Utrecht University

Faculty of Sciences

Optimizing Train Shunting Operations at Kijfhoek

ABSTRACT

With an ever growing economy, the freight train shunting yards have become more and more congested. It is crucial that research is done on solving the industrially sized *Freight Shunting Problem* (FSP). In this thesis, we consider the FSP at *Kijfhoek*.
Kijfhoek, being the only shinting yard with a hump in the Netherlands, is of growing importance. It serves the largest port in Europe, the port of Rotterdam. This makes this shunting yard not only vital for the economy of the Netherlands, but also for the economy of Europe.
The Kijfhoek shunting yard, owned by ProRail, is sublet to multiple railway operators, including Deutsche Bahn, currently allowed to use 37 of 43 classification tracks. As the need for shunting increases, ProRail is investigating to facilitate other railway operators, reducing track availability for Deutsche Bahn. Therefore, a more efficient way of shunting is needed. We will research Deutsche Bahn's shunting problem at Kijfhoek to optimize the process and investigate the effect of reduced track availability.
In this thesis, we devise a mathematical model of the Kijfhoek-specific FSP and use historical data from Deutsche Bahn, producing larger problem instances than researched before. We solve the resulting FSP by using *Simulated Annealing* (SA) to produce optimized shunting plans.
We conclude that our SA algorithm can find good, feasible shunting schedules in a reasonable time. We are capable of creating solutions that have more than 95% of the cars depart with the first possible train, minimizing delays. We show that, when classification track availability drops below 29, shunting performance decreases significantly.

Keywords:
Freight Shunting, Simulated Annealing, Shunting Yard Capacity, Kijfhoek

Bachelor Thesis

Pieter Knops

Mathematics

Supervisor:

Dr. J.A. Hoogeveen
Utrecht University

June 9, 2020

Acknowledgements

First of all, I am extremely grateful to Han Hoogeveen for his guidance and for introducing me into the fascinating world of train shunting. His ideas and feedback have enabled me to improve the algorithm, get a more clear way of writing and smooth out the paper itself.

In addition, I am thankful to multiple people from Deutsche Bahn. I want to thank Frank Kamminga and Maarten Brussen, my contact persons at Deutsche Bahn who have freely shared their data and introduced me to the challenges they are facing. Moreover, I am extremely grateful to Hans Werkman for sharing his expertise about train shunting at Kijfhoek. It has been a terrific experience to work with them.

Special thanks to my grantparent Hubert Knops and friend Teun Druijf, who have proofread this paper. This allowed me to fix some technical and grammatical mistakes and make this thesis more clear.

Last but not least, I am grateful to my family for their moral support.

Contents

1 Introduction

With a highly interconnected economy, lots of goods and materials are transported over longer distances. Transport by train has quite some advantages over transport by road. For instance, it is cheaper and has a lower carbon footprint (Kim et al., 2009).
However, one problem with transport by train is that it is seen as unreliable. Krüger et al. (2013) reported that in Sweden a quarter of the freight trains arrived late[1]. The process of *shunting* (sometimes also called *marshalling*) plays a big role in these delays. It is therefore crucial that research in this area is done to optimize the shunting process and improve the reliability of freight trains. We will investigate possibilities of using fewer tracks.

At shunting yards, often located near industrial areas or cargo ports, rolling stock is decoupled from incoming trains and sorted into departing trains. In this way, the shunting yards play a role as hubs in the rail network where cars can be transferred from one train to another.
Shunting yards are also called *hump yards*. This comes from the fact that a shunting yard is a flat yard, on which an artificial hump is formed. One track goes over this hump and connects to multiple tracks at the other side of the hump. Cars are disconnected and pushed up against this hump by a shunting locomotive at a speed of about 5 kilometers per hour. When car goes over top of the hump, it starts to speed up due to gravity. By turning rail switches in the right way, the car is rolled into a certain track. By turning rail switches again, a next car can be rolled into a different track. In this way, the cars can be sorted.

Deciding what operations and movements of rolling stock are needed to decouple trains and sort them in newly formed trains, such that cars depart to the correct destination, is called the *Freight Shunting Problem* (FSP). In this problem, it is not only important that cars depart with the right train, but also that the cars are in the correct order in the train.

This problem is quite hard to solve. In fact, it is proven to be $\mathcal{NP}$-complete[2] (Gatto et al., 2009; Jaehn et al., 2015; Bohlin, Flier, Maue, et al., 2011). With the rail infrastructure becoming more crowded, this problem becomes even more difficult to solve.

The process of shunting mainly consists of two actions, called the *roll-in* and the *pull-out* operation, which are explained below. A shunting yard usually consists of 4 structural components, depicted in Figure 1 (Bohlin, Flier, Maue, et al., 2011; Boysen et al., 2012).

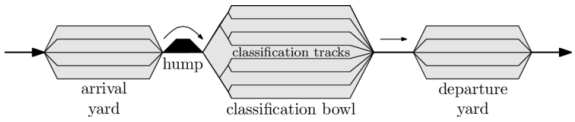


Figure 1: Schematic overview of a simple shunting yard. Modified from Bohlin, Flier, Maue, et al. (2011).

- An *arrival yard* consisting of one or more *arrival tracks*. These arrival tracks store cars from arrived trains.
- A *classification bowl* consisting of multiple *classification tracks*, allowing cars to assemble into trains. These classification tracks are also used to store cars that must be shunted again. Pulling the cars from one of the classification tracks back to the arrival yard is called a *pull-out* operation.

[1] With a median delay of more than one hour.
[2] The term $\mathcal{NP}$-complete comes from the field of Computer Science and tells something about the computational complexity. For this paper, it is sufficient to know that $\mathcal{NP}$-complete problems are intractable (Garey et al., 1974; Ullman, 1975). For more information on computational complexity, including $\mathcal{NP}$-completeness, we like to refer to Papadimitriou (2003).

- A *hump*. After a car is pushed up onto this hump and is disconnected from the train, this elevated part gives the car a momentum to roll into one of the classification tracks. This action is called a *roll-in* operation since the cars are said to be rolled *into* the classification tracks.
- A *departure yard* for storing trains that are ready to depart.

In this study, the shunting process at *Kijfhoek* (see Figure 2) will be investigated and optimized. For this we will use an algorithm built on *Simulated Annealing* (SA). Kijfhoek serves the port of Rotterdam, the busiest port of Europe (Kiprop, 2018). Kijfhoek is unique: it is the only hump yard in the Netherlands. This is one of the reasons why Kijfhoek is vital for the economy for Europe.

Figure 2: Aerial photo of Kijfhoek near Barendrecht (Swart, 2012).

Kijfhoek is property of ProRail and used by Deutsche Bahn. Because ProRail wishes to rent out some tracks to other parties, Deutsche Bahn must investigate ways to use fewer tracks. Therefore, our main interest lies in investigating what effect the number of available classification tracks has on Deutsche Bahn's shunting operations.
For doing this research, we will create a mathematical model of the specific shunting operations and constraints on Kijfhoek. Additionally, we will develop a powerful tool to create shunting plans for Kijfhoek. These plans will be optimized such that the number of incorrect departures, delays and movements are minimized. The optimization algorithm will be run on different numbers of available classification tracks to see the effect of the availability (or lack thereof) of rail infrastructure at Kijfhoek.

This paper is structured as follows. We begin with a detailed example of freight train shunting in Section 2. Then, we define our problem in Section 3 and give an overview of research that has already been done in this area in Section 4. In Section 5, we will go over our model of shunting

at Kijfhoek. Then, in Section 6, the heuristic Local Search, and more specific, SA, is explained. We will demonstrate some theoretical aspects of SA.
In Section 7, we describe how SA is deployed on this model. We will use this implementation on data of Kijfhoek acquired from Deutsche Bahn. We will give a small overview of this data and describe how we use this data in Section 8. This will lead us to the results in Section 9 and finally, in Section 10, results from the experimental survey will be translated back to the shunting process at Kijfhoek. Also, we discuss some possible future research topics and extensions of the model.

There are some differences between terminology used in Dutch and terminology used in literature. In this paper, we will use terminology customary to literature. Some translations are given in Appendix B.

2 Freight shunting: A detailed example

Before we talk about our problem definition, we will first explain what happens at a shunting yard. For this we will give a small example of the freight shunting problem. This section could be skipped by a reader who is familiar with freight shunting but note that we will use this example in Section 5.

We will consider a shunting yard containing 3 arrival tracks, 3 classification tracks and 3 departure tracks (see Figure 3). We have arrival tracks a_1, a_2 and a_3, classification tracks c_1, c_2 and c_3 and departure yards d_1, d_2 and d_3, all labelled from top to bottom.

One thing that directly becomes clear, is that the structural layout of this example is not as simple as the schematic layout as depicted in Figure 1. With the layout in this example, we come closer to the structure of the Kijfhoek (see Figure 6).

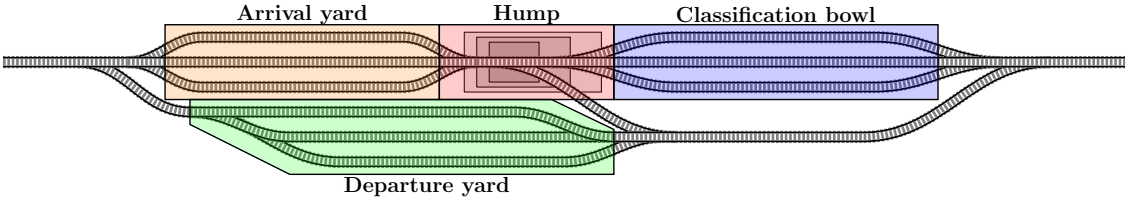


Figure 3: Structure of the shunting yard that we will use in the example. The hump is made graphically visible by three gray rectangles.

In our example, two trains will arrive. The first one from the left and the second one from the right. We will four outbound trains, each serving a different destination. The order of cars in these outbound trains will not be relevant in this example. We will number all cars according to the order of arrival and color-code them according to their final destination. The destinations of the cars with the colors orange, green and blue is located right from this shunting yard and the remaining cars, colored yellow, have a destination located left from this shunting yard. In this way, we will have three outbound trains that will depart to the right and one that will depart to the left.

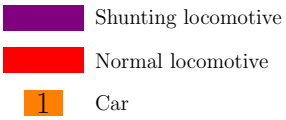
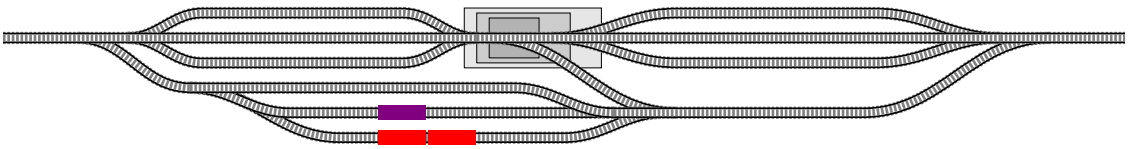


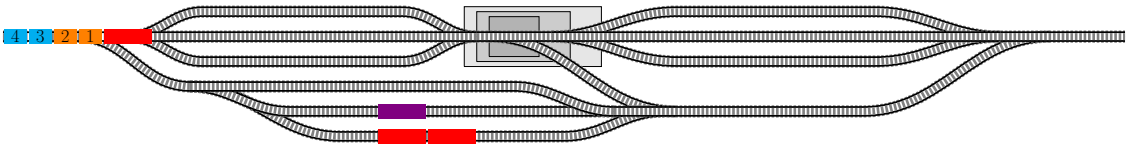
Figure 4: Legend of rolling stock visible in the Example States. The cars will have a unique number and will be color coded according to their destination. The colors purple and red are reserved for the locomotives.

We begin with one shunting locomotive, which is used for shunting operations, and two normal locomotives on our shunting yard (see Figure 4 and Example State 1). The shunting locomotives differ from normal locomotives. For instance, shunting locomotives are allowed to push cars which normal locomotives are not allowed to do.



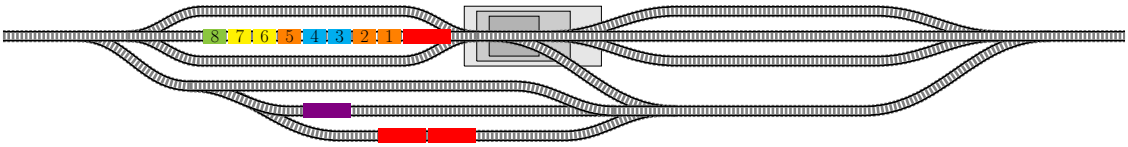
Example State 1

At the beginning of the day, one train arrives from the left. This train brings 8 cars (in Example State 2, only four cars are shown) having multiple destinations.



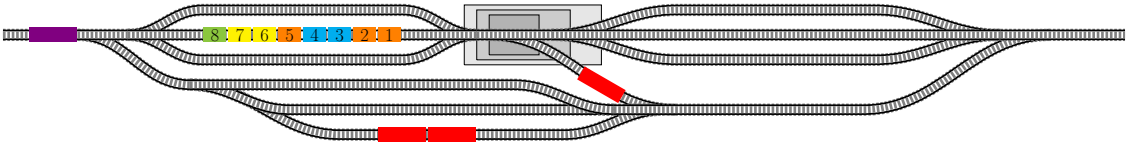
Example State 2

We let the train arrive at arrival track a_2. The locomotive is disconnected from the cars and an arrival check is performed to check the cars on any abnormalities before they are disconnected. In the meantime, we let the shunting locomotive drive to the left side (see Example State 3).



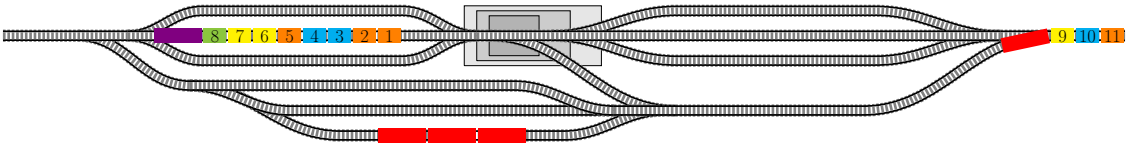
Example State 3

The locomotive drives away to be stored on departure track d_3 (see Example State 4).



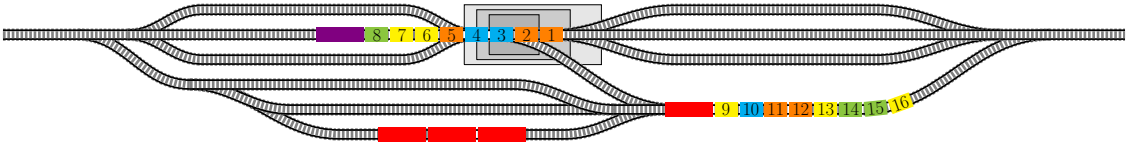
Example State 4

After the shunting locomotive arrives on arrival track a_2, it starts to push the cars on that track to the hump. At the same time, a second train arrives from the right side with cars numbered 9 to 16. Again, cars will be color-coded according to their target destination and numbered according to the order of arrival (see Example State 5). This train can't drive through the classification bowl since the bowl will be occupied by the cars from the first train.



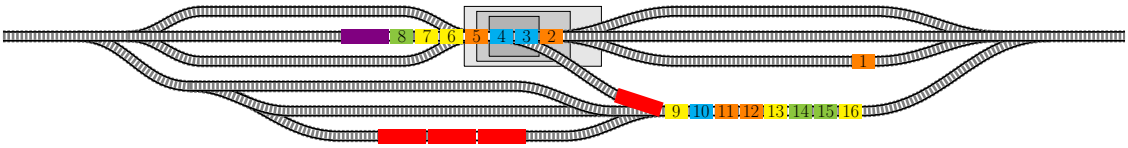
Example State 5

The cars from arrival track a_2 are driven up onto the hump. We decide to split each car onto different classification tracks to sort them. This is the beginning of the roll-in operation (see Example State 6). The train arriving from the right cannot drive to the left and push the cars to the arrival yard, so it has to drive over the hump.



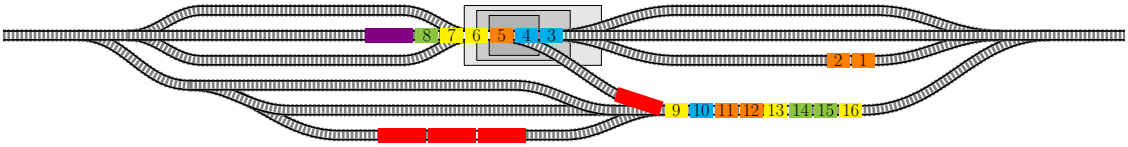
Example State 6

Some rail switches are turned and the first car rolls onto classification track c_3. The train that arrived from the right has to wait until the roll-in operation has finished (see Example State 7).



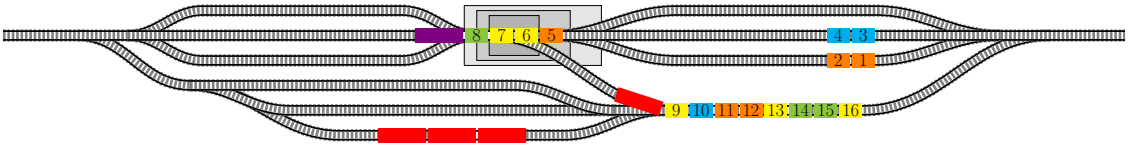
Example State 7

The second car has, having the same destination as the first car, is also rolled onto classification track c_3 (see Example State 8).



Example State 8

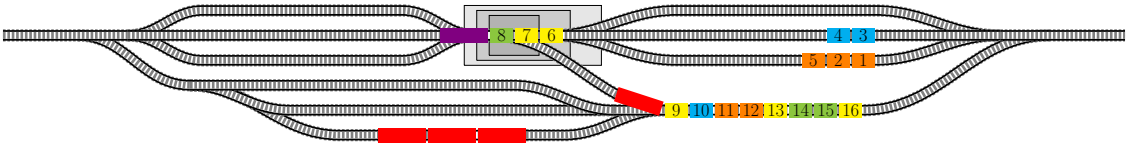
The third and fourth car are rolled onto classification track c_2, since they have a different destination that the first and second car. For this, some rail switches must be turned (see Example State 9).



Example State 9

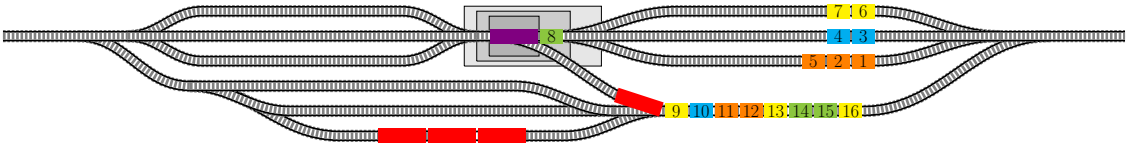
We turn the switches back such that cars can roll in to track c_3 again. In this way the fifth joins the first and second car having the same destination.

From now on, we will omit the fact that switches are turned (see Example State 10).



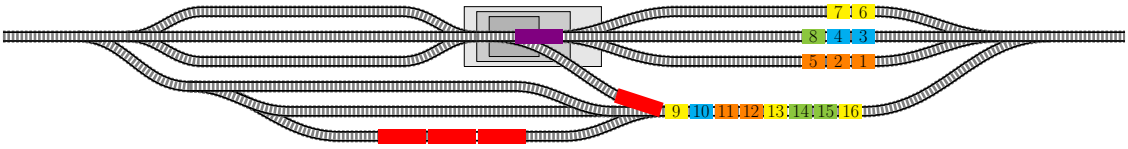
Example State 10

The sixth and seventh car have a new destination. We roll them onto classification track c_1, which is empty (see Example State 11).



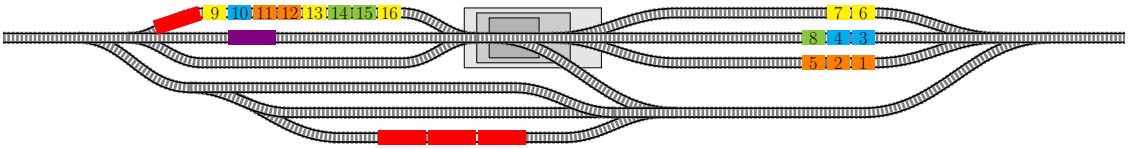
Example State 11

All classification tracks are populated, but our eighth car has a different destination than all cars located in classification bowl. Since track c_2 is a track that contains the fewest cars, we choose track c_2 to roll the eighth car into. Now, the first roll-in has finished (see Example State 12).



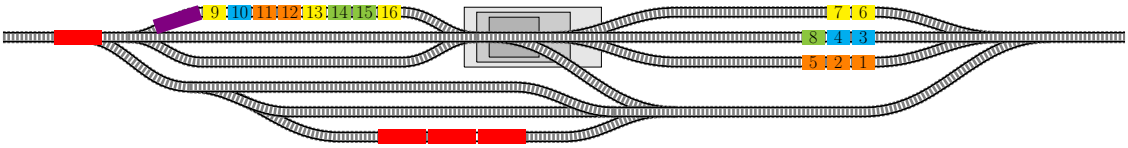
Example State 12

Now, we remove the shunting locomotive from the hump such that the second train can finally drop its cars on the arrival yard (see Example State 13).



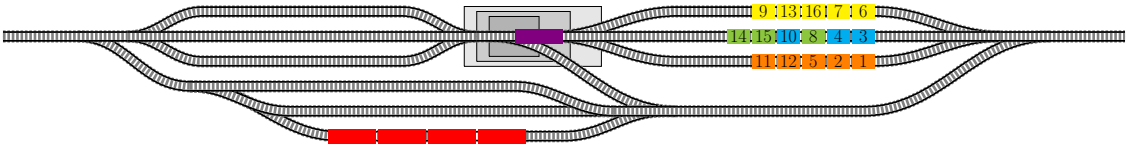
Example State 13

We disconnect the locomotive from its cars and bring this locomotive to track d_3. We distonnect the cars and start a second roll-in from track a_1 (see Example State 14).



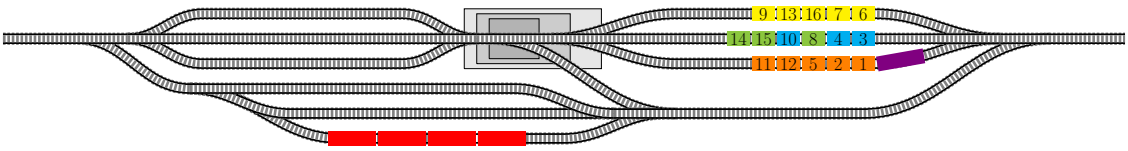
Example State 14

All cars are rolled into the classification tracks corresponding to their destinations. We omit the individual steps (see Example State 15).



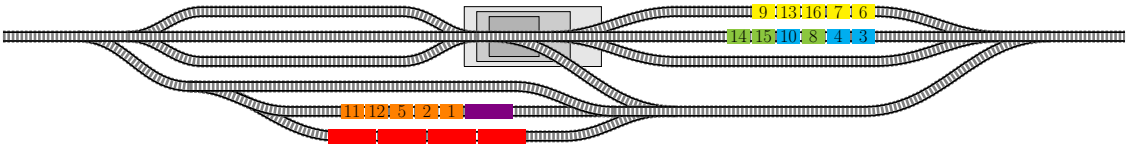
Example State 15

The orange cars have been sorted and can be transferred to the departure yard. For transferring these cars, we bring the shunting locomotive track c_3 (see Example State 16).



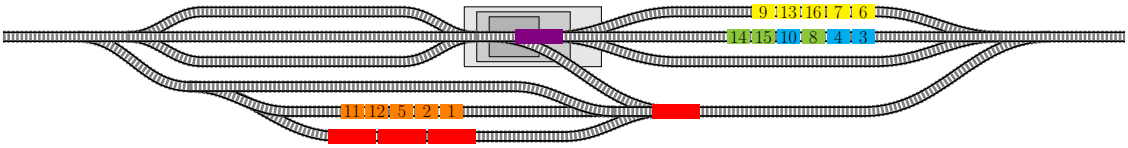
Example State 16

The orange cars are transferred to track d_2 on the departure yard (see Example State 17).



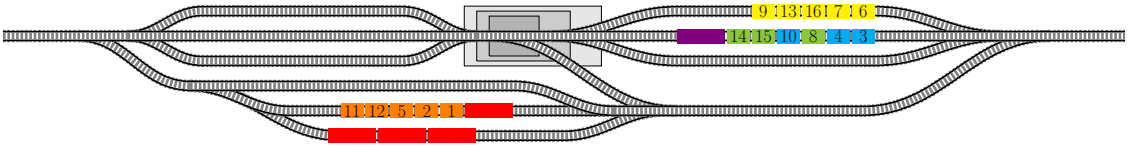
Example State 17

We choose a locomotive to bring the orange cars to their destination. Since a classification track has become available, the shunting locomotive returns to the hump for sorting the blue and green cars (see Example State 18).



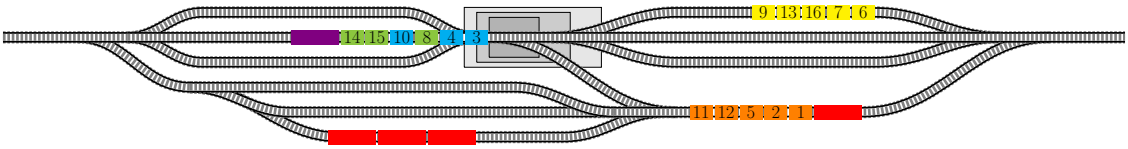
Example State 18

We connect a locomotive to the cars on track d_2 and the shunting locomotive to the cars on track c_2 (see Example State 19).



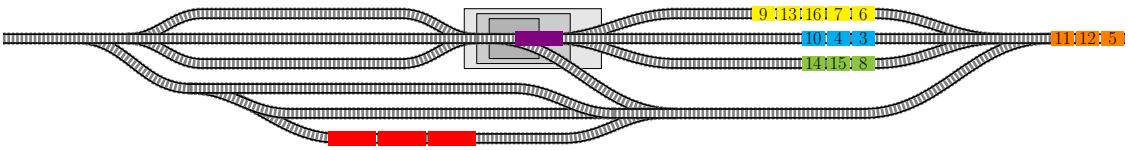
Example State 19

The orange cars leave the departure yard and the cars from track c_2 are pulled out onto arrival track a_2 (see Example State 20).



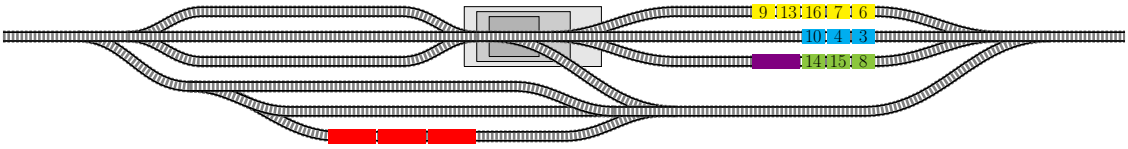
Example State 20

During the third roll-in, the blue and green cars are rolled into tracks a_2 and a_3 respectively (see Example State 21).



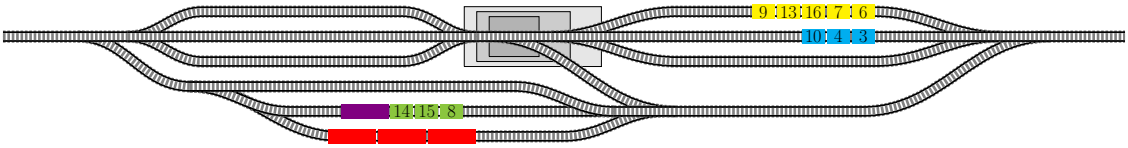
Example State 21

The green cars have the next departure, so we connect the shunting locomotive to them (see Example State 22).



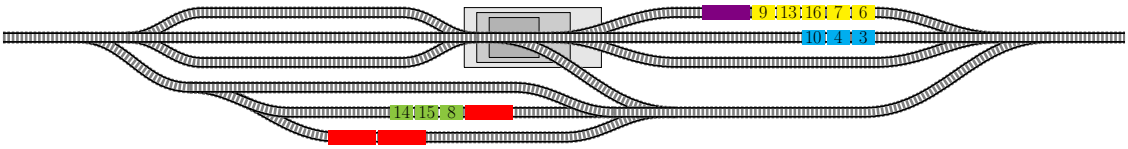
Example State 22

The green cars are brought to departure track d_2 (see Example State 23).



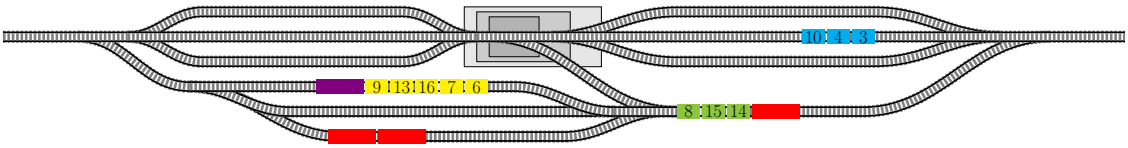
Example State 23

Now, a locomotive leaves track d_3 to pick up the green cars and the shunting locomotive connects to the yellow cars to bring them to departure track d_1 (see Example State 24).



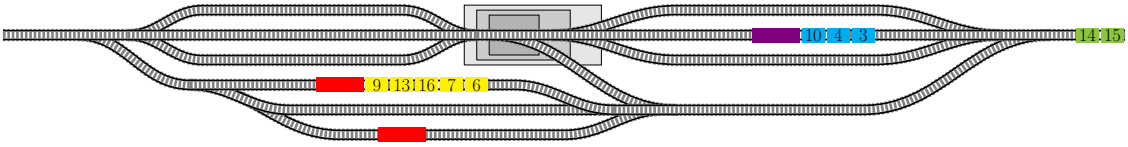
Example State 24

The locomotive departs to the right with the green cars. At the same time, the yellow cars are transferred to departure track d_2 (see Example State 25).



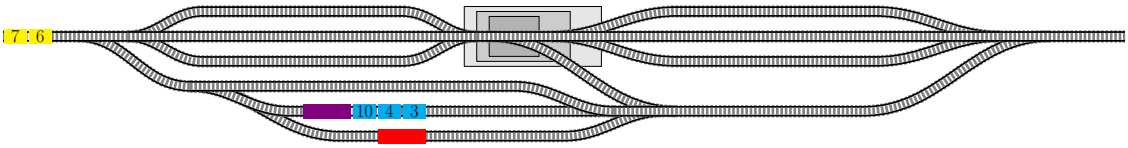
Example State 25

The shunting locomotive goes to track c_2 to transfer the last cars to the departure yard and a locomotive picks up the cars on track d_1, to drive those to the left exit (see Example State 26).



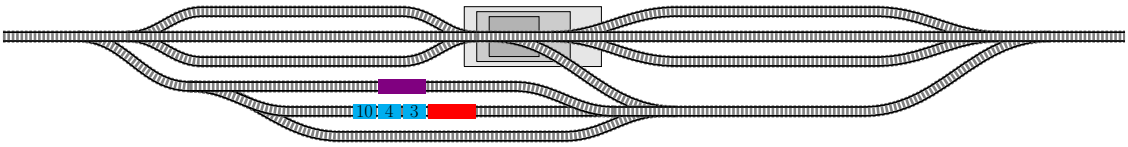
Example State 26

The blue cars are driven to track d_2 (see Example State 27).



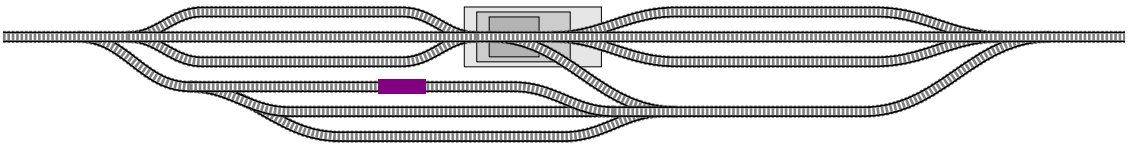
Example State 27

The last locomotive connects to the blue cars to take them to the right exit (see Example State 28).



Example State 28

Finally, all cars have departed and only the shunting locomotive is left on the yard. This concludes our example (see Example State 29).



Example State 29

No example can fully cover all aspects of reality. For instance, most shunting yards, including Kijfhoek, have tracks to store locomotives so that they don't stand in the way on the departure yard like they did in our example. Also, often there are multiple shunting locomotives available to use.
In addition, the size of this example does not do justice to how difficult the FSP can be when considering industrially sized problems with thousands of cars arriving and departing to and from these shunting yards every week.

What this example does illustrate correctly, is the challenges present on a shunting yard. We have seen that deciding the roll-in tracks is not as straightforward as it seems. Especially when too few classification tracks are available, problems arise. Also, the timing of the actions is crucial. In this example, we have seen that a train had to wait to finally being able to deliver its cars onto the arrival yard. Not only did this block a portion of the infrastructure, but it can also cause delays for the cars in that train. This blocking aspect of the shunting problem is strongly dependent on the structure of the shunting yard.

In practice, this problem is even more difficult to solve. For instance, some trains serve multiple destinations and not every order of cars in a train is acceptable. On top of that, any delay at the shunting yard may cause cars to miss their train, as outbound trains cannot be delayed more than a couple of minutes. We will go over these and any other restrictions of our Kijfhoek-specific FSP in the next section.

3 Problem definition

In this FSP, we schedule all operations on the shunting yard Kijfhoek, including on what tracks trains will arrive and from what tracks trains will depart. We will have a planning horizon of one week.

Kijfhoek consists of 14 arrival tracks, 43 classification tracks and 8 departure tracks which makes a total of 65 tracks. Currently, around 6 classification tracks are not available for Deutsche Bahn to use. For each track, we use its length as defined in the *Kijfdis* system (see Appendix A), the system Deutsche Bahn uses for planning their shunting operations at Kijfhoek. Of those classification tracks, 27 tracks allow for a departure in the south direction (see Appendix A). No classification track allows for a departure northbound.

At Kijfhoek, we identify six possible operations (see Figure 5). Note that all actions move one or more cars, so they could also be called movements. Together with Deutsche Bahn, we decided to exclude the push operation from our model since the push operation normally is not used.

Arrival: Trains can arrive from the north and the south on an arrival track. After a train has arrived, an *arrival check*, in which cars are checked on abnormalities such as damages, is performed. This takes 15 minutes. Since the locomotive must be able to leave the track, the arrival track must be empty upon arrival.

Roll-in: The roll-in operation moves cars from an arrival track to one or multiple classification tracks, as explained in the Introduction. The duration of this action depends on how many meters of cars are rolled in, since that dictates how far the shunting locomotive must travel. Together with the necessary preparation steps and checks, it takes approximately 3 hours to roll in around 35 cars of average length. We assume a roll-in operation to fully empty an arrival track.

Pull-out: The pull-out operations takes 15 minutes and moves cars from a classification track back to an arrival track such that they can be rolled in again.

Push: In this operation, which also takes 15 minutes, cars are transferred from one of the arrival tracks to either a classification track or a departure track. No cars are decoupled during a push. Usually, this operation is only used if a train arrives late such that cars still can depart as planned.

Departure: In contrast to the usual structure of a shunting yard, some classification tracks at Kijfhoek allow for a departure southbound. For a departure, 30 minutes are reserved for connecting the locomotive and do some final checks on the cars and connections.

Transfer: Cars that are ready for a departure can get *transferred* from the classification bowl to the departure yard. This takes around 15 minutes. Like stated before, some classification tracks allow for a departure southbound so not all cars have to be transferred.

We will use real data about arrivals, departures and track lengths, acquired from Deutsche Bahn. Arrivals can be delayed for some minutes but for the departures this is unfavorable. If a departure is delayed more than three minutes, a new line plan has to be requested. That is why, in cooperation with Deutsche Bahn, we decided that departures with a delay of more than three minutes are not accepted.

Rail junctions are considered in a fundamental but sufficient way. In Figure 6, five junction groups, including the hump, are depicted. At these places and tracks, only one operation can be undertaken at the same time to make sure a train will not use a junction or a track that is reserved for a different train.

In the data, not all cars that arrive have a departure in the same week. At Kijfhoek, the cars are not only sorted to their departing train, but cars can also be parked there for longer time. In this model, these cars can be left on any of the classification tracks and in any order at the end of

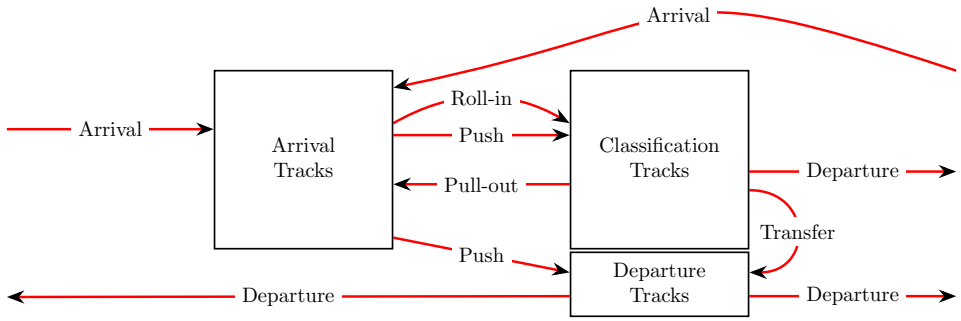
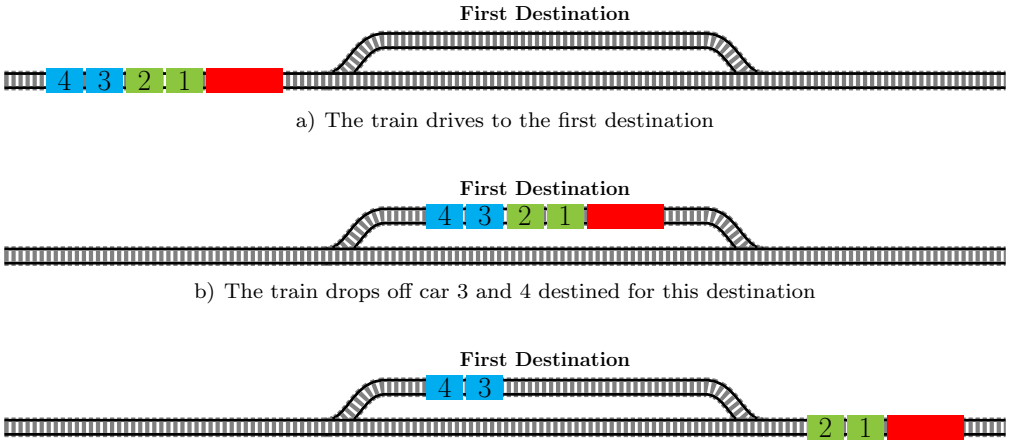


Figure 5: Schematic overview of different parts of Kijfhoek with the possible movements. The north is on the left side and the south on the right side[a].

[a] We call the right-hand side the south and the left-hand side the north, customary to the naming used at Kijfhoek. The reason that they use these names is that the right-hand side exit connects to destinations in the south and that the left-hand side exit connects to the port of Rotterdam, which is located north-west from Kijfhoek.

the week, but during the week they should not block other cars that do have to depart.

Cars cannot just be in any order in departing trains. Cars are put in around 35 destination groups and can be in any order within a destination group. In contrast, destination groups are only allowed in one order in a departing train for easy drop-off. This order is extracted from the data about train departures. Most trains only serve 1 or 2 destinations. see Figure 7 for an example of how having multiple destinations in a train works.



a) The train drives to the first destination

b) The train drops off car 3 and 4 destined for this destination

c) The train drives further to the second destination

Figure 7: A visualization of how a train serving two destinations drops of some cars at the first destination and departs with the rest of the cars to a second destination.

Additionally, We will consider car delays. Our goal is to send each car away to the right destination as fast as possible.

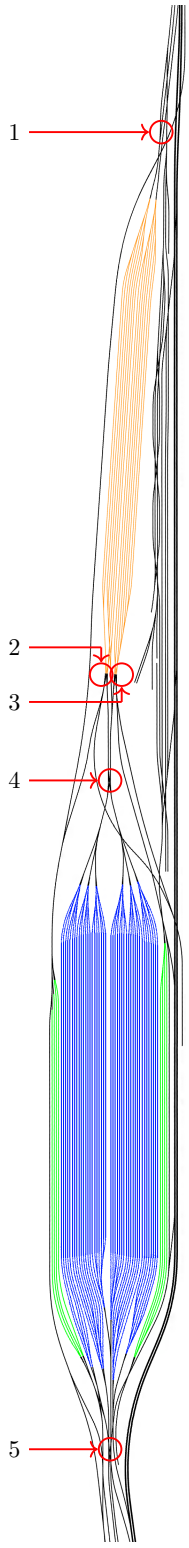


Figure 6: Track layout of Kijfhoek. Arrival tracks, classification tracks and departure tracks are colored orange, blue and green respectively. The junction groups that we consider are numbered and encircled red. The hump has been given number 4. This visual layout of Kijfhoek is created using OpenRailwayMap.

There are also some things that we, in cooperation with Deutsche Bahn, decided to not take into consideration.

For instance, Deutsche Bahn does not only sort cars according to their destination, but also according to their terminals. This means that cars will be in the correct order in the train to be left on the property of a client. Since this data was difficult to acquire, this is not modelled in this paper.

The hump at Kijfhoek allows for two cars to be rolled-in simultaneously, but we will not consider this because Deutsche Bahn does not use this possibility. This means that we only consider situations in which only one car is rolled simultaneously.

Also, some cars cannot be shunted over the hump. This can be because of multiple reasons. There are cars that simply cannot drive over the hump because of the hump's curvature[3]. Normally, these cars must take a detour around the yard to get to the other side of the hump.
Other cars may contain hazardous substances. These cars may not move without a locomotive and because in a roll-in cars move by gravity and not with a locomotive, these cars cannot be part of a roll-in. In our thesis, we do not consider these restrictions, i.e., all cars can be rolled in over the hump.

Also, crew and locomotive scheduling, a research topic on its own, is not considered in this paper.

We will solve the problem we just formulated by creating a mathematical model of the specific shunting operations and constraints on Kijfhoek. We want to maximize the number of correct departures and minimize delays of those departure while obeying the constraints following from this problem definition.

[3] Because of the curvature of the hump, the bottom of the car would touch the rail on the hump if the car is too long.

4 Related Literature

We are not the first to research the freight shunting problem. We give a small overview of what has been researched and how this paper relates to those studies.

Firstly, we will consider Bohlin, Flier, Maue, et al. (2011) who investigate the track-allocation side of the problem. They consider roll-in times and times on which other actions begin to be a given and plan which classification tracks will be used for forming the outbound trains. They reserve an entire classification track for the cars of only one outbound train.
In contrast, we will plan all roll-ins and other actions ourselves. Also, we will not be reserving classification tracks for any outbound train, as it may reduce the capacity of the shunting yard.
In addition, Bohlin, Flier, Maue, et al. (2011) has proven $\mathcal{NP}$-completeness for the FSP with unequal track lengths but that, in the case of equal track lengths and enough tracks, the problem of assigning tracks to outbound trains can be solved in polynomial time. They found heuristics and an *integer program* (IP) approach to solve the problem. They extend their problem by making it possible to store cars on a special subset of the tracks temporarily, called *mixing tracks* and did a case study on that extended problem.
One additional difference is that Bohlin, Flier, Maue, et al. (2011) assume that any order of cars in an outbound train is acceptable, which is not true for us. Also, because we plann all actions ourselves, we have many additional constraints like availability of arrival tracks, the hump, and rail junction switches.

Bohlin, Flier, and Gestrelius (2012) combine integer programming and column generation for a new heuristic to find optimal solutions to industrially sized problems with short planning horizons (two to five days). Finding optimal solutions for larger instances took too much time. This has let us to the conclusion that it may be difficult to find a good IP approach capable of solving our problem.
A year later, Bohlin, Gestrelius, et al. (2013) extended the case study done by Bohlin, Flier, Maue, et al. (2011) in which they use stochastic arrival times.

Adlbrecht et al. (2015) consider the shunting problem on a shunting yard without a hump. They only use trains consisting of single blocks. That is, all cars in an outbound train share the same destination. This is similar to how Bohlin, Flier, and Gestrelius (2012) assume that the order of cars in outbound trains does not matter.
Gestrelius et al. (2013) extend research done by Bohlin, Flier, and Gestrelius (2012) in such a way that cars are sorted in blocks according to their drop-off location. This comes closer to our study because in this way, multiple destinations can be served by one single outbound train (see Figure 7). The planning horizon of Gestrelius et al. (2013) is three days.

In most research, it is assumed that the arrival times of trains and the order of cars in those trains, are known in advance. However, this is often not the case (Bohlin, Flier, and Gestrelius, 2012). For example, Bohlin, Flier, and Gestrelius (2012) assume that arrival times are known in advance. To counter uncertainties and changes of the actual arrival times or changes in car order, they suggest to do more research to flexible shunting schedules.
Cicerone et al. (2009) have researched this aspect of the shunting problem theoretically. They consider effects and recovery strategies for one missing car, a car that has a different position in a train or a track that becomes unavailable.

Not only the freight shunting problem has been researched, but studies also have been done to the shunting problem for passenger trains. Freling et al. (2005) call the problem of passenger train shunting the *train unit shunting problem* (TUSP).The main difference lies in the fact that the passenger train units are mostly self-propelling, so no central hump is needed. As a result, passenger train shunting yards do not follow the main structure as depicted in Figure 1. Also, for passenger trains, a shunting yard does not have as main function to act as a central hub where trains are decoupled and reformed, but is mainly used as central parking spot.
Broek (2016) propose the heuristic *Local Search* (LS) for solving TUSP. They concluded that state-

of-the-art IP algorithms were outperformed by LS for solving TUSP. We concluded that LS might be a good alternative of IP for solving FSP too.

In addition, LS has been combined with graph classification to determine capacity of shunting yards of passenger trains by Ven et al. (2019), which is related to our study because we are interested in how many classification tracks are enough for the shunting operations of Deutsche Bahn.

5 Model

To convert a shunting yard into a model, we must pay close attention to the properties of the shunting yard, as well as to any restrictions present. These could be regulatory constraints or constraints following from physical properties and corporate policies. In this Section, we will focus on how actions are planned in a way that is allowed.

5.1 Actions

Each action in our model (see Section 3) must be well-defined and is characterized by the following properties.

Arrival: Which arrival track the train arrives on as well as what cars the train brings to Kijfhoek.

Roll-in: From which arrival track this roll-in action is performed as well as a sequence of classification tracks defining to which tracks cars will be rolled into.

Pull-out: The source: a classification track, and the destination: an arrival track. Not all cars from the classification track have to be pulled-out, so also the number of cars that are pulled out must be defined.

Departure: A track from which the train wil depart[4] and the number of cars that will be picked up.

Transfer: A classification track as the source and a destination track as the destination. Like with the pull-out, not all cars on the departure track have to be transferred, so also the number of cars that will be transferred has to be specified.

We will combine the planned actions in a sequence which we will call the *schedule*. The order of the actions in the schedule does not dictate when the action will be performed, as will be explained in the next subsection. A formal definition of the schedule is given as follows.

Definition 5.1 *The sequence of all actions a_i that are planned will be called a schedule. A schedule consisting of n actions will be denoted as $s = (a_i)_{1 \leq i \leq n}$.*

Keeping track of what actions will be performed is not enough. We also must keep track of the location of each and every car. This information will be stored in the yardstate, for which we have the following definition.

Definition 5.2 *The set Y that contains a tuple (τ, c_τ) for every track τ where c_τ is the sequence of cars that are located on that track, is called a yardstate.*

We will denote the yardstate after performing the first j actions from schedule s as $Y_s(j)$.

5.2 Blockers

To ensure that not more than one action at the same place and time will be performed, we use blockers. A blocker is one of the 65 tracks or 5 junction groups, including the hump (see Figure 6). For each blocker, we will keep track of when it will be blocked. All actions in the schedule will be performed as soon as the blockers allow. For this, we use an Acyclic Directed Graph.

As an example of how these blockers are used, we will build up a schedule based on the actions in the example described in Section 2. In that example, we identify 14 actions: two arrivals denoted as ar_1 and ar_2, three roll-ins denoted as ro_1, ro_2 and ro_3, one pull denoted as pu_1, four transfers denoted as tr_1, tr_2, tr_3 and tr_4, and finally, four departures denoted as de_1, de_2, de_3 and

[4] Remember that some classification tracks allow for a departure southbound, so this track does not need to be a departure track per se.

de_4. All these actions have been numbered, based on when they were performed.
Together, they form the following schedule.

$$s = (ar_1, ro_1, ar_2, ro_2, tr_1, de_1, pu_1, ro_3, tr_2, de_2, tr_3, tr_4, de_3, de_4).$$

From these actions, we can build a directed acyclic graph representing a partially ordered set. We will have a node for each and every action and an arrow from action a to action b if action a blocks action b. That is, if action b will be using one or more blockers that action a will use before action b does. This graph, in which we have left out redundant arrows (arrows that also follow transitively) is depicted in Figure 8.

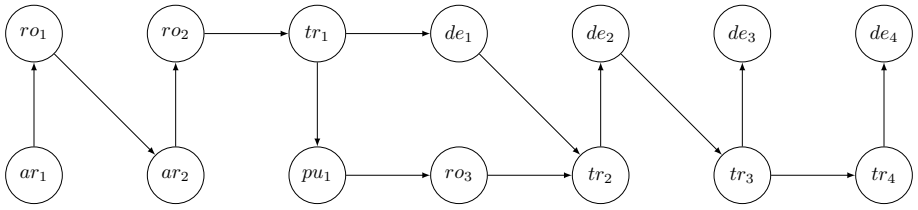


Figure 8: The directed acyclic graph following from our example.

To make this a bit more clear: The reason that there is no connection from de_3 to de_4 is that de_3 and de_4 do not make use of any of the same blockers[5]. In contrast, there is a connection from tr_4 to de_4, because they both use departure track d_2. The reason why the arrow goes from tr_4 to de_4 and not the other way around, is that tr_4 comes first in the schedule.

Remember that we do not take locomotive scheduling into consideration. This means that we will not consider locomotive availability when creating arrows. If we would do so, we would have an extra arrow between tr_1 and pu_1, since they both use the shunting locomotive.

To start an action, we consider the graph and look if we can find nodes that have no incoming arrows. The actions corresponding to these nodes do not have to wait on any other action to finish. If we can find such nodes, we start the corresponding actions and calculate when they will be finished. When an action has finished, we will remove it from the schedule. Now we check again if we can find any actions that can be started.
Some actions do not release their blockers all at once. For instance, an arrival action will release its rail junction blockers just after the train has arrived. The arrival track itself will be blocked for another thirty minutes, as the arrival check has also to be performed. On the moment that an action releases a blocker, we check if an arrow disappears. We give an example of what happens in that case below.

For the arrivals and departures, we have to keep in mind that these actions cannot just be planned as soon as possible. Trains should not depart earlier than described in the line plan and, most certainly, we cannot plan an arrival to happen if its train has not yet entered the yard. We wait until the train arrives or has to depart before starting the corresponding action.

Considering our example, we see that ar_1 can be started first since it is the only node with no incoming arrows. We wait until the train has arrived and start action ar_1. After the arrival check has finished, we remove the arrivals node and its arrows from the graph (see Figure 9).

[5] Remember that de_3 and de_4 used different departure tracks.

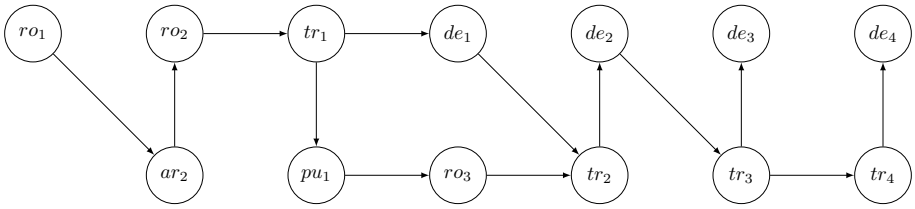


Figure 9: The directed acyclic graph after action ar_1 has finished.

Now ro_1 has no incoming arrows and can be started. We will also plan ar_2, ro_2, and tr_1 in this fashion.

As most actions, tr_1 uses multiple blockers. It will release the blocker of the hump before it will release the blocker of departure track d_2 (see Example State 18). At the point that it has released the blocker of the hump, we will have the graph depicted in Figure 10 that has no arrow from tr_1 to pu_1 anymore. At this point, also action pu_1 can be started.

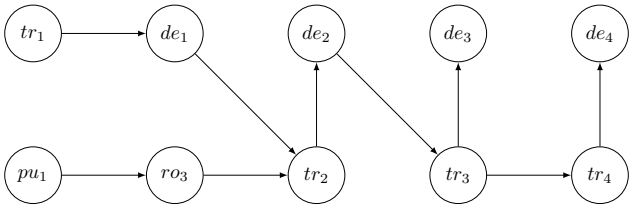


Figure 10: The directed acyclic graph after action tr_1 has released the blocker of the hump.

We continue this fashion of starting actions until all actions have been finished.

What could be noted is that pu_1 is started before de_1[6], even though de_1 comes before pu_1 in our schedule. The execution order of these two actions has switched. This is possible because there those two actions do not use any of the same blockers so there is no arrow between these two actions.

At the Kijfhoek there are many more tracks and trains so many actions can take place simultaneously. By using these directed acyclic graphs, we can start the actions from the schedule in an efficient way while assuring that the actions do not conflict with each other.

[6] This is not exactly the case in our example in Section 2, in which we also considered locomotive scheduling.

6 Local Search

In this section, we explain Local Search and two of its probabilistic versions, Iterative Improvement and Simulated Annealing. We will use the latter for creating optimized solutions. Local Search is a heuristic method for solving computationally hard optimization problems (Lourenço et al., 2003).

6.1 Iterative Improvement

Before Iterative Improvement can be explained, we will need some notation.

Firstly, the (finite or infinite) set of possible solutions of the optimization problem (also named the *search space*) will be denoted by $\mathcal{S}$ and we will denote individual solutions by s. Note that a solution does not need to be optimal. In our case, all schedules are solutions, even the schedules that does not contain any actions.
For deciding if one solution is better than another, we use a cost function $\mathcal{C} : S \to \mathbb{R}$. Usually $\mathcal{C}(s)$ is called the "score" or "cost" of the solution s and often the goal is to find a solution s that has the lowest cost. In other words, the aim is to minimize $\mathcal{C}(s)$ over all s in $\mathcal{S}$. We will define our cost function in the next section.
In addition to the cost, we are also interested in knowing if the solution is an allowed solution, i.e., does the solution obey to all constrains? We call a solution that obeys all constraints *feasible* and a solution that does not obey all constraints an *infeasible* solution. In our case, an example of an infeasible solution would be a solution in which an outbound train is delayed more than three minutes.

Like stated before, it is computationally infeasible to calculate $\mathcal{C}(s)$ for all solutions s in $\mathcal{S}$ to find the best solution. In addition, often it is the case that the set S is not even a finite set (Aarts et al., 2003). There are multiple approaches to solving this problem, for example Iterative Improvement.
The idea of Iterative Improvement is not to calculate $\mathcal{C}(s)$ for all $s \in \mathcal{S}$ to find the optimal solution, but to begin with an initial solution, which does not have to be a good solution and may even be infeasible. Then, we try to find better solutions based on the initial solution (Lourenço et al., 2003). For instance, in our case we might try a solution that we achieve by slightly changing the order of actions in the schedule of the initial solution. Another thing that might work, is to add an extra action to the schedule.

We just named two examples of what we could formulate as *neighborhoods*, for which we have the following definition.

Definition 6.1 *A set* $\mathrm{N}(s)$ *is called a neighborhood of* s *if each and every solution in* $\mathrm{N}(s)$ *can be achieved by slightly changing solution* s *in a certain way*[7].

For instance, in the case of a neighborhood $\mathrm{N}_{order}(s)$ that changes the order of actions in a schedule, all schedules that contain the same actions as s (possibly in a different order) will be contained in $\mathrm{N}_{order}(s)$.

For convenience, let us also define the term *neighbor*.

Definition 6.2 *A solution* s' *is called a neighbor of* s *if* s' *is in some neighborhood of* s*, i.e., there is a neighborhood of* s*,* $N(s)$*, such that* $s' \in \mathrm{N}(s)$*.*

Now we can formulate Iterative Improvement (II) as follows.

First an initial solution is generated. Remember, this initial solution does not need to have a low cost. After that, we stochastically pick a neighbor from a randomly picked neighborhood of that initial solution, and check if that solution is better. If the neighbor is a better solution that we currently have, then we will start picking neighbors of that new solution. We continue this fashion to find better and better solutions (Vaessens et al., 1998).

[7] We will define the neighborhoods we use in the next section.

One might wonder what would happen if we end up with a solution that has no better neighbors, and here lies one of the biggest shortcomings of II. When this happens, it is not possible any more to find better solutions. In this case, a *local minimum* has been reached (see Figure 11a). This can be described more formally with the following definition.

Definition 6.3 *A solution s is said to be a local minimum if $\mathcal{C}(s)$ is smaller than $\mathcal{C}(s')$ for every neighbor s' of s. That is, there is no neighborhood of s that contains a solution with a lower cost than the cost of s.*

Reaching local minima is not considered a good thing, since the cost of these local minima often still are too high. Simulated Annealing tries to solve this problem by making it possible to *escape* these local minima.

6.2 Simulated Annealing

SA is inspired by metallurgy (Lu et al., 2008). In metallurgy, when a liquid slowly cools down, particles arrange themselves in a low energy ground state. With a slowly decreasing temperature, this ground state slowly lowers its energy. When the temperature approaches zero, the particles will be in its lowest energy state. The probability of the solid to be in a state with energy E is characterized by the *Boltzmann distribution*

$$\mathbb{P}(\mathbf{E} = E) = \frac{1}{Z(t)} e^{-Ek_B^{-1}T^{-1}} \tag{1}$$

where $Z(T)$ is a normalization factor. Furthermore, k_B is known as the *Boltzmann constant* and

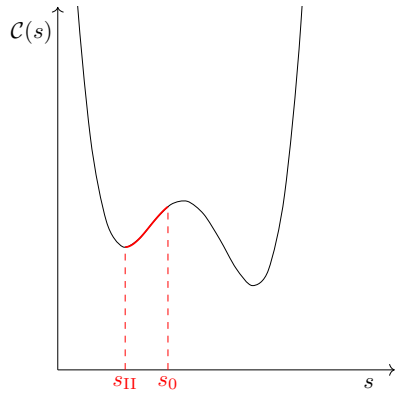
$$e^{-Ek_B^{-1}T^{-1}}$$

as the *Boltzmann factor*. After the cooling process has finished, this Boltzmann distribution will only have non-zero values for the lowest energy, since only minimum energy states can be present (Laarhoven et al., 1987).

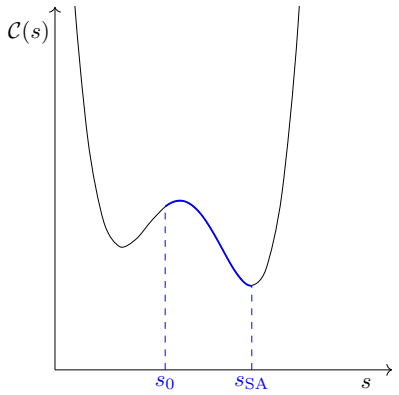
Based on this Boltzmann factor, Metropolis et al. (1953) proposed to use a *Monte Carlo* method to simulate the behavior of the solid as follows. Given the starting state of the solid, a small modification is made randomly. If the energy difference ΔE of that modification is negative, then a state with a lower energy has been found. Hence, the modification is favorable and thus the modification is accepted. However, if ΔE is positive, the modification is not favorable and is accepted with probability $e^{-\Delta E k_b^{-1} T^{-1}}$ (Laarhoven et al., 1987).

This Monte Carlo method, called Simulated Annealing, has not only been deployed in metallurgy but also has been deployed on many other diverse research topics like DNA folding (Diaconis et al., 2004), formation of defects in silicon (Lee et al., 2009) and calculating Feynman integrals (Filinov, 1986).

SA has as key feature that it makes escaping local minima possible for finding global minima (Nikolaev et al., 2010) or local minima with lower costs. That is, when using SA, some inferior solutions will be accepted stochastically like Metropolis et al. (1953) also accepted higher energy states stochastically. In this way, we can arrive at areas in the solution space that have lower local minima (see Figure 11b). This has as additional result that the outcome of SA is less dependent on the starting solution than it would be when using II (Anily et al., 1987).



a) A possible solution-path from s_0 to s_{II} when using II.



b) A possible solution path from s_0 to s_{SA} when using SA.

Figure 11: Graphical comparison between II and SA. In this case, we see that II, having the same starting solution as SA, arrives in a local minimum with a higher cost than the local (or even global) minimum where SA arrives.

In some literature, the acceptance of an inferior solution is called a *hill climbing move* (Jain et al., 1992). Such a hill climbing move is illustrated left from the local optimum in Figure 11b.

For deciding if an inferior solution will be accepted, we will use the following formula, based on Equation 1.

$$p(s_n, s_c, T) = \begin{cases} 1 & \text{if } \mathcal{C}(s_n) \leq \mathcal{C}(s_c) \\ e^{(\mathcal{C}(s_c)-\mathcal{C}(s_n))T^{-1}} & \text{otherwise} \end{cases} \tag{2}$$

In this calculation, $\mathcal{C}(s_c) - \mathcal{C}(s_n)$ is the cost difference between s_n, our potential new solution and s_c, our current solution and we have the *temperature* T. The result of this calculation represents the chance for accepting solution s_n.
Note that this formula follows almost directly from Equation 1 if we remove the normalization factor, take $k_B = 1$ and the cost as the energy, i.e., $E = \mathcal{C}(s_n) - \mathcal{C}(s_c)$.

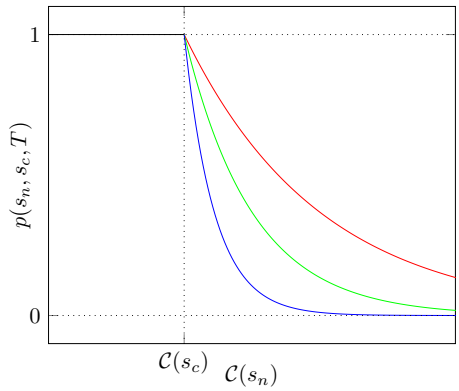


Figure 12: Plot of the chance of acceptance function, $p(s_n, s_c, T)$, for 3 different temperatures T. The red line corresponds to the highest temperature, the green line to a lower temperature and the blue line corresponds to the lowest temperature. For values of $\mathcal{C}(s_n)$ smaller than $\mathcal{C}(s_c)$, the outcome of $p(s_n, s_c, T)$ is 1 for all different temperatures.

If $\mathcal{C}(s_n) \leq \mathcal{C}(s_c)$, then $p(s_n, s_c, t) = 1$ for all temperatures T. Hence, a better solution is always accepted. The higher the temperature, the more likely an inferior solution will be accepted and the higher the cost of an inferior solution, the less likely it will be accepted. This is illustrated in Figure 12.

After we have calculated $p(s_n, s_c, T)$, we take a random number x from $\mathcal{U}(0, 1)$, the uniform distribution on the closed interval $[0, 1]$. If x is smaller[8] than $p(s_n, s_c, T)$, then we will accept solution s_n. In contrast, if x is not smaller[8] than $p(s_n, s_c, T)$, we discard s_n and we generate a new solution s_n. These steps are repeated many times.

In addition to only paying attention to the cost of a solution, we can also check if a solution is *feasible*. If a solution is *infeasible*, then it will not be accepted. However, some constraints may be ignored during the first iterations and are only considered during the last iterations to ensure feasibility for the final solution.
This allows infeasible solutions to be accepted in the beginning. In this way, we can also use infeasible starting solutions that have no feasible neighbors as an initial solution. This makes generating a starting solution less difficult.

While running this SA approach, the temperature of the annealing process is *cooled* each k iterations by multiplying it with a *cooling constant* $0 < \alpha < 1$. In this way, the value of T decreases to zero exponentially. The consequence is that, over time, inferior solutions will be accepted with a lower probability and, as such, we probably still arrive in local minima. The idea is that these local minima will have much lower costs.
Normally, α is chosen to be not much smaller than 1 such that the cooling process becomes really slow, as it is assumed to be by Laarhoven et al. (1987).

This SA approach, where we take T_0 as *starting temperature* and stop after n iterations, can be formulated as is done in Algorithm 1.

Algorithm 1: Simulated Annealing

```
1  Create a starting solution s_c.
2  Set the temperature T to the starting temperature and α to the cooling constant.
3  Set n to number of total steps that will be performed and set j to zero.
4  while j < n do
5  |  Stochastically select a neighborhood of s_n.
6  |  From that neighborhood, pick a random neighbor s_n.
7  |  Calculate p_n = p(s_n, s_c, T), the chance of acceptance of the selected neighbor.
8  |  Pick x randomly from U(0, 1).
9  |  if x < p_n and s_n is feasible then
10 |  |  Accept solution s_n by setting s_c to be s_n.
11 |  end
12 |  Add one to j.
13 |  If j is a multiple of k, cool the temperature by multiplying T with α.
14 end
15 Give s_c as result.
```

In this procedure, we start by generating the solution s_c and setting the temperature T, cooling constant α and the number of total steps n. We also keep track of how many iterations already are completed stored by the variable j. Then we select a random neighbor and either accept it or keep our current solution. This decision, made in lines 7 to 9, is based on Equation 2 and a feasibility check. This step is concluded in lines 12 and 13 by adding 1 to j and by multiplying T by α if another k iterations have finished. These steps are repeated n times. The output of this procedure will be an optimized solution s_n.

[8] Note that it does not matter whether we check here if it is strictly smaller or just smaller, since we picked x from a continuous distribution.

Now, we will show a theoretical aspect of SA. Namely that for hill-climbing moves (moves where solutions with higher costs are accepted), the probability of accepting a solution with score difference Δ is the same for any number of steps in between. We will formulate this as a theorem and give a proof.

Theorem 6.1 (Transitive aspect of SA) *Assume that the temperature T is fixed. Also assume that we have a hill climbing move from s_0 to s_1 and from s_1 to s_2. It is not relevant if we would skip solution s_1 or not for the chance that s_2 would be accepted, starting from solution s_0.*

Proof: We have score differences $\mathcal{C}(s_1)-\mathcal{C}(s_0)$, $\mathcal{C}(s_2)-\mathcal{C}(s_0)$ and $\mathcal{C}(s_2)-\mathcal{C}(s_1)$. Because we consider hill climbing moves, these are all positive.
Now, the chance that s_1 will be accepted from solution s_0 is

$$p(\mathcal{C}(s_1),\mathcal{C}(s_0),T) = e^{(\mathcal{C}(s_0)-\mathcal{C}(s_1))T^-1}$$

and the chance that s_2 will be accepted from solution s_1 is

$$p(\mathcal{C}(s_2),\mathcal{C}(s_1),T) = e^{(\mathcal{C}(s_1)-\mathcal{C}(s_2))T^{-1}}.$$

The chance of accepting s_1 from s_0 and s_2 from s_1, is the product of the two.

$$\begin{aligned} & e^{(\mathcal{C}(s_0)-\mathcal{C}(s_1))T^{-1}} \cdot e^{(\mathcal{C}(s_1)-\mathcal{C}(s_2))T^{-1}} \\ = \; & e^{(\mathcal{C}(s_0)-\mathcal{C}(s_1))T^{-1}+(\mathcal{C}(s_1)-\mathcal{C}(s_2))T^{-1}} \\ = \; & e^{(\mathcal{C}(s_0)-\mathcal{C}(s_2))T^{-1}} \end{aligned}$$

which is exactly equal to $p(\mathcal{C}(s_2),\mathcal{C}(s_0),T)$, the chance of accepting solution s_2 from solution s_0.

Quod Erat Demonstrandum

By repetitive use of this proof, this argument can be made for any number of steps.

This property of SA results in the fact that it matters less how much the neighbors can change the solutions. The chance that a new solution will be accepted is not dependent on how many steps are needed to form that new solution.

7 Optimization

In Section 6, we explained SA. We talked about the cost function, feasibility the neighbors. In this Section, we explain how our cost function is built and what neighbors we use.

7.1 Feasibility

We will define our feasibility check in such a way that tracks will never contain too many cars and that any departure delayed more than three minutes is not allowed. If we name the feasibility check of schedule s feasible(s), then it can be formulated as follows.

$$\text{feasible}(s) = \begin{cases} \text{no} & \text{track capacity issues or a departure delayed more than 3 minutes} \\ \text{yes} & \text{otherwise.} \end{cases}$$

These feasibility checks will only be done at the end of the SA. The idea behind this is to give the SA enough *breathing space* to find good solutions. In the last part of the runs, it is made sure that the solutions will be feasible too.

7.2 Cost function

Our cost function $\mathcal{C}$ is quite complicated and will computationally be the heaviest part of the program. While calculating the cost, the effect of each and every action on the yardstate and blockingstate must be determined. We will give a brief overview of our cost function. Our focus will lie in how it is structured.

First, we will go over how the number of incorrectly departed cars is calculated. For this, we will consider a train serving destinations corresponding to the colors green and blue (see Figure 13). Cars should be sorted into that order in this train.

a) Here, cars 1, 3, 5 and 6 have departed correctly and car 2 and 4 have not.

b) In this case we say that only car 1 and car 2 have correctly departed, since cars 3 and 4 are not in the correct group.

c) In this case, all cars have correctly departed.

d) Even though the green cars are missing, there only are correct departures in this train.

Figure 13: Some examples of what cars are counted to be a correct or incorrect departure. Like in Section 2, the cars have been color coded according to their destinations. In all cases the train serves the destinations with the colors green and blue, in that order as seen from the locomotive (from the right to the left) which is again colored red.

We count the numbers of incorrect cars in two steps.

In the first step, we count all cars that should not be in the train and remove them. In Figure 13a, this would be two cars: car 2 and 4. After this, we have the train depicted in Figure 13c.

In the second step, we check if the groups are in the correct order. The first group should be the green one, so we accept all green cars that are in the front and remove them from the train. The second group should be colored blue, so now, we accept all blue cars in the front and remove them. The train does not serve any other destinations, so the remainder of the cars are considered incorrect. They were not in the right order.

That counting incorrect trains is not trivial, can be shown by considering Figure 13b. One could say that only car 2 was incorrect because, if you remove that car, all other cars would be correct. There is nothing wrong with counting incorrect cars in this clever way, but we have chosen to do this like we described because it seems the most straightforward computationally.

The cost function $\mathcal{C}$ consists of the main objective, a cost for the actions, and penalties for constraints that the solution does not obey. Many parameters are used in this function. These are denoted with the letter $p_{\text{parameter}}$. Tweaking these values can greatly improve or reduce performance of our SA approach.

Some parameters for the penalties are made variable and change over time. We will denote these as $\hat{p}_{\text{parameter}}$. In the start, some of these parameters will be equal to zero and will be grown linearly each 1000 iterations. This again to give the SA breathing space.
As a result, we end up with a variable cost function which cannot be utilized without care. For instance, every 1000 iterations, we need to recalculate the cost of the current solution for the new parameters of the cost function, such that an unbiased comparison between the current and a new solution still can be made. Also, just like the temperature in the theorem in Subsection 6.2 was assumed to be constant, we also have silently assumed the cost function to be constant during hill-climbing moves. As a consequence, the transitive aspect of SA will not be true during these updates of the temperature and the cost function.

The cost function can be summarized in the following form.

$$\mathcal{C}(s) = \sum_{a_j \in s} \mathcal{C}_{action}(a_j) + \sum_{a_j \in s} \mathcal{C}_{yardstate}(Y_s(j)) + \mathcal{C}_{endstate}(Y_s)$$

in which $\mathcal{C}_{action}(a_j)$ is the cost of an action, $\mathcal{C}_{yardstate}(Y(j))$ is the cost or penalty for a forbidden yardstate and $\mathcal{C}_{endstate}(s)$ is the cost for cars that are parked on arrival or departure tracks at the end of the week.
We will start with the cost of the endstate, being the simplest part of the cost function.

The penalty $\mathcal{C}_{endstate}(s)$ is based on the number of cars and their location. If a car remains on the shunting yard, we give a penalty for that car. If a car is located on one of the arrival tracks or one of the departure tracks, we multiply this penalty by 100, as these locations are not designed for car parking. We will denote the set of arrival tracks, classification tracks and departure tracks as Φ, Θ and Ψ respectively. Now we can formulate this penalty as follows. Then penalty can be calculated using the following formula.

$$\mathcal{C}_{endstate}(Y_s) = p_{\text{carLeftOnYard}} \cdot \left(100 \cdot \left(\sum_{\phi \in \Phi} \overline{c_\phi} + \sum_{\psi \in \Psi} \overline{c_\psi}\right) + \sum_{\theta \in \Theta} \overline{c_\theta}\right)$$

where $\overline{c_\tau}$ denotes the number of cars on track τ.

By having the penalty for incorrectly departed cars higher than for cars remaining on the yard at the end of the SA run, it is not favorable to send cars away incorrectly to prevent this penalty.

For the yardstate, we only give a penalty if the cars do not fit on the tracks. Let us denote the length of all cars on a track τ as c_τ^l and the length of the track itself as τ^l, both measured in meters. Then

$$\mathcal{C}_{yardstate}(Y_s(j)) = \hat{p}_{\text{trackCapacity}} \cdot \sum_{(\tau, c_\tau) \in Y_s(j)} \max(c_\tau^l - \tau^l, 0).$$

The parameter $p_{\text{trackCapacity}}$ is zero at the start of the SA run and is grown linearly in the beginning and exponentially after that, ensuring feasibility of the final solution in respect to track capacities.

The remaining part of the cost function consists of the costs and penalties following from the actions. The cost of the actions is a variable p_{action} that is the same for each kind of action. This base cost, while it is made relatively small, has as result that unnecessary actions are removed from the schedule.
We only have penalties for arrival and departure actions. For the rest of the actions (a pull, roll-in or transfer) no penalties are given.

Arrival: A penalty per minute that an inbound train has to wait before it can drop off its cars at the arrival yard. This penalty is made variable. It is calculated as

$$\mathcal{C}_{arrival}(a) = \hat{p}_{\text{arrivalLate}} \cdot \text{number of minutes late.}$$

Departure: A penalty for cars that departed to the wrong destination or could have departed earlier. Because a 2-hour delay is not as bad as a 4-hour delay, we modelled this relative penalty by the following function (plotted for a late car in Figure 14). It is designed such that any delay gives at least a 50% penalty, compared to an incorrectly departed car. Each day, the penalty grows. After one day, it is 75%. After two days, it is 87.5%. Et cetera.

$$\text{relative penalty(car)} = \begin{cases} 1 & \text{if the car departed to incorrect destination} \\ 1 - \frac{1}{2^{d+1}} & \text{if the car has been delayed (for } d \in \mathbb{R} \text{ days)} \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

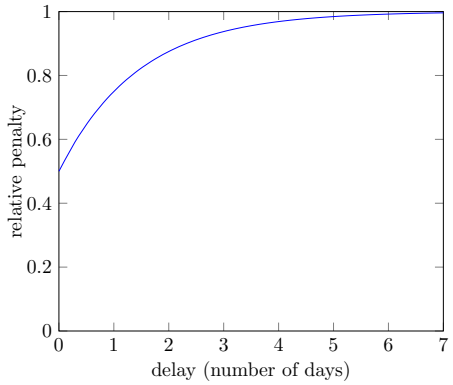


Figure 14: Plot of Equation 3 for a car that had a delay.

Also, a penalty is given for trains that departed too late. These penalties are again made variable and grow during the SA run.

The penalties for a departure come down to

$$\begin{aligned} \mathcal{C}_{departure}(a) = {} & \hat{p}_{\text{trainDepartureLate}} \cdot \text{train minutes late} \\ & + \hat{p}_{\text{wrongDeparture}} \sum_{\text{cars} \in \text{train}} \text{relative penalty(car)}. \end{aligned}$$

If we summarize the penalties of an action by $\mathcal{P}(a_j)$, the cost of an action can be formulated as

$$\mathcal{C}_{action}(a_j) = p_{\text{action}} + \mathcal{P}(a_j).$$

7.3 Neighborhoods

In the Section about LS, we stated that SA picks solutions from neighborhoods from s. We must define these neighborhoods. In our approach, we will have the following neighborhoods. For convenience, we have grouped some of the most closely related neighborhoods together.

Remove: Stochastically remove an action from the schedule. Arrival and departure actions may not be deleted.

Create: Add a new, randomly created action to the schedule. This could either be a pull action, a push action, a transfer action, or a roll-in action. Arrival or departure actions cannot be created since they directly follow from the data.

Order: Change the order of actions in the schedule by moving an action some places to the front or the back of to the schedule. See Equation 4 for a visualization of what neighbor is produced if we move the i'th action four places to the back for creating a new solution.

$$\begin{aligned} s_c &= (\ \ldots\ ,\ a_{i-1}\ ,\quad a_i\ ,\ a_{i+1}\ ,\ a_{i+2}\ ,\ a_{i+3}\ ,\ a_{i+4}\ ,\ a_{i+5}\ ,\ \ldots\ ) \\ s_n &= (\ \ldots\ ,\ a_{i-1}\ ,\ a_{i+1}\ ,\ a_{i+2}\ ,\ a_{i+3}\ ,\ a_{i+4}\ ,\quad a_i\ ,\ a_{i+5}\ ,\ \ldots\ ) \end{aligned} \tag{4}$$

Change: Stochastically pick an action from the schedule and modify that action. This neighborhood acts differently on each action. Here is a list of how it can change each type of action.

- Arrival: The arrival track on which the cars will arrive.
- Roll-in: The arrival track or one of the classification tracks that will be used.
- Pull-out: The classification track, the arrival track or the number of cars that will be pulled out.
- Departure: The number of cars that will be picked up or the track for the departure[9].
- Transfer: The classification track, the destination track or the number of cars that will be transferred.

The chances for picking the neighborhoods are 15% for the Remove, 23% for the create and 32% for the Order and 30% for the Change neighborhoods[10]. When a neighborhood is chosen according to those chances, a neighbor is picked randomly from that neighborhood. Because a roll-in can be changed in many more ways than, for example, a departure action, we boosted the chance of picking a roll-in action in the Change neighbor to 70%.

7.4 Starting Solution

We will construct our initial solution based on the arrivals and departures from the data.

We create an arrival action for each train arrival and pick the arrival track which has been empty the longest. If that track is too short to store all cars in that train, we randomly pick an empty arrival track that is long enough such that SA does not have to fix too many capacity constraints.

After each arrival, we plan a roll-in action to empty the arrival track. In this roll-in action, all cars are split into uninterrupted groups based on destination. Each group is rolled-in into a randomly picked classification track.

For each departure, we first add a transfer action from the classification track containing the most cars to a departure track that has been empty the longest. Finally, a departure action, taking all cars from that departure track, is created and added to the schedule.

[9] We must ensure that the new track does allow for a departure to the direction to which the outbound train will go.

[10] These chances seem to work best. However, better values can most certainly be found as will be mentioned in the Discussion.

7.5 Temperature

We have chosen a starting temperature $T_0 = 15$ and cooling constant $\alpha = 0.9998$ by which we multiply the temperature every 1000 iterations. We will do 15 million iterations in a SA run. The final temperature will be

$$T_0 \alpha^{\frac{15.000.000}{1.000}} = 15 \cdot 0.9998^{15.000} \approx 0.7466.$$

These temperature values are only relevant when you know what percentage of inferior solutions is accepted. With these temperature settings, approximately 20 percent of the inferior solutions will be accepted in the start, approximately 5.5 percent at 5 million iterations which decreases to 1 percent at 10 million iterations. Finally, at the end, only 0.25 percent of the inferior solutions will be accepted.

In this way, we are confident that we will produce good final solutions.

8 Deployment on Data

The model from Section 5, as well as the feasibility check, the cost function and the *PickNeighbor* operation as defined in Section 7 are implemented into a C# program. In this Section, we go over how we use the data acquired from Deutsche Bahn in our program and how we run our SA approach.

8.1 Data Overview

We will use data of two different weeks: week 15 in summertime, and week 45 in wintertime. We will run our SA algorithm for 19, 21, 23, up to 43 classification tracks. For each number of classification tracks we run the algorithm ten times to increase credibility of our results.

We will use the classification tracks with the lowest track numbers (see Appendix A). For instance, if we would allow our SA algorithm to use 3 classification tracks, the classification tracks with track numbers 105, 106 and 107 are used. If we allow our SA algorithm to use 10 classification tracks, the classification tracks with track numbers 105 up until and including 114 are available.

We have split up the data into weeks. Sunday morning comes as a natural splitting point since then, Deutsche Bahn does not handle any trains (no arrival checks, no roll-ins, etc.) and at weekend, there are few trains that arrive or depart. However, this splitting point has some shortcomings. For instance, trains that arrive at Saturday night are not handled before Sunday noon and some cars that depart at Sunday evening were formed before Sunday morning. Therefore, we split the arrivals at Saturday midnight and departures at Sunday midnight.

In week 15, 114 trains have arrived with 2280 cars and 138 trains have departed with 2335 cars. However, some cars do not leave in the same week and we assume the yard will be empty at the beginning of the week. Hence, we cannot match every arriving car to an outbound train. Also, some data on destinations of wagons was missing in the data, presumably because the destination was not known at the time of arrival. In the same way, some destinations have changed after the cars have arrived at the Kijfhoek.

Week	**15**	**45**
Arrived trains	114	120
Departed trains	138	124
Arrived cars	2280	2272
Departed cars	2335	2126

Table 1: Number of arrived and departed trains and cars of simulated weeks.

8.2 Matching Techniques

To be able to match more arrived cars to departing trains, a few matching techniques are used (see Table 5). This subsection can be skipped by readers who are not interested in how the data from Deutsche Bahn is converted and used in our SA approach.

The matching technique *WagonID & dest.* matches an arrived car to an outbound train if the arrived car has departed with that outbound train and has not changed destinations in between. In *WagonID & diff. dest.* and *WagonID & retr. dest.*, the destination of the car has changed or was missing in the data at arrival ór at departure. These cars are matched to an outbound train if they were in that outbound train. In that case, the destinations at arrival and departure are made the same.
The reason why we care so much about having matching destinations, is that the destination of a car dictates its global position in the train. It has to be in the correct destination group.

With above matching strategies, we found a departure for around 2/3 of the arriving cars. Each of these matches is proper: a car is only matched with an outbound train if that car has departed with that very same train. Because this fraction was quite small, the simulations would not reflect the actual capacity of Kijfhoek. That is why we increased the fraction to approximately 5/6 by matching on destination while leaving out the WagonID. This means that cars are also matched with an outbound train if the train served the cars destination. This has as result that, in our simulations, cars can depart earlier that they did according to the data.

Week	**15**	**45**
WagonID & dest.	1313	1242
WagonID & diff. dest.	74	108
WagonID & retr. dest.	77	18
WagonID & virt. dest.	96	74
Total WagonID	1560	1442
Extra on destination	327	507
Total matched	1887	1915
Total unmatched	393	357

Table 2: Matched number arriving cars to outbound trains using different matching strategies.

The remaining cars that were not matched, may not depart with any train and must remain on the shunting yard.

8.3 Hardware and metrics

The SA runs are divided over two computers. One with a Ryzen 7 2700x and 32 GB RAM, and one with 8 GB RAM accompanied by an Intel i-5 4670k. Both CPUs are clocked at 4Ghz. Stopping the program not after a certain amount of time, but after 15 million iterations, ensures that performance is not affected by what hardware is used. With this setup, one SA run takes about 4 hours.

9 Results

Before we go over the main results, we first show a *path* that a SA run can visit in Figure 15 to give some insight in how it performs. It is visible that the first 5 million iterations greatly increases the number of correctly departed cars and that the solution was only infeasible mainly in the first 2.5 million iterations. The number of incorrectly departed cars stays greater than zero, until approximately 11 million iterations when the penalties for incorrect departures have become quite high.

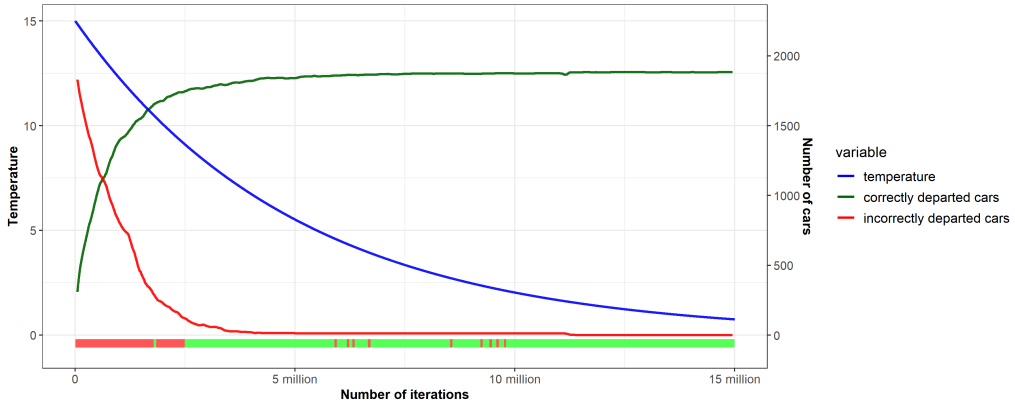


Figure 15: Some statistics over time during a SA run. The bar colored red and green displays whether the solution was feasible where red means infeasible and green means feasible.

In the next subsections, we analyse our solutions using five graphs. Because our main interest lies in the effect of renting out tracks to other parties than Deutsche Bahn, every graph has the number of available classification tracks on its x-axis. In the text, we act like we have inverted the x-axis, i.e., we start with the results of 43 available classification tracks, such that we can talk about a possible decline of performance when fewer classification tracks are available.

9.1 Correct car departures and delays

From our results we see that in week 15 (see Figure 16), when all classification tracks are available, 1883 cars (99.79%)[11] have correctly departed on average of which 1800 cars (95.4%) did not have any delay. This means that, on average, only 4 (0.21%) cars that should have departed were left at the yard at the end of the week. Two of the best SA runs found a solution where only one car did not make its departure and 1886 cars (99.95%) did.

Down to 29 classification tracks, these numbers are not much lower. With 29 classification tracks being available, on average 1881 (99.68%) cars had departed with a correct train and 1797 cars (95.2%). In the best SA run, 1884 cars (99.84%) had departed correctly.
Numbers decrease however when only 19 classification tracks are available. With this many tracks, 27 cars (1.4%) were left on the yard and only 1730 cars (91.7%) had departed without a delay.

The total hours of delay (the sum of the delays of every car in a final solution) also does not grow significantly from 43 tracks to 29 tracks, but does from 29 tracks to 19 tracks (see Figure 17). The average delay does not show a clear trend. For each number of tracks, it lies somewhere around 10 to 12 hours on average.

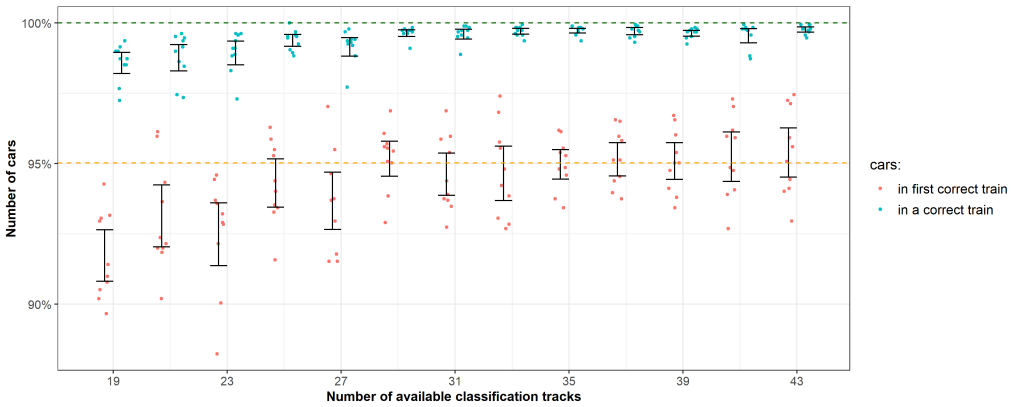


Figure 16: Number of correct departures of week 15. Dots represent individual SA run results. The 95% confidence intervals for the mean are visualized with black intervals. The green horizontal line displays the maximum number of correct departures and the orange horizontal is placed at 95%.

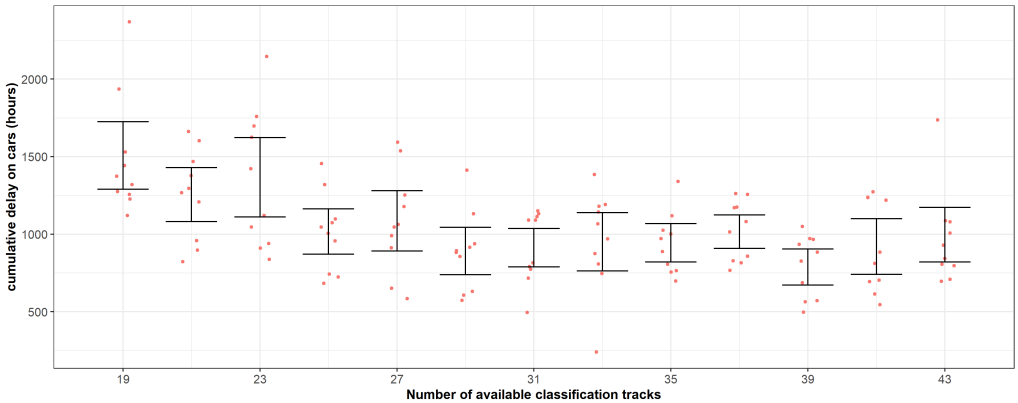


Figure 17: Average of total hours of delay for week 15. Dots represent individual SA run results. The 95% confidence intervals for the mean are visualized with red intervals.

[11] Remember that in week 15, 1887 cars have a departure.

The same can be seen from the results from week 45 (see Figure 18). The number of average correct departures declines from 1887 (98.45%)[12] with 43 classification tracks to 1864 (97.33%) with 29 classification tracks and finally becomes 1818 (94.93%) with 19 available classification tracks. The number of average departures that were on time are 1826 (95.6%), 1812 (93.6%) and 1703 (88.9%) for 43, 29 and 19 tracks respectively.

The total hours of delays of week 45 follows the trend visible in that of week 15 with a heavy outlier in the runs on 43 classification tracks with over 2200 hours of delay in total (see Figure 19).

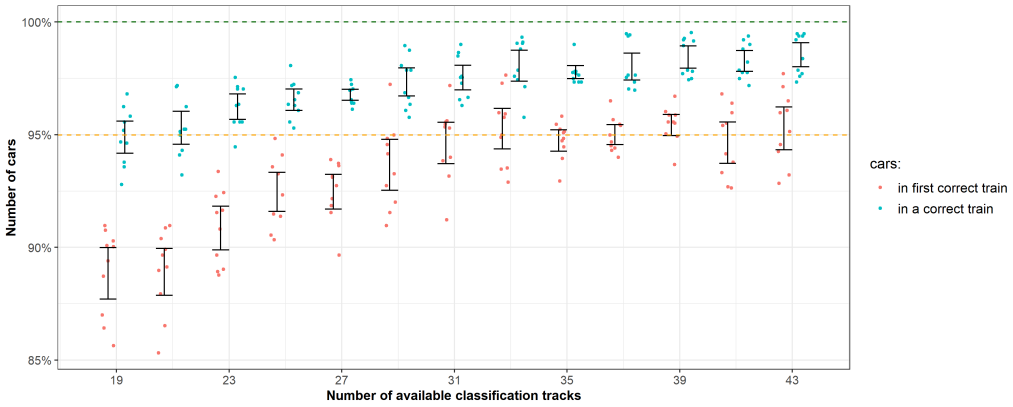


Figure 18: Number of correct departures of week 45. Dots represent individual SA run results. The 95% confidence intervals for the mean are visualized with black intervals. The green horizontal line displays the maximum number of correct departures and the orange horizontal is placed at 95%.

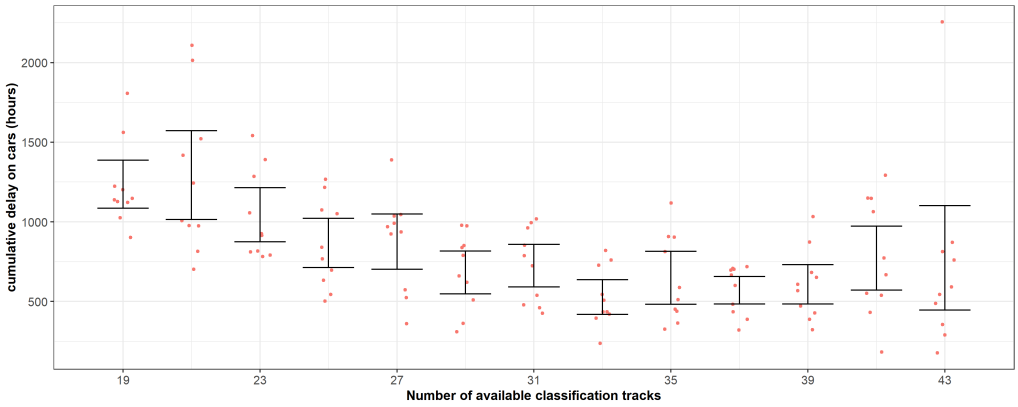


Figure 19: Average of total hours of delay for week 45. Dots represent individual SA run results. The 95% confidence intervals for the mean are visualized with red intervals.

[12] Recall that the number of cars linked to a departure in week 45 was equal to 1915.

9.2 Incorrect departures

One thing that could be considered more important than the number of correct departures, is the number of incorrect departures.

Our results of week 15 (see Figure 20) and week 45 (see Figure 21) tell us that, with more than 32 available tracks, only six of 60 SA runs contained incorrect departures for week 15 and eleven SA runs contained incorrect departures for week 45. This number starts to grow fast when fewer tracks are available. Solutions with zero incorrect departures still can be found when there are fewer classification tracks available, but the number of runs with zero incorrect departures declines heavily.

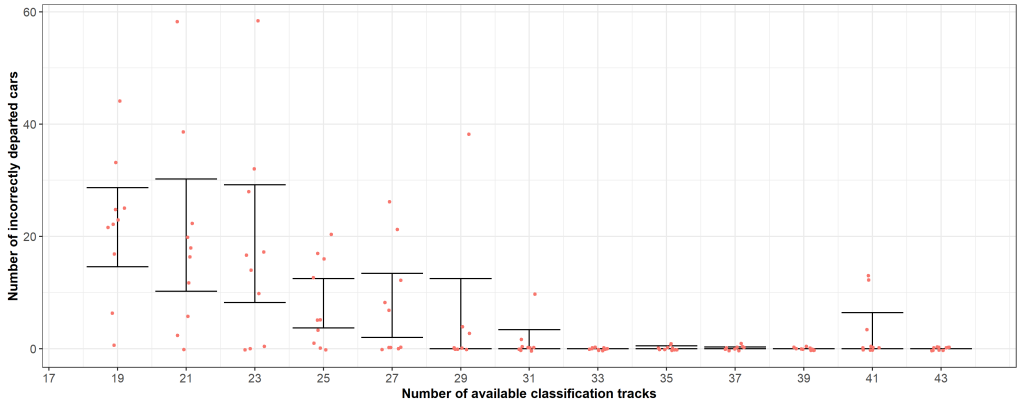


Figure 20: Number of incorrectly departed cars from week 15. Dots represent individual SA run results. The 95% confidence intervals for the mean are visualized with red intervals.

The same pattern is visible in the runs from week 45. Down to 33 available tracks, most runs are able to find solutions with no incorrect departures but when the number of classification tracks falls below 29, almost no solutions with zero incorrectly departed cars are produced (see Figure 21).

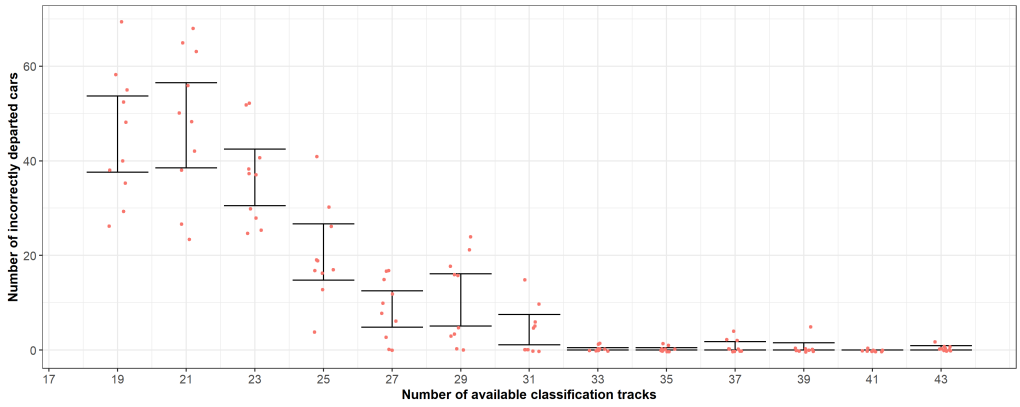


Figure 21: Number of incorrectly departed cars from week 45. Dots represent individual SA run results. The 95% confidence intervals for the mean are visualized with red intervals.

9.3 Train delays

As mentioned, arriving trains could be delayed for some minutes and departures for at most three minutes. Only nine of the 120 generated solutions were infeasible because a train departure was delayed more than three minutes in week 15. This was the case in only four of the 130 runs of week 45.
In week 15, all solutions had at least a total of 5-minutes of delays on inbound trains. In week 45, this was 6 minutes. The delays do not significantly increase or decrease when the number of available tracks decreases (see Figures 22 and 23).

Something that stands out is that every single solution had some delay in one of the arrivals. Further research showed that, according to the data of week 15, two trains from the south arrived exactly on the same time. This caused an unavoidable 5-minute delay on one of these arriving trains. In a similar way, there were unavoidable delays accounting for six minutes in week 45.

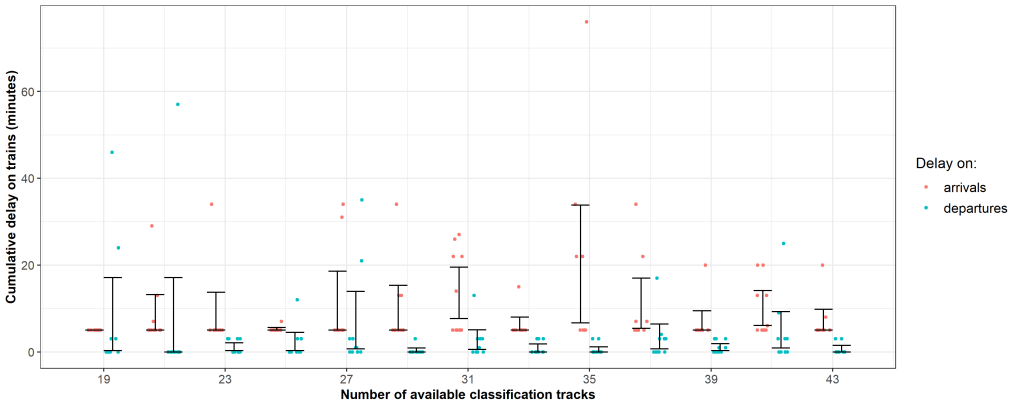


Figure 22: Number of actions performed in week 15. Dots represent individual SA run results. The 95% confidence intervals for the mean are visualized with red intervals.

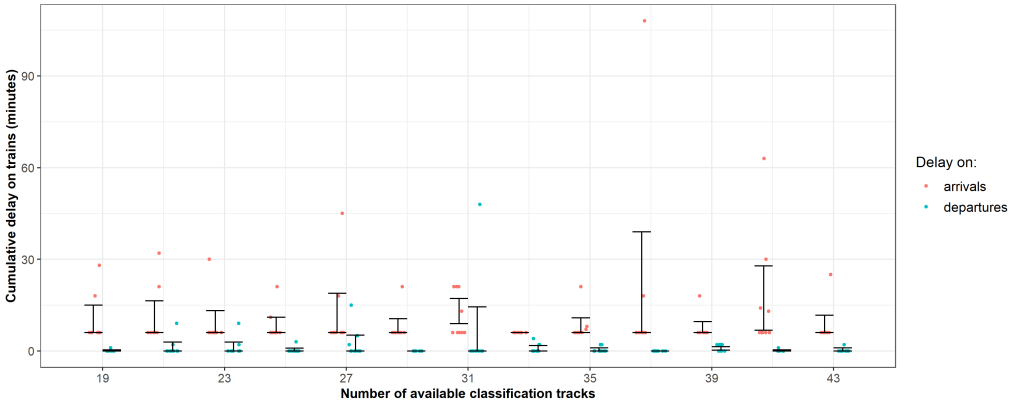


Figure 23: Number of actions performed in week 45. Dots represent individual SA run results. The 95% confidence intervals for the mean are visualized with red intervals.

9.4 Number of actions

If we look to the number of actions that are performed in the schedules produced by the SA runs, we see a steep increase from 31 tracks down (see Figures 24 and 25). Interestingly, simulations seem to either have around 500 or around 850 actions.

In week 45, there seems to be a slight decrease in the number of actions when fewer than 23 classification tracks are available, but this can't be said to be significant.

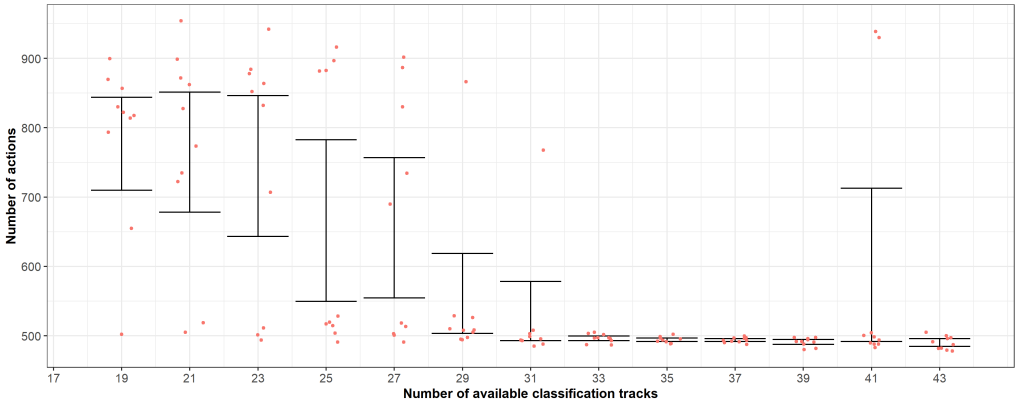


Figure 24: Number of actions performed in the final schedules of week 15. Dots represent individual SA run results. The 95% confidence intervals for the mean are visualized with red intervals.

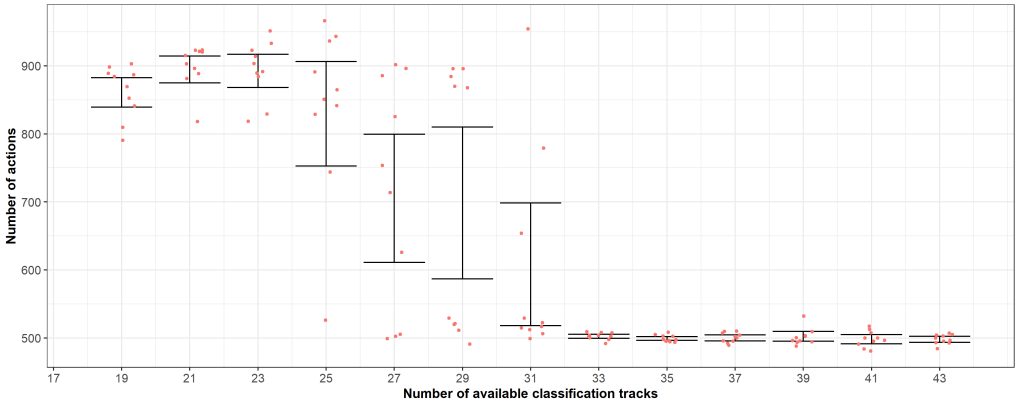


Figure 25: Number of actions performed in the final schedules of week 45. Dots represent individual SA run results. The 95% confidence intervals for the mean are visualized with red intervals.

10 Discussion

Conclusions

From the results, we can conclude that SA is suitable for solving the FSP problem and that we have acquired a powerful tool to produce shunting plans and analyze shunting operations at the shunting yard Kijfhoek.

In the model we devised, we saw that the shunting yards performance decreased when less than 29 classification tracks were available. The performance decrease is visible in a decreasing number of departures, increasing delays, an increasing number of incorrect departures and an increasing number of actions that are undertaken.
If we relate back to the question at hand, we conclude that being restricted on the number of classification tracks can greatly reduce performance of the shunting yard. However, when enough classification tracks are present, adding more classification tracks does not improve shunting performance significantly.

An interesting observation in the results is that the number of actions seems to either be around 500 or around 850 in solutions. Further investigation of the data showed that, in case of a solution with around 500 actions, the pull action was rarely used. When around 850 actions, there were almost as many pull actions as arrivals and in this way, the number of roll-ins also was doubled compared to solutions with around 500 actions. When fewer than 29 classification tracks are available, there seems to be a local minimum area that has twice as many roll-ins to pick up the correct cars for the departures. This is an interesting aspect in the solutions produced by this approach and could be further investigated.

Suggested research and extensions of the model

When making models, decisions are made about what to consider and what not. With the time limits associated with a bachelor's thesis, some things were not included in this preliminary research and are left for the future.

For instance, sorting to terminal is not modelled. When this is implemented, the effect of the extra movements and space needed for that type of sorting would reflect challenges of having limited infrastructure much better.
Also, we did not take into account the fact that there are cars that cannot be shunted over the hump. The effect of these restrictions remains to be investigated.

We have implemented the blockers quite conservatively. Parts of Kijfhoek were fully blocked for an action, when in fact more actions can be planned in the same time span. By doing more intensive research to the shunting operations at Kijfhoek, blockers can be implemented more realistically.

What could also improve this SA approach, is doing more research to the parameters used in the cost definition, the starting temperature and the parameter α. In addition, new neighbors can be invented and implemented, or we could change the chances of picking certain neighbors.

Crew and locomotive scheduling did not play a role in this thesis but these aspects most certainly can also be limiting factors.

We have made plans with data of car arrivals and departures. However, the exact order of cars in the train and arrival times are usually not known at the start of the week. This does not matter much for the topic we have researched in this paper, but it will matter if our approach is be used to create real shunting plans.
When this is the case, it must be investigated what data can be retrieved from the line plan and how dependent our approach is on the rest of the data. Also, some recovery strategies could be investigated to be able to use the shunting schedule even if a train comes in an hour late.

Something that will make our model more realistic, is to consider the cars that are parked at Kijfhoek at the beginning of the week. In this way, the planning horizon could theoretically be made infinite by repeated usage of the optimization approach for every week.

Now we have a tool for creating optimized shunting schedules for Kijfhoek, other topics can be investigated too.
For instance, we could research the effect of using the functionality of the double hump on how the shunting yard performs and if, in that case, fewer tracks are needed.

Before this happens, the program should be made more efficient from a computational perspective. In this thesis, we performed over 320 runs, each taking approximately 4 hours to complete. If we would want to use this approach to research effects of many different parameters, it would take quite long to finish all these computations. This makes this problem interdisciplinary, as efficient algorithms and data structures for the shunting problem must be researched.

References

Aarts, E. and J. K. Lenstra (2003). *Local search in combinatorial optimization*. Princeton University Press.

Adlbrecht, J., B. Hüttler, J. Zazgornik, and M. Gronalt (2015). "The train marshalling by a single shunting engine problem." In: *Transportation Research Part C: Emerging Technologies* 58, pp. 56–72.

Anily, S. and A. Federgruen (1987). "Simulated annealing methods with general acceptance probabilities." In: *Journal of Applied Probability* 24.3, pp. 657–667.

Bohlin, M., H. Flier, and S. Gestrelius (2012). "Optimal Freight Train Classification using Column Generation." In: *12th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems*, pp. 10–22.

Bohlin, M., H. Flier, J. Maue, and M. Mihalák (2011). "Track Allocation in Freight-Train Classification with Mixed Tracks." In: 20, pp. 38–51.

Bohlin, M., S. Gestrelius, and F. Khoshniyat (2013). "Simulation of planning strategies for track allocation at marshalling yards." In: *WIT Transactions on Modelling and Simulation* 55, pp. 465–475.

Boysen, N., M. Fliedner, F. Jaehn, and E. Pesch (2012). "Shunting yard operations: Theoretical aspects and applications." In: *European Journal of Operational Research* 220.1, pp. 1–14.

Broek, R. van den (2016). "Train Shunting and Service Scheduling: an integrated local search approach." MA thesis.

Cicerone, S., G. D'Angelo, G. Di Stefano, D. Frigioni, and A. Navarra (2009). "Recoverable robustness for train shunting problems." In: *Algorithmic Operations Research* 4.2, pp. 102–116.

Diaconis, P. and J. Neuberger (2004). "Numerical Results for the Metropolis Algorithm." In: *Experimental Mathematics* 13.2, pp. 207–213.

Filinov, V. (1986). "Calculation of the feynman integrals by means of the Monte Carlo method." In: *Nuclear Physics B* 271.3. Particle Physics, pp. 717–725.

Freling, R., R. Lentink, L. Kroon, and D. Huisman (2005). "Shunting of passenger train units in a railway station." In: *Transportation Science* 39.2, pp. 261–272.

Garey, M., D. Johnson, and L. Stockmeyer (1974). "Some simplified $\mathcal{NP}$-complete problems." In: *Proceedings of the sixth annual ACM symposium on Theory of computing*, pp. 47–63.

Gatto, M., J. Maue, M. Mihalák, and P. Widmayer (2009). "Shunting for dummies: An introductory algorithmic survey." In: *Robust and online large-scale optimization*. Springer, pp. 310–337.

Gestrelius, S., F. Dahms, and M. Bohlin (2013). "Optimisation of simultaneous train formation and car sorting at marshalling yards." In: *5th International Seminar on Railway Operations Modelling and Analysis RailCopenhagen*.

Jaehn, F., J. Rieder, and A. Wiehl (2015). "Minimizing delays in a shunting yard." In: *OR Spectrum* 37.2, pp. 407–429.

Jain, P., P. Fenyes, and R. Richter (1992). "Optimal Blank Nesting Using Simulated Annealing." In: *Journal of Mechanical Design* 114.1, pp. 160–165.

Kim, N. and B. Van Wee (2009). "Assessment of CO2 emissions for truck-only and rail-based intermodal freight systems in Europe." In: *Transportation planning and technology* 32.4, pp. 313–333.

Kiprop, V. (2018). "The busiest cargo ports in Europe." In: *WorldAtlas.[Mar. 1, 2018] http://worldatlas.com/articles/the-busiest-cargo-ports-in-europe. html*.

Krüger, N., I. Vierth, and F. Roudsari (2013). *Spatial, temporal and size distribution of freight train delays: evidence from Sweden*. Centre for Transport Studies.

Laarhoven, P. van and E. Aarts (1987). "Simulated annealing: theory and applications." In: *Mathematics and its Applications* 37.

Lee, S., R. Bondi, and G. Hwang (2009). "Formation and structure of vacancy defects in silicon: Combined Metropolis Monte Carlo, tight-binding molecular dynamics, and density functional theory calculations." In: *Physical Review B* 80.24, p. 245209.

Lourenço, H., O. Martin, and T. Stützle (2003). "Iterated local search." In: *Handbook of metaheuristics*. Springer US, pp. 320–353.

Lu, Z., L. Xia, and A. Jantsch (2008). "Cluster-based simulated annealing for mapping cores onto 2D mesh networks on chip." In: *2008 11th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems*. IEEE, pp. 1–6.

Metropolis, N., A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller (1953). "Equation of state calculations by fast computing machines." In: *The journal of chemical physics* 21.6, pp. 1087–1092.

Nikolaev, A. and S. Jacobson (2010). "Simulated annealing." In: *Handbook of metaheuristics*. Springer, pp. 1–39.

Papadimitriou, C. (2003). *Computational complexity*. John Wiley and Sons Ltd.

Swart, S. (2012). "Aerial photo of Kijfhoek." In: *Fotografie Siebe Swart*.

Ullman, J. (1975). "NP-complete scheduling problems." In: *Journal of Computer and System sciences* 10.3, pp. 384–393.

Vaessens, R., E. Aarts, and J. K. Lenstra (1998). "A local search template." In: *Computers & Operations Research* 25.11, pp. 969–979.

Ven, A. van de, Y. Zhang, W. Lee, E. Eshuis, and A. Wilbik (2019). "Determining Capacity of Shunting Yards by Combining Graph Classification with Local Search." In: *Proceedings of the 11th International Conference on Agents and Artificial Intelligence*. Vol. 2. SciTePress, pp. 285–293.

Appendix A Track Lengths

Track number	Track length	Allows for departure south
105	619m	Yes
106	616m	Yes
107	642m	Yes
108	631m	Yes
109	593m	Yes
110	563m	Yes
111	565m	Yes
112	717m	Yes
113	649m	Yes
114	606m	Yes
115	617m	Yes
116	587m	Yes
117	586m	Yes
118	610m	Yes
119	606m	Yes
120	526m	Yes
121	507m	Yes
122	508m	Yes
123	483m	Yes
124	483m	Yes
125	514m	Yes
126	549m	Yes
128	510m	No
129	509m	No
130	543m	No
131	544m	No
132	545m	No
133	546m	No
134	589m	No
135	712m	No
136	712m	No
137	678m	No
138	678m	No
139	646m	No
140	646m	No
141	687m	No
142	690m	No
143	753m	No
144	672m	Yes
145	674m	Yes
146	649m	Yes
147	651m	Yes
148	694m	Yes

Table 3: IDs, Lengths (according to system *Kijfdis*) and if the track allows for a departure southbound of classification tracks.

Track number	Track length
203	674m
204	689m
205	776m
206	830m
207	818m
208	779m
209	778m
210	799m
211	776m
212	782m
213	747m
214	665m
215	653m
216	653m

Table 4: IDs and lengths (according to system *Kijfdis*) of arrival tracks.

Track number	Track length
151	737m
152	741m
153	736m
154	763m
155	753m
156	756m
157	741m
158	743m

Table 5: IDs and lengths (according to system *Kijfdis*) of departure tracks.

Appendix B Terminology

English	**Dutch**
Shunting Yard	Rangeerterrein
Classification Track	Putspoor
Classification Bowl	Put
Rolling-in	Heuvelen
Transferring	Omhalen

Table 6: Translation of terminology from literature (English) to terminology used at Deutsche Bahn (Dutch).