Routing Shunt Trains at NS with Constraint Programming

Nevena Gincheva

Routing Shunt Trains at NS with Constraint Programming

Nevena Gincheva

to obtain the degree of Master of Science
in Computer Science
at the Delft University of Technology

to be defended publicly on
12 July 2024 at 11:30

Student number:	5835542	
Thesis committee:	Dr. E. Demirović	*thesis advisor*
	Dr. C. Lofi	*committee member*
	Maarten Flippo	*daily supervisor*
	Jord Boeijink	*external supervisor*

Acknowledgment:
This research is conducted at TU Delft with the collaboration of the R&D Department at NS, the leading railway company in the Nederlands. The thesis is supported by the project ”Towards a Unification of AI-Based Solving Paradigms for Combinatorial Optimisation” (OCENW.M.21.078) of the research program ”Open Competition Domain Science - M” which is financed by the Dutch Research Council (NWO).

Preface

Welcome to my thesis on "Routing Shunt Trains". This work marks the culmination of my academic education. After five years, I look back on this wonderful time filled with proud moments and happiness, but also with moments of challenges and tears. However, as the saying goes, "Nothing worth having comes easy."

I would like to express my gratitude to my TU Delft supervisor, Maarten Flippo, and my thesis advisor Emir Demirović. Your guidance and support have been invaluable to me and the flow of this thesis. I would also like to thank my NS supervisor, Jord Boilink, who always found time to discuss with me the project even during his busiest days. Additionally, I am thankful to Bob for allowing me to conduct my research with NS.

I would like to thank my friends from my bachelor years, Kalina, Annika, Teun, Maartin, and Shanessa, and to those from my master years, Prakhar, Vyshnavi, and Violeta, I cannot imagine the past five years without you. It was wonderful to work and study together. I am thankful to everyone from my Dodgeball Association for the exciting training sessions that provided the much-needed break from my thesis.

Finally, I would like to thank Hristo, for your unwavering help and encouragement. Your willingness to listen and offer assistance has been a constant source of comfort. My deepest appreciation goes to my family, my parents, and my sister for their constant support throughout these five years. Especially to my mom, who never grew tired of listening to me. Your patience and understanding have meant the world to me.

Thank you all for being a part of my journey.

Abstract

The Shunt Routing Problem (SRP) is an important logistic problem at nearly every railway station. It is a subproblem of an even bigger train scheduling problem, the Train Unit Shunting Problem (TUSP). The objective of SRP is to schedule conflict-free routes between the platforms and the yards, for currently non-operational trains that have completed their current journeys and have not yet commenced the next one. We developed a Constraint Programming (CP) model for the Shunt Routing Problem at NS. The model is intended to replace the current algorithm for constructing the initial solution for the Local Search, addressing the TUSP at NS. The model incorporates all essential feasibility requirements and produces conflict-free solutions, in contrast to its forerunner. Moreover, it can be applied to any instance and station layout.

Initially, the model exhibited an average performance of 7-8 minutes on a real-life instance at Enkhuizen station. We managed to enhance the model and reduce the computation time to less than a second. The key improvements were achieved by changing the solver search strategy, imposing additional search guidelines, or further restricting the domains of influential variables. During experimentation, an optimized second version of the model was proposed. The evaluation undoubtedly confirmed that it outperforms the first version.

Both versions of the model, without any additional enhancements, perform exceptionally well on a second real-life instance at Amersfoort station, solving it in milliseconds. Even with a considerable number of additional routes, the performance of the second version remained reasonable, with a computation time of 18 seconds.

A final experiment revealed the remarkable performance of Gecode, achieving computation time in milliseconds with the second version of the model implemented for Enkhuizen in MiniZinc. This performance is an order of magnitude faster compared to the default performance of Google OR-Tools on Enkhuizen with the second version.

keywords: algorithms, constraint programming, train scheduling, routing

Contents

1 Introduction to TUSP 3
- 1.1 Train planning processes . . . 3
- 1.2 Problem description . . . 3
- 1.3 Solution approaches for TUSP . . . 3
- 1.4 Terminology . . . 4
- 1.5 Literature review . . . 4
- 1.6 NS approach for TUSP . . . 5
- 1.7 Research questions . . . 6

2 Shunt Routing Problem 7
- 2.1 Shunt Routing Problem . . . 7
  - 2.1.1 A high-level description . . . 7
  - 2.1.2 Routing concepts . . . 7
  - 2.1.3 Formal definition of SRP . . . 8
- 2.2 Constraint programming . . . 9
- 2.3 Model . . . 9
  - 2.3.1 Scope of the model . . . 9
  - 2.3.2 Requirements . . . 10
  - 2.3.3 Assumptions . . . 10
  - 2.3.4 Parameters . . . 10
  - 2.3.5 Input . . . 11
  - 2.3.6 Variables and function definitions . . . 13
  - 2.3.7 Constraints . . . 15
  - 2.3.8 The second version of the model . . . 18
  - 2.3.9 Applied extension . . . 19
  - 2.3.10 Output . . . 20

3 Instances 22
- 3.1 Instances . . . 22
  - 3.1.1 Real instances . . . 22
  - 3.1.2 Instance creation . . . 22
  - 3.1.3 Data preprocessing . . . 23
- 3.2 Validations . . . 23
  - 3.2.1 Validations for model-v2 . . . 23
  - 3.2.2 Additional checks for model-v1 . . . 24

4 Experiments & Results 26
- 4.1 Goal of the experiments . . . 26
- 4.2 Experiments' overview . . . 26
- 4.3 Hypotheses . . . 27
  - 4.3.1 Hypotheses for changes in the model . . . 27
  - 4.3.2 Hypotheses for the influence of the instance characteristics . . . 27
  - 4.3.3 Hypotheses for the MiniZinc model . . . 28
- 4.4 Experimental Setup . . . 28
  - 4.4.1 Types of experiments . . . 29
  - 4.4.2 Solutions' quality . . . 29
- 4.5 Experiments with model changes . . . 30
  - 4.5.1 Adding parking constraints . . . 30
  - 4.5.2 Variable and value strategies . . . 31

4.5.3 Compare both versions of the model . . . 37
4.6 Experiments with instance characteristics . . . 38
4.6.1 Additional routes for Amersfoort . . . 38
4.6.2 Removing a yard, or reversals . . . 42
4.6.3 Compare other solvers on model-v2 . . . 44
4.6.4 Deeper analysis of some results . . . 45
4.6.5 Main contributions . . . 47
Conclusion . . . 49
Future work . . . 50

A TMP **52**
A.1 Train Matching Problem (TMP) . . . 52
A.1.1 Common notation for both problems . . . 52
A.1.2 Initial general matching model for NS . . . 53
A.2 CP model: . . . 55
A.3 Solve TMP together with SRP . . . 55
A.4 Additional extension for deciding on where to perform a split or a combine . . . 55

B Model with varying time between submovements **57**
B.1 Additional variables . . . 57
B.2 Constraints . . . 57
B.2.1 Constraining the start and end times for section occupations . . . 57
B.2.2 Constraints for synchronization between routes and sections occupations and waiting variables . . . 58
B.2.3 Constraints for determining section occupations . . . 59
B.2.4 Constraints for start and end times of "wait" variables . . . 59

C Other extensions & Objective functions **60**
C.1 Other extensions . . . 60
C.1.1 Additional parameters . . . 60
C.1.2 Yard extension . . . 60
C.1.3 Service extension . . . 60
C.2 Examples of objective functions . . . 61
C.2.1 Minimize total number of shunt movements . . . 61
C.2.2 Minimize total routing time . . . 61
C.2.3 Minimize total distance . . . 61

D Additional Figures **62**

Chapter 1

Introduction to TUSP

1.1 Train planning processes

We begin with the various planning processes involved at a typical railway operator and subsequently focus on the relevant planning stage for our discussion. Figure 1.1 illustrates all planning processes considered at the railway provider.

The *timetabling* planning assigns arrival and departure times at each station for all train services. This generated schedule provides the input for the next phase - *shunt planning*, which is the focus of this thesis. ***This planning phase focuses on organizing the local processes at different stations.*** Following the timetabling stage, the *rolling stock planning* determines the train units that will perform the planned services.

Prior to the *timetabling planning*, the network planning and line planning phases are handled. *Network planning* is the strategic process of designing, implementing, and managing the railway infrastructure based on the estimated future demand [Lentink, 2006]. The *line planning* problem determines the origins and the destinations of the lines, as well as the frequencies of train journeys on them [Lentink, 2006]. The final planning process is the *crew planning*.

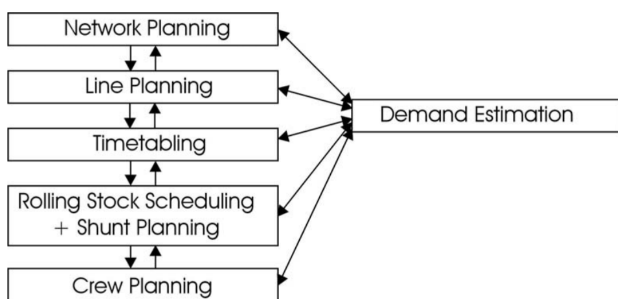


Figure 1.1: Overview of the planning processes (source [Lentink, 2006])

1.2 Problem description

Railway transportation demand is increasing rapidly, in response to this railway operators increase the train services by introducing more frequent and larger trains for certain destinations than before. Increasing the train services would not have been a problem if extending the infrastructure was not difficult, expensive, and sometimes even impossible as stations are usually in city centers. This problem is exacerbated in densely populated countries such as the Netherlands. Not being able to enlarge the infrastructure, but at the same time introducing more train services complicates the planning process. The Dutch Railways, in Dutch Nederlandse Spoorwegen, abbreviated as NS, is the main passenger railway operator of the Netherlands. With more than one million train services scheduled per day and more than 20,500 employees (NS, 2023), NS serves one of the busiest railway networks in the world. Only operators in Switzerland drive more train kilometers per track kilometer (NS, 2019).

This introduces the need for further precise logistic operations. New and faster decision support techniques and models are required to plan the various aspects of the train scheduling. One of the planning stages involves creating *shunt plans* for each station.

A shunt plan describes all movements or actions (any activities, for instance moving, splitting, combining, or standing still at the platform) that the train will undertake from its arrival at the platform to its departure. The station is bounded by entering or also leaving points of the station (the ends of the station). In general, an entering point can also serve as a leaving point and vice versa. The green line on Figure 1.2 marks part of the entering/leaving points at Utrecht Central Station. The movements from an entering point to the arrival platform and from the departure platform to the end of the station are not decided in the shunt planning phase. Those are already predetermined when the timetable is created.

A shunt plan consists of the following processes: matching the arriving trains to departing train services, parking trains on the shunt tracks, routing trains from the platform to the shunting yard and vice versa, and maintenance planning. In literature, the problem of planning a shunt plan is referred to *Train Unit Shunting Problem (TUSP)*.***Creating a shunt plan for a station is independent of other stations.***

1.3 Solution approaches for TUSP

We identified two main approaches for solving TUSP. The first one decomposes the problem into, in general,

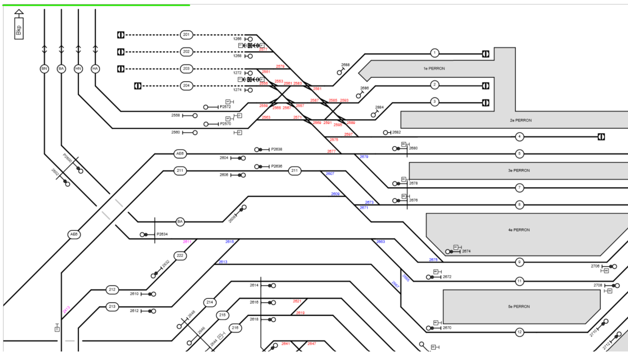


Figure 1.2: Part of the track plan of Utrecht Central Station from *www.sporenplan.nl*

four subproblems that correspond to the tasks of the shunt planning (introduced in the previous section 1.2) and solves those sequentially. We repeat them here - matching arriving train units to scheduled departing train compositions, choosing a routing through the station, parking positions, and scheduling maintenance. Some authors argue that the decomposition method is unlikely to be effective for the TUSP due to the complex interactions between the different subproblems [Broek, 2016]. Furthermore, decomposing the problem eliminates the ability to prove global optimality. Last, decomposition itself is challenging, as the subproblems are intertwined.

The second approach models and considers TUSP completely integrated. While this approach can prove optimality, it often results in a large and complex model that becomes computationally intractable, as demonstrated by [Kamenga et al., 2019]. Consequently, the authors of [Kamenga et al., 2019] with the sequential approach while also exploring combinations of some of the subproblems.

The primary reason for favoring the decomposition approach is its compatibility with the existing algorithms and tools used by NS. A more detailed explanation of these algorithms and their strategy to solve TUSP is provided in a subsequent section (Section 1.6).

Before introducing the subproblems and providing a detailed explanation for those focused on in this thesis, we will define important terminology that will be used throughout the thesis.

1.4 Terminology

Trains (also called ***rolling stock***) are composed of ***train units***. This composition is sometimes referred to as a ***configuration of the train***.

A ***train unit*** is a bidirectional self-propelled railway vehicle. Typically, a train unit consists of multiple carriages that are inseparable. Figure 1.3 shows a train unit of three carriages. Each train unit has a name (a unique ID number) and a ***type***. In addition, the whole train composition has also a unique ID. We will refer to the units of an arriving or departing train as ***arriving (train) units*** and ***departing (train) units***, respectively.

Figure 1.3: An example of an ICM train unit with 3 carriages (ICM-3) (source [Richard Freling, 2005])

The ***types of trains*** are organized in families. For instance, the InterCity family (ICM) consists of two subtypes - ICM-3 and ICM-4. The number indicates the number of carriages in the train unit.

Train units belonging to the same family can be combined to form longer trains. For instance, ICM-3 and ICM-4 can be combined to form a longer train. However, ICM-3 and VIRM-4 cannotbe combined, as they belong to different families.

In general, train units of the same subtype can be used interchangeably. This implies that a planner must determine a matching of arriving to departing units. The order of the train units is important. A train with a composition [VIRM4]-[VIRM6] is different from a train with composition [VIRM6]-[VIRM4].

Last, from the perspective of a single station, a train can be either a ***through train*** or a ***shunt train***. The ***through train*** is a train that arrives at the station, stops for passengers on and off-boarding, and then proceeds towards the next station. In contrast, a ***shunt train*** (also called ***plannable train***) terminates its journey at this station. This station will also serve as a starting point for its next journey.

1.5 Literature review

Train Unit Shunting Problem (TUSP) has been an area of research from the previous century. [Kroon et al., 1997] investigated the complexity of routing trains through railway stations, later named the Shunt Routing Problem (SRP). They demonstrated that SRP is NP-complete as soon as each train has three routing possibilities. A conclusion also derived formally in the Ph.D. thesis of [Lentink, 2006].

[Kroon et al., 1997] also observed that only a subset of the sections and routes of a railway station require consideration. Specifically, these critical sections include those:

(i) containing a switch,

(ii) corresponding to the entering and leaving points,

(iii) corresponding to the platforms.

[Kroon et al., 1997] further proved that it is sufficient to consider only these sections when ensuring that two different trains do not occupy the same part of the route simultaneously.

However, the method of route reservation assumed in this paper differs from the one currently employed by NS, and hence this valuable insight could not be utilized.

[Lentink, 2006] dedicates his Ph.D. thesis to solving TUSP. He introduces TUSP as a decomposition of

four subproblems and names them as *Train Matching Problem (TMP)*, *Shunt Routing Problem (SRP)*, *Train Parking Problem (TPP)* and *Maintenance scheduling*. Each subproblem is extensively explained, and algorithms or models in *Mixed Integer Programming (MIP)* are provided. TMP was modeled and solved with MIP. The model incorporates components from the shortest path problem and classical matching. The infrastructure for the Routing Problem was modeled as a graph and algorithms based on $A*$ and variations of BFS and $Dijkstra$ were applied.

Besides providing formal proofs regarding the complexity of the subproblems, [Lentink, 2006] also offers insights into conditions that simplify them, reducing their complexity from NP-hard to polynomial time-solvable. In his experiments, he utilizes instances from two stations - Zwolle and Enschede. [Richard Freling, 2005] addresses two of the TUSP subproblems - TMP and TPP, using a column generation approach.

[Haahr et al., 2017] explores and compares different solution approaches for solving TUSP, although their definition excludes the routing and maintenance scheduling components. They have developed a Constraint Programming (CP) formulation primarily addressing the parking problem. Additionally, a Column Generation approach and a randomized greedy heuristic algorithm are also provided. Their work also includes a comparison and benchmarking of these approaches against existing methods based on Mixed Integer Linear Programming (MILP) and a two-stage heuristic. The benchmark incorporates multiple real-life instances provided by the Danish State Railways (DSB) and Netherlands Railways (NS).

The experiments revealed that the exact models based on MILP and CP consumed more than 24 gigabytes of memory due to the large number of constraints and/or variables required. These models were outperformed by the Two-Stage heuristic and greedy randomized approach. The authors further explored a variant of the CP where the number of different train types assigned to the same track is limited. This limitation significantly improved performance by reducing the number of combinations for parking trains. A variant of the MIP was also tested where conflict constraints are generated on the fly. These two variants showed a significant improvement.

In his master's thesis, [Broek, 2016] develops a local search algorithm for TUSP that iteratively improves different parts of the solution until no further improvements can be made. His thesis laid the foundation for the Local Search algorithm currently employed at NS, as it significantly outperformed the algorithm previously used by the company.

[Wattel, 2021] develops a CP model for the Shunt Routing Problem (SRP) subproblem at Eindhoven station in his master thesis. He concludes that using CP to solve the SRP subproblem is beneficial for generating the initial solution at NS, which is then utilized by the Local Search algorithm.

A similar problem to TUSP is also found in the tram dispatching domain. [Winter and Zimmermann, 2000] models different variations of the tram dispatching problem using MIP.

[Kamenga et al., 2019] attempts to solve TUSP integrally with MIP, addressing all subproblems simultaneously to obtain an exact solution. However, the resulting model became extremely complex and large, suffering from significant computation time issues.

In their next attempt, [Kamenga et al., 2021] they investigate different possibilities to reduce the computation time by decomposing TUSP. In their evaluation, they also attempt to address different subproblems together and then iteratively proceed with the next ones. This approach results in better solutions compared to their initial integrated model.

[Rodriguez, 2007], [Cappart and Schaus, 2017], [Marlière et al., 2023] focus on real-time train rescheduling problems. In case of disturbance such as a delay of a train or infrastructure malfunction, decisions must be made manually by an operator in a very limited timeframe to reschedule the traffic and reduce the consequence of the disturbances. Operators may modify the departure time of a train or redirect it through an alternative route. Unfortunately, these disturbances and subsequent decisions can have unforeseen negative snowball effects on the delays of subsequent trains, especially in dense railway networks.

To address this issue, the authors proposed a Constraint Programming model to assist the operators in making more informed decisions in real-time. They conclude that CP is well-suited for solving routing problems in real-time railway operations.

In his thesis [Hendrikse, 2021] evaluates the capacity of the shunting yards at NS by constructing feasible shunting plans. The model employs a Branch-and-Cut-and-Price framework to solve a relaxation of the Train Unit Shunting and Service problem. The foundation of this model is based on the Multi-Agent Pathfinding problem, incorporating waypoints and durations.

We conclude this section with a paper by [Peer et al., 2018] that explores TUSP using Deep Reinforcement Learning. The authors argue that the heuristic solutions employed at NS struggle to account for uncertainties during plan execution and do not adequately support replanning. Moreover, these solutions often lack consistency. [Peer et al., 2018] addresses TUSP by formulating it as a Markov Decision Process. To apply Deep Reinforcement Learning (DRL), the authors needed a visual representation of the state space, depicted as an image where different regions are assigned distinct meanings. Their results indicate that DRL agents solved fewer problems compared to the local search algorithm.

1.6 NS approach for TUSP

The approach used at NS, is based on the local search algorithm proposed in [Broek, 2016]. The local search employed a technique called Simulated Anneal-

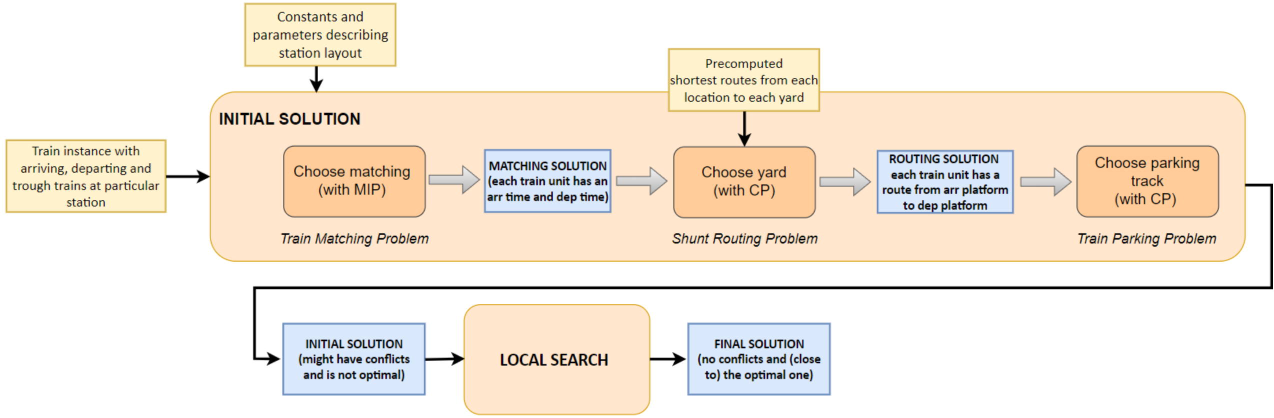


Figure 1.4: An overview of the solving technologies (orange oval rectangles), required inputs (yellow rectangles), and output (blue rectangles) used to solve TUSP at NS.

ing. An interested reader is referred to this resource [Delahaye et al., 2019] for an extensive introduction to Simulated Annealing.

The local search requires an initial solution which is obtained by decomposing TUSP and separately solving 3 of the subproblems - *Train Matching, Shunt Routing,* and the *Parking Problem.* All collections of algorithms are jointly referred to as *HIP (Hybrid Integrated Planning Method).*

Figure 1.4 presents an overview of the entire solving process. Initially, an instance is provided, which includes all arriving trains with their arrival times, tracks, and compositions, as well as all scheduled departing trains with their departure times, tracks, and required compositions. Other essential inputs include constants and parameters describing the station layout, which are fixed for each instance. On Figure 1.4, the yellow boxes represent the inputs, while the orange box is a solving step that could also be divided into smaller steps, for instance, the steps for solving the initial solution.

The figure also illustrates the order of solving the subproblems. We first solve the *Train Matching problem (TMP)* using *Mixed Integer Programming.* TMP decides which arriving train will serve which departure service. The output of the TMP is then used as input for the *Shunt Routing problem (SRP)*, which subsequently provides input for the *Train Parking problem (TPP).*

This thesis is devoted to the *Shunt Routing problem.* The following sections provide a formal description of SRP and define a model, which is addressed using *Constraint Programming (CP).* During an earlier stage of the thesis, a separate model for TMP was also developed. However, as it is not central to the thesis, it is included in Appendix A. The Train Parking problem is already addressed by NS using CP. After all subproblems are solved, the *Initial solution* is constructed. This solution may still contain some conflicts and as explained in Section 1.3, it is seldom optimal. However, in the next step, this solution is iteratively refined by the *Local search.*

1.7 Research questions

The main research question and goal of this thesis:

How can we design a general, applicable to any station layout, constraint programming model that solves the Shunt Routing Problem at NS?

To evaluate the model, the following additional research questions will be answered:

1. *How could performance be further improved?*
2. *How does the solution approach perform under changes in the instances' features, such as the station layout alternation, and the addition of routes?*
3. *What is the quality of solutions the model produces?*
4. *How do other solvers compare in terms of performance?*

Chapter 2

Shunt Routing Problem

Figure 2.1: Two tracks connected with a switch.

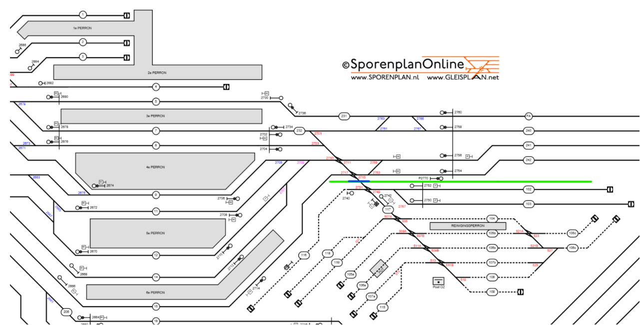


Figure 2.2: Part of the layout of Utrecht central station. The green line indicates the beginning of the yard. All tracks below it are considered part of the yard. The small blue line indicates the only entry point of this yard, the gateway. In general, a yard can have multiple gateways.(retrieved from *www.sporenplan.nl*)

2.1 Shunt Routing Problem

This section begins with a high-level description of the problem. Next, important concepts are explained and problem-specific terminology is introduced in Section 2.1.2. Finally, a formal definition of the problem is stated in 2.1.3.

2.1.1 A high-level description

When a train arrives at its final destination, typically a platform at the terminal station of its scheduled route, it must be routed either to a shunting yard or to the departure platform to commence its next scheduled journey. The focus of the subsequent sections is on the routing of trains from the end platform of one journey to the start (departure) platform of the next one, ensuring collision-free operations.

Before delving deeper into the problem, we will introduce some specific routing concepts to provide the necessary background.

2.1.2 Routing concepts

A ***track*** is a part of the railway line between two switches. The ***switch*** connects two tracks and allows the train to switch from one track to another (Figure 2.1).

Depending on their purpose, some tracks have a special name, for instance, a ***platform***. A ***platform*** is a track used for the on/off-boarding of passengers. Additionally, the track that provides access to a yard is referred to as a ***gateway***. A ***yard or (shunt yard)*** is the place where trains are routed to remain when not in use, in order to not interfere with the other scheduled trains passing through the station. Figure 2.2 illustrates one of the yards at Utrecht Central Station and its corresponding gateway. At the yard, trains undergo internal and external cleaning and any necessary maintenance, collectively referred to as ***service tasks***.

Next, the ***railway infrastructure*** of a station (also ***station layout***) encompasses all tracks and switches within the entering and leaving points of the station. The infrastructure outside these points is irrelevant to the shunting processes and, therefore not considered in this thesis.

A ***route*** refers to a path between two tracks, for instance, the platform and the gateway of the shunting yard. A route can consist of one or more ***subroutes***, each of them - a sequence of tracks and switches. ***During a subroute, the train is permitted to stand still only at the starting point and/or the endpoint of the subroute.*** Furthermore, recall that our objective is to determine routes between the end platform of one train journey and the start of the next one for each train that this station is the terminal station. However, some trains also require routing to the yard, typically for cleaning purposes. Furthermore, trains cannotusually remain at the platform, as they will obstruct the arrivals of other trains. This requires planning for multiple movements per train. Recall also, that the routes from the station entry point to the platform, and from the platform to the station exit, were predetermined during an earlier planning phase.

Tracks vary in length and are further divided into

sections, the smallest unit of an infrastructure that we consider. Sections are important as they can detect the presence of a train.

Figure 2.3 displays two different routes $R1$ and $R2$, both terminating at section $T7$. For instance, route $R1$ consists of the following sections - $[T4, T5, T2, T6, T7]$.

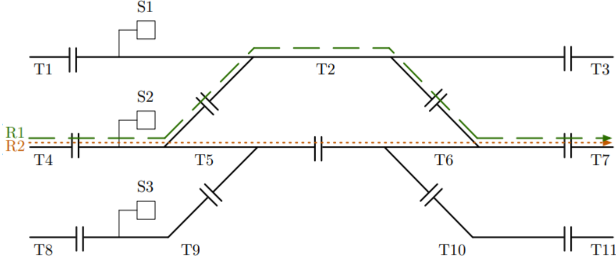


Figure 2.3: A track and section layout of a fictive station with two routes ($R1$ and $R2$). (source [Cappart and Schaus, 2017]).

Without careful planning, trains will be required to frequently stop and start, wasting energy and time. Moreover, without predefined routes, trains may unintentionally move towards each other on the same track from opposite directions.

Once a train's route is established, the corresponding infrastructure is reserved to prevent the scheduling of other trains on it. The reservations consider also the headway times, defined for safety. During these reservations, no other trains are permitted to use the same infrastructure. An exception is when we intend to park the trains, then multiple trains can occupy a single shunting track/block/section at the same time.

Consider Figure 2.4. It illustrates the sections traversed by a train over time. The x-axis denotes the sections along the route, while the y-axis represents time. Each section remains reserved during the entire duration marked by the dashed and filled blue rectangles. The blue-filled rectangle indicates when the head of the train is moving over the section. We can see when the reservation of each section starts and ends. To ensure a safe distance from the previous train (the headway time), the sections are occupied before the movement, as indicated by the dashed rectangle appearing before the filled blue one.

Finally, before proceeding with the formal definition of the Shunt Routing Problem (SRP), we acquaint the reader with the concept of a ***reversal***:

Train movements are restrained, particularly concerning turns. They cannot easily perform sharp turns. Figure 2.5 (first picture) the train can transition from track B to C, but not to A. There is a switch at the point where the tracks meet. Nevertheless, it is still possible for a train to reach track A by first moving to track C, stopping, changing the direction of movement, and then proceeding to track A (again see Figure 2.5, second and third picture). This movement is known as a ***reversal***. Reversals are generally avoided because they are time-consuming. To change the direction of

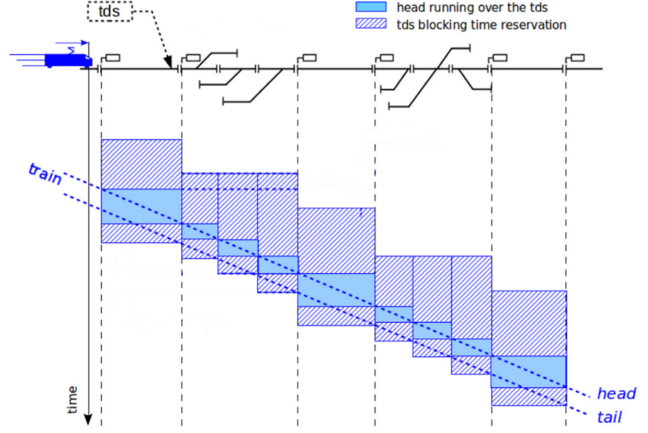


Figure 2.4: The figure is slightly changed, from paper by [Marlière et al., 2023]). In the figure, "tds" refers to a track detection section which is simply a section.

movement, the driver must traverse the entire train length.

2.1.3 Formal definition of SRP

The Shunt Routing Problem (SRP) is the problem of finding feasible routes between platforms and shunting yards for all shunt trains at a given station infrastructure. [Lentink, 2006].

A route is considered feasible if it does not simultaneously utilize infrastructure that is already reserved by another train. Infrastructure may be reserved by the predetermined routes between the entrance points of the station and platforms. It could also be reserved for maintenance activities, or by stationary trains on platforms.

Besides the primary task of SRP to determine conflict-free routes for shunt trains, several additional objectives can further refine the solution. For instance:

- Overall travel distance.
- Number of changes in directions (number of reversals).
- Number of overall scheduled movements.
- Minimize routing time.
- Minimize time on the arrival and departure platforms.
- Increase the time in the shunting yard (implemented in experiment 2 [Wattel, 2021]). However, maintaining a balance is important, as otherwise, this might complicate the parking problem.
- Balance the use of the shunting yards by shunting an equal amount of trains (implemented in experiment 3 [Wattel, 2021]). Alternatively, this balance could also be achieved by ensuring the total duration that shunt trains spend in the various yards is approximately equal.
- Combinations of the above.

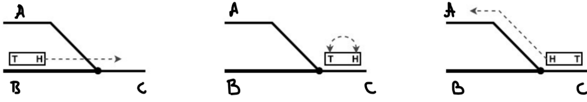


Figure 2.5

An important characteristic of SRP is that while trains have fixed arrival and departure times at the station, the timing for shunting operations between platforms and shunt tracks is flexible. For example, the departure time of a shunt train from the platform to a shunt track can be scheduled within a time interval that starts when the train arrives at the platform and ends before the next train arrives at the same platform.

2.2 Constraint programming

Constraint Programming is a powerful paradigm for solving combinatorial problems [Rossi et al., 2008]. Constraint Programming is built as a generalization of SAT solving. It provides a more expressive language that extends beyond the boolean variables in SAT and includes integers. Furthermore, various constraints are designed, and a generalization of unit propagation, the core algorithm in SAT solvers, is provided.

A constraint is a relation between usually multiple variables that limits the values these variables can take simultaneously. Furthermore, the constraints are not limited to linear as in Mixed Integer Linear Programming (MILP). As some types of constraints can be found in various applications, people have decided to define them and implement efficient algorithms. These algorithms are applied to narrow down the feasible solution space that otherwise needs to be traversed by the solver. Those algorithms are called propagators and the constraints are known as global constraints. The global constraints are considered the main strength of CP.

The search strategy of the solver is based on backtracking. When propagators are unable to further reduce the search space by narrowing the domain of some variables, the solver selects a variable and assigns a value from its current, possibly already reduced, domain. Next, based on the solver's decision, further propagations (deductions) about the domains of other variables may occur. When propagators can no longer reduce the search space, the solver selects the next unassigned variable and assigns a value from its domain. This process continues until either a solution is found (with all variables assigned and no constraint violations) or a conflict is detected. Upon encountering a conflict, the solver backtracks to previous assignments and identifies the combination of values that led to the infeasible solution to avoid repeating it. This process repeats until either the entire search space is explored, at which point the instance is deemed infeasible if no solution is found, or a solution is found. This is a high-level overview of the solver's search process. For a more extensive discussion, refer to [Edelkamp and Schrödl, 2012].

To solve a problem with CP, *one should provide a declarative description of a problem as a set of decision variables with their domains, and a set of constraints restricting the combinations of values.* This will be the goal of the upcoming sections. Before delving into the model, we explain one recent concept in the CP domain that will be utilized: optional time-interval variables.

Optional time-interval variables

To model the timing of route execution, we utilize a specific scheduling concept in CP called conditional time-interval variables. This type of variable represents a period (an interval) in which a particular event occurs, defined by its start, end, and duration. The interval variable can also be optional, meaning it may or may not be part of the solution. Formally, the domain of the interval variable is a set of all possible intervals with the earliest possible start time and latest possible end time, or $\perp$ if not executed: $\perp \cup [s, e)|s, e \in Z, s \leq e$ [Philippe Laborie, 2018].

Handling a non-present variable varies depending on the constraint. However, in general, a non-present interval variable is not considered by any constraint or expression that involves it, and the constraint containing it is considered satisfied. For example, if a non-executed time-interval variable a is used in a constraint with a time interval variable b, this constraint would not impact time-interval variable b.

2.3 Model

This section addresses the main research question 1.7 by proposing a generic model for SRP. The scope of the model is defined in Section 2.3.1, followed by the requirements in Section 2.3.2. Assumptions are stated in Section 2.3.3. Sections 2.3.4 and 2.3.5 specify all constants and the provided input per instance, respectively. Section 2.3.6 defines all variables for the model, while Section 2.3.7 presents the constraints. Section 2.3.9 introduces further extensions to the base model. Finally, Section 2.3.9 introduces further extensions to the base model, and Section 2.3.10 specifies the output.

2.3.1 Scope of the model

The model considers the station area, which is the region between the entering and leaving points. Furthermore, the routing of trains will be to the gateways of the yard. Routing within the shunting yards is not considered in this model. The next subproblem of TUSP, namely the Train Parking Problem (TPP), will determine the specific track where the train will be parked, and then the route within the yard will be fixed.

The scope of the model is limited to scheduling routes for the shunt trains, which involves allocating a route and a time window to execute it. Finding the routes is outside the scope of the model, but allocating shunt trains to routes is within. All available routes

are part of the input and are predetermined. Further details will be provided in the subsequent subsections.

2.3.2 Requirements

The main model should satisfy all requirements that we have summarized below. We have further made some additional extensions which will be presented later and implemented if time permits.

R1: The primary requirement of the model is to produce a correct schedule. That means:

1. No train is moving over infrastructure already reserved by another train. In particular, sections are occupied by only one moving train at a time.
2. Furthermore, all trains are planned, and at every point in time, we know the precise location of each train. Specifically, there should be no unplanned time gaps between consecutive train actions, such as between a movement to a yard and parking.
3. We not only know the positions of the trains but also ensure that their movements between locations are planned and routes are reserved for the appropriate duration. (Trains do not magically transfer from one location to another.)

   For instance, if the solver chooses to park the train in one yard and the next movement is to the departure platform, this movement should begin from the yard where the train is parked, and not from any other location.

R2: The routes of all through trains which are predetermined and fixed, are respected by the model.

R3: Train units that are combined at the yard, follow together the same route from the yard to the platform. Similarly, train units that have not been yet split, are simultaneously steered along the same route from the platform to the yard. This ensures that the train operates as a single unit.

R4: The model utilizes the section occupations specified by HIP and reserves the routes according to them (more explanation in later sections).

R5: The model routes each train to at most two yards.

R6: A train that arrives at and departs from the same platform, is required to transition to a yard.

The last requirement is imposed, because, if the train were not mandated to be routed away from the platform, its schedule would have been already established in prior planning phases. Additional requirements that will be fulfilled upon incorporating the extensions and objectives from Appendix C.1 include:

R7: There is enough time at the yard for the train to execute the required services.

R8: The capacity of each yard is satisfied at each point in time.

R9: The number of the scheduled movements for each train is as least as possible if objective from C.2.1 is specified.

Let us clarify the difference between a movement and a route. A ***movement*** (or ***reposition***) denotes the action of transferring a train from point A to point B. While a ***route*** refers to an option for executing such a movement from point A to point B. There may exist multiple routes that accomplish the repositioning of the train between A and B.

2.3.3 Assumptions

A1: We assume that it is known for each train whether it will undergo splitting or combining with another train. Additionally, it is assumed that splitting occurs at the *first* yard upon arrival, while combining occurs at the *last* yard just before departure.

A2: We assume that the last section of the route serves as the parking location for the train upon completing that route. Similarly, when a train begins a new route, it starts from the section where it was parked. Furthermore, we also assume that parking is performed only on one section, namely the last and first section of the previous and next route.

A3: We assume that the matching is chosen in such a way that the service time is abundantly respected. This guarantees that the input data is not infeasible when including the extensions.

A4: We assume that each yard at the station has infinite capacity for now.

A5: Assume that all types of services can be executed at each yard.

A6: We assume that through trains are not occupying the platform when a plannable train is scheduled to arrive or depart.

The assumptions **A4** and **A5** can be relaxed by considering the Extensions in Appendix C.1. Assumption **A6** was introduced to simplify the model. In practice, is allowed a train to approach a platform if another train is standing still there.

2.3.4 Parameters

We define the following constant for each instance of the SRP problem:

- *Types* - all available subtypes of train units.
- *dw* - a time for off-boarding and on-boarding of passengers.
- *maxMovements* - the maximum number of movements for all trains equal to $min(3, numYards + 1)$ Adhering to requirement **R5**, we limit ourselves

to at most three movements. In practice, three repositions correspond to scenarios where the train moves from the arrival platform to one yard, then to another yard, and finally to the departure platform. A third-yard transition is considered highly undesirable. One could consider allowing for a fourth movement option only if it involves returning to the first yard before proceeding to the departure platform. We proceed with at most three routes.

If there is only one yard at the station, $maxM$ will be at most two since at most two routes (one to the yard and one from the yard to the departing track) are meaningful.

We also define two more parameters:

- $maxM_t = maxMovements - (t \text{ is part of split}) - (t \text{ is part of combine})$ If the train is part of a split, the initial movement from the platform to a yard is executed by the arrival train. However, we must still comply with the requirement (**R5**) (Section 2.3.2), which states that a train can go to a maximum of two yards. Therefore, the maximum number of movements is reduced by one in case of split or combine.
- E_t - all possible endpoints for train t, including arrival and departing platform, all yards, and a dummy place (denoted with 0).

2.3.5 Input

A train that arrives at the station may undergo a split and\or a combine. Usually, when a train is split, the new resulting parts have different departure platforms, hence we cannotdefine a unique departure platform of a such train. For this and similar reasons, we differentiate the trains as follows:

- ***Arrival trains*** - these are the trains that arrive at the station and finish their scheduled journey there, meaning this is their last station. An arrival train ceases to exist after splitting or if no split occurs, then after it arrives at the station, it becomes a shunt train, a type explained below.
- ***Shunt trains*** - These trains are *part of* the arrival trains (***they could also represent the whole train if no split has occurred***). They constituted also a part of (or the whole) the departing composition. The exact part is decided by the matching, solved before the SRP. We further differentiate between those trains as follows:
  - ***base shunt trains*** - all shunt trains that are neither part of split nor combined, we will denote them as BT. In particular, they correspond to the whole arrival and departure compositions.
  - ***split shunt trains*** - all trains that will result from a split of an arrival train, but are not going to be combined with any other, (denoted as ST).
  - ***combined shunt trains*** - all trains that will be combined, but have not been part of a split (denoted as CT).
  - ***split-combined shunt train*** - a train that is both a part of a split and a combine (denoted as SCT).
- ***Departure composition*** - a composition of the types of train parts/units, needed for this departure. These train parts/units are unknown before the matching, however, in the SRP we know each shunt train that can be base, split, combined, or split-combined, in which departure composition will depart.

We now proceed with the input:

- T - all shunt trains ($BT \cup ST \cup CT \cup SCT$) and all arrival trains (AT) and departure compositions (DT) that do not have already fixed plan for movements on the station.
- R_i^t - all possible routes that a train t can undertake for one reposition, where i represents the specific movement number.
- r - a route that can consist of one or more subroutes.
- $start(r)$ - start of route r, start(r) $\in E_t$.
- $end(r)$ - end of route r, end(r) $\in E_t$.
- $d_{r'}^t$ - duration for subroute r' that is part of r and the route is taken by train t (r' also be the whole route r).
- $numSub_r$ - One route can consist of a few separate subroutes. This parameter shows the number of subroutes for a route r.
- dis_r - the length of route r.
- $durStop(r'_1, r'_2)$ - the minimum duration required between two subroutes r'_1 and r'_2, where $r'_1, r'_2 \in r$, for some route r.
- R - all considered/available routes for trains in T, $R = \bigcup_{t \in T} R^t$, where $R^t = \bigcup_{1 \leq i \leq maxM_t} R_i^t$.
- S_k^r - all sections of a given subroute of r. A route is split into subroutes when the train needs to perform a reversal halfway through the route. $1 \leq k \leq numSub_r$.
- S^r - all sections of a given route, $S^r = \bigcup_{1 \leq k \leq numSub_r} S_k^r$.
- S - all sections used by trains in T, $S = \bigcup_{r \in R} S^r$.
- Y - Set of all yards at the station. Furthermore, for each yard $y \in Y$ we know:
  - G_y - a set with the gateways of y.
- G - all gateway tracks for the station.

The minimum duration - $durStop(r'_1, r'_2)$, could be substituted with a variable whose domain starts from $durStop(r'_1, r'_2)$ and extends further. The rationale for this addition is that in busier and larger stations, where the through traffic is high and trains frequently need to perform reversals between platforms and yards, it may be unfeasible to plan a compound route with a fixed duration in between. We adapt the model to include such a variable and the necessary constraints, as detailed in Appendix B. This extension introduces an additional layer of complexity. Therefore, we proceed further with the model without these additional variables.

2.3.5.1 Input for section occupations

Each route consists of a sequence of sections with predefined start and end times in seconds relative to the time at which train t begins the route. For example, the following information will be included as part of the input for a specific route r:
$[(s_1, start_{s_1}, end_{s_1}),$
$(s_2, start_{s_2}, end_{s_2})...$
$(s_n, start_{s_n}, end_{s_n})]$

Hence, we can define the following parameters for each section $s_i \in S^r$ and $r \in R_t$ and train t:

- $start^t_{s_i,r}$ - the start of the section occupation relative to the commencement of route r by train t.
- $end^t_{s_i,r}$ - the end of the section occupation again relative to the start time of route route r by t.

2.3.5.2 Input for each train t

For each arrival train $at \in T$, we know:

- $init_{at}$ - the initial point of train at, considered in the planning. That could be a platform, another track, or a gateway, represented by a trackID.
- $aside_{at}$ - arrival side of train at, can be {0, 1}.
- arr_{at} - arrival time of at at $init_{at}$ in seconds.
- $at \rightarrow \{t_1, t_2\}$ - the arrival train is going to be split to t_1 and t_2, where t_1 and t_2 are either split shunt trains (ST) or split-combined shunt trains (SCT), depending on whether they will combine as well. If no split is to be performed, then at is matched to one shunt train ($at \rightarrow \{t_1\}$). In this case, this shunt train is either a base shunt train or a combined train.

For each departing composition $dt \in T$, we know:

- des_{dt} - departure (destination) point (platform, side track).
- $dside_{dt}$ - departure side, the side of the track at which the departure will happen.
- dep_{dt} - scheduled time for departure from destination des_{dt}.
- $dt \rightarrow \{t_3, t_4\}$, - each departing composition consists of shunt trains (t_3 and t_4). These trains are either $\in CT$ or split-combined trains ($\in SCT$) depending on whether they have also undergone a split from an arrival train. t_3 and t_4 will be combined and depart as a single train - dt to the departure platform. Alternatively, the departure composition could consist of a single shunt train ($dt \rightarrow \{t_3\}$), in this case, t_3 is $\in ST$ (split shunt train) or $\in BT$ (base shunt train).

For each base shunt train $t \in T$ we know the corresponding arrival train and to which departing train it is matched, hence we also know:

- $type_{tu}$ - the type of each $tu \in t$, $type_{tu} \in Types$
- $init_t$ - the initial point of train t (the initial point of the arrival train that this shunt train was part of)
- $aside_t$ - arrival side of train t, can be {0, 1}
- arr_t - arrival time of t at $init_t$ in seconds
- des_t - destination point (platform, side track) of the scheduled departure - dep_t, deduced from the departure composition that this train will be part of
- $dside_t$ - departure side
- dep_t - scheduled time for departure from destination des_t
- $t \rightarrow dt$ - the matching telling us which train of which departure composition is part of

To avoid repetition, we will specify which information compared to the information above for the base shunt trains is irrelevant and missing.

For each split shunt train $t \in T$ All arrival information such as $init_t$, arr_t, $aside_t$ is not present, as the train only starts to exist after the split occurs in the yard.

For each combined shunt train $t \in T$ All departure information such as des_t, dep_t, $dside_t$ is not present, as the train does not exist after the combine at the yard is performed.

For each split-combined shunt train $t \in T$ For this type of train, both arrival and departure information is not present.

2.3.5.3 Input for through trains

Each through train has a schedule with predetermined routes for its station activities. These trains enter the station through the designated entry/exit points, follow a predetermined route to reach the platform for passenger boarding and alighting, and then depart the station via another fixed route. In addition, arrival trains and departure compositions have also a single already scheduled route to and from a platform, respectively.

For each of those already predetermined routes and parking times at the station, the corresponding section occupations must be identified and incorporated into the model. Based on the start of the route and the relative start and end times, we define a constant interval - $occ^{pe}_{s_j}$ for each section s_j involved in any predetermined event - pe that requires infrastructure reservations, such as movements, parking, etc.

2.3.6 Variables and function definitions

In the following text, t can be an arrival train, shunt train, or departure composition. Recall that an arrival train can at most have one movement, namely from the arrival platform to the first yard since afterward will be split. However, if the arrival train does not require a split at the yard, it will not be considered and no variables will be introduced for it. In such cases, we will plan the first movement for the corresponding shunt train instead.

Similarly, if the departure composition is created by combining two shunt trains, we plan one movement from the yard to the departure platform. However, if the departure composition consists of a single shunt train, then we do not plan for the departure composition, but for the shunt train itself.

We begin with introducing a variable representing the number of movements that train t will perform at the station. For each reposition, the train will choose one route from a set of available routes - R^t_i.

- $numr_t$ - the number of taken repositions/movements for train t during the whole stay of the train at the station. The domain of this variable is [0, $maxM_t$].

  The number of movements can be 0 ($numr_t = 0$) if the shunt train is split and combined in the same yard, or if it arrives at the yard and is combined there. Similarly, if it departs from a yard where it was previously split. The last situation is if it arrives and departs from the same yard.

  We do not allow the train to remain at the platform if it arrives and departs from the same platform. If that was possible, the train would have already been managed in prior planning stages.

We revisit now R^t_i and clarify the types of routes that can belong to this set. Recall that R^t_i denotes the set of available routes for the i-th repositioning of train t. Depending on the type of the train, some set of routes could be excluded.

- R^t_1 - contains all possible routes from the arrival point $init_t$ to each gateway for each yard at the station and from the arrival point to the departure platform dep_t, $t \in T \backslash \{ST, SCT, DT\}$. We exclude those types of trains as they do not execute a movement from the arrival platform.
- R^t_2 - contains all routes from each gateway to the departure platform, as well as from each gateway to all gateways of the other yards, where $t \in T \backslash \{AT, DT\}$.
- R^t_3 - all routes between each gateway of a yard and departure platforms, $t \in T \backslash \{AT, CT, SCT\}$.

We define for each train $t \in T$ and $r \in R^t_i$, $1 \leq i \leq maxM$ the following interval variables and functions:

- $a^{t,r}_i$ - an optional time-interval variable for the period when the train will execute route r for movement i.
- $s(), e(), d()$ - functions that return the start, the end, and the duration of interval variables, respectively.
- $pres()$ - a boolean function that returns the presence status for an optional time-interval variable a. If a certain route r is not chosen for train t, then the optional variable $a^{t,r}_i$ is not present.
- The domains of the start $(s(a^{t,r}_i))$ will be $[arr_t, dep_t - d^t_r]$, meaning the start of the interval will occur between $[arr_t, dep_t - d^t_r]$. Analogously for the end $(e(a^{t,r}_i))$. The duration is fixed to the duration of the route $d(a^{t,r}_i) = d^t_r$). The domain can be further tightened in case the train needs to be shunted to the yard and certain service tasks are required.

  The duration of route r (d^t_r) is minimum the sum of all durations of all subroutes in route r - $d^t_r = d(a^{t,r}_i) \geq \sum_{r' \in r} d^t_{r'}$. Hence, the end time of the period is fixed once the start time $s(a^{t,r}_i)$ is chosen since $e(a^{t,r}_i) = s(a^{t,r}_i) + d(a^{t,r}_i)$.

Additionally, for each train t and each possible movement i:

- A^t_i - high-level optional interval variable for all variables $a^{t,r}_i$ with $r \in R^t_i$, so essentially all optional variables for one reposition of the train.

Figure 2.6 represents how the first high-level group variable A^t_1 and the route variables for all possible routes are organized. For this first reposition of the train, the train can choose out of three different routes - $\{a^{t,r_1}_1, a^{t,r_2}_1, a^{t,r_3}_1\}$. The ***alternative*** constraint ensures that the train will follow only one of the routes, by selecting only one route variable to be present. We will refer back to this constraint in Section 2.3.7.4.

Last, we define:

$occ^{t,r}_{s_j}$ - an interval variable for the occupation of section s_j by t for route r.

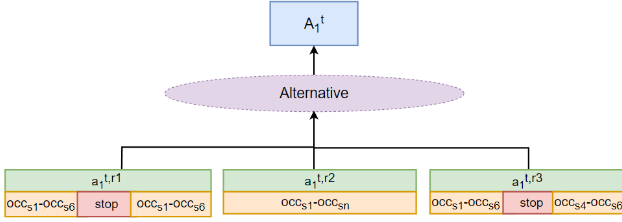


Figure 2.6: Breakdown structure of the route variables - the high-level variable A_1^t and the variables for each route - a_1^{t,r_j} and together with the section occupations and the stop time. An alternative constraint 2.9 models the relationship between A_1^t and a_1^{t,r_j}.

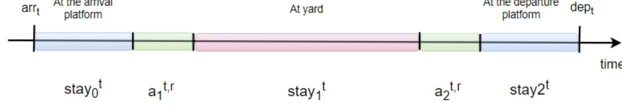


Figure 2.7: A typical shunt plan for a train t. The blue rectangles represent the time intervals when the train will be at the arrival or departure platforms. The green ones show the periods when routing will happen. The red one indicates the time when the train is located in a specific yard. arr_t and dep_t denote the arrival and departure times, respectively.

2.3.6.1 Variable for parking between movements

Generally, after each reposition, the train may remain on the track for an unspecified duration. As these durations are not predefined, we will use an optional interval variable to represent the period during which the train is parked on a specific track after each route. We introduce an optional variable for $\forall t \in T, i \in \{1, maxM_t + 1\}$:

- $stay_i^t$ - an *optional* interval variable for the parking after movement i.
- $stay_0^t$ - a *mandatory* variable for the first parking. Each train is parked at least once.

For arrival and departure trains, only one parking variable is required for the parking at the platform after arrival or before departure. According to assumption **A1** in Section 2.3.3, the train will be split immediately after its first move, thus no parking is needed at the yard. Similarly, it will be combined just before it begins the move to the departure platform, so no additional parking is required.

Domains for the start, end, and duration of $stay_i^t$:

- $d(stay_i^t) \in [0, dep_t - arr_t]$
- $s(stay_0^t), e(stay_0^t) \in [arr_t, dep_t]$.

Figure 2.7 and 2.8 illustrate two types of shunt plans. Both figures have abstract details such as the concrete route paths that the train will traverse. Figure 2.7 depicts the scenario when the train will be parked at the yard as opposed to Figure 2.8 where

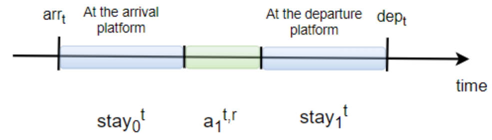


Figure 2.8: A shunt plan for trains that will not require shunting to the yard but will be immediately reallocated from the arrival platform to the departure platform.

going to the yard is not required.

The distinct routes have different numbers of subroutes. For instance, as shown in Figure 2.6 a_1^{t,r_1} and a_1^{t,r_3} consists of two subroutes, while a_1^{t,r_2} have one subroute. The interval variable a_1^{t,r_3} with section occupations can be seen in Figure 2.9. The light orange boxes represent the section occupation intervals. The red represents the time interval for the stop between the subroutes. In this model, we take for the stop the minimum time required between two submovements.

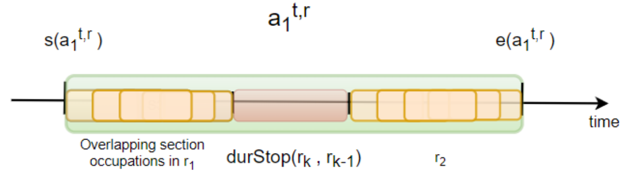


Figure 2.9: Visualization of the time-span of the different time-interval variables - route variables, section occupations, and duration between the subroutes.

2.3.6.2 Introduce variables for endpoints of each reposition:

If the train is planned to perform only one movement, in particular from the arrival to the departure platform, there are no intermediary stops. However, once more than one movement is demanded, then we need to know where the train is positioned. If the train performs 2 movements, an intermediary position between the two routes is required.

Therefore, for each reposition $1 \leq i \leq maxM$ we also define a variable that represents the end place of the reposition. Since the end of one reposition corresponds to the beginning of the next one, only one variable is necessary.

- e_i^t - an integer variable with domain $[0, |E_i^t|]$. Recall E_i^t are all places where train t can be located after movement i - all endpoints of all routes in R_i^t and dummy position. The dummy position is utilized when the number of movements for a train is less than $maxM$. For instance, if the train is routed directly from the arrival to the departing platform, only two endpoints to denote the arrival and departure platform are utilized. Any addi-

tional endpoints remain unused and are assigned a value of 0.

2.3.6.3 Channeling variable

To facilitate expressing constraints between group variables A_i^t, route variables ($a_i^{t,r}$) and section occupation variables ($occ_{s_j}^{t,r}$), we introduce a variable for each movement i and train t:

- $chosenRoute_i^t$ - an integer variable for the index of the chosen route from R_i^t. The domain is $\{-1, (\#\text{of routes} - 1)\}$, where -1 is assigned if the movement is not performed (no route is taken).

2.3.7 Constraints

The following constraints are $\forall t \in T, 1 \leq i \leq maxM_t$ and $r \in R_i^t$ or simply R^t (all routes for all movements for train t) if i is not in the equation.

2.3.7.1 Constraining the start and end times for section occupations

To simplify the constraints for the start and end times for the section occupations, we introduce an integer channeling variable - $startOfSubm$ for the start of each subroute (submovement) of each route $r \in R$ for the routes of all trains. Code snippet 1 introduces the required constraints for this variable:

Algorithm 1 Constraints for the start of each submovement variables $startOfSubm_k^{r,t}$, where $k \in \{1, numSub_r\}$

```
1: if (k == 1) then        ▷ The start of the first
   submovement coincides with the start of the route
2:     startOfSubm_k^{r,t} = s(a_i^{t,r})
3: else        ▷ The start of each other submovement
   is the start of the previous + the duration of the
   previous + the min duration required
4:     startOfSubm_k^{r,t} = startOfSubm_{k-1}^{r,t} + d^t_{r_{k-1}} +
   durStop(r_{k-1}, r_k)
5: end if
```

The start times of the section occupations:

Sections within each subroute are categorized based on their position to determine the start times of their occupations. These sections are classified into "first sections" $S^r_{first,k}$ and middle and last sections - $S^r_{last,k}$.

- $S^r_{first,k}$ - This set comprises sections that the train occupies at the beginning of subroute k of route r.
- $S^r_{last,k}$ - These sections are the sections on which the train will remain after completing subroute k and until the next subroute begins.

All other sections are considered "middle sections".

The pseudocode 2 presents the logic of how sections' occupation start times will be inferred. We explain the code briefly. Sections in $S^r_{first,1}$ are the initial sections of the route where the train is positioned prior to routing. They must be reserved for parking, provided the route commences outside of a yard (such as at an arrival platform). Hence, to calculate their start times, we must subtract the parking duration from the start of the movement. In all other cases, the section occupations' start times are equal to:

- If $s_j \in S^r_{first,k}$, the section is in the first sections of the subroute, and their occupation start times are aligned with the start of the subroute.
- Otherwise, the start times are determined relative to the route start, adjusted by $start^t_{s_j,r}$.

The duration for the intermediate stops is accommodated by the end time of the section occupation, a topic covered in the next paragraph.

Algorithm 2 Code snippet for the different constraints introduced in different situations for the start times of section occupations for $s_j \in S^r_k, i \in \{1, maxM\}$

```
1: if k == 1 then                          ▷ the first subroute
2:     if e_0^t ∉ Y then              ▷ at a platform (not yard)
3:         if s_j ∈ S^r_{first,k} then      ▷ the first sections
4:             s(occ_{s_j}^{t,r}) = startOfSubm_k^{r,t} − d(stay_i^{r,t})
5:         else▷ all other sections of the first subroute
6:             s(occ_{s_j}^{t,r}) = startOfSubm_k^{r,t} + start^t_{s_j,r}
7:         end if
8:     else               ▷ first subroute, but not at arrival
   platform
9:         code snippet 3
10:    end if
11: else                        ▷ other subroutes of any route
12:    code snippet 3
13: end if
```

Algorithm 3

```
if s_j ∈ S^r_{first,k} then                ▷ the first sections
    s(occ_{s_j}^{t,r}) = startOfSubm_k^{r,t}
else           ▷ all other sections of the first subroute
    s(occ_{s_j}^{t,r}) = startOfSubm_k^{r,t} + start^t_{s_j,r}
end if
```

The end times of the section occupations:

The constraints for the end occupations times of the last sections of each subroute also differ. It depends on the train's destination (platform or yard) and whether the last sections of the subroute are also the last sections of the entire route.

The pseudocode 4 summarizes the constraints for the end occupation times for all sections. We proceed with brief explanation. If the route ends at a yard, the occupation times of the last sections are not extended with the time for the parking (line 4). However, if we are not going to a yard, then it must be a departure platform, we extend the occupations of the variables with $d(stay_i^t)$.

Last, if this is not the last subroute then we extend the end of the occupation variables with the $durStop(r_{k-1}, r_k)$ to reserve for the time spent at an intermediate stop (line 9). In all other cases, we add the given offset ($end^t_{s_j,r}$) to the start of the subroute.

Algorithm 4 Code snippet for the different constraints introduced in different situations for the end times of section occupations, $\forall s_j \in S^r_k$, for $i \in \{1, maxM\}$ and $k \in \{1, numSub_r\}$

```
1: if s_j ∈ S^r_{last,1} then                        ▷ the last sections
2:    if k == numSub_r then   ▷ the last subroute of the r
3:       if e^t_i ∈ Y then      ▷ if the endpoint of the route is a yard, we should not occupy the sections
4:          e(occ^{t,r}_{s_j}) = startOfSubm^{r,t}_k + end^t_{s_j,r}
5:       else  ▷ not a yard, extend the last sections to be occupied while parking
6:          e(occ^{t,r}_{s_j}) = e(stay^t_i)
7:       end if
8:    else
9:       e(occ^{t,r}_{s_j}) = startOfSubm^{r,t}_k + d^t_{r_k} + durStop(r_{k-1}, r_k) = ▷ Here k is at most numSub_r - 1, not a last subroute, but last sections
10:   end if
11: else                                                ▷ Not last sections
12:   e(occ^{t,r}_{s_j}) = startOfSubm^{r,t}_k + end^t_{s_j,r}
13: end if
```

Finally, the duration of an occupation is completely determined by:

$$d(occ^{t,r}_{s_j}) = e(occ^{t,r}_{s_j}) - s(occ^{t,r}_{s_j}), \forall s_j \in S^r \quad (2.1)$$

On a final note, if a train is split or combined from/to a through train at the arrival/departure platform, then the section occupations constraints for the sections corresponding to the platforms (i.e., sections in $S^r_{first,k}$ and $S^r_{last,1}$) will differ. This will be discussed later.

2.3.7.2 Constraints for presence synchronization between routes and sections occupations

A route variable $a^{t,r}_i$ is present, if route r will be taken for movement i. This is allowed if the corresponding chosenRoute variable is assigned to the index of this route within R^t_i (the set of routes for movement i).

$$pres(a^{t,r}_i) = (chosenRoute^t_i == indexOf(r, R^t_i)) \quad (2.2)$$

$pres(a^{t,r}_i)$ is true, if and only if all occupation's section variables of that route are present as well. These are the sections from all subroutes of the route.

$$pres(a^{t,r}_i) = pres(occ^{t,r}_{s_j}), \quad \forall s_j \in S^r \quad (2.3)$$

2.3.7.3 Constraints for section occupations

If the occupation of a section is present ($pres(occ^{t,r}_{s_j})$ is true), then the start and the end times of the occupation variables are constrained based on Section 2.3.7.1. All constraints are presented in the pseudocode 2 and 4. If the variable is not present ($pres(occ^{t,r}_{s_j})$ is false), the constraints are not considered.

Pseudocode 2 and 4. (2.4)

A section can be occupied by at most one train at a given time. This constraint ensures that all occupation intervals of each section are non-overlapping. We also add the constant intervals from each of the predetermined events pe from the set of all fixed events FE - movements, parking, etc.

$$noOverlap(\bigcup_{t \in T, r \in R_t} occ^{t,r}_s \bigcup_{pe \in FE} occ^{pe}_s), \forall s \in S \quad (2.5)$$

2.3.7.4 Time constraints for route interval variables and group variables

The duration of each route variable is equal to the duration of all subroutes and intermediary stops.

$$\begin{aligned}\forall r \in R^t_i \text{ if } & \\ numSub_r = 1 \text{ then } & d(a^{t,r}_i) = d^t_{r'} (r' \in r) \\ \text{else } & d(a^{t,r}_i) = \sum_{r' \in r} d^t_{r'} + \\ & \sum_{k \in \{1..numSub_r\}} durStop(r_{k-1}, r_k), \end{aligned} \quad (2.6)$$

The following constraint ensures that any group variable that corresponds to a movement greater than the number of executed movements ($i > numr_t$), is not present.

$$i > numr_t \Leftrightarrow pres(A^t_i) = 0 \quad (2.7)$$

If a movement is not taken, meaning ($pres(A^t_i)$ is false) then the duration of it should be zero.

$$pres(A^t_i) == 0 \Rightarrow e(A^t_i) - s(A^t_i) == 0 \wedge d(A^t_i) == 0 \quad (2.8)$$

The next constraint states that if A^t_i is present (meaning that the train will perform a particular reposition), then exactly one of $\{a^{t,r_1}_i, a^{t,r_2}_i ..\}$ is present, and A^t_i starts and ends together with the chosen one. Variable A^t_i is absent if and only if none of the route variables is present. This article presents detailed information about this constraint [Philippe Laborie, 2018]. Figure 2.6 illustrates this constraint.

$$alternative(A^t_i, \bigcup_{r_j \in R^t_i} a^{t,r_j}_i) \quad (2.9)$$

Unfortunately, Google-OR-Tools does not provide a global constraint for "alternative". Therefore, to achieve the required functionality, the following simpler constraints were used:

$$pres(A_i^t) \Rightarrow \sum_{r \in R_i^t} pres(a_i^{t,r}) == 1 \tag{2.10}$$

$$pres(A_i^t) == 0 \Rightarrow \sum_{r \in R_i^t} pres(a_i^{t,r}) == 0 \tag{2.11}$$

$$pres(a_i^{t,r}) \Rightarrow s(A_i^t) == s(a_i^{t,r}) \tag{2.12}$$

$$pres(a_i^{t,r}) \Rightarrow e(A_i^t) == e(a_i^{t,r}) \tag{2.13}$$

$$pres(a_i^{t,r}) \Rightarrow d(A_i^t) == d(a_i^{t,r}) \tag{2.14}$$

The variable $chosenRoute_i^t$ for movement i of train t should be assigned to -1 if and only if the movement is not executed (i is greater than the number of executed movements - $numr_t$).

$$i > numr_t \Leftrightarrow chosenRoute_i^t == -1 \tag{2.15}$$

Route one is completed before the start of route two. In the general form, as follows:

$$e(a_i^{t,r}) \leq s(a_{i+1}^{t,r'}) \qquad \forall r \in R_i^t, r' \in R_{i+1}^t \tag{2.16}$$

Specify the constraints as such, requires $|R_i^t| \cdot |R_{i+1}^t|$ (the # of routes for movement i times the # of routes for movement $i+1$) for each pair of movements i and $i+1$. However, if we utilize group variables, regardless of the number of routes per movement, we require only one constraint for each movement and train, as follows:

$$e(A_i^t) \leq s(A_{i+1}^t) \tag{2.17}$$

We proceed using the group variable for the same reason.

Between each two movements there is a parking (we allow this parking to be with zero duration). The start of the movement i is at the end of (i-1)-th parking - $stay_{i-1}^t$.

$$s(A_i^t) = e(stay_{i-1}^t) \tag{2.18}$$

The start of $stay_i^t$ is at the end of the previous movement A_{i-1}^t.

$$s(stay_i^t) = e(A_{i-1}^t) \tag{2.19}$$

Analogously to Eq. 2.17, we can also define similar constraints for the parking variables. Note that once Eq. 2.17, Eq. 2.18 and 2.19 are defined, this constraint will be redundant:

$$e(stay_i^t) \leq s(stay_{i+1}^t) \tag{2.20}$$

2.3.7.5 Constraints for endpoints of routes

The endpoint after movement i is equal to the end of the present route for movement i.

$$pres(a_i^{t,r}) \Rightarrow e_i^t = end(r) \tag{2.21}$$

The start of the present route is at the previous endpoint.

$$pres(a_i^{t,r}) \Rightarrow e_{i-1}^t = start(r) \tag{2.22}$$

The last route finishes at the departure platform, and we introduce a dummy departure yard to represent it. This constraint applies only to trains intended to reach the departure platform, hence excluding arrival trains, combined trains, and split-combined trains.

$$e_{numr_t}^t = dummy(des_t), \qquad t \in T \backslash \{AT, CT, SCT\} \tag{2.23}$$

The endpoints after the last route are assigned to a dummy endpoint:

$$i > numr_t \Leftrightarrow e_i^t = dummy \tag{2.24}$$

The first endpoint is the dummy yard representing the arrival platform. Similarly, we do not constraint the trains that do not start from the arrival platform:

$$e_0^t = dummy(init_t) \qquad t \in T \backslash \{DT, ST, SCT\} \tag{2.25}$$

With the constraints stated in this and the previous section, we have ensured that the routes are connected, and the last route terminates at the departure platform.

2.3.7.6 Constraints for splits and combines at the yard

In case a train splits, we introduce constraints to ensure that the newly spawned trains from the split, start at the end position of the composition. We denote the composition (the arrival unsplit train) - at and the resulting trains from the split as t_1 and t_2. Algorithm 5 outlines the constraints enforced when a train splits or not.

When the arrival train at requires a split, the number of routes for at is fixed to one. According to Assumption **A1** in 2.3.3, the split will be performed in the first yard reached by at, hence only one movement is allowed. If the train is not split, the first route is planned for the shunt train corresponding to this arrival train.

Last, again based on Assumption **A1** in 2.3.3, we have assumed that the arrival train is always split immediately upon reaching the yard. Therefore, as indicated in line 5, the start of the parking for t_1 and t_2 is when the movement for at ends ($e(A_1^{at})$).

We handle also the combine analogously in Algorithm 6. We denote with dt - the combined train, and with t_1 and t_2 - the shunt trains before combining. Again, the specified constraints are based on Assumption **A1** in 2.3.3.

Algorithm 5 Constraints for $at \in AT$, and $t_1, t_2 \in ST$ or SCT

1: **if** $at \rightarrow \{t_1, t_2\}$ **then** ▷ the arrival train will require splitting
2: $numr_{at} = 1$
3: $numr_{t_1}, numr_{t_2} \in \{0, 1, 2\}$
4: $e_0^{t_1} = e_0^{t_2} = e_1^{at}$
5: $s(stay_0^{t_1}) = s(stay_0^{t_2}) = e(A_1^{at})$
6: **else**
7: $numr_{at} = 0$
8: **end if**

Algorithm 6 Constraints for $dt \in DT$, and $t_1, t_2 \in CT$ or SCT

1: **if** $dt \rightarrow \{t_1, t_2\}$ **then** ▷ the arrival train will require splitting
2: $numr_{t_1}, numr_{t_2} \in \{0, 1, 2\}$
3: $numr_{dt} = 1$
4: $e_0^{dt} = e_{numr_{t_1}}^{t_1} = e_{numr_{t_2}}^{t_2}$
5: $s(A_1^{dt}) = e(stay_{numr_{t_1}}^{t_1}) = e(stay_{numr_{t_2}}^{t_2})$
6: **else**
7: $numr_{dt} = 0$
8: **end if**

2.3.7.7 Constraints for "stay" variables

The initial parking starts at the arrival time for all trains that have an arrival. A split shunt train does not have an arrival.

$$s(stay_0^t) = arr_t, \quad \forall t \in T \backslash \{DT, ST, SCT\} \tag{2.26}$$

The last parking ends at departure time for all trains that will depart.

$$e(stay_{maxM_t}^t) = dep_t, \quad \forall t \in T \backslash \{AT, CT, SCT\} \tag{2.27}$$

If the train performs fewer movements than the maximum allowed movements ($maxM_t$), the parking variables that correspond to movements that are not taken, should not be present. The number of parking variables is $maxM_t + 1$, after each movement and before the last one.

$$pres(stay_i^t) = 0, \quad \forall\ numr_t + 2 \leq i \leq maxM + 1 \tag{2.28}$$

Finally, the last present "stay" variable should end at the departure time for each train type that moves to the departure platform.

$$stay_{numr_t}^t = dep_t, \forall t \in T \backslash \{AT, CT, SCT\} \tag{2.29}$$

This constraint can be also enforced as follows:

$$pres(stay_i^t) == 0 \Rightarrow e(stay_i^t) - s(stay_i^t) == 0 \wedge d(stay_i^t) == 0 \tag{2.30}$$

2.3.8 The second version of the model

The second version of the model was developed to improve upon the performance of the first one. In the following lines, we outline only the modifications, in particular any removals, changes, or additions. Creating this version originated from the observation that the route variables could be removed, which would simplify the model.

2.3.8.1 Changes to variables

We remove all route interval variables $a_i^{t,r}$, the variables corresponding to the period when the train executes a route r when they are present. To substitute their function in the model, we utilize the group variables for each move.

2.3.8.2 Changes to constraints

The first change is in the constraints about $startOfSubm_k^{r,t}$ in Subsection 2.3.7.1. In particular, the constraint on line 2 at Algorithm 1 depends on $a_i^{t,r}$. The constraint is as follows:

$$startOfSubm_k^{r,t} = s(a_i^{t,r})$$

The new constraint is:

$$startOfSubm_k^{r,t} = s(A_i^t) \tag{2.31}$$

In the next Subsection 2.3.7.2, constraint 2.2, repeated below, is removed.

$$pres(a_i^{t,r}) = (chosenRoute_i^t == indexOf(r, R_i^t))$$

Constraint 2.3 from:

$$pres(a_i^{t,r}) = pres(occ_{s_j}^{t,r}), \forall s_j \in S^r$$

is changed as follows:

$$pres(occ_{s_j}^{t,r}) = (chosenRoute_i^t == indexOf(r, R_i^t)) \tag{2.32}$$

Constraint 2.6 for the duration of the route interval variable, is changed to:

$$d(A_i^t) = d_r^t, \text{ where } r = R_i^t[chosenRoute_i^t] \tag{2.33}$$

Next, we also remove the alternative constraint 2.9 which enforces that exactly one of the alternative routes, represented with the route variables $a_i^{t,r}$, for each move is present or none if the movement is not executed. In the revised model, each movement is represented by a single variable - A_i^t that is assigned to one route or not based on its presence. Therefore, the functionality provided by the alternative constraint is by design satisfied.

Finally, we substitute constraints 2.21 and 2.22, we repeat them below:

$$pres(a_i^{t,r}) \Rightarrow e_i^t = end(r)$$

$$pres(a_i^{t,r}) \Rightarrow e_{i-1}^t = start(r)$$

Before we present them, we first remind the reader that E_i^t where all possible end yards for routes for move $i \in \{1..maxM_t\}$ and E_0^t is just the initial yard if the train arrives at a yard or a dummy yard representing the arrival platform.

$$pres(A_i^t) \Leftrightarrow start(R_i^t[chosenRoute_i^t]) = E_i^t[e_i^t] \quad (2.34)$$

$$pres(A_i^t) \Leftrightarrow end(R_i^t[chosenRoute_i^t]) = E_{i+1}^t[e_{i+1}^t] \quad (2.35)$$

Every other constraint that was not mentioned does not include $a_i^{t,r}$ and remains the same.

2.3.9 Applied extension

Splits and combines with a through train

The distinction between the splits and combines discussed herein and those in the previous section 2.3.7.6 pertain to the types of trains included. In the previous section, an arrival train splits into two shunt trains (both plannable trains), or two shunt trains combine into a departure composition.

In this section, the split involves a through train, resulting in an arrival train and a smaller through train. Similarly, the combine entails a departure composition and a through train, resulting in a longer through train.

We first focus on the split. The scenario when a combine occurs is analogous. After a split on the platform, there is a shorter through train that must continue its scheduled route according to the timetable within a short time. Meanwhile, the second train, originally part of the through train, becomes an arrival train.

There is one through train and one plannable train on the platform. The first issue is whether the arrival train obstructs the through train to proceed. In case the arrival train is blocking the through train, it requires immediate routing, otherwise, routing can be postponed.

Based on the through-train decomposition (input information), we can determine the location of the arrival unit(s). Additionally, we know the direction of the next movement of the through train. Hence, we can deduce whether the arrival train is positioned ahead of the through train, meaning it is blocking the through train, or behind it. Depending on the situation, different additional constraints are applied:

Case 1: *If the arrival train is blocking the through train,* it requires an immediate routing. Hence, we account for the following matters:

- Set the end of the stay at the platform to be at most the departure time of the through train:

$$e(stay_0^t) \leq dep_{through} \quad (2.36)$$

- **No parking occupation -** As previously mentioned in pseudocodes 2 and 4 for defining section occupation constraints, each arriving train extends the occupation of the first section with the platform dwell time before routing to the yard. However, since through trains also occupy this section simultaneously, dual occupation renders the model infeasible. Therefore, for trains resulting from a split at the platform, we do not occupy the initial section.
- **No occupation of the first section of the taken route -** Additionally, we must avoid occupying the initial section of any route leading to a yard, as this section corresponds to the arrival platform and, as previously stated, is already occupied by the through train.
- **Move only from the free side -** Only one side of the platform is free, the other is occupied by the through train (Figure 2.11). Since the arrival train will be routed before the departure of the through train, it is required to exit from that free side. We can ensure this by filtering the routes to consider only those that exit from the correct side. Alternatively, we can impose the following constraint:

$$originSide(r) == freeSide, \quad \text{where } r = R_1^t[chosenRoute_t] \quad (2.37)$$

  The first approach is better as it does not introduce additional constraints.

Case 2: *In case the arrival train is at the back and does not obstruct the through train.* Here again, the following points need to be taken into account:

- First, there are two possibilities for departing:
  - Either the platform is a LIFO track and hence, the shunt train is obstructed by the through train and requires to wait (Figure 2.10).

$$e(stay_0^t) \geq dep_{through} \quad (2.38)$$

  - Or the platform is a free track, allowing the shunt train to move at any time before the next scheduled arrival. Again, the shunt train can only move from the free side of the platform before the through train departs. Once the through train has departed, the shunt train can exit from any side of the platform. To simplify operations, we have decided to always route the arrival train after the through train, thereby mitigating the

issue with LIFO platforms. We enforce again:

$$e(stay_0^t) \quad \geq \quad dep_{through} \quad (2.39)$$

- Last, we should again avoid occupying parking sections for both the through train and the arrival train simultaneously. Since we have decided that the arrival train will depart after the through train, the section corresponding to the arrival platform will be occupied by only by the arrival train, as the arrival train will remain on the platform longer. Additionally, the first section of the route is also occupied.

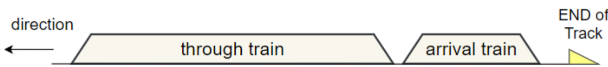


Figure 2.10: A split at the platform that results in a through train and a plannable train. The plannable train requires to wait until the through train has departed as the platform is LIFO.

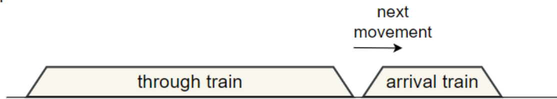


Figure 2.11: A split at the platform that results in a through train and an arrival train. The arrival train must be routed promptly, as it obstructs the next scheduled movement of the through train.

2.3.10 Output

Figure 2.12 illustrates the output that can be obtained from the model. Initially, we provide the important input information for the train - the arrival and departure platforms, and arrival/departure times if available for the type of the train. Next, for each parking and each movement, we indicate whether it occurs, along with the time interval and duration. For the movement, the chosen route is also printed.

```
BaseShuntTrain: (940227), Type: VIRM4 MaxM: 3, numM: 2,
Arrival Platform: Ekz_2 (272 m A), Arrival Time: 1.20:53:00,
Departure Platform: Ekz_2 (272 m A), Departure Time: 1.22:09:00

Stay 0: present, Time interval: (1.20:53:00 - 1.20:53:05)  Duration: 0.00:00:05
Move 1: present, Time interval: (1.20:53:05 - 1.20:54:19), Route: [1/1] Allow : Ekz_2 -> Ekz_405  Duration: 0.00:01:14

Stay 1: present, Time interval: (1.20:54:19 - 1.22:06:24)  Duration: 0.01:12:05
Move 2: present, Time interval: (1.22:06:24 - 1.22:07:38), Route: [1/1] Allow : Ekz_405 -> Ekz_2  Duration: 0.00:01:14

Stay 2: present, Time interval: (1.22:07:38 - 1.22:09:00)  Duration: 0.00:01:22
Move 3: NOT present, Time interval: (1.22:09:00 - 1.22:09:00) Duration: 0.00:00:00
Stay 3: NOT present, Time interval: (1.22:09:00 - 1.22:09:00) Duration: 0.00:00:00
```

Figure 2.12: An example of the output of the model for one of the base shunt trains in the Enkhuizen instance.

Chapter 3

Instances

3.1 Instances

Two real-world instances were provided by NS - one small instance from the Enkhuizen station and one medium instance from Amersfoort station. To extensively test and evaluate the created model, additional instances, based on those, were created. In this section, we first give a brief overview of these two real instances (Section 3.1.1). Next, the used instance creation strategies are discussed (Section 3.1.2). Finally, we conclude with a description of the performed data preprocessing (Section 3.1.3).

3.1.1 Real instances

Enkhuizen instance

The first instance is located at Enkhuizen station, a small station in the northern region of the Netherlands. Figure 3.1 illustrates the station layout, and presents the two platforms: platform 401 and platform 402, positioned on the left side of the figure. Additional details are provided in Table 3.1.

There are 12 shunt (plannable) trains that arrive and require routing to any of the three yards. Additionally, 160 predetermined movements must be considered and satisfied by the model. The number of routes available per movement ranges from 3-4 up to approximately 9-10 when transfers between yards are considered. Only the shortest routes between all possible endpoints are included.

Lastly, this small instance involves no splits or combines at the yards, in particular, only base shunt trains are present.

Amersfoort instance

The next real-world instance is located at Amersfoort. Amersfoort has a larger and busier station with more through trains which leads to more fixed movements. Table 3.1 provides the specific details.

On one hand, having a single yard simplifies the instance, as the search space terms of the number of possible routes, and the number of possible endpoints is reduced compared to Enkhuizen. However, on the other hand, since all trains are shunted to the same yard, they frequently require routing over overlapping routes.

The instance can easily become infeasible, in case of insufficient free time to move a train between the platforms and the yard. This issue may arise due to parts of the infrastructure that are heavily utilized by both through trains and plannable trains.

3.1.2 Instance creation

This section discusses possible methods to generate more valid instances based on the two real-world examples. Due to specific checks before running the model, it is rather difficult to introduce new trains to the existing instances. Not to mention generating entirely new instances. The next subsections outline a set of changes that could be utilized to create new instances.

3.1.2.1 Modifying the plannable (shunt) trains

These instances are created by selecting a subset of the shunt trains from the real-life scenario. All other aspects of the real instance, such as through train movements, yards, and routes, remain unchanged.

There are many options to generate instances based on altering the set of shunt trains. We have opted to select for each possible number of trains, a fixed number from all combinations with that number of trains. The number of trains ranges from 1 to up to the total number of trains in the real-world instance. In essence, we generate a fixed number of instances from all combinations with one train, two trains, three, and so forth.

3.1.2.2 Remove/Add a yard

Additionally, one could explore altering the station layout. For instance, we could establish a new yard by defining a set of tracks that allow for parking, to compose a new yard.

Alternatively, a yard could be removed, allowing trains to only reverse or briefly stop on those tracks. For instance, in the Enkhuizen, one might eliminate the yard with track 403, allowing trains to perform only reversals there. This could either simplify the problem as fewer routes are considered or could result in harder instances when the parking space at the yard is considered. As the yard extension C.1.2 is not included, the instance should become easier.

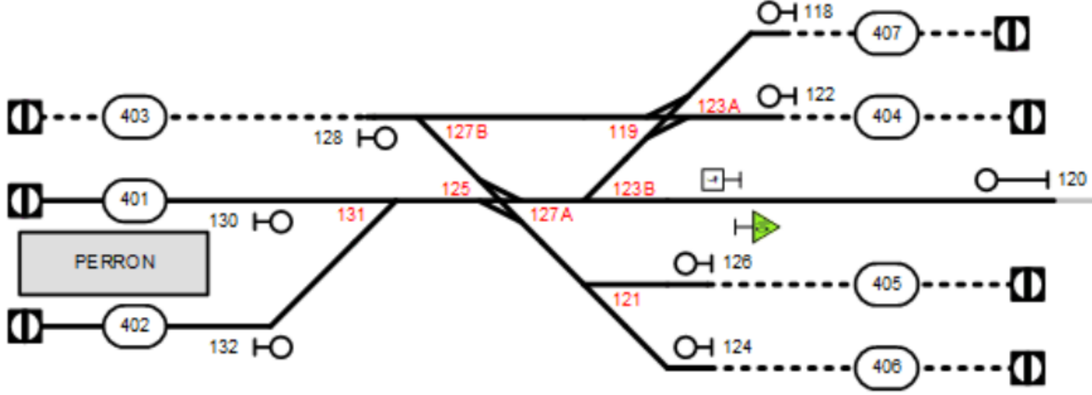


Figure 3.1: The layout of the station in Enkhuizen instances.(sporenplan.nl)

Instance	#Shunt trains	#Fixed movements	#Routes/movement	Yards
Enkhuizen	12	160	3/4 - 9/10	405_406, 403, 404_407
Amersfoort	23(arrive)	1600	around 2	Bokkeduinen

Table 3.1: The table outlines the specific characteristics of the two real-life instances. For each instance, the following information is provided: the number of arrival plannable trains, the number of fixed movements, required to be additionally included, the number of available routes per movement, and the yards.

3.1.2.3 Remove/Add (compound) routes

A further modification could be to remove and add (compound) routes. For instance, removing compound routes, those consisting of more than one subroute, can reduce the search space and potentially shorten computation time. These routes are often less desirable. In addition, not considering longer routes would also result in better solutions.

Additionally, introducing alternatives to the shortest routes could result in harder instances.

3.1.2.4 Reserving certain important/busy sections/tracks for long duration

Lastly, one could test the model in scenarios where specific key sections at junctions are unavailable due to maintenance or accidents. In such cases, certain trains would be forced to take longer routes. Therefore, it is crucial to ensure that alternative routes are available and considered by the model.

3.1.3 Data preprocessing

The developed preprocessing ensures that the input data for the section occupations for the predetermined routes is feasible. Specifically, it guarantees there are no simultaneous section occupations by different trains for the same section. This situation could occur if two trains are parked over the same section, which is permissible in practice.

To conform to the occupations for the predetermined routes while also making the section occupations feasible for the solver, all overlapping intervals for each section were combined into one. This not only ensures feasibility but also reduces the number of occupation intervals, thereby lessening the workload of the noOverlap constraint.

In the Amersfoort instance, five plannable trains were excluded due to their arrival or departure coinciding with the presence of another through train on the platform. This is not allowed, according to Assumption **A6** (in Section 2.3.3). After the removal, the plannable trains that arrived are 23. The total number of trains, including all types, was 40, as some of these trains split and combined to form new trains.

3.2 Validations

This section presents the validations performed on each obtained solution. These validations aim to confirm that the solutions satisfy indeed the requirements outlined in Section 2.3.2. Specifically, we strive to ensure that the implemented constraints accurately correspond to the desired behavior. The validations are similar to the defined constraints but are now applied to the obtained solution.

In the following text, we first briefly state each requirement and then describe the corresponding validation check. We begin with the validation checks for the second version of the model (model-v2) and subsequently extend with those necessary for the first version (model-v1).

3.2.1 Validations for model-v2

R1: The model should produce a feasible schedule.

This requirement was further divided into three necessary conditions for it to hold. We validate each of

them separately.

- **No train is moving over infrastructure already reserved by another train.**

  After obtaining a solution from the model, we know all present occupation intervals for each section. These occupation intervals correspond to the reservations of routes that include this section.

  To verify that no two trains have reserved the same section simultaneously, we require that none of the intervals overlap with each other. A naive approach would check that no interval overlaps with any other interval, running in $O(n^2)$, where $n = \#$ present occupation intervals for section s. However, we opt for a more efficient version.

  Our approach runs in $O(n * log(n))$. Initially, we sort all the start and end times of all present intervals for a specific section. Afterward, we check whether we can encounter the start of any interval before the end of another one. In case such a situation occurs, this indicates overlapping reservations for the same section. Figure 3.2 illustrates this situation.

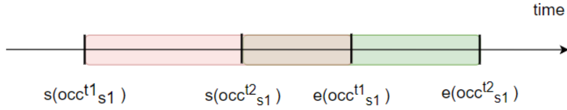


Figure 3.2: On the timeline, the start and end times of two present occupation intervals for the same section, but for different trains, are ordered. As we process each boundary (start or end) of each interval, we will encounter the start of the second interval for train t_2 before the end of the previous interval. This can only happen if they overlap.

- **All trains are planned at every point in time.**

  This requirement is verified by ensuring that for each train, the parking times and the movements in between are continuous, meaning there is no gap where the train's place is unclear. For instance, we validate that the first parking period ends exactly when the first movement starts, and then the first movement ends precisely when the second parking interval begins, and so on.

  Additionally, we verify that the first parking period begins when the train arrives (for arriving, base shunt train, and combining trains) or when the train spawns (in the case of a split or departure composition). Similarly, the last parking period should end when the train either departs from the platform or is combined with another train, ceasing to exist.

- **Trains do not move from one place to another, without reserving a route in between.**

  For this, we verify that the start of each taken route corresponds to the assigned position (e_i^t) of the train prior to this move. Similarly, the end of the taken route corresponds to the next assigned position (e_{i+1}^t).

  We also confirm that the initial and final positions of trains are the arrival and departure platforms for trains that arrive or depart. For the rest, we ensure that the initial position, where the train is spawned, matches the last position of the train from which it was created. For instance, the initial position of a split train should match the last position assigned to the arrival train containing the split train.

  If fewer than the maximum number of movements are taken by the train, we check whether the end positions are assigned to the dummy position.

R2: The routes of all through trains which are predetermined and fixed, are respected by the model.

This requirement is fulfilled by considering the section occupations for the routes of the through trains.

R3: Train units that are combined at the yard, traverse simultaneously the same route from the yard to the platform. Analogously, for the trains before splitting.

This requirement is satisfied by the modeling approach, in particular, by the way we define and create the different types of trains.

R4: The model uses the section occupations specified by HIP.

The start and end times of each section ($end_{s_i,r}^t$, $start_{s_i,r}^t$), relative to the beginning of the route, are provided by HIP.

R5: The model routes each train to at most two yards.

The restriction on the maximum number of movements ($maxM_t$) ensures that the solutions adhere to these requirements as well.

3.2.2 Additional checks for model-v1

We further validate that if and only if A_i^t is present, then exactly one route variable per move is present, otherwise, none of the route variables should be present. The equations 3.1 and 3.2 illustrate the check.

$$pres(A_i^t) \quad \Leftrightarrow \quad \sum_{r \in R_i^t} pres(a_i^{t,r}) \quad == \quad 1 \tag{3.1}$$

$$pres(A_i^t) \; == \; 0 \; \Leftrightarrow \; \sum_{r \in R_i^t} pres(a_i^{t,r}) \; == \; 0 \tag{3.2}$$

Furthermore, we also verify that the start and end times of the route variable coincide with the start and

end times of the corresponding group variable. This ensures that the route occurs exactly when the movement is planned. Equation 3.3 presents this validation.

$$s(a_i^{t,r}) \quad == \quad s(A_i^t) \wedge e(a_i^{t,r}) \quad == \quad e(A_i^t) \tag{3.3}$$

Last, we ensure that the present route variable ($a_i^{t,r}$) corresponds to the route at position $chosenRoute_i^t$ within the set of available routes for this move - R_i^t.

$$pres(a_i^{t,r}) \quad \Leftrightarrow \quad R_i^t[chosenRoute_i^t] \quad == \quad r \tag{3.4}$$

Chapter 4

Experiments & Results

4.1 Goal of the experiments

The initial evaluation of the entire instance of Enkhuizen requires from 4-5 to 10-15 minutes. While this is a reasonable time, for larger instances the running time may considerably worsen. Therefore, the primary goal of the experiments concentrated on improving the performance of the model. Several experiments were designed to achieve this by further constraining the search space, guiding the solver's search, and evaluating and comparing the second version of the model (Section 2.3.8).

Next, a second set of experiments was designed to identify how certain characteristics of the instances affect the model performance. In particular, a set of experiments with changing the number and type of routes and removing a yard were conducted.

The final set of experiments was conducted on the second version of the model, implemented also in MiniZinc. The goal was to investigate and compare the performance of other solvers. The designed experiments also address all additional research questions outlined in Section 1.7.

4.2 Experiments' overview

Constraining the parking intervals

The initial tests reveal promising results when we constrain the durations of the variables corresponding to the parking time at arrival and departure platforms to a maximum of 90 seconds. Therefore, we further design experiments in this direction. However, limiting the parking duration to be within 90 seconds could easily result in an infeasible solution and does not align with a realistic platform dwell time. To increase the likelihood of feasible solutions, we allowed the parking durations to be up to 15 minutes. This upper bound is considered more appropriate, as solutions, where trains are planned to be parked on the arrival or departure platform for longer than 15 minutes (900 seconds), are also not desirable.

Using the notation introduced in Section 2.3, the constraints are as follows:

$$\textbf{if}\ \ init_{at} \notin Y\ \ \textbf{then:}\quad d(stay_0^t) <= 900, \quad \forall t \in T \backslash \{ST, SCT, DT\} \tag{4.1}$$

$$\textbf{if}\ \ des_{at} \notin Y\ \ \textbf{then:}\quad d(stay_{numr_t}^t) <= 900, \quad \forall t \in T \backslash \{CT, SCT, AT\} \tag{4.2}$$

Variable and value selection strategies

Variables and value selection strategies are beneficial as they can guide the search process of the solver, potentially leading to a feasible solution more efficiently.

Variable selection strategy "instructs" the solver which variables to initially assign during the search. For instance, variables with small domains are usually better to fix at the beginning of the search compared to those with large domains. This is because the solver will require this initial stage of the search at most as many times as there are values in the domain. Hence, the larger the domain, the higher the number of times the solver will potentially revisit this stage of the search. For example, a good strategy is to assign initially the variables for the number of movements for each train as those have at most four different values (0 to 3).

Once a variable is selected, the value selection strategy suggests which values the solver should attempt initially. In particular, a value selection strategy can guide the solver to aim for the most promising values. These could be values that based on some domain insights, are expected to more likely result in a feasible or better solution.

For example, solutions selecting fewer movements for trains are generally better solutions. These solutions are also easier to find, as fewer movements are expected to result in fewer route conflicts that the solver must resolve.

We conducted experiments with the variable and value strategies for the number of movements. We further also design experiments to fix the variables for the arrival platform to the smallest values first. There, we aimed for a similar effect as constraining the parking time to 15 minutes, which was previously noted as

beneficial. Analogously, we developed a search strategy for the departure platform variables to minimize the durations as well.

Seed influence

To ensure that the seed used by the solver for generating reproducible solutions did not impact performance, we ran all experiments with 5 different seeds. This approach allows us to also assess the influence of the seed on the solver's performance.

Influence of instance characteristics

We investigated the change in the performance when altering specific features of the instances, such as:

- increasing the number of routes for Amersfoort instances.
- removing a yard for Enkhuizen instances.
- removing compound routes for Enkhuizen instances.

Compare the first and second versions of the model

We compare the performance of both versions on the default instances and the instances with additional routes for Amersfoort.

Evaluate different solvers

Finally, we investigated the performance of other solvers on the second version of the model, implemented in the MiniZinc, and using Enkhuizen instances.

Before further diving into the concrete experiments and the obtained results, we present the supposed results in the form of hypotheses.

4.3 Hypotheses

General Guidelines: If we have simply mentioned model-vX or (default/base) model-vX, then we refer to version X of the model without any additional constraints or search strategies. X can be 1 or 2. In case we consider a model with search strategies or additional constraints enabled, we will explicitly mention that.

4.3.1 Hypotheses for changes in the model

Between model versions

H1: Model-v2 will have better performance than model-v1.

Reason: The second version improves the first by removing the interval variables for the routes and the extra constraints for them. This simplification of the model is expected to enhance search efficiency and propagation, thereby improving overall performance.

Between the default model-v1 and the model-v1 with variable and value strategies

H2: The models with variables and values strategies for the duration on the arrival/departure platform and the number of movements will improve upon the default model-v1.

Reason: The search strategies aim to prevent the solver from branching on variables that cause slow propagation and extensive search for feasible solutions. However, why do we envision that these variables are crucial for finding a solution faster?

A lower number of movements leads to fewer section occupations, reducing the conflicts in the noOverlap constraint that the solver must resolve.

Platforms are heavily utilized, especially in busy stations. Therefore, guiding the solver to prioritize lower values for platform durations increases the likelihood of finding a feasible solution, compared to allowing the solver to assign any value within the domain. Note that the duration domain is defined as the departure time minus the arrival time. This can be substantial, especially when trains arrive and depart on different days.

Between the default model-v1 and model-v1 with additional parking constraints

H3: We envision that model-v1 with the additional parking constraints will perform better.

Reason: Firstly, the additional constraints will reduce the search space. Secondly, these constraints will also facilitate better propagation and tightening of the domains of other variables, such as the start of the next movement and the next parking variable.

4.3.2 Hypotheses for the influence of the instance characteristics

With more allowed routes for Amersfoort

H4: We expect the performance of the default model-v1 to considerably degrade when additional routes are enabled.

Reason: The additional routes expand the search space. More routes increase route interval variables and section occupation variables, and consecutively number of constraints as well.

H5: We expect the performance of model-v2 with additional routes to worsen compared to running without them.

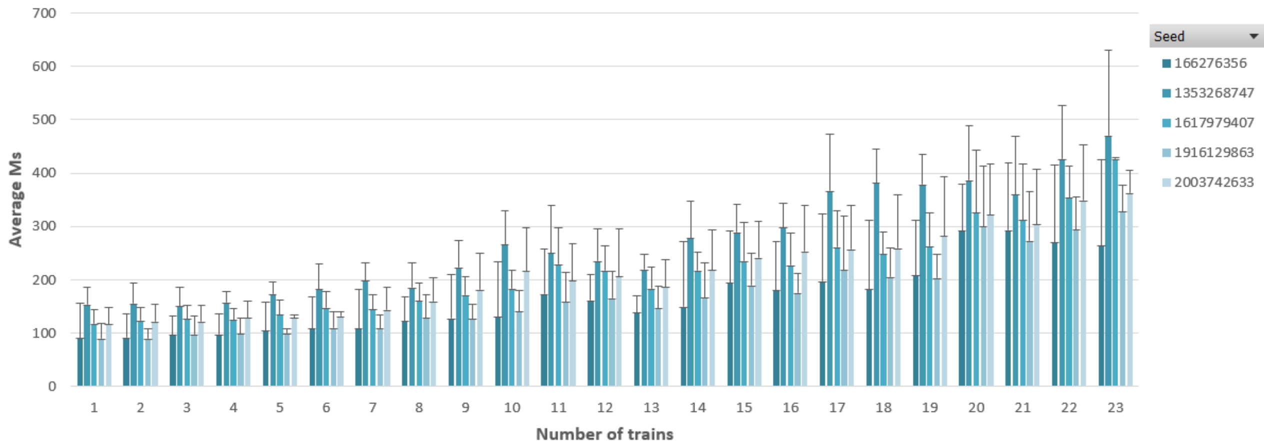


Figure 4.1: Comparison between the different seeds for the Amersfoort instances. The values are obtained by averaging the running time in **milliseconds** of instances with the same number of trains. Seed 1353268747 exhibits consistently the highest average computational time. Moreover, performance degrades twice between seed 166276356 and 1353268747 for instances with 17 and 18 trains.

Reason: The reasoning is analogous, without, in this case, introducing additional route variables for each new route. This leads to the next hypothesis.

H6: Model-v2 will outperform model-v1 when additional routes are enabled for both.

Reason: The reasoning is analogous to the explanation of why model-v2 is expected to perform better than model-v1 (**H1**, Section 4.3.1).

Upon removal of the yard, or the routes with reversals

H7: We hypothesize that the performance of model-v1 will be comparable when a yard or routes with reversals are removed.

Reason: In both cases the number of the removed routes is approximately the same, equal to 1/3. Therefore, we presume similar performance.

4.3.3 Hypotheses for the MiniZinc model

H8: We envision that model-v2, implemented in MiniZinc, will exhibit slightly worse performance when solved with Google OR-Tools, compared to the model, implemented with the Google OR-Tools API.

Reason: For the same instance the model in MiniZinc creates more variables and requires more constraints than the one implemented in Google OR-Tools.

4.4 Experimental Setup

Hardware

Both versions of the model are implemented in C# using Google OR-Tools Constraint Catalog. CP-SAT solver was used, as this one is faster and recommended by Google OR-Tools themselves.

The mentioned experiments were performed on the DelftBlue cluster. For the MiniZinc experiments a single thread, 1 CPU, and 16GB of RAM were utilized. For all others, a single thread, 1 CPU, and 4GB of RAM were employed.

MiniZinc version 2.8.3 1160085037 was used. Google OR-Tools 9.3.10497 was used for most experiments, except for the variable and value strategies, where version 9.10.4067 was utilized.

Benchmarks

We generate instances for all experiments by modifying only the set of plannable trains from the real-life instance 3.1.2.1. Recall, that this strategy selects a fixed number of instances from all combinations with one train, two trains, three, and so forth. The reason for choosing this strategy is that the number of trains is typically a key parameter influencing performance. By using this approach, we can systematically examine the performance as the difficulty of the instances increases. For some experiments, additional features such as the yards or the number of routes are altered as well.

The fixed number is 10 and each instance is executed 3 times, this produces 30 instances with the same seed and the same number of trains. However, as there is only one combination with all trains, namely the entire instance, ***we execute it 5 times.*** The number of instances based on Enkhuizen for one seed is 335 (10 ·

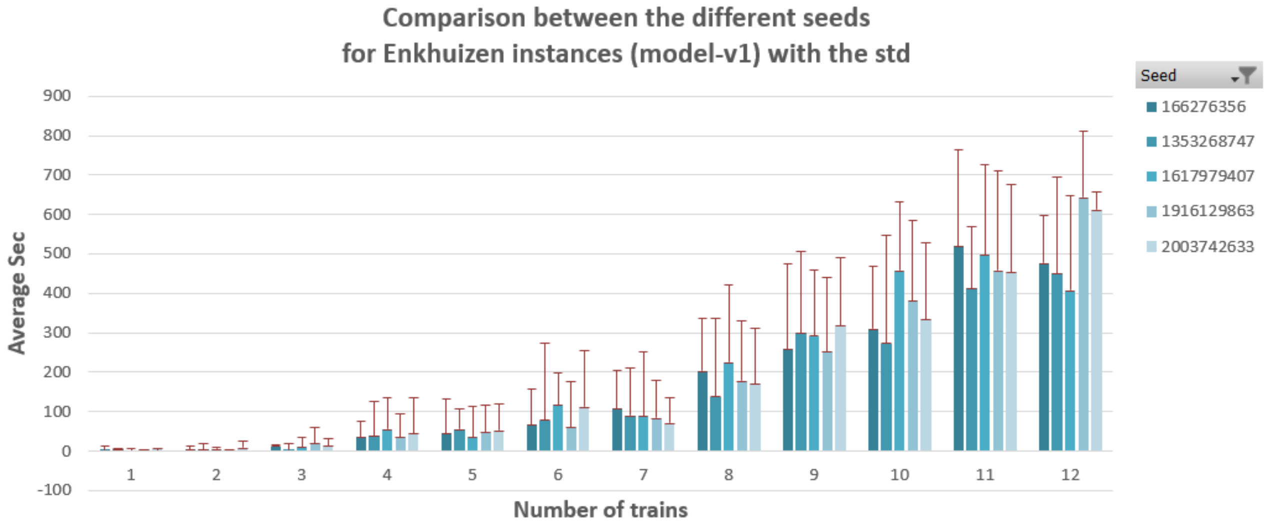


Figure 4.2: Comparison between the different seeds for Enkhuizen instances. The values per number of trains are produced by averaging the running time in **seconds** for all instances with the same number of trains. Using seed 1916129863 instead of 1617979407 results on average in a few minutes higher computational time.

$3 \cdot 11$ (the $\#trains$ - 1 (the entire instance)) + 5 (# of runs or the whole instance)), while for Amersfoort - 665 instances. Executing all these instances for all experiments with 5 seeds resulted in more than 30,000 instances. For each instance, an upper bound of 15 minutes was imposed. Furthermore, upon timeout, we assign 15 minutes as the running time of the time-outed instance.

In all experiments, we seek a feasible solution, and no objectives are applied.

Pre-experiment - Seed influence

Figures 4.2 and 4.1 depict the influence of seed by comparing the average performance of instances with the same number of trains for model-v1 on Enkhuizen and Amersfoort, respectively. The lines above the bars show the first standard deviation (34% of the data).

In Amersfoort, the variations in running times across different seeds are consistent among instances with different numbers of trains (Figure 4.1). For the instances with 17 and 18 trains, the average computation time can double when altering the seed. However, as the measurement unit is in milliseconds (ms), minor variations in the performance are not considered significant. Nevertheless, based on the repeated differences in the running time, we conclude that for Amersfoort, the computation time depends on the seed.

The performance of the distinct seeds for Enkhuizen exhibit varying patterns. For some instances of a certain number of trains, certain seeds result in better performance, while for others, different seeds. Nevertheless, there could be on average a few minutes difference (Figure 4.2).

In general, it is difficult to determine whether a seed is consistently better or worse. The seed that performs best on Enkhuizen is worse for Amersfoort. However, the seed does influence the results, sometimes considerably. Hence, the seed is an important instance-dependent parameter that should be fine-tuned further.

To mitigate this variability and produce more accurate and robust results in the next experiments, we will aggregate the results from five different randomly selected seeds.

4.4.1 Types of experiments

We categorize the experiments into three main groups: experiments about evaluating changes in the model, assessing how modifications in the instances impact the model's performance, and experiments with MiniZinc model. Table 4.1 presents an overview of all.

4.4.2 Solutions' quality

After each experiment, we analyze the solutions focusing on three key quality characteristics. We examine the duration on the arrival platform, the duration on the departure platform, and the number of movements (specifically for Enkhuizen instances). These analyses are conducted only on the entire instances. For Amersfoort, the number of movements is not shown in a plot as the values are consistent among experiments.

The intuition behind these metrics is that a prolonged duration on the arrival or departure platform is generally not considered optimal, as it results in wasted time for the drivers. An increased number of scheduled movements requires more drivers and leads to higher energy consumption. Furthermore, this further intensifies the traffic at the station and reduces schedule robustness. Besides, these metrics are also deemed satisfactory by the planners (the people who currently manually develop the shunt plans).

Type of Experiment	Experiment	Model version	Solver version	Solver search	Special Parameters
Changing the model	Adding parking constraints	v1	9.3	default	-
	Variable and Value Strategies	v1	9.10	fixed	keep all feasible solutions in presolve
	Compare versions	v1 and v2	9.3	default	-
Changing instance characteristics	Adding additional routes	v1 and v2	9.3	default	-
	Removing a yard, or Reversals	v1	9.3	default	-
MiniZinc	Compare different solvers	v2	Multiple solvers (noted later)	default configurations	-

Table 4.1: An overview of the experiments and parameters utilized. The default search method is also referred to as automatic.

Guidelines for reading the solutions' quality plots

Before discussing the results, we outline general guidelines for interpreting the figures on solution quality.

The x-axis allows for a comparison of values across different seeds, except in one experiment where results for a single seed are presented for clearer understanding. On the y-axis, the duration (in minutes) that each train spends on the arrival or departure platform is plotted. The green triangle indicates the mean value. The box represents the interquartile range, with the bottom and top edges of the box marking the 25th and 75th percentiles, respectively. The box encloses the middle 50 percent of the data.

Recall, the entire instance is executed five times per seed. Each boxplot displays the duration on the arrival/departure platform for all trains in these 5 instances. A single circle represents the duration of one train arriving or departing from a platform. The trains that arrive or depart from a yard are excluded, as opposite to the platform, there, trains are preferred to remain longer. Last, the table, for instance, table 4.4c, presents the number of trains assigned one, two, or three movements for one execution of the entire instance. For Amersfoort, the trains that do not arrive or depart are excluded from the respective figure. For instance, a split shunt train is not included in the arrival time plot but is part of the departure plot.

Finally, we note the observed slight variations in solutions from different executions of model-v1 with the same seed and instance when utilizing version 9.3 of Google OR-Tools.

4.5 Experiments with model changes

General Guidelines: By default instances, we refer to those where only the number of trains is altered. Other characteristics such as the number of routes or yards are as in the original real-life instance. Any deviations from this will be explicitly noted.

4.5.1 Adding parking constraints

To answer the hypothesis **H1** in 4.3.1 on the usefulness of the additional parking constraints, we will compare the running time of the first version of the model with and without these constraints.

Figure 4.3 below summarizes the results for Enkhuizen. Instances with constrained parking variables are solved within milliseconds and are not included in the plot. However, the default model reaches an average of 518 seconds (around 8-9 min).

We conclude that constraining these types of variables significantly improves solution times for instances similar to Enkhuizen. Further analysis of these results is provided in Section 4.6.4.2. In particular, the analysis demonstrates that only the variables controlling the durations on the departure platform lead to improvement by enabling faster convergence for Simplex, the core algorithm for linear programming (LP) ([sim, 2020]).

This experiment is conducted only for Enkhuizen. In Amersfoort, some trains are required to stay longer than 15 minutes on the platform to achieve a feasible solution.

4.5.1.1 Solutions' quality

Figure 4.4 illustrates the three quality metrics for the real-life instances of Enkhuizen with and without additional constraints.

Figure 4.4 presents that the solver has selected the duration on the arrival platform to the minimum possible value for both the model with and without additional constraints. In contrast, the duration on the departure platform is not minimized. This outcome can be attributed to the default search strategy employed by the solver, which selects the first unassigned variable and assigns it its minimum value. As a result, the

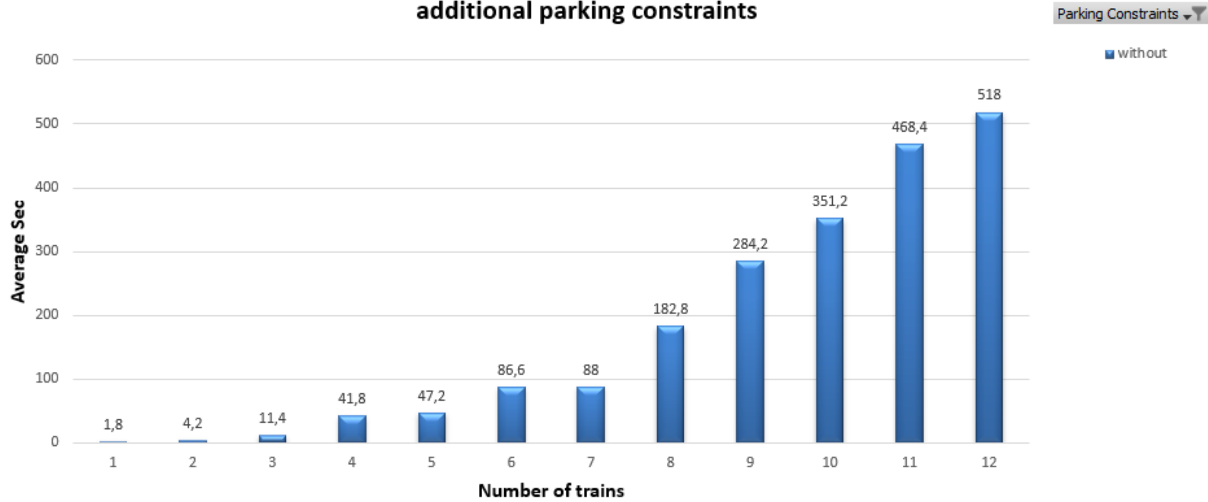


Figure 4.3: The figure displays the average performance of Enkhuizen instances on the model without the parking constraints. The obtained values on top of the blue bars are averages of the running times of the different instances with the same number of trains and different seeds. The same instances, executed on the model with the constraints, are solved within milliseconds.

arrival durations remain short regardless of any added constraints, For the departure, the solver again assigns a low value for the start of the parking, resulting in a longer duration on the departure platform.

The number of movements on average is between 3 and 2, where 3 is the most common choice for both versions (Table 4.4c).

4.5.2 Variable and value strategies

The experiments in this section address hypothesis **H2** in 4.3.1. We investigate variable and value selection strategies for the following variables for each train:

- *For the end of the parking on the arrival platform*
- *For the start of the parking on the departing platform*
- *For the number of movements*

Controlling the start and the end of the parking corresponds to restraining the durations on the arrival and departure platforms, respectively.

In all aforementioned cases, the value selection strategy is chosen based on the reasoning from hypothesis **H2** in 4.3.1. as it is anticipated to perform the best. These strategies are expected to produce better solutions, as discussed in Section 4.4.2 about the solution quality metrics. For the variable selection strategy, we explore each of the available options in Google OR-Tools.

Strategies for the platforms durations

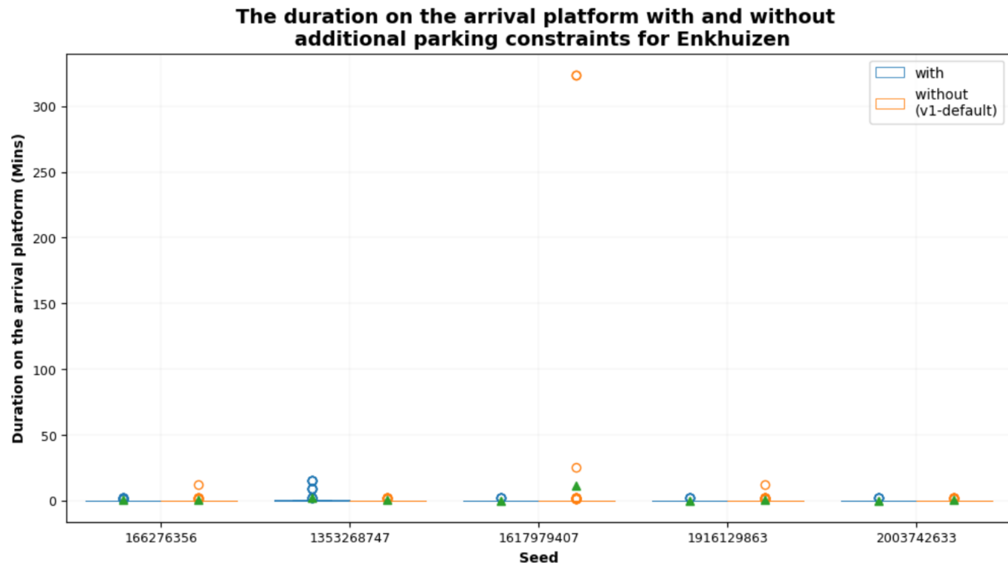
For the variables for the end of the parking at the arrival platform the *SelectMinValue* and *SelectLowerHalf* strategies were employed. *SelectMinValue* attempts to find a solution first with the lowest value in the domain, while *SelectLowerHalf* selects a value from the lower half of the domain. For the start of the parking at the departure platform, the *SelectMaxValue* and *SelectUpperHalf* strategies were utilized. These work analogously to the *SelectMinValue* and *SelectLowerHalf.*

Figures 4.5 and 4.6 present the average results across all seeds for each combination of variable and value selection strategies for the variables controlling the parking on the arrival platform. The corresponding results for the departure platform are shown in Figures 4.7 and 4.8 for the Enkhuizen and Amersfoort instances, respectively. The bars labeled with "None-None" correspond to the default model-v1 when no variable and value strategy is applied.

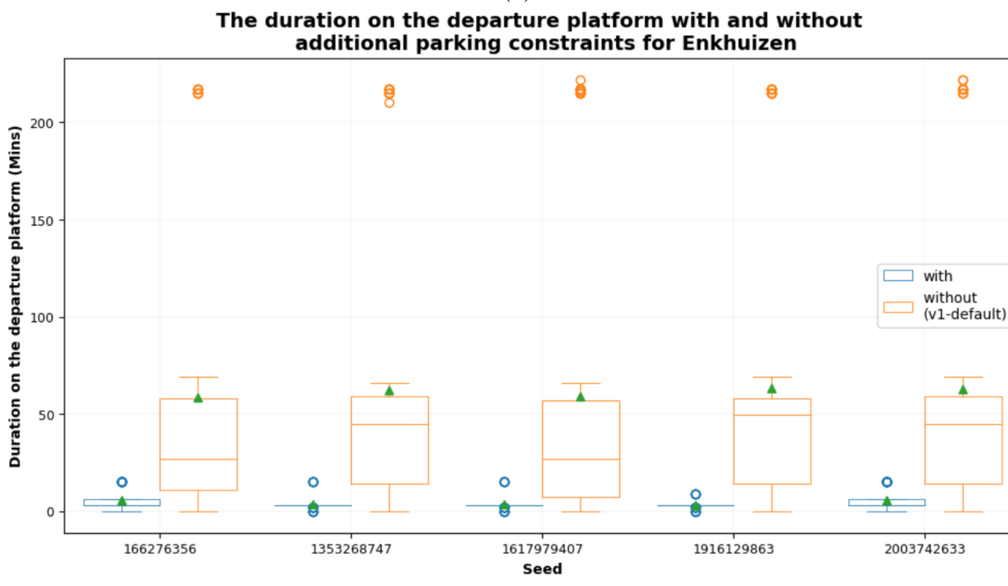
First, note the values on the y-axis: for Amersfoort, in milliseconds as before, while for Enkhuizen, in seconds, but now approximately 20 times lower (from an average of 518 to 30 seconds) than in the experiment involving additional constraints (Figure 4.3). For Amersfoort, the improvement is approximately two times, reducing the running time from around 400 ms in the pre-experiment to less than 200 ms. The difference is partially due to the new version of Google OR-Tools (9.10) but to a larger extent due to the different solver search strategy — fixed search, observed in several local executions.

Specifying a search strategy for the variables on the arrival platform does not improve the performance of either Enkhuizen or Amersfoort. In fact, for the larger instances, the strategies worsen the solutions. This outcome is consistent with the findings from the previous experiment involving additional parking constraints.

In contrast, Figure 4.7 demonstrates that instructing the solver to prioritize the variables on the departure



(a)



(b)

	Seed	Experiment	Count_of_1	Count_of_2	Count_of_3
0	166276356	v1	0	2	10
1	166276356	with constraints	1	2	9
2	1353268747	v1	0	4	8
3	1353268747	with constraints	1	2	9
4	1617979407	v1	0	1	11
5	1617979407	with constraints	0	2	10
6	1916129863	v1	0	2	10
7	1916129863	with constraints	0	2	10
8	2003742633	v1	0	4	8
9	2003742633	with constraints	0	4	8

(c)

Figure 4.4: The three quality aspects for both the model with and without the additional parking constraints. The results are from the entire instance of Enkhuizen and solutions are categorized based on different seeds.

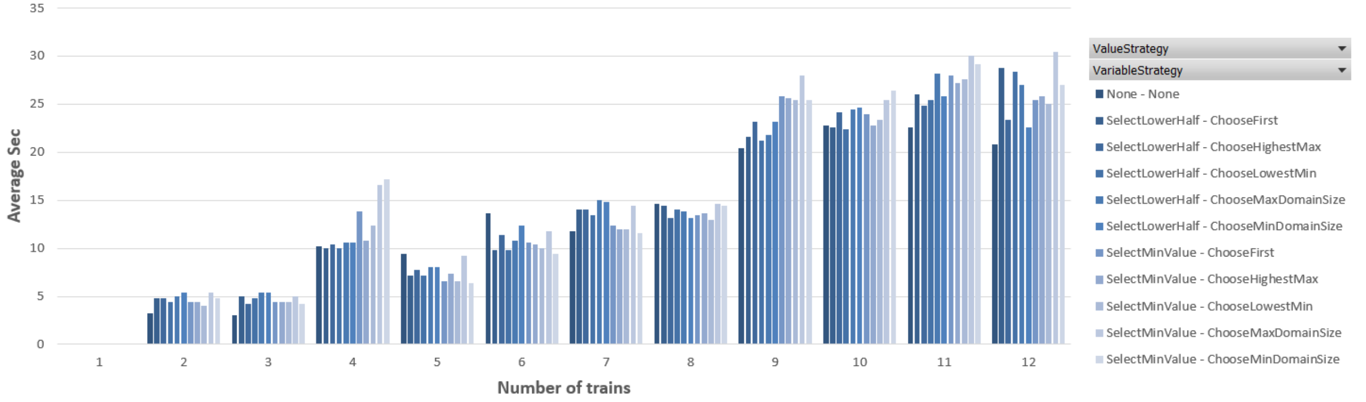


Figure 4.5: Comparison between the variable and value selection strategies for the variables denoting the parking time on the **arrival** platform on **Enkhuizen** instances and model-v1. The bars labeled "None-None" in the legend represent the performance of the default model-v1.

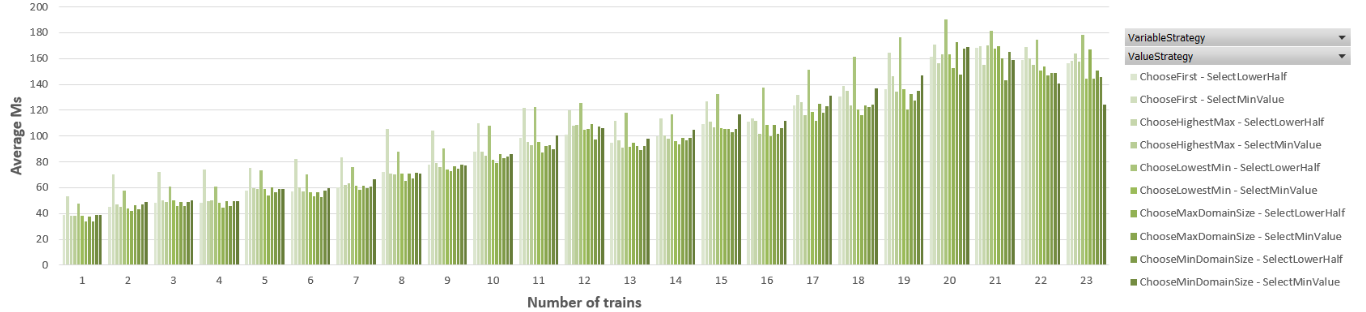


Figure 4.6: Comparison between the variable and value selection strategies for the variables denoting the parking time in **milliseconds** on the **arrival** platform for **Amersfoort** instances for model-v1. Again, "None-None" corresponds to the default model-v1.

platforms is beneficial for Enkhuizen. The plot shows that using the value strategy *SelectMaxValue* improves the performance by at least half. However, *SelectUpperHalf* yields even better results, solving the entire instance within 3-4 seconds.

This result, however, does not follow for Amersfoort (Figure 4.8). We identify two potential reasons for this result, perhaps both attributing to some extent. First, as the computational time is already low (in milliseconds), changing the default search strategy of the solver to custom could perhaps introduce some complexity and overhead that potentially shadows the gain from adding a strategy.

The second reason pertains to the different characteristics of Amersfoort. In a busier station like Amersfoort, trains cannot depart from the yard to the departure platform at the last moment; an average of 3 minutes (180 seconds) is required, as shown in the 4.11c. The time intervals during which a route from the yard to the departure platform can be executed vary between trains. Consequently, it is difficult to determine a search strategy that will guide the solver to the most promising values for parking on the departure platform. Likely, assigning values, based on a search strategy, to those variables would not result in a feasible solution, in contrast to the situation in Enkhuizen.

To conclude, using fixed search with the new version of Google OR-Tools reports better results for the default model-v2. Furthermore, specifying a search strategy for the variables on the departure platform appears to be useful for Enkhuizen, the same does not extend to the variables on the arrival platform.

4.5.2.1 Strategies on the number of movements

Figure 4.9 summarizes the results for variable and value selection strategies on the number of movements for Enkhuizen. This experiment was conducted only for Enkhuizen, because, in Amersfoort, the number of movements per train is already defined by the station layout and the high number of through trains. Furthermore, running time is already considerably low.

SelectLowerHalf reports better average performance for smaller instances, but for larger ones, performance is comparable to not using any strategy. *SelectMedi-*

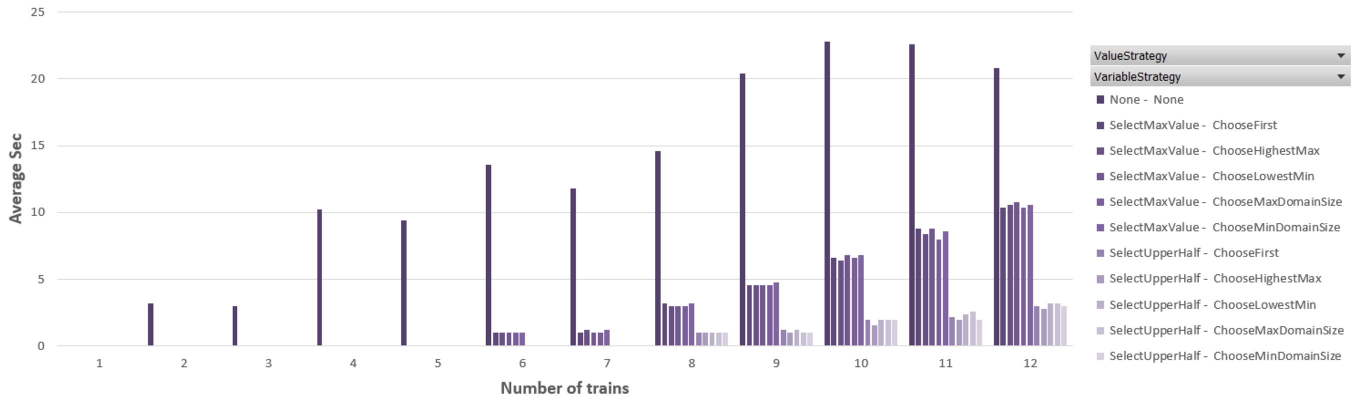


Figure 4.7: Comparison between the variable and value strategies for the variables denoting the parking time on the **departure** platform on **Enkhuizen** instances for model-v1.

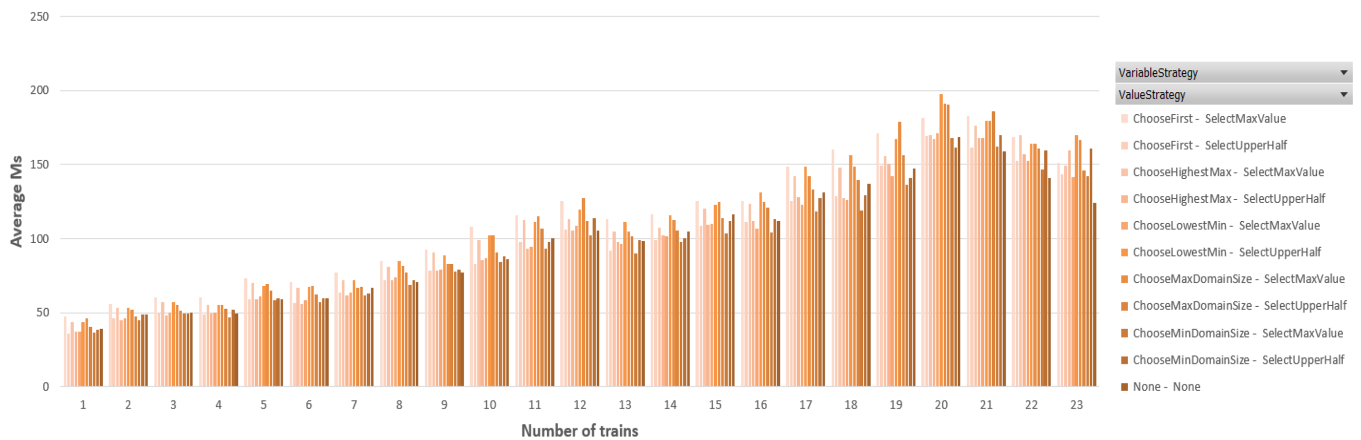


Figure 4.8: Comparison between the variable and value selection strategies for the variables denoting the parking time on the **departure** platform on **Amersfoort** instances for model-v1.

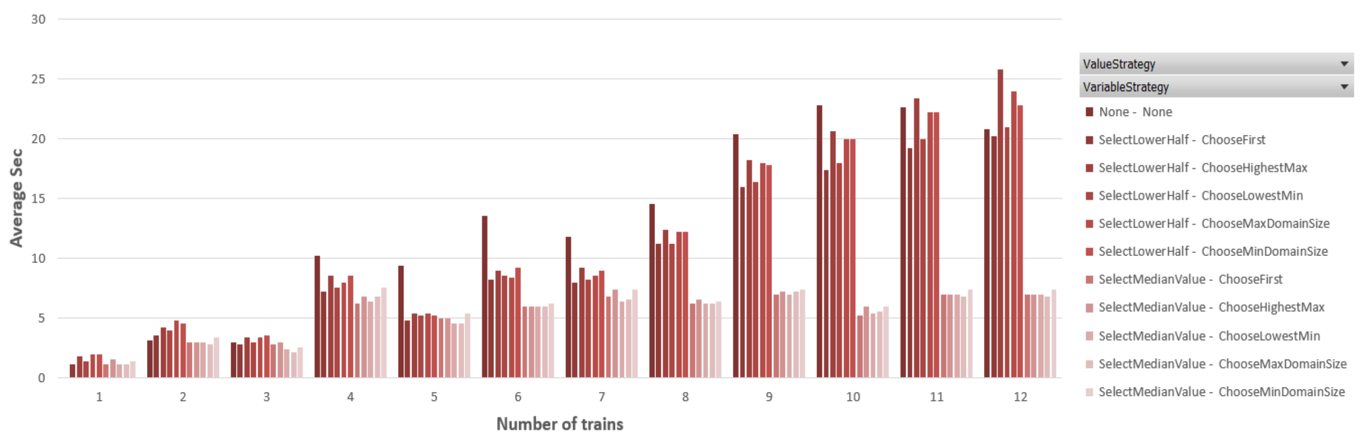
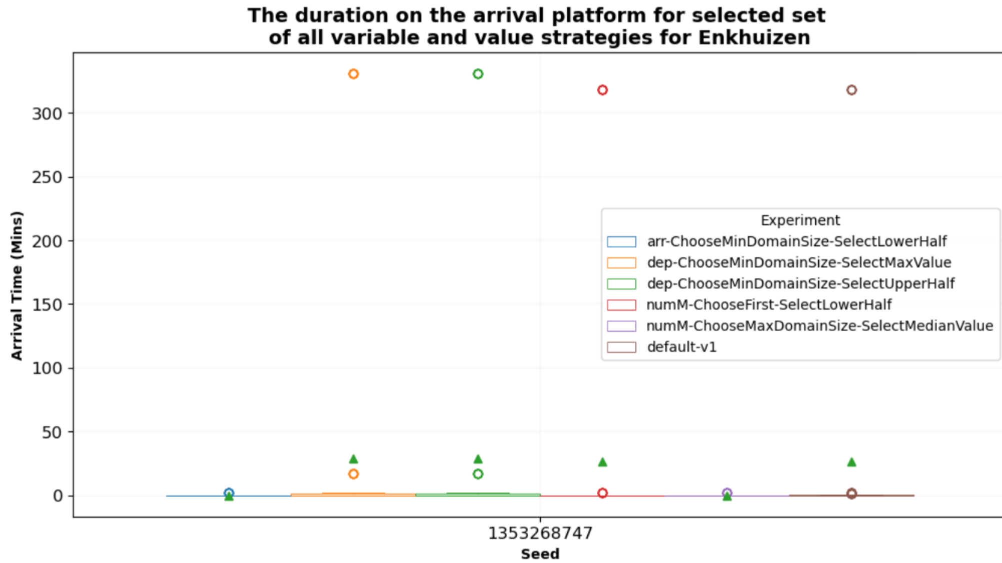
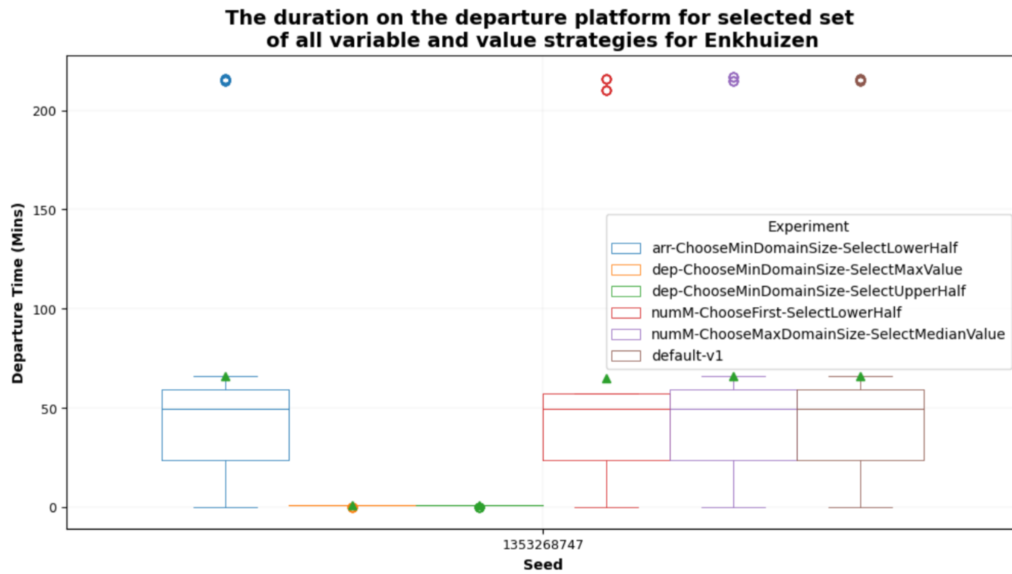


Figure 4.9: Comparison between the variable and value selection strategies for the variables denoting the **number of movements** for **Enkhuizen**.



(a)

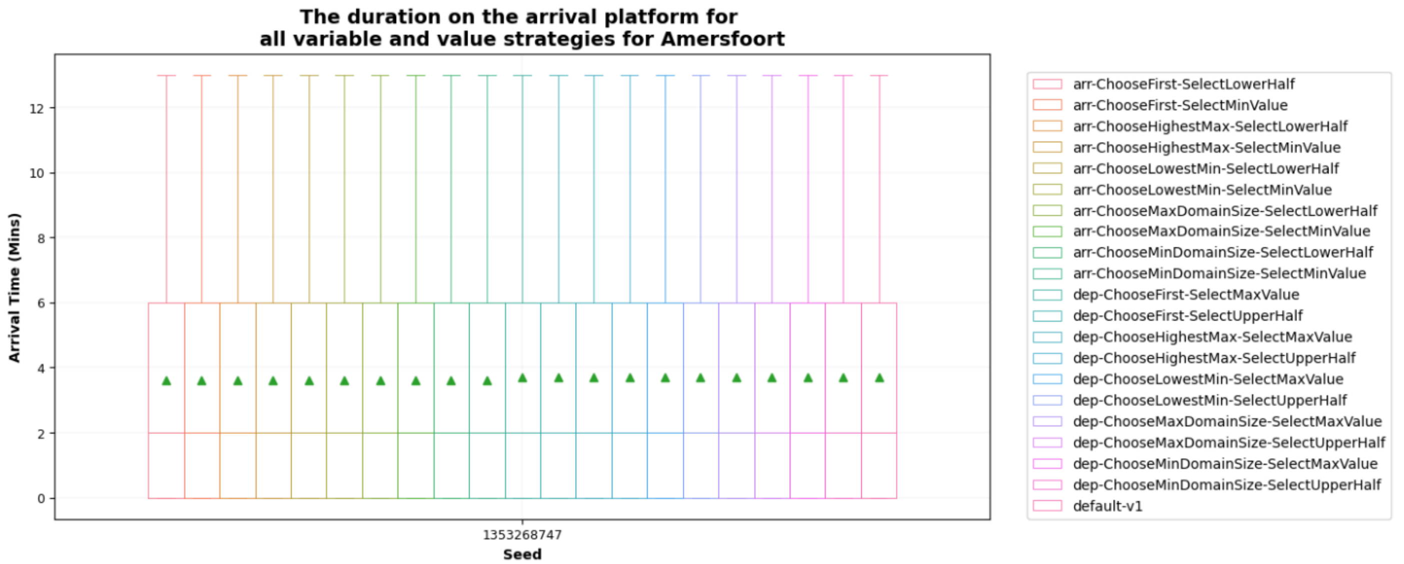


(b)

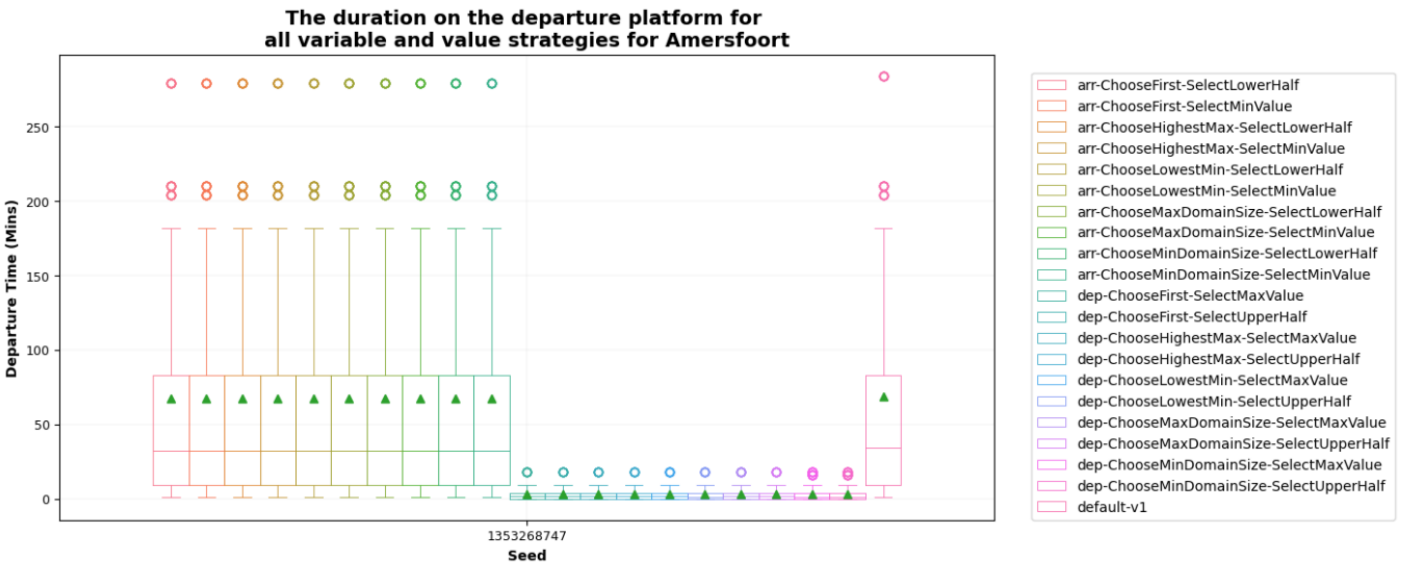
Seed	Experiment	Count_of_1	Count_of_2	Count_of_3
1353268747	arr-ChooseMinDomainSize-SelectLowerHalf	1.0	2.0	9.0
1353268747	default-v1	2.0	2.0	8.0
1353268747	dep-ChooseMinDomainSize-SelectMaxValue	2.0	2.0	8.0
1353268747	dep-ChooseMinDomainSize-SelectUpperHalf	1.0	4.0	7.0
1353268747	numM-ChooseFirst-SelectLowerHalf	3.0	9.0	0.0
1353268747	numM-ChooseMaxDomainSize-SelectMedianValue	0.0	7.0	5.0

(c)

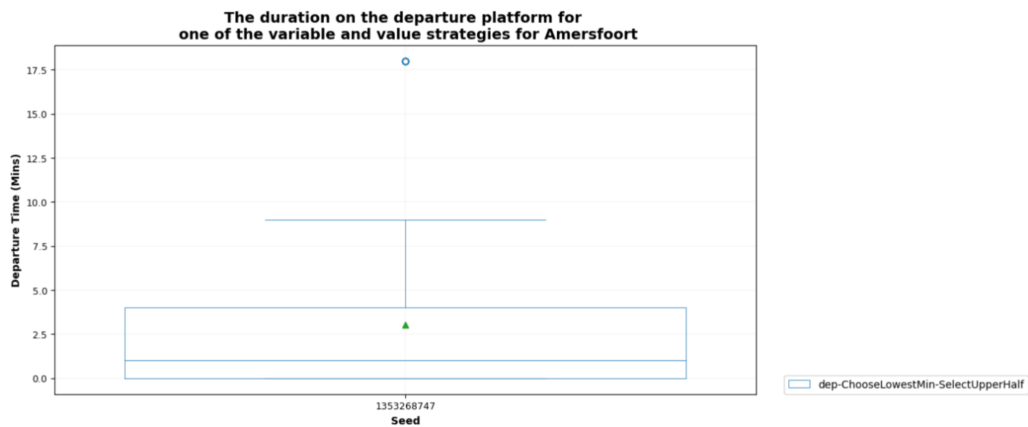
Figure 4.10: The three quality aspects for the instances with 12 trains for Enkhuizen and selected set of the variables and value strategies. One or two variables and value strategies are selected for each type of variable. The performance of the not presented pairs of strategies is analogous to those of the same type.



(a)



(b)



(c)

Figure 4.11: The two quality aspects for the real-life instance of Amersfoort for all executed pairs of variables and value strategies for one seed. The rest of the seeds are not presented, as they report the same results for those two metrics. In c), we detailed one of the strategies on the departure platform. The rest are analogous.

anValue, however, consistently records a better average performance of around 7-8 seconds. The reason for this result relies on the specifics of this instance. *SelectLowerHalf* attempts to assign 0 or 1 movement to different trains first. However, only three trains in Enkhuizen can move directly from the arrival to the departure platform immediately as in the other cases the arrival and departure platforms overlap. Hence, the train will be required to stay at the platform, however, we did not allow this (**R6** in 2.3.2). *SelectMedianValue* chooses for most of the trains two movements for the other three (Table 4.10c). The better running time is due to selecting first the values that lead to a feasible solution.

Last, during experimenting with model-v2, we observed that restricting the number of movements for all trains to less than or equal to two, or exactly three movements, produces instantaneous results (in less than a second).

Solutions' quality

Figure 4.11 presents the metrics on a subset of all pairs of variable and value strategies. The results for the rest of the seeds and strategies are similar.

Figure 4.10a demonstrates that there are no outliers when we impose a value strategy that minimizes the parking time on the arrival platform compared to the rest of the strategies that do not enforce this behavior. An exception is one strategy for the number of movements. Most values for all strategies are near zero again due to the default search strategy of Google OR-Tools.

Figure 4.10b shows the durations of the departure platform for the same pairs of strategies. The results align with previous discussions about the default solver strategy.

The Table in Figure 4.10c presents the number of movements. Here, *SelectLowerHalf* assigns the smallest values for each train, while *SelectMedianValue* assigns some trains with three movements.

For Amersfoort, the strategies on the duration on the arrival platform do not improve upon the default search strategy (Figure 4.11a) that is utilized for the arrival variables in the other cases (the default model and strategies for the departure platform). Figure 4.11b exhibits also expected behavior based on the previous discussion.

In Amersfoort, 7 trains are planned with zero movements (these are mostly the ones that split and combine at the yard), 22 will move once, and 11 - twice. 40 are all trains from all different types, while 23 are all trains that arrive at the platform (arrival, combined trains, and base shunt trains).

4.5.3 Compare both versions of the model

In this experiment, we answer the first hypothesis in 4.3.1 by comparing the performance of the first with the second version of the model, without adding any additional constraints or search strategies. For the instances, we again alter only the set of plannable trains for both Enkhuizen and Amersfoort.

Figure 4.13 presents the results for Amersfoort, with again each bar showing the average running time of all instances with the same number of trains and all seeds. Despite the already good performance of model-v1 on Amersfoort instances, Figure 4.13 shows that the second version consistently outperforms the first. Furthermore, the larger the instances (in terms of the number of trains), the greater the difference in the average running time. These results also follow for Enkhuizen (Figure 4.12).

In conclusion, in those two types of instances, model-v2 outperforms model-v1. In later Section 4.6.4.1, we analyze the number of variables and constraints in both models to elucidate the improvement. In the next experiment, we explore whether this trend persists when enabling more routes for the Amersfoort instances.

Solutions' quality

We observe similar patterns in the durations on the arrival and departure platforms as in the previous experiment in Figure 4.14 for Enkhuizen. In both versions, the solver selects similar values to those variables. Three routes per train are again the prevailing option. We investigate the solutions for both versions of Amersfoort in the next experiment.

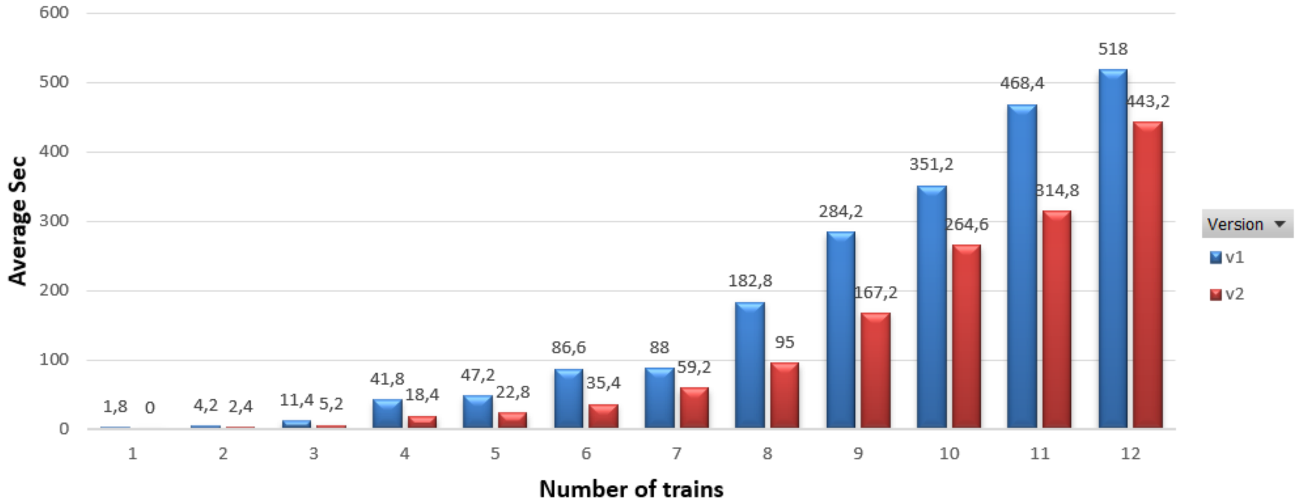


Figure 4.12: Comparison between the two model versions for Enkhuizen instances. Each bar shows the average running time in **seconds**, averaged between all instances with the same number of trains and all seeds.

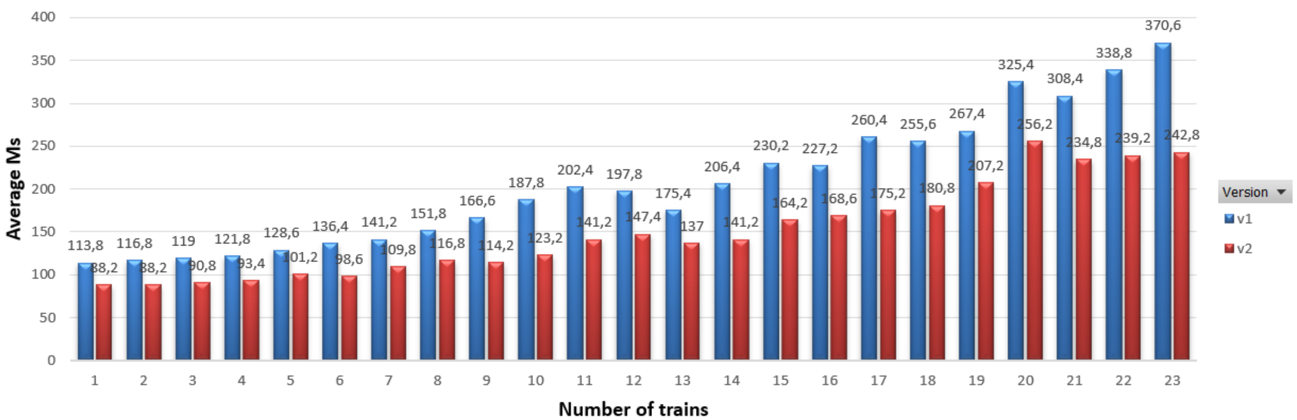


Figure 4.13: Comparison between the two model versions for Amersfoort instances. Each bar shows the average running time in **milliseconds**, averaged between all instances with the same number of trains and all seeds.

4.6 Experiments with instance characteristics

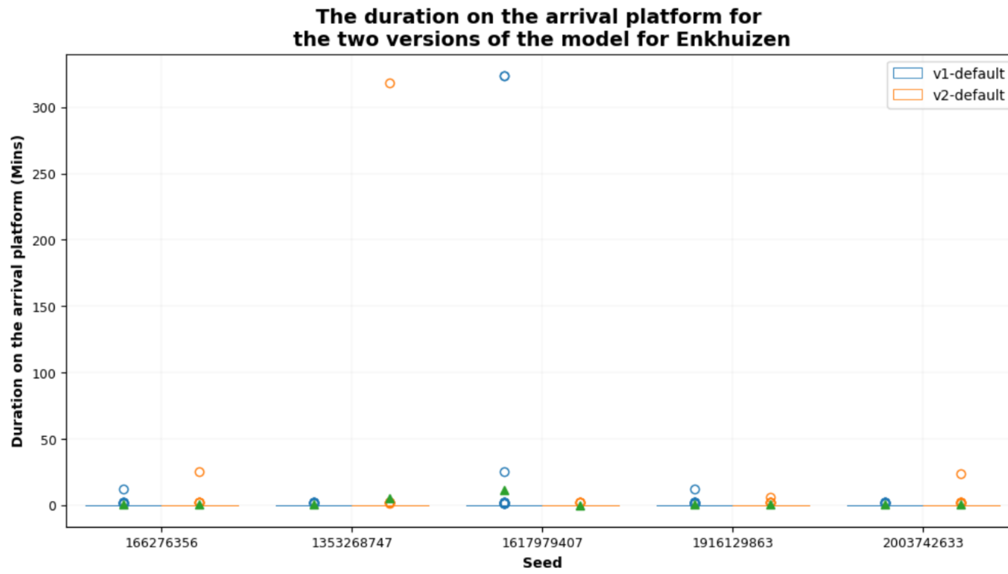
4.6.1 Additional routes for Amersfoort

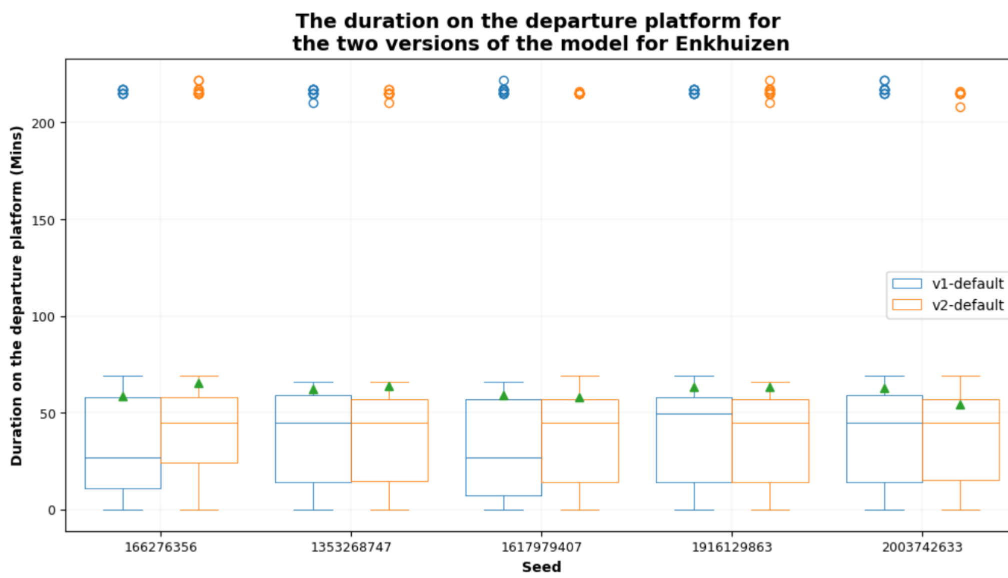
This experiment answers the hypotheses in 4.3.2 about increasing the number of routes. In general, there are multiple options for routing from platform A to platform B. In the default instances, we always utilized the shortest route between two points. In the following, we investigate the performance of both versions when allowing several alternative routes in addition to the shortest route.

Consider Figure 4.15 for model-v1. In the "alternatives-7" scenario, we include all shortest routes plus up to 7 alternative routes for each shortest route. Furthermore, the "alternatives-9" scenario extends with up to two more routes for each shortest route on top of all the routes considered by "alternatives-7". We say "up to" since not every shortest route has that many alternatives available. A specific route may only have 2-3 alternative routes, while others have up to 10-15. If the required number of alternatives exceeds those available, only the maximum available alternatives are included.

Furthermore, this experiment focuses on the bigger instances with more than 8 trains. The computation times for the smaller instances remain under one second.

As expected, increasing the number of possible routes leads to a higher computational time. Except for the entire instance with 23 trains, the orange bar, representing the performance with all possible alternative routes, is always the highest, followed by the





	Seed	Experiment	Count_of_1	Count_of_2	Count_of_3
0	166276356	v1	0	2	10
1	166276356	v2	0	4	8
2	1353268747	v1	0	4	8
3	1353268747	v2	0	2	10
4	1617979407	v1	0	1	11
5	1617979407	v2	0	2	10
6	1916129863	v1	0	2	10
7	1916129863	v2	1	2	9
8	2003742633	v1	0	4	8
9	2003742633	v2	1	1	10

Figure 4.14: The three quality aspects for both versions of the model and the real-life instances for Enkhuizen.

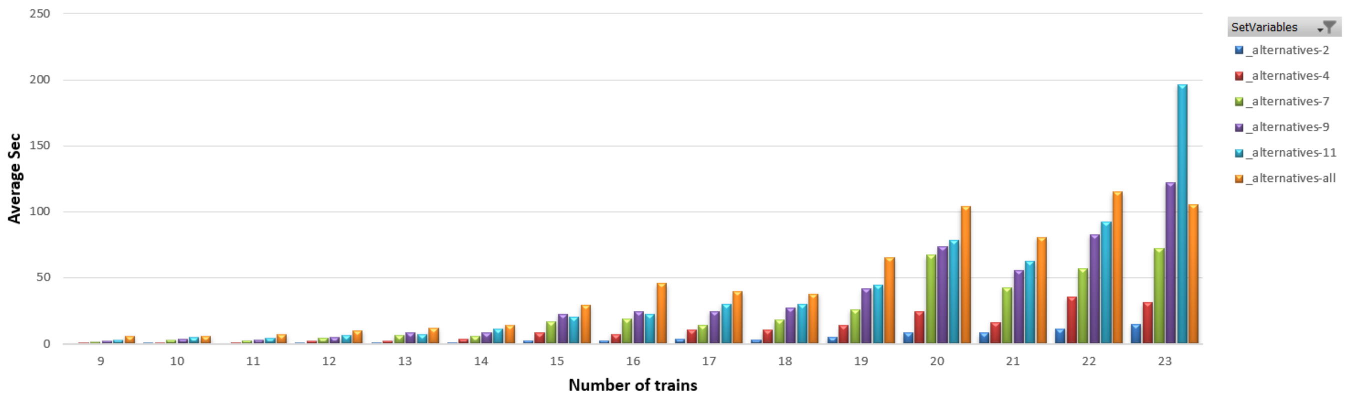


Figure 4.15: Comparison between the average run time when adding additional routes between 2 and all possible additional routes.

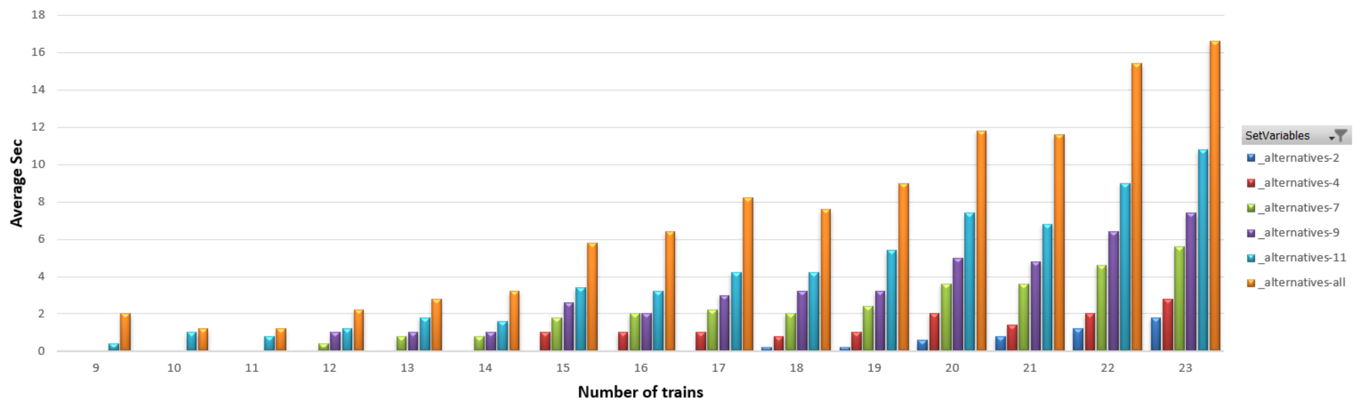


Figure 4.16: Comparison between the average run time when adding additional routes between 2 and all possible. A single bar is obtained by averaging all instances with the same number of trains and all seeds.

blue and purple bars. For the entire instances, 11 or 9 additional routes perform worse than adding all possible routes. Detailed performance for this exception is shown in Appendix Figure D.2.

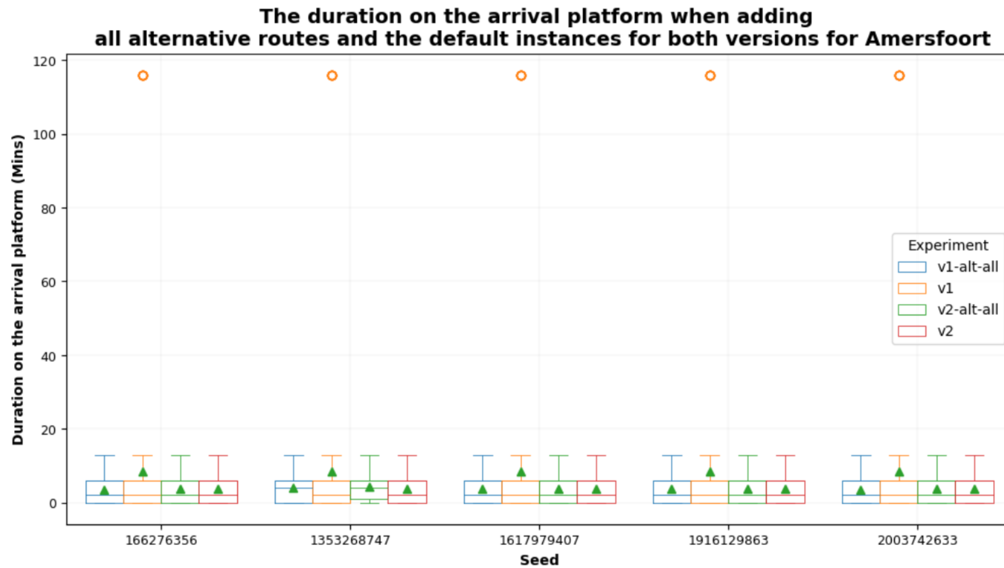
There is no way to measure route goodness in terms of which routes will lead to faster solutions. In a sense, it appears that adding a few more routes from "alternatives-11" to "alternatives-all" results in easier instances. Therefore, we cannotconclude that the running time will certainly increase upon including a few additional routes, but it is evident that the performance with all routes downgrades with respect to "alternatives-2".

Figures 4.16 summarize the findings of running model-v2. An important but subtle observation is that the average running time decreases from 200 for model-v1 "alternatives-11" to 18 seconds for model-v2 and all alternative routes. For this version of the model, the trend of higher numbers of alternative routes leading to higher average running times is consistent across all instances.

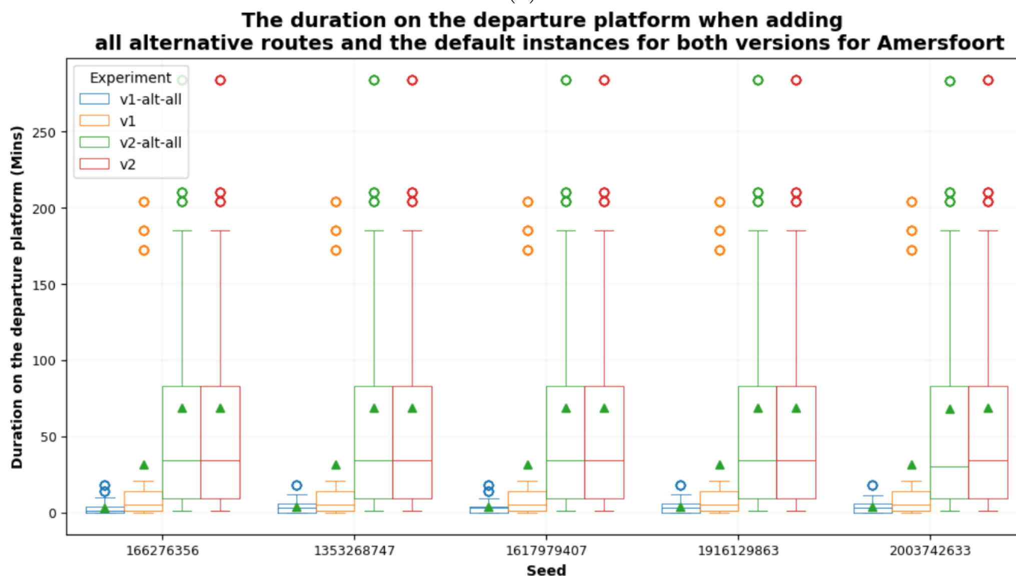
4.6.1.1 Solutions' quality

Figure 4.17 illustrates the metrics for both versions of the models on the real-life Amersfoort instance, with all additional routes and without any (the default). Figure 4.17a indicates that additional routes enable the solver to select slightly lower values for the duration on the arrival platform for the v1. Both v2 and v2-alt-all (v2 with alternatives) perform as v1 with alternatives (v1-alt-all). Not that similar are the results on the departure platform presented in Figure 4.17b. V2 assigns longer durations on the departure platform and does not improve upon adding more routes, in contrast to v1.

We investigated the solutions for the second version with the parameter to keep all feasible solutions during presolve set to true. The duration on the departure platform for the second version was significantly lower (see the additional plot in Appendix D.3). The durations, when alternatives are considered, do not change with or without the parameter. However, without any variable or value strategy enabled, the default search tends to select higher values for the variables control-



(a)



(b)

Figure 4.17: The two quality aspects for model-v1 and the instances with all trains for Amersfoort. Only trains that do arrive at the arrival platform and the departure platform are taken into account in the first and second figures, respectively.

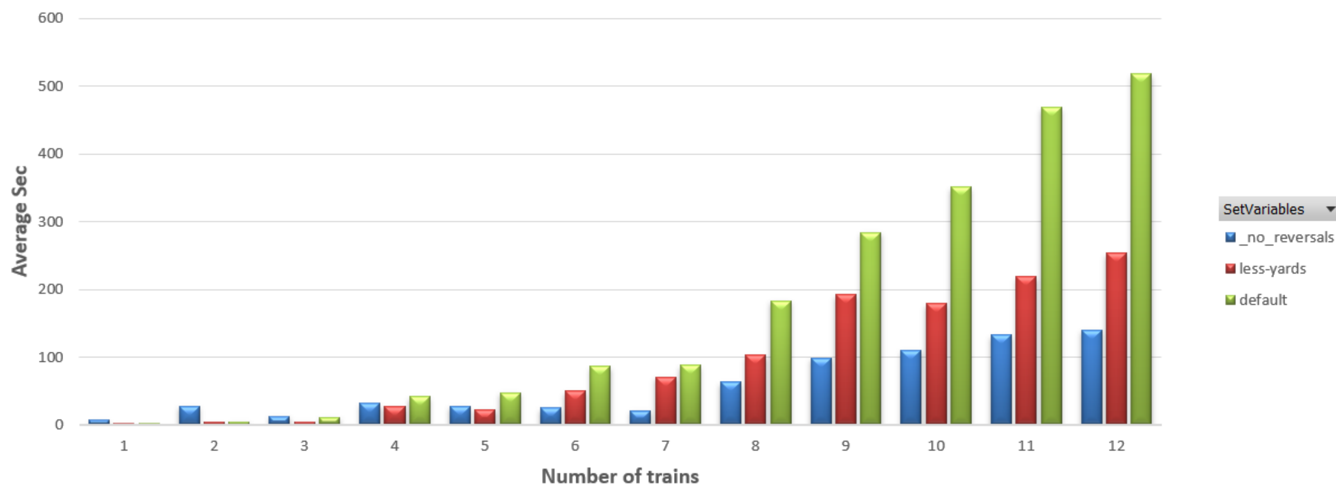


Figure 4.18: Comparison of the performance on Enkhuizen instances when a yard or the reversals are removed with the default model-v1.

ling the duration on the departure platform, influenced by the already assigned variables.

The number of movements for all versions with and without alternatives is the same: 7 trains do not perform any movements, 22 - one, and 11 take two.

4.6.2 Removing a yard, or reversals

Figure 4.18 presents the aggregated results for model-v1 on Enkhuizen instances under two modifications: without the reversal routes (red), without the yard containing the 405_406 tracks (Figure 3.1 for the station layout) and in green the default instances. Removing reversals leads to the greatest improvement. Interestingly, removing a yard does not seem to enhance the performance to the same extent as stated in hypothesis 7 (4.3.2). We further analyze the results, but first investigate the solutions in terms of the three metrics.

4.6.2.1 Solutions' quality

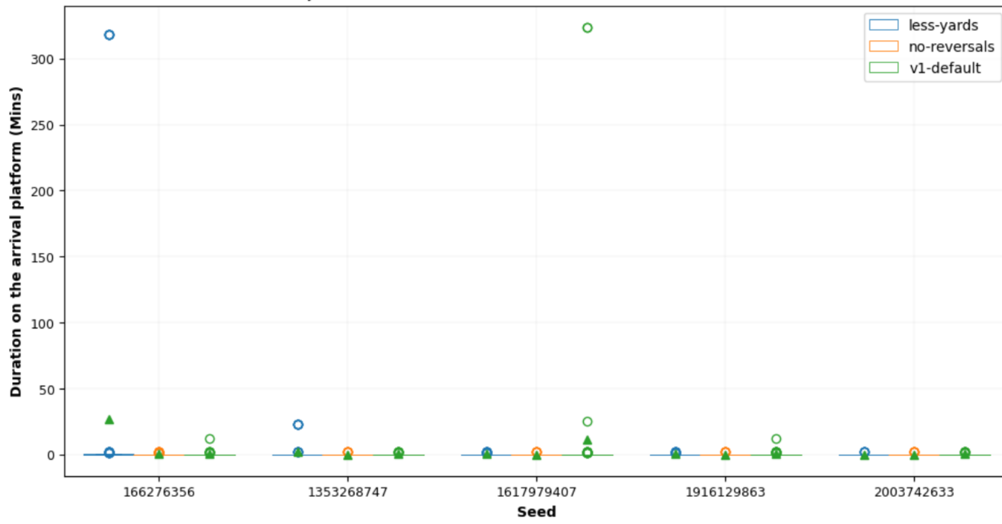
The values of the variables on the arrival and departure platforms align with the default solver strategy, as shown in Figure 4.19.

The number of movements in the default instances and when a yard is removed are comparably the same, with three movements prevailing in both cases. However, removing the reversals reduces the number of movements to two for most trains.

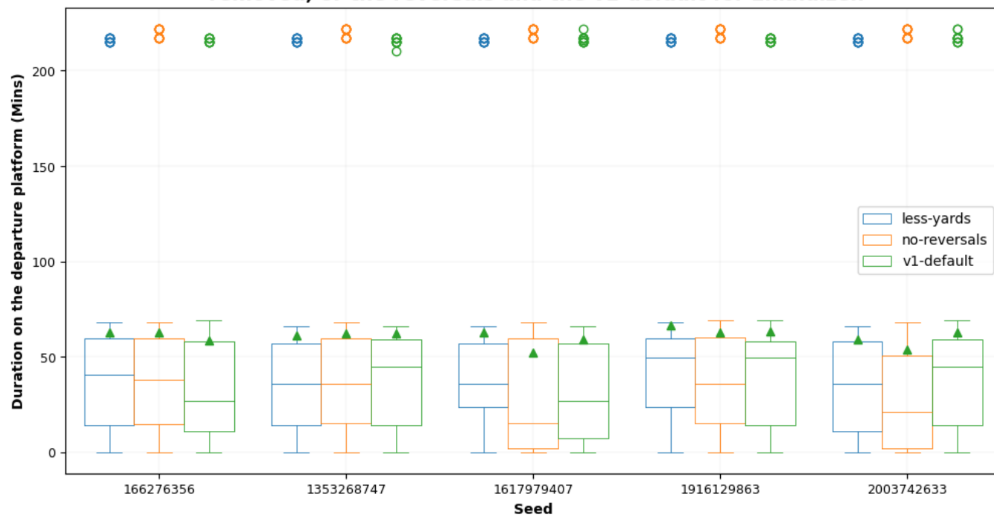
Removing the reversals allows a train that arrives at a platform, to only move to one of the yards opposite the platforms (either 404_407 or 405_406) and then return to the platforms (Figure 3.1 for the station layout). For instance, if a train arrives at Ekz_1 and departs at Ekz_2 and the solver selects yard 404_407. After reaching 404_407, going to 405_406 is impossible as a reversal is required. Similarly, moving to yard 403 is not feasible as a reversal is required to reach the departure platform, Ekz_2. As most trains exhibit this scenario, most trains are constrained to a maximum of two movements.

Removing the reversals or a yard eliminates approximately one-third of the available routes, which explains the improved performance compared to the default instances. Furthermore, we previously observed that a strategy assigning the most promising values for the number of movements improves performance. Removing the reversals implicitly constrains the number of movements for most of the trains to two. Figure 4.19c does not impose such a restriction on the number of movements. Hence, the additional improvement comes from this implicit constraint.

To conclude, this experiment confirms some of the previous results. In particular, the number of routes and number of movements are again shown as important key parameters.



(a)



(b)

Seed	Experiment	Count_of_1	Count_of_2	Count_of_3
166276356	less-yards	1	2	9
166276356	no-reversals	0	11	1
166276356	v1	0	2	10
1353268747	less-yards	0	1	11
1353268747	no-reversals	0	11	1
1353268747	v1	0	4	8
1617979407	less-yards	0	1	11
1617979407	no-reversals	0	11	1
1617979407	v1	0	1	11
1916129863	less-yards	0	2	10
1916129863	no-reversals	0	11	1
1916129863	v1	0	2	10
2003742633	less-yards	0	1	11
2003742633	no-reversals	0	11	1
2003742633	v1	0	4	8

(c)

Figure 4.19: The three quality aspects for the instances with 12 trains for Enkhuizen when one yard or all reversals are removed and the default model-v1.

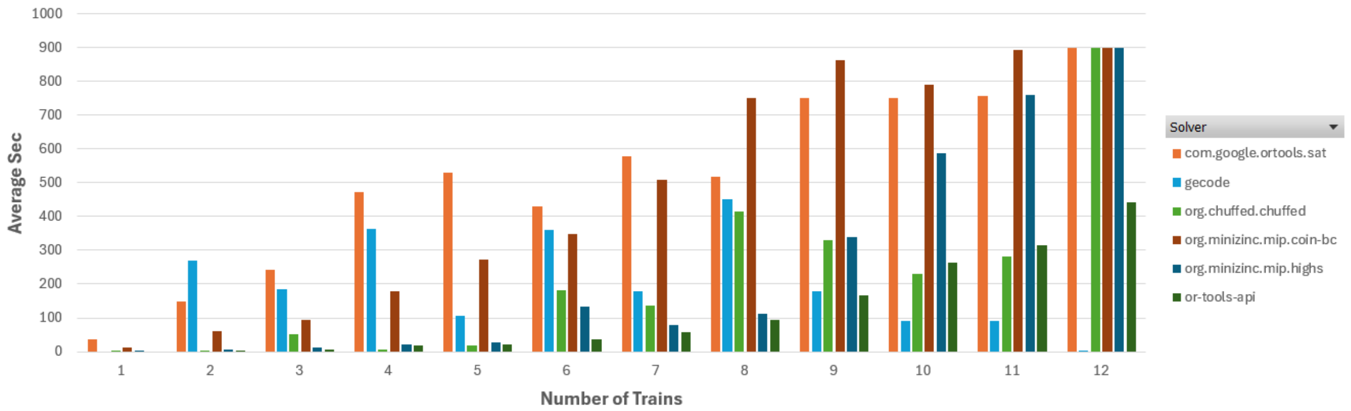


Figure 4.20: The average performance for each solver per number of trains for the MiniZinc model (version 2) and also the average performance of the model-v2, implemented with Google OR-Tools API (or-tools-api).

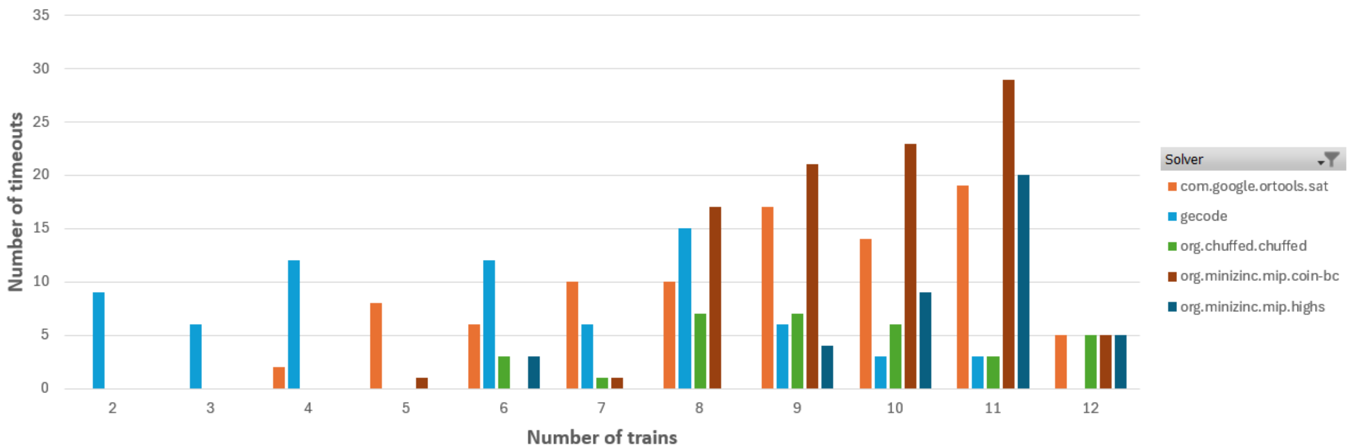


Figure 4.21: The number of timeouts on Enkhuizen instances per solver for the MiniZinc model (version 2). For each number of trains from one to eleven, there are 30 instances (10 different instances, each executed 3 times). The instances with 12 trains correspond to the real-life scenario, executed five times.

4.6.3 Compare other solvers on model-v2

Finally, to answer the hypothesis in 4.3.3, we have re-implemented the part of model-v2, required for Enkhuizen, in MiniZinc. The advantage of using MiniZinc is the generic interface for writing models for different solving backends.

The model is executed on all Enkhuizen instances with the following solvers Google OR-Tools, Gecode, Coin-BC, HiGHS, and Chuffed using their default configurations. Gecode and Chuffed are open-source constraint programming solvers. Chuffed builds upon Gecode with lazy clause generation, an important constraint programming concept discussed in [Stuckey, 2010]. According to the authors, disabling the lazy clause generation reports performance comparable with older versions of Gecode [chu, 2010]. In addition, they claim that overhead from lazy clause generation ranges from negligible to perhaps around 100%. HighS and Coin-BC are open-source mixed integer linear programming solvers. HighS is also enhanced with serialization and parallelization. Google OR-Tools is a portfolio solver that employs various solving technologies when multi-threading is enabled. However, unrelated to the number of threads Google OR-Tools also utilizes LP-cutting planes to remove suboptimal or unfeasible parts of the search space.

We evaluate each solver again on all 335 default instances of Enkhuizen. The performance is again averaged across instances with the same number of trains, depicted in Figure 4.20. Due to a considerable number of timeouts (again with a 15-minute upper bound on the running time), we also provide the number of timeouts per solver and instance size in Figure 4.21.

Two interesting observations emerge. Firstly, we compare the performance of model-v2, implemented in

MiniZinc and solved with the Google OR-Tools backend, and model-v2, implemented with the Google OR-Tools API. Based on the number of timeouts, reaching almost 20 for the instances with 11 trains, and a higher running time (much over 443 seconds, the average performance of model-v2 on the entire Enkhuizen instance (Figure 4.12), we conclude that the model-v2, implemented with the Google OR-Tools API, performs better. The outcome is likely due to the design of the MiniZinc model that hinders propagation and/or efficient presolve.

Secondly, on smaller instances (up to 8 trains) Chuffed and HighS demonstrate the best performance. For those, Gecode exhibits the highest number of timeouts, leading to poorer performance. However, as the instance size increases, the number of timeouts for Gecode decreases and the solve time is significantly reduced, falling below 0 seconds for the entire instance. This is orders of magnitude faster than the performance of the default Google OR-Tools model, reducing the average solve time from 443 seconds to less than a second.

Gecode is the only solver that does not time out on the entire instance, while the rest of the solvers time out on all five executions of the entire Enkhuizen instance. The interested reader can further observe the data. Gecode either solves the instance in less than a second or times out. This intriguing result warrants further exploration. However, due to time limitations and slightly out of the project's scope, a reasonable explanation is deferred to future work.

Possible explanations could be the different structure of some smaller instances in terms of search space, compared to the larger ones. Alternatively, perhaps optimizations and heuristics are more effective when the full instance with all trains is used. Nevertheless, it appears that either the newly built features of Gecode or the overhead of the lazy clause generation of Chuffed are the main reasons for Gecode outperforming Chuffed.

The remaining solvers exhibit the expected behavior: as the number of trains in the instance increases, both the number of timeouts and the average solving time increase.

4.6.4 Deeper analysis of some results

4.6.4.1 The two model versions

We begin with the differences in the search log of the solver for both versions of the model. The search statistics are gathered from running the real-life instance of Enkhuizen with the same seed. Listings 4.1 and 4.2 illustrate the initial models.

First, observe the number of variables, which is slightly lower in v2. Next, all types of constraints (denoted as k) and their occurrences in the model are presented. The first version contains constraints with more than 3 linear expressions ($\#kLinearN$). This indicates higher complexity. Additionally, it includes the boolAnd constraint ($kBoolAnd$), which is part of the alternative constraint and has been removed in the second version. Although the other constraints are present in both versions, they are more prevalent in the first version.

```
Initial satisfaction model '':
#Variables: 3262
  - 158 different domains in [-1,201900]
  with a largest complexity of 1.
  - 32 constants in {-1,0,1,2,3,4,121980
  ,124140,
  14...93140,194940,196740,
  197820,198540,199620,
  201600}
#kBoolAnd: 70 (#enforced: 70)
(#literals: 358)
#kElement: 28
#kInterval: 1709 (#enforced: 1065)
#kLinear1: 1903 (#enforced: 1175)
#kLinear2: 2402 (#enforced: 642)
#kLinear3: 1021 (#enforced: 918)
#kLinearN: 36 (#enforced: 36) (#terms: 258)
#kNoOverlap: 12 (#intervals: 1447,
#optional: 815, #variable_sizes: 815)
```

Listing 4.1: The **initial** model statistic when running **model-v1** on the whole Enkhuizen problem. The full version here.

```
Initial satisfaction model '':
#Variables: 3083
...
#kElement: 28
#kInterval: 1530 (#enforced: 886)
#kLinear1: 1903 (#enforced: 1175)
#kLinear2: 2042 (#enforced: 282)
#kLinear3: 989 (#enforced: 886)
#kNoOverlap: 12 (#intervals: 1447,
#optional: 815, #variable_sizes: 815)
```

Listing 4.2: The **initial** model statistic when running **model-v2** on the whole Enkhuizen problem. The full version here.

After gathering the initial statistic, the solver undergoes a presolve step. During this phase, the solver deduces as much information as possible from the model and proceeds to reduce and simplify it based on intelligent heuristics.

After presolve, the solver again produces statistics for the models. In Listings 4.3 and 4.4 demonstrate the results for the first and second versions, respectively. The smaller number of constraints and an initial number of boolean variables for the second version indicate that the solver has fixed and inferred more about version two than version one.

This investigation further supports our claim that v2 improves upon v1.

```
Presolved satisfaction model '':
#Variables: 606
  - 261 different domains in [0,201600]
```

```
  with a largest complexity of 4.
#kBoolAnd: 51(#enforced: 51)(#literals: 96)
#kExactlyOne: 41 (#literals: 208)
#kInterval: 1661 (#enforced: 982)
#kLinear1: 400 (#enforced: 400)
#kLinear2: 417 (#enforced: 393)
#kLinear3: 233 (#enforced: 111)
#kLinearN: 6 (#enforced: 6) (#terms: 51)
#kNoOverlap: 15 (#intervals: 1010,
#optional: 775, #variable_sizes: 87)
...
Initial num_bool: 514
```

Listing 4.3: The model statistic after **presolved** when running **model-v1** on the whole Enkhuizen problem. The full version here.

```
#kBoolAnd: 1 (#enforced: 1) (#literals: 2)
#kExactlyOne: 12 (#literals: 38)
#kInterval: 1064 (#enforced: 379)
#kLinear1: 84 (#enforced: 84)
#kLinear2: 14 (#enforced: 10)
#kLinear3: 93 (#enforced: 32)
#kNoOverlap: 23 (#intervals: 593,
#optional: 367, #variable_sizes: 57)
...
Initial num_bool: 128
```

Listing 4.4: The model statistic after **presolved** when running **model-v2** on the whole Enkhuizen problem. The full version here.

4.6.4.2 Additionally constraining the parking time

The search statistics when constraining the parking time to be up to 15 minutes are shown in Listings 4.5 and 4.6. Surprisingly, the model after the presolve (Listing 4.6) is almost identical to the model after the presolve of the first version without the additional constraints (Listing 4.3). The significant difference lies in the LP statistics, which reports solutions with fewer simplex iterations and lower optimal values. The part with the LP statistics is visible in Listings 4.7 4.8 for the model with and without additional constraints, respectively. After presolve, the problem with the additional constraints is solved faster, indicated by both the running time and the lower number of simplex iterations. Simplex is the core algorithm for linear programming (LP). Details about it in [sim, 2020].

```
Initial satisfaction model '':
#Variables: 3262
  - 158 different domains in [-1,201900]
  with a largest complexity of 1.
  - 32 constants in {-1,0,1,2,3,4,121980,
  124140,14 ... 93140,194940,196740,
  197820,198540,199620,201600}
#kBoolAnd: 70 (#enforced: 70)
(#literals: 358)
#kElement: 28
#kInterval: 1709 (#enforced: 1065)
#kLinear1: 1903 (#enforced: 1175)
#kLinear2: 2402 (#enforced: 642)
#kLinear3: 1021 (#enforced: 918)
#kLinearN: 36 (#enforced: 36)
(#terms: 258)
#kNoOverlap: 12 (#intervals: 1447,
#optional: 815, #variable_sizes: 815)
```

Listing 4.5: The **initial** model statistic when running **model-v1** on the whole Enkhuizen problem **with additional parking constraint**. The full version can be seen here.

```
 Presolved satisfaction model '':
#Variables: 582
  - 236 different domains in [0,201600]
  with the largest
  complexity of 4.
#kBoolAnd: 96 (#enforced: 96)
(#literals: 186)
#kExactlyOne: 36 (#literals: 183)
#kInterval: 1607 (#enforced: 918)
#kLinear1: 215 (#enforced: 215)
#kLinear2: 406 (#enforced: 384)
#kLinear3: 215 (#enforced: 92)
#kLinearN: 1 (#enforced: 1) (#terms: 6)
#kNoOverlap: 15 (#intervals: 961,
#optional: 726,
#variable_sizes: 78)
...
Initial num_bool: 424
```

Listing 4.6: The model statistic after **presolved** when running **model-v1** on the whole Enkhuizen problem **with additional parking constraint**. The full version can be seen here.

```
LP statistics:
  final dimension: 2 rows, 10 columns,
  6 entries with magnitude in
  [1.000000e+00, 1.000000e+00]
  total number of simplex iterations: 2
  num solves:
    - #OPTIMAL: 9
  managed constraints: 5
  total cuts added: 0 (out of 0 calls)
LP statistics:
  final dimension: 7 rows, 16 columns, 21
  entries with magnitude in
  [1.000000e+00, 1.000000e+00]
  total number of simplex iterations: 7
  num solves:
    - #OPTIMAL: 9
```

Listing 4.7: The **LP statistic** after presolved when running **model-v1** on the whole Enkhuizen problem with **additional parking constraint**. The full version can be seen here.

```
LP statistics:
  final dimension: 4 rows, 10 columns,
  12 entries with magnitude in
  [1.000000e+00, 1.000000e+00]
  total number of simplex iterations: 6
  num solves:
    - #OPTIMAL: 41792
  managed constraints: 5
```

```
    total cuts added: 0 (out of 0 calls)
LP statistics:
    final dimension: 6 rows, 16 columns,
    18 entries with magnitude in
    [1.000000e+00, 1.000000e+00]
    total number of simplex iterations: 1382
    num solves:
      - #OPTIMAL: 37615
```

Listing 4.8: The **LP statistic** after presolved when running **model-v1** on the whole Enkhuizen problem **without parking constraint**. The full version can be seen here.

As previously acknowledged, imposing a search strategy for the duration of the arrival platform is not beneficial for reducing computation time. Therefore, we opt to separately constrain the duration on the departure and arrival platform to 15 minutes and examine the performance and the search log of the solver.

We summarize the main findings. Constraining the duration on the departure platform yields nearly the same performance in terms of running time, the number of simplex iterations, and the size of the found LP solutions (search log - here). In contrast, when we restrict only the duration on the arrival platform (see here) the performance is similar to the model without any additional constraints. This outcome is likely related to the fact that the solver's default search strategy already acts as if constraining the durations on the arrival platform, according to the solutions' quality results. this aspect. This result aligns with the outcome of the experiments with the search strategies.

4.6.5 Main contributions

The evaluation demonstrated that the second version of the model improves upon the first. The second version requires fewer variables and constraints to model the same instance (see initial and presolve statistics in Section 4.6.4.1). Both versions were evaluated on the default instances and the instances with the alternative routes for Amersfoort. In both experiments, the second version consistently outperforms the first.

The experiments produce valuable insights into the influences of more trains and more routes on the model for two different real-life instances. As expected, a higher number of trains results in slower performance for a specific instance type. However, this result does not translate between different instances. For instance, Amersfoort, despite being a larger instance, exhibits better running time compared to Enkhuizen. Additionally, as anticipated, including additional routes generally leads to slower performance.

The experiments and the deeper analysis further revealed several insights into the model regarding the instance characteristics.

The variables controlling the duration on the departure platform were found to enhance the performance of the model. Further constraining them on model-v1 with default search, results in instantaneous solutions, in less than a second, compared to 518 seconds otherwise. Similarly, implementing a variable and value strategy that prioritizes the selection of those variables and guides towards shorter departure parking durations improves running time.

Both the strategies and the constraints also enhance the quality of the solutions produced for Enkhuizen. Other instances with characteristics similar to Enkhuizen's would also benefit from the above-mentioned enhancements. In particular, these features are a higher number of allowed movements, indicated by more than one yard, and the larger search space, corresponding to less through traffic.

Highly constrained instances, such as Amersfoort, where routing of trains is possible only within limited time windows that are not necessarily close to the arrival or departure times, may not exhibit improvement, especially if the solving time is already low. However, specifying a strategy in such cases can still be beneficial for obtaining better solutions.

Performing analogous experiments on the variables controlling the duration on the arrival platform did not result in any improvement. Neither was the quality of the solutions enhanced, as the default strategy of the solver was already preferring values, leading to shorter durations on the arrival platform.

The number of movements is another important parameter that reduces the computation time and can improve the solutions for instances where a higher number of movements is possible, such as Enkhuizen. The experiment with the removal of a yard and reversals further confirmed that the number of movements, even when implicitly constrained, enhances the computation time.

Introducing additional alternative routes does increase the running time. However, even with all alternative routes, the second version of the model solves Amersfoort on average under 18 seconds. The correlation between an increased number of routes and slower computation was also validated by the experiment with a yard or the reversals removed. Despite this result, it also became evident the addition of a few more routes does not necessarily degrade performance. Incorporating all routes in the first version of the solver yields better average performance than limiting the number of routes to up to 11 routes.

The new version of Google Or-Tools (9.10) together with the transition from default to fixed search, corresponds to an improvement of up to 20 times for Enkhuizen and two times for Amersfoort.

Finally, the experiments in MiniZinc yielded intriguing outcomes. Model-v2, implemented with the Google OR-Tools API, defeats its twin (the model-v2, implemented in MiniZinc and solved with the Google OR-Tools backend, reducing the average solve time from over 900 seconds to approximately 443 seconds. Although this result was attributed to a less efficient MiniZinc model, this did not hinder Gecode's excellent performance on the entire instance. Gecode experienced timeouts primarily on smaller instances. However, as the instance size increased, its performance improved, eventually achieving no timeouts on the entire real-life instance for Enkhuizen and solving it within

milliseconds. This is an order of magnitude faster than the default model-v2, implemented with the Google OR-Tools API, reducing the average solve time from 443 seconds to less than a second.

Conclusion

In this research, we designed a constraint programming model for the Shunt Routing Problem (SRP) at NS. A second improved version was also proposed and evaluated. The model will be utilized for constructing the initial solution for the Local Search at NS. It incorporates all essential feasibility requirements and generates conflict-free solutions, unlike the current algorithm at NS. The performance on Amersfoort is instantaneous (under a second). In comparison, the Local Search algorithm requires up to 14 hours to solve all four subproblems for the same instance and often being impeded by the decisions part of SRP. For instance, identifying routing possibilities if the possible time interval to execute the movement is very short. By providing a conflict-free solution for the Shunt Routing problem, we anticipate a significant reduction in the computation time for the Local Search.

As already acknowledged, a model solving the Shunt Routing Problem the Eindhoven station, was developed previously by [Wattel, 2021]. In his thesis, he concludes that CP appears a promising and well-suited approach for addressing the Shunt Routing Problem. Our evaluations also demonstrated this result by applying the model to two other real-life instances, as well as multiple others, generated from these. We confirmed that the CP model exhibits reasonable performance in solving the ***generic*** Shunt Routing Problem. Hence, aligning with the goal of NS to produce an algorithm capable of solving TUSP at any station.

In contrast to the experimental setup described by [Wattel, 2021], which applied the model to time intervals of 2 or 4 hours and once over a 24-hour period, we evaluate the performance over a continuous time window of two to three days.

In addition, route reservations are performed in terms of sections that more accurately reflect the situation in practice compared to the previously employed method. There, the entire route is reserved before the movement, and the infrastructure is released only after the train has completed the transit. The downside of the new approach is the more interval variables and constraints for infrastructure reservations. However, this complication did not significantly hinder performance. For instance, solving the Amersfoort instance with all additional routes and in the presence of all 202 sections still demonstrates reasonable performance (within 18 seconds for model-v2). This result contributes to the objective of NS, to replicate the real-world situation while still adhering to reasonable performance.

We further proceed with important insights regarding the model that could be considered to further improve and adjust the performance and solutions obtained.

The experiments reveal the usefulness of imposing a search strategy on both the solution quality and the computation time. Furthermore, variable and value strategies can, in certain cases, serve as alternatives to an objective function. For instance, rather than imposing an objective function to minimize the duration on the departure platform which would likely degrade the performance, one could impose a variable or value strategy. This approach not only produces more desirable solutions but also reduces computation time. This effect of the variable and value strategies could be applied to other types of instances at NS and explored and utilized instead of other objective functions.

Furthermore, the implication of some of the variables, such as the duration on the departure platform and the number of movements for Enkhuizen, on performance could be considered for solving other instances and improving the model. This could be achieved by, for instance, suggesting the most probable values for these variables using solver hints.

Insights into how other solvers perform on the Shunt Routing Problem at NS can identify and open new avenues for future research. In particular, Gecode seems to be a promising competitor of Google OR-Tools for this problem. All evaluated solvers are open-source and can be utilized by NS at no cost, although additional efforts would be required to integrate them into operational use.

Additionally, the experiments provided a further understanding of the parameters and search strategies available within Google OR-Tools.

To summarize, the second version of the model could provide a good foundation for a model addressing the Shunt Routing Problem at NS. Understanding the impact of certain variables, solver parameters, and search strategies on performance is invaluable, as this knowledge can be applied to further enhance the model and improve solution quality for various types of instances.

Future work

While we conducted several experiments with both versions of the model, the project's duration did not permit the exploration of others, by no means less important. The following are our recommendations for future research.

Local Search

Future research should begin with an investigation of the influence of the new model on the Local Search. In particular, to confirm whether the model is still useful to decrease the running time of the Local Search at least to some extent, a conclusion initially suggested by [Wattel, 2021]. To achieve this, the output of the model should be integrated with the input of the subsequent model addressing the Train Parking Problem. The model may require the addition of certain details, such as the reversal time on the platform or the inclusion of other idle trains on the platform, to more closely resemble a final solution of the Local Search. The closer to the final solution, the fewer adjustments the Local Search will need to perform, thereby shortening the computation time.

Other types of instances

The experiments provided several key insights into the model, as well as the underlying problem and the types of instances evaluated. Nevertheless, to obtain an even deeper understanding of the performance strengths and weaknesses of the model, it is beneficial to evaluate the model in more real-life instances from different stations.

Other searches of the solver

Modifying the solver's default search strategy to a fixed search significantly reduced computation time for Enkhuizen. Additional strategies, such as LP search (which utilizes heuristics from LP relaxation), pseudo-cost search, and others, specified here worth exploring.

Solving in phases

Constraining the variables for the duration on the departure platform produces instantaneous solutions for Enkhuizen. However, this approach is not always feasible, for example, as evidenced by the Amersfoort instance. Therefore, future research could explore a phased approach. Initially, a more constrained version of the problem could be considered for a brief period that is expected to produce better and faster results. Should this approach prove infeasible, the second phase will resolve the instance using the default model. The constraint version could vary. For instance, we could constrain the number of movements or the time at the departure platform. Additionally, a search strategy could be employed in the first phase; if it proves inefficient, it could be abandoned and proceed with the default model.

Time reduction

Imposing additional constraints was one strategy to minimize the search space, alternatively one could consider discretizing the time into units of several seconds. For instance, instead of allowing the solver to assign a specific action at every possible second, we consider as insignificant whether it performs a route one second earlier or later. To enforce this behavior, the time for all time interval variables and time-related events (e.g., route durations) could be divided by a factor, such as 6, before solving. This would constrain the solver to assign actions only every 6 seconds. In the obtained solution, all time-related events should again be multiplied by the reduction factor - 6.

This approach is expected to enhance performance for instances with a larger search space, particularly where train movements could potentially be planned at any time. However, careful consideration and fine-tuning of the reduction factor parameter are required, as a larger factor could easily result in an infeasible solution, while the advantages of a smaller factor may be lost.

Imposing hints

When domain insights are available, hints can be employed as a warm start, potentially accelerating computation time significantly. The advantage of using hints is that they are not hard constraints; even if a hint renders the instance infeasible or non-optimal, the solver will disregard the hint and still find a feasible or optimal solution. Specific to our case, we could utilize hints, for instance, to assign promising values for the number of movements for each train.

Other objectives

It is not always possible to guide the search toward better solutions with a search strategy. Therefore, it is worthwhile to explore the model's performance under various objective functions, as outlined in Section C.2. Additional objective functions, as mentioned and

evaluated by [Wattel, 2021, inline], could also be considered.

Furthermore, we could compare the solution quality and performance when imposing an objective analogous to the search strategies for the departure parking duration. As previously mentioned, we anticipate that the search strategy will be a faster option, while in both cases the same solutions' quality will be obtained.

Additional model improvement

[Marlière et al., 2023] proposes a custom constraint that decides based on the present route, the present status of all other sections for this movement and this train. Currently, in the proposed model, this connection is performed with binary constraint between the route and the section occupation variables, as illustrated below (repeated from Section 2.3.7.2):

$$pres(a_i^{t,r}) = pres(occ_{s_j}^{t,r}), \qquad \forall s_j \in S^r \tag{4.3}$$

We could alter it, as follows:

$$pres(a_i^{t,r}) = \bigwedge_{\forall s_j \in S^r} pres(occ_{s_j}^{t,r}) \bigwedge_{\forall s_j \in S^{r'}, r' \in R_i^t, r' \neq r} \neg pres(occ_{s_j}^{t,r'}), \tag{4.4}$$

The aforementioned constraint specifies that if a route r for a particular movement i is present, then only the occupations for the sections included in that route should be present, while all other section occupations for alternative routes in this movement and this train should be non-present.

The usefulness of this compact constraint may have already been shadowed by the heuristics of Google OR-Tools. Nevertheless, as claimed by the authors in [Marlière et al., 2023], this constraint strengthens the propagation for a similar problem, solved, however, with CP Optimizer.

Appendix A

TMP

A.1 Train Matching Problem (TMP)

In this section, we introduce the Train Matching Problem (TMP), but first, we begin with some relevant notation and train terminology. We refer to trains that pass the station and either stop or not as ***through trains***. In this problem, however, we are interested in the trains that start and end their service at the station.

The trains that reach their final destination need to be associated (also referred to as ***matched***) with another departure and this is the task of TMP.

The Train Matching Problem (TMP) was first specified by [Lentink, 2006]. The task of the matching problem is to find a feasible matching of arriving train units to departing train units while adhering to certain conditions and minimizing resources.

One of the main objectives of TMP is to keep the train units of the same train together as much as possible since this results in the minimum required resources [Lentink, 2006]. More splits will require more crew assistants for splitting and then combining the train units. Each split will result in two new separate trains, hence more drivers are necessary and the routing becomes harder. Resources can be crew assistants, shunt drivers, or even routing plans to the shunting yard/platform.

A matching of an arriving train unit to a departing train unit means that the same unit from the arriving train will be used in the departing train. The matching of an arriving train unit to a departing train unit is feasible if the following is satisfied:

- The time difference between the arrival and departure of the trains whose parts are matched is sufficiently large. This time can vary depending on the task that needs to be performed. For instance, time is needed to accommodate passenger out/in-boarding and routing from arrival to departure track. If the train requires maintenance checks and cleaning, it needs to be routed to the shunting track, and time for those tasks also has to be accounted for.
- The types of train units are the same.

It is important to note that a considerable part of the matching is fixed by the planning of the timetable. These are train units that arrive and are already assigned a departure. For instance, passing trains have usually insufficient dwell time on the platform to make bigger changes to the configuration of those trains. One could at most combine or split a train unit from the train.

As we have already mentioned, the main objective of TMP is to keep train units of arriving trains as much as possible together which will result in fewer splits and combines. Other possible objectives [Lentink, 2006] that could be combined with the main objective look into the average time a train has to stay at the station which could be either on the shunting yard or on the platform.

- For instance, one could try to minimize the average stay of the trains at the station which should potentially simplify the parking problem.
- Alternatively, we could maximize the average time the train stays at the station. This will prefer to match trains such that they either need to stay at the shunting yard for a short period or a long one.

A.1.1 Common notation for both problems

- $trackID$ - an integer representing the unique number of the track, Figure ...
- $sectionID$ - the ID of a section, also unique integer

Constants:

- $Types$ - all available subtypes of train units
- Tr - a set of all tracks at a specific station
- m_{ty} - maintenance time for each type of train unit $ty \in Types$
- ic_{ty} - internal cleaning time for each type of train unit $ty \in Types$
- ec_{ty} - external cleaning time for each type of train unit $ty \in Types$
- dw - mandatory dwell time for off-boarding and on-boarding of passengers
- $reach_{(tr_1,tr_2)}$ - boolean representing whether track tr_2 is reachable from tr_1

There could be multiple routes between two tracks, however, the following constants represent the shortest one in terms of time/distance or number of reversals.

- $tmove^{ty}_{(tr_1,tr_2)}$ - the shortest time required to move from one track tr_1 to the other tr_2, where $tr_1, tr_2 \in Tr$ and ty is the type of the train $\in Types$
- $dmove^{ty}_{(tr_1,tr_2)}$ - distance from track tr_1 to the other tr_2 based on the shortest route
- $rev^{ty}_{(tr_1,tr_2)}$ - min number of reversals required from track tr_1 to tr_2

A.1.2 Initial general matching model for NS

A.1.2.1 Input

- T_a - all arriving train units that have not been assigned to a departing train unit
- T_d - all departing train units

For each arriving train unit t with an id $IDtu_a$, we know the following information:

- $trainID_t$ - unique integer number representing the ID of the whole train that this train unit is part of
- $init_t$ - initial/arrival track (platform or a side track id), $init_t \in Tr$, where Tr is the set of all tracks' numbers
- $aside_t$ - side of the arrival track, $aside_t \in \{0,1\}$ (boolean) since the side of the arrival platform can only be A or B
- arr_t - arrival time of t at $init_t$ in seconds (minutes or a few seconds (1/10 of minute))
- $ty_t \in Types$
- t_{IDn} - the ID of the left neighboring of an arriving train unit
- t_m - whether the arriving train unit needs maintenance is needed, $t_m \in \{0,1\}$
- t_{ec} - whether the arriving train unit needs external cleaning is needed, $t_{ec} \in \{0,1\}$
- t_{ic} - whether the arriving train unit needs internal cleaning is needed, $t_{ic} \in \{0,1\}$

For each departing train unit t' with an id - $IDtu_d$, we know the following information:

- $trainID_d$ - unique integer number representing the ID of the whole train that this train unit is part of
- $des_{t'}$ - destination point (platform, side track) of the scheduled departure at dep_t time
- $dside_{t'}$ - side of the arrival track, $dside_t \in 0,1$ (boolean) since the side of the arrival platform can only be A or B
- $dep_{t'}$ - scheduled time for departure from destination des_t
- $ty_{t'}$, where $ty_{t'} \in Types$
- t'_{IDn} - the ID of the left neighboring of a departing train unit
- t'_m - whether the departing train unit requires maintenance, $t'_m \in \{0,1\}$
- t'_{ec} - whether the departing train unit requires external cleaning, $t'_{ec} \in \{0,1\}$
- t'_{ic} - whether the departing train unit requires internal cleaning, $t'_{ic} \in \{0,1\}$

A.1.2.2 Variables

Let us first introduce some relevant notation. We define DU_t to be the set that contains all departing train units (departing units) that depart after the arrival of t. The matched train units should also have the same type. Furthermore, there needs to be a way to reach the new departure track from the arrival track. Sometimes that could not be possible due to having maintenance that blocks certain tracks or any other type of obstacles.
The definition of DU_t can be expressed formally as follows:

$$\forall t' \in T_d, \forall t \in T_a : dep_{t'} \geq arr_t \wedge \\ ty_t = ty_{t'} \wedge reach_{(init_t,des_{t'})} \Leftrightarrow t' \in DU_t \quad (A.1)$$

We can now introduce an integer variable X_t for each arriving train unit t that will state departing train $t' \in DU_t$ to which t is matched to:

$$X_t \in DU_t, \; \forall t \in T_a \quad (A.2)$$

A.1.2.3 Constraints

- 1. We require each arriving unit of each arriving train, denoted as t, to be matched to exactly one departing train unit of a departing train. This constraint is satisfied by the way we have defined our variables.
  2. Each departing unit of each departing train has exactly one matched arriving unit from an arriving train.

     Furthermore, for the period we are looking at, the number of arriving train units of a certain type is equal to the number of departing train units of that type. Hence, if we enforce that each arriving train unit is matched to different departing train unit, the above constraint will be satisfied.

$$alldifferent(\bigcup_{t \in T_a} X_t...), \quad (A.3)$$

3. **Connectedness:**
   One should make sure that the arrival train unit that is matched to specific departure can reach the departing platform from the arrival platform. This is already included when creating the set of possible departure trains DU_t.

A.1.2.4 Objectives

- For each pair of matched train units t to $t' = X_t$ there is sufficient time between arrival and departure for the execution of all needed tasks (moving between platforms, on/offboarding, or cleaning and maintenance).

$$Time_t^{t'} = t_m \cdot m + t_{ic} \cdot ic + t_{ec} \cdot ec + 2 \cdot dw + t'_m \cdot m + t'_{ic} \cdot ic + t'_{ec} \cdot ec + tmove^{ty_t}_{(init_t, des_t)}) \quad (A.4)$$

  This equation finds the time for the service tasks by summing for each task the time needed for the task multiplied by whether the task needs to be executed.

  We can include it in the objective as follows:

$$IT_t = max((arr_t + Time_t^{t'} - dep_t), 0) \cdot p_s \quad (A.5)$$

  p_s is a fraction that represents the penalty for each insufficient minute. If the arrival time plus the time needed for services and moving is less than the departing time, then $(arr_t + Time_t^{t'} - dep_{t'})$ will be negative and no penalty will be incurred.

  This objective can be introduced as a constraint if we include it in the definition of DU_t where instead of $dep_{t'} \geq arr_t$, we use $dep_{t'} \geq arr_t + Time_t^{t'}$ and $Time_t^{t'}$ is calculated for each $t' \in DU_t$. The effect of this will be quite beneficial for the performance, the domain of X_t will decreased. The arriving train unit will be only matched to departing train units that depart sufficiently later to execute all required tasks.

  However, since the solution may become infeasible and in practice, it can happened that certain services are shorten or not executed if there is not enough time. Therefore, we opt to introduce $dep_{t'} \geq arr_t$ as a hard constraint, but $dep_{t'} \geq arr_t + Time_t^{t'}$ as a soft constraint.

- **Service requirement objective:**
  Match arriving and departing train units such that the service requirements of the departing train unit are covered by the arriving train unit.

  A penalty - p_{ms} for a service requirement of the departing unit that the arrival unit does not have. This penalty could be multiplied by the needed time for the extra services that the departing train needs. No penalty will be acquired if the arrival train unit has more service requirements than the departing train. Since it is more important that the services of the departing unit are covered by the matched unit, hence are a subset of the services of the arriving unit.

  We will denote the service cost that the objective function accumulates when matching arriving train unit t to departing train unit $t' = X_t$ by $S_t^{t'}$.

$$S_t^{t'} = p_{ms} \cdot (max(t'_m - t_m, 0) + max(t'_{ic} - t_{ic}, 0) + max(t'_{ec} - t_{ec}, 0) \quad (A.6)$$

  $(max(t'_m - t_m, 0)$ will be 1 iff the departing train unit has a maintenance requirement, but the arriving does not.

- **Match close trains objective:**
  Match arriving and departing train units that arrive and depart from platforms that can be reached from the same yard or are close enough. We will express this objective in terms of the number of reversals one needs to perform to get from the arrival track to the departing track. We do so, since reversals are most time consuming. Penalty - p_r is introduced for each reversal.

  Note, that one could also use either the time - $tmove^{ty_t}_{(init_t, des_t)}$ or the distance $dmove^{ty_t}_{(init_t, des_t)}$ between the tracks.

- **Minimize the number of splits and combines:**

  Introduce a penalty - p_{sc} for each split/combine. In sense, we count the number of splits and combines by looking at the left neighbor of each train unit. If the left neighbor of a train unit in the arriving train is different from the left neighbor of the same unit in the departing train then a penalty is accumulated, as follows:

$$(t_{IDn} \neq t'_{IDn}) \cdot p_{sc}, \text{ again } t' = X_t \quad (A.7)$$

  We next give two variants of the objective function:

A.1.2.5 Objective function

First variant:

Objective function:

$$min \sum_{t \in T}(S^t_{t'} + (t_{IDn} \neq t'_{IDn}) \cdot p_{sc} + rev^{ty_t}_{(init_t, des_{t'})} \cdot p_r + IT_t) \quad (A.8)$$

Second variant: It is only based on objective 2.) and introduce constraints for the other three objectives:

Objective function:

$$min \sum_{t \in T}((t_{IDn} \neq t'_{IDn}) \cdot p_{sc}), \text{ where } t' = X_t \quad (A.9)$$

Additional Constraints:

1. A constraint for the service requirements objective:

$$S^{t'}_t = 0, \forall t \in T_a \text{ and } t' = X_t \quad (A.10)$$

   This constraint states that if t is matched to t', then $S^{t'}_t = 0$, which essentially means that all service requirements are covered by the arriving train unit.

2. Introduce a constraint for the needed reversals:

$$rev^{ty_t}_{(init_t, des_{t'})} \leq 1, \forall t \in T \quad (A.11)$$

   We bound the reversals to 1, since in practice it is almost always the case that moving from one platform to another one contains at least one reversal.

3. Last for IT_t, the definition of DU_t will changed as follows:

$$dep_{t'} \geq arr_t + Time^{t'}_t \wedge ty_t = ty_{t'} \wedge reach_{(init_t, des_{t'})} \Leftrightarrow t' \in DU_t \quad (A.12)$$

The reason for introducing a second objective function is that we believe that the more constraint the solution space is the better will be the performance in CP.

A.1.2.6 Output

Each train unit of each arriving train is matched to train unit of a departing train. An arriving train unit with an $IDtu_a$ to departing train unit with an $IDtu_d$ (after this the departing unit stops to exists) ($IDtu_a$) - ($IDtu_d$)

A.1.2.7 Additional objectives

- For instance, one could try to minimize the average stay of the trains at the station which should potentially simplify the parking problem.
- Alternatively, we could maximize the average time the train stays at the station. This will prefer to matched trains such that they either need to stay at the shunting yard for a short period or for a long one.

A.2 CP model:

The simple model in MiniZinc does not take into account:

- service requirements
- connectedness
- numReversals

A.3 Solve TMP together with SRP

The combination of both models the TMP and SRP will enable more accurate and correct estimate of the number of required reversals/distance between the arrival train unit and a concrete considered departing unit.

Furthermore, I believe also the service time between arrival and departure can be seen whether it is indeed feasible to perform between the arrivals and departures.

In terms of the other aspects incorporated in the objective function in the TMP such as *matching services*, *#splits and combines*, SRP would not help.

Instead of $rev^{ty_t}_{(init_t, des_{t'})}$, we could include the actual number of of the required reversals for train t.

$$\sum_{i \in numr_t} \sum_{i \in R^t_i} numSub_r \cdot (a^{t,r}_i) = \text{ all reversals for train } t \quad (A.13)$$

A.4 Additional extension for deciding on where to perform a split or a combine

Until now the place where the split or the combines are executed was given prior to solving. However, this decision may not sometimes result in a worse solution and less optimal solution. For this reason, we introduce an extension in which the model will choose when to perform the splits and combines. In the following piece of text, we will explain how this could be modeled.

The main idea is to introduce (a) new train(s) when a split or combine is performed

Let us first consider only splitting an arriving train. The split should be performed prior to the last movement from the yard to the departing platforms. Hence, there are three places where the split can happen - either on the arrival platform, at the first yard, or at the second yard. The following constraints could be introduced for the number of routes of the composition (the unsplit train) - t and for the already new resulting from the split train t_1 and t_2.

$$numr_t \leq 2 \tag{A.14}$$

The overall number of movements is still preserved.

$$numr_{t_1} \leq maxr - numr_t \tag{A.15}$$

$$numr_{t_2} \leq maxr - numr_t \tag{A.16}$$

The initial place (track or yard) of t_1 and t_2 is the last place of the composed train t.

$$e_0^{t_1} = e_0^{t_2} = e_{numr_t}^t \tag{A.17}$$

$$e_0^t = init_t \tag{A.18}$$

$$e_{numr_{t_1}}^{t_1} = des_{t_1}, \ e_{numr_{t_2}}^{t_2} = des_{t_2} \tag{A.19}$$

Arrival/Creation times of those trains are given by the end of the last route of the composition. Furthermore, we do not introduce a stay variable for the composition t at the place where the split is performed, however we do introduce "stay variables for the new trains $stay_0^{t_1}$, $stay_0^{t_2}$.

$$arr_{t_1} = arr_{t_2} = e(A_{numr_t}^t) \tag{A.20}$$

$$s(stay_0^{t_1}) = s(stay_0^{t_1}) = arr_{t_1} \tag{A.21}$$

One should only note that enabling the solver to decide for a place where to split, restricts us to define a more narrow set of available routes. In particular, since we do not know where the train will be split, we do not know from where to where will be the first route. In particular, whether it will be between the arrival platform and yards or between yards and departure platform. In a sense, the route could be between platforms and the arrival platform and any of the gateways of the yards, but it could also be between any of the yards or even between any of the yards and the departure platform. We shall include all possible routes in each set $R_1^{t_1}$, $R_2^{t_1}$, $R_3^{t_1}$ for train t_1 and as well as for train t_2.

We briefly note the combining, which is analogous. Combining can be done either at the first yard, the second, or at the departing platform.

$$numr_t \leq maxr - max(numr_{t_1}, numr_{t_2}) \tag{A.22}$$

$$e_0^t = e_{numr_{t_1}}^{t_1} = e_{numr_{t_2}}^{t_2} \tag{A.23}$$

Here, $R_{1/2}^t$ consists of all routes between gateways of the yards and all routes between gateways and the departure platform.

Appendix B

Model with varying time between submovements

B.1 Additional variables

As we have already mentioned, a route can consist of a few separate subroutes. Between each of the subroutes, the train can wait at the last track (sections) of the previous subroute for some time. In this section, we extend Section 2.3.6 with a variable that will capture the parking between submovements:

- $wait_k^{t,r}, k \in \{1, numSub_r - 1\}$

Note that when the route consists of only one subroute, then k is between $1 \le k \le 0$, which is not possible. Hence, no "wait" variables are introduced.

The distinct routes have different numbers of subroutes, for instance, we can see in Figure B.1 that a_1^{t,r_1} and a_1^{t,r_3} consists of two subroutes, while a_1^{t,r_2} is one whole (this can be deduced by the wait variables in between the section occupations). A more detailed picture for the interval variable a_1^{t,r_3}, can be seen in Figure B.2, where the light orange boxes represent the section occupation intervals which in practice will overlap. The red represents the time interval of the first and only one wait variable.

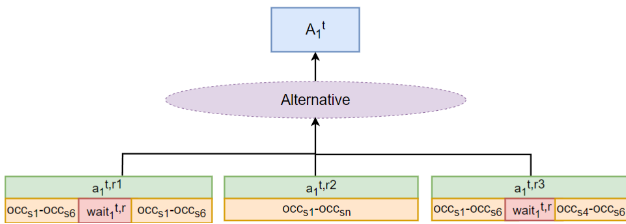


Figure B.1: Breakdown structure of the route variables (Extension of Figure 2.6 with waiting variables.

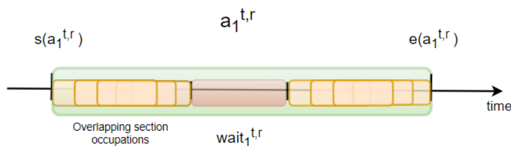


Figure B.2: Visualization of the time-span of the different time-interval variables (Extension of Figure 2.9 and wait variables.

B.2 Constraints

The following constraints are $\forall t \in T, 1 \le i \le maxM_t$ and $r \in R_i^t$ or simply R^t (all routes for all movements for train t) if i is not in the equation.

B.2.1 Constraining the start and end times for section occupations

To simplify the constraints for the start and end times for section occupations, we introduce an integer channeling variable - $startOfSubm$ for the start of each subroute (submovement) of each route $r \in R$ for the routes of all trains. The following code snippet 7 introduces required the constraints:

Algorithm 7 Constraints for the start of each submovement variables $startOfSubm_k^{r,t}$, where $r \in R_i^t, k \in \{1, numSub_r\}$

if $(k == 1)$ **then** ▷ The start of the first submovement coincides with the start of the route

  $startOfSubm_k^{r,t} = s(A_i^t)$

else ▷ The start of each other submovement is the start of the previous + the duration of the previous + the time for the intermediary stop

  $startOfSubm_k^{r,t} = startOfSubm_{k-1}^{r,t} + d_{r_{k-1}}^t + d(wait_k^{r,t})$

end if

The start times of the section occupations: We differentiate between the sections based on their position within the route and subroute in order to calculate the start times of their occupations. The sections of each subroute are divided into first sections $S_{first,k}^r$ and middle and last sections - $S_{last,k}^r$. The set of first sections $S_{first,k}^r$ contains the sections which the train is currently on at the beginning of subroute k in route r. The set of last sections - $S_{last,k}^r$ will be also the sections on which the train will stay once the subroute has finished and until the next subroute starts. The middle sections are all other sections.

The pseudocode 8 presents the logic of how sections' occupation start times will be inferred. We explain the code briefly. The first sections of the route, the sections that the train is currently on (sections in

$S^r_{first,1}$), should be occupied for the parking, but only if the route starts from outside a yard (so if we are at the arrival platform). Hence, their start times will be equal to subtracting the parking time from the start of the movement. In all other cases, the section occupations start times are equal to the start of the route if $s_j \in S^r_{first,k}$, otherwise relative to the start of the route with an offset of a given $start^t_{s_j,r}$.

Hence, the occupations of the intermediate stops are only handled once, by extending the end times of the section occupations. This will be the topic of the next paragraph.

Algorithm 8 Code snippet for the different constraints introduced in different situations for the start times of section occupations for a route $r \in R^t_i$, $s_j \in S^r_k, i \in \{1, maxM\}$

```
if k == 1 then                                   ▷ the first subroute
    if e_0^t == init_t then                  ▷ at a platform (not yard)
        if s_j ∈ S^r_{first,k} then              ▷ the first sections
            s(occ^{t,r}_{s_j}) = startOfSubm^{r,t}_k − d(stay^{r,t}_i)
        else              ▷ all other sections of the first subroute
            s(occ^{t,r}_{s_j}) = startOfSubm^{r,t}_k + start^t_{s_j,r}
        end if
    else ▷ first subroute, but not at arrival platform
        code snippet 9
    end if
else                               ▷ other subroutes of any route
    code snippet 9
end if
```

Algorithm 9

```
if s_j ∈ S^r_{first,k} then                      ▷ the first sections
    s(occ^{t,r}_{s_j}) = startOfSubm^{r,t}_k
else                  ▷ all other sections of the first subroute
    s(occ^{t,r}_{s_j}) = startOfSubm^{r,t}_k + start^t_{s_j,r}
end if
```

The end times of the section occupations:

The calculation of the end occupations times of the last sections of each subroute also differ. Depending on where the train goes (platform or a yard) and whether the last sections from the subroute are also the last sections of the route.

The pseudocode 10 summarizes the calculation of the end occupation times for all sections. We now briefly explain the code. If the route ends at a yard, the occupation times of the last sections are not extended with the time for the parking (line 4). However, if we are not going to a yard, then it must be a departure platform, we extend the occupations of the variables with $d(stay^t_i$. Last, if this is not the last subroute then we extend the end of the occupation variables with the $d(wait^{t,r}_k)$ to reserve for the time spent at an intermediate stop (line 9). In all other cases, we add the given offset ($end^t_{s_j,r}$) to the start of the subroute.

Finally, the duration of an occupation is completely determined by:

Algorithm 10 Code snippet for the different constraints introduced in different situations for the end times of section occupations, $\forall s_j \in S^r_k$, for $i \in \{1, maxM\}$ and $k \in \{1, numSub_r\}$

```
 1: if s_j ∈ S^r_{last,1} then                         ▷ the last sections
 2:     if k == numSub_r then  ▷ the last subroute of the r
 3:         if e_i^t ∈ Y then         ▷ if the endpoint of the route is a yard, we should not occupy the sections
 4:             e(occ^{t,r}_{s_j}) = startOfSubm^{r,t}_k + end^t_{s_j,r}
 5:         else  ▷ not a yard, extend the last sections to be occupied while parking
 6:             e(occ^{t,r}_{s_j}) = e(stay^t_i)
 7:         end if
 8:     else
 9:         e(occ^{t,r}_{s_j}) = startOfSubm^{r,t}_k + d^t_{r_k} + d(wait^{t,r}_k) = e(wait^{t,r}_k)   ▷ Here k is at most numSub_r - 1, not a last subroute, but last sections
10:     end if
11: else                                              ▷ Not last sections
12:     e(occ^{t,r}_{s_j}) = startOfSubm^{r,t}_k + end^t_{s_j,r}
13: end if
```

$$d(occ^{t,r}_{s_j}) = e(occ^{t,r}_{s_j}) - s(occ^{t,r}_{s_j}), \forall s_j \in S^r \quad \text{(B.1)}$$

On a final note, if a train is split or combined from/to a through train at the arrival/departure platform, then the section occupations constraints for the sections corresponding to the platforms (so sections in $S^r_{first,k}$) and $S^r_{last,1}$) will differ, but this will be discussed later.

B.2.2 Constraints for synchronization between routes and sections occupations and waiting variables

A route variable $a^{t,r}_i$ is present, meaning route r will taken for movement i if the chosenRoute variable is assigned to the index of this route within the set of routes for movement i.

$$pres(a^{t,r}_i) = (chosenRoute^t_i == indexOf(r, R^t_i)) \quad \text{(B.2)}$$

An optional variable for a route r and train t is present, meaning it is taken and $pres(a^{t,r}_i)$ is true, iff all occupation's section variables of that route are present as well. These are the sections from all subroutes of the route.

$$pres(a^{t,r}_i) = pres(occ^{t,r}_{s_j}), \quad \forall s_j \in S^r \quad \text{(B.3)}$$

Analogously, each "wait" variable - $wait^{t,r}_k$ for the time between subroutes of route r is present if and only if the variable for route r is present.

$$pres(a_i^{t,r}) = pres(wait_k^{t,r}), \quad \forall k \in \{1, numSub_r - 1\} \quad \text{(B.4)}$$

B.2.3 Constraints for determining section occupations

If the occupation of a section is present ($pres(occ_{s_j}^{t,r})$ is true), then based on the previous subsection B.2.1 the start and the end times of the occupation variables are constrained. All equations are compactly written in the pseudocode snippets 8 and 10. When a variable is present, then the solver will impose the constraints to be satisfied, otherwise, no.

Pseudocode 8 and 10. (B.5)

B.2.4 Constraints for start and end times of "wait" variables

The duration of the "wait" will usually be at least some time, given by the parameter $minWait$. Furthermore, to ensure that the driver is not in the middle of the route for an incredibly long time, the duration is also bounded by $maxWait$.

$$maxWait \geq d(wait_k^{t,r}) \geq minWait, \quad \forall k \in \{1, numSub_r - 1\} \quad \text{(B.6)}$$

The start time of the intermediate parking between routes is the time when the train finishes subroute r, where $d_{r_1}^t$ is the respective duration of this subroute.

$$s(wait_1^{t,r}) = s(a_i^{t,r}) + d_{r_1}^t \quad \text{(B.7)}$$

$$s(wait_k^{t,r}) = s(wait_{k-1}^{t,r}) + d_{r_{k-1}}^t \quad \text{(B.8)}$$

The following constraints can be seen in the main documentation of Google-OR-Tools. There is not a lot of explanation there, so one could refer to this website. The end of parking $wait_k^{t,r}$ should be before the start of the next parking/staying.

$$EndBeforeStart(wait_k^{t,r}, wait_{k+1}^{t,r}) \quad k \in \{1, numSub_r - 1\}, r \in R^t \quad \text{(B.9)}$$

The route should start before the start of the first wait variable. Actually, before the start of any "wait" variable.

$$StartBeforeStart(a_i^{t,r}, wait_1^{t,r}) \text{ if } numSub_r \geq 2 \quad \text{(B.10)}$$

A section can be occupied by at most one train at a given time. This constraint ensures that all occupation intervals of each section are non-overlapping. We also add the constant intervals from each of the predetermined events pe from the set of all fixed events FE - movements, parking, etc.

$$noOverlap(\bigcup_{t \in T, r \in R_t} occ_s^{t,r} \bigcup_{pe \in FE} occ_s^{pe}), \forall s \in S \quad \text{(B.11)}$$

The remaining constraints are identical to those described in Section 2.3.

Appendix C

Other extensions & Objective functions

C.1 Other extensions

C.1.1 Additional parameters

- m - maintenance time for each type of train unit. We opt to use one parameter for all types of trains to simplify notation.
- ic - internal cleaning time for each type of train unit
- ec - external cleaning time for each type of train unit Furthermore, for each yard $y \in Y$ we know:
  - c_y - the capacity of y (the length of all tracks)
  - Ser_y - a set of all services that can be performed at yard y

Additional known information for each train ($t \in T$)

- len_t - the length of the whole train, this is the sum of the length of all train units in the train
- t_m - whether the train part/unit needs maintenance, $t_m \in \{0, 1\}$
- t_{ec} - whether the train part/unit needs external cleaning, $t_{ec} \in \{0, 1\}$
- t_{ic} - whether the train part/unit needs internal cleaning, $t_{ic} \in \{0, 1\}$

C.1.2 Yard extension

This extension lifts the assumption for the infinite capacity of the yards (Section 2.3.3).

We have already defined the most important notation (Section 2.3.4) that will be used for extending the model. To remind the reader Y was the set of all yards at the station, c_y and G_y the capacity and the set of gateways of $y \in Y$, respectively.

The main task of this extension is to ensure that for each yard at the station, the yard capacity is *NOT* exceeded at any point. To model this, we will use the concept of **cumul functions**.

A **cumul function** illustrates the amount of particular resources over time. It is an expression built as the algebraic sum of the elementary cumul functions or their negations $f = \sum_i \epsilon_i \cdot f_i$, where $\epsilon_i \in \{-1, 1\}$ and f_i is the **elementary cumul function** which illustrates the contribution (an increase or decrease) of an (interval) variable to the resource. We will present only one type of elementary cumul function - $pulse(a, h)$, where h is the amount with which the resource increases during the interval of a. Figure C.1 presents the function graphically.

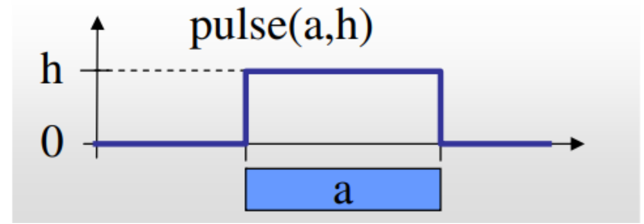


Figure C.1: Illustrates the contribution of the elementary function - *pulse* to the resource. The resource increases with the amount h during the interval of a.

We can now define the cumul function f_y for each yard y as follows:

$$f_y = \sum_{st \in ST} \sum_{i \in \{0,1\}} pulse(stay_i^{st}, len_{st}) \cdot bool(e_i^t == y),$$
$$\text{where } st \text{ is a shunt train and } ST \text{ is the set of all shunt trains} \quad \text{(C.1)}$$

Each $stay_0^t$ and $stay_1^t$ if they are present, correspond to parking at the first or the second yard for all shunt trains.

The yard capacity is constrained as follows: The yard is at most c_y occupied at a given point in time.

$$f_y \leq c_y, \forall y \in Y \quad \text{(C.2)}$$

C.1.3 Service extension

This section extends the model with two important aspects of service management. First, we want to shunt trains to yards where they can execute the services they demand, as not all services are possible at each yard. Next, we should ensure that the train stays for enough time in the yard to execute all services.

We define $SerTime^t$ as the sum of the time for all services that t (shunt train) requires, hence

$SerTime^t = t_m \cdot m + t_i c \cdot ic + t_e c \cdot ec$. We have assumed that the difference between the arrival and departure time is more than the required service time for each train 2.3.3, otherwise the input will not be feasible.

In the current model, services can be handled at one or two yards. Unfortunately, fixing the number of visited yards for all trains before solving, will result in a further decrease of the optimality of the solution. Nevertheless, there are certain cases where we can fix the number of yards to two in case we insist all service needs to be executed. For instance, if no yard at the station can handle all services of train t, then $numr_t = 3$, or if the set of services is non-empty then $numr_t \geq 2$ (shunting to yard is required).

We now present the constraints required to model the aspects, mentioned at the beginning of the section, namely for ensuring that there is enough time at the yards and that it is also possible to execute all services. We also define Srv_t - the set of all services required by t. Depending on the number of visited yards, we have:

- **One Yard:** Hence, $numr_t = 2$.

$$numr_t == 2 \Rightarrow d(stay_1^t) \geq SerTime^t \quad (C.3)$$

$$Srv_t \subset Ser_{e_1^t} \quad (C.4)$$

- **Two Yards:**

$$numr_t == 3 \Rightarrow d(stay_1^t) + d(stay_2^t) \geq SerTime^t \quad (C.5)$$

$$Srv_t \subset Ser_{e_1^t} + Ser_{e_2^t} \quad (C.6)$$

$$pres(stay_i^t) \Rightarrow d(stay_i^t) \geq minServiceDuration, \quad \forall i \in 1,2 \quad (C.7)$$

C.2 Examples of objective functions

This section will give a few variants and examples of the objective function. Nevertheless, those are not part of the experiments.

C.2.1 Minimize total number of shunt movements

The total number of shunt movements can be based either on the number of all movements taken by all trains in T (as in C.8) or on the number of the subroutes contained in the routes for each movement (as in C.9). Instead of C.9, C.10 can be utilized as it eliminates the summation over a variable, which is generally preferred.

$$min \sum_{t \in T} numr_t \quad (C.8)$$

$$min \sum_{t \in T} \sum_{i \in numr_t} \sum_{r \in R_i^t} numSub_r^t \quad (C.9)$$

$$min \sum_{t \in T} \sum_{i \in maxM} \sum_{r \in R_i^t} numSub_r^t \cdot pres(a_i^{t,r}) \quad (C.10)$$

Additionally, to count the number of reversals, objective C.9 could be employed. The only modification required is to replace $numSub_r^t$ with $numSub_r^t - 1$, as a route with two subroutes involves one reversal.

C.2.2 Minimize total routing time

The total routing time is obtained by summing the duration of all present route variables.

$$min \sum_{t \in T} \sum_{i \in maxM} \sum_{r \in R_i^t} d(a_i^{t,r}) \quad (C.11)$$

We use $maxM$ instead of $numr_t$ as $numr_t$ is a variable, while $maxM$ is a parameter. This allows the objective function to be fully defined before assigning $numr_t$ for each train $t \in T$.

C.2.3 Minimize total distance

The total distance is obtained by summing the length of each present route.

$$min \sum_{t \in T} \sum_{i \in maxM} \sum_{r \in R_i^t} dis_r \cdot pres(a_i^{r,t}) \quad (C.12)$$

Appendix D

Additional Figures

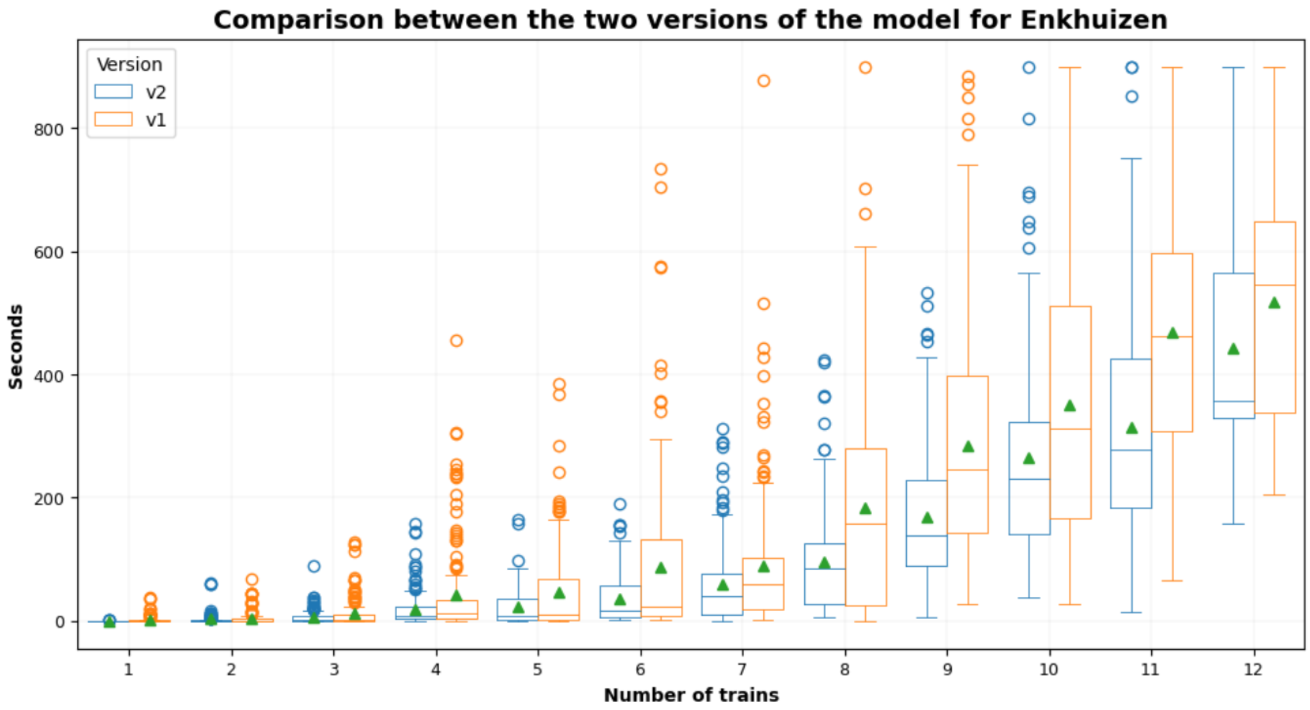


Figure D.1: Comparison between the two model versions for Enkhuizen instances. Each bar shows the average running time, averaged between all instances with a specific number of trains and all seeds. Time is in seconds.

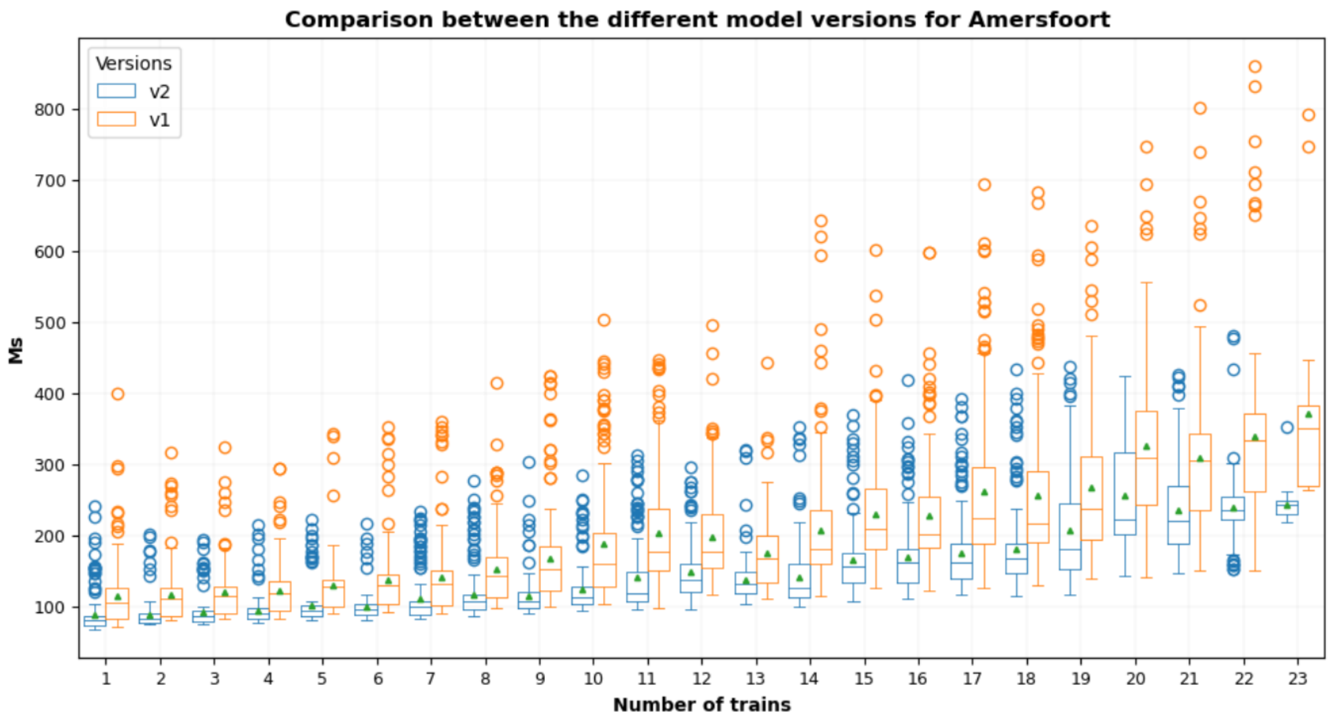


Figure D.2: Comparison between the two model versions for Amersfoort instances presented with boxplots. Again, time is in order of **milliseconds**.

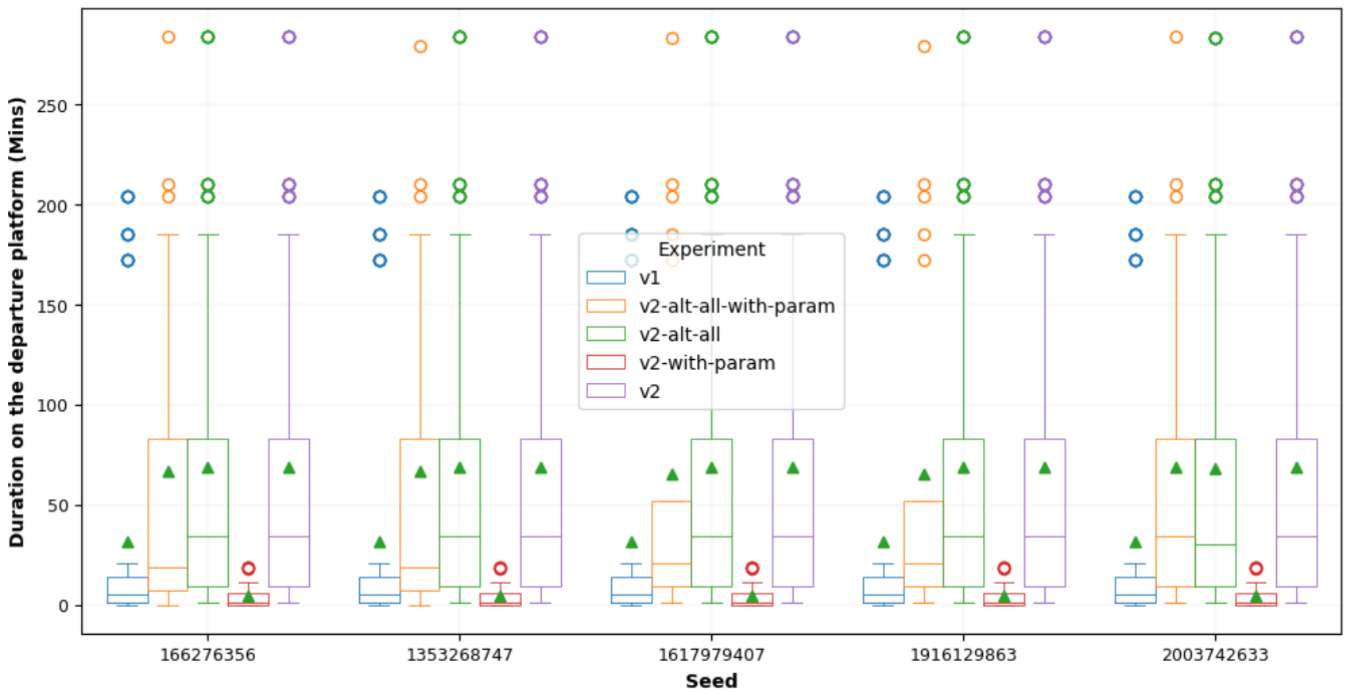


Figure D.3: The figure presents the additional executions (v2-with-param and v2-alt-all-with-param), produced locally with the parameter for keeping all feasible solutions set to true for the second version and the second version with alternative routes. These are compared with v1, v2, and v2 with all alternative routes.

Bibliography

[chu, 2010] (2010). https://github.com/chuffed/chuffed?tab=readme-ov-file.

[sim, 2020] (2020). https://www.pnw.edu/wp-content/uploads/2020/03/attendance5-1.pdf.

[Broek, 2016] Broek, R. v. d. (2016). Train shunting and service scheduling: an integrated local search approach.

[Cappart and Schaus, 2017] Cappart, Q. and Schaus, P. (2017). Rescheduling railway traffic on real time situations using time-interval variables. In Salvagnin, D. and Lombardi, M., editors, *Integration of AI and OR Techniques in Constraint Programming*, pages 312–327, Cham. Springer International Publishing.

[Delahaye et al., 2019] Delahaye, D., Chaimatanan, S., and Mongeau, M. (2019). Simulated annealing: From basics to applications. In Gendreau, M. and Potvin, J.-Y., editors, *Handbook of Metaheuristics*, volume 272 of *International Series in Operations Research & Management Science (ISOR)*, pages 1–35.ISBN 978–3–319–91085–7. Springer.

[Edelkamp and Schrödl, 2012] Edelkamp, S. and Schrödl, S. (2012). Chapter 13 - constraint search. In Edelkamp, S. and Schrödl, S., editors, *Heuristic Search*, pages 571–631. Morgan Kaufmann, San Francisco.

[Haahr et al., 2017] Haahr, J. T., Lusby, R. M., and Wagenaar, J. C. (2017). Optimization methods for the train unit shunting problem. *European Journal of Operational Research*, 262(3):981–995.

[Hendrikse, 2021] Hendrikse, K. (2021). Branch-and-cut-and-price to solve a relaxation of the train unit shunting and service problem. Master's thesis.

[Kamenga et al., 2021] Kamenga, F., Pellegrini, P., Rodriguez, J., and Merabet, B. (2021). Solution algorithms for the generalized train unit shunting problem. *EURO Journal on Transportation and Logistics*, 10:100042.

[Kamenga et al., 2019] Kamenga, F., Pellegrini, P., Rodriguez, J., Merabet, B., and Houzel, B. (2019). Train unit shunting: Integrating rolling stock maintenance and capacity management in passenger railway stations. In *8th International Conference on Railway Operations Modelling and Analysis (RailNorrköping 2019).*

[Kroon et al., 1997] Kroon, L. G., Edwin Romeijn, H., and Zwaneveld, P. J. (1997). Routing trains through railway stations: complexity issues. *European Journal of Operational Research*, 98(3):485–498.

[Lentink, 2006] Lentink, R. (2006). *Algorithmic Decision Support for Shunt Planning.* PhD thesis, E.

[Marlière et al., 2023] Marlière, G., Sobieraj Richard, S., Pellegrini, P., and Rodriguez, J. (2023). A conditional time-intervals formulation of the real-time railway traffic management problem. *Control Engineering Practice*, 133:105430.

[Peer et al., 2018] Peer, E., Menkovski, V., Zhang, Y., and Lee, W.-J. (2018). Shunting trains with deep reinforcement learning. In *2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 3063–3068.

[Philippe Laborie, 2018] Philippe Laborie, Jérôme Rogerie, P. S. . P. V. (2018). Ibm ilog cp optimizer for scheduling. *SpringerLink.*

[Richard Freling, 2005] Richard Freling, Ramon M. Lentink, L. G. K. D. H. (2005). Shunting of passenger train units in a railway station. *Transportation Science 39(2):261-272.*

[Rodriguez, 2007] Rodriguez, J. (2007). A constraint programming model for real-time train scheduling at junctions. *Transportation Research Part B: Methodological*, 41(2):231–245. Advanced Modelling of Train Operations in Stations and Networks.

[Rossi et al., 2008] Rossi, F., van Beek, P., and Walsh, T. (2008). Chapter 4 constraint programming. In van Harmelen, F., Lifschitz, V., and Porter, B., editors, *Handbook of Knowledge Representation*, volume 3 of *Foundations of Artificial Intelligence*, pages 181–211. Elsevier.

[Stuckey, 2010] Stuckey, P. J. (2010). Lazy clause generation: Combining the power of sat and cp (and mip?) solving. In Lodi, A., Milano, M., and Toth, P., editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 5–9, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Wattel, 2021] Wattel, N. (2021). Routing of shunt trains at logistics hubs of ns using constraint programming.

[Winter and Zimmermann, 2000] Winter, T. and Zimmermann, U. T. (2000). Real-time dispatch of trams in storage yards. *SpringerLink*, 96:287–315.