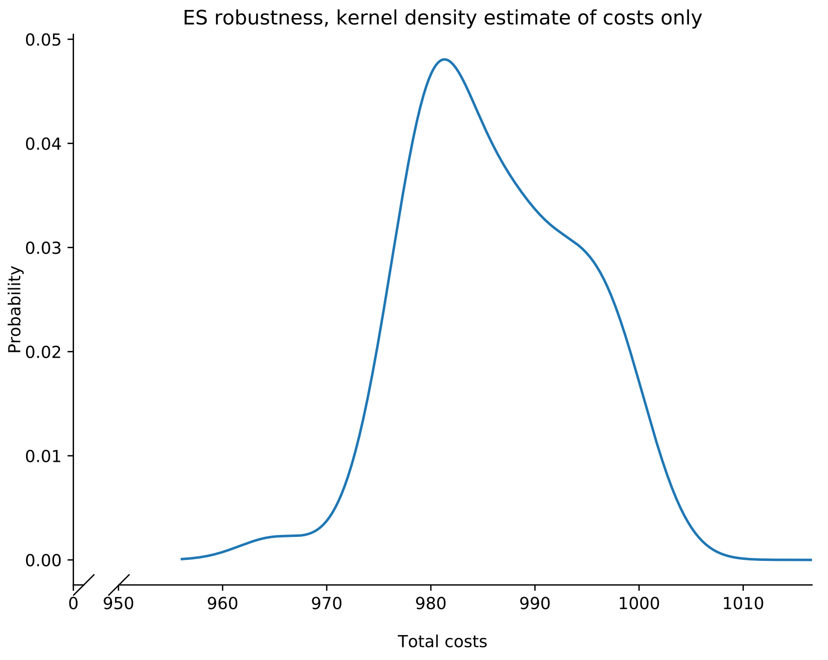


Figure 8.2: Kernel density estimate of costs only, ES robustness analysis

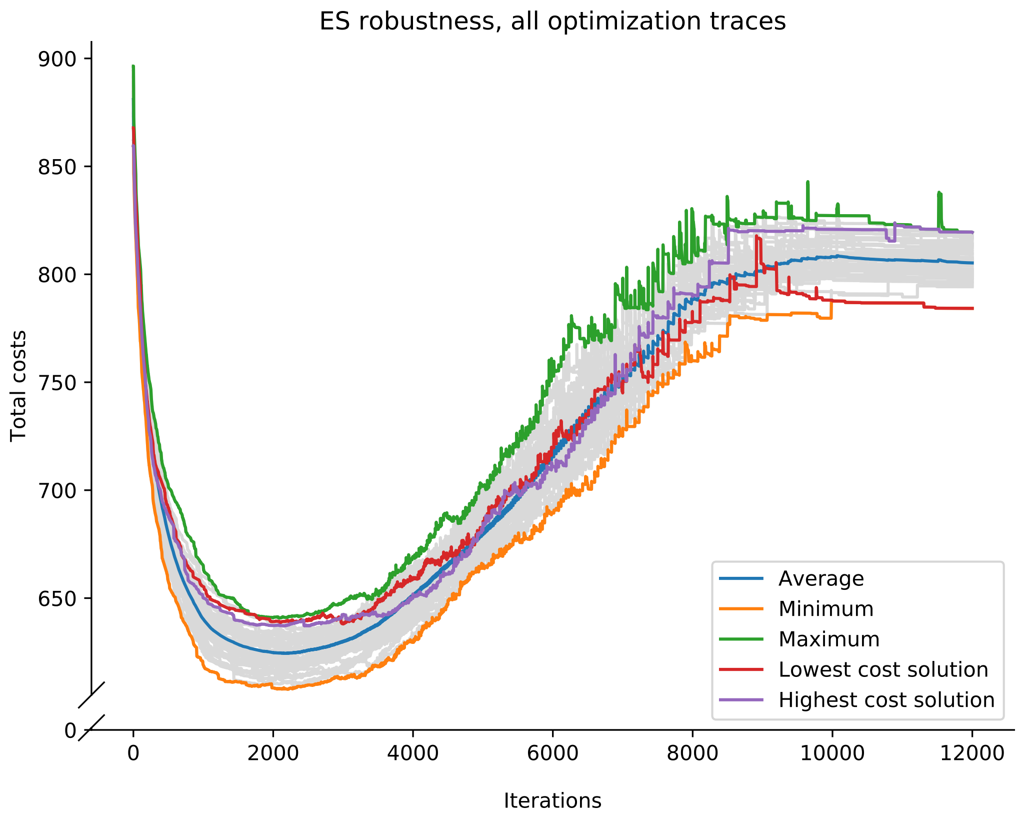


Figure 8.3: Optimization trace of all runs in the ES robustness analysis

8.2. Adaptive mutation

As indicated by the parameter analysis, it may be disadvantageous to use certain mutations in situations where they would have little to no effect. However, simply decreasing the weight for these mutations would not be a good way to solve this problem, as this would lead to an underutilization of the given mutations in situations where they are useful. This led to the implementation of an *adaptive* mutation. This mutation adapts the weights of different mutation groups based on the situation at hand.

The adaptive mutation works with the same three groups as used in the parameter analysis: shift mutations, bucket mutations and fix mutations. A default weight for each group can be specified. Then, the weights are adapted given the situation, as follows:

- The weight for shift mutations will always stay the same. However, its relative importance will change when the weights for the other mutation groups change.

- The weight for the bucket mutations will change based on the number of candidate pairs that are left to create buckets. When this number is high, more bucket mutations will be performed as there are still many opportunities to create and expand buckets. When this number is low, fewer bucket mutations will be performed as the effect of such a mutation would be low to none in many situations.

- The weight for the fix mutations will change based on the number of allowed constraints and the actual number of constraints. When the individual is infeasible, the weight for fix mutations increases. When the individual is feasible, the weight changes inversely proportional to the gap between the allowed violations and the actual violations.

This method should prevent many useless mutations being performed in the latter stages of the algorithm and is therefore expected to give better results.

8.2.1. Testing the adaptive mutation

A total of five runs were performed with the adaptive mutation. All were run for 24 hours, and other parameters were not changed in comparison with the robustness analysis. Table 8.2 shows the quality of the achieved solutions. Table 8.3 shows the averages of the solutions with and without the adaptive mutation.

Run number	Total costs	Constraint violations
1	985.6	10
2	984.7	10
3	982.0	10
4	976.1	10
5	979.1	17
Average	981.5	11.4

Table 8.2: Results of evolution strategy with adaptive mutation

Algorithm	Total costs	Constraint violations
Evolution strategy, average from robustness analysis	986.1	11.4
Evolution strategy with adaptive mutation	981.5	11.4

Table 8.3: Average result of evolution strategy with and without adaptive mutation.

Although the number of runs is limited, the adaptive mutation looks to lead to a small improvement. In the robustness analysis, the average costs were 986.1. The average of the runs using the adaptive mutation is 981.52. Furthermore, all the runs with the adaptive mutation have lower costs than the average of the runs without it.

Figure 8.4 shows a probability density estimate of both the runs with and without the adaptive mutation. The averages of both densities are shown as a dotted vertical line.

As expected, the average of the density of the runs with the adaptive mutation is lower than that of the runs without it. Furthermore, the density of the runs with the adaptive mutation is narrower, indicating a somewhat higher robustness. This makes sense; by choosing a useful mutation more often, the algorithm is less susceptible to an unlucky streak of mutations that are not useful, and will converge to a more optimal point. On the other hand, this may also mean that points with low costs will not be found very often. This may be a negative result if the algorithm can be run many times and only the best result is cared about. However, only three out of the 53 runs without the adaptive mutation achieved lower costs than the lowest run with the adaptive mutation. Furthermore, the sample size is small and it is very possible that a lower cost solution can still be found using this method. This low sample size also decreases the certainty of conclusions drawn above; it is possible that the robustness and/or average solution quality are not represented well by these runs.

Looking at the hard constraints, the results are quite similar to the runs without the adaptive mutation in the sense that most runs achieve ten hard constraint violations, and sometimes it happens that the constraint violations remains slightly higher.

Overall, it can be concluded that the adaptive mutation is a useful addition to the algorithm. Although the benefits in terms of costs are relatively small, the increased robustness is also a positive aspect of the adaptive mutation and therefore, the adaptive mutation will be kept in as an improvement to the evolution strategy.

8.3. Mintime violation mutation

In section 7.4, a weak point of the evolution strategy was already identified. The algorithm is quite bad in optimizing the necessary minimum time between TVPs, which is a soft constraint in the base constraint scenario. Because, in the current setup, the penalties for this specific soft constraint make up more than 85% of the total soft constraint penalties, making the algorithm more capable to solve this constraint will benefit the solution quality significantly. The results of the greedy and hybrid greedy-ES algorithms indicate that there are indeed possibilities to decrease the soft constraint penalties.

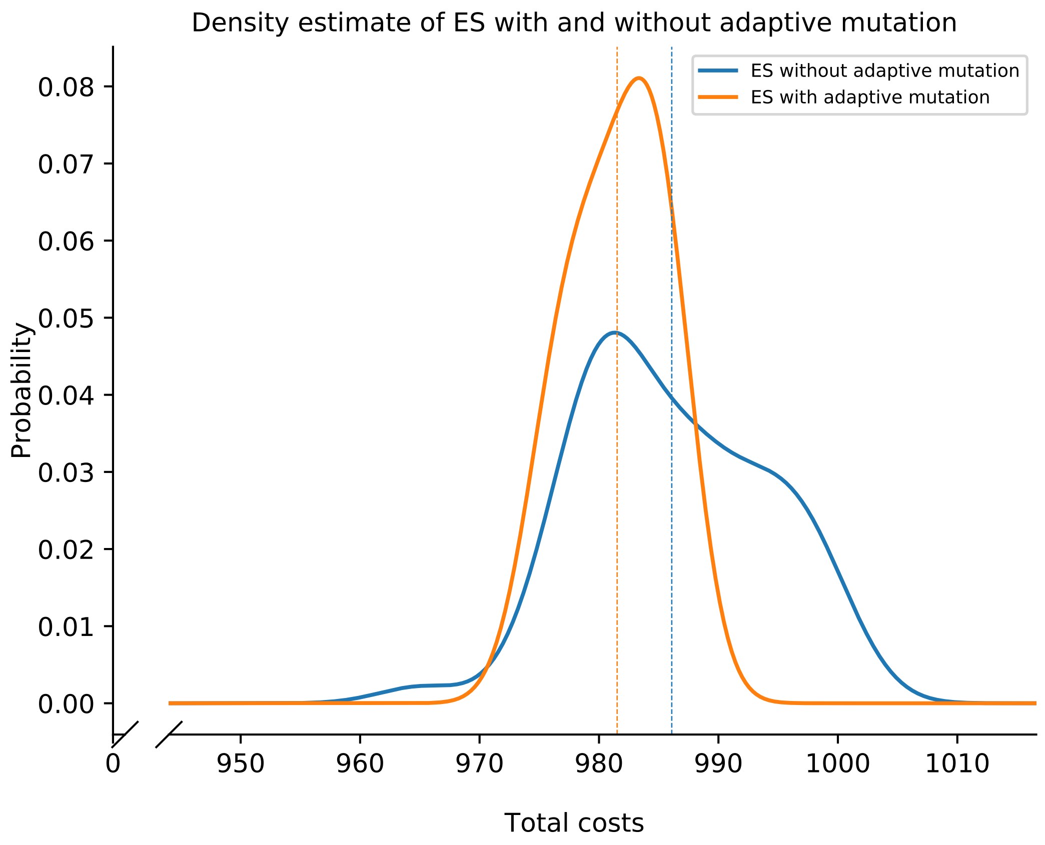


Figure 8.4: Probability density estimate of ES with and without adaptive mutation

To try and tackle this issue, a new mutation has been added. This mutation is specifically focused on fixing mintime violations. It works by finding two TVPs which currently cause a mintime violation. Remember that a TVP in this context is an overlapping period of at least 24 hours long, on the corridor level. TVPs and overlapping periods are explained in detail in section 4.1.3.

After the TVPs are selected, the violation is fixed by closing the gap; either the latter TVP is moved backwards to start exactly when the other TVP ends, or vice versa, with equal probability. This means that the TVPs will now be combined to a single TVP and this specific mintime violation will disappear with 100% certainty. Of course, it is possible that other constraints get broken or costs rise, but this is not taken into account in this mutation. If the fix results in a worse solution this will simply be handled by the evolution strategy not selecting it for the next generation and in the next generation, another pair may be fixed with more success.

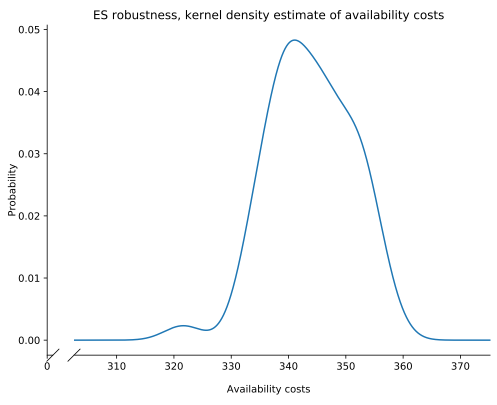
A parameter for minimum violation grade can be specified so that only the most severe violations are solved by this mutation. It should be kept in mind that the more severe the violation, the easier it is to fix on average because the TVPs will be closer together already and the resulting change will be relatively small.

8.3.1. Testing the mintime mutation

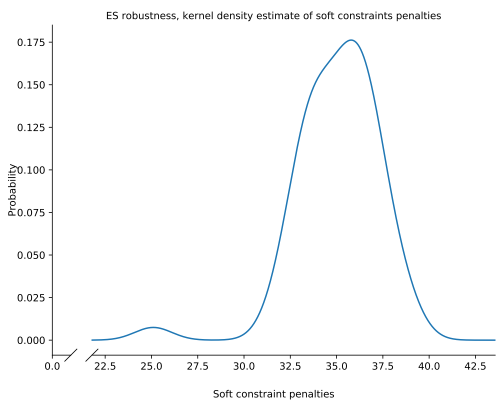
To test the mintime mutation and specifically its effect on the soft constraint penalties, first, some information on the values for availability costs and soft constraint penalties in the robustness analysis must be given. Figure 8.5 shows kernel density estimates for the availability costs and soft constraint penalties of the solutions from the robustness analysis. As expected, the density for the availability costs is very similar to that of the total costs. The average availability costs are 343.8. The density for soft constraint penalties is somewhat narrower, and the average penalty is 35.1.

The mintime mutation has been added to the algorithm. The other mutations have not changed and are all still present. A total of five runs were performed, all with a runtime of 24 hours. The results are given in table 8.4. Table 8.5 shows the average result of the runs with and without the mintime mutation.

The average total cost of the runs with the mintime mutation added is 975.83, 10.26 lower than the average



(a) Kernel density estimate of availability costs



(b) Kernel density estimate of soft constraint penalties

Figure 8.5: Kernel density estimates of availability costs and soft constraint penalties of solutions from robustness analysis in section 8.1
.

Run	Total costs	Hard constraints	Maintenance	Availability	Soft constraint
1	968.7	10	607.3	334.3	27.1
2	975.9	10	607.1	345.3	23.2
3	968.0	10	605.5	336.8	25.6
4	982.2	14	607.3	349.5	25.4
5	984.4	10	607.1	351.3	26.0
Average	975.8	10.8	606.8	343.5	25.5

Table 8.4: Result of evolution strategy with mintime mutation added

of the runs without it. It can also be seen that this difference is clearly in the soft constraint penalties. The average soft constraint penalty of the runs with the mintime mutation is 25.53, compared to 35.05 for the runs without. The average for the maintenance and availability costs is approximately the same over the two groups. This indicates that the mutation greatly reduced the soft constraint penalties, and did not negatively impact the other two cost groups. The mutation is therefore concluded to be a beneficial addition and will be kept in the algorithm.

8.4. Combining the mintime and adaptive mutations

Since both the mintime and adaptive mutation showed improvement over the baseline evolution strategy when added separately, it is also interesting to see how they work together. Therefore, an experiment has been performed in which both mutations have been added. The mintime mutation has been added to the "shift mutations" group of the adaptive mutation. Even though this mutation fixes a constraint, the "fix mutations" group should be used specifically for mutations that fix hard constraints, as only the hard constraints are taken into account when determining the weight for this group. Adding the mintime mutation to the fix mutations group could have the undesired effect of the mutation not being used at the end of the algorithm when the solution is already feasible, but there is still improvement possible in the soft constraints.

As before, five runs have been made, all of which have run for 24 hours. Table 8.6 shows the results. Table 8.7 shows the comparison of the ES without the mutations, both mutations added separately, and both mutations added.

Looking at the constraint violations, there are now two out of five runs which have more than ten constraint violations compared to one out of five for both the mintime and adaptive mutation tests. Figure 8.6 shows a kernel density estimate of four solution sets: the regular ES, the ES with mintime mutation, the ES with adaptive mutation, and the ES with both the mintime and the adaptive mutation.

It can be seen that the solutions with both mutations added do have lower costs on average than both of its counterparts as well as the original ES. Although the minimum cost solution of the mintime mutation only is better than that of the solutions with both mutations, using both mutations provides a more robust

Algorithm	Total costs	Constraint violations
Evolution strategy, average from robustness analysis	986.1	11.4
Evolution strategy with mintime mutation	975.8	10.8

Table 8.5: Average result of evolution strategy with and without mintime mutation.

Run number	Total costs	Hard constraints	Maintenance	Availability	Soft constraints
1	971.4	17	607.9	340.0	23.5
2	966.6	10	605.6	338.2	22.8
3	971.3	10	608.1	338.6	24.6
4	970.5	10	606.7	339.6	24.3
5	977.0	16	606.3	347.7	22.9
Average	971.4	12.6	606.9	340.8	23.6

Table 8.6: Results of evolution strategy with both mintime and adaptive mutations added

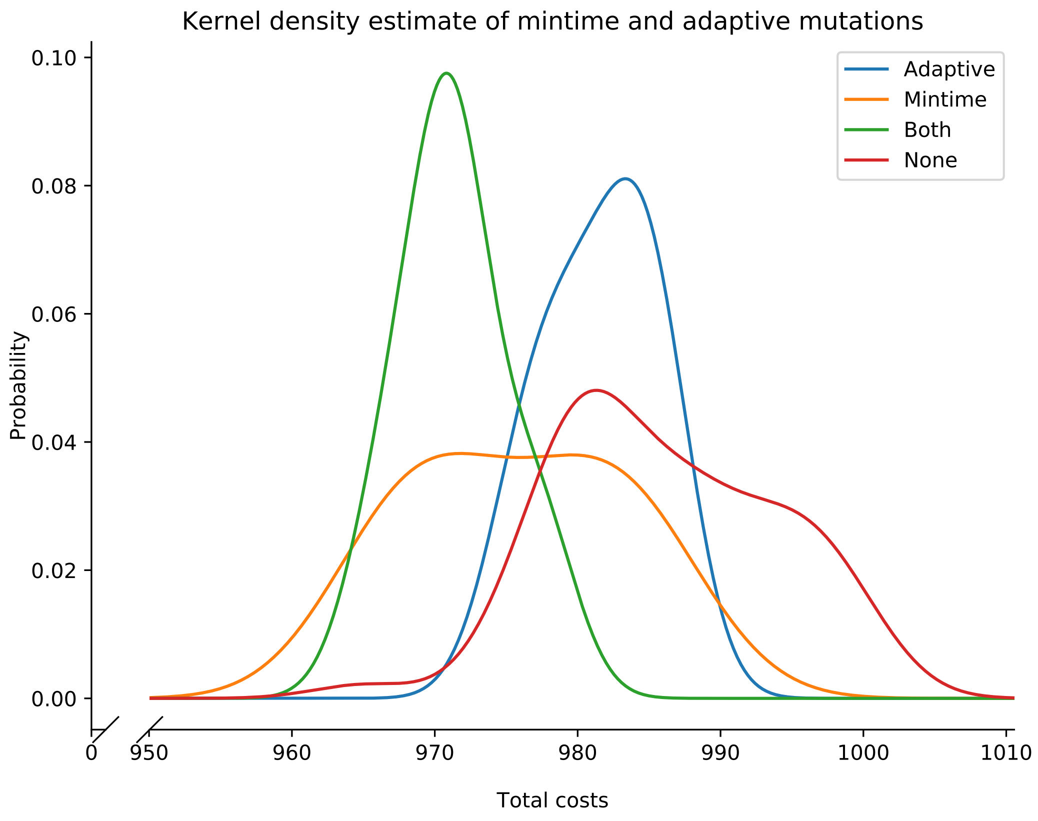


Figure 8.6: Kernel density estimate of various configurations of the evolution strategy

Algorithm	Total costs	Constraint violations
Evolution strategy, average from robustness analysis	986.1	11.4
Evolution strategy with adaptive mutation	981.5	11.4
Evolution strategy with mintime mutation	975.8	10.8
Evolution strategy with mintime and adaptive mutations	971.4	12.6

Table 8.7: Comparison of evolution strategy with and without adaptive and mintime mutations

Configuration	Costs	Constraints	Maintenance	Availability	Soft constraints
Regular 1	1092.6	27	409.4	632.0	51.2
Regular 2	1101.5	27	410.0	639.5	52.1
Regular 3	1092.4	34	408.6	629.3	54.4
Regular 4	1086.7	25	407.3	629.2	50.2
Regular 5	1102.9	25	409.2	640.3	53.3
Mintime 1	1091.9	29	410.3	647.7	33.9
Mintime 2	1089.7	26	408.9	649.9	30.9
Mintime 3	1096.5	25	409.6	654.7	32.2
Mintime 4	1096.8	26	408.6	648.1	40.1
Mintime 5	1077.6	28	408.2	638.7	30.6
Adaptive 1	1110.2	23	409.8	647.3	53.1
Adaptive 2	1076.8	26	408.4	613.0	55.3
Adaptive 3	1126.6	26	409.0	663.4	54.2
Adaptive 4	1131.4	24	409.9	668.0	53.5
Adaptive 5	1169.6	27	409.4	704.4	55.8
Both 1	1113.4	25	408.0	680.2	25.2
Both 2	1086.5	28	408.2	651.2	27.1
Both 3	1089.2	25	409.2	654.0	25.9
Both 4	1079.7	26	407.7	648.7	23.3
Both 5	1023.7	34	408.5	588.6	26.6

Table 8.8: Results of mintime and adaptive mutations on 2020 data

algorithm with the lowest average costs. Although the small sample size should be kept in mind, it can be concluded that the combination of both the mintime and adaptive mutations works well and both mutations should be added to the algorithm.

8.5. Validation of mintime and adaptive mutations

In the previous sections, the mintime and adaptive mutations have been tested on the 2019 data. To ensure the results and conclusions still hold on other datasets, some runs have been done on the 2020 data as well in order to validate the results. As before, each run was done for 24 hours. Five runs have been done for each configuration, giving a total of 20 runs. Table 8.8 shows the results of each of the runs.

Figure 8.7 shows a kernel density estimate of the total costs of each of the four groups.

Some interesting results can be seen. First of all, the solutions for the "regular" and "mintime" configurations are much more robust than the solutions of the other two groups. For the 2019 data, this result was opposite.

The results of the adaptive mutation are quite bad compared to the regular evolution strategy, whereas on the 2019 data it gave a small improvement. This is probably due to the fact that the adaptive mutation required quite a lot of parameters which were set specifically for the 2019 data to work well, and were not changed. Next to the fact that these parameters should have been changed for the 2020 runs, it would be better to have an algorithm or improvement that works well without changing anything, regardless of input data. This is clearly not the case for the adaptive mutation. Therefore, the adaptive mutation has been slightly adjusted where certain parameters are now automatically determined based on the input data, rather than the user having to set them manually. The runs for "both" mintime and adaptive mutation have been run using this new adaptive mutation and, although the robustness is still not good, the average result is much more in line with the "regular" and "mintime" groups.

Comparing the regular evolution strategy with the one with the mintime mutation added, it can be seen that the results are quite similar, and the mintime mutation gives a small improvement on average. This improvement is smaller than on the 2019 data. Looking at table 8.8, it can be seen that there seems to be a larger trade-off between soft constraints and availability in the 2020 data than in the 2019 data. Using the mintime mutation pushes down the soft constraints significantly, as seen on the 2019 data as well, but this causes a significant rise in availability costs.

Overall, the following conclusions regarding these two mutations can be made:

- The mintime mutation gives an improvement for both datasets, albeit smaller for 2020 than for 2019, and it should therefore be used in the algorithm.
- The adaptive mutation was very sensitive to input data. This seems to have improved after removing some manual parameters, but the results on the 2020 data are not entirely convincing. Combined with the fact that the adaptive mutation only gave a small improvement on the 2019 data, more testing is needed to determine whether this adaptive mutation is a useful improvement. For now, it will not be included in the algorithm.

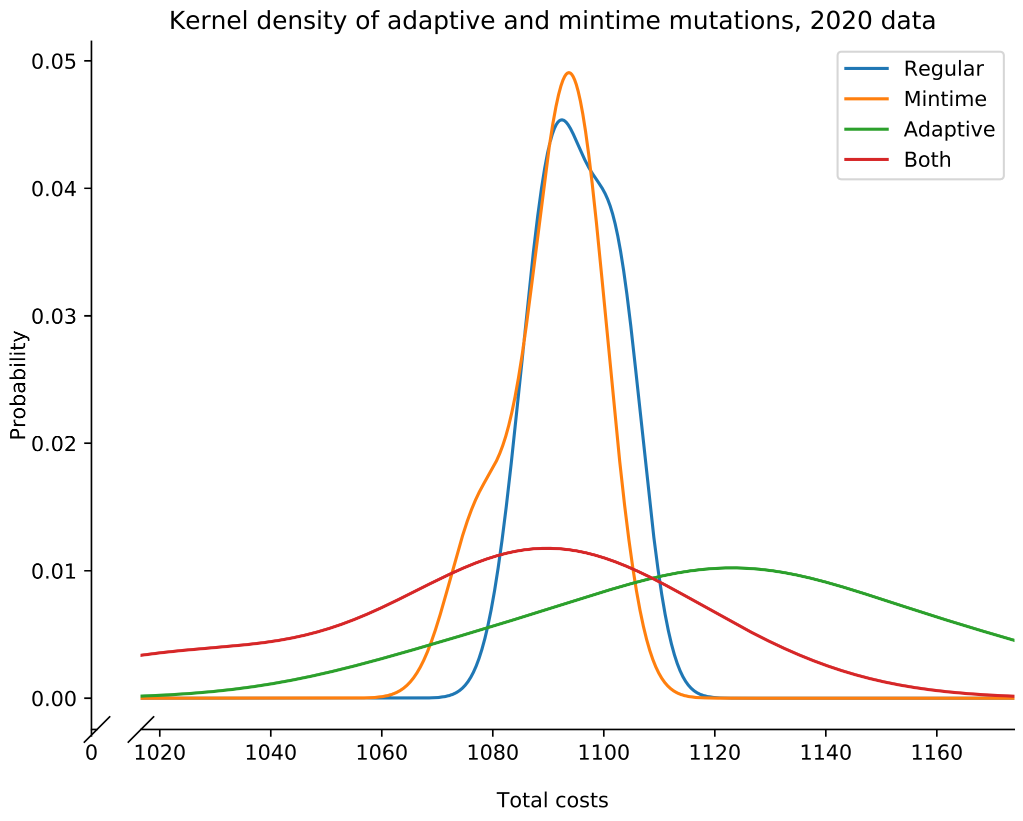


Figure 8.7: Kernel density estimate of mintime and adaptive mutations on 2020 data

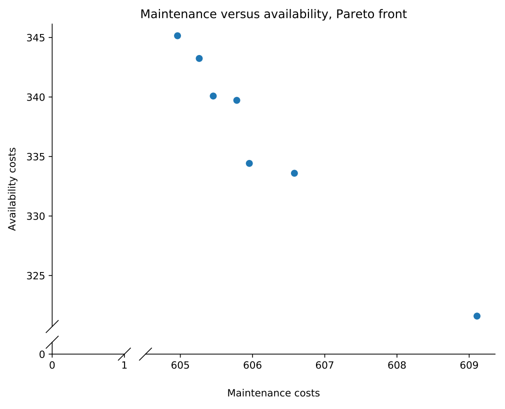
8.6. Trade-off between maintenance and availability

One of the areas that was identified as a possible interesting research direction was the trade-off between maintenance and availability costs. In chapter 5, this trade-off was already explored using multi-objective algorithms. Unfortunately, the results of these algorithms were not satisfactory; the total costs were too high and the diversity of the discovered solutions was too low.

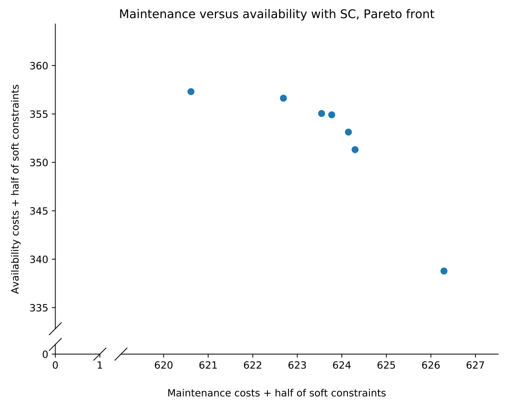
However, the trade-off can also be explored in other ways. First of all, the robustness analysis has provided a set of solutions from which a Pareto front can be computed. These solutions were not obtained by doing multiple runs and changing the weighting of the two components, but are rather a result of a suboptimal result in terms of the total costs being a point that, when considering Pareto dominance as a metric instead, belongs to the Pareto front, usually because it has slightly lower maintenance costs than a solution that is more optimal in terms of total costs. The solutions achieved in this analysis are not optimal but can be used to get a clearer picture of this trade-off.

8.6.1. Pareto front of the evolution strategy robustness

To create the Pareto front, the solutions with ten constraint violations have been selected from all solutions in the robustness analysis. From these solutions, the nondominated points have been computed. In figure 8.8, two Pareto fronts are shown. The first, figure 8.8a, shows the Pareto front when only considering maintenance and availability costs and discarding the soft constraint penalties completely. Figure 8.8b shows the Pareto front when the soft constraints are taken into account by adding half of them to both objectives.



(a) Pareto front of ES, soft constraints not included



(b) Pareto front of ES, soft constraints included

Figure 8.8: Pareto fronts of evolution strategy

Some interesting conclusions can be drawn. First, when looking at the Pareto front without soft constraints, it is seen that the relation between the availability costs and maintenance costs is somewhat linear. As was already seen before, the availability costs rise much faster than the maintenance costs decrease, so when considering total costs, decreasing availability costs is the way to go. This is also in line with the way maintenance is usually planned by ProRail in the Netherlands. Maintenance is almost always planned during the night, weekend, or holiday, and almost never during the busiest hours.

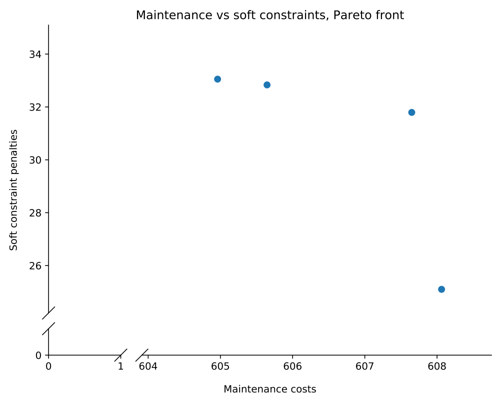
A note has to be made here regarding the data being used to test with. In the real 2019 data, 578 of the 1033 project requests do not have any maintenance cost at all due to incomplete data delivered by ProRail. This does mean that for these project requests, there is no chance at all to increase or decrease the maintenance costs, therefore limiting the variability in maintenance costs when it comes to the trade-off with availability. It is hypothesized that, when the data is filled in completely, focusing on decreasing the availability costs will create a larger increase in maintenance costs.

Looking at the Pareto front with soft constraints included in figure 8.8b, a clear difference can be seen compared to the other Pareto front. The relation is much less linear, and the trade-off between availability and maintenance behaves differently at different parts of the Pareto front. At low maintenance costs, an increase only creates a small decrease in availability costs. When the maintenance costs rise further, the availability costs start to decrease much faster.

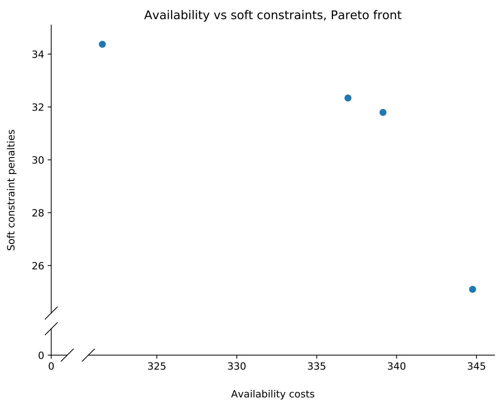
This difference in Pareto fronts between including or excluding soft constraints indicates there is also some kind of trade-off going on between soft constraints and either maintenance costs or availability costs, or both. To investigate this, a Pareto front has been plotted for both of these relations. They can be seen in figure 8.9.

It can be seen that there is indeed a trade-off between both cost types and soft constraint penalties. For the maintenance costs, this trade-off is small. The point at the bottom right of figure 8.9a, although not clearly visible here, is an outlier when looking at all data points. Most data points follow the small trade-off of the first three points. It is therefore not expected that this trade-off causes the differences visible between figures 8.8b and 8.8a.

Looking at the Pareto front for soft constraint penalties and availability costs in figure 8.9b, the trade-off is larger and follows a more interesting pattern. When the availability costs are high, a relatively small decrease can cause a relatively large increase in soft constraints penalty. When the availability costs keep decreasing, the increase in soft constraint penalties decreases. This behaviour also explains the difference between the Pareto fronts of maintenance and availability costs, with and without soft constraint penalties. Here, it was observed that the decrease in availability was smaller when those availability costs were still high. This is



(a) Pareto front of maintenance costs vs soft constraints



(b) Pareto front of availability costs vs soft constraints

Figure 8.9: Pareto fronts of soft constraint penalties with two cost types

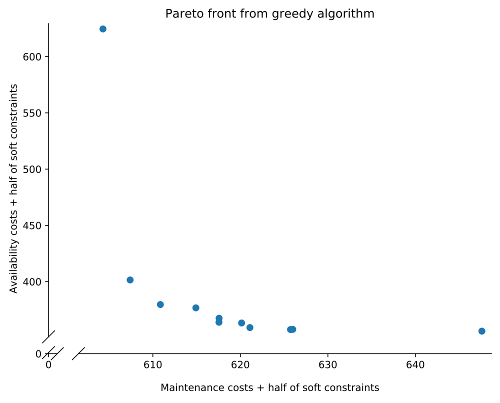
possibly due to the fact that at that point, the soft constraints increase quite fast, mitigating the decrease of availability.

8.6.2. Pareto front with the greedy algorithm

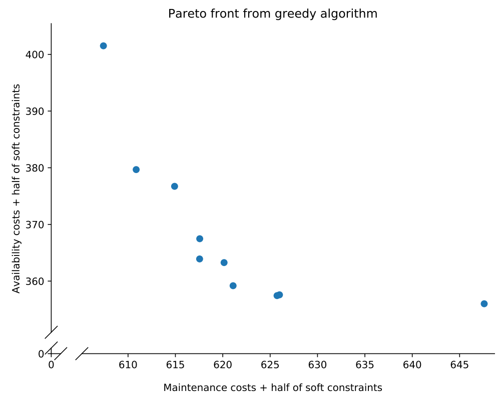
The solutions from the evolution strategy robustness analysis used to create the Pareto fronts in the previous section were all run with equal weights for both objectives and their values are simply due to randomness. It would be interesting to see what happens if the actual weights used in the objective function change. This way, the extreme values of only maintenance and/or availability can also be found.

For this, the greedy algorithm has been used. The greedy algorithm can quickly provide a solution, and it will optimize any weighted function of maintenance and availability automatically since it will just pick the option with the lowest total (weighted) costs. A simple linear weighting has been used, and the soft constraint penalties are always included in full. The deterministic greedy algorithm has been used. A total of eleven runs have been performed, with weights ranging from 0.0 to 1.0 with step 0.1, and total weight always equal to 1.0. The first experiment was done using the 2019 data.

Figure 8.10 shows the generated Pareto front. Figure 8.10a shows the full Pareto front, figure 8.10b shows the front with the first point removed so the differences between the other points can be seen easier.



(a) Pareto front of the greedy algorithm



(b) Pareto front of the greedy algorithm, first point removed

Figure 8.10: Pareto front of the greedy algorithm. Right plot with the first point removed for better scaling for the other points

The greedy algorithm was run without any allowed hard constraints meaning all solutions have five to seven hard constraint violations. This also means they have higher costs than the solutions of the evolution strategy runs, as they mostly have ten hard constraint violations.

The results are mostly as expected. When not taking account availability at all these will rise very far, but as soon as they are taken into account somewhat the costs stay within reasonable bounds and a clear trade-off is taking place. The availability costs still decrease faster than the maintenance costs increase, although the difference looks to be less than for the Pareto front generated from the evolution strategy runs. This could be due to the fact that a lower number of hard constraint violations allows for less freedom in optimizing costs, especially availability.

To get an idea of the effect on the number of allowed constraints on the Pareto front, the experiment has been rerun with a slightly higher number of allowed constraints, with the aim of most solutions getting around ten hard constraint violations as in the evolution strategy algorithm.

The expected effect was successful; the solutions in this experiment all have ten to eleven hard constraints. The resulting Pareto fronts can be seen in figure 8.11.

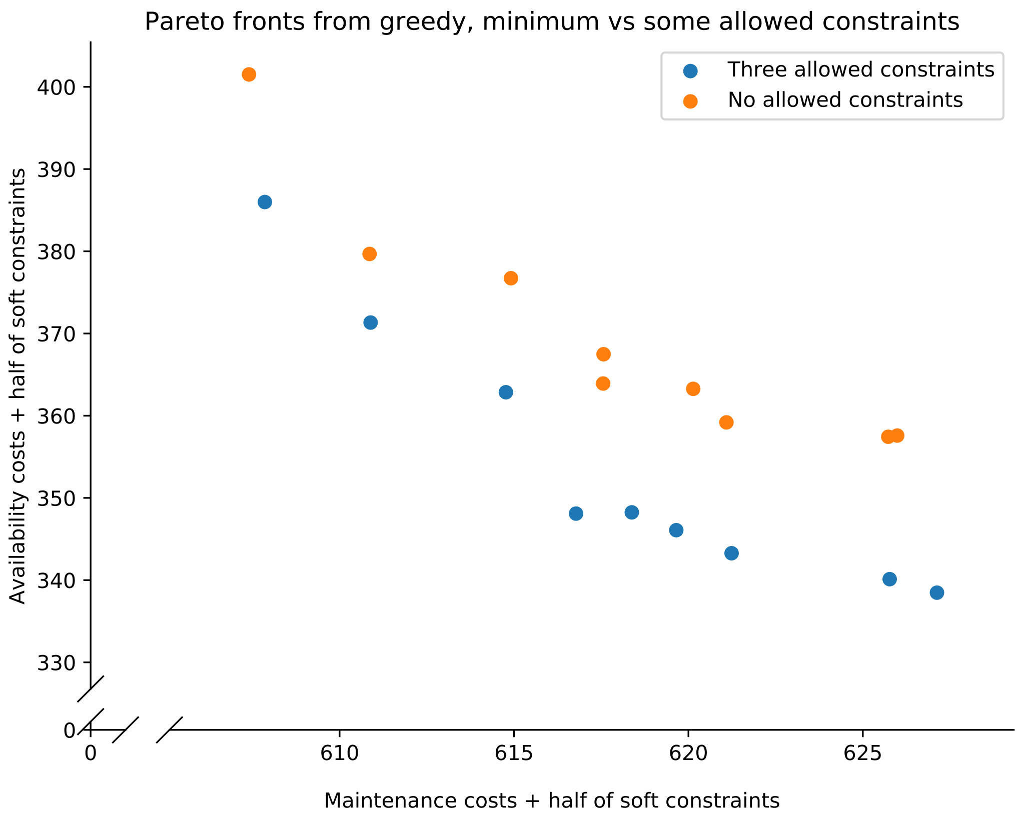


Figure 8.11: Pareto front using greedy algorithm, no constraints allowed versus some constraints allowed

Some interesting observations can be made. First of all, as expected, the total costs are lower for the solutions with higher constraint violations. It is however interesting to see that this cost reduction is pretty much only achieved through a decrease in availability costs. The range for maintenance costs stays the same over both Pareto fronts. This makes sense, as the added constraint violations are usually conflict violations, and are being used to create overlap between two impactful project requests that conflict, but also affect a very similar set of subcorridors, therefore leading to a large save in availability costs. The maintenance costs have little benefit from such overlap though; only the relatively small security costs would decrease somewhat. This means that, if ProRail would want to decrease the maintenance costs, this would have to happen by increasing the availability costs, and is hard to do by allowing more constraint violations.

Second, it can be seen that the overall shape of the Pareto front is similar to the original front. At the first few points where the weight for availability is low, the availability costs decrease much faster than the maintenance costs increase, meaning the availability costs are more important to optimize when minimizing the total costs is the objective. This is also confirmed by the fact that the reduction in availability costs decreases significantly from the fourth point onward. This signifies that, even with a relatively low weight for availability, most project requests are already optimized for availability rather than for maintenance.

Finally, it is observed that the gap between the two Pareto fronts is somewhat consistent over the whole front. This indicates that allowing more constraints has a similar effect over the whole weight range, when using the greedy algorithm.

Adding missing values for maintenance costs
As indicated in section 8.6.1, many project requests are missing data about maintenance costs meaning their values are set to zero. This limits the variability between availability and maintenance costs. To get a picture of the trade-offs that would happen when all project requests have maintenance costs, the missing values have been filled in. This has been done by fitting a simple linear regression to the project requests that have a nonzero maintenance cost. The length of the project request has been used as the independent variable. This variable has been chosen because it is the most indicative variable for the maintenance costs, also because the personnel costs are the primary cost type in the trade-off between maintenance and availability and the personnel works per hour. The distribution of the total costs of a project request over the different sub-costs such as personnel and security costs is currently done using a fixed distribution, so this is kept the same for the predicted values. Figure 8.12 shows a scatterplot of the project requests with nonzero maintenance costs, with length plotted against total maintenance cost, and the fitted line used to fill in the missing maintenance costs.

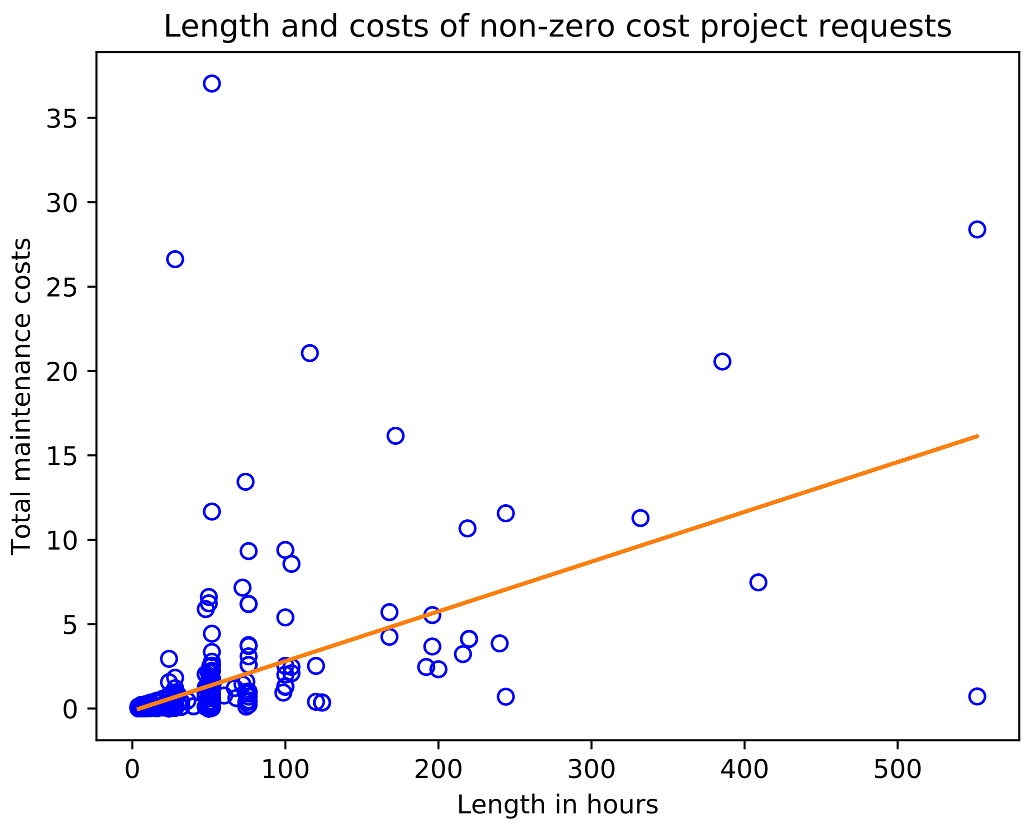
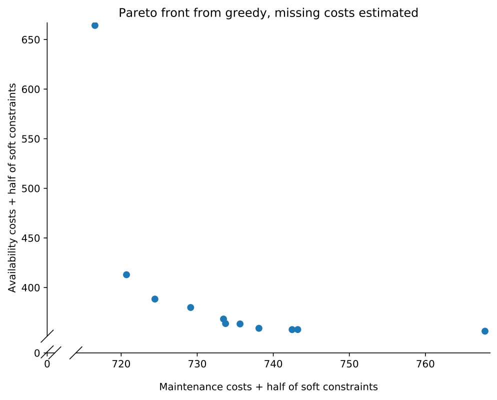


Figure 8.12: Scatterplot of project requests with non-zero maintenance costs, length versus maintenance costs, and a fitted linear regression

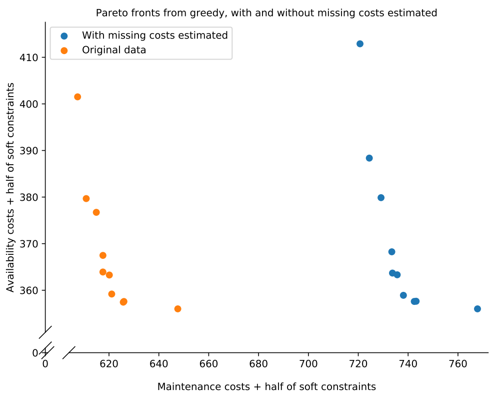
The total (gross) personnel costs in the original data are 140.7 and the total security costs are 16.9. When adding the costs according to the linear regression, the total personnel and security costs rise to 167.5 and 20.1 respectively. The increase is relatively small, primarily due to the fact that project requests of four hours or less in length still do not get any maintenance costs. This is because the linear regression indicates a negative value for these lengths. Of course, a negative value has not been filled in, instead, the costs remain zero for these requests. Second, the project requests missing maintenance costs are, on average, much shorter in length than the project requests with maintenance costs, meaning the relative addition in costs is not that high. Even though the absolute differences in costs are quite small, a difference in results is still expected, simply because there are more requests that can be used to trade off maintenance and availability.

As before, a Pareto front has been created by using a simple linear weighting between both objectives, with the soft constraint penalties always included. Eleven runs have been performed with the weights ranging from 0.0 to 1.0 and the total weight always being equal to 1.0.

Figure 8.13 shows the resulting Pareto front. In figure 8.13b the Pareto front of the original data can be compared with the new Pareto front easily. It can be seen that they are quite similar. For low maintenance costs, the availability costs are somewhat higher for the adjusted data. This is logical; the blocks that did not have maintenance costs could always be optimized for availability, while now, they will be optimized for maintenance if the weight for availability is low enough. However, from the third point onwards in both fronts, the behaviour is almost the same. The difference in total maintenance costs is likely to be too small to make a large difference. This also means that, if the missing data turns out to be somewhat in line with the existing data, the conclusions drawn about the trade-off between maintenance and availability are likely to still hold then as well.



(a) Pareto front of greedy algorithm, missing maintenance costs estimated



(b) Pareto front of greedy algorithm, original data versus data with missing maintenance costs estimated

Figure 8.13: Pareto fronts of greedy algorithm with missing maintenance costs estimated

Test with 2020 data

To check the consistency of the conclusions regarding the trade off between maintenance and availability over multiple data sets, a Pareto front was also created for the 2020 data. Compared to the 2019 data, this data set has more availability cost and less maintenance costs. It is therefore the expectation that the conclusions regarding the importance of availability over maintenance will be confirmed, if not strengthened by this data.

Figure 8.14 shows the generated Pareto front.

The number of points in this Pareto front is smaller because some of the generated points were dominated by other points, mostly due to a difference in soft constraint penalties. The shape of the Pareto front is very similar to that of the front for the 2019 data, where the decrease in availability costs is large for low availability weights and flattens when this weight increases further. There is no reason to change the earlier conclusions based on the results on this dataset.

8.7. Trade-off between costs and hard constraints

Next to the trade-off between maintenance and availability discussed in detail in the previous section, another trade-off is present in the problem: that of the total costs versus the hard constraint violations. Logically, this trade-off exists; if the costs and constraint violations would not have an inversely proportional relationship, the constraints would hardly increase the difficulty of the problem. In this section, this trade-off is made more explicit.

To visualize the trade-off between costs and constraints, two sets of data points have been used. The first one comes from a single evolution strategy run. Due to the fact that constraint cooling is used, as explained in section 4.2, it was possible to store several (cost, constraint violations) points during a single optimization run. It should be mentioned that the intermediate results for high constraint violations are probably not

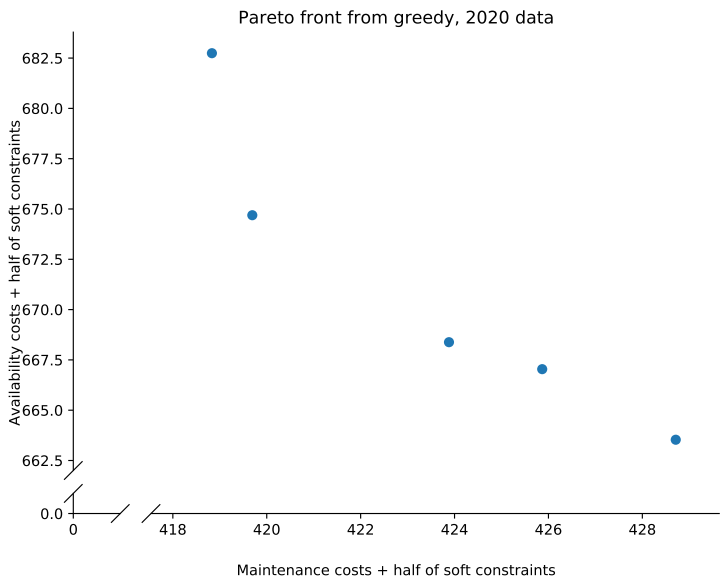
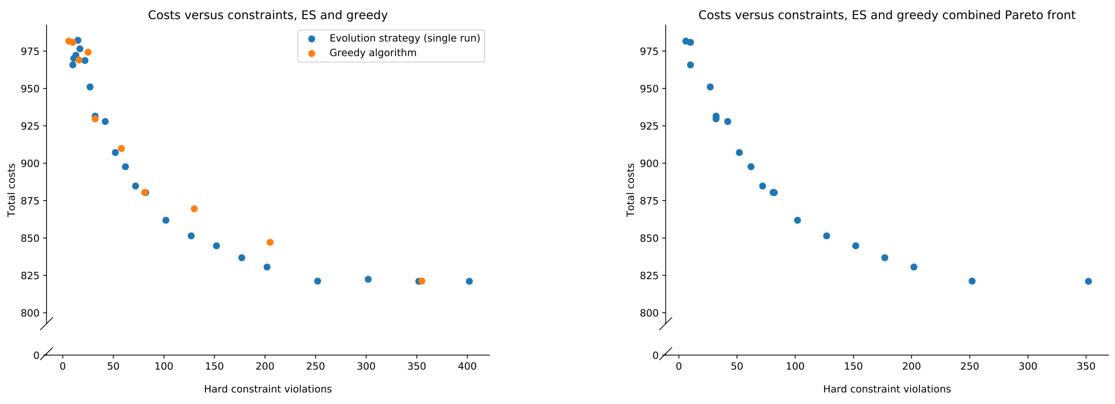


Figure 8.14: Pareto front from greedy algorithm on 2020 data

optimal in terms of costs since the algorithm was not optimized for intermediate results. The run that was used was the run with the lowest total costs from the evolution strategy. This does not mean the costs were low during the whole run, as could already be seen in figure 8.3. The second data set was generated by running the greedy algorithm multiple times with a different number of allowed constraints.

Figure 8.15 shows the results in terms of costs and constraints. In figure 8.15a, the points for both algorithms can be seen separately. Figure 8.15b shows a combined Pareto front, with the dominated points removed.



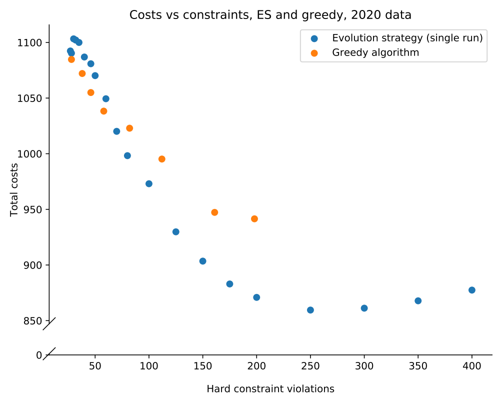
(a) Costs versus constraints, evolution strategy and greedy (b) Costs versus constraints, combined Pareto front

Figure 8.15: Costs versus constraints

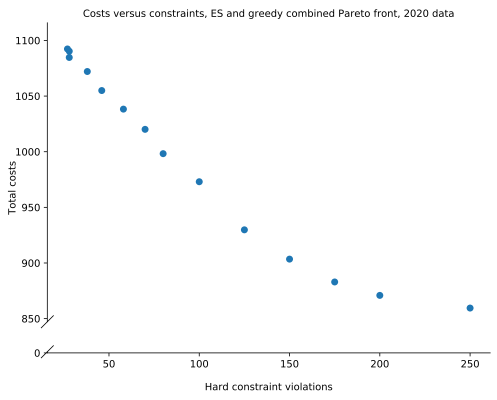
First, let's consider the differences between the points from the evolution strategy and the greedy algorithm. The points follow a somewhat similar shape. However, at high constraint violations, the evolution strategy achieves lower cost solutions than the greedy algorithm. This indicates that the greedy algorithm does not use its allowed constraint violations in the most optimal way. When the constraint violations go towards zero, the two solution methods are quite close. Looking closely, one can also see that for the evolution strategy, the point with the lowest number of constraint violations actually has lower costs than some points with slightly higher constraint violations. This is due to the fact that at the end of the evolution strategy, approximately one-third of its runtime is used to optimize the solution that is already at the minimum constraint violations at that time.

The shape of the combined Pareto front confirms an observation that was already seen implicitly in previous experiments; most of the costs are "lost" when optimizing the final few hard constraint violations. This is seen by the fact that the (implicit) graph through the points is concave up and decreasing. This shows that going from 400 to 100 broken constraints causes a much smaller cost increase than going from 100 to 10 broken constraints. This conclusion makes sense considering the data. The fact that the data has a lot of relatively small project requests and relatively few large project requests, causes the costs to remain relatively low when solving the first hard constraint violations, as they can mostly be solved by moving small blocks, which have a relatively low impact on the total costs. Only when the constraint violations are mostly caused by large blocks, will solving the constraints cause larger cost increases.

A similar test has been done with the 2020 data, to see if there are any significant differences between the two data sets. The results can be seen in figure 8.16.



(a) Costs versus constraints, evolution strategy and greedy, 2020 data



(b) Costs versus constraints, combined Pareto front, 2020 data

Figure 8.16: Costs versus constraints, 2020 data

The results for the 2020 data are similar to those of the 2019 data, but there are some subtle differences. First of all, the graph from approximately 25 to 150 broken constraints is a bit more linear for the 2020 data, indicating that the cost increase per constraint violation is near constant on this interval. Previously, it was observed that this increase went up while going towards a lower number of constraint violations. This difference is likely due to the fact that the 2020 data has fewer, but larger blocks. This means the "cross over point" where the constraint violations are mostly caused by large blocks, and the costs per constraint violation increase, is reached earlier.

A second interesting result which is not directly visible from the graphs is the result of the greedy algorithm under a high number of allowed constraints. When running the greedy algorithm with 400 and 300 allowed constraints respectively, the exact same solution was returned, which had only 198 constraint violations, that also being the reason that there are no points shown for the greedy algorithm beyond that. This means that even though the algorithm was still allowed to choose plan moments which were more optimal in terms of costs but caused extra constraint violations, such moments were not available. However, the evolution strategy was able to find solutions with much lower costs for the same number of constraint violations. This shows the sub-optimality of the greedy algorithm, which can be caused by making the "wrong" choices early on in the algorithm, even though these are locally optimal at that point. This sub-optimality seems to be larger when the allowed constraints are higher; this was observed in both the 2019 and 2020 data. For a low number of constraint violations, the results of the greedy algorithm are actually clearly better in the 2020 data.

9 Discussion

This chapter is designed to give an overview of all results of the experiments in the previous chapters and to give a brief discussion of these results. No new results will be presented in this chapter, and most of the actual plots and numbers will not be repeated. For those, the reader is referred to the original sections with the experiments and results.

In the previous chapters, various areas of the problem have been experimented with, and they will be discussed here. First, a brief summary of the algorithms and results and the chapters they can be found in is given. Then, the different types of algorithm and their benefits and drawbacks are discussed. Afterwards, the trade-off between maintenance cost and availability is considered. Finally, the trade-off between costs and constraint violations will be discussed.

9.1. Summary of algorithms and results

Multiple algorithms and improvements were implemented and tested throughout the thesis. In this section, they are briefly summarized.

In chapter 4, various improvements to the existing evolution strategy algorithm were proposed. These improvements were successful and improved the solution quality of the results. In section 4.6, a simulated annealing algorithm was also tested on the problem, using these same improvements. The results were found to be unsatisfactory in terms of costs.

Afterwards, two multi-objective algorithms were tested on the problem in chapter 5. The first algorithm, NSGA-II, was found to be highly ineffective in solving this problem; the solutions were not diverse and of bad quality. The second algorithm, AMOSA, was much better than NSGA-II, both in terms of costs and diversity. However, the total costs were still significantly higher than those of the results from the (improved) evolution strategy.

In chapter 6, an initial parameter analysis was performed for the improved evolution strategy. Based on this, a number of parameters were changed slightly to improve the expected result. Furthermore, it was found that certain specific mutations might be overused, a problem area which was investigated further in section 8.2.

In chapter 7, a greedy algorithm was implemented and tested on the problem. This algorithm was designed to have a much shorter runtime than a metaheuristic algorithm. The results were surprisingly good; in a much shorter runtime, the greedy algorithm was found to produce results of similar solution quality to the improved evolution strategy.

After noticing that the evolution strategy and greedy algorithms were effective in different areas of the problem, they were put together into a hybrid version in section 7.4. This method proved to be very effective in combining the strengths of the two algorithms, yielding a large cost reduction compared to any of its counterparts.

In chapter 8, two further improvements were made to the evolution strategy. One of these improvements clearly improved the results, while another seemed to give a small improvement, but will need further testing to confirm this. A robustness analysis of the evolution strategy was also performed in section 8.1, where it was found that there is quite some variability in the results, and it is hard to determine early on in the algorithm whether the final result will be relatively good or bad.

In section 8.6, the trade-off between maintenance and availability costs was investigated further using the single-objective algorithms. It was found that the variability in availability costs is much higher than that in maintenance costs, meaning the most effective way of minimizing the total cost is to focus on minimizing the availability costs. In section 8.7, the trade-off between costs and hard constraints was made more explicit, concluding that there is a significant cost increase when adhering to the constraints.

An overview of the best solutions of the various algorithms can be found in table 10.1 in the next chapter.

9.2. Comparison of algorithms

Multiple types of algorithms have been applied to the given rail maintenance scheduling problem. In this section, an overview of these algorithms, their results, and their drawbacks and benefits is given.

9.2.1. Single-objective metaheuristics

The first type of algorithm that has been applied to the problem is the single-objective metaheuristic. These use the sum of both cost types as well as soft constraint penalties as their objective value which they try to minimize and do not consider the trade-off between the cost types.

The existing algorithm that was developed by Macomi is an evolution strategy which is one of these single-objective metaheuristics. This algorithm was improved in various ways, making it both faster through a new scoring function, and more effective by using smarter mutations and other operators. Experiments showed that these improvements worked well and the costs were decreased significantly while not increasing the number of hard constraint violations. The improvements are described in detail in chapter 4.

Another single-objective metaheuristic that was briefly tested in section 4.6 was a simulated annealing algorithm. Most improvements that were made for the evolution strategy were also used in this algorithm. It was found that this algorithm performed worse than the evolution strategy, leading to the simulated annealing algorithm being discarded in favour of the evolution strategy, which was then optimized further with a parameter analysis and used in a hybrid version with a greedy algorithm.

The main benefits of a single-objective metaheuristic and especially the (improved) evolution strategy are as follows:

- The algorithm is able to find results with a low number of constraint violations consistently.
- The algorithm is effective in minimizing availability costs.

The main drawbacks are as follows:

- The runtime of the algorithm is quite long. Although the runtime of the algorithm can be specified by the user, a good result requires at least twelve hours of runtime.
- The algorithm is somewhat unrobust in terms of costs. Although no major outliers were found during the robustness analysis, there is enough variability to warrant the need for multiple runs, therefore increasing the effective runtime of the algorithm.
- The algorithm is relatively ineffective in optimizing soft constraint penalties (in the current constraint scenario).

Overall, the evolution strategy is preferred over the simulated annealing, and in a broader context, the evolution strategy performed very well as an early algorithm but has since been overtaken by the greedy algorithm in terms of robustness, and the greedy-ES hybrid in terms of solution quality.

9.2.2. Multi-objective algorithms

The trade-off between maintenance and availability costs was deemed an interesting research area, and it warranted the use of multi-objective algorithms. Two multi-objective algorithms were implemented and tested in chapter 5 NSGA-II [30], a multi-objective genetic algorithm, and AMOSA [16], a multi-objective simulated annealing algorithm.

The results for NSGA-II were unsatisfactory. The costs were much higher than for any of the single-objective algorithms, and there was almost no diversity in terms of the two objectives. This is believed to be a combination of the fact that a genetic algorithm does not work well on this problem, as also confirmed by an earlier test with a single-objective GA, and the fact that the population converged towards a middle-of-the-ground solution quickly due to an expected lower number of constraint violations in that part of the search space, and the algorithm not being able to escape from that local optimum.

The results for AMOSA were significantly better. However, the total costs were still significantly higher than for the single-objective algorithms. Especially the availability costs were not optimized in a satisfactory way. The diversity was higher than for NSGA-II, but still, only a small part of the Pareto front was covered, even though other algorithms confirmed the other part of the Pareto front exists.

The main benefits of a multi-objective algorithm are as follows:

- By returning multiple solutions, the trade-off between maintenance and availability costs is made more explicit, and a domain expert can make informed decisions on which trade-offs he does or does not want to make.

The main drawbacks are the following:

- The performance of the algorithm in terms of total costs is worse than single-objective algorithms.
- The diversity of solutions in terms of the two objectives is small.
- The runtime of the algorithm is long; at least twelve hours of runtime are needed to get a representative result.

Overall, it was concluded that the benefits of a multi-objective algorithm do not outweigh the inferior solutions and lack of diversity. Therefore, it is concluded that a multi-objective algorithm is not a good solution method for this problem.

9.2.3. Greedy algorithms

A greedy algorithm was originally developed with the idea of having an algorithm which was able to quickly give a good indication of a possible result and to use in an iterative manner. However, the algorithm worked surprisingly well compared to the single- and multi-objective metaheuristics. In a much shorter runtime, the greedy algorithm provides solutions that are similar in quality compared to the average result of the improved evolution strategy. The fact that the greedy algorithm is, by default, deterministic, means that after a single run it is clear what solution the algorithm will give, and there is no uncertainty about whether the solution was on the high or low end of the distribution of solutions, as there would be for the evolution strategy. Although the evolution strategy does provide better results in its "lucky" runs, the greedy algorithm is a great way to get a guaranteed decent result within a short time. The greedy algorithm is described in chapter 7.

As shown in section 7.3, adding randomization to the greedy algorithm decreased its robustness, but also revealed that the solution of the deterministic algorithm was sometimes improved. Given the short runtime of the greedy algorithm, multiple runs can be done in the same time as a single evolution strategy run can be done, meaning the greedy algorithm has a higher chance of finding a good solution within that time.

The main benefits of a greedy algorithm are as follows:

- The runtime is very short compared to single- and multi-objective metaheuristics.
- The algorithm can be set to be deterministic, meaning a single run is enough and no re-runs are needed. Furthermore, the algorithm can be randomized in multiple ways, allowing for less robustness, but a possible chance to find a better solution.
- The results are similar to the results of the improved evolution strategy, despite the runtime difference.

The main drawbacks are as follows:

- There is little use in allowing more runtime to the algorithm. The only way it can be used is by re-running the algorithm and storing the best result, but the algorithm itself has a set runtime and will not improve its solution when given more time.
- The best runs of the evolution strategy are of better solution quality than the greedy algorithm, so if plenty time is available to do multiple runs, the evolution strategy will eventually return a better result.

Concluding, the greedy algorithm is a good algorithm to use when there is a need for short runtime. Although the best results of the evolution strategy are better than the results of the greedy algorithm, the difference is small and many long runs of the evolution strategy may be needed to get such a good result. The greedy algorithm can also be used with different weights for the two objectives to get a set of trade-off solutions relatively quickly, with much better solution quality than the multi-objective algorithms.

9.2.4. Hybrid greedy and evolution strategy

Noting that both the greedy algorithm and evolution strategy gave similar results, but the greedy algorithm is especially good in optimizing soft constraints and the evolution strategy is good in optimizing availability costs, a hybrid algorithm was developed in section 7.4, to try and combine the strengths of the algorithms. This hybrid algorithm uses multiple stages of planning in a number of requests greedily, performing some evolution strategy on the generated individuals, and planning in more requests in greedily, continuing this loop until all requests are planned and a final optimization through evolution strategy is performed.

The hybrid algorithm works very well on the problem. Performing evolution strategy on a smaller set of project requests which are perceived to be most impactful turns out to be a great way to optimize these requests specifically, decreasing the maintenance and especially availability costs compared to both the greedy and evolution strategy algorithms. Furthermore, the low soft constraint penalties of the greedy algorithm are still seen in the results of the greedy algorithm.

The hybrid algorithm is quite hard to set up optimally. The number of stages, requests to add per stage and runtime or generations of evolution strategy per stage can be fully specified. This gives a lot of freedom when it comes to customizing the algorithm, but also makes it hard to choose the optimal values for these parameters. Experiments showed that a difference in runtime or number of stages can make substantial differences, and there is currently no clear way to determine the optimal set-up for this problem or even a specific instance of this problem. At the moment, trial and error combined with some educated guesses regarding the structure of the data is the way the parameters are optimized.

The main benefits of the hybrid algorithm are as follows:

- Good solution quality. The hybrid algorithm has provided the best solutions in terms of costs while not increasing the hard constraint violations.
- Customizability. By allowing a full custom specification of the number of stages, requests per stage and runtime per stage, the algorithm can be tailored to different data sets or other minor changes to the problem.

The main drawbacks of the hybrid algorithm are as follows:

- Long runtime. Although a reasonable result can be achieved in a relatively short runtime, to reach the full potential of the algorithm, a runtime of at least twelve hours is necessary.
- Hard to set up optimally. The customizability also comes with a negative side; it is hard to determine which parameters are best due to the sheer number of possibilities. Furthermore, the parameters need to be changed for a different data set, as the number and type of project requests changes.

9.2.5. Conclusions

Overall, some conclusions can be drawn regarding the comparison of algorithms on the given rail maintenance scheduling problem.

- Multi-objective metaheuristic algorithms are not a good way to solve this problem; their results lack diversity and have bad objective values.
- In terms of pure costs and constraints, the hybrid greedy-ES algorithm is a clear winner, with significantly better results than either of its counterparts and other algorithms.
- With all implemented improvements, the evolution strategy produces results that are on average similar to the greedy algorithm. However, its runtime is much higher. Which algorithm is better depends a lot on the available runtime; if there is time for multiple runs of ES, it is likely one of these runs produces a better result than the greedy algorithm, but when time is limited, the greedy algorithm is more suitable and produces a more robust result.

9.3. Maintenance versus availability

The trade-off between maintenance and availability was an interesting area of the problem that was not researched much before. In this thesis, the trade-off was explored, both by using multi-objective algorithms as well as by results of single-objective algorithms.

The results from the multi-objective algorithms were not good and therefore do not provide a representative Pareto front, as already discussed in the previous section. The conclusions on the trade-off therefore come mostly from the results of single-objective algorithms, as discussed in detail in section 8.6. Based on the experiments in that section, the following conclusions could be drawn:

- In both the 2019 and 2020 data, the variability in availability costs is higher than that of the maintenance costs. This leads to the observation that the availability usually decrease faster than the maintenance costs increase, and it therefore being optimal to optimize availability at the expense of maintenance, when minimizing the total costs is the objective.
- Due to missing data regarding maintenance costs, the conclusions on the low variability of maintenance could be flawed. However, an experiment with missing costs estimated and filled in, the observed results were very similar to those from the original data. It is therefore believed that the conclusions still hold when the full data would be available.
- Allowing more hard constraint violations leads to a decrease in availability, whereas the maintenance costs stay mostly the same. It is therefore not easily possible to decrease the maintenance costs by increasing the hard constraint violations, but it rather has to happen through increasing availability costs.
- Regarding the soft constraints, a trade off between soft constraints and availability costs was also observed. The trade-off between maintenance costs and soft constraints was much less strong. Of course, these trade-offs depend heavily on the used soft constraint penalties.

9.4. Costs versus constraints

Next to the trade-off between maintenance and availability, another interesting trade-off that was explored is the one between costs and hard constraints. The hard constraints limit the feasible search space and cause solutions with higher costs. Through the experiments done during this research project, some interesting insights regarding this trade-off were discovered, which are briefly discussed below.

- The constraints impose a significant increase in the costs of a solution. A solution with around 250 hard constraint violations can have a cost reduction of around 145, compared to a solution with 10 or fewer hard constraint violations, while not even being fully optimized yet.
- The amount the total costs increase with for each fewer hard constraint violation goes up with the number of constraint violations going down. The cost difference between a solution with 100 or 400 broken constraints is relatively small, but the difference between solutions with 100 or 10 broken constraints is substantial. This means that allowing a few more hard constraint to be broken can result in relatively large cost decrease. The effect is slightly less strong in the 2020 data, where a more linear curve is observed between costs and constraints, from 10 to 200 broken constraints. This is likely due to the distribution of the 2020 data; it having fewer, larger blocks means that when a constraint violation needs to be fixed, a large block needs to be moved relatively often.
- The previous point, especially for the 2019 data, leads to the observed behaviour that for stochastic algorithms like the evolution strategy, almost all of the performance of the algorithm is determined through the way it optimizes the final 20-30 hard constraint violations. There was little to no correlation between intermediate results with a larger number of broken constraints and the end result of an evolution strategy run.
- The greedy algorithm is relatively ineffective in finding solutions with a large number of allowed constraint violations compared to the (intermediate) results of the evolution strategy algorithm. Although it will probably not be used in practice this way, it is good to know that it might be better to use another algorithm like the evolution strategy to create low cost, high constraint solutions.

10 Conclusion

In this chapter, the research questions defined in section 1.4 are answered. First, the sub-questions will be answered, after which the main research question is answered. Second, some possible future research directions are suggested to further improve and validate the results that were obtained during this thesis project.

10.1. Answers to the research questions

First, the sub-questions are answered:

- **How can the problem be defined in terms of a minimization problem with multiple objectives and constraints?**
  In section 2.3 a mathematical model was presented for the problem at hand. It was also concluded that the problem could not, or at least not with a reasonable number of decision variables, be written in a linear or quadratic form. This means certain solutions techniques like linear or quadratic programming could not be used.

- **What scheduling approaches are available and being used in problems with similar characteristics?**
  In chapter 3 various existing solution methods for similar problems in literature were discussed. It was concluded that both exact and approximate methods were used. However, exact methods were used on small test instances, mostly with a linear or quadratic structure. Due to the size of ProRail's problem, combined with a lack of such a problem structure, exact methods were found to be not well suitable for this problem. Approximate methods that were used are mostly metaheuristics like genetic algorithms. Both single and multi-objective metaheuristics have been used.

- **What is the current situation in terms of solution quality as well as runtime for both the manual ProRail planning as well as the algorithm created by Macomi?**
  The current scheduling process at ProRail is mostly manual and therefore takes a long time. A schedule can take months to make and as such, only a single schedule is made each year. The solution quality is also unoptimized; both due to the fact that it is hard for a human to optimize such a complex problem, as well as through missing data and unmodelled logic. These shortcomings had already been addressed by Macomi through the modelling of the problem and an initial evolution strategy algorithm. Using this algorithm, a schedule can be created much quicker. A run of approximately 72 hours led to a large decrease in constraint violations for a relatively small increase in costs. Furthermore, when allowing as many constraint violations as were in ProRail's original schedule, a run with similar runtime gave a cost improvement of approximately 188.8. The anytime nature of the used algorithm also allows schedules to be generated even faster, but this will lead to a loss in solution quality.

- **What are the shortcomings of the current algorithm and how may it be improved?**
  Several shortcomings were identified in chapter 4. To address these shortcomings, some improvements were made; a faster scoring function, smarter mutations, a constraint cooling process and a separation of the problem based on the difficulty of certain maintenance blocks over others. These improvements were successful; the runtime of the algorithm decreased and the costs of the obtained solution did as

well, while the hard constraint violations did not increase. Furthermore, a parameter analysis was performed to obtain an idea of the important parameters and optimal values. Based on insights obtained through this parameter analysis, as well as results of other algorithms, another mutation was added. Furthermore, the selection of mutations was changed to be more dynamic based on the current situation in terms of costs and (allowed) constraints. These improvements proved to give further cost reductions.

- **Which multi-objective algorithms exist, how can they be used to make the trade-off between maintenance costs and availability more explicit, and what is their performance on the problem?**
  Many multi-objective algorithms exist; a number of them were discussed in section 3.3. Two of them were implemented and tested on this problem: NSGA-II [30] and AMOSA [16], a genetic algorithm and simulated annealing based multi-objective algorithm respectively. The results provided by these algorithms were unsatisfactory. The solution set of the NSGA-II algorithm had almost no diversity and much higher costs than the single-objective evolution strategy. The AMOSA algorithm performed better, both in terms of quality and diversity, but still, only a small part of the Pareto front was covered and the total costs were still significantly higher. It was concluded that multi-objective algorithms are not a suitable solution method for this problem.

- **Can other single-objective algorithms, such as different metaheuristics or a greedy algorithm, outperform Macomi's algorithm?**
  Next to the (improved) evolution strategy, a simulated annealing algorithm was also tested. Although this algorithm also outperformed the multi-objective algorithms, it was somewhat inferior to the evolution strategy. Therefore, the evolution strategy algorithm was developed further throughout the remainder of the thesis, and the simulated annealing algorithm was discarded. A greedy algorithm was also implemented and tested, and the solutions were surprisingly good. The solutions were only slightly worse than the average result of the evolution strategy, but the runtime was in the order of minutes rather than hours. Furthermore, the greedy algorithm is much more robust; it is originally deterministic and can be tuned with a certain amount of randomness. Due to its short runtime, the greedy algorithm can also be used to get a quick indication of a possible result on new data.

- **Can algorithms be combined to improve their performance?**
  After the realization that the evolution strategy and greedy algorithms were strong in optimizing different parts of the objective function, they were combined to test whether they could both employ their strengths and provide better solutions than either of its counterparts. This turned out to be the case; the solutions were found to have a significant cost reduction. Several configurations were tested to find an optimal configuration. Using this, a cost reduction of around 42.4 was found compared to the fully improved evolution strategy.

Now, the main research question can be answered. The main research question was

Which solution method is the most suitable to improve the yearly maintenance schedule of ProRail?

Of the algorithms that were tested in this research project, it can be concluded that the *hybrid greedy and evolution strategy algorithm* is the most suitable algorithm to use. Only if a much shorter runtime is required, the greedy algorithm could be a more suitable candidate, but even then, a short time of evolution strategy added will already improve the results substantially. This new algorithm allows for both faster and better maintenance schedules for ProRail, leading to cost reductions and a decrease in infrastructure unavailability. Table 10.1 gives an overview of the results obtained by the various algorithms and configurations that have been tested with.

10.2. Future research

Next, some possible pointers for future research directions are given.

- Further testing and validation of results. Due to time limitations and the relatively long runtime of most of the algorithms, the improvements and configurations could only be tested with a limited number of runs. Furthermore, only two data sets were available to test on. It would be good to repeat the existing tests more often, and test with other data sets, to gain more insight in the distribution of solutions under different configurations and improve the confidence in the conclusions.

Algorithm (+ configuration)	Runtime	Total costs	Hard constraints	Maintenance	Availability	Soft constraints
ProRail manual schedule	> 6 months	965.2	690	654.4	**285.4**	25.5
Unimproved ES	24 hours	1005.6	**9**	610.2	353.7	41.6
Improved ES, average	24 hours	986.1	11.4	607.0	343.8	35.1
Improved ES, best seen result	24 hours	965.1	10	609.1	321.6	34.4
ES with adaptive mutation, average	24 hours	981.5	11.4	607.4	339.2	34.9
ES with mintime mutation, average	24 hours	975.8	10.8	606.8	343.5	25.5
ES with adaptive and mintime, average	24 hours	971.4	12.6	**606.9**	340.8	23.6
Simulated annealing	24 hours	1011.0	10	610.1	360.1	40.8
NSGA-II, best total costs result	48 hours	1148.2	13	628.0	466.7	53.5
AMOSA, best total costs result	24 hours	1050.0	10	611.0	402.0	37.1
Initial greedy	10 minutes	995.9	13	634.0	331.7	30.3
Improved greedy	15 minutes	980.5	10	608.5	348.2	23.8
Hybrid greedy-ES, individuals as starting pop	1.5 hours	959.3	10	607.7	325.9	25.7
Hybrid greedy-ES, short run	1 hour	957.2	10	607.9	327.8	**21.5**
Hybrid greedy-ES, long run	24 hours	**928.1**	10	607.4	294.0	26.7

Table 10.1: Overview of algorithm results. The best result in each column is highlighted in **red**

- Integration with asset operation clustering. The input to this problem is a set of project requests; where they come from was out of the scope of this thesis. However, as briefly explained in section 2.1.2, these project requests are based on clusters of asset operations, which are smaller units of maintenance, which are selected from a pool of asset operations that spans multiple years. The algorithms for scheduling the project requests developed in this thesis could be used to make a smarter clustering for asset operations. For example, the fast greedy algorithm may be used to quickly gain insights into the costs and constraints of a certain asset operation clustering, which may then be used to change this clustering. Iteratively, the clustering and planning of these asset operations over multiple years may be improved.

- Performance under different scenarios. Currently, the base constraint scenario as described in section 2.2.3 was used in all experiments. It would be interesting to see how the different algorithms perform when this scenario changes, and whether the optimal configurations would be robust under this change, or whether they should be adjusted.

- Improved parameter analysis. As explained in section 6, the performed analysis is only the first step towards finding the optimal parameters. Furthermore, after doing the parameter analysis, new improvements through mutations and other, better performing algorithms have been found. Doing a full parameter analysis on, for example, the hybrid greedy-ES algorithm would be beneficial to achieve the maximum performance of the algorithm.

Bibliography

[1] Prorail maintenance. https://www.prorail.nl/reizigers/aanbesteden-en-inkoop/spooronderhoud-pgo, November 2018. Accessed: 2018-11-16.

[2] Prorail corridorboek 2014. https://www.prorail.nl/sites/default/files/corridorboek_2014.pdf, May 2019. Accessed: 2019-05-15.

[3] Irg rail market monitoring 2019. https://www.irg-rail.eu/irg/documents/market-monitoring/220,2019.html, May 2019. Accessed: 2019-05-13.

[4] Macomi. https://www.macomi.nl, May 2019. Accessed: 2019-05-13.

[5] Prorail. https://www.prorail.nl, May 2019. Accessed: 2019-05-15.

[6] Prorail ambities. https://www.prorail.nl/reizigers/over-prorail/wat-doet-prorail/onze-ambities, May 2019. Accessed: 2019-05-13.

[7] Prorail in cijfers. https://www.prorail.nl/reizigers/over-prorail/wat-doet-prorail/prorail-in-cijfers, May 2019. Accessed: 2019-05-13.

[8] Prorail exploitatielasten 2018. https://www.jaarverslagprorail.nl/verslag/jaarrekening/s1069_toeliodbadweverliehkasst/a1051_14-Exploitatielasten, May 2019. Accessed: 2019-05-13.

[9] Prorail kerncijfers 2018. https://www.jaarverslagprorail.nl/verslag/kerncijfers2, May 2019. Accessed: 2019-05-13.

[10] Prorail meerjarenoverzicht 2018. https://www.jaarverslagprorail.nl/verslag/s1077_meerjarenoverzicht, May 2019. Accessed: 2019-05-13.

[11] Toekomstbestendig werken aan het spoor. https://www.rijksoverheid.nl/documenten/rapporten/2017/12/19/bijlage-5b-samenvatting-pva-programma-toekomstbestendig-werken-aan-het-spoor, May 2019. Accessed: 2019-05-01.

[12] Wat doet prorail. https://www.prorail.nl/reizigers/over-prorail/wat-doet-prorail, May 2019. Accessed: 2019-05-13.

[13] Herman Aguinis, Ryan K. Gottfredson, and Harry Joo. Best-practice recommendations for defining, identifying, and handling outliers. *Organizational Research Methods*, 16(2):270–301, jan 2013. doi: 10.1177/1094428112470848.

[14] T. W. Anderson and D. A. Darling. A test of goodness of fit. *Journal of the American Statistical Association*, 49(268):765–769, dec 1954. doi: 10.1080/01621459.1954.10501232.

[15] A. Ramos Andrade and P. Fonseca Teixeira. Biobjective optimization model for maintenance and renewal decisions related to rail track geometry. *Transportation Research Record: Journal of the Transportation Research Board*, 2261(1):163–170, jan 2011. doi: 10.3141/2261-19.

[16] S. Bandyopadhyay, S. Saha, U. Maulik, and K. Deb. A simulated annealing-based multiobjective optimization algorithm: AMOSA. *IEEE Transactions on Evolutionary Computation*, 12(3):269–283, jun 2008. doi: 10.1109/tevc.2007.900837.

[17] Aymeric Blot, Marie-Éléonore Kessaci, and Laetitia Jourdan. Survey and unification of local search techniques in metaheuristics for multi-objective combinatorial optimisation. *Journal of Heuristics*, 24(6):853–877, may 2018. doi: 10.1007/s10732-018-9381-1.

[18] P Boonma and J Suzuki. Pibea: Prospect indicator based evolutionary multiobjective optimization algorithm. In *Proc. IEEE Congress on Evol. Computat*, 2011.

[19] Dimo Brockhoff and Eckart Zitzler. Improving hypervolume-based multiobjective evolutionary algorithms by using objective reduction methods. In *2007 IEEE Congress on Evolutionary Computation*. IEEE, sep 2007. doi: 10.1109/cec.2007.4424730.

[20] G Budai, D Huisman, and R Dekker. Scheduling preventive railway maintenance activities. *Journal of the Operational Research Society*, 57(9):1035–1044, sep 2006. doi: 10.1057/palgrave.jors.2602085.

[21] Gabriella Budai-Balke, Rommert Dekker, and Uzay Kaymak. Genetic and memetic algorithms for scheduling railway maintenance activities. *Erasmus University Rotterdam, Econometric Institute, Econometric Institute Report*, 12 2009.

[22] Luis Filipe Caetano and Paulo Fonseca Teixeira. Availability approach to optimizing railway track renewal operations. *Journal of Transportation Engineering*, 139(9):941–948, sep 2013. doi: 10.1061/(asce) te.1943-5436.0000575.

[23] Xinye Cai, Xin Cheng, Zhun Fan, Erik Goodman, and Lisong Wang. An adaptive memetic framework for multi-objective combinatorial optimization problems: studies on software next release and travelling salesman problems. *Soft Computing*, 21(9):2215–2236, dec 2015. doi: 10.1007/s00500-015-1921-0.

[24] G. Cancian, G. Chai, and W. Pullan. Budget constrained pavement maintenance scheduling using a parallel genetic algorithm. In *Proceedings of the Fourth International Conference on Soft Computing Technology in Civil, Structural and Environmental Engineering*. Civil-Comp Press, 2018. doi: 10.4203/ ccp.109.31.

[25] Carlos Catania, Cecilia Zanni-Merk, François de Bertrand de Beuvron, and Pierre Collet. A multi objective evolutionary algorithm for solving a real health care fleet optimization problem. *Procedia Computer Science*, 60:256–265, 2015. doi: 10.1016/j.procs.2015.08.125.

[26] W. T. Chan, T. F. Fwa, and C. Y. Tan. Road-maintenance planning using genetic algorithms. i: Formulation. *Journal of Transportation Engineering*, 120(5):693–709, sep 1994. doi: 10.1061/(asce) 0733-947x(1994)120:5(693).

[27] Ruey Long Cheu, Ying Wang, and Tien Fang Fwa. Genetic algorithm-simulation methodology for pavement maintenance scheduling. *Computer-Aided Civil and Infrastructure Engineering*, 19(6):446–455, nov 2004. doi: 10.1111/j.1467-8667.2004.00369.x.

[28] BGW Craenen, AE Eiben, and E Marchiori. How to handle constraints with evolutionary algorithms. *Practical Handbook Of Genetic Algorithms: Applications*, pages 341–361, 2001.

[29] Cuong Dao, Rob Basten, and Andreas Hartmann. Maintenance scheduling for railway tracks under limited possession time. *Journal of Transportation Engineering, Part A: Systems*, 144(8):04018039, aug 2018. doi: 10.1061/jtepbs.0000163.

[30] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *Trans. Evol. Comp*, 6(2):182–197, April 2002. ISSN 1089-778X. doi: 10.1109/4235.996017. URL `http://dx.doi.org/10.1109/4235.996017`.

[31] Kalyanmoy Deb. Constrained multi-objective evolutionary algorithm. In *Studies in Computational Intelligence*, pages 85–118. Springer International Publishing, jun 2018. doi: 10.1007/978-3-319-91341-4_6.

[32] Kalyanmoy Deb, Udaya Bhaskara Rao N., and S. Karthik. Dynamic multi-objective optimization and decision-making using modified NSGA-II: A case study on hydro-thermal power scheduling. In *Lecture Notes in Computer Science*, pages 803–817. Springer Berlin Heidelberg, 2007. doi: 10.1007/ 978-3-540-70928-2_60.

[33] Kalyanmoy Deb, Karthik Sindhya, and Jussi Hakanen. Multi-objective optimization. In *Decision Sciences*, pages 145–184. CRC Press, nov 2016. doi: 10.1201/9781315183176-4.

[34] Roman Denysiuk, André V. Moreira, José C. Matos, Joel R. M. Oliveira, and Adriana Santos. Two-stage multiobjective optimization of maintenance scheduling for pavements. *Journal of Infrastructure Systems*, 23(3):04017001, sep 2017. doi: 10.1061/(asce)is.1943-555x.0000355.

[35] Marco Dorigo. *Optimization, Learning and Natural Algorithms*. PhD thesis, Politecnico di Milano, Italy, 1992.

[36] Iyad Abu Doush, Mohammad Qasem Bataineh, and Mohammed El-Abd. The hybrid framework for multi-objective evolutionary optimization based on harmony search algorithm. In *Advances in Intelligent Systems and Computing*, pages 134–142. Springer International Publishing, may 2018. doi: 10.1007/978-3-319-91337-7_13.

[37] Aurélien Froger, Michel Gendreau, Jorge E. Mendoza, Éric Pinson, and Louis-Martin Rousseau. Maintenance scheduling in the electricity industry: A literature review. *European Journal of Operational Research*, 251(3):695–706, jun 2016. doi: 10.1016/j.ejor.2015.08.045.

[38] Hui Gao and Xueqing Zhang. A markov-based road maintenance optimization model considering user costs. *Computer-Aided Civil and Infrastructure Engineering*, 28(6):451–464, mar 2013. doi: 10.1111/mice.12009.

[39] Raneem Gashgari, Lamees Alhashimi, Raed Obaid, and Thangam Palaniswamy. A survey on exam scheduling techniques. *2018 1st International Conference on Computer Applications & Information Security (ICCAIS)*, pages 1–5, 2018.

[40] Zong Woo Geem, Joong Hoon Kim, and G.V. Loganathan. A new heuristic optimization algorithm: Harmony search. *SIMULATION*, 76(2):60–68, feb 2001. doi: 10.1177/003754970107600201.

[41] Fred Glover. Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research*, 13(5):533–549, jan 1986. doi: 10.1016/0305-0548(86)90048-1.

[42] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, New York, 1989.

[43] Aaltje Rianda Jenema. An optimization model for a train-free-period planning for prorail based on the maintenance needs of the dutch railway infrastructure. mathesis, Delft University of Technology, September 2011.

[44] Liangjun Ke, Qingfu Zhang, and Roberto Battiti. Hybridization of decomposition and local search for multiobjective optimization. *IEEE Transactions on Cybernetics*, 44(10):1808–1820, oct 2014. doi: 10.1109/tcyb.2013.2295886.

[45] Safa Khalouli, Rachid Benmansour, and Said Hanafi. An ant colony algorithm based on opportunities for scheduling the preventive railway maintenance. In *2016 International Conference on Control, Decision and Information Technologies (CoDIT)*. IEEE, apr 2016. doi: 10.1109/codit.2016.7593629.

[46] Alexander Kiefer, Michael Schilde, and Karl F. Doerner. Scheduling of maintenance work of a large-scale tramway network. *European Journal of Operational Research*, 270(3):1158–1170, nov 2018. doi: 10.1016/j.ejor.2018.04.027.

[47] Scott Kirkpatrick, C. D. Gelatt, and Mario P. Vecchi. Optimization by simulated annealing. *Science*, 220 4598:671–80, 1983.

[48] Marco Laumanns, Lothar Thiele, Kalyanmoy Deb, and Eckart Zitzler. Combining convergence and diversity in evolutionary multiobjective optimization. *Evolutionary Computation*, 10(3):263–282, sep 2002. doi: 10.1162/106365602760234108.

[49] H. Li and D. Landa-Silva. An adaptive evolutionary multi-objective approach based on simulated annealing. *Evolutionary Computation*, 19(4):561–595, dec 2011. doi: 10.1162/evco_a_00038.

[50] Qiang Long. A constraint handling technique for constrained multi-objective genetic algorithm. *Swarm and Evolutionary Computation*, 15:66–79, apr 2014. doi: 10.1016/j.swevo.2013.12.002.

[51] Jianping Luo, Yun Yang, Xia Li, Qiqi Liu, Minrong Chen, and Kaizhou Gao. A decomposition-based multi-objective evolutionary algorithm with quality indicator. *Swarm and Evolutionary Computation*, 39:339–355, apr 2018. doi: 10.1016/j.swevo.2017.11.004.

[52] Aditya Medury and Samer Madanat. Incorporating network considerations into pavement management systems: A case for approximate dynamic programming. *Transportation Research Part C: Emerging Technologies*, 33:134–150, aug 2013. doi: 10.1016/j.trc.2013.03.003.

[53] Krzysztof Michalak. Evolutionary algorithm with a directional local search for multiobjective optimization in combinatorial problems. *Optimization Methods and Software*, 31(2):392–404, dec 2015. doi: 10.1080/10556788.2015.1121485.

[54] D.C. Montgomery. *Design and Analysis of Experiments*. Student solutions manual. John Wiley & Sons, 2008. ISBN 9780470128664. URL `http://books.google.de/books?id=kMMJAm5bD34C`.

[55] Max D. Morris. Factorial sampling plans for preliminary computational experiments. *Technometrics*, 33 (2):161–174, 1991. ISSN 00401706. URL `http://www.jstor.org/stable/1269043`.

[56] Daniel Peralta, Christoph Bergmeir, Martin Krone, Marta Galende, Manuel Menéndez, Gregorio I. Sainz-Palmero, Carlos Martinez Bertrand, Frank Klawonn, and Jose M. Benitez. Multiobjective optimization for railway maintenance plans. *Journal of Computing in Civil Engineering*, 32(3):04018014, may 2018. doi: 10.1061/(asce)cp.1943-5487.0000757.

[57] Stuart J. Russell and Peter Norvig. *Artificial intelligence - a modern approach, 2nd Edition*. Prentice Hall series in artificial intelligence. Prentice Hall, 2003. ISBN 0130803022. URL `http://www.worldcat.org/oclc/314283679`.

[58] João Santos, Adelino Ferreira, Gerardo Flintsch, and Veronique Cerezo. A multi-objective optimisation approach for sustainable pavement management. *Structure and Infrastructure Engineering*, 14(7):854–868, feb 2018. doi: 10.1080/15732479.2018.1436571.

[59] Paolo Serafini. Simulated annealing for multi objective optimization problems. In *Multiple Criteria Decision Making*, pages 283–292. Springer New York, 1994. doi: 10.1007/978-1-4612-2666-6_29.

[60] Bernard W. Silverman. *Density Estimation for Statistics and Date Analysis*. Chapman and Hall/CRC, 1986.

[61] Karthik Sindhya, Kalyanmoy Deb, and Kaisa Miettinen. Improving convergence of evolutionary multi-objective optimization with local search: a concurrent-hybrid algorithm. *Natural Computing*, 10(4): 1407–1430, 2011. URL `http://dblp.uni-trier.de/db/journals/nc/nc10.html#SindhyaDM11`.

[62] Karthik Sindhya, Kaisa Miettinen, and Kalyanmoy Deb. A hybrid framework for evolutionary multi-objective optimization. *IEEE Trans. Evolutionary Computation*, 17(4):495–511, 2013. URL `http://dblp.uni-trier.de/db/journals/tec/tec17.html#SindhyaMD13`.

[63] Hemant Kumar Singh, Amitay Isaacs, Tapabrata Ray, and Warren Smith. A simulated annealing algorithm for constrained multi-objective optimization. In *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*. IEEE, jun 2008. doi: 10.1109/cec.2008.4631013.

[64] Sarah S. Soh, Nor. H.M. Radzi, and Habibollah Haron. Review on scheduling techniques of preventive maintenance activities of railway. In *2012 Fourth International Conference on Computational Intelligence, Modelling and Simulation*. IEEE, sep 2012. doi: 10.1109/cimsim.2012.56.

[65] N. Srinivas and Kalyanmoy Deb. Muiltiobjective optimization using nondominated sorting in genetic algorithms. *Evolutionary Computation*, 2(3):221–248, sep 1994. doi: 10.1162/evco.1994.2.3.221.

[66] Anupam Trivedi, Dipti Srinivasan, Krishnendu Sanyal, and Abhiroop Ghosh. A survey of multiobjective evolutionary algorithms based on decomposition. *IEEE Transactions on Evolutionary Computation*, pages 1–1, 2016. doi: 10.1109/tevc.2016.2608507.

[67] Jannet I. van Zante-de Fokkert, Dick den Hertog, F. J. van den Berg, and J. H. M. Verhoeven. The netherlands schedules track maintenance to improve track workers' safety. *Interfaces*, 37:133–142, 2007.

[68] D.A.G. Vieira, R.L.S. Adriano, J.A. Vasconcelos, and L. Krahenbuhl. Treating constraints as objectives in multiobjective optimization problems using niched pareto genetic algorithm. *IEEE Transactions on Magnetics*, 40(2):1188–1191, mar 2004. doi: 10.1109/tmag.2004.825006.

[69] Yu yuan Chang, Omar B. Sawaya, Athanasios K. Ziliaskopoulos, Yu yuan Chang, Omar B. Sawaya, and Athanasios K. Ziliaskopoulos. A tabu search based approach for work zone scheduling, 2000.

[70] Qingfu Zhang and Hui Li. Moea/d: A multiobjective evolutionary algorithm based on decomposition. *IEEE Trans. Evolutionary Computation*, 11(6):712–731, 2007. URL `http://dblp.uni-trier.de/db/journals/tec/tec11.html#ZhangL07`.

[71] Qingfu Zhang, Wudong Liu, and Hui Li. The performance of a new version of MOEA/d on CEC09 unconstrained MOP test instances. In *2009 IEEE Congress on Evolutionary Computation*. IEEE, may 2009. doi: 10.1109/cec.2009.4982949.

[72] Tao Zhang, John Andrews, and Rui Wang. Optimal scheduling of track maintenance on a railway network. *Quality and Reliability Engineering International*, 29(2):285–297, mar 2012. doi: 10.1002/qre.1381.

[73] E. Zitzler and L. Thiele. Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach. *IEEE Transactions on Evolutionary Computation*, 3(4):257–271, 1999. doi: 10.1109/4235.797969.

[74] E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the strength pareto evolutionary algorithm. In *EUROGEN 2001. Evolutionary Methods for Design, Optimization and Control with Applications to Industrial Problems*, pages 95–100, 2002.

[75] Eckart Zitzler and Simon Künzli. Indicator-based selection in multiobjective search. In *Lecture Notes in Computer Science*, pages 832–842. Springer Berlin Heidelberg, 2004. doi: 10.1007/978-3-540-30217-9_84.

A Parameter analysis numbers

This appendix contains the ANOVA tables and interaction plots for the parameter analysis described in chapter 6.

A.1. As-is analysis

A.1.1. 2019 data

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
ParentSize	1	9.156E+03	9.156E+03	54.38	0.0858
OffspringSize	1	1.339E+04	1.339E+04	79.52	0.0711
SelectionType	1	1.552E+04	1.552E+04	92.20	0.0661
ConstraintsAllowed	1	1.906E+02	1.906E+02	1.13	0.4803
ShiftMutation	1	1.039E+04	1.039E+04	61.70	0.0806
BucketMutation	1	3.007E+02	3.007E+02	1.79	0.4090
FixMutation	1	5.163E+02	5.163E+02	3.07	0.3303
ParentSize:OffspringSize	1	9.449E+03	9.449E+03	56.12	0.0845
ParentSize:SelectionType	1	1.684E+04	1.684E+04	100.01	0.0634
ParentSize:ConstraintsAllowed	1	2.773E+02	2.773E+02	1.65	0.4214
ParentSize:ShiftMutation	1	1.262E+04	1.262E+04	74.97	0.0732
ParentSize:BucketMutation	1	2.686E-01	2.686E-01	0.00	0.9746
ParentSize:FixMutation	1	1.048E+02	1.048E+02	0.62	0.5747
OffspringSize:ConstraintsAllowed	1	2.999E+02	2.999E+02	1.78	0.4093
Residuals	1	1.684E+02	1.684E+02		

Table A.1: ANOVA table, as-is analysis, 2019 data, full model

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
ParentSize	1	9.156E+03	9.156E+03	53.81	0.0007
OffspringSize	1	1.339E+04	1.339E+04	78.69	0.0003
SelectionType	1	1.552E+04	1.552E+04	91.23	0.0002
ConstraintsAllowed	1	1.906E+02	1.906E+02	1.12	0.3384
ShiftMutation	1	1.039E+04	1.039E+04	61.05	0.0006
BucketMutation	1	3.007E+02	3.007E+02	1.77	0.2411
FixMutation	1	5.163E+02	5.163E+02	3.03	0.1420
ParentSize:OffspringSize	1	9.449E+03	9.449E+03	55.54	0.0007
ParentSize:SelectionType	1	1.684E+04	1.684E+04	98.97	0.0002
ParentSize:ShiftMutation	1	1.262E+04	1.262E+04	74.19	0.0003
Residuals	5	8.507E+02	1.701E+02		

Table A.2: ANOVA table, as-is analysis, 2019 data, reduced model

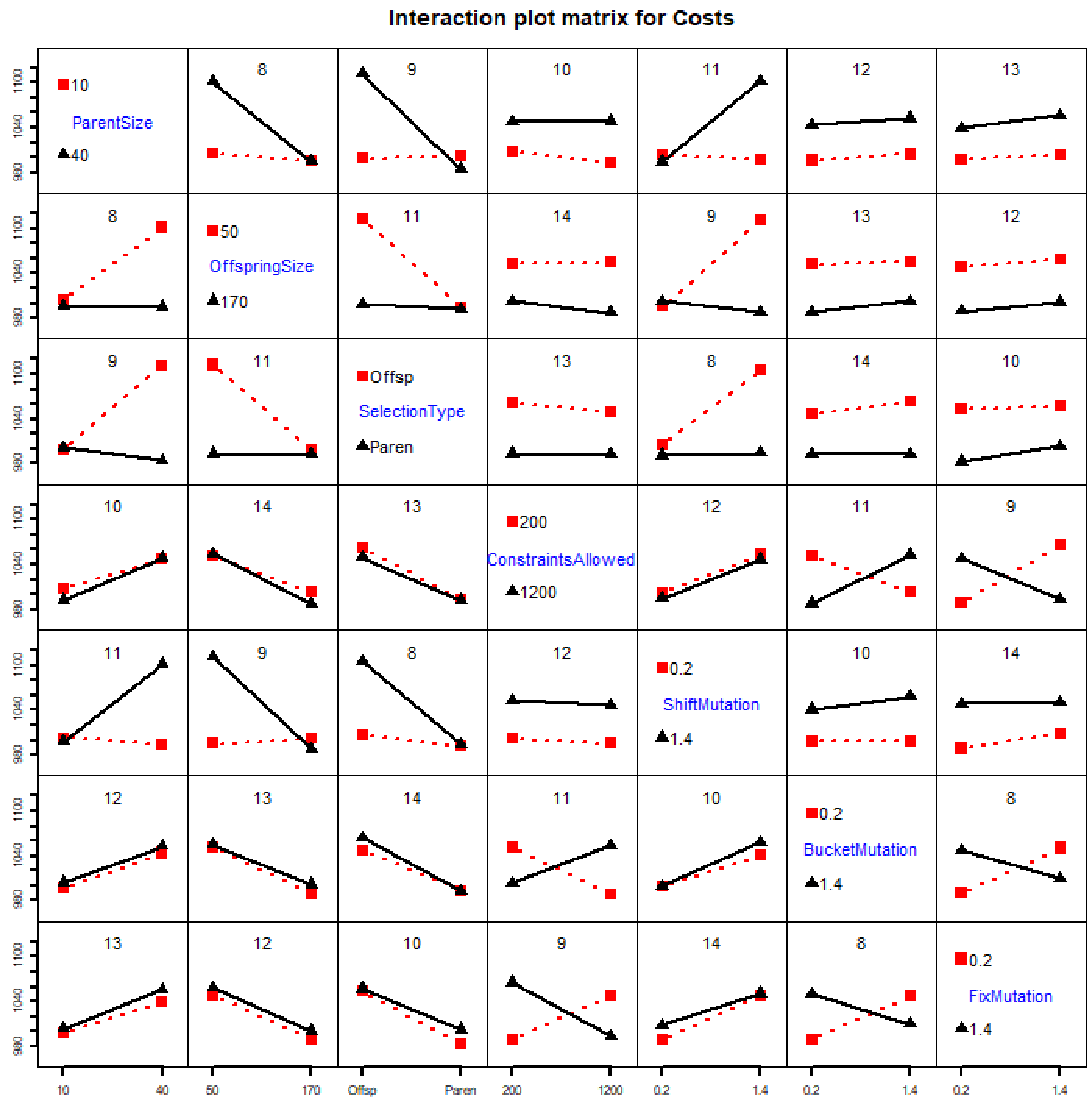


Figure A.1: Interaction plots, as-is analysis, 2019 data

A.1.2. 2020 data

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
ParentSize	1	6.603E+03	6.603E+03	4.83	0.2718
OffspringSize	1	8.215E+03	8.215E+03	6.01	0.2465
SelectionType	1	8.909E+03	8.909E+03	6.52	0.2377
ConstraintsAllowed	1	7.797E+02	7.797E+02	0.57	0.5882
ShiftMutation	1	2.784E+03	2.784E+03	2.04	0.3891
BucketMutation	1	1.051E+03	1.051E+03	0.77	0.5417
FixMutation	1	6.726E+01	6.726E+01	0.05	0.8610
ParentSize:OffspringSize	1	5.292E+03	5.292E+03	3.87	0.2993
ParentSize:SelectionType	1	7.073E+03	7.073E+03	5.18	0.2636
ParentSize:ConstraintsAllowed	1	6.633E+02	6.633E+02	0.49	0.6126
ParentSize:ShiftMutation	1	9.130E+03	9.130E+03	6.68	0.2350
ParentSize:BucketMutation	1	6.921E+01	6.921E+01	0.05	0.8591
ParentSize:FixMutation	1	2.130E+03	2.130E+03	1.56	0.4299
OffspringSize:ConstraintsAllowed	1	1.481E-02	1.481E-02	0.00	0.9979
Residuals	1	1.367E+03	1.367E+03		

Table A.3: ANOVA table, as-is analysis, 2020 data, full model

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
ParentSize	1	6.603E+03	6.603E+03	7.81	0.0383
OffspringSize	1	8.215E+03	8.215E+03	9.71	0.0264
SelectionType	1	8.909E+03	8.909E+03	10.53	0.0228
ConstraintsAllowed	1	7.797E+02	7.797E+02	0.92	0.3811
ShiftMutation	1	2.784E+03	2.784E+03	3.29	0.1294
BucketMutation	1	1.051E+03	1.051E+03	1.24	0.3158
FixMutation	1	6.726E+01	6.726E+01	0.08	0.7893
ParentSize:OffspringSize	1	5.292E+03	5.292E+03	6.26	0.0544
ParentSize:SelectionType	1	7.073E+03	7.073E+03	8.36	0.0341
ParentSize:ShiftMutation	1	9.130E+03	9.130E+03	10.79	0.0218
Residuals	5	4.229E+03	8.459E+02		

Table A.4: ANOVA table, as-is analysis, 2020 data, reduced model

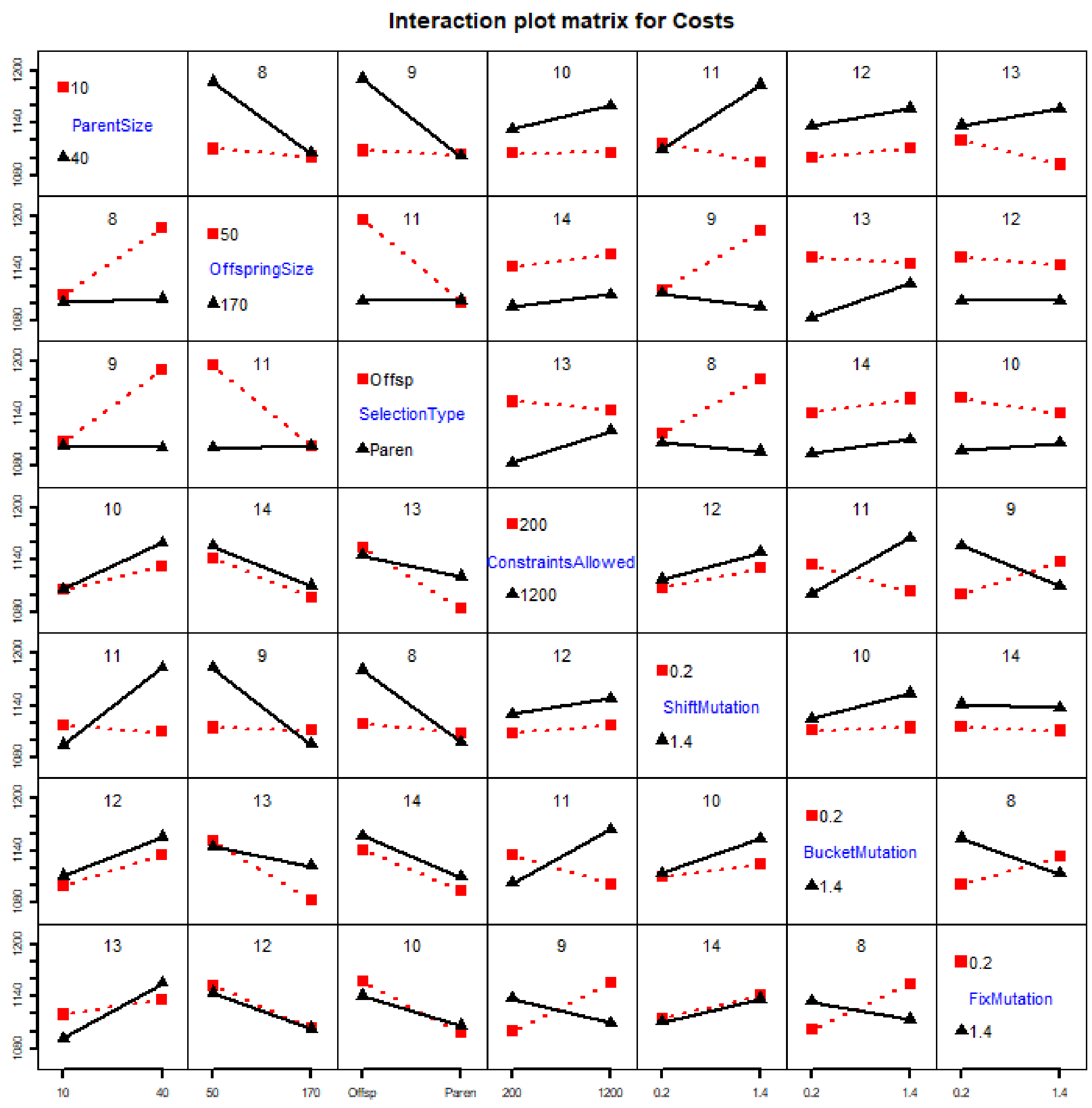


Figure A.2: Interaction plots, as-is analysis, 2020 data

A.2. Constraint adjusted analysis

A.2.1. 2019 data

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
ParentSize	1	1.648E+04	1.648E+04	42.45	0.0969
OffspringSize	1	2.166E+04	2.166E+04	55.81	0.0847
SelectionType	1	2.214E+04	2.214E+04	57.04	0.0838
ConstraintsAllowed	1	6.212E+02	6.212E+02	1.60	0.4258
ShiftMutation	1	1.684E+04	1.684E+04	43.38	0.0959
BucketMutation	1	1.015E+03	1.015E+03	2.62	0.3525
FixMutation	1	1.531E+02	1.531E+02	0.39	0.6430
ParentSize:OffspringSize	1	1.914E+04	1.914E+04	49.32	0.0900
ParentSize:SelectionType	1	2.446E+04	2.446E+04	63.01	0.0798
ParentSize:ConstraintsAllowed	1	8.269E+02	8.269E+02	2.13	0.3824
ParentSize:ShiftMutation	1	1.832E+04	1.832E+04	47.20	0.0920
ParentSize:BucketMutation	1	6.112E+02	6.112E+02	1.57	0.4283
ParentSize:FixMutation	1	3.096E+02	3.096E+02	0.80	0.5359
OffspringSize:ConstraintsAllowed	1	1.258E+03	1.258E+03	3.24	0.3227
Residuals	1	3.881E+02	3.881E+02		

Table A.5: ANOVA table, constraint adjusted analysis, 2019 data, full model

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
ParentSize	1	1.648E+04	1.648E+04	24.27	0.0044
OffspringSize	1	2.166E+04	2.166E+04	31.91	0.0024
SelectionType	1	2.214E+04	2.214E+04	32.61	0.0023
ConstraintsAllowed	1	6.212E+02	6.212E+02	0.92	0.3827
ShiftMutation	1	1.684E+04	1.684E+04	24.80	0.0042
BucketMutation	1	1.015E+03	1.015E+03	1.50	0.2758
FixMutation	1	1.531E+02	1.531E+02	0.23	0.6548
ParentSize:OffspringSize	1	1.914E+04	1.914E+04	28.20	0.0032
ParentSize:SelectionType	1	2.446E+04	2.446E+04	36.03	0.0018
ParentSize:ShiftMutation	1	1.832E+04	1.832E+04	26.99	0.0035
Residuals	5	3.394E+03	6.788E+02		

Table A.6: ANOVA table, constraint adjusted analysis, 2019 data, reduced model

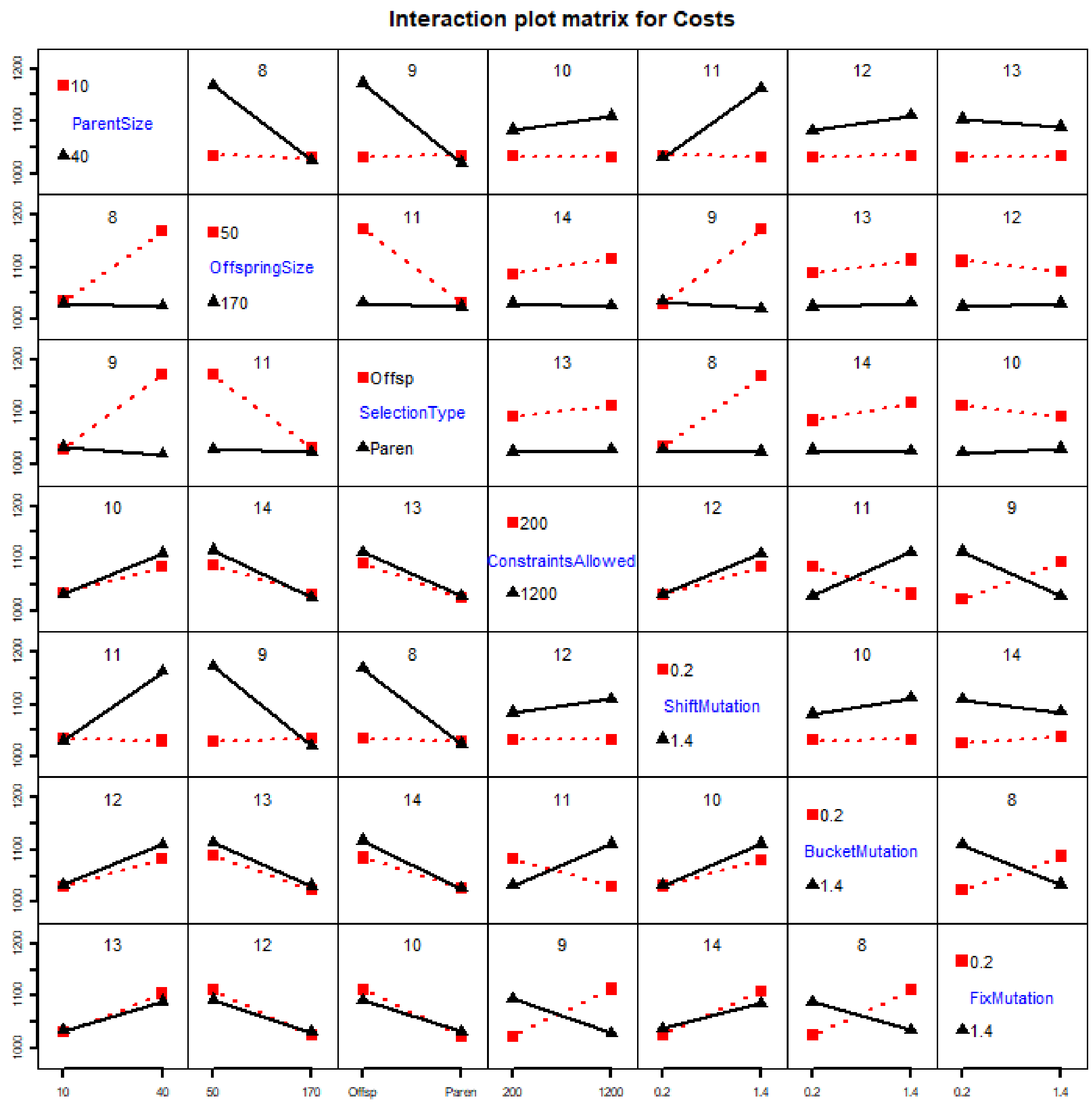


Figure A.3: Interaction plots, constraint adjusted analysis, 2019 data

A.2.2. 2020 data

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
ParentSize	1	3.171E+04	3.171E+04	3341.50	0.0110
OffspringSize	1	4.129E+04	4.129E+04	4350.35	0.0097
SelectionType	1	3.796E+04	3.796E+04	4000.04	0.0101
ConstraintsAllowed	1	4.605E+03	4.605E+03	485.25	0.0289
ShiftMutation	1	2.095E+04	2.095E+04	2207.65	0.0135
BucketMutation	1	4.237E+03	4.237E+03	446.44	0.0301
FixMutation	1	2.202E+03	2.202E+03	232.06	0.0417
ParentSize:OffspringSize	1	3.084E+04	3.084E+04	3249.74	0.0112
ParentSize:SelectionType	1	3.058E+04	3.058E+04	3222.23	0.0112
ParentSize:ConstraintsAllowed	1	4.003E+03	4.003E+03	421.86	0.0310
ParentSize:ShiftMutation	1	3.338E+04	3.338E+04	3516.82	0.0107
ParentSize:BucketMutation	1	2.819E+03	2.819E+03	297.09	0.0369
ParentSize:FixMutation	1	2.507E+01	2.507E+01	2.64	0.3512
OffspringSize:ConstraintsAllowed	1	2.596E+03	2.596E+03	273.56	0.0384
Residuals	1	9.490E+00	9.490E+00		

Table A.7: ANOVA table, constraint adjusted analysis, 2020 data, full model

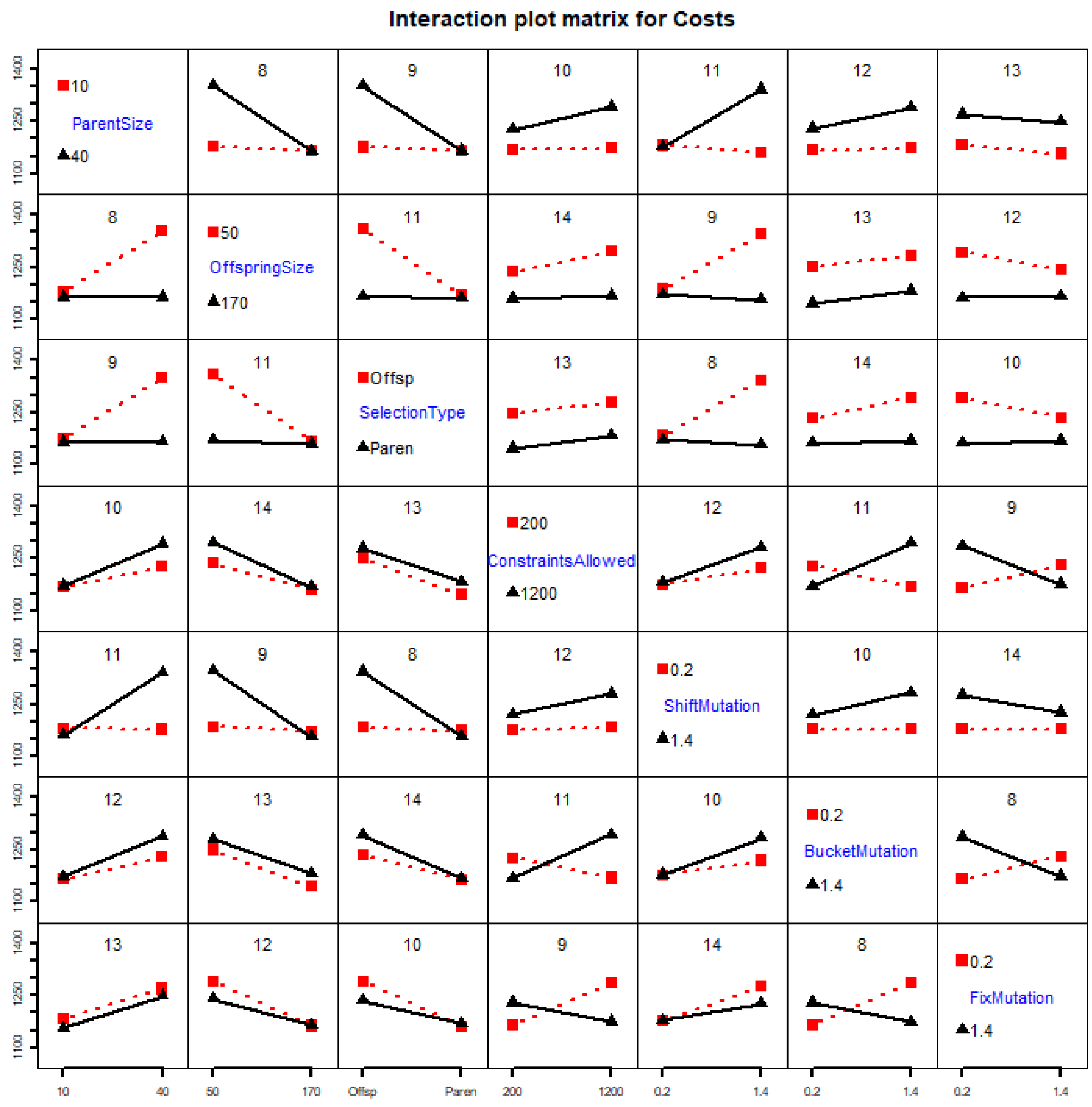


Figure A.4: Interaction plots, constraint adjusted analysis, 2020 data

A.3. Winsorized analysis

A.3.1. 2019 data

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
ParentSize	1	4.161E+01	4.161E+01	1.86	0.4025
OffspringSize	1	1.527E+02	1.527E+02	6.84	0.2325
SelectionType	1	1.954E+02	1.954E+02	8.75	0.2075
ConstraintsAllowed	1	2.474E-01	2.474E-01	0.01	0.9332
ShiftMutation	1	2.561E+01	2.561E+01	1.15	0.4782
BucketMutation	1	5.532E+01	5.532E+01	2.48	0.3603
FixMutation	1	1.452E+02	1.452E+02	6.50	0.2379
ParentSize:OffspringSize	1	1.252E+01	1.252E+01	0.56	0.5908
ParentSize:SelectionType	1	4.654E+02	4.654E+02	20.84	0.1373
ParentSize:ConstraintsAllowed	1	1.874E+01	1.874E+01	0.84	0.5279
ParentSize:ShiftMutation	1	2.811E-01	2.811E-01	0.01	0.9289
ParentSize:BucketMutation	1	8.804E-02	8.804E-02	0.00	0.9601
ParentSize:FixMutation	1	4.665E+01	4.665E+01	2.09	0.3853
OffspringSize:ConstraintsAllowed	1	1.220E+02	1.220E+02	5.46	0.2573
Residuals	1	2.233E+01	2.233E+01		

Table A.8: ANOVA table, winsorized analysis, 2019 data, full model

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
ParentSize	1	4.161E+01	4.161E+01	3.86	0.1068
OffspringSize	1	1.527E+02	1.527E+02	14.15	0.0131
SelectionType	1	1.954E+02	1.954E+02	18.11	0.0081
ConstraintsAllowed	1	2.474E-01	2.474E-01	0.02	0.8856
ShiftMutation	1	2.561E+01	2.561E+01	2.37	0.1840
BucketMutation	1	5.532E+01	5.532E+01	5.13	0.0730
FixMutation	1	1.452E+02	1.452E+02	13.46	0.0145
ParentSize:FixMutation	1	4.665E+01	4.665E+01	4.32	0.0922
ParentSize:SelectionType	1	4.654E+02	4.654E+02	43.13	0.0012
OffspringSize:ConstraintsAllowed	1	1.220E+02	1.220E+02	11.31	0.0201
Residuals	5	5.396E+01	1.079E+01		

Table A.9: ANOVA table, winsorized analysis, 2019 data, reduced model

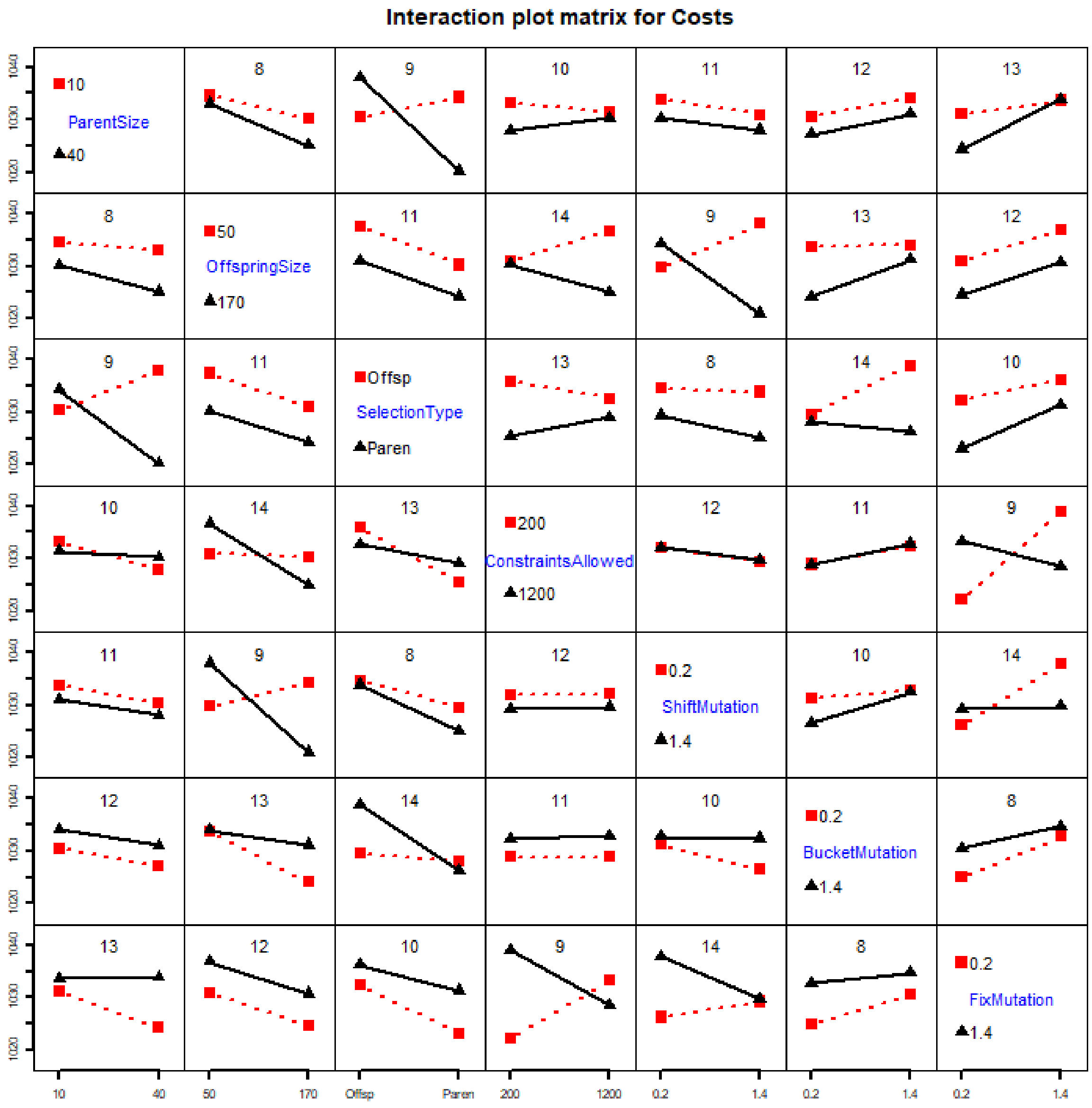


Figure A.5: Interaction plots, winsorized analysis, 2019 data

A.3.2. 2020 data

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
ParentSize	1	1.279E+02	1.279E+02	0.06	0.8448
OffspringSize	1	1.326E+03	1.326E+03	0.64	0.5701
SelectionType	1	7.878E+02	7.878E+02	0.38	0.6479
ConstraintsAllowed	1	6.488E+02	6.488E+02	0.31	0.6749
ShiftMutation	1	4.850E+02	4.850E+02	0.23	0.7128
BucketMutation	1	5.153E+02	5.153E+02	0.25	0.7052
FixMutation	1	2.061E+01	2.061E+01	0.01	0.9367
ParentSize:OffspringSize	1	7.826E+01	7.826E+01	0.04	0.8777
ParentSize:SelectionType	1	6.564E+01	6.564E+01	0.03	0.8877
ParentSize:ConstraintsAllowed	1	4.361E+02	4.361E+02	0.21	0.7259
ParentSize:ShiftMutation	1	2.535E+02	2.535E+02	0.12	0.7856
ParentSize:BucketMutation	1	1.147E+02	1.147E+02	0.06	0.8528
ParentSize:FixMutation	1	2.246E+03	2.246E+03	1.09	0.4868
OffspringSize:ConstraintsAllowed	1	7.332E+01	7.332E+01	0.04	0.8815
Residuals	1	2.068E+03	2.068E+03		

Table A.10: ANOVA table, winsorized analysis, 2020 data, full model

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
ParentSize	1	1.279E+02	1.279E+02	0.27	0.6277
OffspringSize	1	1.326E+03	1.326E+03	2.76	0.1573
SelectionType	1	7.878E+02	7.878E+02	1.64	0.2563
ConstraintsAllowed	1	6.488E+02	6.488E+02	1.35	0.2974
ShiftMutation	1	4.850E+02	4.850E+02	1.01	0.3609
BucketMutation	1	5.153E+02	5.153E+02	1.07	0.3476
FixMutation	1	2.061E+01	2.061E+01	0.04	0.8440
ParentSize:ConstraintsAllowed	1	4.361E+02	4.361E+02	0.91	0.3842
ParentSize:ShiftMutation	1	2.535E+02	2.535E+02	0.53	0.5000
ParentSize:FixMutation	1	2.246E+03	2.246E+03	4.68	0.0828
Residuals	5	2.399E+03	4.799E+02		

Table A.11: ANOVA table, winsorized analysis, 2020 data, reduced model

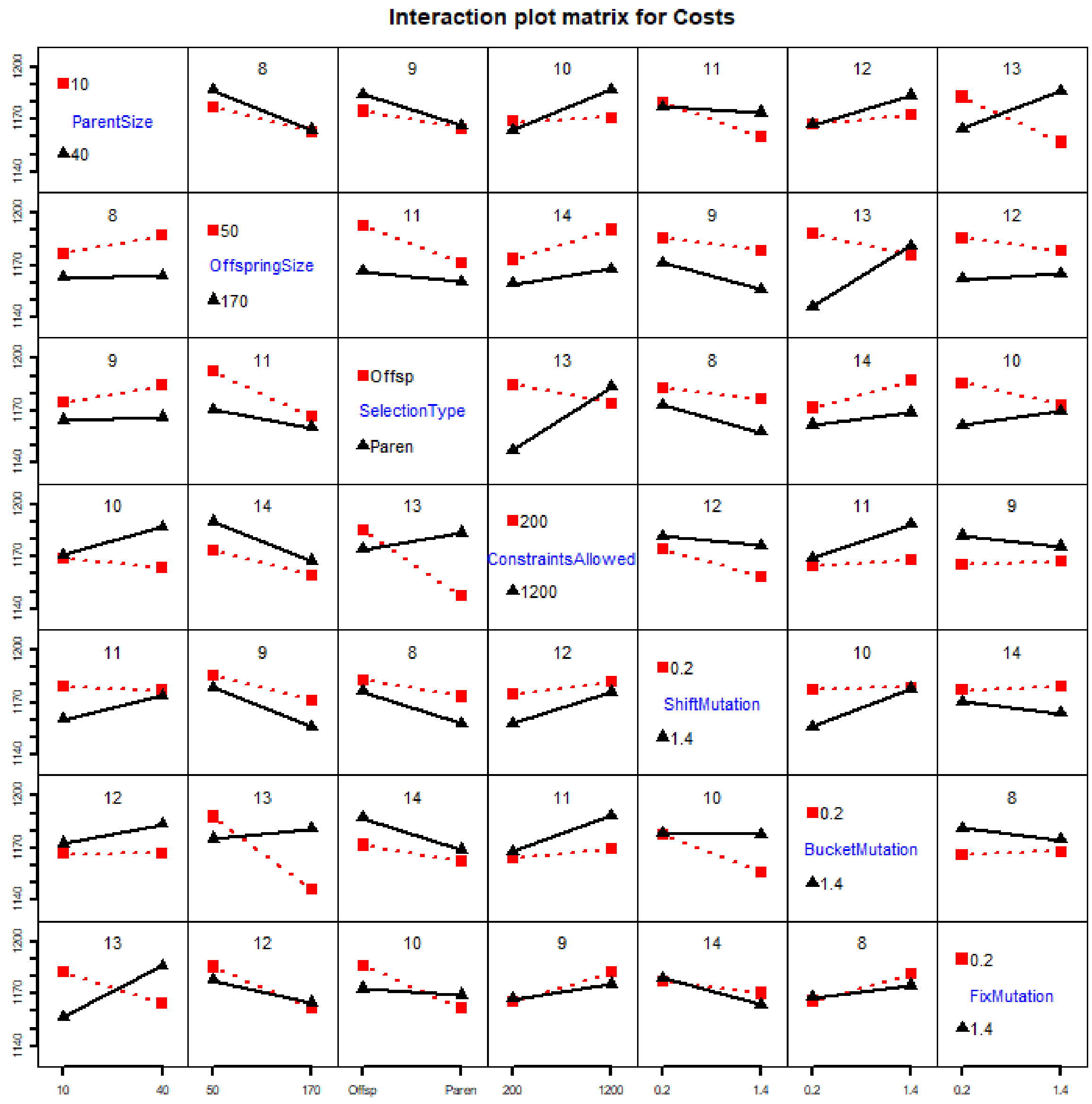


Figure A.6: Interaction plots, winsorized analysis, 2020 data

B
Evolution strategy robustness analysis results

Table B.1 shows the full results for the robustness analysis performed in section 8.1.

Run	Costs	Constraints	Maintenance	Availability	Soft constraints
1	980.1	10	606.9	339.8	33.4
2	993.5	10	607.3	352.9	33.3
3	988.5	10	606.6	347.8	34.0
4	978.7	10	606.6	337.8	34.3
5	981.4	10	609.1	335.0	37.3
6	994.7	10	607.7	352.7	34.2
7	994.2	10	607.4	348.9	37.8
8	987.6	10	608.5	342.5	36.7
9	983.2	10	605.0	345.1	33.1
10	980.5	10	605.5	340.1	35.0
11	986.9	13	608.5	339.5	38.9
12	988.5	14	606.6	344.7	37.2
13	992.9	14	607.4	347.7	38.6
14	976.6	15	606.5	333.4	36.8
15	991.0	15	607.8	348.4	34.8
16	974.5	16	604.6	333.8	36.1
17	982.1	10	606.3	339.3	36.5
18	980.1	10	606.3	340.5	33.3
19	979.3	10	605.8	339.7	33.8
20	996.9	13	607.4	352.2	37.3
21	975.6	10	606.6	333.6	35.4
22	986.8	10	606.6	343.7	36.5
23	981.7	10	606.6	339.3	35.8
24	980.6	10	605.8	341.3	33.5
25	985.9	10	605.2	347.6	33.1
26	980.9	14	607.0	337.9	35.9
27	993.3	10	605.8	351.7	35.8
28	994.7	17	607.3	350.9	36.5
29	992.0	11	606.3	349.9	35.8
30	1000.9	14	608.1	355.9	37.0
31	987.0	10	606.5	346.0	34.5
32	978.6	10	606.0	334.4	38.2
33	990.0	10	608.1	344.8	25.1
34	996.6	14	608.5	352.5	35.5
35	998.5	10	608.6	353.1	36.9
36	981.4	10	605.6	342.9	32.8
37	998.1	16	609.4	355.5	33.2
38	999.3	10	607.4	356.8	35.1
39	987.3	16	607.0	345.5	34.8
40	977.6	10	607.7	334.5	35.4
41	998.2	10	608.8	353.2	36.2
42	983.4	12	606.8	344.6	32.0
43	977.3	10	608.0	337.0	32.3
44	980.3	10	608.1	338.1	34.0
45	987.6	10	605.5	345.9	36.2
46	996.2	10	607.5	352.3	36.5
47	978.6	10	607.6	339.1	31.8
48	982.1	16	608.0	340.5	33.6
49	965.1	10	609.1	321.6	34.4
50	989.0	11	606.7	346.7	35.6
51	984.5	14	606.0	341.0	37.6
52	981.8	10	605.3	343.2	33.3
53	980.9	10	608.6	337.2	35.1

Table B.1: Results of robustness analysis of evolution strategy