Universiteit Utrecht

Department of information and computing sciences

Disruption Management on Shunting Yards with Tabu Search and Simulated Annealing

Master Thesis

Author
E.A.S. Onomiwo
Utrecht University

First Supervisor
Dr. H. Hoogeveen

Second Supervisor
Dr. M. van den Akker

Supervisor NS
MSc R. van den Broek

August 2020

Abstract

Trains are cleaned and inspected regularly to keep passengers happy, and costs low by preventing accidents. These services are done within a shunting yard based on a schedule created by the Dutch Railways. In practice, the execution of a schedule often deviates because a train arrives too late, or arrives with carriages in an unexpected order. If these disruptions break the schedule in any way, it will have to be corrected manually. We propose a local search technique to aid the adjustment of the schedule, while also aiming for the resulting schedule to be similar to the original. We introduce a score for similarity based on comparing the machinist schedules.

Preface

This thesis is submitted in partial fulfilment of the requirements for the degree of Master of Science in the Department of Information and Computing Sciences in the Graduate School of Natural Sciences.

The project was performed at the request of Dutch Railways, where I undertook an internship for the first nine months of working on this thesis, on which I worked on from September 2019 to September 2020. At the start, they mentioned that no prior research was done and that it might be a challenging problem; they were right.

I did have problems getting started, and I would not have made it this far if there were no excellent people to guide me. Special thanks to my supervisors: Han Hoogeveen, Marjan van den Akker, and Roel van den Broek for guiding me. Furthermore, thanks to the lovely people from the NS R&D department, who are always fun to talk to and were willing to help out. Finally, thanks to friends, family and God who supported me with their positivity, and granted me the determination to see this project through.

Please enjoy reading this thesis,

Zino Onomiwo

August 2020

Contents

Chapter 1

Introduction

The Dutch Railways (NS) is the biggest railway company in the Netherlands, each day transporting around a million passengers. Most will be travelling during peak hours in the morning and the evening.

During the off hours and especially at night, there will be fewer passengers, and consequently, the NS requires fewer trains on the rails. Trains that were used during the peak hours and are not required in the off hours need to be taken off the railway-network to make space. These trains are stored on shunting yards throughout the Netherlands. Yards also provide room for services to be performed on trains. These services can be: cleaning both inside and outside the train, inspections for safety, and small repairs. Some of these services can only be performed at specific locations within a yard, forcing trains requiring the same service to wait for each other.

Shunting yards tend to be located close to stations. As a result, they are found in urban areas with limited space available, and trains need to be spread throughout the yard efficiently. Another issue is the trains being constrained to tracks, hence limiting their flexibility: If a train is parked on a track, another train cannot drive over it.

Due to the many trains that need to be stored at a yard, they can not be thoughtlessly moved whenever they want or be parked wherever they want. Doing so could cause complications for future trains, causing extreme delays. Therefore, a schedule is created, ensuring that every service can be completed in time and no train will be blocked by other trains. This schedule results in every train being able to depart on time.

The schedule describes the execution of services and movements. It dictates how the trains move through the yard and at what times. It prevents the trains from conflicting with each other, and the required services are performed. It might be possible that the composition of some trains needs to change between their arrival and departure. This will be included in the schedule by telling trains when and where to split or combine.

Before such a schedule can be created, we need information regarding the trains that will be arriving and departing at a yard. This information is known in advance and is the deterministic input for a schedule. It consists of the composition of arriving and departing trains, combined with their arrival and departure times. Furthermore, each arriving train contains a list of services that need to be performed.

The NS has a team of planners who create the schedule based on the deterministic input. Because of the constraints present on a shunting yard, this is no easy task.

Moreover, the number of passengers is increasing every year, and soon the number of trains will not be able to keep up with passenger growth. This means that the NS will have to add more trains to meet demand.

Consequently, more and more trains will be found in the service yards, resulting in higher capacity utilization and vastly increasing the difficulty of creating new schedules. The NS is already expending effort to create new tooling for planners to help them with making decisions to increase the speed of creating new schedules and improve the quality. Computationally generating these schedules is a difficult problem, as it is similar to several known NP-hard problems such as Resource-Constrained Project Scheduling Problem; Open Shop Scheduling Problem or Train rearrangement[1].

As a consequence of executing the schedule in the real world, deviations during the execution will frequently occur, rendering the schedule infeasible due to conflicts. Conflicts can range from trains departing too late, to two trains being scheduled to drive through each other, which would lead to a crash.

A schedule containing conflicts is unusable. However, creating an entirely new one is not recommended due to the difficulty of creating one and time constraints. For this reason, the NS has a team of experts who manually adjust the plan to remove conflicts. While it is not as much work as generating a new schedule, it will still be a difficult task. Therefore, the NS aims to develop tooling to help the teams of experts with suggesting adjustments, increasing the speed and improving the quality of schedules in preparation for future disruptions.

This paper aims to develop and evaluate a local search algorithm that can be used to repair the schedule by adjusting it to remove conflicts by applying small changes in meaningful ways. Besides the removal of conflicts, another aim for the algorithm should be to keep as many features of the original schedule as possible, thereby having the adjusted schedule proceed in more or less the same manner as it did before. We will describe a method of measurement in Section 4.6.1.

In Chapter 2, an in-depth description of the problem is given. Then in the third chapter, a literature overview will be shown. In the fourth chapter, an approach is introduced, which will be evaluated in the fifth chapter. Lastly, a conclusion will be written in the final chapter, Chapter 6.

Chapter 2

Problem Description

In this chapter, we will provide an in-depth description of the individual components that play a part in the problem, followed by the problem itself.

2.1 Shunting yards

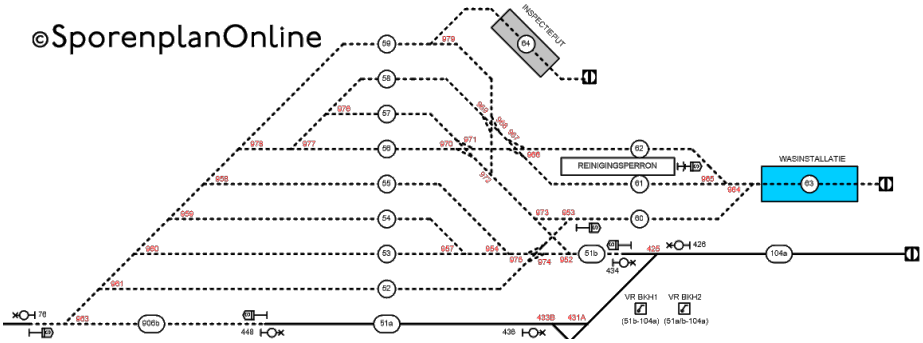


Figure 2.1: A Shunting yard at The Hague: Kleine Binckhorst

NS manages several shunting yards throughout the Netherlands. At these locations, they store passenger trains and perform necessary services. The layout of such a shunting yard consists of multiple tracks connected by switches, an example of which is seen in Figure 2.1. Trains enter these yards via one or more entrance tracks connected to the main railway network, which is managed by the Dutch task organization ProRail. Multiple trains enter via these tracks. For this reason, trains cannot park there.

The available space at a shunting yard is small. Hence, tracks cannot be made longer than the location allows, and the number of trains parked on a track is limited. This constrains the total amount of trains found at a shunting yard, limited even further by safety regulations. Tracks only support one traversing train at a time and require some leeway for safety before the next train can traverse the track. For this reason, they have to wait for each other, which limits their ability to reach their destinations in time.

Some yards contain electric switches. These can be flipped from a central command system. However, most of the yards contain manual switches. Whenever a machinist wants to use a switch, he has to walk and ensure their correct placement before traversing them. The benefit of electric switches is their flexibility; they allow a higher throughput of trains. However, installing and maintaining electric switches is difficult. The yard will be out of commission during their instalment, making it difficult for other locations.

If a train is not moving, it is either parked or serviced at a facility. Parked trains are wholly contained in a track and do not occupy switches. Facilities are connected to tracks where they can service trains standing on them. Most facilities are specialized for specific activities, and some facilities are the only ones found in a yard where their service can be performed. One is a washing complex for externally cleaning trains. Having many of these facilities is too costly, and they take too much space. Other facilities provide services such as internal cleaning, multiple kinds of inspections, or small repairs.

2.2 Trains

The NS uses various types of trains divided into different types, each having its properties and purposes. For example, the NS uses intercity trains, consisting of types: DDZ, ICM, and VIRM. They are used for longer distances while Sprinters, SLT or Flirt, are used for shorter distances. These specialities, together with different connectors, stop trains from using multiple types.

To accommodate variable amounts of passengers, a train is designed to be modular and consists of a combination of up to four train units of the same type. The units themselves are a combination of multiple carriages that cannot be divided. When two units are of the same type, they do not need to contain the same number of carriages. Types can be split up in multiple sub-types to denote that the units have the same type and can be connected, but contain a different number of carriages. For example, the Flirt unit-train appears in two variants: Flirt3, containing three carriages and Flirt4, containing four.

Figure 2.2: The upper train consists of two units of type ‘Flirt3‘ while the lower train consists of two units of type ‘Flirt4’. Both trains are of the same type, namely ‘*Flirt*‘ but are of a different sub-type.

Within a yard, a few limitations are placed on trains. One of these limitations is the requirement of being driven by a machinist at one end of the train. A machinist can only drive in the direction he is facing. If the train needs to drive the other way, the machinist will have to exit the train and walk to the other end.

2.3 Shunting schedule

A shunting schedule describes the activities in a yard for a time-window of up to 24 hours. Yards tend to have a high capacity utilization rate, making it

challenging to move trains around freely; they are easily blocked by one another. Consequently, having services complete, and depart on time becomes challenging and requires a good schedule to perform the work on a yard efficiently.

A schedule describes which trains arrive, their arrival time, and where they should go after entering the yard. It also shows a list of activities that occur, together with detailed information per activity. The schedule describes three parts: The *scenario, matching, and a list of planned activities* showing the starting time, finishing time, and locations where they should happen.

The scenario contains information about trains arriving and departing within the time-window applicable. It is the deterministic input from which the rest of the schedule is derived. The scenario itself consists of two smaller parts: the arriving trains and the departing trains. The first part is a list containing trains arriving at the yard. Each train has an arrival-time, location of arrival, which units it contains, and the type and sub-type composition. Individual units also have services that need to be performed during their stay at the yard. The second part contains a list of trains which will leave the yard and their departure times, together with a type/sub-type combination of units in which it should depart.

For any scenario, we work with the assumption that all units included within the arriving train will be part of some departing train. Consequently, no train is found in the yard at the start or end of the schedule.

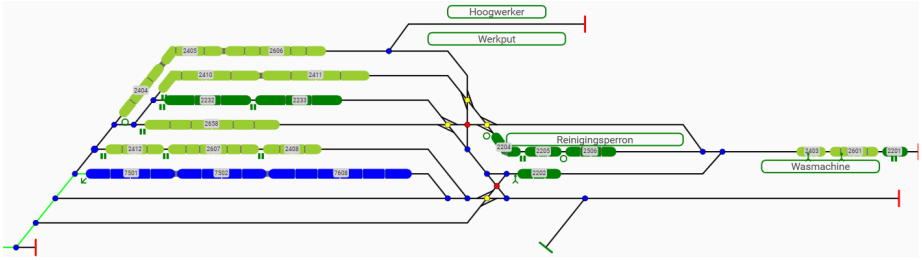


Figure 2.3: A yard with a high utilization

The second part of the schedule is the matching, and it uses the scenario as input. The matching assigns arriving units to departing trains. Unfortunately, the units within arriving trains do not necessarily have the same unit composition as their matched departing train; trains possibly have to be split and recombined before being able to depart.

The final part of the schedule is the list of activities that take place in the yard. The activities on this list include their starting time and the train units to which they are applied. The activities also represent services, movements, combinations, and splits, along with some information specific to the activity. In a period where none of the activities uses a train, the train will stay parked and waits for the next activity that uses it. The activities used in this paper will be further defined in the following paragraphs.

The first type of activity, the service activity, contains additional information regarding the performed types of services. The different types of services consist of, among other things, internal- and external-cleaning, inspections, and small repairs. It also shows which team of qualified experts performs the service and at which facility they perform it. In this paper, we assume that a team is always

available, preventing constraints regarding an insufficient workforce at the yard. The time it takes to complete the service depends on the type of service. Each service has its expected duration, which is used in the schedule as the amount of time the activity takes to complete.

The second type of activity is the movement of a train. The movement contains a starting track, a destination and, a route between the two. The person responsible for moving the train is a machinist, he ensures that the train can safely reach its destination by walking past every switch along the route and flipping it if necessary. After correcting the switches, the train starts to move, but can only move in the direction with the machinist in front. Consequently, this hinders the train from reaching certain tracks within the yard. If the train has to reach these tracks, then it has to reverse. The machinist exits the train, and he walks to the other end, but not before ensuring that all switches for the next part of the path are flipped correctly.

During the duration of the movement, tracks used in the route are reserved. This reservation blocks other trains from traversing them for safety. To approximate the time needed for a movement, we assign approximated times to flipping a switch and traversing tracks. Flipping a switch gets an approximation of 30 seconds, the traversal of a track, including the starting and destination tracks, is approximated to be 1 minute. The time to reverse the train depends on its length.

For example, a train traverses three tracks and two switches, after which it reverses and traverses four more tracks connected with three switches, completing a single movement. The first three tracks and two switches will take $3 \cdot 60 + 2 \cdot 30 = 240$ seconds. Reversing train takes the machinist 60 seconds. The second part of the journey takes $4 \cdot 60 + 3 \cdot 30 = 330$ seconds. It should be noted that four tracks are used in the second calculation: The track on which the train reverses is used twice to calculate the total duration of the movement. The total amount of seconds this movement takes equals to $240+60+330 = 630$ seconds.

The third and fourth activities are related to splitting and combining. The split activity describes how a single train splits into two, and a combine-activity shows how two trains combine into one.

2.4 Problem

The NS generates efficient schedules, one for every shunting yard. Activities may deviate from the schedule during the execution, potentially creating conflicts. A plan with conflicts cannot execute if it results in undesirable scenarios, like trains departing late or trains crashing.

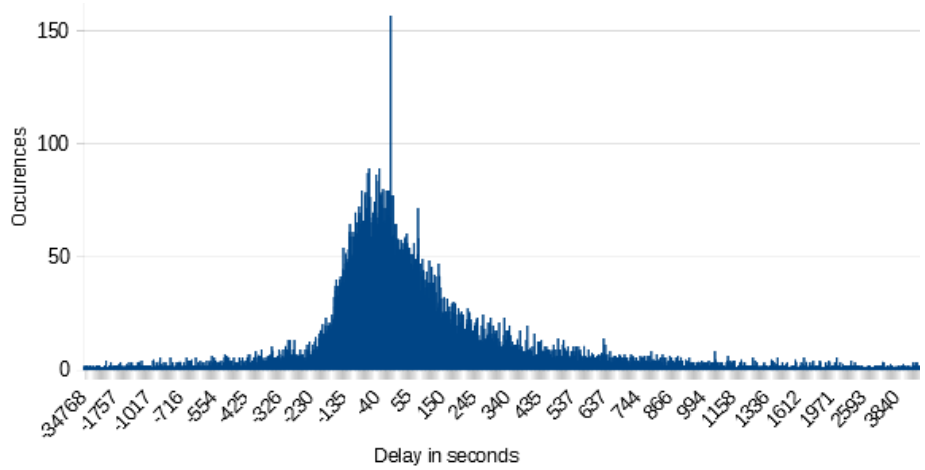


Figure 2.4: A plot of the disruptions occurring at Kleine Binckhorst, negative delay is the same as a train arriving early.

In this paper, we will model two types of disruption that may cause a deviation. The first disruption is a train arriving late or early. The disruption may force other activities to wait and have trains depart late. Figure 2.4 shows delay occurrences from Kleine Binckhorst collected over six months; most fall between being five minutes early and 10 minutes late. The next disruption happens when a train has a different sub-type order than was written in the scenario, causing trains to depart in a wrong order or to drive through each other.

We will give an example of a disruption occurring while executing a schedule. The example first shows a schedule executing without any disruption. After that, it shows the same schedule but with a disruption.

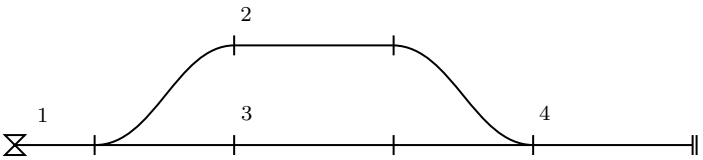


Figure 2.5: A small yard

A small schedule will execute on an example shunting yard, shown in Figure 2.5. Trains arrive and depart from the entrance track 1, and they are not permitted to park there. The tracks where trains can park are tracks 2, 3, and 4. These tracks are long enough to contain both trains used in the scenario.

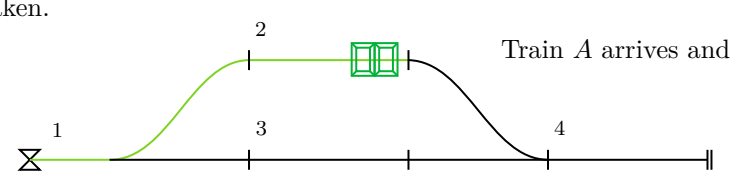
The scenario proceeds as follows: train A containing two units of sub-type a arrives at 9:00. After that, train B arrives at 9:30 with three units of different sub-types (b_1, b_2, b_3). Train A contains two units of type a, both the same sub-type, and train B units of type b with the different sub-types b_1, b_2 and b_3. Both trains park for some time before departing. Train B departs first at 15:00 and train A at 17:00.

Action	train	time
arrival	$A(a, a)$	9:00:00
arrival	$B(b_1, b_2, b_3)$	9:30:00
departure	$B(b_1, b_2, b_3)$	15:00:00
departure	$A(a, a)$	17:00:00

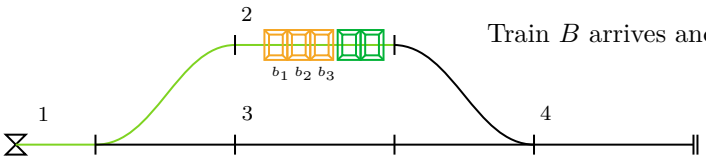
Train $A(a, a)$ arrives at 9:00 and moves to track 2. At 9:30, train $B(b_1, b_2, b_3)$ arrives and proceeds to move towards track 2.

The next action within the schedule is moving train B to track 1, after which it proceeds to depart. Moving the train to Track 1 takes approximately 2.5 minutes; to let the train depart at 15:00, it needs to start moving at 14:57:30. Two hours after the departure of B, train A makes preparations for its departure. Train A starts moving to track 1 at 16:57:30 before immediately departing at 17:00.

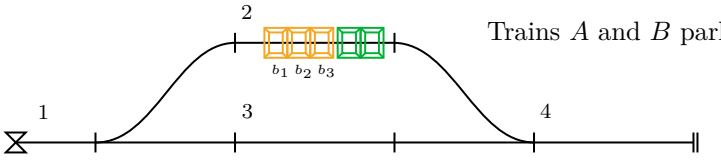
The next figures show the states at various stages on the shunting yard in chronological order. The green coloured lines represent that route a train has taken.



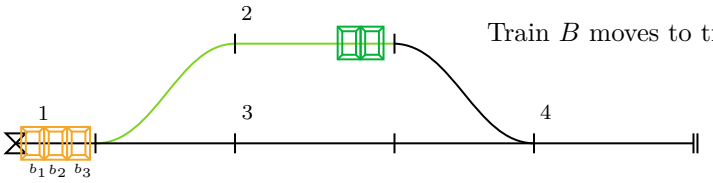
Train A arrives and moves to track 2



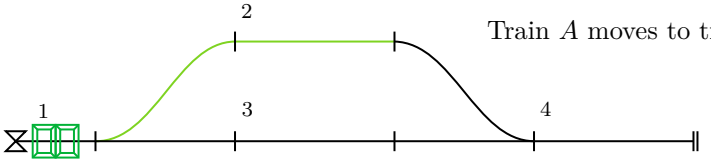
Train B arrives and moves to track 2



Trains A and B parked on track 2



Train B moves to track 1 and departs



Train A moves to track 1 and departs

A disruption during execution

Next, we take a look at a scenario where Train B arrives in an unexpected order: (b_2, b_1, b_3), but still has to leave in (b_1, b_2, b_3). The scenario requires the train to depart in the correct order, which is not possible if the original schedule is maintained. Therefore, the schedule has to be adjusted, starting from the point where Train B arrives at the yard. Various states occur in the yard, which is displayed chronologically in the images below.

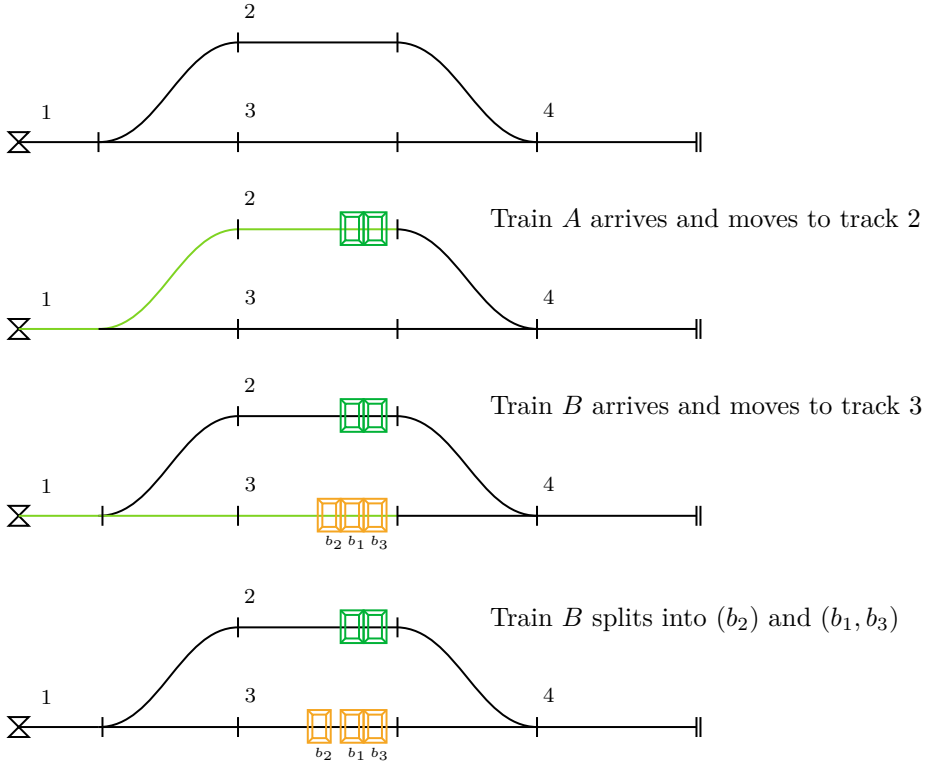
Once train B arrives, it moves to Track 3 instead. There it is immediately split into two smaller trains: (b_2) and (b_1, b_3). After the split, Train (b_1, b_3) immediately continues moving to Track 4, where it splits again, resulting in trains (b_1) and (b_3).

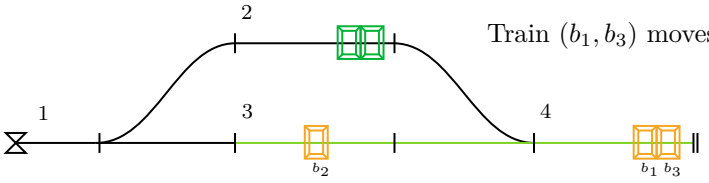
After the second split, Train (b_1) moves to Track 2, after which Train (b_2) moves next to Train (b_3) at Track 4. These two movements cannot be executed in parallel because they both make use of the same tracks.

Train (b_2) is now next to (b_3), and they combine into Train (b_2, b_3). The last step is to put Train (b_1) in front of it again. After moving Train (b_1) from Track 2 back to Track 4 it is added to Train (b_2, b_3) to create Train (b_1, b_2, b_3), now B', and replaces Train B for departure.

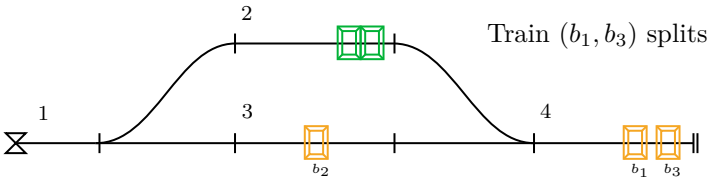
To prepare Train B' for departure at 15:00, it starts the move to Track 1 at 14:56 before departing.

Train A finds no change in actions related to it and proceeds according to the old schedule.

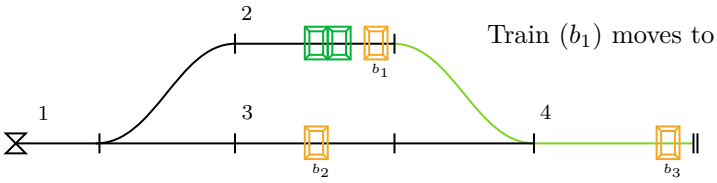




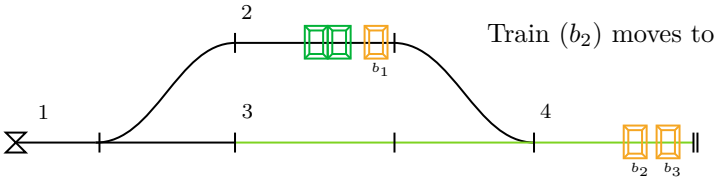
Train (b_1, b_3) moves to track 4



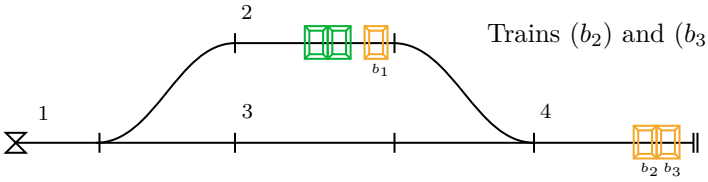
Train (b_1, b_3) splits into (b_1) and (b_3)



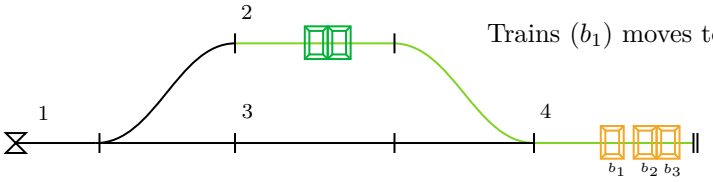
Train (b_1) moves to track 2



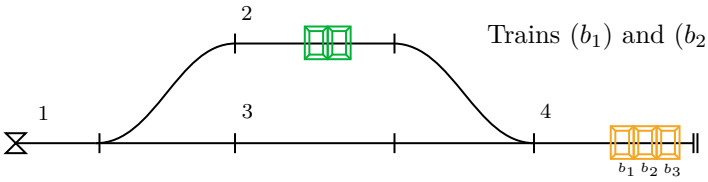
Train (b_2) moves to track 4



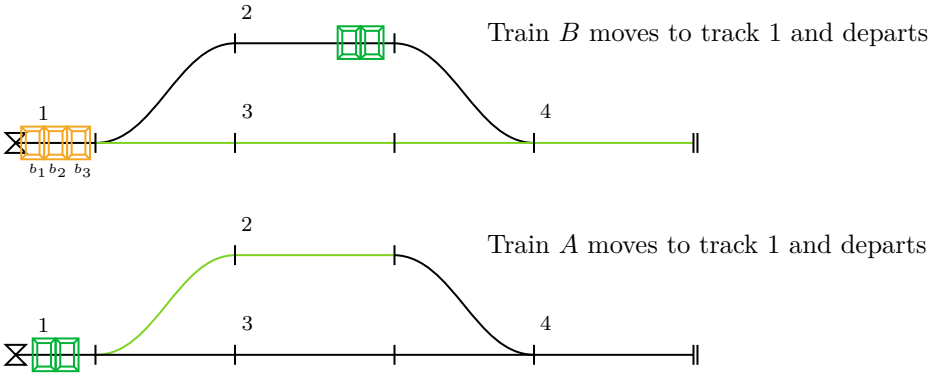
Trains (b_2) and (b_3) combine into one



Trains (b_1) moves to track 4



Trains (b_1) and (b_2, b_3) combine into one



Currently, the NS has dedicated teams of experts who manually adjust the shunting schedule to remove the conflicts. The NS wants to develop tools that assist this team of experts in decision-making and to speed up the process.

In this paper, we want to develop and test a local search based technique to be used in disruption management on shunting yards.

Research questions

Given an existing schedule for a shunting yard and a disruption that introduces conflicts in said schedule. We want to adjust the original schedule using a local search algorithm. This leads to the next three research questions:

- How fast does a local search solve a conflicted schedule.
- How similar are rescheduled solutions to the original schedule.
- How does a local search compare to van den Broek's[2] Simulated Annealing algorithm. Described in Chapter 3.

Chapter 3

Literature review

In this chapter, we will give a small overview of the work related to the train shunting problem and similar fields. As far as our knowledge goes, there is no literature written solely about repairing train shunting schedules after disruptions. For this reason, we also review the progress on disruption management in other fields.

Many industrial fields have an interest in disruption management, and most research has been done in the context of aircraft rescheduling. Clausen et al.[3] give an overview of techniques used in operation research within the airline industry. This generally includes the generation of flight schedules, the assignment of planes to flights and crew to aeroplanes. This has some resemblance to the train unit shunting problem, but aircraft have greater freedom in the air and on the ground than trains attached to the track. This paper will study disruption management in the context of shunting yards.

Clausen et al.[4] give a comprehensive overview of methods used for disruption management in Operation Research, specifically in the field of Aircraft Scheduling. One promising solution regarding the disruption of aircraft has been researched by Argüelo et al.[5] They attempt to reconstruct aircraft routings using a GRASP approach. This is done in two phases: The first phase generates a solution using a greedy heuristic and then utilizes a local search to find a feasible solution. The next phase uses local search and three neighbourhoods to improve the feasible solution and reach a local minimum. The three neighbourhoods are simple and involve moving a flight to different routes, exchanging flights between two routes or cancelling a flight. Using this approach manages to give good results close to the minimum within roughly 15 seconds on instances with 42 flights assigned to 16 aircraft.

A more comprehensive research on disruption management is done by Zhu et al.[6], with a focus on resource-constrained project scheduling. Their approach consists of a hybrid algorithm combining Mixed Integer Programming and Constraint Propagation. The MIP formulation iteratively gets tightened with the help of CP. They reasoned that adjusting precedences with the help of CP would give better results than calculating everything with a MIP approach. To test this method, a collection of instances containing 20 activities was evaluated. Many of which solved in roughly a minute.

A related field to disruption management is recoverable robustness. Ci-

cerone et al.[7] have been trying to define the robustness of optimizations. Their definition of a robust solution consists of the capability of such a solution to handle small disruptions using a robust algorithm. This is an algorithm that is able to compute a new robust solution from any disruption. To evaluate the quality, a *price of robustness* is constructed by taking the maximum ratio between the optimal and the solution created by the algorithm, of every problem instance. Cicerone et al. then proceed to apply these ideas to a simplified version of a shunting yard designed for freight-trains to demonstrate the effectiveness. During this evaluation, they show how robustness heavily affects performance.

Freling et al.[8] are the first ones who study the problem of creating a shunting schedule. They call it the Train Unit Shunting Problem (TUSP). Their primary focus is to match arriving with departing trains and assign them a track to park while waiting for departure. Their approach to generating new schedules is to split the problem into two steps and solve them sequentially. The first step is matching the arriving units to departing trains by using an ILP formulation. This formulation minimizes the number of splits and combinations needed by taking subsets of adjacent train units as blocks and assigning them to slots in the departing trains. The second step continues with the blocks defined earlier, assigning them a parking spot where they do not block the arrival or departure of one another. They solve this problem by using ILP with column generation. The master problem selects parking assignments which are generated by a dynamic programming subroutine. These parking assignments are made for individual blocks of trains and are independent of each other. Using this two-step approach, they manage to generate feasible solutions within a reasonably short amount of time. They show their ability to generate schedules for a generic weekday on a yard in Zwolle within an hour.

TUSP can be extended with the scheduling of services and assigning crew to these services. Van den Broek et al.[2] propose a novel approach using local search with simulated annealing as a metaheuristic. This is realized by modelling the activities taking place on a yard in a partial ordering schedule, which is iteratively improved by making small mutations. An initial solution is created in two small steps. The first step creates a feasible match assignment and proceeds to minimize the number of splits and combines necessary by running a simulated annealing algorithm with limited time. The second step continues with the matching and greedily assigns tasks, parking spots and routes between the tasks to the units. The resulting approach manages to generate a feasible schedule for a realistic scenario within roughly two minutes.

In further research, van den Broek et al[9] searched for strong robustness measurements, to measure the strength against possible disruptions. They introduced two new robustness measurements. The first is a path based measure. For the top K most relevant paths in a partial ordering schedule, a normal distribution is estimated which combines the distributions of service completion times. The second measure is a probability distribution based on the total makespan of the schedule.

These two measures were compared with existing measurements by way of a Monte Carlo simulation study. The findings suggest that the newly introduced measurements strongly correlate with the used performance statistics. Van den

Broek concludes that the measurement based on the makespan performs equally well as the path based measurement, but that the former measurement is preferred based on speed.

The Simulated Annealing algorithm made by van den Broek assumes that all information is known in advance and no disruptions occur. In an attempt to create a solution taking disruptions during execution into account, Peer et al.[10] devise a method to create a shunting schedule using deep reinforcement learning. Their aim is to introduce consistency to the schedules. For their tests, the neural network was trained on a yard containing nine tracks on problem instances with no services to be performed. Training their model consisted of 14 hours with $30,000$ problem instances, after which it developed a preferred way of solving problem instances. This preference introduced consistency to the schedules, which is convenient for repairing schedules with disruptions. Regenerating the schedule with an introduced disruption will return a new schedule with similar features as the original one. The deep learning model of Peer et al. [10] is an experimental approach and did not consider more complex yards. More complex problem instances will require more training data and more training time, whereas our approach does not have these requirements while also being more flexible.

Chapter 4

Local search

4.1 Local Search: Simulated Annealing and Tabu search

This chapter describes a local search algorithm capable of repairing schedules that became infeasible after a disruption occurred.

We assume that the reader is familiar with the basics of local search; we will only explain the essential parts of our local search approach that are specific to our problem. We refer to the work done by Blum et al. [11] for a general introduction into the area of Local Search.

Local Search approaches are capable of getting stuck in a local maximum. Over the years, multiple techniques have been developed to escape local maxima; two of these techniques are *Simulated Annealing* and *Tabu search*.

Simulated Annealing is a metaheuristic that allows moving to a worse solution under certain probabilities to escape local maxima. These worse solutions are accepted based on a probability that decreases the longer that algorithm is running. The probability of accepting a worse solution depends on the quality of the two solutions and a temperature $T > 0$. As the algorithm runs, the temperature cools down, resulting in lower probabilities of accepting a worse solution. The quality of a solution is measured with an objective function f taking a solution as input. This score has to be maximized. The probability p of accepting the worse solution b over the current solution a, $f(b) < f(a)$ can be defined as:

$$p = e^{\frac{f(b) - f(a)}{T}} \tag{4.1}$$

Another metaheuristic is Tabu search. Tabu search attempts to escape the local maxima by accepting worse solutions. However, unlike Simulated Annealing, which accepts a worse solution with a given probability, Tabu search evaluates a randomly sampled set of neighbours with size E and selects the best neighbour for further evaluation. If the selected neighbour is worse than the current one, it is possible that the current neighbour can be found in the neighbourhood of the selected solution, to prevent the algorithm from going back and selecting the solution it came from, a tabu list is maintained. Once a solution is selected, it will be added to the tabu list, and it cannot be reselected.

This paper uses a combination of Simulated Annealing and Tabu search. The best solution is chosen from a random subset of the neighbourhood, containing E solutions and no solutions found in the tabu list. Then, this solution is accepted based on the probability function as described in Simulated Annealing. If the solution is accepted, it will be used in the next iteration.

In the next sections, we describe the different parts of our local search and how they come together.

4.2 Solution Representation

The local search moves through a solution space in search of an optimal solution. For the local search to work with these solutions, they need to be represented in a data structure that contains all the information needed to construct a schedule and that can be easily adjusted to find neighbouring solutions.

To this end, van den Broek[2] uses an acyclic directed graph to represent solutions. We use a similar graph to represent a solution. This graph consists of multiple nodes representing actions a_i that are executed within a shunting yard and directed edges representing precedence relations between these actions. The nodes used in the graph consist of 7 different nodes representing: Arrival, Departure, Movement, Service, Split, Combine, and finally Turning actions. Parking a train does not have a specific node; it is implied that trains are parked in a period where they do not act.

> Every node has a starting time $\alpha(a_i)$ and a finishing time $\omega(a_i)$ representing start and finishing times of the action.

The first node is the ***arrival node***. These nodes are derived from the scenario, where every single node stands for an arriving train in the scenario. It mainly consists of three components: Firstly the arrival time, secondly the track from which the train enters the yard and finally the train composition of sub-types.

The next node, also derived from the scenario, is the ***departure node***. These nodes contain information for scheduling the departure of a train. This information includes the planned departure time from the scenario, the actual departure time, train composition and the track it leaves the yard from. While it is possible that the actual departure time differs from the planned one, it should not be happening and is an infeasibility.

The third node is the ***movement node***. The movement node is used to show when and how a specific train should be moving. This includes where it starts, which tracks it drives over, the destination track and finally, the expected duration time the movement will take.

Fourthly, the ***service node*** shows which service is performed on a train at a facility and the duration. More specifically, it shows the set of unit-trains on which the service is performed.

The next type is the ***split node***. When a train needs to be split into two, this node provides a description of how the split should proceed. It contains the original train and the two trains it splits into, along with the duration it takes to split the train. Similarly, the ***combine node*** describes a combination of two trains into one, along with the combination duration.

The final node is the ***turning node***. A train can be scheduled to reverse its direction to be able to reach other locations. To make the reversal possible, the machinist has to walk to the other side of the train. This act, together with the duration it takes to walk to the other side, is described with the turning node. The reason turning is not included in the movement node, is to increase flexibility; it is easier to modify multiple smaller nodes than a single big movement.

Directed edges are used to model the precedence relations within the graph and determine an order of execution within the nodes.

A precedence $a_i \rightarrow a_j$ models the starting time of node a_j to occur after a_i is finished.

Consequently, it can be stated that for starting time $\alpha(a_j)$ and the finishing time $\omega(a_i)$, the constraint $\alpha(a_j) \geq \omega(a_i)$ holds. In $a_i \rightarrow a_j$, a_i is the parent and a_j is the child.

We define two different kinds of precedence relations. The first type is a train relation. These can be found between two nodes referencing the same train, or in the case of split or combine, overlapping train units.

The next relations are the 'other' relations defining precedences between nodes referencing different trains. These relations are needed to order nodes that access the same resources at the same time, for example: when two movement-nodes make use of the same tracks, or when two service nodes require the same facility. Coincidentally, these two types of nodes are the only ones who need 'other' relations.

Using the precedence relations, we can determine starting times for nodes. Arrival nodes are assigned a starting time, after which the starting times can be propagated through the graph via the precedence relations.

All nodes execute as early as possible conform the precedence relations. However, the departure node is an exception because it should not depart earlier than planned. The preceding movement action should be executed directly before the departure to prevent trains waiting on an entrance track. Having a departure depart after the planned time is possible, but it will be counted as an infeasibility.

Figure 4.1 shows a visualization of such a graph. This graph shows the actions executed in the second example from Chapter 2.4. The straight arrows are train relations, and the dotted arrows are other relations.

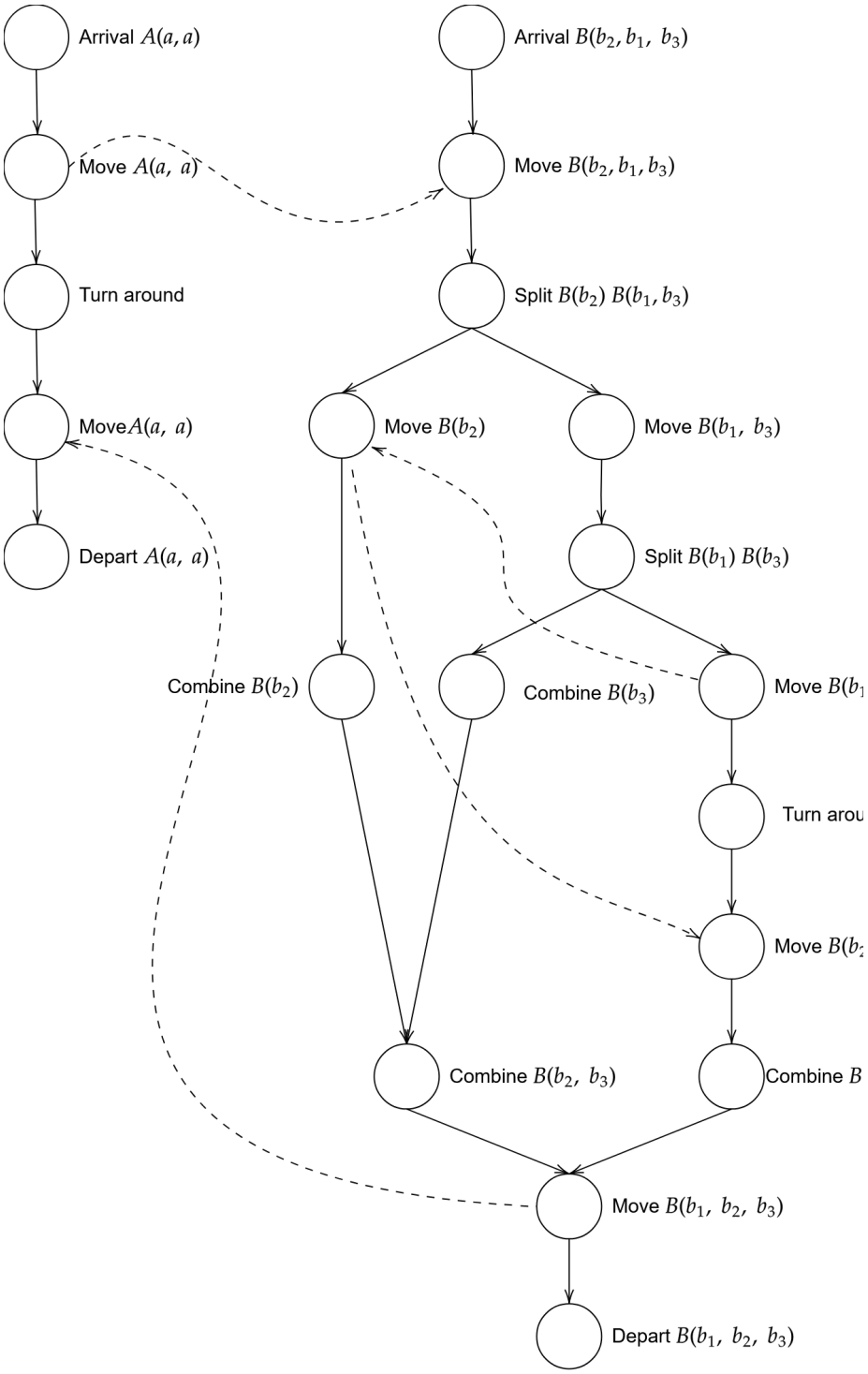


Figure 4.1: The graph displays the procedure to solve the example with disruption described in Chapter 2.4

.

4.3 Pathfinding

Each movement stores its route from the origin to the destination. If a new movement is added to the graph, then a pathfinding algorithm finds new routes for these movements. Finding a route is done with an A*-algorithm which evaluates the more promising paths first.

Rails within a yard are laid out in graph, in which a path is searched. Nodes in the graph represent either a track, switch or an intersection, and they are connected by edges to represent which rails connect.

The A*-algorithm starts searching for a path from the origin node and stops as soon as it finds the destination node. While searching, the train's driving direction is kept in mind. Some nodes, mainly switches, can reach different nodes depending on the direction the train is moving.

While exploring a node, reachable neighbouring nodes are added to a priority queue, which orders nodes based on a summation of a weight and a lower bound. The weight of a node is based on the duration to reach the node, starting from the origin. This time is similar to the one in movement-actions, a straight track adds 60 seconds to the weight, and each switch adds another 30. Additionally, the turning of a train adds an approximated 180 seconds and moving through a track where trains are parked adds another 600 seconds per train, to discourage these tracks.

The lower bound is similar to the weight mentioned above. It is the time to reach the destination from the current node. The time is pre-calculated using a Floyd–Warshall algorithm. It is used to find the shortest paths between all pairs of nodes, including the trains being able to turn. A straight track adds 60 seconds, a switch adds 30 and turning the train adds 180 seconds.

While a pre-calculated path is the shortest path between two locations, it does not account for obstacles standing in the way. The obstacles need to be taken into account while finding a path, hence the A*-algorithm.

4.4 Schedule changes

Besides containing all the information required to construct a schedule, the solution encoding also has to be flexible enough to make changes easily. A change applied to a solution results in a new solution which is a neighbour of the original. In the following section, we will describe the possible changes in detail.

4.4.1 Modelling disruptions

The first two changes are used to introduce disruptions into the plan. They are applied at the start of the local search to mirror the disruption that occurred during execution.

Add or remove arrival delay

The first change that can be applied is to add or remove an arrival delay. Planners apply this change to indicate a train arriving early or late. It can be adjusted within the solution encoding by merely adjusting the starting time of

the corresponding arrival node to the new starting time, followed by propagating this new time through the directed graph.

Change arrival order

The next change changes the arrival order of units within a train. This models a disruption where a train arrives with its units in a different order than was given by the scenario. The schedule has to be adjusted to ensure that the unexpected order does conform to the departing trains in the scenario.

The original train, as described by the scenario, will henceforth be referred to as the original train, while the train with the changed order will be referred to as the new train.

We make the assumption that a train arriving in unexpected order does not consist of more than three sub-types. On a shunting yard, trains containing more than three sub-types are rare, and we will ignore them.

This change starts with modifying the arrival node to refer to the new train, but it also adjusts other nodes and possibly adds new nodes to the graph. When applying the change, it becomes important to adjust the graph based on what happens to the train during its stay at the yard. If the train, at some point, is split into the same sub-type compositions as with the original split, then nothing needs to be adjusted. The possibly introduced conflict, such as trains driving through each other, will be resolved during the execution of the local search. If, on the other hand, the train can not split into the same compositions as the original, then we will have to add some nodes to move units and reorder them according to the original ordering of units.

If the original train is not scheduled to be split, but the train arrives in a different sub-type order, the train needs to be split up and recombined in the correct order. This is done by introducing new nodes to the graph and having the local search fix occurring conflicts.

4.4.2 Changes for exploring neighbourhoods used in the algorithm

The next changes are applied during the local search and are used to change a solution into a neighbouring one. Change and neighbourhood will be used interchangeably.

Add a precedence

This change adds 'other precedence relations' between a parent and a child node of the same type. For example, if two movements overlap each other, they share tracks and their times overlap, then adding a precedence relation between the two will force one to wait for the other.

Reverse precedence

Another useful change is to reverse a precedence relation, such that the order of execution for two nodes is reversed. This reversal makes it possible for an action to start earlier at the cost of a different one starting later.

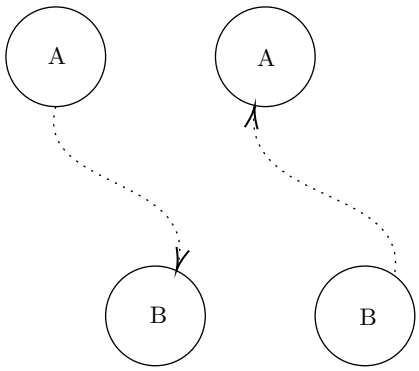


Figure 4.2: Reversing the precedence $a \to b$ to become $b \to a$

Remove a service

When a train can not depart in time, it might be beneficial to skip a service executed on a unit. Skipping a service can be represented in the graph by removing the corresponding node. After removal, the parents and the children of the removed node need to be connected with precedence relations to preserve a correct ordering.

Add a service

On the other hand, when a service has been previously removed, it might be possible to add the service back again. To facilitate this possibility, a new change introduces services to the graph by adding a new service node and precedence relations. If the train does not visit a facility where the service can be performed, a new movement has to be introduced, moving the train to a facility where it can be serviced.

Changing parking or service location

This change will adjust the parking location of a single train by replacing the necessary movements to move the train to a different track instead of the original one, potentially relocating the services at the old location as well. This has the added benefit that the change can be used to relocate services as well. This change does not add a new movement to move the train away, but it ensures that the train is never parked at the original track in the first place.

Besides changing the location of parking and services, this change is also used to change locations where splits and combinations are to happen. Changing these locations is a little different because there are three trains involved instead of one. In the case of splitting a train, this will be the one train being split into two other trains. Conversely, during combination, it will be the two trains combining into a single train. This causes the need to create new movements for three trains instead of one.

Move train out of the way

A movement is blocked if it traverses a track on which a train is parked. This change moves the train out of the way. The blocking train has to move to a new parking location from its current track. An efficient path is created to reach this location while potentially scheduling a stop in between to let the blocked train through. Ensuring that the initially blocked train can complete its movement.

The difference between this change and the previous change is as follows: This change allows the train to be parked still at the track where the blocked train moves through but then moves away to let the train through. While on the other hand, the previous change ensures that the train never parked there in the first place.

Add in between stop during a movement

Sometimes it is handy to split one big movement into two smaller ones. For example, a train arrives and needs to move to Track c, via Track b. If Track c is temporarily unavailable, then this change could let the train park at Track b, splitting the entire movement into two, one movement from the entrance to Track b, and another one from Track b to Track c.

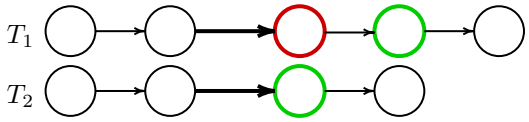
Recalculate movement

This change adjusts a movement to depart and/or arrive from a different side than it currently does. This is useful, for example, when a train should depart from one side of a track when the other side is blocked. Another use case is when a train should arrive at a different side of the track. For example, when a train needs to arrive at a different side to correctly combine.

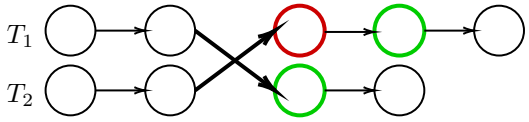
Swapping trains of the same sub-type

The last change swaps trains referenced by two groups of actions. A group is a chain of actions connected by a train precedence relationship. These two trains have the same sub-type composition; the swapping change forces one train taking over part of the other's planning and vice versa. Due to a train exchanging a part of the planning, it will happen that some performed services are not required anymore while others are missing. The non-required services are removed from the graph. Missing services will be added during the further iterations of the local search. Because the two trains are likely found in different places, new movements will be created to connect the trains properly.

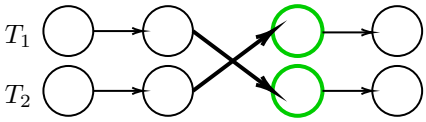
For example, Figure 4.3a shows two chains in an existing schedule. The upper chains corresponds to Train T_1, while the lower corresponds to Train T_2. Both trains have the same sub-type composition. The green nodes represent services both trains require, while the red node is a service only the upper chain, Train T_1, requires. In this example, the destination of the two thick arrows will be swapped, resulting in Figure 4.3b. Because Train T_2 does not require the red service node, it will be removed from its chain, becoming the final scenario displayed in Figure 4.3c.



(a)



(b)



(c)

Figure 4.3: Three phases of a train exchanging parts of their planning. Black nodes are nodes with nothing special, green nodes represent services that are executed on both trains and a red node represents a service that is only executed on Train T_1.

4.4.3 Resemblence to van den Broek's simulated annealing

Because of the similarities between previously described changes and the neighbourhoods described by van den Broek, this section will compare the two.

Van den Broek's algorithm has some more neighbourhoods. For example, a neighbourhood left out in this paper is the combination of two movement nodes. It is not included because we believe it decreases the flexibility for later rounds.

Multiple iterations of our changes can imitate some neighbourhoods from van den Broek. One such neighbourhood in van den Broek's algorithm is taking a movement and shifting it to an earlier or later position of ordering. We can achieve this by switching precedences one or more times. Another neighbourhood is swapping the parking location of two trains, we do not see the immediate usefulness of this neighbourhood, but if it is useful, it can be recreated with two iterations of changing parking locations.

There are two neighbourhoods that van den Broek does not have. One neighbourhood has the ability to remove services from trains, and one can add services.

4.5 Conflicts

The previously described changes, the neighbourhoods, are used to create neighbouring solutions. To limit the number of changes and explore more promising solutions first, we opted for conflict based neighbourhoods: The conflicts generate sensible changes. We proceed to describe conflicts, including the changes they generate.

Facility over capacity

Facilities can only manage a certain number of trains; it can happen that some change incorrectly burdens the facility with too many tasks at the same time.

Adding precedences between pairs of actions has a high possibility to solve this conflict with minimal impact on the entire schedule. The conflict generates a precedence relation for all permutations of two services performed at the facility.

In other cases, it might be handy to relocate a service to be executed at a different facility instead. For every service action, a change is generated to change the service location to a different facility.

Track over capacity

Just like a facility being over capacity, a track can also be over capacity. The combined length of parked trains exceeds the size of the track.

A possible change is to add some precedences to the movement actions of trains related to the conflict. The goals of adding precedences are to either let a train depart early, thereby creating space for another train to arrive. Alternatively, let an arriving train wait for the departure of a different train to create enough space. Another option to resolve the conflict is to park trains somewhere else instead.

Late departure

The departure of a train is recorded in the scenario, but it also has the earliest departure time, which is derived from precedence relations. It occurs that changes made to the schedule can force a train to depart after the time it was supposed to, creating a conflict.

This conflict works with the assumption that every node in the graph starts as soon as possible. With this assumption, a chain of consecutive actions can be found, which ends at the conflicting departure node.

If the chain contains a consecutive pair of nodes connected by 'other' precedences, then it can be concluded that one train has to wait for another causing a *bottleneck*. If there are no bottlenecks in the chain, then the only option for the train to depart on time is to drop non-essential services.

On the other hand, if there are bottlenecks, there are a few changes able to fix the bottlenecks. Flipping the precedence between the two actions causing the bottleneck causes the child action to start before its parent, thereby allowing the chain to finish faster. However, it is not a given that the chain will finish faster because they can have other influencing relations. Another option is to relocate one of the actions in the bottleneck to a different location, thereby removing the precedence relation between the two and allowing the waiting action to start early.

Missing service

A scenario describes which services should be executed on which train units. If one of these services is not planned in the solution, it will be counted as a conflict. The apparent change is to introduce the service for the units to the solution as described in the add service change.

Two movements sharing tracks at the same time

A movement reserves the tracks it uses for the duration of its executions. If another movement attempts to use the tracks while they are reserved, they risk colliding with each other. Even if they do not crash into each other, there will be the issue of switches. For example, if the first train uses a track a minute before the second, the paths of the two would diverge at some point by means of a switch. At some yards the machinists have to walk and manually flip the switches to the correct state, they both require the switch to be in their own state at the same time, which is not possible. What we can do, however, is to split a single movement into multiple, effectively adding an in-between stop and freeing reserved tracks.

This occurrence can be resolved in two different ways. The first and most obvious way is to add a precedence relationship between the two movements. The second way is to change the destination of one of the movements, parking the train at a different location and ensuring that the tracks are not shared.

Combination conflict

Sometimes, a change causes a combination-node to become invalid: the node tries to combine two trains, but they are not placed on the track in the order the combination wants them to be. What we can do to remedy this conflict, is to add precedences letting the trains arrive in the required order. Another possibility is, if there are no other trains on the track, to let a train arrive at the other side.

Train blocking a movement

The last conflict occurs when a train is scheduled to move through a track having a train parked on it. Again, multiple ways to change the schedule exist to stop the moving train from crashing into the parked one.

The first change is to move the parked train out of the way, adding a new movement node, such that it does not interfere with the other one.

Another change is changing the parking location of the blocking train to a different track. Unlike the previously proposed change which moves the train away by adding a new movement, changing the parking location modifies the existing movement moving the train to the parking spot to move somewhere else instead. The next possible change is to swap one of the trains with an identical one. Finally, the last change is forcing the moving train to start moving after the parked one has moved by adding new precedences between the movements.

4.6 Objective function

An objective function gives desirable solutions a better score while giving worse solutions a worse score. An essential aspect of the objective function is the *aim* of the Local Search. We aim to create a feasible schedule without conflicts while resembling the original schedule.

This aim can be partly reflected in the objective function by assigning penalties to occurring conflicts and other undesirables, such as having too many movements. The objective function does not contain a measure of similarity between the original schedule and a current solution; we believe the main focus of the objective function should be to find a feasible solution, moreover due to the nature of the neighbourhoods used, non-drastic changes, a feasible solution should be quite similar to the original schedule. In Chapter 5, we back up our claims of not needing a similarity score in the objective function.

4.6.1 Penalties

The objective function consists of a summation of penalties and a weight assigned to the penalty. The various conflicts are responsible for most of the penalties, but other undesirables are also penalized. These are having too many movements, splits, combination or turning nodes within the solution.

We will give a small overview of the penalties derived from conflicts that are found on a solution s. Each penalty has its own weight.

For a given solution s.

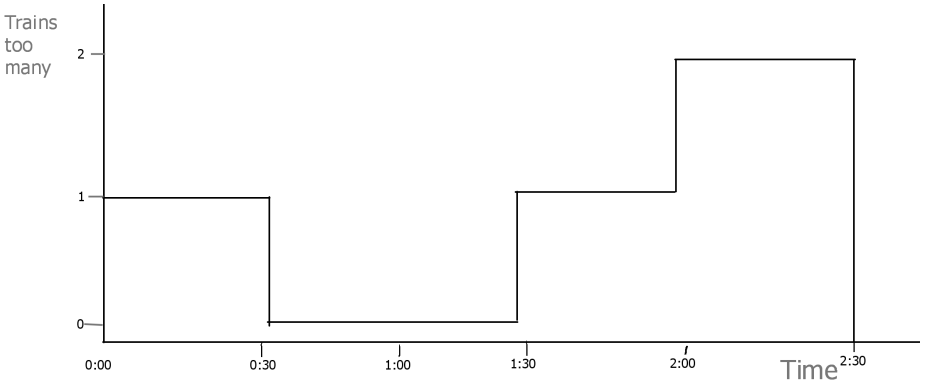


Figure 4.4: The average of the area under the line. The y-axis shows the excess amount of services and the x-axis the time.

The first penalty discourages facilities being over capacity. Each facility f used in the solution s gives a penalty value $foc_s(f) \geq 0$. This value is calculated by multiplying the number of excess services in a time period with the time period in seconds. When plotting the excess services over a time period as shown in Figure 4.4, the surface under the line is taken as the penalty value $foc_s(f)$.

The next penalty discourages the track being overcapacity. Each track τ gives a penalty value $tlv_s(\tau) \geq 0$. This value is calculated in the same way as the previous penalty, but the meters exceeding the length of track τ replace the number of excess trains. However, it is possible that this value becomes very low and is drowned in other penalties. For this reason, another penalty $tlvc_s$ is introduced. $tlvc_s$ is equal to the number of tracks containing too many trains. A track can be counted multiple times if it becomes over capacity multiple times. Incidentally, this can cause the combination of two small conflicts into one bigger conflict, depending on the weights taken.

The next penalty penalizes the delay of trains. The penalty $delay_s(d)$ is the delay of departure train d in minutes. Assuming that a train cannot depart earlier than scheduled, the $delay_s(d)$ function can only be higher or equal than 0. $delay_s(d) \geq 0$. This penalty has the same problem as the previous penalty: The delay can be very small and drowned out by others. A new penalty $delayc_s$ is introduced, equal to the number of trains departing late.

The following penalty penalizes combination conflicts. The penalty $combine_s$ represents the number of combination conflicts.

Next, services not performed on train unit u are penalized as well. The penalty for a unit u becomes $ms_s(u)$, representing the amount of services that are not performed on u.

The next penalty punishes two movements sharing tracks at the same time. The penalty $cr_s(m_1, m_2)$ becomes the amount of tracks shared between two conflicting movements m_1 and m_2.

The final penalty related to conflicts occurs when a parked train is blocking a moving train. The penalty $ct_s(m)$ is the amount of times a movement m drives through a parked train.

Unfortunately, we forgot to include the penalty and weight of the combination conflict. With the deadline quickly approaching, we believe it infeasible to rerun the experiments in time. However, these results will be displayed in the revision, and conclusions will be updated accordingly.

The next four penalties are designed to limit the amount of nodes needed. The type of nodes to limit are the movement nodes m_s, the combination nodes c_s, the split nodes s_s and the turning nodes w_s.

As mentioned before, every penalty has its own weight. This paper uses the following weights:

w_{foc}	w_{tlv_p}	w_{tlvc_p}	w_{delay_p}	w_{delayc_p}	w_{ms_p}	w_{cr_p}	w_{ct_p}
2	0.0003	40	0.08	40	2	20	60

$w_{combine_s}$	w_{m_p}	w_{c_p}	w_{s_p}	w_{t_p}
10	0.5	1	1	1

All these penalties combine into a single equation, where M is the set of

movement nodes, T the set of tracks, D the departing trains and F the set of facilities.

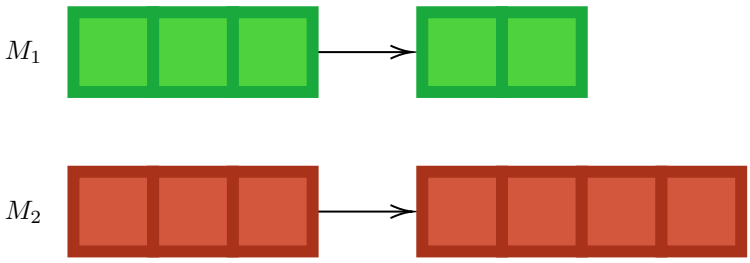
$$
\begin{aligned}
P(p) = w_{foc} \cdot \sum_{f \in F} foc_p(f) && (4.2)\\
+ w_{tlv_p} \cdot \sum_{\tau in T} tlv_p(\tau) && (4.3)\\
+ w_{tlvc_p} \cdot tlvc_p && (4.4)\\
+ w_{delay_p} \cdot \sum_{d \in D} delay_p(d) && (4.5)\\
+ w_{combine_s} \cdot combine_s && (4.6)\\
+ w_{delayc_p} \cdot delayc_p && (4.7)\\
+ w_{ms_p} \cdot \sum_{u \in U} ms_p(u) && (4.8)\\
+ w_{cr_p} \cdot \sum_{m_1 \in M} \sum_{m_2 \in M} cr_p(m_1, m_2) && (4.9)\\
+ w_{ct_p} \cdot \sum_{m \in M} ct_p(m) && (4.10)\\
+ w_{m_p} * m_p && (4.11)\\
+ w_{c_p} * c_p && (4.12)\\
+ w_{s_p} * s_p && (4.13)\\
+ w_{t_p} * t_p && (4.14)
\end{aligned}
$$

Each penalty has its own weight to denote the importance. The lines 4.2 to 4.10 represent conflicts within the solution, if all of these lines sum to 0 with weights bigger than 0, the solution is conflict free.

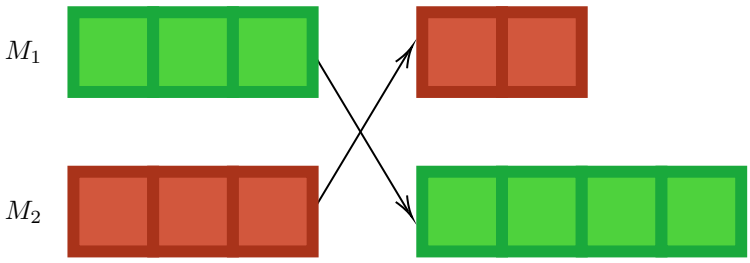
4.6.2 Similarity measure

The similarity measure focusses on machinists. We believe they are the individuals most affected by rescheduling a schedule.

Every machinist available is assigned a chain of movement actions. The initial assignment is created greedily. Every movement action is assigned in order of starting time to the machinist that can reach the starting location first. A precomputed table of walking times between tracks is used to calculate which machinist arrives first. If none of the machinists can arrive in time for the start of the movement, it has to wait for the first one available.



(a)



(b)

The initial mapping is improved via a simple local search. The neighbourhood of a mapping consists of mappings where the chains of two machinists are partly swapped. The score function is the averaged walking time of every machinist, which should be minimized. The local search iteratively improves the mapping by moving to a better neighbour with a lower score until no more improvements are possible.

As an example, Figure 4.5a shows a subset of a mapping, machinist M_1 performs the movements depicted as green blocks and machinist M_2 performs the movements in the red blocks. One of the neighbouring mappings is shown in Figure 4.5b. Machinists M_1 and M_2 exchange some of their movements.

Once the machinist schedules are optimized, the similarity measure is the difference of average walking times between the two. Durations of machinists driving the trains are not considered in the similarity values. We believe that these durations will not be significantly different in the context of this paper. In this paper, the difference will be taken between two machinist schedules pertaining to four machinists: One of the original non-disrupted shunting schedule, and the other one, found by the algorithm. If the difference is larger than zero, then the machinist has to walk that many minutes extra. Otherwise, if the difference is less than zero, the machinist walks that many minutes less.

Unfortunately, this method is elementary, because it averages the walking times. Multiple times can cancel each other out. A more advanced method could be created that uses scores the walking times on a more individual basis.

More extensive research in scheduling personnel is done by van den Broek et al. [12] He proposes two advanced techniques to schedule personnel. Not only

machinists are scheduled, but also other specialized personnel, such as engineers or cleaners.

Chapter 5

Experiments and Results

This chapter evaluates the performance of the algorithm. Experiments are carried out to accomplish this. They give insight in the performance of the algorithm using statistics.

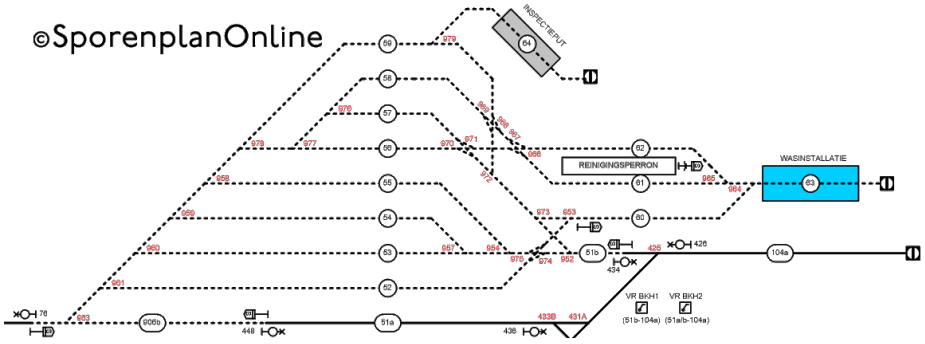


Figure 5.1: A Shunting yard at The Hague: Kleine Binckhorst

The experiments require test cases to give insight into performance. We use 1421 different scenarios that are generated using tools provided by the NS. The time starting at 14:00 until 07:00 is the busiest on the yard. Consequently, that is when most problems occur under a disruption. The test cases are placed in this time frame because they result in more relevant situations. On average, a scenario contains 11 trains, 15 train units and roughly 30 performed tasks.

This paper looks at two different types of disruptions: Trains arriving with units in an unexpected order, and trains arriving early or late. A set of test cases is created for both disruptions. The first set is for the unexpected order disruption. For every train in a scenario, a permutation is taken as a new test case, resulting in 5000 different test cases. The second set is for disruptions in the arrival time. Four different time adjustments are selected from real-world data displayed in Figure 2.4. The first adjustment is a train arriving 5 minutes early. It does not often occur and is an adjustment with a high chance of creating conflicts. Another disruption is a train arriving 10 minutes late, also with a significant chance of conflicts. Trains arriving 2 minutes early and 1 minute late, are two disruptions with a small probability to create a disruption.

We are applying these four disruptions to every train results in 44890 test cases. Running all of these test cases would take a long time. Therefore, we sample 5000 cases with high confidence in them representing the entire set.

The algorithm requires parameters to be set up. It will use the weights shown in Chapter 4.6.1 for running test-cases:

w_{foc}	w_{tlv_p}	w_{tlvc_p}	w_{delay_p}	w_{delayc_p}	w_{ms_p}	w_{cr_p}	w_{ct_p}
2	0.0003	40	0.08	40	2	20	60

$w_{combine_s}$	w_{m_p}	w_{c_p}	w_{s_p}	w_{t_p}
10	0.5	1	1	1

In chapter 5.1, we will use four variations of local search to solve instances. Three of these require specific parameters for running. The first variation is Iterated Improvement which is a variation that needs no parameters. The second variation is Tabu search, which only requires the single parameter E. It evaluates E random solutions in the neighbourhood from which the best solution is taken for next iterations. The current solution is then put in a tabu list which contains solutions that will never be re-evaluated. After various small runs using 250 cases with E set to 2, 5, 8, or 10, we concluded that Tabu search with $E = 10$ solves the most instances.

E
10

The third variation, Simulated Annealing, has two parameters: Temperature T and cooling factor α. This variation evaluates a random solution from the neighbourhood, until one is accepted based on a probability based on T. To cool the temperature, we multiply T with the cooling factor α after every iteration. Like Tabu search, we let Simulated Annealing run on 250 cases to find an optimal configuration for the two parameters. The attempted combination of variables are $(T, \alpha) = (340, 0.8), (550, 0.9), (300, 0.8)$ and $(600, 0.9)$.

T	α
600	0.9

Finally, the combination of Tabu search and Simulated Annealing combines evaluating E random solutions from the neighbourhood with accepting the best solution from these, based on a probability function based on temperature T. Again, T is cooled with a cooling factor α. These values are chosen based on the runs of the previous two variations.

T	α	E
600	0.9	10

5.1 Running times

Disruptions often occur without warning, but trains do not wait for conflicts to be removed from the schedule. Therefore, NS believes it is vital that a feasible

schedule is found quickly. The first experiment explores the runtime of the algorithm and compares it to different versions of the algorithm.

The primary algorithm is a local search that combines Tabu search with Simulated Annealing(TSSA). We compare the combination against Tabu search(TS), against Simulated Annealing(SA), and Iterated Improvement(II). All of which are described in the section above. These variations are run on the sets of test cases. A single case runs up to 15 times per algorithm but stops earlier if a feasible schedule is found. A single run searches for a maximum of 20 seconds.

We evaluate the two sets of test-cases independently. Table 5.1a shows the run time results of the first set, while Table 5.1b shows the run times of the second set. Another difference between the algorithms can be found in the number of neighbourhoods explored and changes applied. These results are shown in tables 5.2a and 5.2b.

	Solved			
	mean	std	max	count
TS	776 ms	1095 ms	6,961 ms	4,492
SA	101 ms	266 ms	11,477 ms	3,297
II	38 ms	52 ms	538 ms	2,111
TSSA	766 ms	1109 ms	6,724 ms	4,389

(a) First test case set

	Solved			
	mean	std	max	count
TS	423 ms	683 ms	5,703 ms	4,969
SA	93 ms	194 ms	5,993 ms	4,037
II	64 ms	128 ms	1,274 ms	2,948
TSSA	419 ms	676 ms	5,742 ms	4,958

(b) Second test case set

Table 5.1: Results of two sets. Only results for cases for which a feasible solution is found are shown. The results contain a mean, standard deviation, and the number of occurrences.

	Solved	
	evals	depth
TS	75	4
SA	12	2
II	5	1
TSSA	74	4

(a) Averaged results of the first test case set.

	Solved	
	evals	depth
TS	48	2
SA	10	2
II	7	1
TSSA	48	2

(b) Averaged results of the second test case set.

Table 5.2: Evals is short for the amount of solutions evaluated, while depth is the amount of changes applied.

We notice a trend in the results. If the algorithm finds a feasible schedule, it will find it very quickly. Otherwise, if it does not find one, it will take

significantly longer before the algorithm stops searching. For example, if the TSSA-algorithm runs longer than 6.7 seconds, then the algorithm will likely not find a feasible schedule. In conclusion, a run time of 20 seconds is not restraining the algorithms, because a solution should be found within that period.

Another observation is that SA and II solve an instance significantly faster than TSSA. II is fast, but weak in finding feasible schedules requiring more than one change. This is because it cannot escape local maxima. Evidently, a schedule often has to become worse by introducing new conflicts in order for a feasible one to be found. On the other hand, SA is able to solve more cases, but still gets stuck in local maxima relatively quickly.

TS and TSSA have a similar performance in run-time and the number of cases solved. While TS has a faster average run time, it is not statistically significantly faster than TSSA. Both are able to escape local maxima and generally find a feasible schedule with a few changes applied.

5.2 Similarity

The goal of the next experiment is to test the similarity, defined in Chapter 4.6.2, between feasible schedules and the original undisrupted schedule. We want to see if the algorithms create schedules with a good similarity value. A good result is when the similarity value is smaller or equal to zero.

We look at, and compare, the four algorithms again. A single algorithm runs 15 times on a single test case and takes the average similarity of feasible schedules. Tables 5.3a and 5.3b show the resulting similarity scores of the first and second set of test cases, respectively. Only the results are shown of cases for which a feasible solution is found. The results show the mean, min, and max similarities. If the similarity has a value less than 0, then the machinists have to walk on average that many seconds less within the schedule. Conversely, if the value is larger than 0, the machinist has to walk that many seconds more.

II has a better similarity score. However, we believe that due to its inability to escape local maxima, it does not apply many changes. As a result, the schedule is not able to change significantly.

In the previous experiment, we found that TSSA and TS have similar results. After executing students t-test on their similarity score, we find that TSSA yields significantly better similarity scores than TS, for both sets of test cases.

If every machinist has to walk an hour longer, it will be in a period from 14:00 to 07:00, for a total of 17 hours. We believe that the similarity scores are acceptable if they do not exceed 30 minutes. With 30 minutes, the machinist has to walk on average 1.8 minutes longer per hour if it was spread out over the 17 hours. However, the machinist schedule is affected after the disruption occurs. Therefore, the machinist has to walk its seconds more, in a smaller time-frame than 17 hours. We believe that an extra time of 30 minutes covers most scenarios.

	mean	min	max
TS	117 s	-1,143 s	2,637 s
SA	42 s	-1,429 s	2,028 s
II	13 s	-1,112 s	1,823 s
TSSA	103 s	-1,108 s	2,194 s

(a) Results for first test cases in seconds

	mean	min	max
TS	45 s	-970 s	1,845 s
SA	28 s	-1,357 s	1,861 s
II	6 s	-935 s	1,201 s
TSSA	43 s	-935 s	1,755 s

(b) Results for second test cases in seconds

Table 5.3: Cases where a feasible schedule is found

We show the weight distribution of the results in Figures 5.2 and 5.3. We see multiple distributions with the average close to 0. While the curves look like normal distributions, according to the Shapiro-Wilk test, none of them are. The curves show that most of the similarity scores are centred around their mean value, values larger than 1,500 seconds are rarely encountered. Consequently, most feasible schedules are acceptable in similarity.

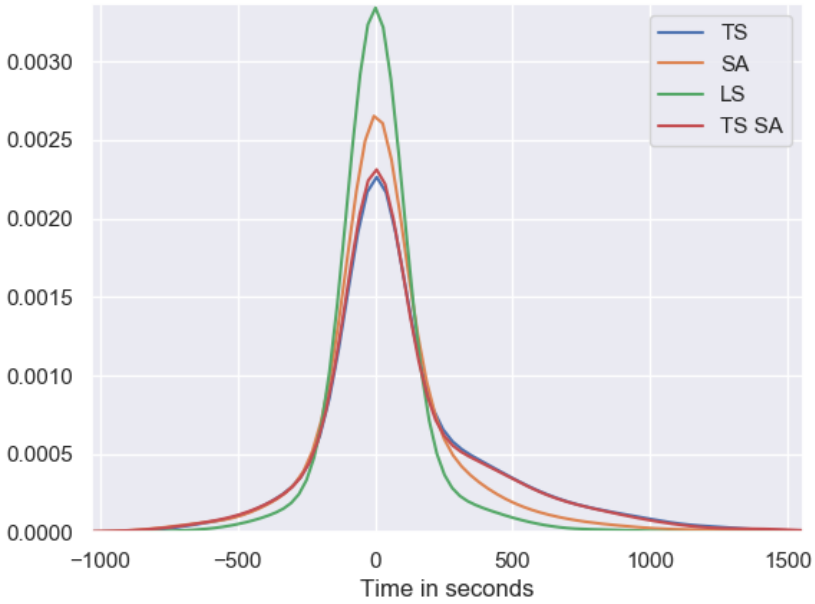


Figure 5.2: The weight distribution of similarity scores on the first set of cases

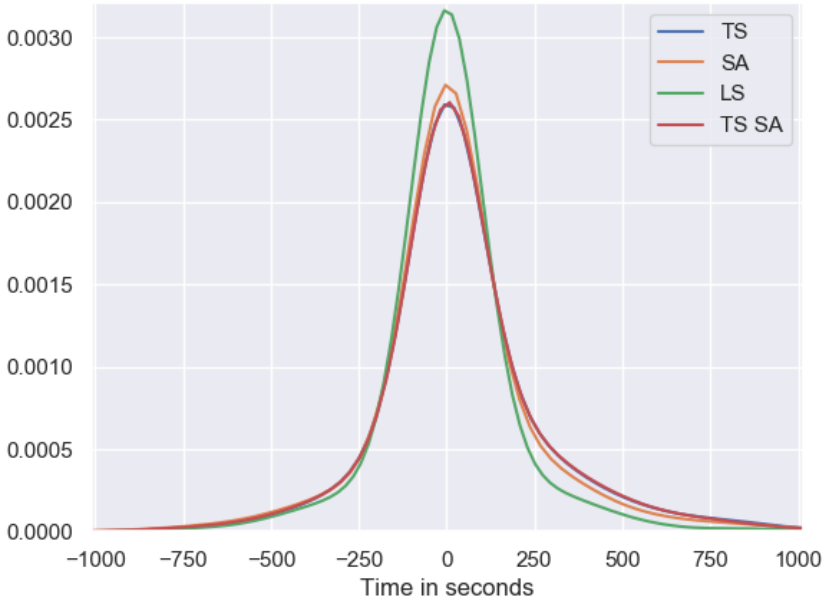


Figure 5.3: The weight distribution of similarity scores on the second set of cases

5.3 Comparison with van den Broek's Simulated Annealing

The final experiment compares the number of feasible schedules. It compares the schedules created by TSSA with the schedules created by the Simulated Annealing algorithm of van den Broek et al.[2], which we will shorten to VDB. This algorithm creates an entirely new schedule. Therefore, it requires the scenario to be adjusted. We adjust the scenario to capture the new situation after the disruption happened, including removing services that have been executed and putting trains on tracks where they are supposed to be at the time of the disruption. VDB runs with a maximum run time of 60 seconds.

From the four local search implementations, we chose TSSA for comparison because it performs well enough, and is not significantly better than TS.

Tables 5.4a and 5.4b both show four numbers representing test cases solved only by TSSA, only by VDB, solved by both, and solved by neither of the two.

	Solved
Only TSSA	869
Only VDB	417
Both	3,022
Neither	130

(a) Results on first set of cases, trains units arrive in unexpected order

	Solved
Only TSSA	913
Only VDB	30
Both	4,104
Neither	19

(b) Results on second set of cases, trains arrive early or late

Our algorithm yields more feasible solutions than VDB in less time. This is likely because TSSA works from an existing schedule containing conflicts, while VDB recreates an entirely new one. While our algorithm is capable of solving more cases than VDB, VDB can solve quite a few cases that our algorithm can not. This observation shows that TSSA does not always find a feasible schedule when there is one. We believe that this new schedule structure is so different from the original, that our algorithm could not find it, and could only be found by creating an entirely new schedule.

A keen reader might notice that the results do not count up to 5000, this is due to the author making mistakes. Rerunning the algorithms for results will not complete before the deadline of this paper, but correct results will be present in the revision.

Chapter 6

Conclusion

Shunting schedules often deviate from their original plan due to disruptions: a train arriving at a different time, or arriving with units in an unexpected order. If these disruptions break the schedule in any way, it will have to be corrected manually.

We proposed a local search technique with the help of Tabu search combined with Simulated Annealing to aid in removing created conflicts while aiming for a minimum deviation from the original schedule. The algorithm uses a directed graph to model activities taking place on the yard. In this graph, nodes model the activities and directed edges imply the order in which they execute. Our algorithm transforms an existing schedule into a graph, applies the disruption and then proceeds to resolve conflicts by applying small changes to the graph.

In Chapter 5, we performed experiments using variations of the algorithm. These variations include local search, Simulated Annealing, Tabu search, and the combination of the latter two. In the first experiment, we showed that the algorithms are capable of finding feasible schedules within seconds if it can find one. With the second experiment, we showed that newly found schedules generally are similar enough to the original schedule. This was decided by using a similarity score based on a comparison of average walking times between machinist schedules of the original and the new schedule. This approach is the first attempt at quantifying similarity between schedules, and can still be improved. The final experiment compared our algorithm against the Simulated Annealing algorithm designed by van den Broek et al.[2] This showed that, while our algorithm was able to solve more test cases, it does not always find a feasible schedule if there is one available.

Overall, we believe that Tabu search combined with Simulated Annealing is a promising approach for aiding NS planners in rescheduling.

6.1 Further research

The current similarity function is mainly based on the emotions of the machinists, while we believe these should certainly be incorporated, a method based on structural differences would be more desirable. However, finding such a method would be difficult, because comparing graphs is difficult.

More research could be done in finding smart neighbourhoods and constraining search space by applying constraint programming techniques.

Bibliography

[1] C. Eggermont, C. A. Hurkens, M. Modelski, and G. J. Woeginger, "The hardness of train rearrangements," *Operations Research Letters*, vol. 37, no. 2, pp. 80–82, Mar. 2009. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0167637709000157

[2] R. van den Broek, "Train Shunting and Service Scheduling: An integrated local search approach," Master's thesis, Universiteit Utrecht, 2016. [Online]. Available: https://dspace.library.uu.nl/handle/1874/338269

[3] J. Clausen, A. Larsen, J. Larsen, and N. J. Rezanova, "Disruption management in the airline industry—Concepts, models and methods," *Computers & Operations Research*, vol. 37, no. 5, pp. 809–821, May 2010. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0305054809000914

[4] ——, "Disruption management in the airline industry—Concepts, models and methods," *Computers & Operations Research*, vol. 37, no. 5, pp. 809–821, May 2010. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0305054809000914

[5] M. F. Argüello, J. F. Bard, and G. Yu, "A Grasp for Aircraft Routing in Response to Groundings and Delays," *Journal of Combinatorial Optimization*, vol. 1, no. 3, pp. 211–228, 1997. [Online]. Available: http://link.springer.com/10.1023/A:1009772208981

[6] G. Zhu, J. F. Bard, and G. Yu, "Disruption management for resource-constrained project scheduling," *Journal of the Operational Research Society*, vol. 56, no. 4, pp. 365–381, Apr. 2005. [Online]. Available: https://www.tandfonline.com/doi/full/10.1057/palgrave.jors.2601860

[7] S. Cicerone, G. D'Angelo, G. D. Stefano, D. Frigioni, and A. Navarra, "12. Robust algorithms and price of robustness in shunting problems," in *7th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'07)*, ser. OpenAccess Series in Informatics (OASIcs), C. Liebchen, R. K. Ahuja, and J. A. Mesa, Eds., vol. 7. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2007. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2007/1175

[8] R. Freling, R. M. Lentink, L. G. Kroon, and D. Huisman, "Shunting of Passenger Train Units in a Railway Station," *Transportation*

Science, vol. 39, no. 2, pp. 261–272, May 2005. [Online]. Available: http://pubsonline.informs.org/doi/abs/10.1287/trsc.1030.0076

[9] R. van den Broek, H. Hoogeveen, and M. van den Akker, "How to Measure the Robustness of Shunting Plans," p. 13 pages, 2018. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2018/9708/

[10] E. Peer, V. Menkovski, Y. Zhang, and W.-J. Lee, "Shunting Trains with Deep Reinforcement Learning," in *2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. Miyazaki, Japan: IEEE, Oct. 2018, pp. 3063–3068. [Online]. Available: https://ieeexplore.ieee.org/document/8616516/

[11] C. Blum and A. Roli, "Metaheuristics in combinatorial optimization: Overview and conceptual comparison," *ACM Computing Surveys*, vol. 35, no. 3, pp. 268–308, Sep. 2003. [Online]. Available: http://portal.acm.org/citation.cfm?doid=937503.937505

[12] R. van den Broek, J. van den Akker, and J. Hoogeveen, "Personnel Scheduling on Railway Yards," in *20th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Sep. 2020.