Optimizing Shunting Operations
A case-study at Kijfhoek shunting yard

Teun Druijf

April 14, 2022

Delft University of Technology

TU Delft Delft University of Technology

Challenge the future

Optimizing Shunting Operations
A case-study at Kijfhoek shunting yard

by

Teun Druijf

to obtain the degree of Master of Science in Applied Mathematics (Optimization Specialization),
at the Faculty of Electrical Engineering, Mathematics and Computer Science
at the Delft University of Technology,
to be defended publicly on Thursday April 14, 2022 at 9:00.

Student number:	5253608	
Project duration:	August 23, 2021 - April 14, 2022	
Thesis committee:	Dr. ir. L. J. J. van Iersel,	Delft University of Technology
	Dr. M. T. J. Spaan,	Delft University of Technology
	M. F. Brussen MSc,	DB Cargo Nederland

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft

Abstract

In the current economy, there is an increasing focus on sustainability. Green transport solutions, like rail freight, are becoming more and more popular. We will look into the shunting operations at Kijfhoek shunting yard. This yard functions as the central hub for DB Cargo Nederland and connects the Port of Rotterdam with the hinterland. About 2000 cars are sorted at this yard each week. At the yard, there are 43 parallel classification tracks.

Shunting problems come in many shapes and sizes but are generally hard to solve. To that end, we will use both exact algorithms and heuristics to solve the shunting problem at Kijfhoek within reasonable time. In practice, it is not known at the beginning of the planning horizon which cars will arrive. To that end, we will create general solutions which will be updated over time when new information becomes available. We test how well the solutions perform and/or can be reoptimized with the latest information using a simulator.

We find that our solution approaches can generate solutions for instances with 28 classification tracks of similar quality as solutions found for the same instances with 43 classification tracks with complete information at the beginning of the planning horizon. Furthermore, with incomplete information, we find good solutions with 32 classification tracks.

Keywords: Shunting Operations, Integer Programming, Simulated Annealing, Simulation

Preface

During the past eight months, I had the opportunity to combine mathematics and computer science in the fascinating area of logistics. It may seem like a rather trivial problem; sorting cars. However, if you take a deep dive into the subject, you are confronted with the many details that make the problem hard yet even more interesting. With this thesis, I hope to introduce you to the wonderful world of shunting.

This project wouldn't have been possible without the help of others and I would like to take the opportunity to name a few in particular.

First and foremost, I would like to thank my daily supervisors, Leo van Iersel and Maarten Brussen for their guidance, inspiration, and feedback. I would also like to thank the third member of the thesis committee, Matthijs Spaan, for his time.

In addition, I wish to express gratitude to all the colleagues at Deutsche Bahn for their guidance, inspiration, and confidence. Without their expertise and corporation, this thesis wouldn't have been possible.

Furthermore, I am grateful to Pieter Knops for getting me in touch with Deutsche Bahn.

Finally, I am thankful to my family for their support and encouragement.

Teun Druijf
Houten, April 2022

Contents

1 Introduction

In the current economy, there is an increasing focus on sustainability. Green transport solutions, like rail freight, are becoming more and more popular. However, the limited infrastructure makes it more and more challenging to keep up with demand. Smart planning solutions can help increase capacity without an infrastructure expansion.

In this thesis, we will look at the shunting operations at the Kijfhoek shunting yard. This yard functions as the central hub for DB Cargo Nederland and connects the Port of Rotterdam with the hinterland.

Recently, DB Cargo has started a project to offer shunting services to third parties at Kijfhoek. This will impact the operations of DB Cargo at Kijfhoek as they will have to use their resources to perform shunting operations for third parties. Furthermore, the available track capacity for DB Cargo will also decrease as more parties wish to use the limited tracks available at Kijfhoek.

Apart from the fact that an increasing number of rail freight enterprises see Kijfhoek as a way to improve their undertakings, starting in 2023, the yard will be renovated for 18 months. During the renovations, the available capacity is cut in half.

To gain insights into the capacity in a non-discriminatory way, we will describe, test, and implement shunting algorithms for the Kijfhoek shunting yard.

We will see how the algorithms perform in real-life test cases compared to the existing scheduling solutions and make recommendations on how the algorithms can be used in practice.

The shunting problem at Kijfhoek has been looked at by Knops ([25]). They used Simulated Annealing to find better solutions when compared to the current solution strategy. However, as they point out, they have made certain assumptions and simplifications.

Shunting problems come in many shapes and sizes and are generally hard to solve. In fact, the decision problems related to the shunting problem we will consider are $\mathcal{NP}$-complete[5, 7, 13]. To that end, we will not only look into exact algorithms in this thesis but also propose heuristics to solve the problem within reasonable time. Additionally, we will introduce ways to heuristically 'presolve' the instances to make them easier to solve.

We can now outline the following research questions for this thesis:

1. Is there a way to find optimal solutions for the shunting problem at Kijfhoek? How well do heuristics perform relative to these solutions, both measured in solution quality as well as computation time?

2. Can the solutions found by the algorithms be used in practice - possibly under some conditions?

3. Is there a way to recover solutions after a perturbation occurred, either before or during the planning horizon?

4. What is the impact of a reduction of the number of classification tracks on the quality[1] of the solutions?

[1]When we mention the 'quality' of a solution in this thesis, we mean the number of *carrolls* (i.e. number of cars rolled over the hump) and the number of reclassifications.

The thesis is structured as follows. First, in Chapter 2, we will present the problem and introduce the necessary terminology for the shunting yard problem. Next, in Chapter 3, we will go over some of the literature that has already been published on the subject. We will look at some of the simple well-known algorithms for shunting problems.

Then, in Chapter 4, we will present multiple Integer Programming formulations used to solve the problem to optimality as well as a heuristic to preprocess instances before we will solve them. With this preprocessing heuristic, we will try to optimize the order in which the arrival trains are rolled over the hump, otherwise, we use first-in-first-out. In Chapter 5, we will describe how we use Simulated Annealing to solve the problem heuristically.

Thereafter, we will present our two simulators used to test the algorithms in Chapter 6. In Chapter 7, we will show how well the algorithms perform, compared to each other and to the current planning strategy. Finally, yet importantly, in Chapter 8, we will answer the research questions, point to risks and opportunities, make some recommendations, and discuss areas of future research.

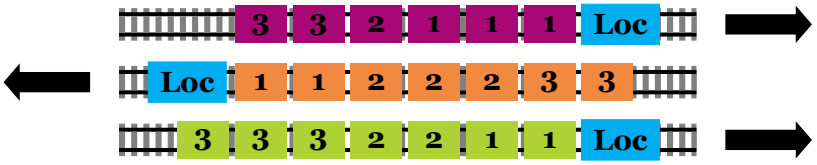
Author's note

For the sake of clarity, before continuing with the thesis, we would like to define some of the notations we use in this thesis.

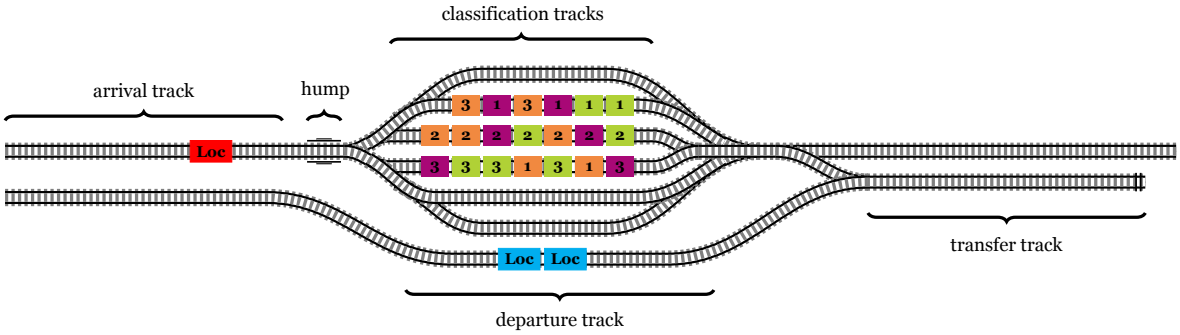
Notation When we talk about the `log`, we mean the natural logarithm. We use `lg` for the 2-log.

Legend Railway States In this thesis, we will use Railway States to visualize our examples. At the beginning of each example, we will present the 'goal trains' (see Railway State 1.1), this is an overview of how the cars should be sorted at the end of the example. An arrow indicates the direction of the train.

In Railway State 1.2 we present an example of how we will visualize our examples. We will use two types of locomotives, shunting locomotives will be red and ordinary locomotives blue. The different infrastructure components - tracks and the hump - will be indicated each time we introduce a new layout.



Railway State 1.1: Goal trains.



Railway State 1.2: Example Railway State.

2
Problem Description

In this thesis, we will consider the freight shunting problem at the Kijfhoek shunting yard in the Netherlands. The yard is located near the port of Rotterdam and is directly connected to the *Maasvlakte* by the *Betuweroute*. Kijfhoek plays an integral role in the railway operations to and from the Port of Rotterdam.

Before we can look into possible shunting algorithms, it is vital to understand the complex situation we are working with. In this chapter, we will introduce the world of hump yard shunting and all considerations that have to be taken into account when making a planning. First, we will describe the general idea of the operations at Kijfhoek in Section 2.1. Next, in Section 2.2 we will outline the current planning solution. Lastly, in Section 2.3, we come to one of the main motivations for this thesis: 'Shunting as a service'.

Betuweroute

The *Betuweroute* is a 170-kilometer dedicated cargo rail line through the Netherlands. Starting at the *Maasvlakte*, the line passes through Kijfhoek onto the Dutch-German border near Zevenaar (NL) | Emmerich (DE). After numerous delays, construction finished in 2007. In the first years, the line was used scarcely, but over time, more and more trains started using the €4.7 billion line.

In 2019, 57% of all cargo trains going into Germany used the *Betuweroute*, for about 40 million tons worth of goods, the equivalent of 2 million truck loads. ProRail plans to increase this number to 60 million tons by 2030. One necessary condition to reach the full potential of the *Betuweroute* is the extension into Germany. Currently scheduled to open around 2030, the *Betuweroute* will become 70 kilometers longer, going all the way to Oberhausen. [1, 33–35]

2.1. General Workflow

Before discussing the operations at Kijfhoek, it is important to get an idea of the role of Kijfhoek. In the world of Rail Cargo Operations, there are two varieties. The first is the most simple - Block transshipment. Here, a train | vehicle | boat | plane is sent to pick up a load and it will go directly to the destination to drop it off. It may pass through (and park temporarily) at a hub, but the load will not change in any way.

The other variety is Unit Cargo transshipment. Unit Cargo transshipment can best be compared with how the post office works. Trains are sent out from a central hub (e.g. Kijfhoek) to deliver cars to multiple different destinations. On the way back, the train will pick up new cars. These will be sorted at Kijfhoek before being placed on another train for delivery. Kijfhoek acts as a sorting center in the hub-and-spoke system that connects the Port of Rotterdam and its many terminals to the hinterland.

Hub-and-Spoke vs. Point-to-Point

In the world of transportation - goods or people - there are two main ways to design the transportation network: Hub-and-Spoke and Point-to-Point. In practice, most networks are a combination of both - although there is most often one variety that has the upper hand.

As the names suggest, the Hub-and-Spoke (see Figure 2.1) model uses a central hub that acts as a sorting center, all cargo is brought here before being sent out to its final destination or routed to another hub from which it is delivered.

The Point-to-Point (see Figure 2.2) model is not centered around one hub and cargo is transported directly from its origin to the destination. This means that there are more different routes as many destinations need to be connected directly with each other.

Each variety has its (dis)advantages. With a Hub-and-Spoke model, generally, fewer but larger transportation means are used and the hub is used to get high load factors. The Point-to-Point model has generally more different direct routes and therefore requires more but smaller transportation means.

Looking at our case, we see that by using Kijfhoek as a central hub for the Rotterdam region, locations can be served more often because trains serve more than one destination making it easier (and cheaper) to operate at a higher frequency and thus increasing the level-of-service for the client.

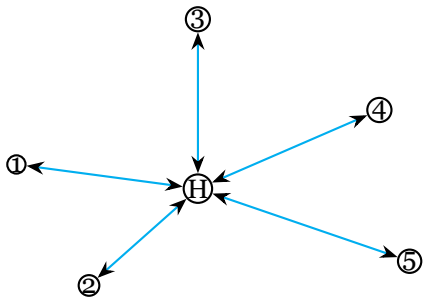


Figure 2.1: Hub-and-Spoke

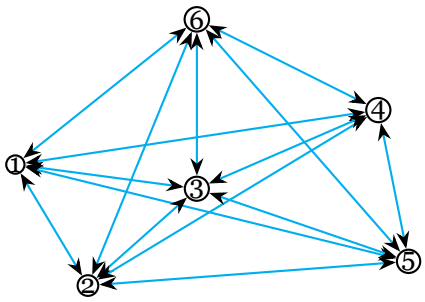
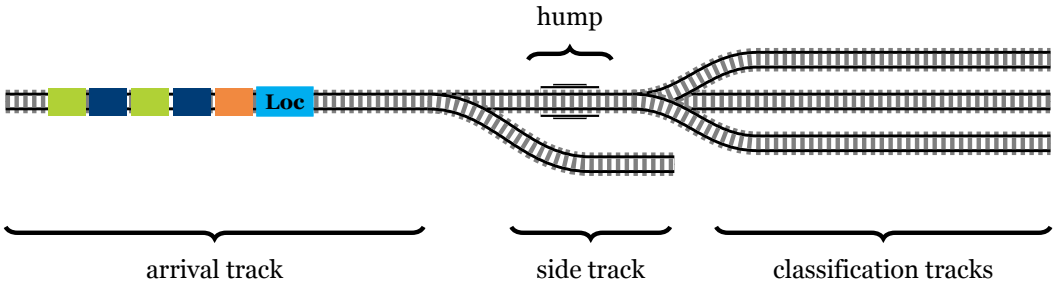


Figure 2.2: Point-to-Point

The incoming trains are called **assorted** trains. This means that they contain freight that needs to be sorted because it does not share the same destination. This is where Kijfhoek comes into play as a shunting yard. In the following example (Example 2.1), we will introduce the basic workflow at a hump yard when an assorted train comes in. The operations will be further expanded upon afterward.

Example 2.1 (Basic workflow at a hump yard)**.** Kijfhoek is a so-called 'hump yard'. In this example, we will look at one incoming train where we need to sort the cars into three different outbound trains, the colors of the cars correspond to the outbound that they are a part of.

First, the train arrives. It consists of a locomotive and several cars (see Railway State 2.1). There are cars for three different outbound trains (green, blue, and orange) and there are three classification tracks available. This means that we can assign each outbound train to a classification track and classify the cars in such a way that they end up on the track of their respective train.



Railway State 2.1: Start scenario for Example 2.1.

Before the cars can be pushed onto the hump, the ordinary locomotive will move out of the way and a special shunting locomotive will position itself behind the cars. This way it can push them forwards (see Railway State 2.2). The cars that need to go onto different classification tracks will be loosened. After this is completed, the cars will be pushed onto the hump (but not yet over) (see Railway State 2.3).

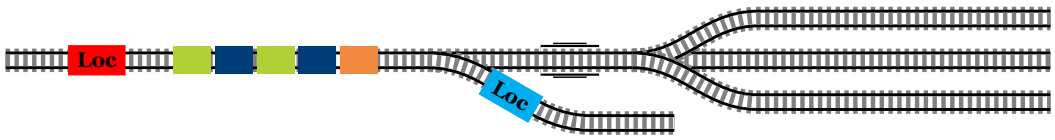
Then, the driver will flip a switch in the locomotive and the traffic controller of the hump yard (employed by ProRail, not DB Cargo in the case of Kijfhoek) will take remote control of the locomotive and start rolling.

Loosening connections & Clubbing

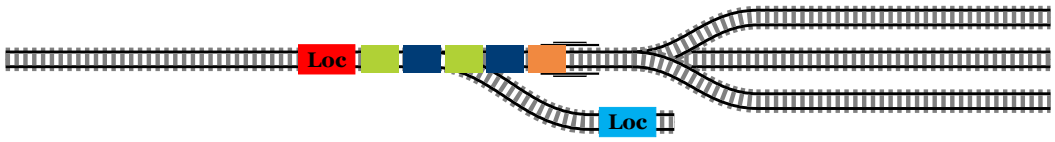
If two adjacent cars need to go to different tracks, the connections will be loosened. This happens at a flat arrival track.

When a connection is loosened, all air pipes will be completely disconnected. The physical connection between the two cars will be unscrewed as much as possible without actually disconnecting the cars. This way, the cars are only 'on the loose' when they strictly have to be.

At the top of the hump, right before the cars are pushed over, a *clubber* will use a long pole to disconnect the loosened connections - we will call this *clubbing*. Only then, the cars are really disconnected and the different groups of cars can go down the hump and be classified.

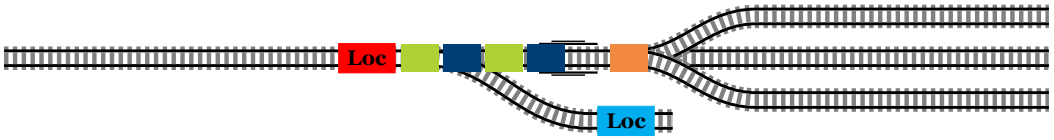


Railway State 2.2: The ordinary locomotive goes to a side track and the shunting locomotive moves in.

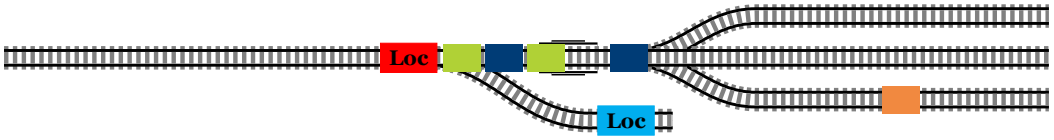


Railway State 2.3: The shunting locomotive pushes the cars onto the hump.

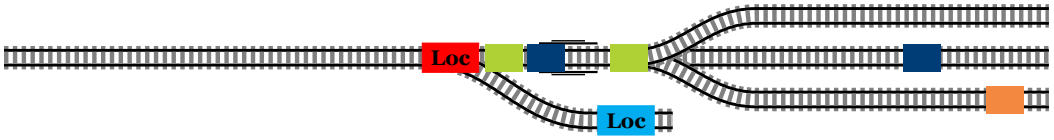
Once the first car is pushed on the hump, the loose connection will be *clubbed*. By the time the car descends on the other side, the switches will automatically change so that the car is put on the right track (see Railway State 2.4 through 2.7). A series of brakes will make sure that the car will be slowed down to approximately 5-6 kilometers an hour, taking the weight, current speed (measured by radar), and wind conditions into account.



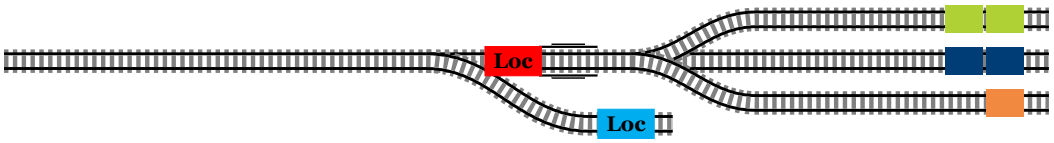
Railway State 2.4: The first (orange) car is pushed over the hump and will move to the third classification track.



Railway State 2.5: The second (blue) car is pushed over the hump. The switches are now set that the car will roll onto the second classification track.

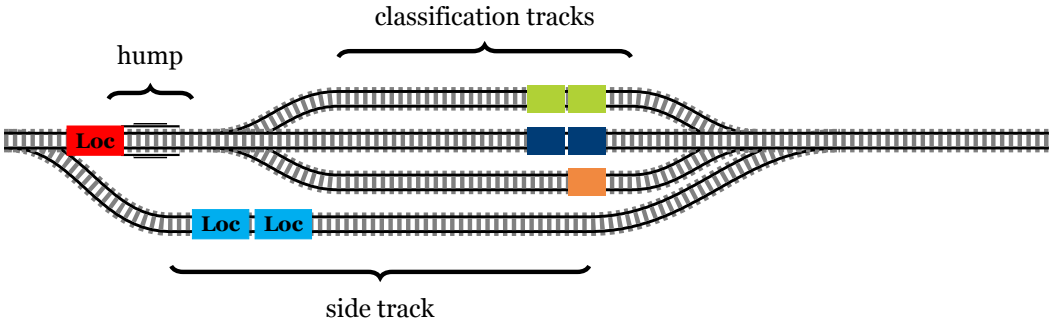


Railway State 2.6: The third (green) car is classified to the first classification track.

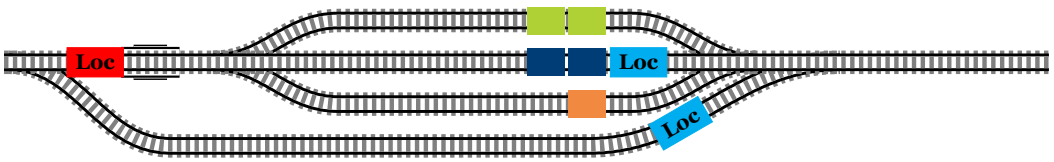


Railway State 2.7: The classification is complete, cars sharing the same destination are grouped together on separate tracks.

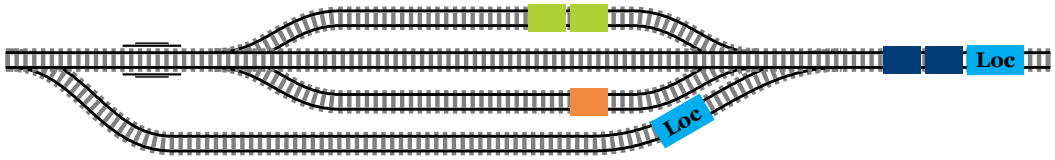
Now, if all cars for a given train are nicely lined up on a single classification track, they will be linked together and a locomotive is coupled so that the train can leave (see Railway State 2.8 through 2.10). This can either be directly from the classification tracks - depending on the infrastructure - or the cars can be temporarily stored at a departure track.



Railway State 2.8: Continuation of Example 2.1. Compared to Railway State 2.7, the end of the classification tracks is now visible.



Railway State 2.9: The first ordinary locomotive moves into place, getting ready to depart with the blue cars.



Railway State 2.10: The first outbound train leaves with the blue cars. The next locomotive gets into position to pick up its cars.

◊

The situation at Kijfhoek is more complex. First of all, in this example, we were able to apply single-stage sorting. This means that all cars can be sent directly to a track with other cars with the same destination. Unfortunately, this is generally not the case as the number of classification tracks is limited. In Example 2.2, we will introduce the concept of reclassification. There, cars are pulled from a classification track, over the hump, back onto an arrival track. Then, the cars are rolled in once more.

Before going to these more complex examples, it is paramount to create clarity over some of the terminology we will use. Below, we will describe the seven possible operations at a hump yard like Kijfhoek.

Arrival A train arrives at the hump yard, this must be at one of the arrival tracks. Which track is used should be specified in the planning.
Duration: Upon arrival, there must be a technical inspection and the cars must be decoupled. This takes about 9 minutes per 100 meters of train.

Roll Push the cars, which are disconnected, from the arrival track onto the hill to be classified. This operation can only be done by a shunting locomotive. The classification track onto which the different cars are rolled should be specified.
Duration: About 20 minutes per roll (remark that this is an average and highly dependent on the number of cars and whether or not all cars can be rolled over the hump).

Bypass Some cars are not allowed to go over the hump. There are three different reasons why a car is not allowed. The car might be too large to be physically pushed over the hill without the risk of 'bottoming out'. The second has to do with safety, some cars contain dangerous materials and are therefore not allowed to be 'on the loose', like cars containing chlorine. Lastly, clients can request that their cars are not rolled into the classification yard. For example, because the car contains vases. About 10-15% of all cars have to take the bypass.

Pull The pull operation is where the cars are pulled from the classification tracks to one of the arrival tracks so that they can be classified again. They are pulled over the hump (which therefore can't be used during this operation) and before the train can be rolled in again, it must be brought to a standstill on an arrival track.
Duration: Before a train can be pulled, a small brake check must be performed, this takes about 20 minutes.

Push Cars are pushed from the arrival tracks to either a classification track or departure track without being disconnected. This operation must be performed by a shunting locomotive.

Transfer When all cars for a train are present at a classification track, but the train does not yet need to leave or the classification track does not allow departures, the entire set of cars is transferred from a classification to a departure track.
Duration: Before a transfer, a check of 15 minutes is necessary.

Departure A train leaves the hump yard. This can either be from a departure track or a selected set of classification tracks.
Duration: Before departure, the cars must be coupled and necessary checks have to be performed. This takes 30 minutes and 9 minutes per 100 meters of train.

In Figure 2.3, we present a schematic overview of Kijfhoek and all the operations that can take place between the different locations. Working from left to right, we see that arrival tracks, which are located in the Northern area of the yard, can be entered from both sides (North and South). From the arrival tracks, they will either be classified (roll and/or bypass) or pushed onto one of the classification or departure tracks.

The classification tracks are the core of the yard and from here, cars can be transferred onto a departure track, ready to depart, be pulled over the hill for reclassification or (in some cases[1]) depart in the southbound direction.

The departure tracks are located on either side of the yard, however, due to the availability of air pressure tanks needed for the brakes, the western tracks 156 - 159 are highly favored as cars can be put here without the need for the presence of a locomotive to ensure the cars stay where they are.

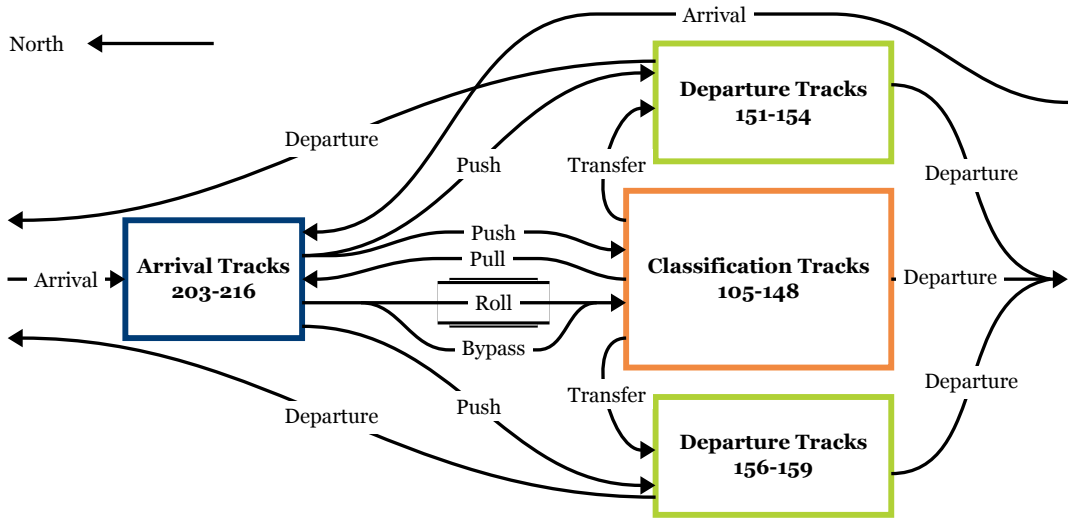
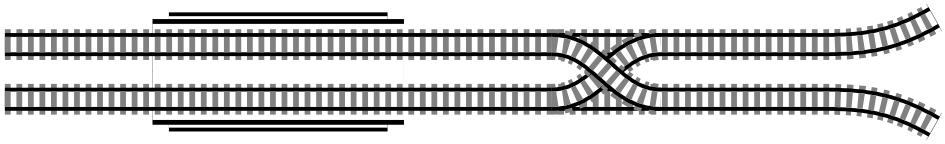


Figure 2.3: Schematic overview of the operations at Kijfhoek

Simultaneous Classification

The hump at Kijfhoek is designed to allow for simultaneous classification. With simultaneous classification, two trains will be rolled in at the same time. However, when this is done, the yard should be partitioned into two halves to prevent accidents. This means that the cross switch located right after the hump (see Railway State 2.11) can't be used in this case.

However, due to a combination of factors, such as safety concerns and the need to partition the yard during the operation, this feature has not been used for quite some time. This led to the decision to stop the support for this feature in the software used at the traffic control center - Kijfdis. Also, in the specification of the new version of Kijfdis - Kijfdis Light, this is no longer requested as a feature to be included. Therefore, unless otherwise stated, we will assume that the hump yard functions as one on which only one train at the time can be rolled in.

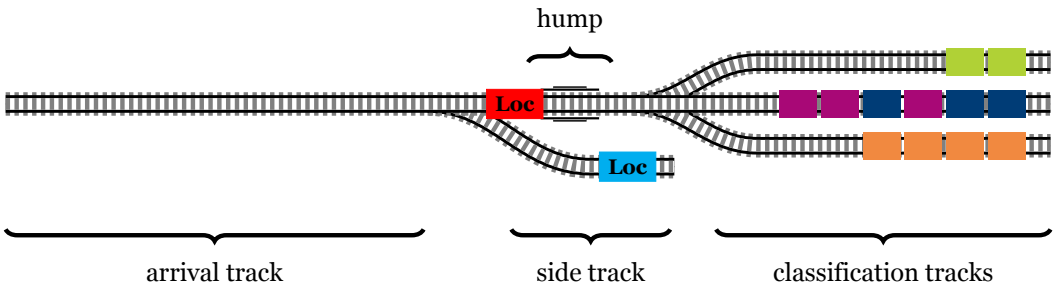


Railway State 2.11: Example of crossswitch as used at the hump at Kijfhoek.

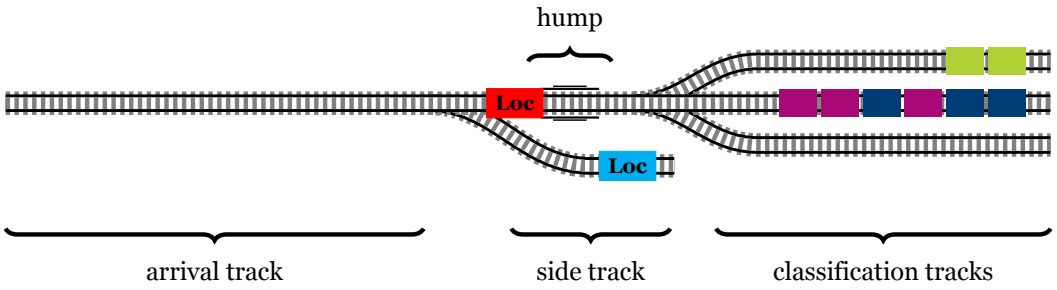
[1]Only from tracks 105-126.

Example 2.2 (Pull and Roll operation to reclassify)**.** In this example, we will look into the exact workings of a reclassify operation. Here, cars are pulled from a classification track to be rolled in again. This could be necessary as there was not yet a track available for each outbound train at the time of the initial roll-in. In that case, cars for multiple outbound trains are placed on the same track until an empty classification track is available.

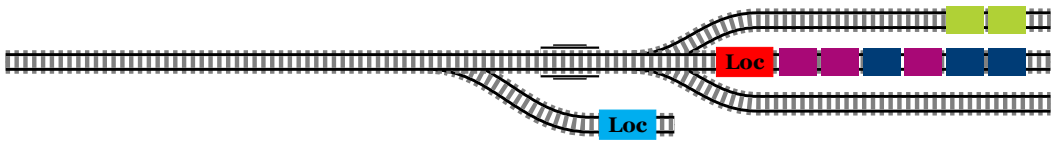
The infrastructure in our example is the same as in Example 2.1. As can be seen in Railway State 2.12, all tracks are filled. On the top track, we have only green cars, on the middle track, we have a combination of purple and blue cars, and on the bottom track, we have orange cars. First, the 'orange train' leaves the yard and this creates an empty track (see Railway State 2.13). On this track, we will place all purple cars. The first step is to maneuver the shunting locomotive in place (see Railway State 2.14).



Railway State 2.12: Start scenario for Example 2.2.

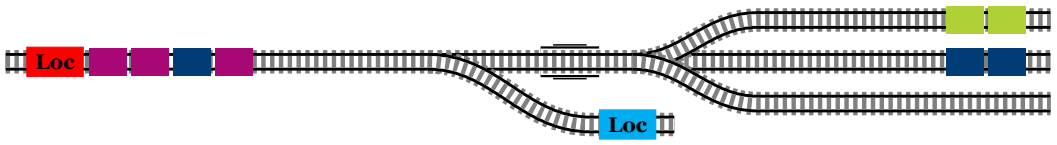


Railway State 2.13: The orange train has left the yard and now there is an empty train on which we can place the purple cars.

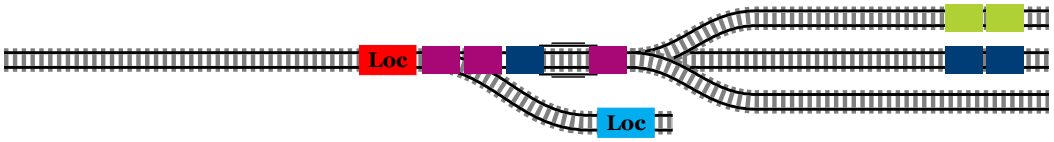


Railway State 2.14: The shunting locomotive goes over the hump to pick up the cars that need to be reclassified.

Next, the cars are pulled over the hump (see Railway State 2.15) until they are completely over the hump and onto an arrival track before the roll-in procedure is started (see Railway State 2.16). The last two blue cars are left on the track as it is not necessary to pull and roll them onto the track they originally came from.

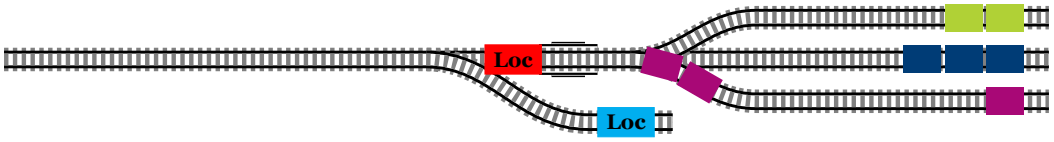


Railway State 2.15: The shunting locomotive arrived with the cars at an arrival track. Now, the cars will be prepared for classification.

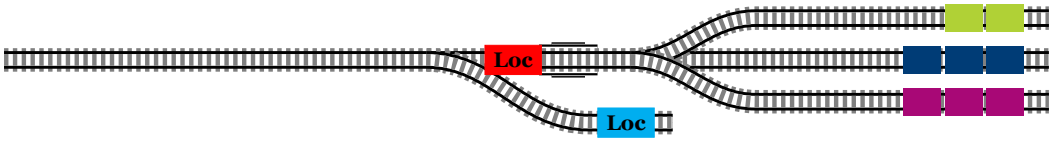


Railway State 2.16: The first cars are reclassified. The purple cars will move to the third track. The blue car will join the other blue cars at the second track.

The blue cars will be pushed onto the middle track and the purple ones to the previously empty track until all cars are classified. In the end, all cars are now sorted by color. (see Railway States 2.17, and 2.18).



Railway State 2.17: The last cars move together over the hump.



Railway State 2.18: The reclassification is completed. On each of the three classification tracks, there are cars for only one destination.

When all cars are rolled over the hump, the reclassification is complete and the hump is available again for another reclassification or new arrival. ◊

2.1.1. Locomotives

As mentioned before, there are restrictions on which locomotives can perform certain operations. We have the distinction between the shunting locomotive and 'ordinary' locomotives. The shunting locomotives are the only locomotives allowed to perform the roll operation. In total, there are four shunting locomotives at Kijfhoek (all property of DB Cargo), of which at most three are used at the same time. This way there is always one standby in case of repair or a mechanical breakdown.

These shunting locomotives are equipped with a switch that allows the *heuvelcoördinator* (hump traffic controller) from ProRail (the owner and operator of the hump yard) to take control of the locomotive. Then, a supervised, yet automated, program takes over and will adjust the speed of the locomotive to make sure the cars go over the hill at the correct speed.

The 'ordinary' locomotives are the ones that operate the trains that arrive at and depart from Kijfhoek. They can also be used to perform the transfer operations, under the condition that the locomotive is remote controlled, most of the locomotives Deutsche Bahn uses have this feature.

2.1.2. Order of cars per train

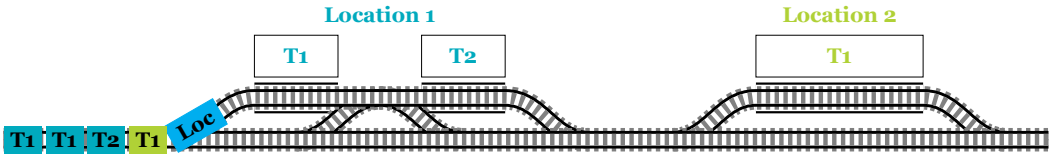
In the previous section, we have seen that cars are sorted by outbound train. Next to this *rough* sort, the cars that are part of the same train are also *fine* sorted. This is done by destination group. In Example 2.3, we will show the importance of the order of the cars within the train for a smooth delivery.

Definition 2.1 (Destination group)**.** A **destination group** is a set of cars that share a common destination | client. Cars that are part of the same destination group can be in any relative order. An outbound train might have multiple destination groups. The order of the destination groups within the train is fixed.

Example 2.3 (Wagon delivery - multistation)**.** Consider the following situation (see Railway State 2.19), we have a train visiting two locations, Location 1 and Location 2. The colors of the cars correspond to the locations and the T_ indicates the specific terminal.

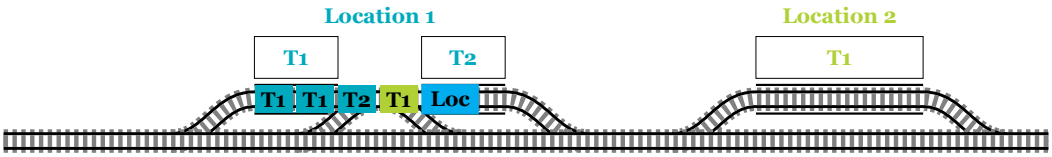
At Location 1 there are two terminals, at the first terminal, two cars should be dropped off and at the second terminal one car. Then, the train will continue on its journey and deliver the final car at Location 2.

To allow for an easy drop-off, the cars are sorted in the same order as the destinations (seen from the back of the train).

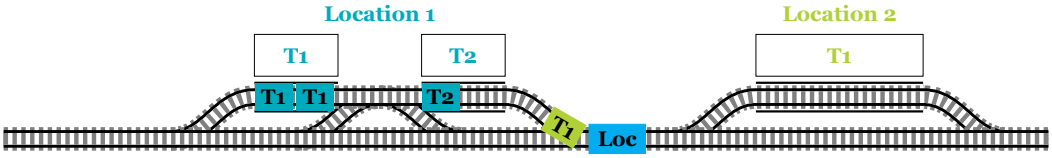


Railway State 2.19: Start scenario for Example 2.3.

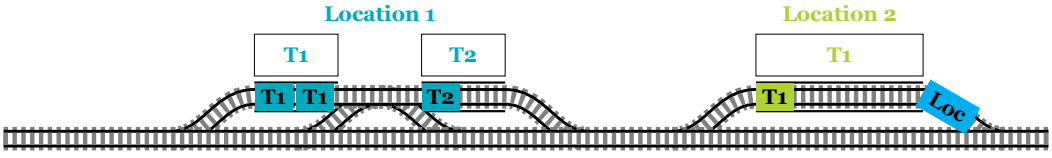
When the train arrives at the first terminal of Location 1, it drives as much forward so that the cars for that terminal are positioned correctly[2] (see Railway State 2.20). Then, these are decoupled and the train continues and delivers the car for the second terminal (see Railway State 2.21). Last, but not least, the final car is delivered at Location 2 (see Railway State 2.22).



Railway State 2.20: The trains arrives at Terminal 1 of the first destination. The last cars will be placed such that they can be decoupled and the train can continue in one go.



Railway State 2.21: The car for the second Terminal has now been delivered as well and the train continues to the final destination.



Railway State 2.22: The train has delivered all cars without any shunting operations.

◊

Note that the restrictions regarding the order of the cars are different for each train. While the fine-grained sorting is necessary for all the trains leaving for the Port of Rotterdam, trains departing for Germany (especially Keulen-Gremberg and Emmerich) will follow the first-in-first-out principle for their cars, there is no need to sort those cars further than by outbound train and once the train is full, cars that are left at Kijfhoek will be booked on one of the next trains.

[2]There are no restrictions on the order of the cars within a destination group.

To make it even more complicated, the order of the destination groups is not always the same. As an example, the daily train to the *Maasvlakte* (the 61118 for insiders), will have a different order on Wednesdays compared to the rest of the week. This has to do with the arrival times the client has provided, they differ on Wednesdays and hence the order of the destination groups is switched on that day.

2.1.3. Parking Cars

Some clients ask Deutsche Bahn to temporarily store a set of cars without a specified date on which they need them. Each day, they will send a list of specific cars or the number of cars they want in their next delivery. Which cars is not known before.

There are different ways this information can be communicated from the client to DB Cargo. Some will send an email with the list of cars attached, others use the service desk of the operations centrum, and (for an added fee) it is also possible to use the 'priority'-list on which customers can specify which cars they will need. This last list will be updated throughout the day between 8:00 and 16:00, it is the responsibility of the operations division to make sure these cars are indeed sent to the client on time.

The implications of this 'parking service' are that it is not known to the planners which cars will remain at Kijfhoek and which are needed days in advance. However, the eventual destination of these cars is known. Hence, all cars that share a common destination are parked together so that by pulling the track that contains them, it is relatively easy to filter out the necessary cars. We will see this in practice in Example 2.4.

2.2. Current Planning Strategy

For the shunting operations at Kijfhoek, it is important to remark that the shunting operations at Kijfhoek are not 24/7, but rather 22/6. Each day, the hump is closed for maintenance between 13h30 and 15h30, and on the weekend, between Saturday 15h and Sunday 15h, the hump is closed as well.

During the closures, other operations like transfers, departures, and arrivals can proceed as usual, as long as the hump is not used. The day each week on which there are no reclassifications is an ideal moment to check if all the cars that should be on given tracks are indeed there and to evaluate the planning for the next week. This is also what we will do in this thesis as we will use a seven-day planning horizon starting each time on Sunday.

Currently, the sorting decisions are made manually and can be seen as a two-step process. First, quarterly plannings are made. These specify per 15 minutes which trains are scheduled for each classification and departure track. To allow for reclassification, some tracks are reserved for *reclassification parts*. For each of these parts, it is specified which cars should be rolled into that track if their departure train is not yet ready.

Definition 2.2 (Reclassification Part)**.** A **Reclassification Part** is a set of cars that are placed on the same classification track to be reclassified later on. For each reclassification part, it is specified at which time (step), the cars are pulled to be rolled-in again.

Each day, every one of those reclassification parts is pulled once and the cars needed for a train that day (next 24h) are sent to their destination track and the cars that are not yet needed in the next 24 hours will be put back on the classification track they came from. The cars are not put on a random classification track but are already 'presorted' by destination group.

As a short example, in the situation with only two destination groups per train and two classification tracks reserved for reclassification parts, the cars that should be in the first destination group (seen from the locomotive) will be put on one track and the ones for the second group will be put on the other track. This way, if the track with all 'first-groups' is pulled, all 'first-group-cars' are put first on the track on which the train is formed before the other cars are rolled in (for a larger example, we refer you to Example 2.4).

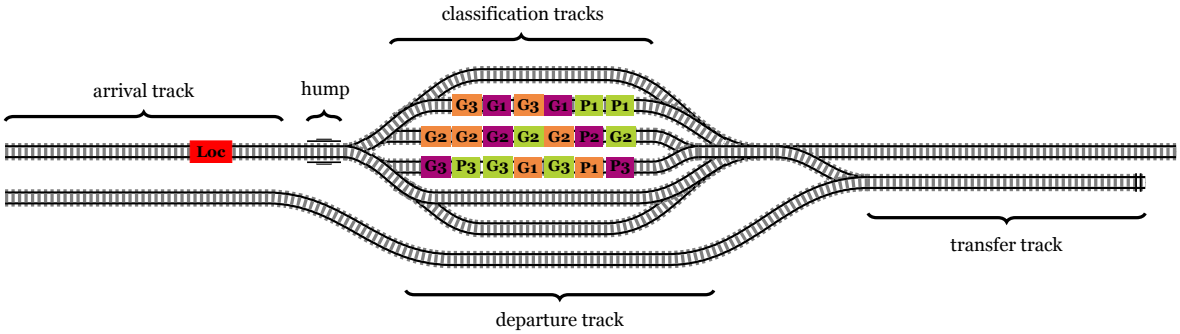
Due to the fast-changing environment of rail cargo, it is only known which cars will come in on any given train after the train has left the station and is en route to Kijfhoek. Because of this, it is not possible to make a detailed planning for days in advance. Instead, a general planning made for three months will indicate which tracks should be used for the reclassification parts and on which tracks the parts should be formed for each day of the week. However, on the day itself, the process coordinator (Proco) will call the shots on which cars have to be sent to which tracks. There are many reasons why there might be deviations from the quarterly planning. Tracks might be full or closed and trains delayed or canceled.

Example 2.4 (Reclassification with parked cars)**.** In this example, we will look into the way the cars are currently sorted with the additional complexity that parked cars offer. In Railway State 2.24, we present the layout for this example, which closely resembles Kijfhoek. There is one arrival track on which the shunting locomotive is located. After the hill, there are six classification tracks. We assume that trains can depart in the southern direction from all of those. Furthermore, we find one transfer track (similar to Track 53 at Kijfhoek). Trains that need to exit the yard at the northern side, will go there via this track.

In Railway State 2.23, we have the three trains that we need to form this time around. Each train has its own color and consists of three destination groups. Note that, there are no cars for the first destination group in the green train.

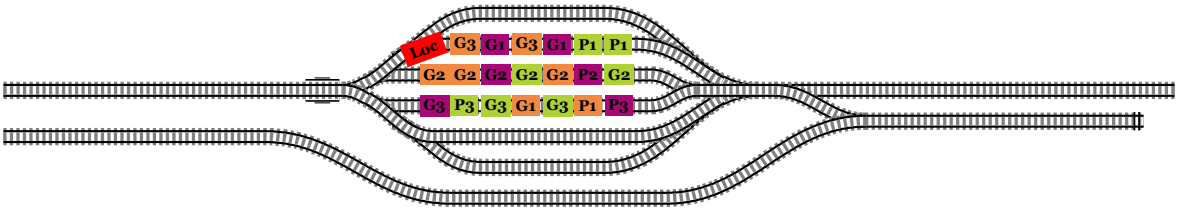
Cars that are not requested by the client and hence must be left at Kijfhoek will remain sorted by destination group. These cars will have a 'P' (of Park) instead of a 'G' (of Goal) in our example. The number does indicate the destination group.

Railway State 2.23: Goal trains for Example 2.3.

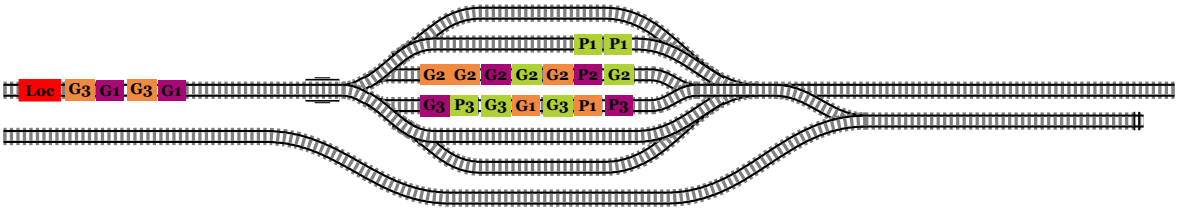


Railway State 2.24: Start scenario for Example 2.3.

The first step of the reclassification is to pull the first train, as seen in Railway States 2.25 and 2.26. On this track, we have stored the cars which should end up at the southern side of all trains (Group 1 for the purple and Group 3 for the orange train). The cars that should remain parked and have no cars behind them (seen from the shunting locomotive) are not pulled, in this case, the two green cars.

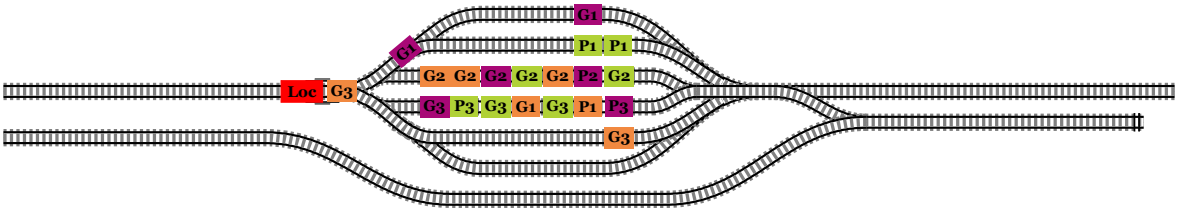


Railway State 2.25: The shunting locomotive moves in to pick up the cars from the second classification track for reclassification.

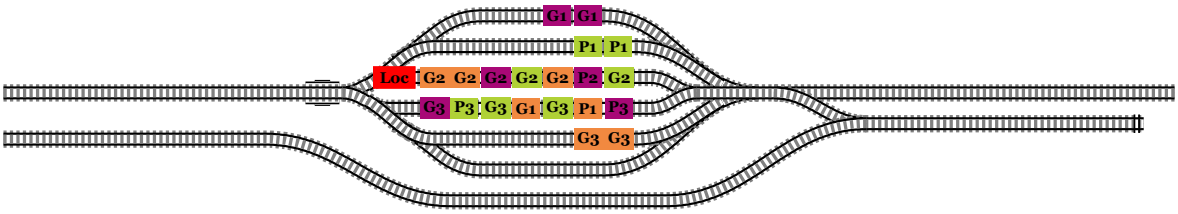


Railway State 2.26: The cars are prepared for reclassification at the arrival track.

For each outbound train, we have a classification track on which we will form it. The first (as seen from the top) one will be used for the purple train, the fifth for the orange, and the sixth for the green train. As depicted in Railway States 2.27 and 2.28, we sort the cars by train.

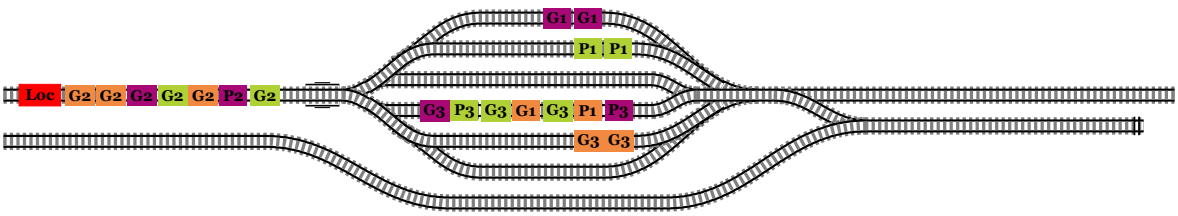


Railway State 2.27: The first cars are reclassified. The purple cars will go to the first track and the orange ones to the fifth classification track.

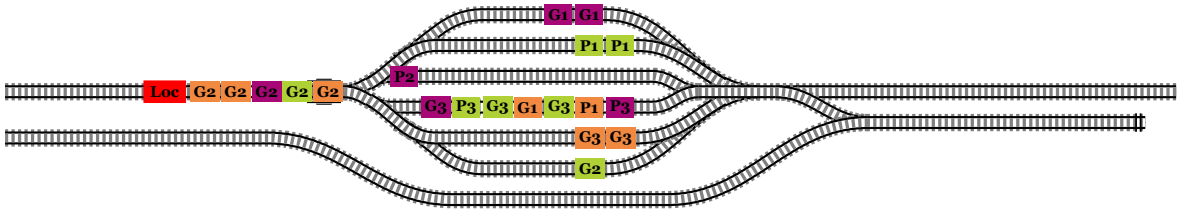


Railway State 2.28: The cars from the third classification track are pulled for reclassification.

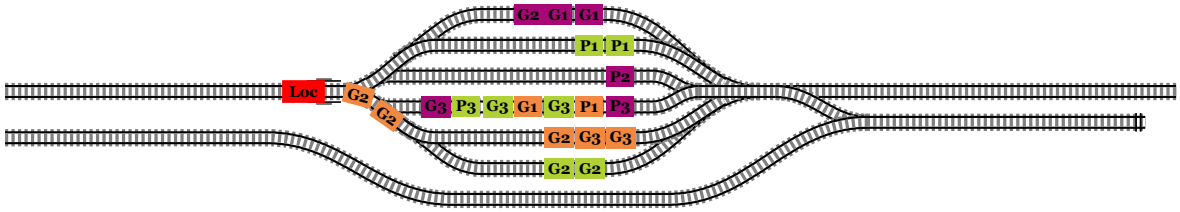
Next, we will pull the cars that are part of the middle destination group (see Railway State 2.29). The second car from the right is not needed this time and is put back on the classification track it came from (see Railway State 2.30) and the other cars are sorted by train (see Railway State 2.31).



Railway State 2.29: The cars are prepared for reclassification at an arrival track.

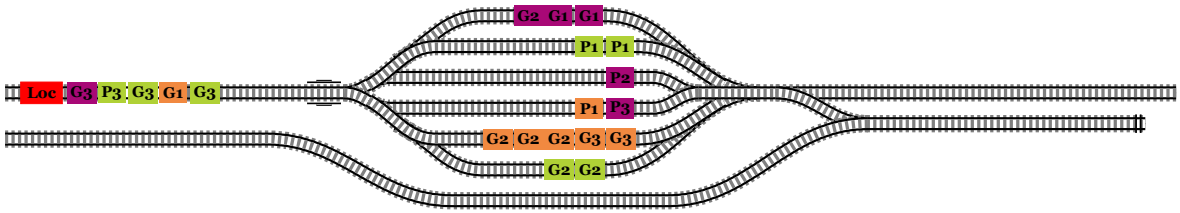


Railway State 2.30: The first cars are reclassified. The green cars will go the sixth track, the purple car that is not needed will go back to the third track.

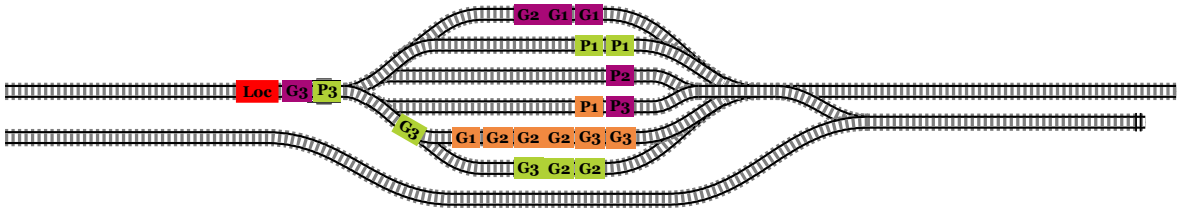


Railway State 2.31: The final cars are being reclassified. The shunting locomotive will now go on to pull the cars from the fourth classification track.

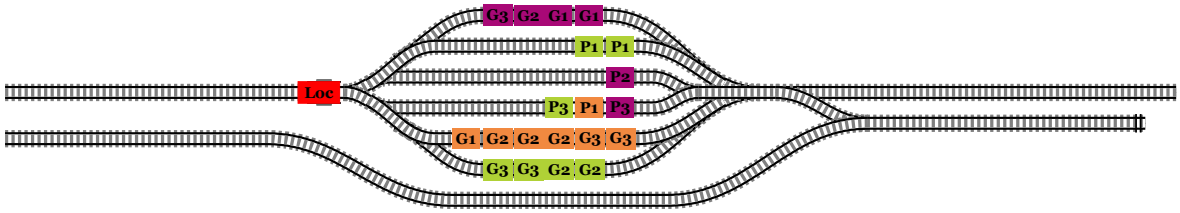
Subsequently, we pull the last cars from the last classification track (see Railway State 2.32) and sort them (see Railway States 2.33 and 2.34).



Railway State 2.32: The cars from the fourth classification track are pulled and prepared for reclassification.

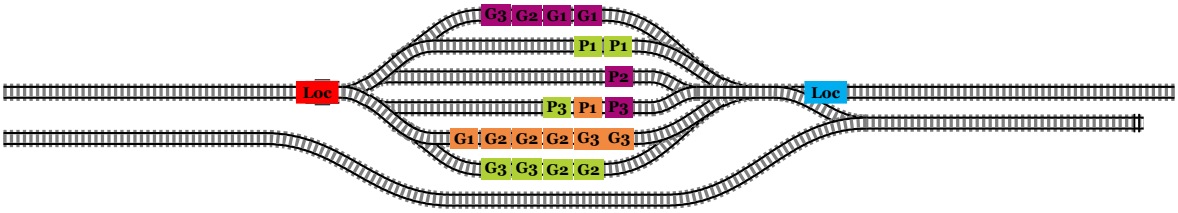


Railway State 2.33: The cars are reclassified. The cars that are not needed will be placed back on the fourth classification track.

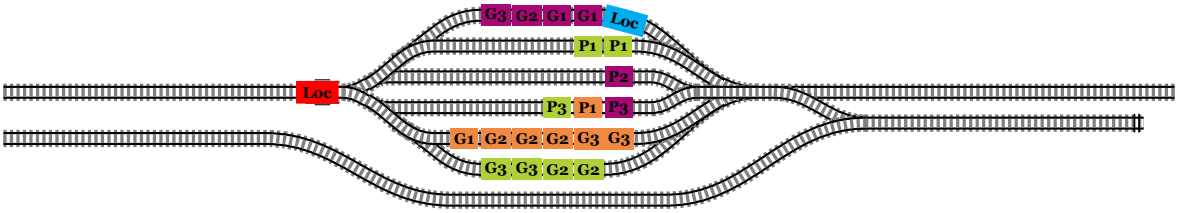


Railway State 2.34: The reclassification process is complete. The goal trains are now nicely placed on the first, fifth, and sixth track.

Now that all cars are sorted by train and the cars that should remain parked are filtered out, we can start bringing in the locomotives. As seen in Railway States 2.35 and 2.36, the purple train is the first to leave.

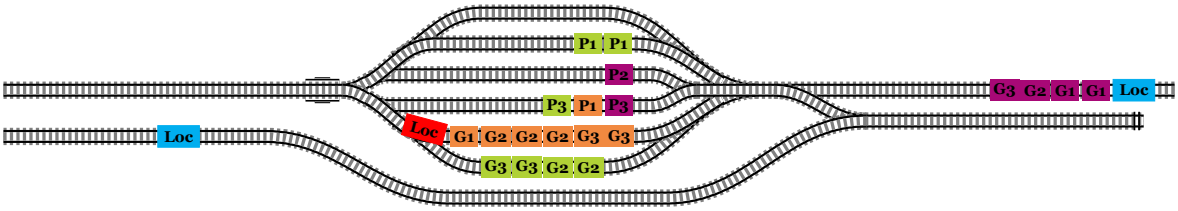


Railway State 2.35: The first ordinary locomotive moves in to pick up the purple train.

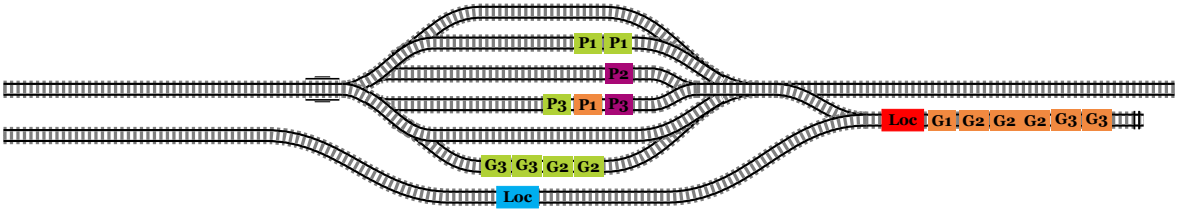


Railway State 2.36: The first ordinary locomotive picks up the cars of the purple train.

While the purple train could depart quite straightforwardly, it is more complicated for the orange train as it needs to be transferred. To that end, we bring in the shunting locomotive to push the cars from the classification track to the transfer track (see Railway States 2.37 and 2.38).

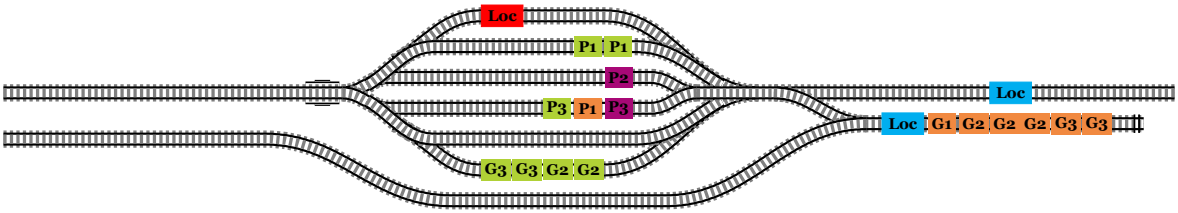


Railway State 2.37: The shunting locomotive moves in to transfer the orange cars.

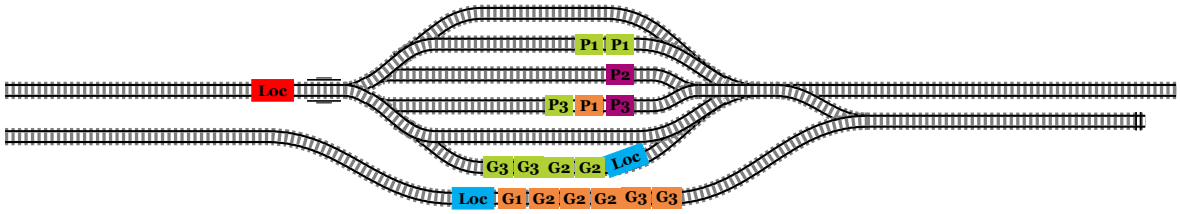


Railway State 2.38: The orange cars are pushed onto the transfer track by the shunting locomotive.

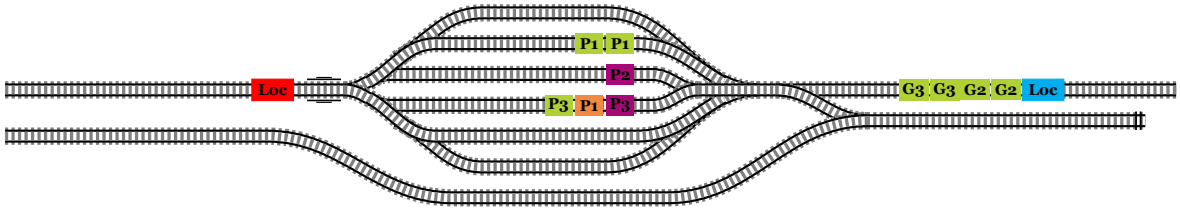
At the same time, the locomotives for the orange and green train arrive and once the shunting locomotive got out of the way, both moved in and picked up their cars (see Railway States 2.39, 2.40, and 2.41).



Railway State 2.39: The shunting locomotive has moved out of the way and an ordinary locomotive will pick up the cars from the transfer track, ready to depart.

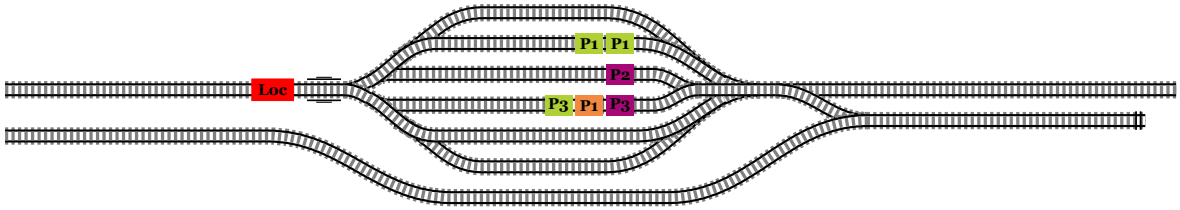


Railway State 2.40: While the orange train departs, the final ordinary locomotive moves in to pick up the green cars.



Railway State 2.41: The final train has left the classification tracks.

After all trains have left (see Railway State 2.42), the cars that were not requested by the clients are still sorted by destination group and the trains were able to be formed with a single pull for each classification track, three in total.



Railway State 2.42: All cars that needed to be part of an outbound train have departed in the correct train and order. The cars that were not yet needed are left at the classification tracks.

◊

2.3. Sorting as a public service

Kijfhoek is built in the late 1970s and was part of the Dutch Railways (NS) who owned and operated on the yard for the first 20 - 30 years. Then, as a part of the privatization of the railway operations as a whole, the company was split. Later the NS split from ProRail. The latter would go on to own and manage Kijfhoek. Later, NS sold their cargo operations to Deutsche Bahn, the German equivalent who saw an opportunity to enter the Dutch Rail Cargo market. This way, they could start operating trains from the Port of Rotterdam to the Ruhr area and beyond.

To overlook the privatization and prevent monopolies, the Dutch Authority for Consumers and Markets (ACM) was asked to check if the way the market operated was fair and square. First, they focussed on how the capacity on the main railway tracks was divided. Did all operators have the same chances and were they treated equally? Once this was all on track, they shifted their focus onto the more 'invisible' monopolies, this is when Kijfhoek came on their radar.

While any rail cargo operator can request capacity at Kijfhoek, some hurdles are deemed too high. As a part of the operations at Kijfhoek, an operator would need a special locomotive that can be linked to the traffic management systems of ProRail to roll cars over the hill. These locomotives are expensive and the ACM ruled them too expensive for any company to be able to reasonably enter the market. For more on the rules and regulations, see [15].

To break this monopoly, DB Cargo is asked to publicly offer a sorting service to third parties. As it stands now[3], DB Cargo won't reclassify cars for third parties and only provide the initial roll-in. Nonetheless, this service will impact the operations of DB Cargo itself. When they perform a roll-in for a third party, they can't use the hump, the locomotive, and crew during the roll-in. Furthermore, the third parties will require space at the arrival, classification, and departure tracks. There will be no mixing of cars from DB Cargo and third parties for the time being and each company will have its 'own' tracks. This will reduce the capacity available for DB Cargo.

[3]Remark that at the time of writing this thesis, the exact specifications are yet unknown.

3
Literature Review

The subject of hump yard sorting has been the subject of several papers over the past decades. In this chapter, we will go over some of the most important and take away key lessons.

First, in Section 3.1, we will present some related work in the area of shunting problems. Then, we will go over some of the more well-known shunting algorithms. First, we will look into single-stage sorting in Section 3.2. Then, we will consider multi-stage sorting algorithms in Section 3.3. Following, we will look at research done in the area of robustness in Section 3.4 and lastly, we present an overview of the contributions of this thesis in Section 3.5.

First of all, we will have to make the distinction between **single-stage** and **multi-stage** sorting. In short, with single-stage, reclassifications are not (or very limited) allowed, while with multi-stage sorting, there is no limit on reclassifying.

Remark that if we use an algorithm (or variation thereon) for our use-case, we will present a further, more detailed explanation in the respective chapters.

3.1. Related Work

In [7], Bohlin *et. al.* try to minimize the number of reclassifications. They introduce a Column Generation (CG) and Arc-Based (AB) approach and compare these to one other IP (found in [5]) and a heuristic (found in [4]). For their experiments, they use instances based on historical data for the Hallsberg shunting yard in Sweden. This yard is comparable in size with Kijfhoek albeit a bit smaller. They find that their methods produce optimal solutions for their instances, up to five days ahead. However, in their approach, they use fixed times and orders in which the departing trains should leave the yard, limiting critical flexibility on that end.

Belosevic and Ivic look into a variable neighborhood search approach for hump yard scheduling in [2]. They show that the heuristic can find optimal solutions, however, the instances are rather small (at most 500 cars) and in their formulation, they leave out the arrival and departure times of inbound | outbound trains. Another heuristic is proposed by Hauser and Maue in [21]. Like the previous, they use limited-size instances without taking arrival | departure times into account.

There have been some literature reviews on the topic of rail shunting operations. Gatto *et. al.* ([16]) present an overview of frequently used algorithms (most of which we will cover later in this chapter) for general shunting problems as well as a review of more specific use cases and algorithms. Boysen *et. al.* present the research done in the 40 years leading up to their paper in [8] and use the classification system first introduced by Hansmann and Zimmermann ([20]) to outline the literature on the subject.

3.2. Single-stage sorting

As we mentioned before, in single-stage sorting, cars only move 'forward' on the yard. Hence they go from an arrival track to a classification track and end at a departure track without any steps back (unlike multi-stage). In practice, almost all shunting yards are forced to apply some way of multi-stage sorting as the number of required tracks to apply single-stage sorting is usually way higher than the number of tracks available. In their paper, Dahlhaus *et. al.* ([13]) find an upper bound on the required number of tracks. Consider the following problem which describes the single-stage shunting problem in short.

Train Marshalling Problem

Given a partition $\mathcal{S}$ of $\{1, \dots, n\}$ into disjoint sets $S_1, \dots, S_t$, find the smallest number $k = K(\mathcal{S})$ so that there exists a permutation $p(1), \dots, p(n)$ of $\{1, \dots, t\}$ with the property: The sequence of numbers $1, 2, \dots, n, 1, 2, \dots, n, \dots, 1, 2, \dots, n$ where the sequence $1, 2 \dots, n$ is repeated k times contains all elements of $S_{p(1)}$, then all elements of $S_{p(2)}$, and so on until $S_{p(n)}$. Remark that k is the number of classification needed.[13, p. 1]

Example 3.1 (Train Marshalling Problem)**.** Consider the sets

$$S_1 = \{1, 6, 11\},\ S_2 = \{2, 7\},\ S_3 = \{3, 8\},\ S_4 = \{4, 9\},\ S_5 = \{5, 10\}.$$

In this case, we find that $K(\mathcal{S}) = 4$ since if we have four copies of the sequence 1 2 ... 11, we get

1 2 3 4 5 **6** 7 8 9 10 **11** 1 **2** 3 4 5 6 **7** **8** 9 10 11 1 2 **3** **4** 5 6 7 8 **9** **10** 11 1 2 3 4 **5** 6 7 8 9 10 11

So, if we now add cars 1, 6, and 11 on the first classification track, 2, 7, and 8 on the second, 3, 4, 9, and 10 on the third, and 5 on the fourth classification track and we combine these tracks by first placing the first, then the second, third, and fourth on a departure track, we get the sequence

1 6 11 | 2 7 | 8 3 | 4 9 | 10 5.

We find that the identity permutation is in this case the optimal permutation. [13, p. 2] ◊

They show that the decision problem whether or not a solution with $K(\mathcal{S}) \leq k$ exists is $\mathcal{NP}$-complete and that an upper bound on K_n (number of required classification tracks for any instance with n cars) is given by $\left\lceil \frac{n}{4} + \frac{1}{2} \right\rceil$. Remark that this upper bound only holds if the sets S_i can be permuted (i.e. one destination group per train).

A more hybrid version of single and multi-stage sorting is extended single-stage sorting. With the extended version, there are - *although very limited* - some reclassifications. In his papers ([26–28]), Kraft looked into extended single-stage sorting where three reclassifications occurred each day, each spaced 8 hours apart. For more on applying single-stage sorting, we refer to those papers.

3.3. Multi-stage sorting

There are many multi-stage sorting algorithms and in practice, more often than not, a combination of algorithms is used. Here, we will go over some of the most-used sorting algorithms.

3.3.1. Basic sorting

One of the most basic sorting strategies is **sorting-by-train**. Here, all cars for each outbound train are first placed on one track. Then, for each train, the cars are pulled and sorted by destination group, with one destination group per track. When this reclassification is completed, the cars are combined again to make one sorted outbound train.

Take d as the number of departing trains and $g_{\max}$ as the maximum number of groups in a single train. Then, for the sorting-by-train strategy, we need $\mathcal{O}(d + g_{\max})$ classification tracks. Note that this number could be reduced depending on arrival and departure times.

The second basic sorting strategy is the **sorting-by-block** strategy. This is also the strategy that forms the basis for the current sorting strategy at Kijfhoek (as seen in Section 2.2). With the strategy, all cars that are in the first destination group of any outbound train (as seen from the locomotive) are placed on the first available classification track. All cars in the second group are placed on the second track, and so on until all cars are placed on a classification track. When all cars are initially classified, the first reclassification part is pulled and the 'first-group cars' are each placed on a classification track based on the outbound train. This is continued until all cars have been reclassified.

Just like as with the sorting-by-train strategy, this will require $\mathcal{O}(d + g_{\max})$ tracks. Again, this can be reduced depending on arrival and departure times. Remark that with this strategy there are fewer reclassifications (assuming that $g_{\max} \leq d$). However, if the lead time (difference between arrival and departure) of cars is low, more reclassifications might be necessary if there is no option to wait until all trains can be formed. [12]

3.3.2. Triangular Sorting

Triangular sorting is a more sophisticated way of sorting cars when compared to the previously discussed sorting-by-train and sorting-by-block methods. With triangular sorting, the destination groups in each outbound train are numbered first. To start off, we sort the outbound train by number of destination groups. The first train will have the most and the last train the least destination groups. Assume that train o has g_o destination groups. Then, the destination groups in train o are numbered as follows. The first is numbered $\frac{o(o-1)}{2} + 1$, the second $\frac{o(o-1)}{2} + 2$, and so on until $\frac{o(o-1)}{2} + g_o$.

At the initial roll-in, we will sort the cars as follows. At a track t, we will have all cars which are in a destination group with number $\frac{t(t-1)}{2}$ and all cars with numbers $\frac{t(t-1)}{2} + jt + 1 + \frac{(j-1)(j-2)}{2}$ for all $j \geq 2$.

The next phase is the reclassification phase. Here, the tracks are pulled one by one. On the first classification track, we will form the first train, on the second, the second train, and so on. At the first step, the cars from the first track will be pulled, the cars for the first group of the first train will be put back on the first track. Furthermore, all cars of the next (which is numbered the third) destination group will be put on the second track. The cars of the following group (which is numbered the fifth) are placed on the third track. This process continues until all cars are reclassified. Now, the second track is pulled and all cars that can 'safely' go to the track on which their train is formed, go there and the others are divided over the subsequent tracks.

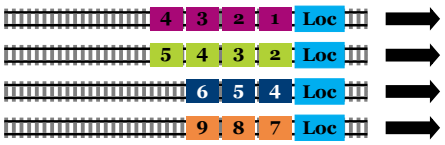
It has been shown that for this method, an upper bound on $g_{\max}$ is given by $\frac{h(h+1)}{2}$ where h is the number of reclassifications[23]. This also tells us that the number of sorting steps for a given instance is given by

$$h = \left\lceil \sqrt{2g_{\max}} - \frac{1}{2} \right\rceil.$$

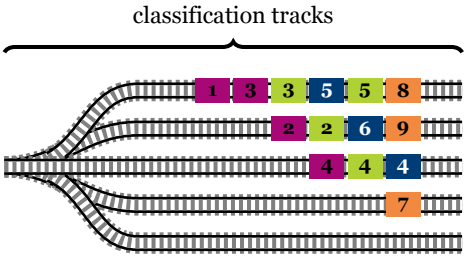
Using this, we find that the number of tracks required is between $\left\lceil \sqrt{2g_{\max}} - \frac{1}{2} \right\rceil$ and $\left\lceil \sqrt{2g_{\max}} - \frac{1}{2} \right\rceil + d - 1$. [8, 12, 23, 29]

Example 3.2 (Triangular Sorting)**.** In this example, we will show how triangular sorting can be used to sort cars at a shunting yard. In Railway State 3.1, we present the four trains we would like to get. We have numbered the destination groups according to the algorithm. In Railway State 3.2, we show how the cars are sorted at the initial roll-in. Remark that the cars can be at any order at the classification tracks and are just sorted per track.

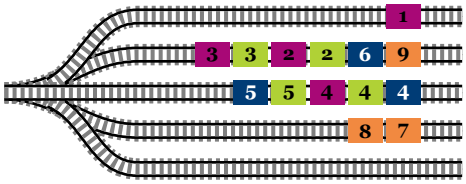
In the next three Railway State (3.3 through 3.5), we perform three reclassifications and find that after these reclassifications, the cars are all sorted as they should be according to Railway State 3.1.



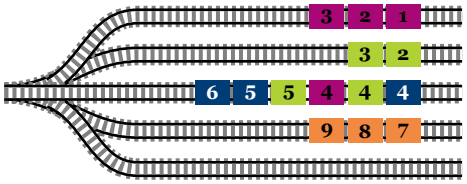
Railway State 3.1: Goal outbound trains and incoming train for Example 3.2.



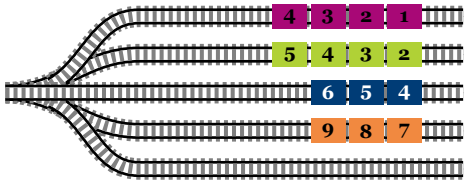
Railway State 3.2: Location of the cars after the initial roll-in of Example 3.2.



Railway State 3.3: Location of the cars after the first reclassification of Example 3.2.



Railway State 3.4: Location of the cars after the second reclassification of Example 3.2.



Railway State 3.5: Location of the cars after the final reclassification of Example 3.2.

◊

3.3.3. Geometric Sorting

Geometric sorting has some similarities when compared to the Triangular sorting method. Just like with the previous, we sort the trains by number of destination groups. In this case, for train o, we number the destination groups $2(o-1)+1$, $2(o-1)+2$, and so on. At the initial roll-in, we send to track t all cars which have been numbered $2^{t-1}(2j-1)$ for $j = 1, 2, \ldots$.

In the reclassification phase, we will pull the tracks one by one. When we pull track t, we will send all cars with a destination group j number less than or equal to $2t-1$ to their final track, the others, we send to the track where the cars with destination group $j-1$ are, except if a car is the first of an outbound train, in that case, it will go to an empty track.

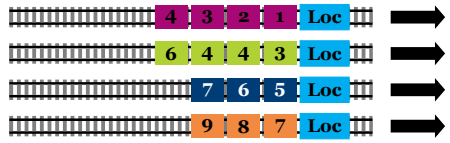
For Geometric sorting, it is proven that the maximum number of destination groups is bounded by $2^h - 1$[23]. This way, we find that for any instance, we need a minimum of

$$\lceil \lg(g_{\max} + 1) \rceil$$

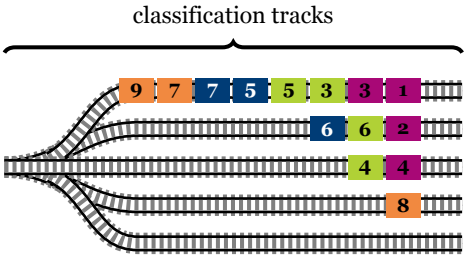
sorting steps. So, the number of classification tracks required is between $\lg(g_{\max}+1)$ and $\lg(g_{\max}+1) + d - 1$. [8, 12, 23, 29]

Example 3.3 (Geometric Sorting)**.** To illustrate the workings of the geometric sorting algorithm, we will use the same goal trains as in the previous example, however as seen in Railway State 3.6, we have updated the numbering of the destination groups.

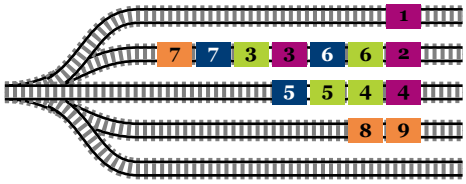
In Railway States 3.7, we present how the cars are located on the tracks after the initial roll-in and in Railway State 3.8 through 3.11, we show how the reclassifications must be performed and how the cars are sorted in the end.



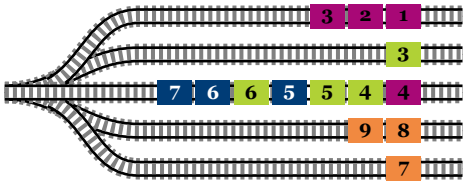
Railway State 3.6: Goal outbound trains and incoming train for Example 3.3.



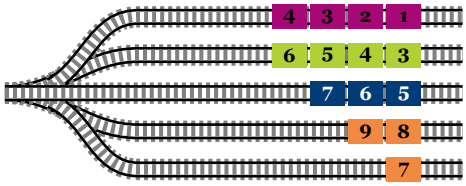
Railway State 3.7: Location of the cars after the initial roll-in of Example 3.3.



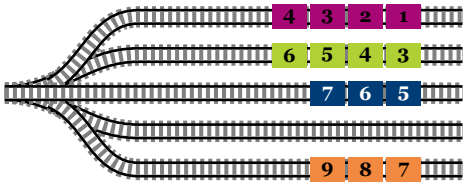
Railway State 3.8: Location of the cars after the first reclassification of Example 3.3.



Railway State 3.9: Location of the cars after the second reclassification of Example 3.3.



Railway State 3.10: Location of the cars after the third reclassification of Example 3.3.



Railway State 3.11: Location of the cars after the final reclassification of Example 3.3.

◊

3.4. Robustness

Planning algorithms can only be used in practice if they manage to respond and adapt to unforeseen changes in the instance. In the fast-changing world of rail freight logistics, many possible disturbances can make a solution infeasible. Delays, Early Arrivals, Extra Cars, and Changing Orders to name a few.

We acknowledge that no solution can stay feasible for all different scenarios. To that end, we do not focus on creating strictly robust solutions. Instead, we focus on creating recoverable robust solutions. These kinds of solutions come with an algorithm - the recovery algorithm - which prescribes how a solution should be changed to adapt to a change in the instance.

There has been very little research done on this subject and - to the best of our knowledge - only two papers actually present algorithms to create recoverable robust solutions. In both cases was the yard different from Kijfhoek in that there was only a single arrival track. This means that there is little to no freedom to change the arrival order of the inbound trains. Next to this do both consider a fixed outbound sequence. Again, this means that there is no freedom in deciding the order of the outbound trains and the order of cars within the same destination group - as both papers use destination groups of only a single car.

In [11], Cicerone *et. al.* have studied recoverable robustness for four different scenarios. They consider the cases where there is a new car introduced, a car is removed, a car is misplaced, and a track is made unavailable. These changes are all very small with respect to the size of practical instances. In their paper, they do give some bounds on the performance ratios of recoverable robust solutions without any experiments in real-life scenarios.

Büsing and Maue look into recoverable robustness algorithms in [9]. They present an algorithm to create recoverable-robust solutions. In their approach, they look for solutions that can be recovered by adding additional sorting steps. For this approach, they assume that there are no restrictions on the number of classification tracks used, something that is most certainly not the case in real-life. The algorithm to create such solutions is $\mathcal{NP}$-complete, however, practical experiments show a polynomial running time for most small (compared to Kijfhoek) scenarios.

3.5. Contribution of this thesis

This thesis will look into the shunting problem as a whole, instead of focusing only on parts of the complete shunting process. Most research in the area of shunting algorithms is concentrated around the operations at the classification tracks. We will, however, focus on the shunting yard as a whole and look at how we can use the arrival and departure tracks to create extra space at the classification tracks by having trains wait at the arrival tracks or transferred early to the departure tracks.

Furthermore, we will look into how shunting algorithms perform in practice with incomplete information. To the best of our knowledge, there has been no research done in this area. In practice, not all information is known at the beginning of the planning horizon (e.g. which cars do arrive). We will propose a solution approach that can create general solutions that can then later be adapted based on the latest information.

Lastly, we will see how well we can recover solutions after perturbations. As mentioned before, there has been very little research done in this area. We will look into how the same solution approaches that created the initial solutions can be used to recover solutions after perturbations to the instances.

4
Integer Programming

The shunting problem at Kijfhoek can be formulated as an Integer Program. In this chapter, we will present two formulations. The Direct Formulation (DF) (Section 4.1) and a Flexible Arrival Formulation (FAF) (Section 4.2). Next to these two formulations, we will show how we can use a preprocessing heuristic in combination with the DF in Section 4.3.

The shunting problem at hump yards has been formulated as an IP by many different papers ([2, 4, 6, 29]). We have chosen to loosely follow the approach taken by Márton *et. al.* ([29]).

In the aforementioned paper, the authors consider the Laussane-Triage hump yard. The yard has a total of 11 arrival tracks and 38 classification tracks, this is very similar to Kijfhoek considering that Kijfhoek has 14 arrival, 43 classification, and 8 departure tracks.

While we have used the formulation presented in the paper by Márton *et. al.* as the framework for the DF we will use, there are some key differences. In contrast to the Laussane case, we (currently) do not use simultaneous classifications - meaning using both tracks on the hump at the same time to classify two trains at the same moment. Furthermore, we extend the formulation to use the arrival and departure tracks as buffer locations where trains can be temporarily placed.

4.1. The Direct Formulation

The solution we get from this formulation won't be assigning cars to specific tracks. Instead, it will ensure that at each time step, the number of tracks used at the time stays within the limits of the infrastructure. After we get the solution from the formulation, we will use a greedy planner to assign trains and cars to tracks.

In total, we will allow at most h reclassifications, where h is part of the input. For each car, the result will be a bitstring which defines in which of the h allowed pulls, the car is part of the reclassification part (i.e. set of cars that is classified at a given moment). If there are no 'ones' left (in the bitstring), this means that the car will be placed on the track on which the outbound train of which it is a part is being built. We will limit the number of reclassifications per block to one.

Given the maximum number of pulls h, we can define for each car the bitstring:

$$b_j = \left(b_{jh-1} \dots b_{j2} b_{j1} b_{j0}\right).$$

An example, if for a car j the bitstring is defined as 00101, the car will be pulled in the first and third timeslot (read from right to left) and when it is rolled in the third slot, it will be placed on the track on which the outbound track of which it is part is built (see Example 4.1 for more details on this encoding).

Remark that for this formulation we will discretize the planning horizon into blocks. As discussed before, while no classification is the same, it is a reasonable assumption that three classifications can be made each hour. Therefore, we can classify up to 66 times a day (keeping in mind that the hump is only available 22 hours per day). Note however that we can not do 66 reclassifications per day as we also have to classify the incoming trains.

We will split the day into three shifts, coinciding with the shifts of the crew at Kijfhoek. There is a Morning (7h-15h), Late (15h-23h), and a Nightshift (23h-7h). All inbound and outbound trains will

be assigned to the shift that they arrive/depart in. For each shift, we will limit the number of classifications, but there is no direct restriction on the order in which the arrivals, reclassifications, and departures must take place, as long as they happen in a feasible order (e.g. cars must arrive before they are reclassified).

Programming limitations

Since we have chosen to use bitstrings to store the information and by design, these can be up to 64-bits long, we have a limitation here in the maximum number of reclassifications that can be part of any schedule created by this and the other implementations presented in this thesis, namely 63.

No implementation in this thesis will be able to - in one run - create a schedule with more than 63 reclassifications. This is a necessary trade-off between speed and flexibility.

Furthermore, under the assumption that the arrival order of trains is fixed, we know the order in which all cars are classified. If n cars arrive during the planning horizon, this gives the sequence

$$\mathrm{K} = \kappa_1 \dots \kappa_n.$$

We know that if for an outbound train the sequence is given by

$$\dots \kappa_k \kappa_l \dots$$

with $k > l$, then car κ_l needs to be pulled after car κ_k is pulled, otherwise, it is not possible that the order of the cars is reversed in the outbound train relative to the inbound sequence. We say that

$$\rho_{kl} = \begin{cases} 1 & \text{if } k > l \\ 0 & \text{otherwise} \end{cases}.$$

Now, for the set of all outgoing trains O, define the set $\mathcal{F}$ of all 'first' cars (cars in the first destination group after the locomotive) and $\mathcal{F}(o)$ the 'first' cars of train o.

In the case of the train depicted in Railway State 4.1, there are cars for three different destination groups (blue, purple, and orange). The cars for the blue destination, cars 1 and 2, are the ones closest to the locomotive and hence, for this train $\mathcal{F}(o) = \{1, 2\}$.

Railway State 4.1: Example train with six cars for three different destination groups.

Now, we can define the DF as follows. First, we will give an overview of all variables, then we will present the formulation itself (objective function and constraints), and at the end, we will take a closer look at the variables and individual constraints.

We define $\alpha(a)$ to be the arrival time of train a and $đ(o)$ to be the departure time of train o. We assume that the trains are not at Kijfhoek before (after) these times. Furthermore, let $\propto(j)$ as the arrival time of car j.

The constants A_t, N_t, and Δ_t indicate how many arrival | classification | departure tracks are available at time t.

Direct Formulation

$$b_{jt} = \begin{cases} 1 & \text{if car } j \text{ is reclassified at time step } t. \\ 0 & \text{otherwise.} \end{cases}$$

$$\alpha_{at} = \begin{cases} 1 & \text{if arriving train } a \text{ has not yet been classified at time step } t. \\ 0 & \text{otherwise.} \end{cases}$$

$$p_{ot} = \begin{cases} 1 & \text{if outbound train } o \text{ is being 'built' at a classification track at time step } t. \\ 0 & \text{otherwise.} \end{cases}$$

$$\delta_{ot} = \begin{cases} 1 & \text{if outbound train } o \text{ has left the classification tracks at time step } t. \\ 0 & \text{otherwise.} \end{cases}$$

$$\psi_{ot} = \begin{cases} 1 & \text{if for train } o \text{ not yet a single car has entered the classification tracks at time step } t. \\ 0 & \text{otherwise.} \end{cases}$$

$$\mu_{\tau t} = \begin{cases} 1 & \text{if the reclassification part scheduled for time } \tau \text{ consists of at least one car at time step } t, \text{ with } t \le \tau. \\ 0 & \text{otherwise.} \end{cases}$$

$$\min \quad \sum_{t,j} b_{jt} \tag{4.1}$$

$$\text{s.t.} \quad \sum_{t=0}^{h-1} 2^t \left(b_{lt} + \alpha_{a(l)t}\right) \ge \rho_{lk} + \sum_{t=0}^{h-1} 2^t \left(b_{kt} + \alpha_{a(k)t}\right), \quad (l,k)\text{s.t. } l,k \text{ part of same train and adjacent destination groups} \tag{4.1a}$$

$$\sum_{a:\alpha(a)\le t} \alpha_{at} \le \mathrm{A}_t \qquad t \in \{0,\dots,h-1\} \tag{4.1b}$$

$$\sum_{t\le\tau} \mu_{\tau t} + \sum_{o} p_{ot} \le N_t \qquad t \in \{0,\dots,h-1\} \tag{4.1c}$$

$$\sum_{đ(o)>t} \delta_{ot} \le \Delta_t \qquad t \in \{0,\dots,h-1\} \tag{4.1d}$$

$$\mu_{\tau t} \ge b_{j\tau} - \sum_{t\le t'\le\tau} b_{jt'} \qquad \tau \in \{0,\dots,h-1\}, t \in \{\propto(j),\dots,\tau\}, j \in J \tag{4.1e}$$

$$p_{ot} \ge 1 - \sum_{t'\le t} b_{jt'} - \psi_{ot} - \delta_{ot} \qquad o \in O, t \in \{0,\dots,h-1\}, j \in \mathcal{F}(o) \tag{4.1f}$$

$$\alpha_{at} \ge \psi_{ot} \qquad a\text{s.t. train } o \text{ has at least one car from } a \tag{4.1g}$$

$$\alpha_{at} \ge 1 \qquad a \in A, t < \alpha(a) \tag{4.1h}$$

$$\delta_{ot} \ge 1 \qquad o \in O, t > đ(o) \tag{4.1i}$$

$$\sum_{t'\le t} b_{jt'} \le M(1-\alpha_{at}) \qquad t \in \{0,\dots,h-1\}, j \in J \tag{4.1j}$$

$$\sum_{t'\ge t} b_{jt'} \le M(1-\delta_{ot}) \qquad t \in \{0,\dots,h-1\}, j \in J \tag{4.1k}$$

$$\alpha_{at} \ge \alpha_{at+1} \qquad a \in A, t \in \{0,\dots,h-2\} \tag{4.1l}$$

$$\alpha_{at} \ge \alpha_{a+1t} \qquad a \in A\backslash\{\text{last train}\}, t \in \{0,\dots,h-1\} \tag{4.1m}$$

$$\psi_{ot} \ge \psi_{ot+1} \qquad o \in O, t \in \{0,\dots,h-2\} \tag{4.1n}$$

$$\delta_{ot+1} \ge \delta_{ot} \qquad o \in O, t \in \{0,\dots,h-2\} \tag{4.1o}$$

$$\psi_{ot} + p_{ot} + \delta_{ot} \le 1 \qquad o \in O, t \in \{0,\dots,h-1\} \tag{4.1p}$$

Variables The variable b_{jt} indicate if car j is reclassified at time step t. The next variable, α_{at}, is related to the arriving trains. The variables indicate if the arriving train has been classified yet. Notice that the variable does not indicate if a train has yet arrived at the arrival tracks or is still en route to Kijfhoek. Only when the train is classified, the variable will change value from one to zero.

For each outbound train o, we have three variables, ψ_{ot}, p_{ot}, and δ_{ot}. The first is the minimum of all α_{at} where train a has at least one car for train o. This means that when ψ_{ot} is changed from one to zero, there is at least one car for train o on the classification tracks.

The next variable, p_{ot}, indicates if we have started 'building' train o yet. If this variable is one, this means that we have a single classification track dedicated to train o and that the cars on that track must adhere to the order constraints.

The last variable, δ_{ot}, indicates if the train has left the classification tracks. There are two options here, the train can be transferred to a departure track or it has departed already (note the resemblance with the α_{at} variables).

To keep track of the number of classification tracks used, we use the $\mu_{\tau t}$ variables which indicate whether or not there is a classification track in use for the reclassification part. There are no further restrictions on the train or order of these cars as they will be reclassified at τ.

Objective function Given the constraints, we will look for a feasible solution minimizing the total number of *carrolls*. Each time a car rolls over the hump as part of a reclassification part, we count this as a *carroll*. Note that we do not count the times a car rolls over the hump as part of an initial classification as this number is a constant - each car goes at least one time over the hump.

While we are also interested in finding solutions that use a minimal number of reclassifications and reclassification tracks. We handle this as an input parameter and limit the number of reclassification parts per day.

Constraints The first constraint, Constraint 4.1a, is there to enforce that the order of the cars in the outbound trains is correct.

$$\sum_{t=0}^{h-1} 2^t \left(b_{lt} + \alpha_{a(l)t}\right) \geq \rho_{lk} + \sum_{t=0}^{h-1} 2^t \left(b_{kt} + \alpha_{a(k)t}\right), (l,k)\text{s.t. } l,k \text{ part of same train and adjacent destination groups} \tag{4.1a}$$

As mentioned before, all cars in an outbound train are assigned a destination group. Cars within the same destination group can have any relative order, as long as the destination groups are sorted properly.

The constant ρ_{lk} indicates if the cars are in the wrong order. Hence, if ρ_{lk} is one, we need to reclassify car l at least once so that at the track on which we form the outbound train. Car l is positioned behind car k. There are two options, we can either reclassify car l after the latest time step at which car k has been (re)classified for the first time, or we reclassify car l after the arrival of car k so that they end up at the same reclassification track and then, we reclassify both one (or more) times together.

Note that we can only 'start' the process of reversing the order after both cars have arrived. If we reclassify car l before the arrival of car k, this will not change anything to the relative order of both cars. Only when both cars have arrived, we have that $\alpha_{a(l)t} = \alpha_{a(k)t} = 0$. Let $\mathrm{t} = \min\{t | \alpha_{a(l)t} = \alpha_{a(k)t} = 0\}$, then, Constraint 4.1$a$ is the same as

$$\sum_{t=\mathrm{t}}^{h-1} 2^t b_{lt} \geq \rho_{lk} + \sum_{t=\mathrm{t}}^{h-1} 2^t b_{kt}, (l,k)\text{s.t. } l,k \text{ part of same train and adjacent destination groups}.$$

Remark that this corresponds to the requirements presented above. If the cars are in the correct relative order already (i.e. ρ_{lk} is zero), we make sure that this stays this way.

To present a clearer picture of the pairs (l, k) for which we add these constraints, consider the train as depicted in Railway State 4.2. Here we have three destination groups, and hence two pairs of adjacent groups (blue & purple and purple & orange). To that end, we add the constraint for the pairs

$$(1,3),(1,4),(1,5),(2,3),(2,4),(2,5),(3,6),(4,6), \text{ and } (5,6).$$

This way, we allow the cars to be in any order within the group as long as the groups are sorted correctly.

Railway State 4.2: Example train with six cars for three different destination groups.

The next constraints - 4.1b, 4.1c, and 4.1d - ensure that there are enough arrival, classification, and departure tracks. They count how many tracks are used at each step t and restrict this to the allowed number at that time.

$$\sum_{a:\alpha(a)\leq t} \alpha_{at} \leq A_t \qquad t \in \{0,\dots,h-1\} \tag{4.1b}$$

$$\sum_{t\leq\tau} \mu_{\tau t} + \sum_{o} p_{ot} \leq N_t \qquad t \in \{0,\dots,h-1\} \tag{4.1c}$$

$$\sum_{\delta(o)>t} \delta_{ot} \leq \Delta_t \qquad t \in \{0,\dots,h-1\} \tag{4.1d}$$

Given $\alpha(a)$ and $\delta(o)$ we can count the number of trains at that are at the arrival (departure) tracks before classification (departure) at each time step. To count the number of classification tracks, we both look at the time steps we need one of the reclassification parts as well as the time steps in which outbound trains are being 'build'.

To ensure that the μ variables indeed indicate whether or not we need a track for classification part τ at t, we introduce Constraint 4.1e.

$$\mu_{\tau t} \geq b_{j\tau} - \sum_{t\leq t'\leq\tau} b_{jt'} \qquad \tau \in \{0,\dots,h-1\}, t \in \{\propto(j),\dots,\tau\}, j \in J \tag{4.1e}$$

Intuitively, we are looking for the car which is placed at the classification track the longest. This means that we want to find the longest line of zeroes with a one at position τ. The starting point can either be the arrival time of a car or the last time it was reclassified. This line does indicate the longest time that a car is waiting on a classification track to be reclassified at time step τ.

For the variables p_{ot}, we do something very similar in Constraint 4.1f.

$$p_{ot} \geq 1 - \sum_{t'\leq t} b_{jt'} - \psi_{ot} - \delta_{ot} \qquad o \in O, t \in \{0,\dots,h-1\}, j \in \mathcal{F}(o) \tag{4.1f}$$

Remember that this variable indicates if we are forming an outbound train at the classification tracks. We assume that this is the case unless, there are no cars for that train at classification tracks (ψ), the train has left the arrival tracks and is now at the departure tracks or already away from the yard (δ), or all cars that are at the classification tracks are now still part of a reclassification part.

Remark that we defined the ψ_{ot} variables to be the minimum of the α_{at} variables for all a such that there is at least one car in train a that needs to get into train o. Constraint 4.1g enforces this.

$$\alpha_{at} \geq \psi_{ot} \qquad a \text{ s.t. train } o \text{ has at least one car from } a \tag{4.1g}$$

We do not allow a train be classified before the arrival time, hence we add Constraint 4.1h. Trains must depart before their departure time, to that end we add Constraint 4.1i.

$$\alpha_{at} \geq 1 \qquad a \in A, t < \alpha(a) \tag{4.1h}$$
$$\delta_{ot} \geq 1 \qquad o \in O, t > đ(o) \tag{4.1i}$$

If cars are not at the classification tracks - either because their train did not yet arrive or their train has left - we must have that the corresponding b_{jt} variables are zero. This relation, formulated using the α and δ variables, is enforced by Constraints 4.1j and 4.1k. Note that M can be any sufficiently large number.

$$\sum_{t' \leq t} b_{jt'} \leq M(1 - \alpha_{at}) \qquad t \in \{0, \dots, h-1\}, j \in J \tag{4.1j}$$
$$\sum_{t' \geq t} b_{jt'} \leq M(1 - \delta_{ot}) \qquad t \in \{0, \dots, h-1\}, j \in J \tag{4.1k}$$

Some variables have a transitive relation, these are formulated by Constraints 4.1l, 4.1m, 4.1n, and 4.1o. Remark that we currently do not allow changing the order of the arriving trains. In Section 4.2, we will explain how the formulation can be adjusted to allow the change the order of the arriving trains.

$$\alpha_{at} \geq \alpha_{at+1} \qquad a \in A, t \in \{0, \dots, h-2\} \tag{4.1l}$$
$$\alpha_{at} \geq \alpha_{a+1t} \qquad a \in A \backslash \{\text{last train}\}, t \in \{0, \dots, h-1\} \tag{4.1m}$$
$$\psi_{ot} \geq \psi_{ot+1} \qquad o \in O, t \in \{0, \dots, h-2\} \tag{4.1n}$$
$$\delta_{ot+1} \geq \delta_{ot} \qquad o \in O, t \in \{0, \dots, h-2\} \tag{4.1o}$$

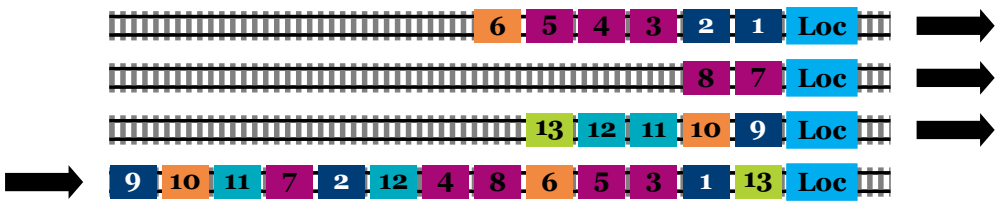
A train can be in at most one state (no cars on the classification tracks yet, building, and departed), hence we add Constraint 4.1p. Remark that the sum of the three variables may be zero if all cars for an outbound train that are already on the classification tracks are all part of one or multiple reclassification part(s).

$$\psi_{ot} + p_{ot} + \delta_{ot} \leq 1 \qquad o \in O, t \in \{0, \dots, h-1\} \tag{4.1p}$$

Example 4.1 (Interpreting a solution found using the Direct Formulation)**.** To illustrate how solutions from the formulation can be interpreted, we will consider the situation where there are three outgoing trains that should be formed from a single incoming train. We allow two reclassifications and assume that there are sufficient classification tracks (five in this case, two for reclassification and three for train formation).

We have the outgoing trains as depicted in Railway State 4.3. Here, each color represents a destination group, the order of the cars within each group is free. On the last track, we present the incoming train.

After the instance is solved using the DF, we get the solution as presented in Table 4.1.

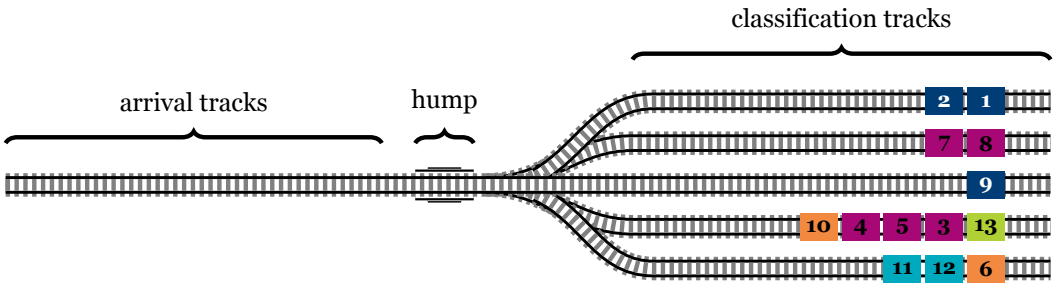


Railway State 4.3: Goal outbound trains and incoming train for Example 4.1.

Car	Sol.
9	00
10	01
11	10
7	00
2	00
12	10
4	01
8	00
6	10
5	01
3	01
1	00
13	11

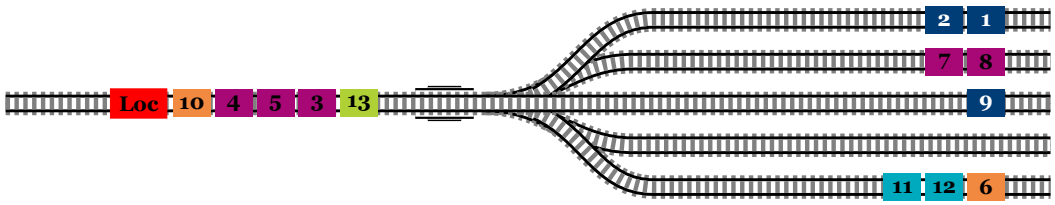
Table 4.1: Solution of instance as presented in Railway State 4.3 with two reclassifications.

In the initial roll after the arrival of the incoming train, we sort the cars from the inbound trains according to the solution. As explained previously, all cars with an all-zero solution will go directly to the classification track on which their respective outbound train is formed. We will use the three top tracks for this as seen in Railway State 4.4. The fourth (fifth) track will be used for the cars part of the first (second) reclassification. The cars with a 1 in the first column (from the right) will be put on the fourth classification track as seen from above. The other cars (i.e. those with a 0 in the first and a 1 in the second column), will be placed on the fifth track.

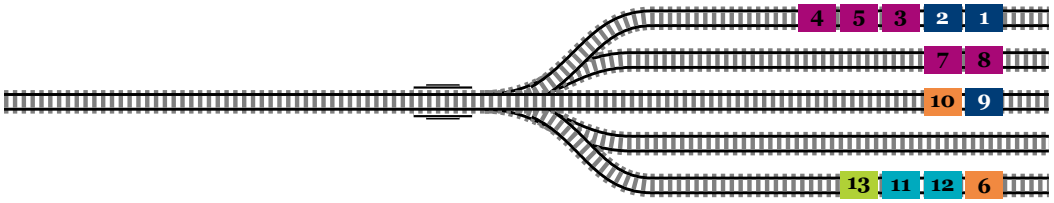


Railway State 4.4: Location of the cars after the initial roll-in of Example 4.1.

After this, the first classification track is pulled for reclassification. We remark that all cars except car 13 have all zeroes after the first column and are thus sent to the tracks of their train. Car 13 has a 1 in the second column and is thus placed on the second classification track (see Railway States 4.5 and 4.6).

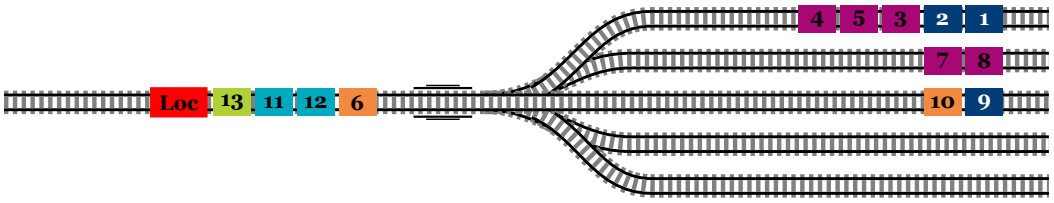


Railway State 4.5: The cars that were on the fourth classification track have been pulled and prepared for reclassification.

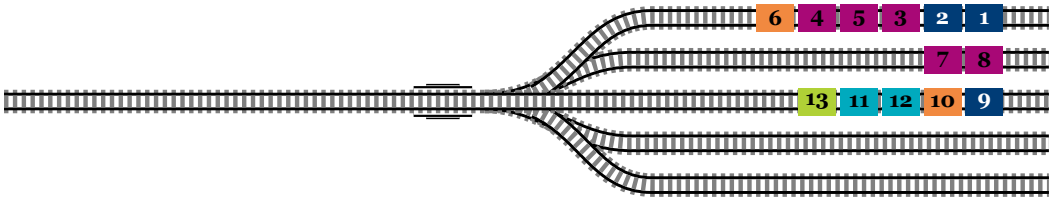


Railway State 4.6: The first reclassification is finished.

The last step is to pull and roll the second classification track. Since this is the last step, all cars will be sorted to the track of their departing train (see Railway States 4.7 and 4.7).



Railway State 4.7: The cars that were on the fifth classification track have been pulled and prepared for reclassification.



Railway State 4.8: The second reclassification has finished and the three desired outbound trains are located on the first, second, and third track.

This example clearly shows the power of reclassification in situations with limited space. In the incoming sequence, cars 13, 12, 11, 10, and 9 were in the opposite order of what they should be in the outgoing train. This train could be formed in three steps by pulling car 13 twice, first to make the order correct relative to cars 11 and 12 so that the three together formed a correctly sorted group. Then, these three were added to the in the meantime correctly sorted first two cars, 10 and 9. ◊

As mentioned before, the formulation presented is loosely inspired by the formulation by Márton *et. al.* ([29]). We will shortly go over the key differences between our formulation and the one presented in the paper. The way the order of the cars is managed coincides with the way it is done in their paper. However, we consider destination groups where there is no need for an exact sort within these groups. As mentioned before in the discussion of the constraints, this means that we will add these 'order'-constraints for all pairs of cars of adjacent destination groups. Furthermore, we had to the α-varialbes to Constraint 4.1a to make sure that shunting operations only 'count' if both cars are at the classification tracks. Otherwise, a car might be reclassified before the other car has yet arrived.

Another key difference is the way we handle arrivals and departures. The formulation by Márton *et. al.* does only consider the classification track capacity where we also take the arrival and departure tracks into account. This way, we can utilize the arrival and departure tracks to have trains temporarily stored at these tracks to open up space at the classification tracks.

Next, we also allow inbound trains to arrive during the planning horizon, where the original formulation assumes that all trains arrive at the beginning of the planning horizon.

Lastly, the formulation by Márton *et. al.* does allow for simultaneous classification, something we do not up to this point. However, we would like to point out that this could be added in the future if required.

4.2. The Flexible Arrival Formulation

The aforementioned formulation has one major assumption - the arrival trains arrive in a fixed order. However, due to the layout of Kijfhoek with multiple arrival tracks, we can change the order as we like (within the limitations of arrival and departure times and arrival track capacity). This is not possible in the DF and to that end, we created a slightly modified formulation, the Flexible Arrival Formulation (FAF), in which this is an option.

First of all, we will add a new variable, ξ_{ij}, where we have that

$$\xi_{ij} = \begin{cases} 1 & \text{if train } i \text{ is classified before train } j. \\ 0 & \text{otherwise.} \end{cases}.$$

With this new variable, we are able to upgrade Constraint 4.1a. For a pair of cars (l,k) which belong to the same incoming train, we keep the constraint as it is,

$$\sum_{t=0}^{h-1} 2^t \left(b_{lt} + \alpha_{a(l)t}\right) \geq \rho_{lk} + \sum_{t=0}^{h-1} 2^t \left(b_{kt} + \alpha_{a(k)t}\right)$$
$$(l,k)\text{s.t. } l,k \text{ part of same train and adjacent destination groups}.$$

For the pair (l,k) coming in on different inbound trains and departing in the same outbound train, let $\mathfrak{a}(k)$ and $\mathfrak{a}(l)$ be the respective inbound trains, we get

$$\sum_{t=0}^{h-1} 2^t \left(b_{lt} + \alpha_{a(l)t}\right) \geq \xi_{\mathfrak{a}(l)\mathfrak{a}(k)} + \sum_{t=0}^{h-1} 2^t \left(b_{kt} + \alpha_{a(k)t}\right)$$
$$(l,k)\text{s.t. } l,k \text{ part of same outbound train and adjacent destination groups}.$$

Next to these constraints, we also need to modify the Constraints 4.1m. These handle the transitivity of the arrival times of the different incoming trains. Due to the fact that we can now change the arrival order, we will add Constraint 4.2 for each pair (i,j) of incoming trains to replace Constraint 4.1m.

$$\alpha_{at} \geq \alpha_{a+1t} \qquad a \in A\backslash\{\text{last train}\}, t \in \{0,\dots,h-1\} \qquad (4.1\text{m})$$
$$a_{it} + \xi_{ji} \geq a_{jt}. \qquad i,j \in A, t \in \{0,\dots,h-1\} \qquad (4.2)$$

For the order of incoming trains, it is evident that the triangle inequality must hold to guarantee transitivity. So, we will add for each triplet (i,j,k) the following inequality:

$$\xi_{ij} + \xi_{jk} \geq \xi_{ik} \qquad i,j,k \in A.$$

Lastly, we know that trains can not be classified at the same time, hence we have for each pair (i,j) that the following must hold:

$$\xi_{ij} + \xi_{ji} = 1 \qquad i,j \in A.$$

4.3. The Arrival Order Heuristic

Preliminary experiments showed that reducing the number of constraints related to the order of the cars positively effects the objective value. Especially when we focus on the ones where two cars should be reversed (i.e. $\rho_{lk} = 1$ in Constraint 4.1a).

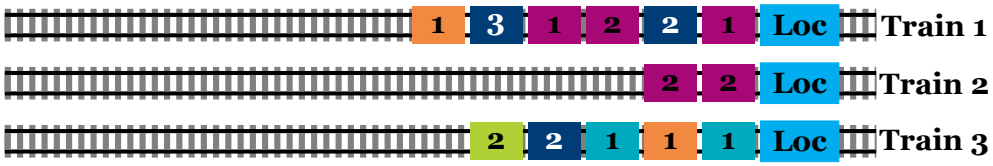
To that end, we propose a heuristic that will find a better order for the arrival trains to reduce the number of cars that should be reversed after the initial roll-in. Remark that we can determine the number of constraints for each sorted pair of two arriving trains.

For n arrival trains we can create a cost matrix $\mathcal{C}$ where c_{ij} is the number of violated order constraints (i.e. $\rho_{lk} = 1$) if train i was rolled before train j and no classifications would happen (see Algorithm 1 for the calculation of $\mathcal{C}$).

Algorithm 1 Calculation of the cost matrix C.

```
1: procedure findCostMatrix(set of n trains)
2:     for i ← 1 to n do
3:         for j ← 1 to n do
4:             c_ij ← 0
5:             for each Car k in train i do
6:                 for each Car l in train j do
7:                     if Cars k and l are assigned to the same outbound train and car l is in the destination group before the destination group of car k then
8:                         c_ij ← c_ij + 1
9:                     end if
10:                end for
11:            end for
12:        end for
13:    end for
14: end procedure
```

Example 4.2 (Cost Calculation)**.** Consider that we have the incoming trains as depicted in Railway State 4.9. Here, cars with the same color should form an outbound train and the number indicates the destination group. Cars with the same destination group can be in any order, the destination groups should be ordered correctly (i.e. first group 1, then group 2, ...).



Railway State 4.9: Trains of Example 4.2.

Let us find the costs for each pair of trains. First Trains 1 and 2. The only cars with a common outbound train are the purple ones. Hence, we look at the destination groups and find that if Train 2 is classified after Train 1, the cars are in the correct order. Remark that the third and fourth cars of Train 1 are not in the right order, but since they are part of the same train, we do not consider this violation now. So $c_{12} = 0$. If Train 2 is classified first, however, there are four violations. Each of the cars of Train 2 is now classified before two cars of Train 1 that should be in the front of the outbound train. Hence, $c_{21} = 4$.

Now, if we look at Trains 2 and 3, we find that there is no pair of cars that should be on the same outbound train that are not part of the same inbound train, hence, we find that $c_{23} = c_{32} = 0$. So, no matter in which relative order we classify Trains 2 and 3, it will have no impact on the number of order violations.

The last pair of trains, Trains 1 and 3 have cars with a common outbound train. First, we find that the orange cars have the same destination group. This is different for the dark blue cars. Here we find that there is a car for destination group 2 on the third train and one for destination group 3 on the first train. Hence, if we roll Train 3 after Train 1, we have violation, so $c_{13} = 1$ and if we roll Train 3 first, all dark blue cars are in the correct relative order which gives $c_{31} = 0$.

This way, we get the cost matrix

$$C = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} \times & 0 & 1 \\ 4 & \times & 0 \\ 0 & 0 & \times \end{bmatrix} \end{array}.$$

◊

The goal of the arrival order heuristic is to find a permutation of the inbound trains such that the total cost of that permutation is minimized. The cost of a permutation

$$a_1 a_2, \dots, a_n,$$

is given by

$$\sum_{(i,j):\text{pos}(i)<\text{pos}(j)} c_{ij}.$$

Here pos(i) is the position of train i in the permutation. Going back to the cost matrix of Example 4.2, we have that the permutation (123) has costs

$$c_{12} + c_{13} + c_{23} = 0 + 1 + 0 = 1$$

whereas the permutation (312) has costs

$$c_{31} + c_{32} + c_{12} = 0 + 0 + 0 = 0.$$

Since the goal is to find a permutation of minimal cost, (312) is optimal.

Remark that for n inbound trains, the number of possible permutations is given by $n!$, this is too much to go over all options. Therefore we have developed a heuristic that will look for a better - not by definition optimal - permutation (see Algorithm 2). Given an instance with n arriving trains, we will create a partition where each set is a group of trains with arrival times that are the same or close. For each of these sets, we will then find the optimal permutation.

We have already defined the cost function and we will use Simulated Annealing to solve the problem based on the algorithm described in Chapter 5. We will use three neighborhoods:

Swap In a sequence

$$a_1 a_2, \dots, a_i, \dots, a_j, \dots, a_n,$$

swap two randomly selected trains i and j to get

$$a_1 a_2, \dots, a_j, \dots, a_i, \dots, a_n.$$

Mirror In the sequence

$$a_1 a_2, \dots, a_i, a_{i+1}, \dots, a_{j-1}, a_j, \dots, a_n$$

select two indices i and j and reverse the order of all trains within the interval $[i, j]$[1] to get

$$a_1 a_2, \dots, a_j, a_{j-1}, \dots, a_{i+1}, a_i, \dots, a_n.$$

Push In the sequence

$$a_1 a_2, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_{j-1}, a_j, a_{j+1}, \dots, a_n$$

select two indices i and j and 'move' train i to place j (can be both forward and backward) to

$$a_1 a_2, \dots, a_{i-1}, a_{i+1}, \dots, a_{j-1}, a_j, a_i, a_{j+1}, \dots, a_n.$$

[1]Without loss of generality assume $i < j$.

Algorithm 2 Arrival Order Heuristic

1: **procedure** `arrivalOrderHeuristic`(set of incoming trains, number of time steps h)
2: **for** $t \leftarrow 1$ **to** h **do**
3: $\mathcal{C} \leftarrow$ `findCostMatrix`(all that arrive at t)
4: Apply `Simulated Annealling` with the neighborhoods `Swap`, `Mirror`, and `Push` to find a good permutation.
5: **end for**
6: **end procedure**

5
Simulated Annealing

In the previous chapter, we have seen how we can formulate the problem as an IP using the DF. However, solving IPs can become time-consuming. To that end, we will use heuristics to get good solutions quickly.

In this chapter, we will look into how we can find solutions for the problem at hand using Simulated Annealing (SA). First, in Section 5.1 we will give an introduction into local search techniques in general. Then, in Section 5.2, we describe how a natural phenomenon (Annealing) can be translated into an algorithm to solve combinatorial problems. Following the algorithm, in Section 5.3, we will lay out the theoretical framework for SA. Lastly, in Section 5.4, we describe our choices made concerning the configuration of the algorithm.

5.1. Local Search Techniques

Local Search is an iterative optimization technique. This means that we will take a given solution to a problem and look for 'close' solutions that might be an improvement. There are many different local search algorithms, but they can be divided up into two categories. Algorithms where we will only look to improve upon the current solutions and algorithms that will accept 'worsening' steps sometimes to get to a better solution in the long run.

The first type of algorithms solely serves the purpose of *intensification*, hence, it looks for solutions that are quite 'close' to the original one, only better. The second type of algorithms also incorporates *diversification*. This means that it will look for solutions 'further away' if they turn out to be better and/or no better solutions exist closely.

Example 5.1 (Intensification and Diversification)**.** In order to understand the difference between intensification and diversification, consider the function

$$f(x) = \sin(x) + \cos\left(\frac{3x\pi}{2}\right) + \sin(3x) + 3$$

as an example (see Figure 5.1). Assume that we want to minimize this function for $x \in [0, 5]$ and that we start at around $x = 2$.

If we are only allowed to move in a direction that will result in an improvement, we end up in a local minimum with $f(x) \approx 2$. This is the goal with intensification, stay close to the existing solution and find the closest local optimum.

However, if we allow ourselves to take a big step (diversification) and increase x to 4.5, we could end up in the global optimum solution with $f(x) \approx 0$.

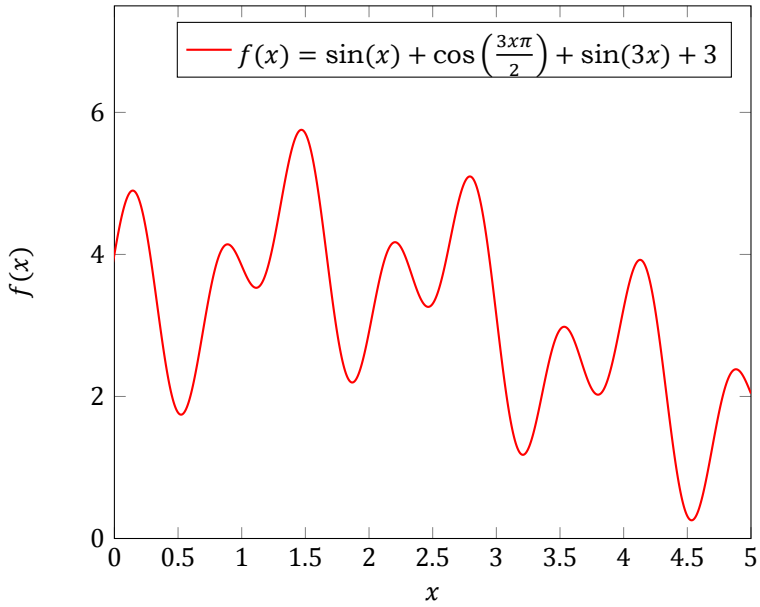


Figure 5.1: Example 5.1

◊

Solutions are changed using predefined neighborhoods. To have a good algorithm, it is paramount to have good neighborhoods which allow the algorithm to find the optimal solution.

Definition 5.1 (Neighborhood & Neighbors)**.** A **Neighborhood** $N(i)$ of state i is a set of neighbors of state i. For an initial solution, we define a **neighbor** to be a solution that is changed according to a predefined procedure. All neighbors in a neighborhood are changed following the same procedure.

Example 5.2 (Neighborhood & Neighbors)**.** For the Knapsack[1] problem, we can define the neighborhood of adding an item to the knapsack. For an instance where there are items i, j, and k which are not in the knapsack, a neighbor could be to add either item i, j, or k. These three neighbors together from one neighborhood. ◊

5.1.1. Iterative Improvement

The first algorithm we will discuss is the Hill-Climbing algorithm or Iterative Improvement (II). As mentioned before, this is the most basic way of applying local search. In general, with II, you look for a neighbor that is an improvement and keep doing this until there is no neighbor that improves the objective value.

To find a neighbor that improves the solution, you can either randomly select one and accept this if it is an improvement, or go over (a subset of) all neighbors and select the best option (i.e. the one that improves the solution the most).

Iterative Improvement is a very effective way to find the optimal solution in a solution space with very little local optima (only one in the ideal scenario). However, if there are multiple local optima, it can be beneficial to run the algorithm multiple times, with a different initial solution each time. This way, all local optima will pass by and the best can be selected. Nonetheless, if there are many local optima, this class of algorithms will only be effective in special cases and/or when there is a need for finding a local optimum close to the initial solution.

[1]For the sake of brevity, we have left out the definition of this problem, you can find more here: `https://en.wikipedia.org/wiki/Knapsack_problem`

5.1.2. Metaheuristics

The other flavor of local search will also allow 'worsening' steps under some conditions. This way, the algorithm can 'escape' from local optima and continue looking for a better local optimum. Remark that at all times, we will store the best-known feasible solution. We will briefly discuss two well-known techniques that we will not use in this thesis - Tabu Search and Genetic Algorithms.

5.1.2.1 Tabu Search

Tabu Search is a metaheuristic that will accept 'worsening' steps if there are no neighbors available that will improve the solution. Tabu Search differs from other metaheuristics in that it keeps a list of 'tabu moves'. This list specifies solutions - or features of solutions - which cannot be considered. This way, the algorithms prevent oscillations.

Going back to the Knapsack example. A 'tabu move' might be to add or remove a given item from the knapsack. Once a neighbor is accepted, the old solution will be added to the tabu list. After a predefined number of steps, t, the item that was added earliest will be removed from the list. So, Tabu Search differs in two main ways from the II algorithm. Foremost in that it will accept 'worsening' steps if there are no improvements left to be made and secondly, that it keeps a list of solutions that cannot be accepted for several steps. [17, 18]

5.1.2.2 Genetic Algorithms

Genetic Algorithms are based on the theory of 'Survival of the fittest'. With this class of algorithms, there is a set of initial solutions which form the first population. At each step, a subset of solutions is chosen. Within this subset, pairs of solutions (parents) are used to create new solutions (children) by combining features from both solutions. If the new solutions are better than the solutions they originate from, they will replace their 'parents'. Another way solutions can be modified is via mutations. Just like in real-life, solutions can mutate - independent from the parent-solutions. Possibly for the worse, but hopefully for the better.

This type of algorithms allows for the escape from local optima by having no restrictions on the kind of moves that are allowed. Where both Tabu Search and SA solely depend on neighborhoods to get to new solutions, Genetic Algorithms also combines features from different solutions to create new ones without any restrictions on how much the solutions should be alike. [36]

For this thesis, we have chosen to go with SA to solve our shunting problem. We have chosen this method because it fits best with the characteristics of our problem. With many cars, trains, and tracks, the solution space is very high-dimensional. This makes it hard to consider all neighbors (or a significant portion of them) at each iteration, making Tabu Search less useful as it would take too much time to consider (a subset of) all neighbors at each step. Genetic Algorithms face the same issue. Because of the high number of decision variables, creating children will take up a considerable amount of time - both to create the children as well as to check the feasibility of the solutions.

5.2. Annealing in Nature | Physics

The history of SA dates back to the 1980s, when it was first used by Kirkpatrick *et. al.* ([24]) and Cerny ([10]). They based their solution approach on a model created by Metropolis *et. al.* ([30]) three decades prior. We will first describe the *metal coolling*-side and then describe how the annealing process is translated into SA.

5.2.1. Metal coolling

In 1953, Metropolis *et. al.* ([30]) described a way to simulate the annealing process. Annealing is a thermodynamical process where crystals are formed in metals. When a metal is in a 'frozen' state, the atoms in the lattices don't move. However, if the temperature is increased, the atoms may start to move to a different state of lower total energy. As the total energy of the system gets lower, the crystals that form will be longer as they 'fall' into place and there will be fewer imperfections.

The formation of these crystals takes time, and just like with many combinatorial problems, there is no direct way without bumps (i.e. a temporary increase in energy before a further decrease) to get there. For each temperature T of the system, the probability that the system will transition to another state is given by $\exp\left(\frac{-\Delta E}{k_B T}\right)$. Here, ΔE is the change in temperature and k_B Boltzmann's constant. Decreasing the temperature too fast will result in shorter crystals as there is less time to transition into these lower energy states. [24, 30]

5.2.2. Translation to Simulated Annealing

Thirty years after the simulation model for the annealing process was first presented, Kirkpatrick *et. al.* ([24]) and Cerny ([10]) found a way to translate the phenomenon to a way of solving combinatorial problems (see Algorithm 3).

The block of material in which the crystals are formed can be seen as the solution space in which we are looking for a solution (state) of minimal cost (energy). After we have found an initial solution (start state), we increase the parameter T_0 (temperature), and then while we decrease (cool down) this parameter, we will accept 'worsening steps' according to the Boltzmann distribution. [14, 32]

Algorithm 3 Basic SA

```
1:  procedure basicSA(start temperature T_0, M)
2:      S ← createInitialSolution
3:      S_best ← S
4:      while Stopping criterion has not been met do
5:          for 1 → M do
6:              Select a random neighbor N and intitialize.
7:              R ← random(0, 1)
8:              if e^(Δ(N)/T) ≥ R then
9:                  Accept the neighbor
10:                 if S < S_best then
11:                     S_best ← S.
12:                 end if
13:             else
14:                 Reject the neighbor
15:             end if
16:         end for
17:         T_{k+1} ← T_k                    According to the cooling schedule.
18:     end while
19:     return S_best
20: end procedure
```

5.3. Theoretical Framework

When we use an algorithm, one of the things we are interested in is whether or not it will converge to an optimal solution in reasonable time. Before we look at the convergence of SA, we will illustrate how the evolution of the cost during the run can be seen as a Markov Chain.

5.3.1. Markov Chains

The way SA finds a solution can be seen as a discrete-time Markov Chain where the solution space $\mathcal{S}$ is given by all possible solutions for the problem at hand. To find the probability that we will move from one solution to the other, we need to determine the probability that we will *generate* and *accept* the goal solution.

Definition 5.2 (Distrete-Time Markov Chain)**.** A sequence $(X_n)_{n\geq 0}$ with $X_n \in I$ for all n and state space I is a **Discrete-Time Markov Chain** if X_n can take any value in the state space independent of n and if

$$\mathbb{P}(X_{n+1} = i_{n+1} | X_n = i_n, \dots, X_1 = i_1) = \mathbb{P}(X_{n+1} = i_{n+1} | X_n = i_n).$$

A Markov Chain is **homogeneous** if for all $i, j \in I$, we have that $\mathbb{P}(X_{n+1} = j | X_n = i)$ is independent of n, in that case, we say that $P_{ij} = \mathbb{P}(X_{n+1} = j | X_n = i)$. If $\mathbb{P}(X_{n+1} = j | X_n = i)$ is dependent on n, the Markov Chain is **inhomogeneous**. ⧫

First of all, we know that $\mathcal{N}(i)$ is the set of all neighbors of a solution i. Thus, for all $\eta \in \mathcal{N}(i)$, η is part of at least one neighborhood of i. Remark that the Markov Chain that defines the sequence of solutions that is visited by the algorithm is homogeneous for a fixed temperature. So between temperature changes, the Markov Chain is homogeneous, but for solutions that were considered during different temperatures, the Markov Chain is inhomogeneous.

First, we will discuss the probability of considering a given neighbor j given that we are have solution i. We define this *generating probability* as G_{ij}. Remark that we are free to choose the distribution of G as long as it adheres to the following three conditions for each solution i:

$$\begin{aligned} G_{ij} &= 0 && \text{if } j \notin \mathcal{N}(i) \\ \sum_{j \in \mathcal{N}(i)} G_{ij} &= 1 && \\ G_{ij} &\geq 0 && \forall j \end{aligned}$$

In our implementation of the algorithm, we will first uniformly select a neighborhood from all neighborhoods. Then, within this neighborhood, we will uniformly select a neighbor from this neighborhood. Note that this distribution fits the three criteria as described above. Furthermore, we note that $G_{ij} = G_{ji}$.

We are not only interested in how likely it is that we will generate a given neighbor, we are also interested in the *acceptance probability* (i.e. given that we have generated a neighbor, how likely is it that we will accept it). For this, we will use a slightly modified Boltzmann distribution. As discussed previously, the likelihood of going from one state to the other depends on the change in costs (energy). For an acceptance function f and two solutions i and j, we say that solution j is better than i if $f(j) < f(i)$. If this is the case, we will accept the new solution immediately. If not, we will accept it with probability $\exp\left(\frac{f(i)-f(j)}{T}\right)$. So,

$$A_{ij}(T) = \begin{cases} 1 & \text{if } f(j) < f(i) \\ \exp\left(\frac{f(i)-f(j)}{T}\right) & \text{otherwise} \end{cases}.$$

Now, we can define the distribution of transition probabilities P_{ij} as follows:

$$P_{ij}(T) = \begin{cases} G_{ij} A_{ij}(T) & \text{if } i \neq j \\ 1 - \sum_{l \in \mathcal{N}(i)} G_{il} A_{il}(T) & \text{if } i = j \end{cases}. \tag{5.1}$$

5.3.2. Convergence

In this subsection, we will show that while SA might be fast, there are no guarantees that it will converge to an optimal solution, or to a solution close to an optimal solution.

Consider the transition probabilities as presented in Equation 5.1. We remark that this is a homogeneous Markov Chain under the condition that T is fixed.

Theorem 5.1. The stationary distribution $\vec{\pi}(T)$ of the homogeneous discrete-time Markov Chain with transition probabilities as given in Equation 5.1 is given by

$$\pi_i(T) = \frac{\exp\left(\frac{-f(i)}{T}\right)}{\sum_{j \in \mathcal{S}} \exp\left(\frac{-f(j)}{T}\right)} \tag{5.2}$$

where $\mathcal{S}$ is the solution space. [14, 31]

Proof. To prove this stationary distribution, we need to show that for the matrix of transition probabilities $P(T)$, we have that

$$\vec{\pi(T)}P(T) = \vec{\pi(T)}.$$

Which is equivalent to

$$\sum_{j \in \mathcal{N}(i) \cup \{i\}} \pi_j(T) P_{ji}(T) = \pi_i(T) \qquad \forall i. \tag{5.3}$$

First, we show that $\pi_i(T)P_{ij}(T) = \pi_j(T)P_{ji}$. Take $\frac{\pi_i(T)}{\pi_j(T)}$, with the given distribution, we know that is is equal to

$$\frac{\pi_i(T)}{\pi_j(T)} = \frac{\exp\left(\frac{-f(i)}{T}\right)}{\exp\left(\frac{-f(j)}{T}\right)} = \exp\left(\frac{f(j) - f(i)}{T}\right).$$

When we consider $\frac{P_{ji}(T)}{P_{ij}(T)}$, we find that this is equal to

$$\frac{P_{ji}(T)}{P_{ij}(T)} = \frac{G_{ji}A_{ji}(T)}{G_{ij}A_{ij}(T)} = \frac{A_{ji}(T)}{A_{ij}(T)}.$$

If $f(j) > f(i)$ this is equal to

$$\frac{\exp\left(\frac{f(j)-f(i)}{T}\right)}{1} = \exp\left(\frac{f(j) - f(i)}{T}\right)$$

and if

$$f(j) \leq f(i)$$

it is equal to

$$\frac{1}{\exp\left(\frac{f(i)-f(j)}{T}\right)} = \exp\left(\frac{f(j) - f(i)}{T}\right),$$

which is the same. Hence, we find that

$$\frac{P_{ji}(T)}{P_{ij}(T)} = \frac{\pi_i(T)}{\pi_j(T)}$$

and thus

$$\pi_i(T)P_{ij}(T) = \pi_j(T)P_{ji}(T).$$

We can now use this result in Equation 5.3 to prove the distribution as given in Equation 5.2:

$$\sum_{j \in \mathcal{N}(i) \cup \{i\}} \pi_j(T) P_{ji}(T) = \sum_{j \in \mathcal{N}(i) \cup \{i\}} \pi_i(T) P_{ij}(T)$$

$$\pi_i(T) \sum_{j \in \mathcal{N}(i) \cup \{i\}} P_{ij}(T) = \pi_i(T).$$

This way, we have proven that

$$\pi_i(T) = \frac{\exp\left(\frac{-f(i)}{T}\right)}{\sum_{j\in\mathcal{S}} \exp\left(\frac{-f(j)}{T}\right)} \tag{5.4}$$

is the stationary distribution for the Markov Chain chararterized by the transition probabilities as given in Equation 5.1. [31] □

Theorem 5.2. Let $\hat{\mathcal{S}}$ be the set of locally optimal solutions. Then, we find that

$$\lim_{t\downarrow 0} \pi_i(T) = \begin{cases} \frac{1}{|\hat{\mathcal{S}}|} & \text{if } i \in \hat{\mathcal{S}}. \\ 0 & \text{otherwise.} \end{cases}.$$

[14, p. 1631]

Proof. First, consider a solution i that is part of the set of locally optimal solutions. Then, we have that by Theorem 5.1

$$\begin{aligned}\lim_{T\downarrow 0} \pi_i(T) &= \lim_{T\downarrow 0} \frac{\exp\left(\frac{-f(i)}{T}\right)}{\sum_{j\in\mathcal{S}} \exp\left(\frac{-f(j)}{T}\right)} = \lim_{T\downarrow 0} \frac{\exp\left(\frac{-f(i)}{T}\right)}{\sum_{j\in\hat{\mathcal{S}}} \exp\left(\frac{-f(j)}{T}\right)} \\ &= \lim_{T\downarrow 0} \frac{\exp\left(\frac{-f(i)}{T}\right)}{\exp\left(\frac{-f(i)}{T}\right)\sum_{j\in\hat{\mathcal{S}}} 1} = \lim_{T\downarrow 0} \frac{1}{|\hat{\mathcal{S}}|} = \frac{1}{|\hat{\mathcal{S}}|}.\end{aligned}$$

In the case that solution i is not part of the set of locally optimal solutions, we have that there is at least one solution $j \in \hat{\mathcal{S}}$ with $f(j) < f(i)$ and thus, we have that

$$\begin{aligned}\lim_{T\downarrow 0} \pi_i(T) &= \lim_{T\downarrow 0} \frac{\exp\left(\frac{-f(i)}{T}\right)}{\sum_{j\in\mathcal{S}} \exp\left(\frac{-f(j)}{T}\right)} = \lim_{T\downarrow 0} \frac{\exp\left(\frac{-f(i)}{T}\right)}{\sum_{j\in\hat{\mathcal{S}}} \exp\left(\frac{-f(j)}{T}\right)} \\ &\leq \lim_{T\downarrow 0} \frac{\exp\left(\frac{-f(i)}{T}\right)}{\exp\left(\frac{-f(j)}{T}\right)} = \lim_{T\downarrow 0} \exp\left(\frac{-f(i)+f(j)}{T}\right) = 0.\end{aligned}$$

□

By Theorem 5.2, we know that if we have a cooling schedule for which we have that $\lim_{k\to\infty} T_k = 0$ and we have an infinite amount of iterations, we will end up in a local optimum.

Definition 5.3 (Height of two communicating solutions)**.** For two solutions i and j we say that they communicate at **height** h if there is a path from i to j where for all solutions along the path, there is no solution with costs higher than $f(i) + h$.

Definition 5.4 (Depth of a combinatorial optimization problem)**.** The **depth** d of a combinatorial problem is the smallest value such that for all solutions i and j where j is part of the global optima they communicate at height at most d.

Example 5.3 (Depth & Height)**.** To illustrate the difference between the height and depth, consider Figure 5.2. Here, we have two solutions i and j (green dots). There is only one path from i to j. To do so, the solution value will need to increase by as much as the blue arrows point out, we call this the height.

The depth is the smallest value so that all solutions communicate with a global optimum. In the given figure, there are two extreme minima, the one on the right is only a local minimum and the one on the left is the global minimum. In this case, the depth is given by the difference in solution value of the local minimum and the top of the 'hill' to the left.

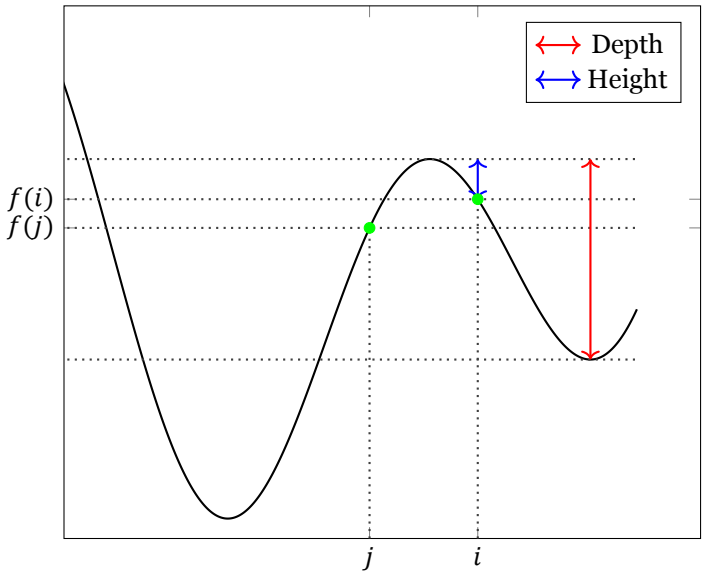


Figure 5.2: Example 5.3

◊

In his paper, Hajek ([19]), proved that there exists a cooling schedule such that the SA algorithm will converge to an optimal solution. Given that there is an infinite amount of iterations at each step.

Theorem 5.3 (Hajek)**.** The SA algorithm will convergence to a global optimum if and only if

$$\lim_{k\to\infty} T_k = 0$$

and

$$\sum_{k=1}^{\infty} \exp\left(\frac{-d}{T_k}\right) = \infty$$

where d is the depth of the problem. [19, p. 4]

A cooling schedule that meets the requirements stated in Theorem 5.3 is

$$T_k = \frac{d}{\log k}.$$

However, finding the depth of a combinatorial optimization problem is $\mathcal{NP}$-hard[3, 14]. Furthermore, as was the case with Theorem 5.2, there is no bound on the required number of iterations to reach the stationary distribution of Theorem 5.1. This means that there is no guarantee that the SA algorithm will converge to a global optimum within reasonable time. However, in practice, SA algorithms will converge to solutions (close to) the optimal solution in reasonable time.

5.4. Algorithm Configuration

Before applying SA to a problem, there are decisions to be made. First, we discuss the problem-specific decisions. To start with, we will shortly discuss which problem we solve and how we have added penalty costs to the objective function. Following, we discuss how we will create an initial solution and which neighborhoods we will use. Next, we will discuss the three main decisions related to the algorithm itself, the starting temperature, cooling scheme, and stopping criterion.

5.4.1. Problem-Specific

5.4.1.1 Problem & Objective Function

To find solutions to the shunting problem, we will use the same structure (binary encoding) as introduced in Chapter 4. However, we make two changes. First, because of the complexity of determining the number of classification tracks used for reclassification parts (the μ variables in DF | FAF), we reserve a fixed (adjustable) amount of tracks for reclassification parts. Furthermore, we change the objective function slightly. With SA, we allow infeasible solutions during the run, however, at a cost (see Table 5.1).

Violation	**Penalty Cost (Relative to *Carrolls*, per violation)**
Cars are in the wrong order in an outbound train.	4
Cars are classified before arrival.	2.5
Cars are classified after departure.	2.67
Trains arrive before scheduled arrival time.	1.33
Trains depart after scheduled departure time.	2.67
More arrival tracks are needed than available.	2.67
More classification tracks are needed than available.	2.67
More departure tracks are needed than available.	3.33
If the order in which arrival trains should arrive is ignored.	2.33
If a train is being formed before a classification track is assigned to that train.	1.33
General penalty, if there is at least one violation	33.33

Table 5.1: Penalty costs for Local Search based algorithms.

5.4.1.2 Initial Solution

To start the algorithm with an as good as possible solution, we try to find an initial solution that should be close to, or already, feasible. We use Algorithm 4 to find the initial solution.

To begin with, we use a 'zero'-solution. No cars are reclassified, hence the `schemeValue` (integer value of bit representation) is zero for all cars and the arrival | departure times for the trains are set at the scheduled arrival | departure times.

The first loop (start at Line 4) will try to resolve as many issues as possible regarding the classification schedules. There will at most `noCars` (total number of cars) attempts and if nothing has changed, the variable `lastUpdate` will make sure we won't do any unnecessary checks.

The first check (Line 6) will be if any cars have a `schemeValue` less than or equal to one of the cars it should have a strictly larger value for. If this is the case, we will update the value of this car to twice the value of the car it should be larger than (remark that this is equivalent to shifting all bits one step).

Next, we check (Line 10) if there are any transitivity constraints violated. Hence, if the `schemeValue` of a given car should at least be as large as that of another car. Instead of updating the `schemeValue` to twice the value of the other car, we just make them equal.

With the next two checks (Lines 15 and 19), we make sure cars are only classified after arrival and before departure.

Algorithm 4 Initial Solution

```
1: procedure createInitialSolution
2:     lastUpdate ← true
3:     counter ← 0
4:     while lastUpdate and counter < noCars do
5:         lastUpdate ← false
6:         for All 'reverse order' constraints do                    i.e. constraints where ρ_lk = 1.
7:             Fix violation by increasing schemeValue of the violating to that of two times the value
   of the other.
8:             lastUpdate ← true
9:         end for
10:        for All 'transitivity order' constraints do               i.e. constraints where ρ_lk = 0.
11:            Fix violation by increasing schemeValue of the violating to that of the other.
12:            lastUpdate ← true
13:        end for
14:        for All cars c do
15:            if c is classified before arrival then
16:                schemeValue(c) = 2^(arrival time car c)
17:                lastUpdate ← true
18:            end if
19:            if c is classified after departure then
20:                schemeValue(c) = 2^(min{departure time car c,h})
21:                lastUpdate ← true
22:            end if
23:        end for
24:        counter ← counter +1
25:    end while
26:    for All outbound train d do
27:        if Only one destination group served in train d then
28:            With probability 0.5, move train as early as possible to departure track.
29:        end if
30:    end for
31: end procedure
```

5.4.1.3 Neighborhoods

The Local Search model is equipped with six neighborhoods. In this subsection, we will go over all neighborhoods one by one. In total, we have tested twelve neighborhoods from which these six were selected. During preliminary experiments, we found that the six neighborhoods that weren't selected were seldom accepted during a run. For the sake of completion, we will include the description of the non-selected neighborhoods in Appendix A.

Single In this neighborhood, we will change the reclassification scheme for one of the cars. We will select a random integer as `schemeValue` from the set

$$\{v | 2^{\text{arrival time}} \leq v < 2^{\text{departure time}}, v \pmod{2^{\text{arrival time}}} = 0\} \cup \{0\}.$$

Example 5.4 (Neighborhood: Single)**.** Consider the solution as given in Table 5.2. Here, there are eight cars, each have their own arrival and departure time. For each car, we have the solution and for ease of use, we have also added the `schemeValue`. With the Single neighborhood, we will randomly pick one car, in this case Car 7. The car is currently reclassified once, in the seventh reclassification. Since the arrival time of the car is 4 and the departure time is 7, we can select any integer in the set

$$\{0, 16, 32, 64, 48, 80, 96, 112\}.$$

Car	Arrival Time	Departure Time	Solution	schemeValue
1	0	4	0 0 0 0 1 0 0	4
2	0	5	0 0 0 1 0 0 0	8
3	0	5	0 0 0 1 0 1 0	10
4	1	6	0 0 0 0 1 0 0	8
5	3	7	1 0 1 1 0 0 0	88
6	3	7	0 1 1 0 0 0 0	48
7	4	7	1 0 0 0 0 0 0	64
8	4	7	0 0 0 0 0 0 0	0

Table 5.2: Start solution Examples 5.4, 5.5, and 5.6.

Assume we select 48, in that case, we will update the solution so that we get the solution as presented in Table 5.3 - all changes are colored red.

Car	Arrival Time	Departure Time	Solution	schemeValue
1	0	4	0 0 0 0 1 0 0	4
2	0	5	0 0 0 1 0 0 0	8
3	0	5	0 0 0 1 0 1 0	10
4	1	6	0 0 0 0 1 0 0	8
5	3	7	1 0 1 1 0 0 0	88
6	3	7	0 1 1 0 0 0 0	48
7	4	7	0 1 1 0 0 0 0	48
8	4	7	0 0 0 0 0 0 0	0

Table 5.3: End solution Example 5.4.

◊

Switch For this neighborhood, the name says it all. We will select two cars at random and interchange the classification schemes.

Example 5.5 (Neighborhood: Switch)**.** We again consider the solution as given in Table 5.2. The first step in the Switch neighborhood is to select two cars. We select Cars 5 and 6. If we interchange the solution of both cars, we get the solution as given in Table 5.4.

Car	Arrival Time	Departure Time	Solution	schemeValue
1	0	4	0 0 0 0 1 0 0	4
2	0	5	0 0 0 1 0 0 0	8
3	0	5	0 0 0 1 0 1 0	10
4	1	6	0 0 0 0 1 0 0	8
5	3	7	0 1 1 0 0 0 0	48
6	3	7	1 0 1 1 0 0 0	88
7	4	7	1 0 0 0 0 0 0	64
8	4	7	0 0 0 0 0 0 0	0

Table 5.4: End solution Example 5.5.

◊

Shift With the shift neighborhood, we will start by selecting two time steps. Then, we will for all cars reclassified solely during that interval, we will shift all reclassifications. In terms of `schemeValue`, we will multiply this by two.

Example 5.6 (Neighborhood: Shift)**.** With the Shift neighborhood, we will first select two time steps, a start time step and an end time step. In this example, we choose 2 and 3 (remember, we start counting at zero). This means that all cars that are solely classified in the third and fourth column (as seen from the right) will be considered (see Table 5.2). This means that Cars 1, 2, and 4 qualify. Cars 3 and 5 are also classified during this interval but not exclusively. Car 3 is also classified at time step 1 and Car 5 at time step 4. If we shift the solution for these cars, we get the solution as given in Table 5.5. Notice that this is indeed a multiplication by a factor 2 of the `schemeValue`.

Car	Arrival Time	Departure Time	Solution	schemeValue
1	0	4	0 0 0 1 0 0 0	8
2	0	5	0 0 1 0 0 0 0	16
3	0	5	0 0 0 1 0 1 0	10
4	1	6	0 0 1 0 0 0 0	16
5	3	7	1 0 1 1 0 0 0	88
6	3	7	0 1 1 0 0 0 0	48
7	4	7	1 0 0 0 0 0 0	64
8	4	7	0 0 0 0 0 0 0	0

Table 5.5: End solution Example 5.6.

◊

Arrival Where all previous neighborhoods were focused on the classification schemes for cars, we will now look at the train level. As mentioned before, we can have trains waiting at the arrival tracks after arrival before they are classified. To this end we have two arrival times, the arrival time at the tracks and the classification time. In this neighborhood, we change the latter. We will increase|decrease the arrival time by one for a single train.

Departure This neighborhood is almost identical to the previous one, but instead of changing the classification time of an inbound train, we will update the transfer time of outbound trains in the same way.

FormationStart The last neighborhood has to do with the time at which we start the formation of an outbound train at the classification tracks (the p variables in DF | FAF). This neighborhood will increase|decrease this time by one for a single train.

5.4.2. Generic

The SA algorithm can be tuned in many different ways. The three main parameters are the starting temperature, the cooling scheme, and when to stop the algorithm.

5.4.2.1 Starting temperature

To find the starting temperature, there is a consensus that this should be done in the following manner. Before starting to optimize a solution, generate several neighbors (we will use 50). Then, take either the average or maximum change to get an approximation of the depth of the problem at hand. Then, scale this number by a factor to make sure that a limited number of neighbors is accepted in the beginning (we will use 0.03). [14, 37]

5.4.2.2 Cooling scheme

For the cooling scheme, there are plenty options. As we have seen previously, there is a cooling schedule that will converge in infinite time. However, since we won't be able to find the depth within reasonable time and because we won't perform an infinite number of iterations between cooling steps, we will test multiple cooling schedules to find which one will suit our problem the best. Some more direct ways to cool the temperature are as follows:

$$T_k = \alpha \cdot T \tag{5.5}$$

$$T_k = \frac{T_0}{1 + \beta k} \tag{5.6}$$

$$T_k = \frac{T_0}{1 + \gamma \log(k)} \tag{5.7}$$

For Option 5.5, it is advised use $\alpha \in [0.8, 0.999]$. Options 5.6 and 5.7 will use parameter k. This parameter is a counter of number of completed steps. Both also feature a tuning parameter in β and γ. Remark that Option 5.7 resembles the one found in Section 5.3.

More adaptable cooling schedules have been proposed by [22, 37]. They both use the standard deviation of the change in solution score to either accelerate (when there are little changes) or decelerate (when there are many changes) the cooling of T. [22, 37]

5.4.2.3 Stopping criterion

Lastly, the stopping criterion should be specified. Once more, there are multiple different options. For this thesis, we have chosen to perform a fixed number of iterations in contrast to a stopping criterion that might terminate early (if there are no changes for the past x iterations) or continue on longer (if there are still plenty of changes).

5.4.3. Reoptimizing

When we reoptimize a solution, we can set the number of steps that will be fixed. All events that took place at or before that time step can't be changed. If we reoptimize a solution, it might be that the existing solution is infeasible for the scenario. If that is the case, we can add a preprocessing step. In essence, we will use the same algorithm as we used for the initial solution (see Algorithm 4), of course, taking into account what we can and cannot change.

6
Simulator

To test how well the solutions of our algorithms hold up in practice, we have developed two simulators. In short, the static simulator will check how good a solution is for the given instances and perturbed versions (scenarios) of the instance. It can also calculate basic metrics like the number of reclassifications and number of required tracks.

The dynamic simulator will do the same, with one major difference. The static simulator looks at an instance and solution at one point in time, the start of the planning horizon. The dynamic simulator will check the solutions over time and can do this from any time step. When optimizing and/or creating perturbations it can 'lock' the events that took place in the past and only change future events.

Furthermore, with the static simulator, we assume that all information regarding the inbound | outbound trains is known at the beginning. This is not always the case with the dynamic simulator. To simulate real-life situations as well as possible, the dynamic simulator allows that information gets available during the run. Simply said, on Monday mornings, we don't know exactly which cars will be part of a train that arrives on Thursday in the afternoon. With the dynamic simulator, we can make this information available later on in the simulation (e.g. on Thursday morning) and adjust the solution accordingly.

Both simulators will be used to check if a solution is feasible and if the solutions stay feasible when scenarios are generated. If the latter is not the case, we will check if the present algorithms can recover the solution.

While the dynamic simulator can be seen as an extension of the static simulator, the latter has the advantage of being 'simple'. Additionally, in the implementation of both simulators, the static simulator is more suited for comparing original instances | solutions with scenarios | updated solutions. This is due to the fact that in the dynamic simulator, we focus on the events taking place and how the tracks are utilized at a specific time step and less on the solution for the entire planning horizon.

In this chapter, we will first show how we generate scenarios in Section 6.1. Then, in Section 6.2, we explain how the static simulator works. Lastly, we break down how the dynamic simulator works and how it is used to resemble real-life in Section 6.3.

6.1. Scenario Generation

With both simulators, we will test how well solutions hold up and/or how easy it is to recover a solution for perturbed instances. To create the scenarios, we implemented a scenario generator. First, we will give a short overview of the possible changes that can be made to the scenario. Then, we will take a closer look at how the changes are made.

Before we generate a scenario, we have to specify some parameters. For each possible perturbation, we specify the probability p (more on this in the specification) that it will occur and for changes to the arrival | departure times, the interval of the maximal changes.

Since the scenarios are also used in the dynamic simulator, we add an extra parameter, the number of fixed steps. This indicates which steps are 'in the past' (i.e. events before this time step). Events before this time cannot be changed.

For each instance, we can change it in the following ways:

- → Cancel inbound trains.
- → Change the arrival time of an inbound train.
- → Change the departure time of an outbound train.
- → Change the arrival order.
- → Remove arrival | classification | departure tracks.

Cancelling inbound trains With probability p, we delete a train. If a train is deleted, the cars of this train are not reassigned to another inbound train but are deleted as well.

Changing the arrival (departure) time of inbound (outbound) trains With probability p, we will change the arrival (departure) time of a train and we change the arrival (departure) time according to Algorithm 5[1]. Here, we also have to specify how much we can change the arrival (departure) time by setting `maxDown` and `maxUp`.

Algorithm 5 Scenario Generation: Change Arrival Time

```
1: procedure changeArrivalTime(oldArrivalTime, maxDown, maxUp, fixedSteps)
2:     lowerBound ←max(fixedSteps, max(0, oldArrivalTime - maxChangeDown))
3:     upperBound ←max(fixedSteps, oldArrivalTime + maxChangeUp)
4:     newArrivalTime ← 𝒰(lowerBound, upperBound)
5:     return newArrivalTime
6: end procedure
```

Changing the arrival order To determine the perturbations to the arrival order, we do the following. For each pair of subsequent trains, we do a chance experiment with probability p of succeeding. If successful and if the arrival times of the trains match, we swap them in the arrival order. Note that we can only do this if the arrival time is later than the number of fixed steps.

Removing arrival | classification | departure tracks To find the number of the n arrival | classification | departure tracks that should be removed in the scenario, we do n chance experiments with a probability p of succeeding. The number of times the experiment succeeded is the number of tracks that should be removed.

6.2. Static Simulator

The first simulator is the Static Simulator. Here, we assume that all information on the instance is known in advance. With this simulator, we will check if solutions are feasible and we can do a basic robustness analysis.

6.2.1. Robustness Analysis

To check how robust our solutions are, we test two things. First, we check if a solution is still feasible for a scenario generated based on the instance it was created for. Secondly, we check how well the solutions can be recovered with the different solution approaches that we can use. For each recovered solution, we calculate the difference with the old solution.

In order to calculate the difference between two solutions, we solely look at the reclassifications. We count the number of times that there is a car being reclassified in only one of the two solutions (`xor`-operator).

To do a solid robustness analysis, we have automated this process. We can specify how many different scenarios we would like to use to check the solution and how the solutions should be recovered if infeasible. After the analysis, the simulator will return the number of scenarios for which the solution stayed

[1] For the departure time, replace 'Arrival' with 'Departure'.

feasible and if not, how many violations there were. Next, it can tell how well the different algorithms were at recovering the solution and how much different the recovered solutions were when compared to the original solutions.

6.3. Dynamic Simulator

The Dynamic simulator will be used to test how the solutions hold up in real-life. This simulator is equipped with an option to have information become available over time.

6.3.1. Trains becoming available

To allow trains to become available, the instance has a 'shadow list' of trains. Before the simulation, a solution will be found for 'dummy trains' that temporarily replace the real trains. We will call this a dummy solution. The dummy trains have cars that represent destination groups that might be part of an inbound train. The destination groups that are part of an inbound train are based on historical data. The real trains are those that actually arrive, those are placed on the shadow list.

However, because the specifics of these are not yet known at the beginning of the planning horizon, we use dummy trains to cover all possible destination groups that might come in with a specific arrival train. At a given fixed time (e.g. one shift) before a train will arrive, the dummy will be replaced by the real train as was stored on the shadow list.

At the same time, the simulator will update the solution with the new information. For each car in the real train, we will look if there is a car in the dummy train with the same destination. If this is the case, the real car will get the same solution as the dummy car (i.e. at which time steps it should be reclassified).

Example 6.1 (Trains becoming available)**.** To illustrate how the setup with dummies works, we will look at the following scenario. Given that there are four inbound trains (11001, 11002, 11003, 11004), one each day in the third shift. Assume that there are three shifts per day and that we get the exact information on the contents of an inbound train one shift in advance.

We know that there are three different outbound trains each day (21001, 21002, 21003), with each three different destination groups. If a car is part of the second destination group of train 21001, we say that that car has destination 2100102.

To create the dummy trains, we look at historical data. In Table 6.1, we show which destinations were once present in the inbound trains. In Table 6.2, the real cars are given.

Train	Destinations present at least once (Car, destination)					
11001	(1, 2100103)	(2, 2100102)	(3, 2100201)	(4, 2100203)		
11002	(5, 2100102)	(6, 2100101)	(7, 2100203)	(8, 2100301)		
11003	(9, 2100103)	(10, 2100102)	(11, 2100202)	(12, 2100303)	(13, 2100302)	(14, 2100301)
11004	(15, 2100103)	(16, 2100102)	(17, 2100101)	(18, 2100203)	(19, 2100202)	(20, 2100201)

Table 6.1: Historical data destinations from inbound trains.

Train	Destinations present at least once (Car, destination)					
11001	(21, 2100101)	(22, 2100101)	(23, 2100101)	(24, 2100203)		
11002	(25, 2100102)	(26, 2100102)	(27, 2100203)	(28, 2100203)		
11003	(29, 2100102)	(30, 2100102)	(31, 2100102)	(32, 2100102)	(33, 2100102)	(34, 2100102)
11004	(35, 2100101)	(36, 2100203)	(37, 2100203)	(38, 2100203)	(39, 2100203)	(40, 2100203)

Table 6.2: Real destinations from inbound trains.

At the start of the simulation, we will create a solution based on the information from Table 6.1, because we do not yet know the content of each of the different inbound trains. Assume we find the solution as given in Table 6.3.

Train	Destinations present at least once (Car, `schemeValue`)					
11001	(1, 2)	(2, 1)	(3, 0)	(4, 0)		
11002	(5, 1)	(6, 0)	(7, 0)	(8, 4)		
11003	(9, 3)	(10, 2)	(11, 0)	(12, 2)	(13, 1)	(14, 0)
11004	(15, 4)	(16, 2)	(17, 1)	(18, 2)	(19, 1)	(20, 0)

Table 6.3: Initial solution for Example 6.1.

At the second shift of the first day, we learn that Train 11001 contains four cars, three with destination 2100101 and one with destination 2100203. When we look at the initial solution, we find that Cars 3 and 4 have the same destinations. So, since Cars 21, 22, and 23 have the same destination as Car 3, as we 'transfer' the solution from Car 3 to those three cars. We do the same for Cars 4 and 24. This way, we get the updated solution as given in Table 6.4.

Train	Destinations present at least once (Car, `schemeValue`)					
11001	(21, 0)	(22, 0)	(23, 0)	(24, 0)		
11002	(5, 1)	(6, 0)	(7, 0)	(8, 4)		
11003	(9, 3)	(10, 2)	(11, 0)	(12, 2)	(13, 1)	(14, 0)
11004	(15, 4)	(16, 2)	(17, 1)	(18, 2)	(19, 1)	(20, 0)

Table 6.4: Updated solution for Example 6.1 after information on Train 11001 became available.

By the time we get to the final day and all information became available, we get the solution as given in Table 6.5. Remark that in this example, we did not re-optimize the solution, but just transferred the solution from the dummy cars to the real cars. In practice, it can be advisable to check if the solution can be improved based on the latest information.

Train	Destinations present at least once (Car, `schemeValue`)					
11001	(21, 0)	(22, 0)	(23, 0)	(24, 0)		
11002	(25, 1)	(26, 1)	(27, 0)	(28, 0)		
11003	(29, 2)	(30, 2)	(31, 2)	(32, 2)	(33, 2)	(34, 2)
11004	(35, 1)	(36, 2)	(37, 2)	(38, 2)	(39, 2)	(40, 2)

Table 6.5: Final solution for Example 6.1.

◊

6.3.2. Crew duties

Next to checking the feasibility and possibly recovering solutions, the dynamic simulator will also create an overview of all crew duties. At each time step, we will check which events take place and where. For each of these events, we will add an appropriate crew duty (e.g. classification preparations or transfers). At the end of the simulation, this will result in a neat overview of the events at each time step and how many 'crew minutes' are required. Based on this information, an assignment can be made; which staff member should do what task. This assignment is beyond the scope of this thesis.

6.3.3. Recovering | Retraining solutions

Just like the static simulator, the dynamic simulator can recover solutions when the scenario changes. It will do so in a similar manner with one addition. As we have mentioned before at the scenario generation and working of the algorithms, we can 'fix' part of the instance and solution if the events took place in the past. When a scenario is created from within the dynamic simulator, all events that took place before or at the time step at which the simulator is, cannot be changed.

6.3.4. Robustness Analysis

Comparatively to the static simulator, we can check the feasibility of the solution for different scenarios. When a new analysis is started, there is also an option to set how often the simulation should be paused to generate a new scenario and (when needed) recover the solution. So compared to the static simulator which creates a single scenario at the beginning of the planning horizon, the dynamic simulator can perform multiple perturbations, even during a simulation. For example, consider a solution of ten time steps. If we specify that we would like to pause the solution four times, these moments will be chosen randomly, so this can be (1 2 7 9), (2 4 6 8), or any other sequence.

7 Results

In this chapter, we will show how our solution approaches perform. First, we will discuss the instances we have used for our experiments in Section 7.1. In Section 7.2, we outline how the solution approaches perform compared to each other in the case where all information is present at the beginning of the planning horizon. Then, in Section 7.3, we consider the case where not all information is present. We use two separate ways to create dummy solutions based on historical data and look into the different ways we can reoptimize the planning as information becomes available. Lastly, in Section 7.4, we see how well the solutions can be recovered for various scenarios.

Hardware and software: All experiments were run on a Windows 10 Desktop PC with an AMD Ryzon 3700X processor (3.6GHz) and 32 GB RAM. We have used Gurobi version 9.1.2 under an academic license to find solutions using the DF and FAF.

7.1. Instances

To create the instances for our experiments, we looked at the historical data of August, September, and October of 2021. We accumulated all cars that passed over the hump in those months and of which arrival | departure train they were a part. For the 'week'-instances, we combined all cars that arrived during the week (Sunday - Saturday).

We found that it happens regularly that cars depart on trains that according to the timetable don't go via the destination of the car. In that situation, the train will make a detour | extra stop on its way. In all of these cases, we combined all cars that are part of a destination that isn't visited according to the schedule in one destination group and added this destination group at the end of the train.

Note that in practice, some clients can receive a limited amount of cars per 'delivery' due to capacity constraints on the client-side. For the instances in this thesis, we have not taken these constraints into account.

As we mentioned with the objective function, we will minimize the number of *carrolls*. Regarding the number of classification tracks and reclassifications, we will limit these as part of the instance. We will allow nine reclassifications per day, split evenly among the three shifts each day. In our experiments, we will see how the number of classification tracks impacts the quality of the solutions and we will therefore experiment with different numbers of classification tracks. We remark that the number of arrival (14) and departure (8) tracks remains constant.

The specifics of the instances we have used for our experiments are given in Table 7.1.

Name	Based on data from	No. Cars	No. Inbound Trains	No. Outbound Trains
wk1	Aug. 1 - Aug. 7	1878	118	172
wk2	Aug. 8 - Aug. 14	2041	127	177
wk3	Aug. 15 - Aug. 21	1866	118	177
wk4	Aug. 22 - Aug. 28	1813	115	179
wk5	Aug. 29 - Sept. 4	1426	98	147
wk6	Sept. 5 - Sept. 11	1699	113	162
wk7	Sept. 12 - Sept. 18	1780	105	179
wk8	Sept. 19 - Sept. 25	1899	122	181

Table 7.1: Characteristics of instances for Base experiments.

7.1.1. Dummy Instances

To create the dummy solutions (i.e. those with dummy cars), we would like to know for which outbound trains there are cars for in an incoming train. Instead of adding a car for *every* destination in an inbound train, we looked at the realization.

We used an overview of all cars that were rolled over the hump in August, September, and October of 2021. For each of these cars, we know the inbound and outbound train - more specifically, the destination group of the given car. This way, we could make a co-occurrence matrix of inbound trains and destination groups. Then, if any destination group was at least once part of an inbound train, we would add a dummy car for that destination group to the inbound train.

In total, we have a set of 226 possible inbound trains with each on average 13.1 cars (total of 2957 cars). Notice that these numbers are higher than the number of trains and cars that will generally be part of an instance (respectively 100 - 125 and 1800-2000).

7.2. Complete Information

One of the first things we are interested in is how well our solution approaches perform if they have all information (i.e. which cars are in which trains) at the beginning of the planning horizon. In total, we will use five solution approaches, the Direct Formulation (DF), Flexible Arrival Formulation (FAF), and Simulated Annealing (SA). Both DF and SA can be combined with the Arrival Order Heuristic (AOH). For all solution approaches, we will use $h = 63$ which translates to 3 reclassifications per shift.

Both the DF and FAF will be solved using Gurobi with a time limit of $900s$.[1] If there is a feasible solution found within this time frame, this will be the solution considered. If not, we say that no feasible solution has been found. For the SA algorithm, we will use the following configuration:

$$T_0 = 0.03 \cdot \text{change}_{\text{avg}} \qquad T_k = \frac{T_{k-1}}{1 + 5\log(1+k)} \qquad M = 3 \cdot h \qquad \text{no. iterations } = 200.$$

Furthermore, we will use three different initial solutions and reserve seven tracks for the reclassification parts.

In Figures 7.1 through 7.8, we present the results for the experiments with the different solution approaches. Only the data points for feasible[2] solutions are given.

In Table 7.2, we present a more elaborate overview with also the number of reclassifications and computation time. Notice that the computation time of the IP-based algorithms was limited to $900s$ and that the 'extra' time is taken up by the processing of the solutions. If no feasible solution was found (either within the given timeframe (DF, FAF) or after completion of the algorithm (SA)), we have noted 'NFSF'.

We find that the DF (+ AOH) can find a feasible solution in all 48 cases. Next, the FAF can find a feasible solution within the given timeframe for only six instances. With fewer than 32 tracks, the FAF is not able to find a solution for more than half of the instances.

The SA (+ AOH) solution approach faces similar issues. While this approach can find good solutions quickly compared to the IP-based approaches, the algorithm struggles to find feasible solutions with fewer tracks. Especially satisfying the maximum number of classification tracks is hard in these cases.

We are also interested in how the solution quality is influenced by the number of classification tracks. Therefore, we have given the solution quality relative to the case with 43 classification tracks in Figure 7.9. Here, we find that up to 28 classification tracks, the solution approaches can find solutions of

[1] Note that if the time limit is reached and no optimal solution has been found, what could be considered 'harder' instances might have a better incumbent solution compared to 'easier' instances. Furthermore, variations in computer load might have small effects on the overall results.

[2] Some solutions had a handful of violations caused by floating-point errors (DF and FAF) or classification track capacity overruns (SA). We have left those in because the violations have little to no impact on the solution quality.

similar quality when compared to the case with 43 tracks. Although, we need to remark here that in these averages, only the feasible solutions are considered. Some approaches might not find a feasible solution (see Figures 7.1 - 7.8). This is however no issue for the DF (+ AOH).

In Figure 7.10, we show the performance of the different solution approaches relative to each other. In each of the cases, we have used DF as the base case. Here we find that, when finding a solution in time, FAF consistently performs about 45% better compared to DF. Also, we find that using the AOH in combination with the DF will result in about 4% better solutions. For the SA algorithm, the improvement by using the AOH is about 7%. When we compare DF to SA, we find that SA performs generally 16% worse than DF, but is way faster.

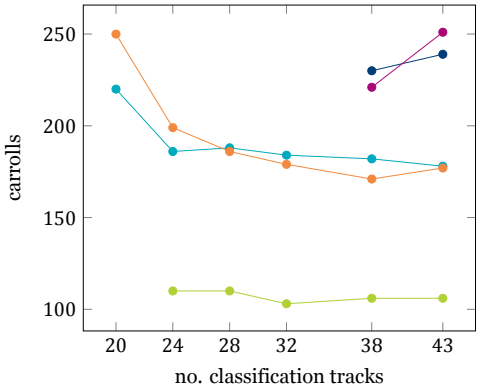


Figure 7.1: Comparison of the different solution approaches for instance `wk1`.

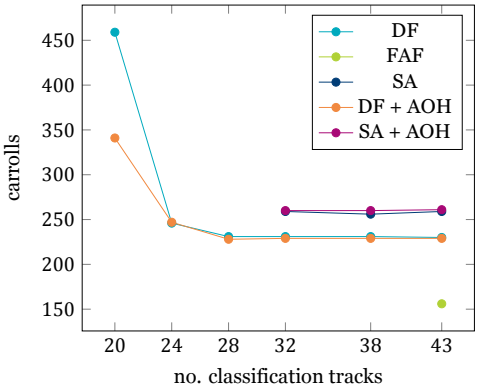


Figure 7.2: Comparison of the different solution approaches for instance `wk2`.

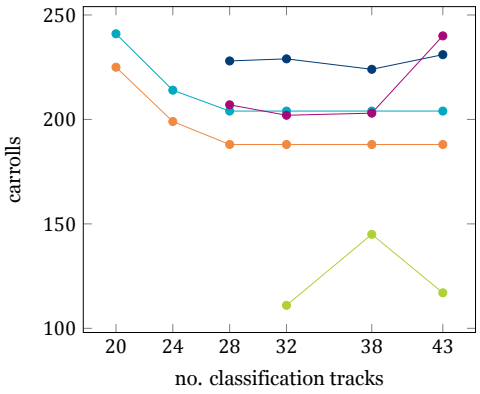


Figure 7.3: Comparison of the different solution approaches for instance `wk3`.

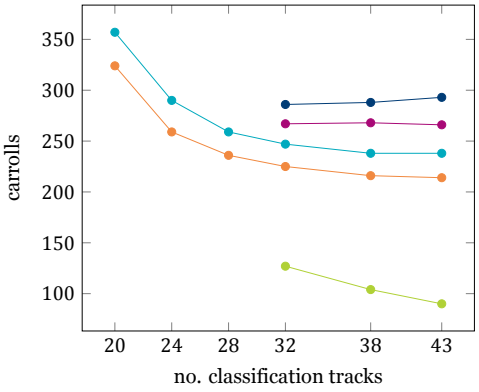


Figure 7.4: Comparison of the different solution approaches for instance `wk4`.

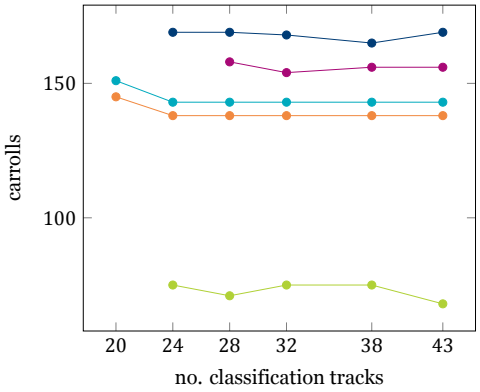


Figure 7.5: Comparison of the different solution approaches for `wk5`.

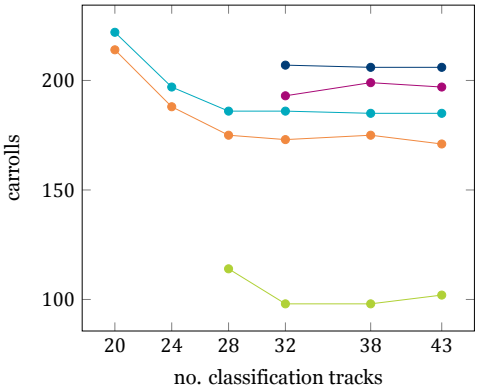


Figure 7.6: Comparison of the different solution approaches for `wk6`.

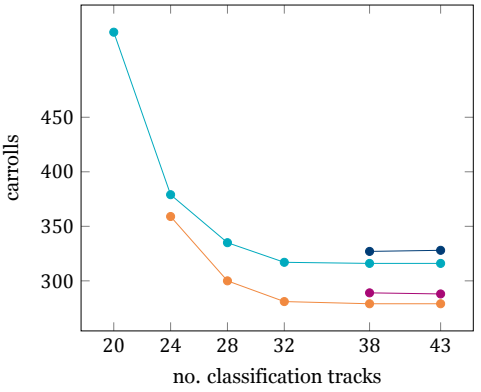


Figure 7.7: Comparison of the different solution approaches for `wk7`.

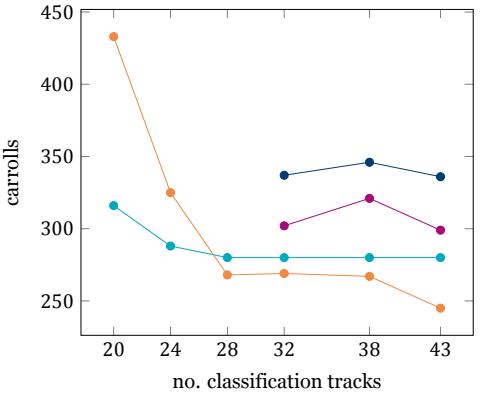


Figure 7.8: Comparison of the different solution approaches for `wk8`.

	Instance			DF			FAF			SA			DF + AOH			SA + AOH		
	Start date	End date	Tracks	carrolls	Reclass.	Time	carrolls	Reclass.	Time	carrolls	Reclass.	Time	carrolls	Reclass.	Time	carrolls	Reclass.	Time
wk1	1-8-2021	7-8-2021	43	178	28	226.00	106	19	941.00	239	30	18.36	177	33	920.40	251	33	18.70
			38	182	32	465.70	106	23	940.00	230	34	18.40	171	32	937.20	221	32	18.58
			32	184	26	921.00	103	23	940.00		NFSF		179	20	920.20		NFSF	
			28	188	30	404.00	110	20	941.00		NFSF		186	25	920.60		NFSF	
			24	186	23	729.00	110	16	940.00		NFSF		199	27	921.30		NFSF	
			20	220	17	921.00		NFSF			NFSF		250	14	920.00		NFSF	
wk2	8-8-2021	14-8-2021	43	230	34	922.09	156	24	945.98	259	31	25.00	229	34	922.67	261	28	24.92
			38	231	36	922.93		NFSF		256	35	25.01	229	37	922.66	260	33	24.67
			32	231	29	922.87		NFSF		259	35	24.51	229	29	922.13	260	30	24.52
			28	231	25	922.64		NFSF		259	35	24.75	228	25	922.37	257	29	23.90
			24	246	24	923.35		NFSF			NFSF		247	20	922.70		NFSF	
			20	459	24	923.03		NFSF			NFSF		341	24	922.00		NFSF	
wk3	15-8-2021	21-8-2021	43	204	22	920.00	117	18	940.70	231	26	24.09	188	19	919.80	240	26	23.42
			38	204	23	921.00	145	24	941.00	224	22	24.31	188	21	920.80	203	22	23.38
			32	204	23	921.00	111	15	941.00	229	27	23.68	188	21	920.30	202	23	23.00
			28	204	22	920.80		NFSF		228	24	24.80	188	19	920.23	207	23	23.45
			24	214	15	921.40		NFSF			NFSF		199	14	920.47		NFSF	
			20	241	19	921.06		NFSF			NFSF		225	15	920.07		NFSF	
wk4	22-8-2021	28-8-2021	43	238	26	919.55	90	19	935.83	293	30	25.49	214	28	232.83	266	26	25.42
			38	238	30	920.17	104	19	938.01	288	29	25.18	216	26	300.46	268	26	25.07
			32	247	24	920.11	127	19	938.63	286	29	25.68	225	25	920.04	267	29	25.59
			28	259	26	919.38		NFSF		285	27	25.39	236	29	919.53	262	22	25.40
			24	290	22	920.26		NFSF			NFSF		259	20	920.10		NFSF	
			20	357	25	920.21		NFSF			NFSF		324	22	920.00		NFSF	
wk5	29-8-2021	4-9-2021	43	143	18	513.40	68	15	928.00	169	23	11.20	138	17	215.90	156	21	10.90
			38	143	19	543.00	75	16	928.00	165	21	11.00	138	24	916.00	156	19	10.90
			32	143	17	917.00	75	13	929.00	168	20	12.16	138	19	915.90	154	20	12.36
			28	143	19	916.40	71	14	928.20	169	20	12.30	138	21	915.70	158	24	12.54
			24	143	19	916.40	75	15	928.10	169	17	12.30	138	22	915.70		NFSF	
			20	151	15	916.40		NFSF			NFSF		145	15	915.70		NFSF	
wk6	5-9-2021	11-9-2021	43	185	23	919.73	102	17	935.65	206	25	12.31	171	23	920.41	197	31	12.29
			38	185	29	918.88	98	19	936.21	206	27	12.46	175	23	919.89	199	26	12.40
			32	186	24	919.49	98	15	936.58	207	27	12.43	173	22	918.80	193	26	12.43
			28	186	14	919.33	114	17	936.46	213	26	12.29	175	16	918.61	198	29	12.24
			24	197	17	919.35	99	12	936.21		NFSF		188	14	918.00		NFSF	
			20	222	16	919.72	121	15	936.21		NFSF		214	16	918.00		NFSF	
wk7	12-9-2021	18-9-2021	43	316	30	921.10		NFSF		328	24	31.16	279	33	919.20	288	26	32.57
			38	316	27	919.80		NFSF		327	26	31.85	279	30	916.90	289	25	32.26
			32	317	27	919.90		NFSF			NFSF		281	26	919.40		NFSF	
			28	335	26	919.40		NFSF			NFSF		300	22	919.50		NFSF	
			24	379	25	919.40		NFSF			NFSF		359	24	919.40		NFSF	
			20	528	30	918.80		NFSF			NFSF		1034	43	919.10		NFSF	
wk8	19-9-2021	25-9-2021	43	280	36	921.05		NFSF		336	30	24.97	245	34	920.40	299	28	24.39
			38	280	36	920.88		NFSF		346	32	25.12	267	30	920.79	321	30	24.99
			32	280	36	920.97		NFSF		337	36	25.14	269	33	920.23	302	29	24.39
			28	280	25	921.02		NFSF		346	35	25.09	268	24	921.34	318	32	24.97
			24	288	25	921.23		NFSF			NFSF		325	21	921.50		NFSF	
			20	316	19	920.96		NFSF			NFSF		433	24	920.70		NFSF	

Table 7.2: Overview of the performance of the different solution approaches in the case where all information is known at the beginning of the planning horizon.

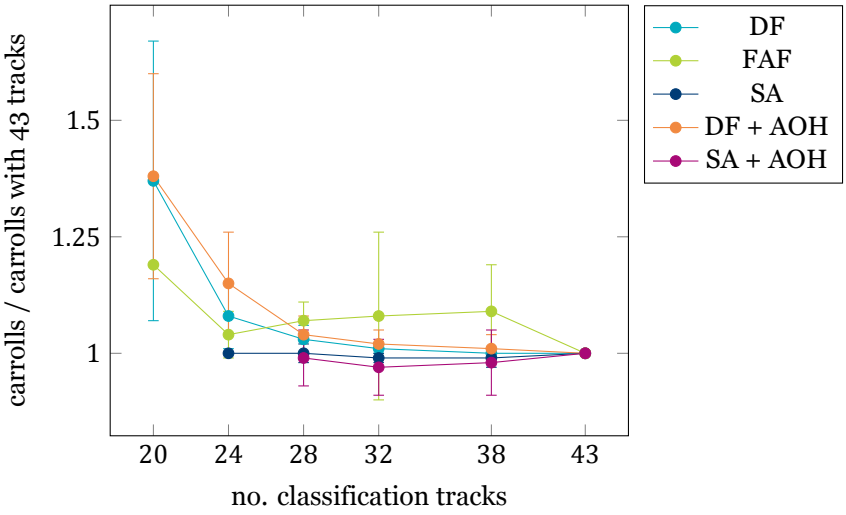


Figure 7.9: Comparison of the performance of the different solution methods relative to the case with 43 tracks. One standard deviation is given. If no feasible solution was found, the instance was left out.

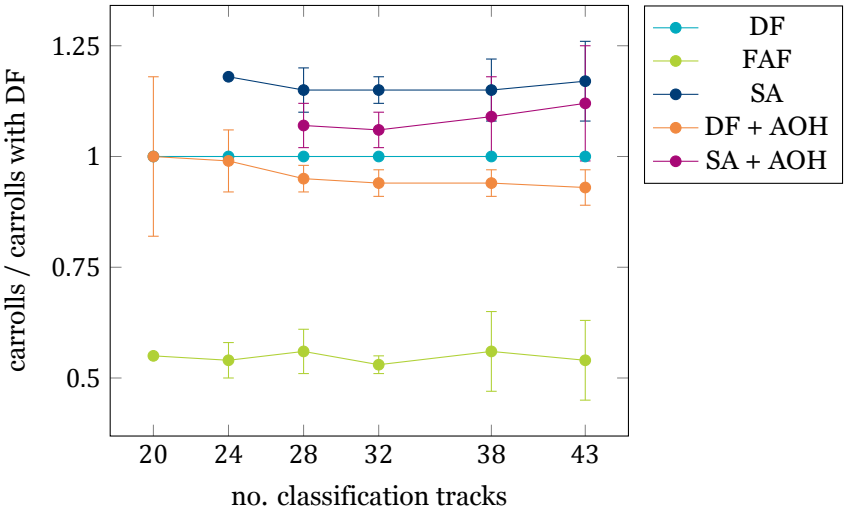


Figure 7.10: Comparison of the different solution methods relative to the DF solution method. One standard deviation is given. If no feasible solution was found, the instance was left out.

7.3. Dummy solutions

As we have mentioned before, at the start of the planning horizon, there is only little known about the cars that will arrive during the week. To that end, we create dummy solutions based on the cars that arrived in the months August, September, and October of 2021 (as explained in Section 6.3.1). We will create both specific dummy solutions and universal dummy solutions.

With the specific dummy solutions, the solution is created with only the arrival trains for that week. In the universal dummy solutions, all possible arrival trains are used in the instance. While the trains that arrive each week are generally the same, there are some differences from week to week. With the specific dummy solution, we assume that we know which subset of trains will arrive in the given week. With the universal dummy solution, we create a solution for all possible arrival trains. So, the instance is larger and harder to solve in the universal case.

When used in practice, this means that each week a new specific dummy solution should be found while a universal dummy solution can found monthly | quarterly and used for that period. The latter is how DB currently operates. They create a week planning each quarter and use this for each week in the planning period.

By making a using both specific and universal dummy solutions, we try to find out if there is a difference in performance.

When we have a dummy solution, we will use a simulation to test the quality. During the simulation, we replace these cars with the real cars and reoptimize the solution. In total, we have tested four reoptimization approaches. The first **Direct** one is the most basic (i.e. no reoptimization). Here, we simply update the solution by replacing the dummy cars with the real ones (see Example 6.1). The next, **II**, uses the iterative improvement algorithm[3]. The same goes for **SA** which uses the simulated annealing algorithm to reoptimize the solution based on the latest available data. The configuration is the same as given in Section 7.2. The last approach, **DF**, uses Gurobi to solve the DF with a time limit of $600s$. Lastly, we refer to the solutions found using complete information at the beginning of the planning horizon by **CI** (results are taken from Section 7.2).

7.3.1. Specific Dummy Solutions

First the specific dummy solutions. These are made based on the inbound trains for a given week. To find the Specific Dummy solutions, we have used the DF with a limited computation time of $7200s$. In Figures 7.11, 7.12, 7.13, 7.14, and 7.15, we show the number of reclassifications and carrolls for both instance `wk2`, `wk3`, `wk4`, `wk5`, and `wk7`. In total, we ran the experiment for three cases, one with 32, one with 38, and one with 43 classification tracks. In Figure 7.16, we show how the algorithms perform relative to the Direct method (no reoptimization).

Reoptimization using the DF approach performs best by both metrics. Both the carrolls and the number of reclassifications are lowered using this approach. However, when comparing the algorithms, it is important to realize that while the DF outperforms the II and SA approaches, it does take considerably more time to do so.

We note that SA has difficulty finding feasible solutions. Especially in the cases with 32 tracks, which aligns with the results from the experiments with complete information. However, when the algorithm produces feasible solutions, they are an improvement over II and the Direct method.

When we look at how well the algorithms perform compared to the CI case, we find that the DF finds solutions with a similar amount of reclassifications, but the number of carrolls is higher. Concretely, this means that there are more cars per reclassification part.

[3]Which is the same as SA but only accepts neighbors that do improve the solution quality.

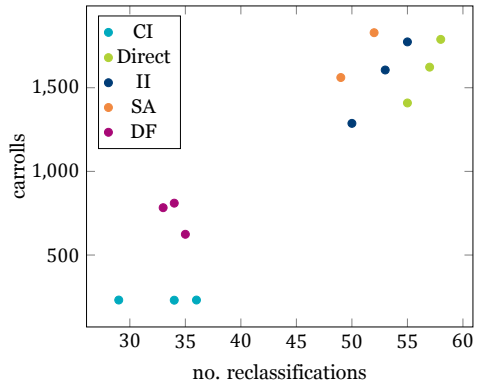


Figure 7.11: Comparison of reoptimization methods of dummy solutions for instance `wk2` with 32, 38, and 43 classification tracks.

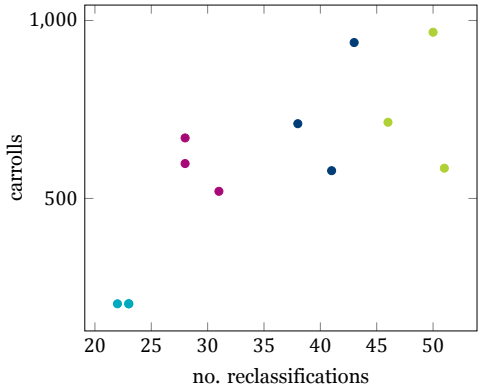


Figure 7.12: Comparison of reoptimization methods of dummy solutions for instance `wk3` with 32, 38, and 43 classification tracks.

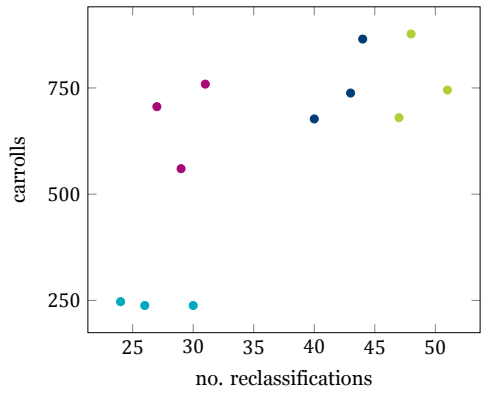


Figure 7.13: Comparison of reoptimization methods of dummy solutions for instance `wk4` with 32, 38, and 43 classification tracks.

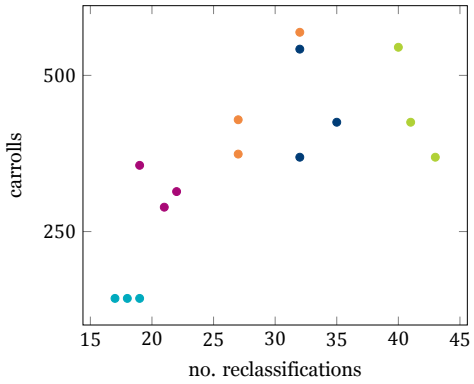


Figure 7.14: Comparison of reoptimization methods of dummy solutions for instance `wk5` with 32, 38, and 43 classification tracks.

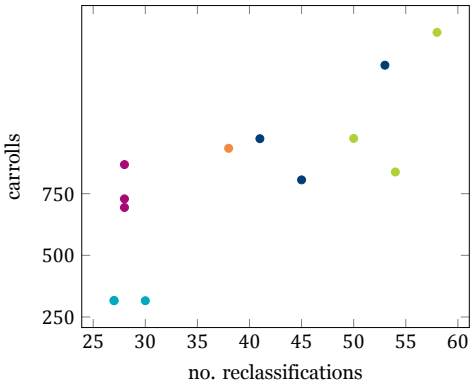


Figure 7.15: Comparison of reoptimization methods of dummy solutions for instance `wk7` with 32, 38, and 43 classification tracks.

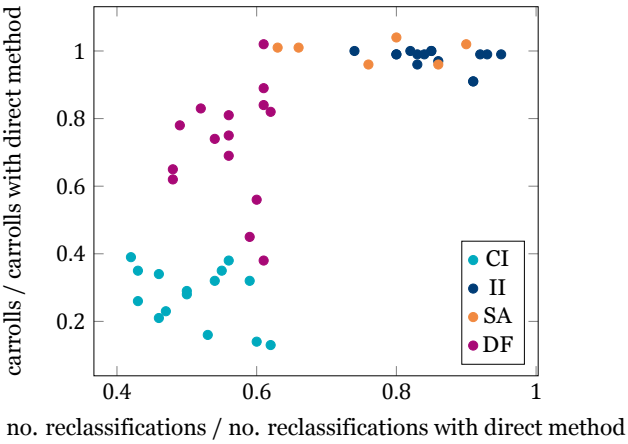


Figure 7.16: Performance of the different reoptimization techniques relative to the direct method. The instances are those from Figures 7.11, 7.12, 7.13, 7.14, and 7.15.

In Figures 7.17 and 7.18, we present the number of classification tracks needed at each time step for the `wk2` and `wk4` instance with a capacity of 38 tracks. We show the required number of classification tracks for both the dummy solution as well as the reoptimized solution (DF). We find that the number of classification tracks needed by the reoptimized solution is in most cases (well) below the number of tracks needed by the dummy solution.

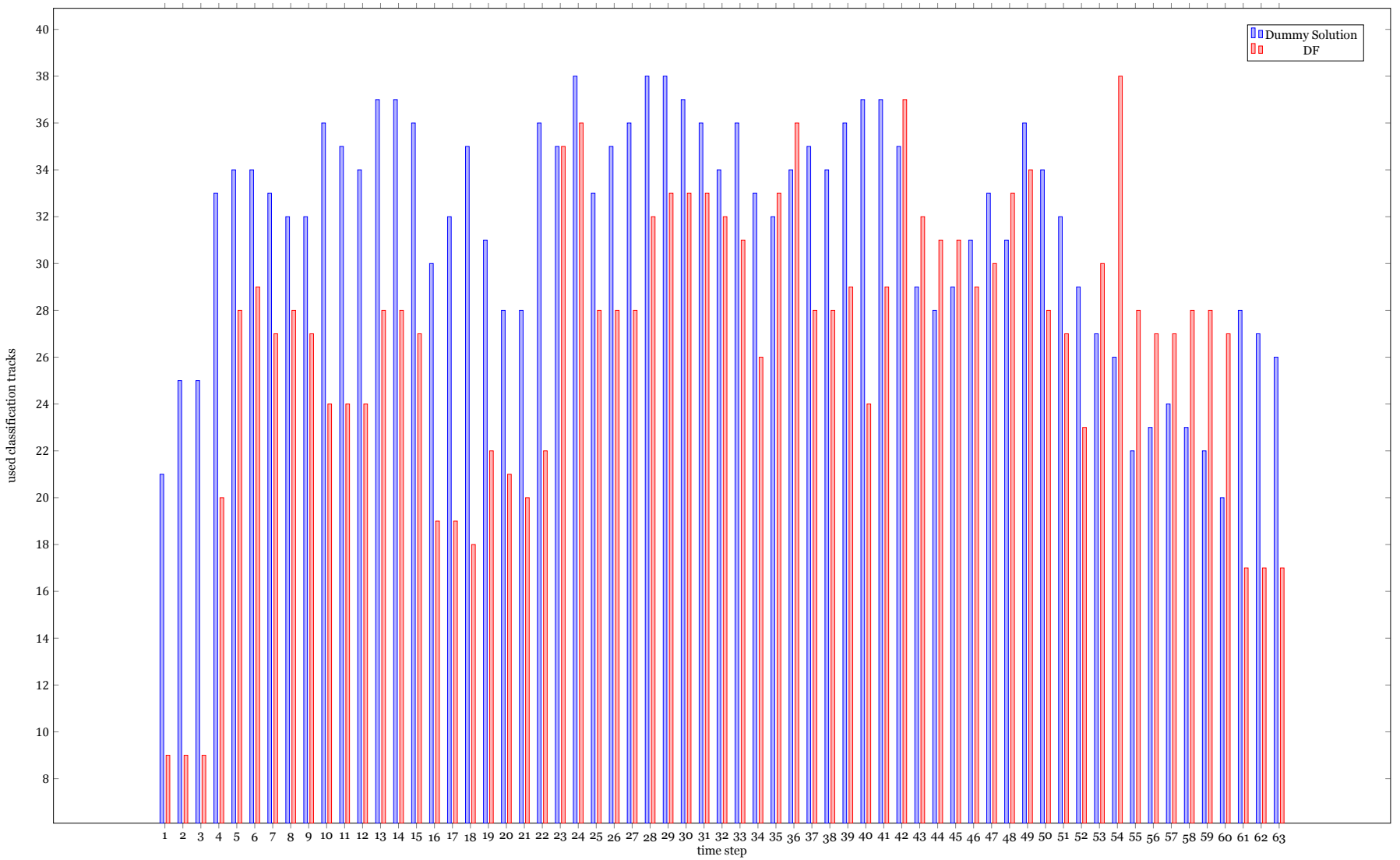


Figure 7.17: Used number of classification tracks per time step. Comparison between specific dummy solution and reoptimized solution using the DF for the instance `wk2` with a capacity of 38 classification tracks.

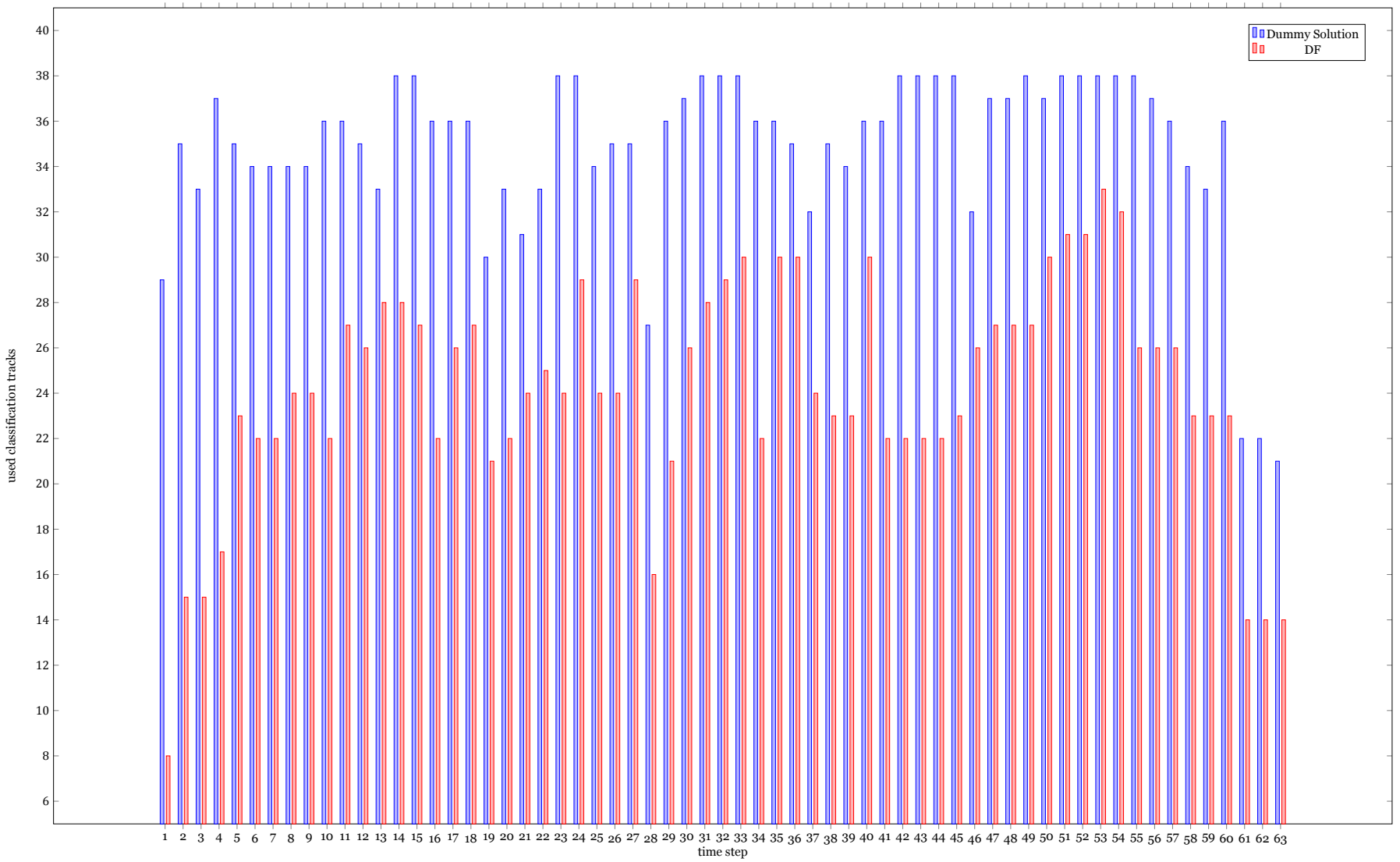


Figure 7.18: Used number of classification tracks per time step. Comparison between specific dummy solution and reoptimized solution using the DF for the instance `wk4` with a capacity of 38 classification tracks.

7.3.2. Universal Dummy Solutions

As mentioned at the beginning of this chapter, we also look at universal dummy solutions. These do resemble the quarterly planning currently used by DB Cargo the most. This because they are made without prior knowledge of the trains that will arrive in any given week. When we start the simulation, we remove the trains that are not part of the specific instance and then continue as we would have in the specific dummy solution case. To find the Universal Dummy Solutions, we have used the DF with a limited computation time of $10800s$.

In Figures 7.19, 7.20, 7.21, 7.22, and 7.23, we show the performance of the different reoptimization techniques in the same manner as with the specific dummy solutions. In Figure 7.24, we show the performance of the different techniques relative to the Direct method. We find that the II approach is an improvement when compared to the direct method, both in carrolls and the number of reclassifications. Furthermore, we find that the DF provides an even greater improvement and performs similar to CI for the number of reclassifications, but performs less on the carrol-side when compared to the 'optimal' CI case. Again, we find that the SA method struggles to produce feasible solutions, unlike the II method which performs steadily.

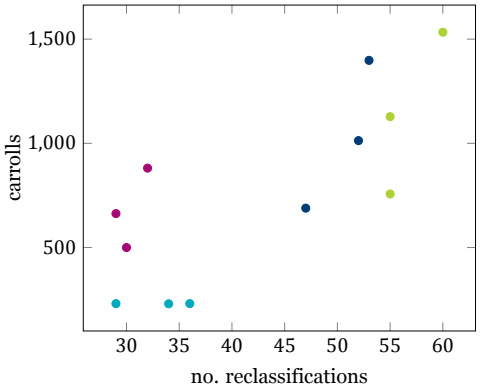


Figure 7.19: Comparison of reoptimization methods of universal dummy solutions for instance `wk2` with 32, 38, and 43 classification tracks.

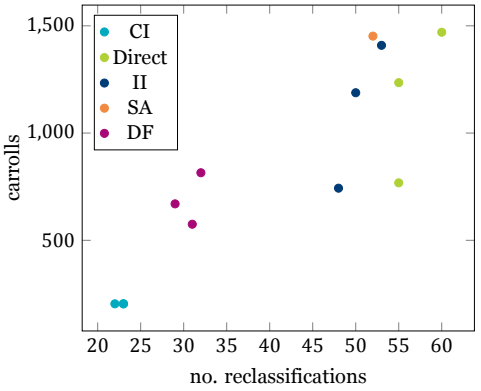


Figure 7.20: Comparison of reoptimization methods of universal dummy solutions for instance `wk3` with 32, 38, and 43 classification tracks.

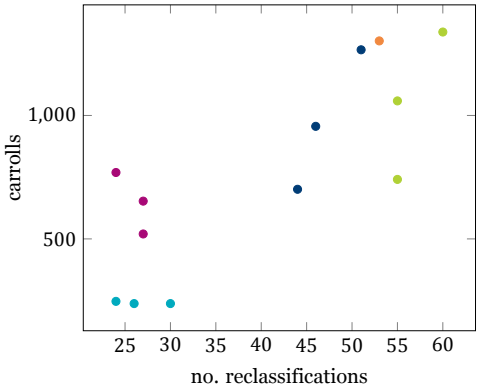


Figure 7.21: Comparison of reoptimization methods of universal dummy solutions for instance `wk4` with 32, 38, and 43 classification tracks.

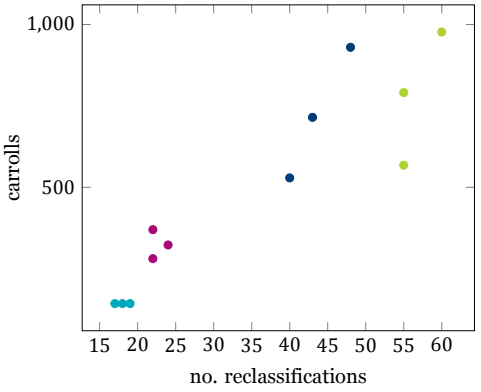


Figure 7.22: Comparison of reoptimization methods of universal dummy solutions for instance `wk5` with 32, 38, and 43 classification tracks.

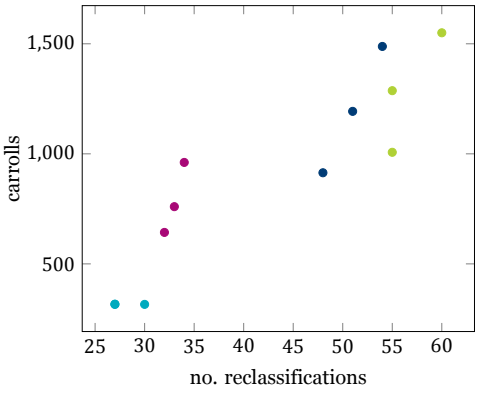


Figure 7.23: Comparison of reoptimization methods of universal dummy solutions for instance `wk7` with 32, 38, and 43 classification tracks.

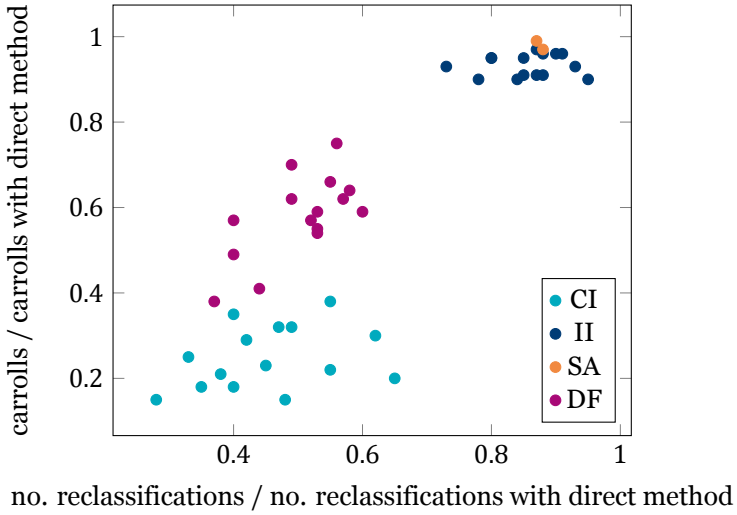


Figure 7.24: Performance of the different reoptimization techniques relative to the direct method.

7.3.3. Comparison Specific and Universal Dummy solutions

In Figure 7.25, we compare the results of the Specific (S) and Universal (U) dummy solutions. All results are scaled to the specific dummy solution with direct replacement. When we look at the quality of the universal dummy solution with the direct method, we find that in general, these solutions are worse than their specific counterparts.

When we reoptimize using the DF, we find that the end results do line up. However, this is not the case with reoptimization using the II and SA methods. Here, we find that in most cases, the end result is worse for the universal dummy solutions when compared to the specific dummy solutions.

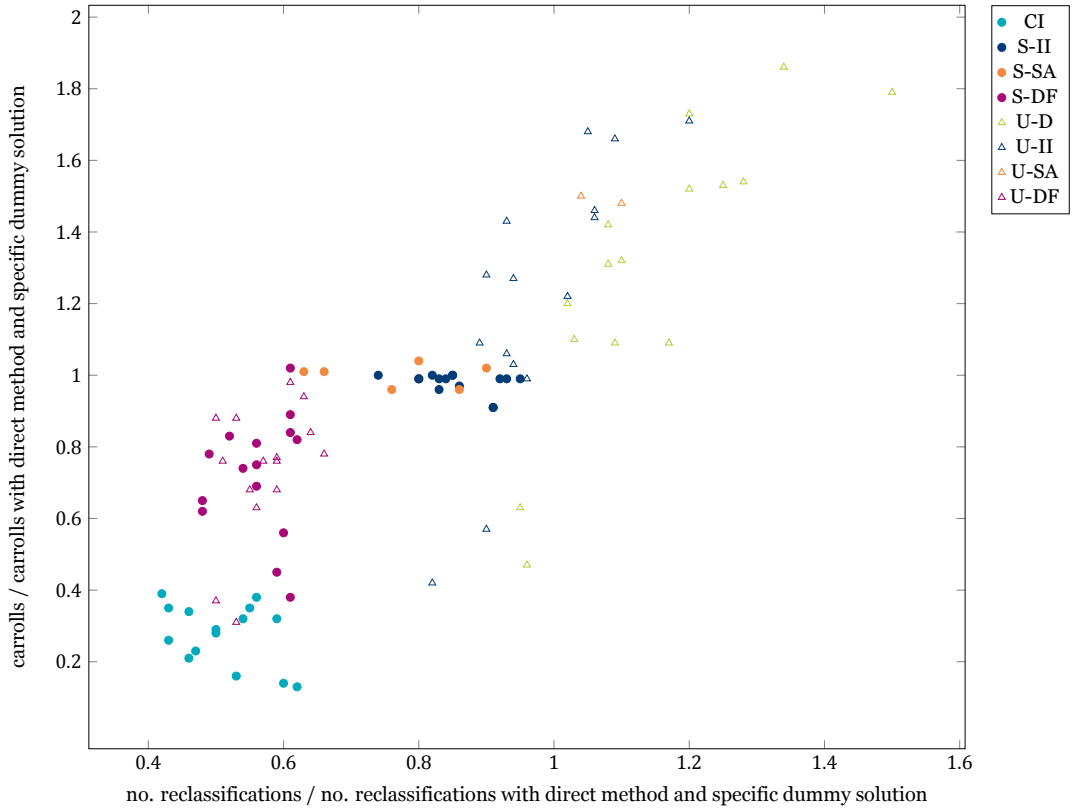


Figure 7.25: Performance of the different reoptimization techniques relative to the direct method with the specific dummy solution.

7.4. Robustness

Lastly, we looked at how well solutions can be recovered in the case of perturbations. One of the key questions here is to find out whether the 'fast' methods (II and SA) can recover solutions. For each experiment, we use the DF to check if a feasible solution exists for the perturbed instance. Then, we test if II and SA can recover the solution 'fast' (i.e. about ten seconds).

We use the same configuration as for the Complete Information case (see Section 7.2). The solutions found in those experiments will also form the start point of the experiments. In total, we will use four different configurations for the scenario generator (as given in Table 7.3). In each experiment, we will generate twenty scenarios based on these configurations.

Scenario	Perturbations (configuration of the scenario generator)
1	Cancel arriving trains with $p = 0.1$, Change arrival time with $p = 0.05$, `maxDown` $= 1$, and `maxUp` $= 1$, and Change departure time with $p = 0.05$, `maxDown` $= 0$, and `maxUp` $= 1$.
2	Change arrival order with $p = 0.15$.
3	Change arrival time with $p = 0.15$, `maxDown` $= 0$, and `maxUp` $= 2$ and Change departure time with $p = 0.05$, `maxDown` $= 0$, and `maxUp` $= 1$.
4	Remove arrival | classification | departure tracks with $p = 0.05$.

Table 7.3: Scenarios for robustness analysis.

7.4.1. Static

First, we consider scenarios generated by the static simulator. Remark that this simulator can change everything and that when we reoptimize, we do not 'fix' past actions as there aren't any. The results are given in Table 7.4.

We find that although II and SA do not find a solution fast in all cases, for most scenarios, these methods suffice to quickly recover a solution. Remark that if these techniques can't find a solution, DF will be able to find one, but this will take longer (a couple of minutes).

When we compare II and SA, we find that the techniques do not differ much with respect to the number of recovered solutions. However, when we look at the difference with the original solution, we find that II performs way better than SA in this area. This means that with II, we make fewer changes to the reclassification schedule which reduces complexity in the day-to-day operations.

Scenario	Instance	No. Issues	Rec. II	Diff. II	Rec. SA	Diff. SA
1	wk2	43.1	17	24.8	15	284.6
	wk3	38.3	18	22.4	19	137.8
	wk5	31.3	19	17.1	15	85.3
2	wk2	3.5	18	0.9	17	278.4
	wk3	7.1	11	1.5	14	127.2
	wk5	11.7	14	1.7	12	60.5
3	wk2	22.2	5	1.0	8	287.0
	wk3	20.7	6	2.0	8	120.0
	wk5	16.3	11	1.1	11	59.8
4	wk2	12.4	9	0.0	13	279.6
	wk3	11.0	15	0.0	18	121.3
	wk5	8.8	19	0.0	17	62.9

Table 7.4: Results of robustness analysis with the static simulator. Each time, twenty scenarios were generated, in all cases, the scenarios were so that a feasible solution exists. We use the abbreviations Rec. for Recovered (i.e. a feasible solution was found) and Diff. for the difference of the solution with the original solution.

7.4.2. Dynamic

Next to the robustness experiments with the static simulator, we also used the dynamic simulator to check how well the solutions can be recovered using the proposed solution approaches if the changes are made during the planning horizon. In Table 7.5, we present the results of these experiments.

During each of the simulations, we have interrupted the simulation at three randomly selected time steps. At these time steps, we create a scenario and find how well the solution could be recovered using the II and SA algorithms. After each interruption, we checked if a feasible solution existed. If this is the case, we would continue to use the solution found by the DF to check if a feasible solution existed for the rest of the simulation, even if II and/or SA returned a feasible solution.

We find, in line with the results from the static simulator, that II and SA are indeed able to recover the solution in a significant portion of the scenarios. One note, for the first scenario and the `wk2` instance, we found that SA was able to recover the solution more often at the second interruption than at the first one. This is possible since we use the solution found by the DF after each interruption.

Furthermore, we find that, just like in the case with the static simulator, the II algorithm performs better when we look at the difference between the original solution and the recovered solution.

Scenario	Instance	Feas. Poss			Rec. II			Diff. II			Rec. SA			Diff. SA		
1	wk2	20	12	3	14	11	3	14.3	29.8	30.0	8	9	3	189.5	158.8	116.7
	wk3	19	15	12	15	13	11	14.5	25.5	33.9	13	13	11	125.0	110.9	69.2
	wk5	20	13	9	18	11	9	6.6	9.5	8.4	16	11	9	28.3	23.5	12.7
2	wk2	20	11	6	15	10	6	0.1	0.2	0.3	13	8	2	15.8	17.5	23.0
	wk3	19	10	4	10	5	1	0.0	0.0	0.0	10	5	1	65.0	107.2	50.0
	wk5	20	11	6	18	9	6	0.1	0.1	0.2	14	10	4	30.0	13.1	7.8
3	wk2	16	4	1	2	0	0	0.0	NA	NA	2	0	0	155.0	NA	NA
	wk3	17	8	2	2	1	1	0.0	0.0	0.0	3	0	0	120.0	NA	NA
	wk5	20	4	0	7	1	0	0.1	0.0	NA	8	1	0	27.6	2.0	NA

Table 7.5: Results of robustness analysis with the dynamic simulator. For each experiment, we indicate the number of solutions with a feasible solution per 'interruption'. We use the abbreviations Rec. for Recovered (i.e. a feasible solution was found) and Diff. for the difference of the solution with the original solution.

8 Discussion

In this thesis, we have seen many solution approaches and in the previous chapter, we have seen how well they perform. But how should they be used in practice, what hurdles need to be taken, where are opportunities to improve the current solution approaches, and how should the results be interpreted?

In this final chapter, we will first answer the research questions as stated in the introduction. Then, we will go over some of the risks and opportunities related to the proposed solution approaches. Thereafter, we will give some recommendations on how the proposed solution approaches could be implemented in practice, what prerequisites there are, and how a successful roll-out could look like.

Finally, yet importantly, we acknowledge that there are still plenty of research opportunities in the area of shunting algorithms. To that end, we will present some pointers for future research.

1. *Is there a way to find optimal solutions for the shunting problem at Kijfhoek in reasonable time and how well do heuristics perform relative to these solutions, both measured in solution quality as well as computation time?*

Within the given time frame, the FAF was able to achieve the best results by a big margin. This indicates that using the flexibility of having arriving trains wait at the arrival tracks yields better solutions as this method performs 46% better when compared to the DF.

With that goal in mind, we have developed the AOH and our experiments show that using this preprocessing technique does result in an improvement of 7% (DF) and 4% (SA). One of the main differences between the AOH and the FAF concerning the arrival train order is that the FAF can have trains 'be overtaken' by inbound trains that arrive at different time steps. This is one of the limitations of the AOH which only looks at the order for trains that arrive at the same time step.

When we compare the SA and DF methods, we find that SA performs 17% worse when compared to DF. However, as we have indicated previously, SA is considerably faster as it takes less than half a minute and most often even less time to find the solutions compared to the fifteen minutes necessary for DF. This makes it more useful in practice when good solutions are needed fast.

2. *Can the solutions found by the algorithms be used in practice - possibly under some conditions?*

Using the dynamic simulator, we have been able to mimic the day-to-day operations in which planning decisions are made with incomplete information. We have shown that dummy solutions, either specific or universal can both be used to create a feasible planning and could be a good first step.

Furthermore, we have found that reoptimizing the dummy solutions will improve the quality. As presented in Figure 7.25, the universal dummy solutions are of lower quality. Nonetheless, they still are an improvement over the current solution approach.

3. *Is there a way to recover solutions after a perturbation occurred, either before or during the planning horizon?*

In our simulation study, we have run some experiments to try to determine how well the solutions can be recovered, especially by the faster II and SA methods. We find that for more than half of the scenarios, both methods were able to provide a feasible solution, both in the static and dynamic case.

Furthermore, we find that the II algorithm will find solutions closer to the original solution. This way, we have found that the solution approaches used to find solutions in general and to reoptimize can also be used for the recovery of solutions for which the instance is perturbed.

4. *What is the impact of a reduction of the number classification tracks on the quality of the solutions?*

In our experiments, we have tried to find feasible solutions for the different instances in cases with fewer tracks than currently available. In the case with complete information, we find that with few exceptions all tested solution approaches can find feasible solutions for the instances with 32 classification tracks of similar quality when compared to the solutions where 43 tracks could be used.

Then, with 24 and 28 classification tracks, we find that most solution approaches fail to find feasible solutions. However, the DF method still performs reasonably well when compared to the solutions found with 43 classification tracks (only an 8% increase in carrolls). However, when the number of tracks is lowered to 20 tracks, the solution quality deteriorates.

It appears that in the case with complete information, the lower bound on the number of classification tracks so that the solution quality isn't affected lies around $24 - 28$ tracks.

For the case with incomplete information, we used dummy solutions to resemble real-life operations as well as possible. With both the specific and universal dummy solutions, we have shown that it is possible to find feasible solutions for 32, 38, and 43 classification tracks that are of comparable quality.

This means that the number of classification tracks used by DB Cargo could be reduced to 32 in these scenarios. However, as we will discuss in the following, some risks could affect the results. On the other hand, we will also provide a list of opportunities or areas of interest which might improve the solution quality further.

Risks

As with all solution approaches, some risks might prevent a successful implementation or affect the quality of the solutions. Therefore, it is paramount to identify these risks and try to mitigate the impact before implementing the new solution approach into daily operations. Below, we will present an overview of the risks we have identified that might undermine a successful implementation.

→ In all of our experiments, we have made the assumption that the shunting yard is empty at the beginning of the experiments. In real-life, this is not the case. This might reduce the available capacity at the beginning of the planning horizon. However, we would like to point out that the experiments did show that there was excess capacity at the beginning. If it were to be the case that some tracks will be used for longer-term storage (> 3 days), capacity should be reduced for the instance and that track should be left out of consideration.

→ As mentioned in Section 7.1, when we created our test instances, we placed all cars that aren't part of an outbound train at the end of that train. In practice, these may be placed in a different position. While this seems unlikely to increase the complexity of the problem at hand, this is something to take into consideration before implementing our proposed solution approaches in practice.

→ We have done our experiments with a fixed number of classification tracks. This isn't the case in the day-to-day operations. Here, it might be that there is a temporary closure of a number of tracks for emergency repairs. This is something that we didn't take into account. For this thesis, we have assumed a fixed number of tracks. Please note that as we remark in Appendix B, we have kept this option open in the solver and future research might look at a dynamic number of tracks available.

→ For the 'dummy' trains, we used the historical data of up to three months before the actual instance. Remark that if the circumstances are changing quickly and radically, these dummy solutions could become less dependable.

→ For now, we have not put a cap on the number of cars that can be part of a reclassification part. This means that, in theory, they could be of infinite length. In practice, tracks have a finite length. To that end, some of the reclassification parts should be split, using extra classification tracks. Our experiments show that these kinds of violations occur infrequently. Meanwhile, the freedom added by not having a restriction on the number of cars on a single track does outweigh the costs at this point.

→ In our planning, we have focussed on the cars and arriving | departing trains. We haven't taken the crew and locomotive planning into account. To have a successful implementation, some attention in that area is necessary.

→ Integrating algorithms into the operations of DB Cargo requires both that the current IT systems can communicate with the algorithms as well as that the current operators should learn how to work will the new program. The latter might prove the most difficult. Applying algorithms at Kijfhoek will mean that the current process coordinators will need to change their way of working. Since the algorithms will just return a planning or list of actions without an elaborate explanation, it is not possible to discuss the results with 'the computer'. This might result in resistance which could reduce the benefits of the algorithms.

Opportunities

Above, we have outlined the risks that might prevent a successful deployment of the proposed solution approaches in real-life. Meanwhile, there are plenty of opportunities that could improve the solutions in practice. Note that later on, we will give an explicit survey of areas of future research.

→ For the dummy solutions, we have identified some risks, but in this area, there are also plenty of opportunities. Currently, we have added a car for each destination group that once was part of an inbound train. Our analysis did also show that many destination groups were part of an inbound train only once. A more thorough analysis might indicate which destination groups can be left out from the 'general instances' as these were one-offs. This can reduce the complexity of the problem and make better solutions possible. Remark that the specific dummy solutions, which used fewer cars and trains than the universal counterparts resulted in better solutions.

→ In this thesis, we have assumed that trains arrive | depart at time steps, without assigning a specific time to the operations. While this greatly reduces complexity, it makes more specific - and most likely better - solutions not possible. For example, a train that departs at 8:00 and one that arrives at 8:30 could use the same track in practice, while they can't according to our formulation if both 8:00 and 8:30 fall in the same 'time block'.

  Given the limitations on the number of blocks (63 in total), this would mean that significant changes are necessary. However, there are workarounds for this issue. One is to use a planning heuristic that works with more specific time blocks or to add extra blocks between reclassifications in which no reclassifications are allowed.

→ By design, we assume that each reclassification part has a fixed start and end time. Some might be on a track for more than 24 hours. In practice, reclassification parts can be combined (e.g. ones that are reclassified a day apart), this can reduce the required capacity at the yard. This might especially be beneficial for the SA approach as this one reserves a fixed number of classification tracks per time step for reclassifications. If reclassification parts have to be reclassified within 24 hours, there is a clearer upper bound on the number of reclassification parts.

→ As mentioned before, there are currently no simultaneous classifications at Kijfhoek. The physical infrastructure could handle it if needed, but it would require an update to the software. While using simultaneous classifications will not have an effect on the number of classification tracks needed, it could be used to handle classifications for third parties while doing other operations at the other track.

→ More research in the designation of reclassification slots might have a positive effect on the solution quality. In this thesis, we have worked with nine reclassifications per day, three in each shift, but it might be that a different distribution can yield better results.

→ For our research, we have assumed that at most one train can be at any given track. In practice, this is not the case. If feasible for the arrival | departure times and track capacity, trains might be put together on the same track. Using this possibility could improve the solution quality as the available capacity is the main bottleneck at Kijfhoek.

Recommendations

We have seen how the proposed solution approaches perform in many different scenarios. Below, we will give some recommendations on how they can be integrated successfully into the operations of DB Cargo. We suggest using the following three-phased approach. Note that in all of these scenarios, the algorithms will be an aid to, not a replacement of, the current way of operating. There are many areas in which the proposed solution approaches are less flexible than may be desired (e.g. emergency maintenance, cars ending up on wrong tracks, special requests, ...).

From dummy solution to quarterly planning. First, we suggest that the dummy solutions found can be used to improve the quarterly planning (i.e. *sporenplan*) that is currently the basis for the daily operations at Kijfhoek. Discussions on this subject have already indicated that the universal dummy solutions found by our solution approaches can help improve the planning.

Pilot: As a pilot, we propose that a quarterly planning is created based upon the dummy solutions found by our solution approaches. Then, for some weeks, a detailed study should be done on the feasibility of these solutions. This to find the weaker spots in the planning and to show the improvements of the new solutions.

Gain insights in available capacity. To perform shunting operations for third parties, capacity is required (i.e. locomotives, crew, 'hump time', tracks). The proposed solution approaches can help indicate when and where there are generally pockets of free time to perform shunting operations for third parties. Remark that this can only be done after the quarterly planning has been looked at.

Pilot: To see if the capacity is really available, we advise that with the improved quarterly planning during the realization, it is verified that at the indicated moments, capacity is indeed available. This way, we could guarantee (under normal circumstances) that the shunting operations for third parties can indeed be performed. An option to make sure the blocks are reserved, one could add a fictional train to the instance that has to be classified during the reserved time slot.

Integrating into daily operations. Lastly, we would like to point out that we have found that reoptimizing the planning during the day based on the latest available information has many benefits. To that end, we recommend that the new planning approaches should be used to reoptimize the general planning during the day to fully exploit the latest available data.

Pilot: In alignment with the first recommendation, we advise that the algorithms be used parallel to the real-life realization and that at first lessons be learned to further finetune the algorithms to iron out the last details. Then, the proposed solution approaches can be used to create suggestions for the process coordinators at Kijfhoek.

Future Research

In this thesis, we were able to model the problem at Kijfhoek and look at it from different directions. However, as always, there are areas for further research. In the overview above on opportunities, we have already indicated some specific improvements that can be made to improve the model for Kijfhoek especially.

Generally, we would recommend future research to be focused on the following. First of all, we have solely looked at the cars and trains and left out the crew and locomotive planning[1]. It might be fruitful to develop an integrated solution approach in which both crew, locomotive, and car planning are done at the same time.

Furthermore, in our formulations, we assume a discrete time frame. In real-life however, time is a continuum. Future research is necessary to check whether or not having a more detailed planning (e.g. quarter or even minute based) will result in better solutions.

In this thesis, we have used a uniform distribution of reclassifications (i.e. three per shift). Future research can be done in the area of 'placing' the reclassifications, tailored to the instance. We believe this can be done either by a preprocessing heuristic or can be integrated into the formulations.

We have found that it is sometimes 'hard' to find a feasible solution for the DF | FAF in reasonable time. It might be fruitful to look into other techniques as well. An option is to model the problem as a satisfiability problem and use a SAT solver to find a feasible solution. Depending on the quality, the solution could be optimized further using the DF | FAF.

For this thesis, we have used Simulated Annealing to find good solutions for the optimization problem at hand. More research into other heuristics might be worthwhile. As our preliminary experiments show, the algorithm to find the initial solution is already quite good at finding solutions that are "almost feasible". Most of the time, only the track assignment (i.e. formation start times and transfer times) is an area of concern. Future research might not only focus on metaheuristics but on constructive heuristics as well.

The main focus point of this thesis is Kijfhoek. There are however many other hump yards around the world. While we have modeled the formulations around the constraints at Kijfhoek, they are generally alike for other hump yards. Moreover, the software created for this project can handle any number of arrival | classification | departure tracks. This means that if there are no changes necessary to the formulation, solutions can be found for other yards as well. Future research into how the found solution approaches can be applied at other hump yards might find that improvements can be made in the shunting operations of those yards as well.

[1]Note that the simulator does provide an overview of the *crew duties* (what, when, where). This can be used as a starting point for creating a crew planning.

A
Extra Neighborhoods

The following neighborhoods were tested as part of the Local Search-based algorithms. However, preliminary experiments showed that they were accepted seldom and to that end, we have removed them from the algorithm. For the sake of completion, we add them here.

Block As the name suggests this neighborhood will move a set of cars who form a 'block'. Two cars are in the same block if they have the exact same reclassification scheme. We take that this value is σ. As the new `schemeValue`, we will pick an integer in the set

$$\{v | 2^{\lfloor \lg(\sigma) \rfloor} \leq v \leq 2^{\lceil \lg(\sigma) \rceil}_v \left(\mathrm{mod}\ 2^{\text{arrival time}}\right) = 0\} \cup \{0\}.$$

Group Where blocks are defined by the reclassification scheme of the cars, groups are formed by cars going to the same destination. This is done based on destination groups. In this case, we will not pick one new `schemeValue` for all cars, but calculate a *shift* which we will add to the `schemeValue` of all qualifying cars. This *shift* is based on how much we would change the `schemeValue` of a random car from the set if we would use that car for the 'Single'-neighborhood.

Follow In the 'Follow'-neighborhood, we will select a random car and then find the set of all cars with a `schemeValue` of the same value or larger than that of the selected car. We will then calculate a new `schemeValue` of the car based on the 'Single'-neighborhood. Next, we update all other cars with the same *shift*.

TrainFollow Similar to the neighborhood described above, only this time, we limit the set of cars to the ones that are part of the same outbound train.

TrainReorder This neighborhood can be seen as a specialized version of the 'Switch' and 'Group' neighborhoods. To begin with, we look for a violated order constraint where $\rho_{lk} = 1$. If we find one or more, we randomly select one violated constraint. Remark that with these violations, there are always two destination groups, one for both cars of the violated constraint. We will collect all cars from both groups. Then, we switch the `schemeValues` of the two cars of the violated constraints and update the `schemeValues` of the other cars to the same values.

TrainTransitivityFix Next to the order violations in the case where $\rho_{lk} = 1$ we can also have violations where $\rho_{lk} = 0$. In this case, we will increase the `schemeValue` of the car with the lowest to that of the other car. We will do the same for all other cars in the same destination group, this way, we make sure that there are no violations more between cars from the destination groups involved.

B
Programming Notes

For this thesis, we have developed a solver, simulator, and integrated testing suite from scratch. In this chapter, we will go over some of the 'highlights' of the project and present how the program is made.

Remark that we won't go into detail here, but just present a short overview of the way the algorithms were implemented and the testing suite was made.

B.1. Basics

The entire project is programmed in `C++` using the `Qt` framework except for some separate scripts to transform real-life data into useful instances. These are programmed in `Python`. To solve the IP formulations, we use Gurobi under an academic license.

B.2. Architecture

Right at the beginning of this project, it became clear that research later on in the project would get ways smoother if all experiments could be done from within one program. To that end, we have created one program to do it all. However, the risk with these types of programs is that it gets hard to expand or adapt when new algorithms | testing methods are introduced. In an attempt to prevent this from happening later on, we have decided to use one data structure for solutions, the `Solution`-data structure and one for instances | levels | scenarios, the `ShuntingYard`-data structure. To make it easy to add new parameters to solutions | scenarios and/or to analyze the data by hand, we store all scenarios | solutions in the `JSON` format.

When we introduced a new algorithm, we would create an adapter (design pattern) to transform the `Solution` and `ShuntingYard` to 'local' data structures and the other way around. This way, we could create the Simulator to work with all algorithms independent of the algorithms the solutions were created | should be reoptimized with.

B.2.1. Solving IP formulations

As touched upon briefly, we will use Gurobi to find (optimal) solutions to the problem using the formulations given before (DF, FAF). To remove redundancy, we use a factory design pattern. This way, we make sure that we only have to define the variables and the common constraints once. Also, we avoid issues that could come forward when there is a difference in translating the variables into the `Solution`.

Next to this, we made sure that the part of the program that 'solves' the formulations stays as detached from the rest as possible. This has two major advantages in future applications. First, we can (relatively) easily integrate this part of the project into another program (e.g. software used by DB Cargo to create a planning). Secondly, for this thesis, we have chosen to use Gurobi but if at a later stage the solver should be interchanged, this can be done with limited effort.

B.2.2. Local Search

From the start, one criterium stood out for the Local Search algorithm - Adaptability. When we first implemented the Local Search algorithm, we knew that there was to be a lot of testing and trying to be done before we could finalize the algorithm and start the experiments. To that end, we made it as easy as possible to add and remove neighborhoods from the algorithm. Here, we follow the factory design pattern again. For each neighborhood, we needed to implement three 'main' functions:

Init: For the neighborhood, find the variables that together define a neighbor. For example, in the case of the 'Single'-neighborhood, this means selecting a car and determining the new `schemeValue` of this car.

Score: Determine the new score if the neighbor was accepted. Remark that we do not calculate the score all over again but use the old one and change it so that it corresponds to the new situation.

Apply: Only called if the neighbor is accepted. In this case, make the changes to the solution as prescribed by the neighbor.

To calculate the score, we use the `Score`-data structure. In this data structure, we keep track of the number of *carrolls* (our main objective) as well as all possible violations. This makes it easy to adjust during the optimization phase and oversee if a solution is infeasible why this is the case and what are the exact violations.

B.3. Algorithm Configuration

To test the algorithms as easily as possible, we have made the configuration as adaptable as possible. We have created a small pop-up window for both the IP- and LS-based algorithms. In these, the user can specify the parameters for the run (e.g. time limit, cooling schedule, ...). In the back-end, we will get the configurations from these and pass them through to the 'internal solver'. One can see these pop-ups as a data structure for algorithm configurations. This way, we can at all times 'intercept' the communication and make changes if necessary, or use the results from one pop-up for multiple runs (especially helpful in the simulator).

B.4. Feasibility Checker

To check if solutions are feasible, we have created a separate feasibility checker for solutions to our problem. This checker is integrated into both simulators and will not only tell if a solution is feasible, but if a solution is infeasible, it will return a complete list of violations. This makes it easy to both spot the issues and judge the 'severity'. The feasibility checker will take a `ShuntingYard` and `Solution` and thus works independently from the chosen solution approach.

B.5. Future Extensions

When the program is to be used in practice, it might be useful to add some extra features. For instance, an option to change the solution by hand from within the UI might be helpful in 'overriding' the solutions from the algorithm based on factors that are hard to implement in the model.

Furthermore, as we have mentioned before, sometimes it might be necessary to make a judgment call if a feasible solution does not exist. In these cases, a Decision Support System might help. Here, the LS algorithm would present multiple solutions and show why they are infeasible. Then, the operator can decide which solution to use.

Lastly, in this thesis, we have assumed that the track capacity is constant. However, in practice, this may not be the case as there could be maintenance on some tracks for a (relatively) short period of time. Right now, it is not yet possible to have a 'dynamic' track capacity, but in case one would like to implement this, we have already taken this option into account in the algorithm. The only part of the program that should be updated is the `ShuntingYard` data structure (and the editor for changing the `ShuntingYard`).

Bibliography

[1] Wikimedia Commons (Apdency). A12 en betuweroute. https://commons.wikimedia.org/wiki/File:A12_en_Betuweroute.jpg. Accessed: 08-2021.

[2] Ivan Belošević and Miloš Ivić. Variable neighborhood search for multistage train classification at strategic planning level. *Computer-Aided Civil and Infrastructure Engineering*, 33(3):220–242, 2018.

[3] Dimitris Bertsimas and John Tsitsiklis. Simulated annealing. *Statistical science*, 8(1):10–15, 1993.

[4] Markus Bohlin, Holger Flier, Jens Maue, and Matúš Mihalák. Hump yard track allocation with temporary car storage. In *4th International Seminar on Railway Operations Modelling and Analysis*, 2011.

[5] Markus Bohlin, Holger Flier, Jens Maue, and Matús Mihalák. Track allocation in freight-train classification with mixed tracks. In *11th workshop on algorithmic approaches for transportation modelling, optimization, and systems*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2011.

[6] Markus Bohlin, Florian Dahms, Holger Flier, and Sara Gestrelius. Optimal freight train classification using column generation. In *12th workshop on algorithmic approaches for transportation modelling, optimization, and systems*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2012.

[7] Markus Bohlin, Sara Gestrelius, Florian Dahms, Matúš Mihalák, and Holger Flier. Optimized shunting with mixed-usage tracks, 2013.

[8] Nils Boysen, Malte Fliedner, Florian Jaehn, and Erwin Pesch. Shunting yard operations: Theoretical aspects and applications. *European Journal of Operational Research*, 220(1):1–14, 2012.

[9] Christina Büsing and Jens Maue. Robust algorithms for sorting railway cars. In *European Symposium on Algorithms*, pages 350–361. Springer, 2010.

[10] Vladimír Černỳ. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of optimization theory and applications*, 45(1):41–51, 1985.

[11] Serafino Cicerone, Gianlorenzo D'Angelo, Gabriele Di Stefano, Daniele Frigioni, and Alfredo Navarra. Recoverable robustness for train shunting problems. *Algorithmic Operations Research*, 4(2):102–116, 2009.

[12] Carlos Daganzo, Richard Dowling, and Randolph Hall. Railroad classification yard throughput: The case of multistage triangular sorting. *Transportation Research Part A: General*, 17(2):95–106, 1983.

[13] Elias Dahlhaus, Peter Horak, Mirka Miller, and Joseph Ryan. The train marshalling problem. *Discrete Applied Mathematics*, 103(1-3):41–54, 2000.

[14] Kathryn Anne Dowsland and Jonathan Thompson. Simulated annealing. *Handbook of natural computing*, pages 1623–1655, 2012.

[15] The Netherlands Authority for Consumers and Markets. Rail-related services and service facilities. https://www.acm.nl/sites/default/files/documents/2019-02/rail-related-services-and-service-facilities.pdf, 2018.

[16] Michael Gatto, Jens Maue, Matúš Mihalák, and Peter Widmayer. Shunting for dummies: An introductory algorithmic survey. In *Robust and online large-scale optimization*, pages 310–337. Springer, 2009.

[17] Fred Glover. Tabu search—part i. *ORSA Journal on computing*, 1(3):190–206, 1989.

[18] Fred Glover. Tabu search: A tutorial. *Interfaces*, 20(4):74–94, 1990.

[19] Bruce Hajek. Cooling schedules for optimal annealing. *Mathematics of operations research*, 13 (2):311–329, 1988.

[20] Ronny Hansmann and Uwe Zimmermann. Optimal sorting of rolling stock at hump yards. In *Mathematics–key technology for the future*, pages 189–203. Springer, 2008.

[21] Alain Hauser and Jens Maue. Experimental evaluation of approximation and heuristic algorithms for sorting railway cars. In *International Symposium on Experimental Algorithms*, pages 154–165. Springer, 2010.

[22] M Huang, Fabio Romeo, and Alberto Sangiovanni-Vincentelli. An efficient general cooling schedule for simulated annealing. *Proceedings ICCAD, Santa Clara, USA*, pages 381–384, 1986.

[23] Riko Jacob, Peter Márton, Jens Maue, and Marc Nunkesser. Multistage methods for freight train classification. *Networks*, 57(1):87–105, 2011.

[24] Scott Kirkpatrick, Daniel Gelatt Jr, and Mario Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[25] Pieter Knops. Optimizing train shunting operations at kijfhoek. B.S. thesis, Utrecht University, 2020.

[26] Edwin Kraft. A hump sequencing algorithm for real time management of train connection reliability. In *Journal of the Transportation Research Forum*, volume 39, pages 95–115, 2000.

[27] Edwin Kraft. Priority-based classification for improving connection reliability in railroad yards-part ii: Dynamic block to track assignment. In *Journal of the Transportation Research Forum*, volume 41, pages 107–119, 2002.

[28] Edwin Kraft. Priority-based classification for improving connection reliability in railroad yards-part i: Integration with car scheduling. In *Journal of the Transportation Research Forum*, volume 41, pages 93–105, 2002.

[29] Peter Márton, Jens Maue, and Marc Nunkesser. An improved train classification procedure for the hump yard lausanne triage. In *9th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'09)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2009.

[30] Nicholas Metropolis, Arianna Rosenbluth, Marshall Rosenbluth, Augusta Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087–1092, 1953.

[31] Debasis Mitra, Fabio Romeo, and Alberto Sangiovanni-Vincentelli. Convergence and finite-time behavior of simulated annealing. *Advances in applied probability*, 18(3):747–771, 1986.

[32] Alexander Nikolaev and Sheldon Jacobson. Simulated annealing. In *Handbook of metaheuristics*, pages 1–39. Springer, 2010.

[33] ProRail. Tien jaar betuweroute. `https://www.prorail.nl/nieuws/tien-jaar-betuweroute`, . Accessed: 08-2021.

[34] ProRail. 180 jaar spoor: De betuweroute. `https://www.prorail.nl/nieuws/180-jaar-spoor-de-betuweroute`, . Accessed: 08-2021.

[35] ProRail. Zevenaar - oberhausen - derde spoor duitsland. `https://www.prorail.nl/projecten/meer-ruimte-goederentreinen-betuweroute-zevenaar-oberhausen`, . Accessed: 08-2021.

[36] Kumara Sastry, David Goldberg, and Graham Kendall. Genetic algorithms. In *Search methodologies*, pages 97–125. Springer, 2005.

[37] Eric Triki, Yann Collette, and Patrick Siarry. A theoretical study on the behavior of simulated annealing leading to a new cooling schedule. *European Journal of Operational Research*, 166(1): 77–92, 2005.