Universiteit Utrecht

Department of Information and Computing Sciences

Planning Drivers for Shunting Yards

Masters Thesis

Author:
Luuk A. van Nes
Utrecht University

First Supervisor:
dr. J.A. Hoogeveen

Second Supervisor:
dr. ir. J.M. van den Akker

Supervisor NS:
Roel W. van den Broek

December 2023

Abstract

Throughout the day, trains are parked on shunting yards operated by the NS. While at the shunting yards, trains undergo many jobs such as riding, combining and splitting. These jobs then need to be executed by drivers for which a planning needs to be made. These tasks come with strict starting times or flexible time windows, considering factors like preferred starting times. The necessity for drivers to travel between jobs is addressed by allowing walking or riding along with a train, introducing considerations for minimizing travel time. Some particular pairs of jobs require synchronization and need to be planned to start at the same time. Additionally, buffers are preferred to mitigate delays, particularly for consecutively performed tasks that do not involve the same train. The NS's current scheduling method, involving heuristics and an Integer Linear Program (ILP), has become too slow. To tackle these issues, we propose a crew scheduling solution using a branch-and-price algorithm that branches on time windows. The branch-and-price algorithm uses Dynamic Programming in a novel way to solve the pricing problem. The proposed algorithm is tested on real-life instances from NS's shunting yards, solving smaller instances faster while finding a better solution. However, for larger instances, the current scheduling method shows better results.

Contents

1 Introduction

The Nederlandse Spoorwegen (NS) is by far the largest passenger railway operator in the Netherlands, serving over one million passengers daily. Throughout the day, the trains commonly used to transport these passengers are parked in shunting yards whenever they are not in use. However, these shunting yards are not simply parking lots. The NS needs to be as efficient as possible and uses this downtime for trains to perform cleaning, repairs, and other preparatory tasks for their departure. Consequently, the shunting yard is very active, with many trains driving around a relatively small and crowded site, especially outside of rush hours. A recent study by Van den Broek et al. (2022) has developed a method (HIP) to schedule these train operations. A large number of these operations need to be performed by a train driver, necessitating a distinct method of assigning these operations, from now on called *jobs*, to create efficient schedules that adhere to all regulations and other constraints.

The current in-house method uses heuristics to limit the possible times jobs can start and then uses an ILP to assign the jobs to drivers. This method has become too inefficient due to several reasons. First, NS is struggling to attract enough drivers in the current economy. Therefore, using the available drivers as efficiently as possible is even more important. Second, the number of train operations continues to increase, placing further strain on the system. Third, while taking drivers into consideration, HIP is focused on optimizing the train schedules instead of the drivers' schedules, which further complicates the driver scheduling process. Fourth, the regulations require drivers to perform an increasing number of safety-related jobs. Finally, the NS would like to add constraints to the problem it is currently unable to handle. These challenges make it critical to develop an efficient method for driver scheduling that considers the availability of drivers, constraints, and increasing workload while also being adaptable to future constraints.

In this thesis, we try to solve the problem of assigning jobs to drivers, with some freedom in how we plan the jobs. For example, it is possible to not plan every job and some jobs have time windows in which they can be planned. This boils down to a crew scheduling problem with complex constraints that gives it many similarities to vehicle routing problems with time windows and synchronization (VRPWSyn). Our primary objective is to schedule as many jobs as possible, with a higher priority for jobs that involve driving the trains from one place to another. Additionally, there are also less important preferences that primarily aim to improve driver satisfaction and robustness.

Every problem instance has a group of drivers with their own (possibly unique) working hours. For example, there could be a group with two drivers working a morning shift and one working an afternoon shift. Moreover, each driver starts their shift at a predetermined location. By labour regulations, workers are prohibited from working more than four consecutive hours before requiring a break. This means that some drivers need a meal break if their shift is longer than four hours, preferably scheduled midway through their shift.

The drivers have to perform several different types of jobs, such as driving the train, combining or splitting train units, and preparing the train for departure. To avoid trains using the same tracks simultaneously, most jobs have a strict schedule that tells them the exact time it needs to start. On the other hand, some jobs, such as combining train units, can happen in place; this allows for a bit more flexibility as it won't interrupt other trains. Therefore, these more flexible jobs have time windows in which they need to be started and completed. The time windows consist of the earliest time after which the job can be started and the latest time at which it has to be finished. In addition, these more flexible jobs also have a preferred time for when they should be started, as these jobs are generally being done in preparation for a train departing, which can be done far in advance of the departure but is preferred to be done right before departure.

Sticking to the train shunting schedule, the trains operated on in the jobs are assigned to move between and park on several different tracks. Therefore, a driver might need to do jobs in different locations. In this case, it may be necessary for drivers to travel between two jobs. The easiest way to do this is by walking between the locations on foot. However, as sometimes trains are riding between two locations anyway, riding along with a train might allow a driver to be on time for a job they would not otherwise be on time for. Therefore, we allow drivers to ride along with jobs where a train moves from one location to another. Riding along with a train does not count as completing the job of driving that train. Due to the system's current limitations, we only allow one driver to ride along per job.

Generally, drivers do not like to travel a lot during their day. To avoid them travelling too much, we try to minimize the distance the drivers have to walk and ride along. The manual planners operating this technology can choose to assign jobs to drivers beforehand. These jobs are fixed jobs (whereas other jobs are free jobs), and can only be assigned to one specific driver and need to be assigned to that driver in the final schedule. Furthermore, the time window of a fixed job is equivalent to its duration, meaning that these jobs can only be scheduled to start at a single predefined time point.

There are also cases where an operation on a train needs to be handled by two drivers. This results in a pair of jobs that need to be synchronized, with one job serving as the main job and one as the assisting job. As the name implies, the main job is more important, and it is feasible to schedule the main job without scheduling the assisting job; however, scheduling the assisting job without scheduling the main job is not possible. If both jobs are scheduled, they must be scheduled equidistantly from the beginning of their respective time windows. For example, if the main job has a time window beginning at time 1 and the assisting job's time window starts at time 4, the assisting job must be scheduled three minutes later than the main job. In rare cases, the first job can be finished before the second job starts. In such cases, it is possible for one driver to do both jobs.

Consecutively doing jobs on the same train if there is little downtime between them is convenient for the drivers. Therefore, to increase the driver's

satisfaction, there are specific pairs of jobs for which we explicitly prefer to line them up in the duty of a single driver. We call it the preferred succession for two jobs that we want to line up this way. To ensure the schedule is affected less by delays, we also prefer a buffer between jobs done consecutively by a single driver that are not performed on the same train. This ensures that if a driver takes too long for a particular job, it won't affect future jobs as much. This buffer is not preferred for successive jobs on the same train, as it would be strange to have a driver waiting to do a second job on the train he has already worked on.

This thesis contributes in two ways. We solve this problem using branch and price with a novel way to solve the pricing problem. On top of that, we are able to test this algorithm on real-life instances provided by the NS, providing valuable information on the performance on practical problems.

The thesis is organized as follows. In Section 2, we describe the relevant literature. In Section 3, we present several ways to describe the problem as an ILP and expand on the current method currently used at the NS. In Section 4, we present the algorithm we created to solve the problem. In Section 5, we show the results of how well our algorithm performed and compare it to the currently used method. Finally, we provide a conclusion and a direction for future work in Section 6.

2 Related Literature

This chapter gives an overview of which papers are relevant to this problem. These papers mostly come from fields about crew scheduling and vehicle routing, with which the problem of this paper shares many similarities. We first mention each of these fields and give a survey paper for each field. We then highlight the most common methods for solving these types of problems and showcase the important papers that implement these methods.

2.1 Crew Scheduling

Numerous studies have been conducted on crew scheduling, initially focusing on airport crew scheduling. Still, more recently, research efforts have also been directed towards addressing similar problems in the railway industry. These problems are particularly relevant due to the constraints imposed by working conditions and employee satisfaction preferences. The constraints typically include limitations on working hours and meal breaks. In contrast, employee satisfaction preferences may include, among other things, the desire for jobs with high diversity where they can do many different things. The thesis written by Gottenbos (2022) dives deeper into these types of employee satisfaction preferences.

In general, the papers that address these problems typically describe them in a similar manner. Specifically, a set of employees are required to perform a given set of jobs, which, similar to our problem, often involve a combination of

duration, time window, and required skills. The allocation of jobs to an employee in a specific order results in the creation of a *duty*. Mathematically, these problems are usually formulated as a set covering problem, where predetermined duties are selected and assigned to the employees to cover all jobs. The paper by Ernst et al. (2004) presents an overview of crew scheduling. It provides a large number of relevant papers. While this paper is quite old, it still explains some of the more common concepts present in many crew scheduling problems. More recent crew scheduling surveys for train and airline crew scheduling are listed below.

Train Crew Scheduling

Extensive research has been conducted in railway operations, focusing on developing efficient schedules for rolling stock to minimize the associated costs of maintenance and material expenses. However, a significant portion of the total costs incurred also relates to scheduling the crew members required to operate the trains. These problems are more complex than merely assigning one crew to each train line due to regulations concerning the working conditions of crew members. For instance, crew members may require a meal break during their shifts, necessitating the involvement of a different crew to operate the train they were previously operating. The initial crew must be assigned to another train after their meal break to create an efficient schedule.

The paper by Heil et al. (2020) presents a literature survey on all railway crew scheduling studies conducted between 2000 and 2020. The paper offers valuable insights into the challenges encountered in scheduling railway crews and the methodologies developed to address these issues.

Airline Crew Scheduling

Labour costs represent airlines' second largest expense, presenting a significant opportunity for cost reduction. Since the late 1950s, considerable research efforts have been dedicated to minimizing this cost, with a primary focus on optimizing the crew schedules. In contrast to railway operations, the majority of research studies have centered on optimizing crew schedules. The paper authored by Kasirzadeh et al. (2017) provides an overview of the research conducted in this area and attempts to solve some general instances using standard algorithms.

2.2 VRP and VSP

The Vehicle Routing Problem (VRP) and the Vehicle Scheduling Problem (VSP) bear significant relevance to our current problem. In VRP, a set of *customers* must be visited by *vehicles*. These visits then add up to become *routes*, which are then allocated to the vehicles to obtain the solution. In VSP, the vehicles must perform trips, i.e., driving the vehicle from one point to another at a given time. In the solution, these trips are similarly lined up to become routes. Our problem is a mixture of these two problems: some jobs can be viewed as visiting

customers (performing an operation on a train at a singular location), and some jobs can be viewed as performing trips (driving the train from one location to another). Although the practical application of VRP and VSP aligns closely with crew scheduling, the notation and terminology used for these problems typically differ. Some extensions present in VRP and VSP, which are not commonly found in crew scheduling, closely reflect our problem. These extensions include Time Windows and Synchronization, both of which are also relevant to our problem, as certain jobs possess time windows, and others require the involvement of two drivers.

Home healthcare is an especially interesting field in which vehicle routing problems are common. Efficient scheduling of staff is crucial in ensuring proper patient care, particularly given the current challenges in attracting an adequate number of caregivers. Generally, finding these schedules is modelled as a vehicle routing problem. Unlike crew scheduling in the airline and railway industries, research in staff scheduling for healthcare services often omits significant constraints related to labour laws. However, due to patients' needs to receive care from multiple caregivers, synchronization is a frequently studied constraint in this field. Moreover, patients are usually only given an indication of a period when caregivers will come by. Therefore, time windows present a significant issue as well. As such, research in this area is highly relevant to our current problem. The paper authored by Haitam et al. (2021) provides a survey of research related to the vehicle routing problem in home health care services and is particularly relevant to our problem.

2.3 Solution Methods

In the field of optimization, various techniques have been developed and applied to solve problems related to crew scheduling and vehicle routing. The selection of these techniques often depends on the specific problem's characteristics and the research's vintage. Four approaches, namely Exact, Heuristic, Column Generation, and Meta-Heuristics, are commonly used to solve the previously mentioned optimization problems. While Column Generation can be considered a technique that potentially falls under both the Exact and Heuristic categories, it is used frequently enough to warrant its review. On top of that, it has some characteristics that make it distinct from most other exact and heuristic methods.

For crew scheduling problems, column generation is identified as the most commonly used technique, owing to its ability to solve large-scale problems efficiently. Additionally, meta-heuristic methods such as local search are frequently employed, although to a lesser extent than column generation. While heuristics are used slightly less than meta-heuristics, they still represent a significant proportion of approaches employed in crew scheduling. Exact solutions were once widely used in solving crew scheduling problems. However, with the increase in problem size, exact solutions have become less popular as they are no longer practical for most real-world instances.

In the case of vehicle routing problems, heuristic and exact solutions are used with the same frequency as in crew scheduling. However, column generation is not as popular in vehicle routing as in crew scheduling, and instead, metaheuristic methods are usually preferred. This is due to the ability of metaheuristic methods to handle the complexity of real-world instances of vehicle routing problems.

Each of the approaches above will be elaborated upon below, linking several important papers for both types of problems.

Exact Approaches

Regarding railway crew scheduling problems, exact ILP models are typically only employed for very small instances. The most common approach is to use a set covering problem (SCP) formulation, which involves calculating all possible duties and selecting a combination that satisfies all constraints. Even the largest of these problems only involves around 60,000 possible duties. Airline crew scheduling is solved similarly using a set covering approach. Boschetti et al. (2004) use an SCP formulation to solve the Multiple Depot Crew Scheduling Problem, where crew members can start at different depots and have to return to their starting depot. Their solution can be used for both airline and railway crew scheduling problems. Although there are exact solutions available for Vehicle Routing Problems (VRP), they tend only to be useful for very basic VRP instances and are not particularly relevant to this paper.

Heuristic Approaches

In crew scheduling, heuristics are frequently employed. Most of these heuristic solutions involve efficiently identifying a promising set of duties and solving the ILP with that specific, smaller set of duties. For instance, Chen et al. (2013) found duties based on promising relief locations for meal breaks. Van den Broek et al. (2020) addressed the problem of scheduling drivers on a shunting yard, where drivers can ride along with any train activity. They used two different heuristics to determine a partial order of driver activities and then used dynamic programming to identify the exact times for each activity. Finally, they presented a branch and price algorithm to compare the two heuristics.

Column Generation based algorithms

Column generation is a commonly employed approach in crew scheduling to solve problems that can be defined as a set covering problem. The columns in such problems represent duties that are usually independent. For example, Zamorano and Stolletz (2017) used column generation to create schedules for technicians, some of whom must work in pairs to complete specific jobs. However, unlike our problem, the technicians in their study were already paired up at the start of the day and remained together throughout. Verhave (2015) also

used column generation to solve a regular crew scheduling problem with schedules as columns that involved different skills, scheduling for an entire week at once, and the possibility of overnight stays.

While ideally, the columns are not connected, in our problem, the columns (driver schedules) are connected by the synchronization constraints. Bredstrom and Ronnqvist (2007) use column generation in their algorithm to solve a problem in which the columns are similarly connected. They solve a vehicle routing and scheduling problem with synchronization constraints. To work around the problem of connected columns, they start by relaxing the synchronization constraints. To ensure the synchronization constraints are met in their final solution, they branch on time windows until none of the supposedly synchronized jobs have a different starting time. Dohn et al. (2011) tackled a similar vehicle routing problem with time windows and temporal dependencies using two other integer linear programming (ILP) models. The first ILP model was a mixed-integer model that defined two variables for the order of the vehicles and the times at which locations are visited. The second model was a time-indexed model that used a graph with a node for each combination of vehicle, time, and location, with edges between them representing the trips. In both cases, the columns they must find are feasible vehicle routes. To find the best routes, they have to solve the pricing problem, which they formulated as a shortest path problem with time windows and capacity constraints. Their strategy also involved branching on time windows, attempting to remove as many options as possible. Building upon these concepts, Rasmussen et al. (2012) developed a novel approach to solve the home care crew scheduling problem (HCCSP) with synchronization and time windows. They also used branch and price to solve the problem, resulting in a similar formulation using caregiver schedules as columns. In addition to branching on time windows, they branched on assigning visits to specific caregivers. This approach resulted in an efficient and effective solution to the HCCSP, with potential applications in a variety of related fields.

Li et al. (2020) addressed a complex split delivery system problem involving multiple different time windows for a single job and synchronization. Their type of synchronization was based on selecting the same time windows for delivery, and they utilized a branch and price algorithm to solve the problem. Their branching rule consists of selecting a particular time window in one branch and removing it as an option in the other branches. Van den Akker et al. (2012) developed a solution for a parallel machine scheduling problem, which included release dates, deadlines, and generalized precedence constraints, using column generation. This method resulted in a master and pricing problem that closely resembles other studies aimed at solving vehicle routing problems with temporal dependencies and time windows. Unlike the previously mentioned studies, they did include the synchronization constraints in the master problem. Furthermore, in his thesis, Van den Broek (2022) utilized a branch and price algorithm to address the scheduling of drivers in the shunting yard. Although the problem lacked synchronization between routes, Van den Broek presented an interesting approach in which the pricing problem calculated only the partial order of jobs

in a duty, while the master problem determined the exact completion times of jobs.

Meta-Heuristics

In crew scheduling problems and vehicle routing problems, solutions are typically represented as an order of jobs or trips per employee or vehicle. This representation allows for simple neighbourhood operators such as move, swap, insert, and remove, which can be efficiently implemented. Therefore, different Meta-Heuristics, such as local search and genetic algorithms, are commonly used methods to solve these problems. Bräysy and Gendreau (2005) provided an overview of the local search methods available at that time to solve the vehicle routing problem with time windows (VRPTW). They detailed the neighbourhood operators that are commonly employed and discussed the evaluation of algorithms for the VRPTW. Another example of a successful application of optimization algorithms in crew scheduling is the work by Hanafi and Kozan (2014), who employed simulated annealing to solve the problem of scheduling railway crews with meal breaks. The main contribution of their work was the introduction of flexible meal break timing at a relief point, as opposed to the more common approach of scheduling meal breaks at the start of a relief point. Their simulated annealing uses fairly simple neighbourhood operators to solve the problem. Shen et al. (2013) applied a genetic algorithm to solve the crew scheduling problem for public transport. Their algorithm uses an adaptive chromosome length that reduces through crossover and mutation and increases in length when an infeasible solution is found.

Ait Haddadene et al. (2016) utilized variable neighbourhood search with multiple different starting points to solve VRPTWSyn, which includes synchronization constraints in the VRPTW. Their neighbourhood operators were based on adjusting the solution by moving and swapping customers. Similarly, R. Liu et al. (2019) used simulated annealing to solve VRPTWSyn. They used simple insert and removal operators that were combined and adjusted to work with the synchronization and time window constraints. More recently, W. Liu et al. (2021) solved the home health care routing and scheduling problem with time windows, synchronized visits, and lunch breaks using four different algorithms, each a combination of a genetic algorithm and a local search algorithm. They found that the hybrid genetic general variable neighbourhood search algorithm performed the best out of the four.

Szabó (2023) completed his thesis on scheduling mechanics on the shunting yard, which turned out to be a problem similar to ours as it has both time windows and synchronization. To solve the problem, he utilized a simulated annealing algorithm. His major contribution was the introduction of a new operator called GOLDS, which efficiently deals with synchronization in the local search. This operator increases the duration of jobs in a way that allows them to have some flexibility to move around during the search by constantly having some breathing room. The simulated annealing algorithm adjusts the temperature, reducing the effect of this increase until the jobs converge to be fully

synchronized. This approach allows for more efficient job scheduling while also accommodating the synchronization constraint.

In the study conducted by van Twist et al. (2021), the authors solve the problem of assisting Passengers with Reduced Mobility (PRMs) while they make their way around an airport. The objective is to devise an employee schedule that maximizes support for PRMs and minimises waiting times. One of their constraints involves the necessity for synchronized task execution to facilitate smooth PRM handovers. This study proposes a novel approach using Simulated Annealing to determine feasible start times for passenger journeys. Subsequently, a heuristic matching algorithm is employed to assign tasks to employees in each iteration of the decomposition model. The experimental results showcase the algorithm's capability to establish robust, smooth connections in deterministic instances.

3 Mathematical formulations

In this section, we will present the problem as an ILP in four different ways. The first two mathematical formulations are commonly used to model vehicle routing problems with synchronization and time windows, both of which rely on converting the problem into a graph. The graphs consist of nodes that represent jobs and arcs that connect them to indicate the order in which they are completed by the driver. The two formulations differ in what data is contained by the nodes. The first formulation, known as the time-indexed model, includes a node for each possible combination of time, driver, and job. In contrast, the second formulation, called the mixed-integer formulation, only includes a node for each possible combination of driver and job, with a separate variable representing the time at which each job is completed. These first two formulations will not be used to solve the problem and are included only to present a complete picture of the problem. The thesis written by Szabó (2023) tested solving another formulation, similar to the first two, using the Gurobi Solver. In his formulation, he used a variable x_{ij} indicating that job i was the predecessor of job j and a variable t_i indicating the time at which job i was completed. While his problem had some additional constraints and mainly focused on robustness instead of trying to complete as many jobs as possible, this ILP approach was so much too slow that even our slightly less complicated problem would not be close to solvable by an ILP solver when using these approaches.

In the third formulation, we describe the model as a set covering problem in which a set of duties is available for selection. It is assumed that each duty is feasible, meaning it does not violate any constraints. This formulation will be the basis for the solution method which we will present in the next section. The fourth formulation is very similar to the time-indexed model and forms the basis for the method currently used by the NS.

3.1 Common Notation

Each of the formulations uses a set of common notations, which will be explained in this subsection. Additionally, each variable or set defined in this subsection is shown in Table 1, Table 2 or Table 3. The set of drivers is denoted by $D = \{1, 2, .., k-1, k\}$, where k is the number of drivers. Each driver d has a time window defined as $[\alpha_d, \beta_d]$, within which all scheduled jobs must start and finish. A preferred break time for each driver d that is required to have a break is defined as b_d. The set of jobs is denoted by $T = \{1, 2, .., n-1, n\}$. Riding along with a train is very similar to regular jobs. The only difference is that for normal jobs, there is a cost for not doing the job, while for riding along, there is a cost for doing it. Riding along is not allowed in every shunting yard. However, for those shunting yards that do allow it, every job that drives a train from one location to another will have an accompanying riding-along job. Similarly, the only difference between regular jobs and breaks is that some drivers need a break in their schedule. Therefore, we represent both breaks and riding along as jobs and add them to the set of jobs. Several subsets of jobs are defined: $T_r \subset T$ represents the set of jobs that represent riding along with the train, $T_b \subset T$ is the subset of jobs representing breaks, $T_x \subset T$ is the subset of jobs that are fixed. We also add a dummy starting and ending job for each driver, which are denoted by 0^d and n^d, which make sure they start at the correct location. We define the superset $T_S \supset T$ that includes all dummy starting and ending jobs. Finally, we define the set M as all the minutes between the earliest possible time and the latest possible time to plan anything.

For each job, $i \in T$, its duration is denoted by l_i and its time window is defined as $[\alpha_i, \beta_i]$. The cost of not completing job i is given by the parameter c_i; in case i is a riding along job c_i is equal to the cost you save by not riding along. The walking time between two jobs i and j is defined as w_{ij}, and the cost for walking between the jobs is given by c_{ij}. The cost per minute of deviation from the preferred time of a job i is defined as c_i^d. This cost is different depending on whether job i is a break, ride-along, or regular job. The parameter a_{im} is defined as the difference of time between the preferred time and scheduled time if job i is planned at minute m, which helps in calculating the cost of deviating from the preferred time.

The set of synchronized pairs of jobs is defined as P, where each pair $(i, j) \in P$ indicates that job i is the main job and job j is the assisting job. The delay between the start of the main and assisting job is denoted by $p_{ij} = \alpha_j - \alpha_i$. Each fixed job i is assigned to a specific driver d_i. Preferred successor pairs $(i, j) \in S$ are also defined, with the cost for job j not being the job planned next after job i given by c^s. Whenever we want a buffer of time between two jobs to increase robustness, there is a maximum time after which extra buffer time is not preferable. This maximum buffer time is defined as b.

Table 1: Common Notation - Sets

Set	**Description**	**Possible Values**
D	Set of drivers	$\{1, 2, .., k-1, k\}$
T	Set of jobs	$\{1, 2, .., n-1, n\}$
T_r	Subset of jobs representing riding along with the train	Subset of T
T_b	Subset of jobs representing breaks	Subset of T
T_x	Subset of fixed jobs	Subset of T
T_S	Superset of T including dummy starting and ending jobs	Subset of T
M	Set of minutes between earliest and latest possible time	Numeric values
P	Set of synchronized pairs of jobs	Set of pairs from T
S	Preferred successor pairs (i, j)	Set of pairs from T

Table 2: Common Notation - Regular Variables

Variable	**Description**	**Possible Values**
α_d, β_d	Time window for driver d	Time intervals
b_d	Preferred break time for driver d	Time points
l_i	Duration of job $i \in T$	Numeric values
$[\alpha_i, \beta_i]$	Time window for job $i \in T$	Time intervals
a_{im}	Difference in time between preferred and scheduled time for job i at minute m	Numeric values
p_{ij}	Delay between the start of main job i and assisting job j	Numeric values
w_{ij}	Walking time between jobs i and j	Numeric values

Table 3: Common Notation - Costs

Cost	**Description**	**Possible Values**
c_i	Cost of not completing job i	Numeric values
c_{ij}	Cost for walking between jobs i and j	Numeric values
c_i^d	Cost per minute of deviation for job i	Numeric values
c^s	Cost for job j not being the job planned next after job i	Numeric values
b	Maximum buffer time	Numeric values
$c_{ijmm'}$	Buffer cost between job i scheduled to start at minute m and job j scheduled to start at minute m'	Numeric values

3.2 Time-indexed Model

The time-indexed model adopts a graph-based approach to model the problem, where vertices represent each possible combination of job, driver, and time. For instance, a vertex might be infeasible if it is a fixed job meant for another driver or if the time is outside of that driver's time window or outside of the job's time window. An arc is created for every pair of vertices that belong to the same driver and for which it is possible to consecutively do the jobs. To formulate the problem mathematically, we define the binary decision variable x_{ijdm} as 1 if driver d starts job i at minute m and then proceeds to job j. By doing so, we select an edge between the vertex representing driver d starting job i at minute m and one of the vertices for job j and driver d. Note that the exact time at which job j will be done is still undecided at this point. On top of that, we generalize the objective and constraint function to sum over M for each job, while many jobs actually only have one possible starting time equal to the start of their time window. We also introduce the binary coverage variable y_i, which is equal to 1 if we plan to execute job i and 0 otherwise. Finally, we define the buffer cost parameter $c_{ijmm'}$, which reflects the additional cost incurred when executing job i at minute m and job j at minute m'. The resulting ILP is shown below in model A, with $(A.1 - A.5)$ making up the objective function and $(A.6 - A.16)$ making up the constraints.

Objective Function

minimize

$$\sum_{i \in T} c_i \cdot (1 - y_i) + \tag{A.1}$$

$$\sum_{i \in T_S} \sum_{j \in T_S} \sum_{d \in D} \sum_{m \in M} c_{ij} \cdot x_{ijdm} + \tag{A.2}$$

$$\sum_{i \in T} \sum_{j \in T_S} \sum_{d \in D} \sum_{m \in M} c_i^d \cdot x_{ijdm} \cdot a_{im} + \tag{A.3}$$

$$\sum_{(i,j) \in S} \sum_{d \in D} \sum_{m \in M} -c^s \cdot x_{ijdm} + \tag{A.4}$$

$$\sum_{h \in T_S} \sum_{i \in T_S / \{h\}} \sum_{j \in T_S / \{h,i\}} \sum_{d \in D} \sum_{m \in M} \sum_{n \in M} c_{ijmn} \cdot x_{ijdm} \cdot x_{jhdn} \tag{A.5}$$

The time-indexed model has an objective that is built up from several smaller goals. The most important objective (A.1) is assigning as many jobs as possible. Objective (A.2) adds the costs for walking between jobs. Objective (A.3) adds the costs for deviating from the preferred time for jobs. Objective (A.4) subtracts the costs for deviating from the preferred succession for a pair of jobs for each succession pair that **is** selected. Objective (A.5) adds the costs for not having the maximum buffer time. This objective is not linear but can be linearized using the 'big M method'.

Constraints

$$\sum_{j\in T}\sum_{d\in D}\sum_{m\in M} x_{ijdm} = 1, \qquad \forall i \in T_x \tag{A.6}$$

$$\sum_{i\in T_b}\sum_{j\in T}\sum_{m\in M} x_{ijdm} = 1, \qquad \forall d \in D \tag{A.7}$$

$$\sum_{j\in T}\sum_{d\in D}\sum_{m\in M} x_{ijdm} = y_i, \qquad \forall i \in T/T_x/T_b \tag{A.8}$$

$$\sum_{j\in T_S}\sum_{m\in M} x_{0^d jdm} = \sum_{j\in T_S}\sum_{m\in M} x_{jn^d dm} = 1, \qquad \forall d \in D \tag{A.9}$$

$$\sum_{i\in T}\sum_{m\in M} x_{ihdm} - \sum_{j\in T}\sum_{m\in M} x_{hjdm} = 0, \qquad \forall h \in T, \forall d \in D \tag{A.10}$$

$$y_i \geq y_j, \qquad \forall (i,j) \in P \tag{A.11}$$

$$\sum_{d\in D}\sum_{h\in T} x_{ihdm} \geq \sum_{d\in D}\sum_{h\in T} x_{jhd(m-p_{ij})}, \qquad \forall m \in M, \forall (i,j) \in P \tag{A.12}$$

$$x_{ijdm} \in \{0,1\}, \qquad \forall i \in T, \forall j \in T, \forall m \in M, \forall d \in D \tag{A.13}$$

$$y_i \in \{0,1\}, \qquad \forall i \in T \tag{A.14}$$

Constraints (A.6-A.7) ensure that each fixed job is assigned to exactly one driver and each driver (that needs one) has been assigned a break job. Constraints (A.8) make sure the remaining jobs are assigned to at most one driver. Constraints (A.9-A.10) guarantee that each driver's schedule begins at its begin job, ends at its end job and every vertex that has an outgoing edge has an incoming edge. Constraints (A.11-A.12) make certain that an assisting job can only be planned if its main job is planned. And that if a pair of synchronized jobs are both planned, they are scheduled equidistantly from the beginning of their respective time windows. Constraints (A.13-A.14) make sure the variables x and y are binary.

3.3 Mixed-integer Model

The Mixed-Integer Model also adopts a graph-based approach to model the problem. The vertices represent every potential combination of job and driver, excluding the infeasible ones. For instance, a vertex might be infeasible if it contains a fixed job assigned to another driver or if it is never possible to reach the second job on time after completing the first job. An arc is created for every pair of vertices that belong to the same driver and for which it is possible to do the two jobs consecutively. Several variables are defined to formulate this problem as a mixed-integer model. The binary coverage variable y_i is defined as 1 if job i is planned and 0 if it is unplanned. The variable x_{ijd} is set to 1 if job j is the direct successor of job i for driver d. The scheduling variable t_{id} is set to 0 if job i is not planned for driver d, and it represents the time at which the job is scheduled otherwise. Finally, the parameter h_i is defined as the preferred start time for job i. The resulting ILP is shown below in model B, with $(B.1 - B.5)$

making up the objective function and $(B.6 - B.19)$ making up the constraints.

Objective Function

minimize

$$\sum_{i \in T} c_i \cdot (1 - y_i) + \tag{B.1}$$

$$\sum_{i \in T_S} \sum_{j \in T_S} \sum_{d \in D} c_{ij} \cdot x_{ijd} + \tag{B.2}$$

$$\sum_{i \in T} \sum_{d \in D} \sum_{j \in T_S} c_i^d \cdot |h_i \cdot x_{ijd} - t_{id}| + \tag{B.3}$$

$$\sum_{(i,j) \in S} \sum_{d \in D} -c^s \cdot x_{ijd} + \tag{B.4}$$

$$\sum_{h \in T_S} \sum_{i \in T_S} \sum_{j \in T_S} \sum_{d \in D} x_{ijd} \cdot x_{jhd} \cdot max(min(b, t_{jd} - t_{id} - e_{ij} - l_i), 0) \tag{B.5}$$

Similarly to (A), the mixed-integer model has an objective built up from several smaller goals. The most important objective (B.1) is assigning as many jobs as possible. Objective (B.2) adds the costs for walking between jobs. Objective (B.3) adds the costs for deviating from the preferred time for jobs. This constraint is not linear but can be linearized using the 'big M method'. Objective (B.4) subtracts the costs for deviating from the preferred succession for a pair of jobs for each succession pair that **is** selected. Objective (B.5) adds the costs for not having the maximum buffer time. This objective is not linear but can be linearized using the 'big M method'.

Constraints

$$\sum_{d\in D}\sum_{j\in T} x_{ijd} = 1, \qquad \forall i \in T_x \qquad (B.6)$$

$$\sum_{i\in T_b}\sum_{j\in T} x_{ijd} = 1, \qquad \forall d \in D \qquad (B.7)$$

$$\sum_{d\in D}\sum_{j\in T} x_{ijd} = y_i, \qquad \forall i \in T/T_x/T_b \qquad (B.8)$$

$$\sum_{j\in T} x_{0_d jd} = \sum_{i\in T} x_{in_d d} = 1, \qquad \forall d \in D \qquad (B.9)$$

$$\sum_{j\in T} x_{ijd} - \sum_{j\in T} x_{jid} = 0, \qquad \forall i \in T_S, \forall d \in D \qquad (B.10)$$

$$t_{id} + e_{ij} + l_i - (M \cdot x_{ijd}) \leq t_{jd}, \qquad \forall s \in S, \forall i,j \in T_S \qquad (B.11)$$

$$\alpha_i \cdot \sum_{j\in T_S} x_{ijd} \leq t_{id} \leq +\beta_i \cdot \sum_{j\in T_S} x_{ijd}, \qquad \forall d \in D, \forall i \in T_S \qquad (B.12)$$

$$\alpha_d \leq t_{id}, \qquad \forall d \in D, \forall i \in T_S \qquad (B.13)$$

$$\beta_d \geq t_{id} + l_i, \qquad \forall d \in D, \forall i \in T_S \qquad (B.14)$$

$$y_j = 1 \implies \sum_{d\in D} t_{id} - \sum_{d\in D} t_{jd} = p_{ij}, \qquad \forall (i,j) \in P \qquad (B.15)$$

$$y_i \geq y_j, \qquad \forall (i,j) \in P \qquad (B.16)$$

$$x_{ijd} \in \{0,1\}, \qquad \forall i \in T_S, \forall j \in T_S, \forall d \in D \qquad (B.17)$$

$$y_i \in \{0,1\}, \qquad \forall i \in T \qquad (B.18)$$

$$t_{id} \in \mathbb{N}, \qquad \forall i \in T_S, \forall d \in D \qquad (B.19)$$

Constraints (B.6-B.7) ensure that each fixed job is assigned to precisely one driver and each driver (that needs one) has been assigned a break job. Constraints (B.8) ensure the remaining jobs are assigned to at most one driver and assign the variable y_i to 1 if a job is planned. Constraints (B.9-B.10) guarantee that each driver's schedule begins at its begin job, ends at its end job, and is not segmented. Constraints (B.11-B.14) ensure that each job is planned at a time that complies with all the relevant time windows. Constraints (B.15-B.16) make certain that each assisting job is only planned if its main job is planned. And that if a pair of synchronized jobs are planned, they are scheduled equidistantly from the beginning of their respective time windows. Constraints (B.16) are currently not linear but can be linearized using the 'big M method'. Constraints (B.17-B.19) make sure the variables x and y are binary and that the variably t is a natural number.

3.4 Set Covering model

In the set covering models, it is assumed that there is a predetermined set of feasible duties R, and the task at hand is to select a subset of these duties in a manner that satisfies all constraints. To achieve this, the binary variable λ_r

is introduced to represent the selection of schedule $r \in R$. Parameter c_r defines the cost of selecting schedule r, the exact costs c_r is made up from are shown below in equation CR which is a summation of $(CR.1 - CR.4)$. Additionally, the binary parameter a_{ir} is introduced, where its value is 1 if job i is included in schedule r, and 0 if it is not. The parameter t_{ir} is also defined, which indicates the start time of job i in schedule r, with a value of 0 if i is not included in the schedule. Finally, we define the parameter b_r^d, which is 1 if driver d can complete schedule r. We also add a set of dummy drivers who can complete a set of dummy schedules $\overline{r}$. These schedules are used to insert a dummy starting time for a pair of assisting jobs. For every job, we insert a series of such dummy schedules containing only that job, with each possible starting time for that job being contained in one of the schedules. The cost for not doing a job i (c_i) is contained within the cost of the dummy schedules. We also add a dummy driver to D for every dummy schedule to select these dummy schedules. Finally, some drivers are exactly equal, meaning they have the same starting time and ending time, no fixed jobs and the same break time. To avoid symmetry in the ILP, we add a parameter q_d. The value of this parameter is equal to the number of drivers who are exactly equal to this driver for one driver of such a group of equal drivers and 0 for all other drivers in that group. For unique drivers, the value of this parameter is equal to 1. The resulting ILP is shown below in model C, with $(C.1 - C.6)$ making up the objective function and the constraints.

$$
\begin{aligned}
min \quad & \sum_{r \in R \cup \overline{R}} c_r \cdot \lambda_r, && && (C.1)\\
st. \quad & \sum_{r \in R} a_{ir} \cdot \lambda_r + \sum_{r \in \overline{R}} a_{ir} \cdot \lambda_r = 1, && \forall i \in T && (C.2)\\
& \sum_{r \in R} b_r^d \cdot \lambda_r \leq q_d, && \forall d \in D && (C.3)\\
& \sum_{r \in R \cup \overline{R}} t_{ir} \cdot \lambda_r - \sum_{r \in R \cup \overline{R}} t_{jr} \cdot \lambda_r = p_{ij}, && \forall (i,j) \in P && (C.4)\\
& \sum_{r \in R} a_{ir} \cdot \lambda_r \geq \sum_{r \in R} a_{jr} \cdot \lambda_r, && \forall (i,j) \in P && (C.5)\\
& \lambda_r \in \{0,1\}, && \forall r \in R && (C.6)
\end{aligned}
$$

The objective function (C.1) is the costs of the selected schedules plus the costs for not completing jobs contained within the dummy schedules. The constraints (C.2) ensure that each job is either chosen by one of the regular schedules or one of the dummy schedules. Constraints (C.3) ensure that each driver has at most one selected schedule. Constraints (C.4-C.5) make certain that each assisting job is only planned if its main job is planned. And that if a pair of synchronized jobs are planned, they are scheduled equidistantly from the beginning of their respective time windows. Constraints (C.6) make sure the variables λ are binary.

In this model, we have two types of schedules: one contains the dummy schedules, while the other includes the normal schedules. The cost c_r of a dummy schedule is shown in Equation 1 while the cost for regular schedules is shown in Equation CR. Variables used in this equation are equal to those used

in model B. This means the binary coverage variable y_i is defined as 1 if job i is planned and 0 if it is unplanned. The variable x_{ijd} is set to 1 if job j is the direct successor of job i for driver d. The scheduling variable t_{id} is set to 0 if job i is not planned for driver d, and it represents the time at which the job is scheduled otherwise. Finally, the parameter h_i is defined as the preferred start time for job i.

$$c_r = \sum_{i \in T} c_i \cdot a_{ir} \tag{1}$$

$$\sum_{i \in T_S} \sum_{j \in T_S} \sum_{d \in D} c_{ij} \cdot x_{ijd} + \tag{CR.1}$$

$$\sum_{i \in T} \sum_{d \in D} \sum_{j \in T_S} c_i^d \cdot |h_i \cdot x_{ijd} - t_{id}| + \tag{CR.2}$$

$$\sum_{(i,j) \in S} \sum_{d \in D} -c^s \cdot x_{ijd} + \tag{CR.3}$$

$$\sum_{h \in T_S} \sum_{i \in T_S} \sum_{j \in T_S} \sum_{d \in D} x_{ijd} \cdot x_{jhd} \cdot max(min(b, t_{jd} - t_{id} - e_{ij} - l_i), 0) \tag{CR.4}$$

3.5 Current Method used at NS

Currently, the method used at the NS to solve this problem is running CPLEX on a variant of the time-indexed model. This model is listed below. For the mathematical formulation, we define x_{idm} as the vertex with job i started at minute m by driver d, with value 1 if the vertex is selected and 0 if it is not. Vertices are not added if it is a fixed job for another driver or if the time is outside of that driver's time window or outside of the job's time window. An arc z_{ijdmn} is created for every pair of vertices that belong to the same driver and for which it is possible to consecutively do the jobs. If a driver d completes job i started at minute m and then goes on to start job j at minute n, the variable z_{ijdmn} is 1. We also define the binary coverage variable y_i, where y_i is 1 if we have planned job i and 0 if job i is unplanned. Additionally, we create a parameter $c_{ijmm'}$, which is the buffer cost for starting job i at minute m and job j at minute m'. Finally, the parameter a_{im} is defined as the difference of time between the preferred time and scheduled time if job i is planned at minute m, which helps in calculating the cost of deviating from the preferred time. The resulting ILP is shown below in model D, with $(D.1 - D.5)$ making up the objective function and $(D.6 - D.16)$ making up the constraints.

Objective Function

minimize

$$\sum_{i \in T} c_i \cdot y_i + \tag{D.1}$$

$$\sum_{i \in T_S} \sum_{j \in T_S} \sum_{d \in D} \sum_{m \in M} \sum_{n \in M} c_{ij} \cdot z_{ijdmn} + \tag{D.2}$$

$$\sum_{i \in T/T_r} \sum_{d \in D} \sum_{m \in M} c_i^d \cdot x_{idm} \cdot a_{im} + \tag{D.3}$$

$$\sum_{(i,j) \in S} \sum_{h \in T_S/\{j\}} \sum_{d \in D} \sum_{m \in M} \sum_{n \in M} c^s \cdot z_{ihdmn} + \tag{D.4}$$

$$\sum_{i \in T_S} \sum_{j \in T_S/i} \sum_{d \in D} \sum_{m \in M} \sum_{n \in M} c_{ijmn} \cdot z_{ijdmn} \tag{D.5}$$

The current model has an objective built up from several smaller goals. The most important objective (D.1) is assigning as many jobs as possible. Objective (D.2) adds the costs for walking between jobs. Objective (D.3) adds the costs for deviating from the preferred time for a job. Objective (D.4) subtracts the costs for deviating from the preferred succession for a pair of jobs, for each succession pair that **is** selected. Objective (D.5) adds the costs for not having the maximum buffer time.

Constraints

$$\sum_{d \in D} \sum_{m \in M} x_{idm} = 1, \qquad \forall i \in T_x \tag{D.6}$$

$$\sum_{i \in T_b} \sum_{m \in M} x_{idm} = 1, \qquad \forall d \in D \tag{D.7}$$

$$\sum_{d \in D} \sum_{m \in M} x_{idm} = y_i, \qquad \forall i \in T/T_x/T_b \tag{D.8}$$

$$\sum_{h \in H} \sum_{n \in M} z_{ihdmn} = x_{idm}, \qquad \forall i \in T_S, \forall d \in D, \forall m \in M, \tag{D.9}$$

$$\sum_{h \in T} \sum_{m \in M} z_{hjdmn} = x_{jdn}, \qquad \forall j \in T_S, \forall d \in D, \forall n \in N \tag{D.10}$$

$$\sum_{h \in T} \sum_{m \in M} z_{0^d hdmn} = \sum_{h \in T} \sum_{m \in M} z_{hn^d dmn}, \forall j \in T, \forall d \in D, \forall n \in N \tag{D.11}$$

$$y_i \geq y_j, \qquad \forall (i,j) \in P \tag{D.12}$$

$$\sum_{d \in D} x_{idm} \geq \sum_{d \in D} x_{jd(m-p_{ij})}, \qquad \forall m \in M \ \ \& \ \ \forall (i,j) \in P \tag{D.13}$$

$$x_{idm} \in \{0,1\}, \qquad \forall i \in T, \forall m \in M, \forall d \in D \tag{D.14}$$

$$z_{ijdmn} \in \{0,1\}, \qquad \forall i \in T, \forall j \in T, \forall m \in M, \forall n \in M, \forall d \in D \tag{D.15}$$

$$y_i \in \{0,1\}, \qquad \forall i \in T \tag{D.16}$$

Constraints (D.6-D.7) ensure that each fixed job has been assigned to exactly one driver and each driver (that needs one) has been assigned a break job. Constraints (D.8) ensure the remaining jobs are assigned to at most one driver. Constraints (D.9-D.10) guarantee that if a node is used, it has both an incoming and an outgoing edge. Constraints (D.11) make sure that each driver starts and ends at its respective starting and ending node. Constraints (D.12-D.13) make certain that each assisting job is only planned if its main job is planned and that if a pair of synchronized jobs are planned, they are scheduled equidistantly from the beginning of their respective time windows. Constraints (D.14-D.16) make sure the variables x, z and y are binary.

Due to the computational complexity of solving the time-indexed model, the NS has imposed several constraints on the number of nodes and arcs in their graph. Specifically, nodes are restricted to containing only logical starting times for jobs. There are a few moments they assume are logical. For every job, its preferred time is added. For every pair of jobs, they add a time for the second job in a way that the buffer time is maximized and in a way that it is maximized while still allowing a third job to be put after the second job. Whenever a new moment is added for a job, another moment is added to its preferred predecessors and successors such that it fits exactly. Node creation is further restricted by allowing nodes to be created only within a maximum distance from their preferred time. This distance starts at 0 and increases to 5, 20, 60, and ∞. Nodes are repeatedly created using the next maximum distance if the number of nodes created is less than 10000.

After the nodes have been created, the arcs between them are generated, subject to several restrictions. The number of arcs generated is limited by dividing the maximum desired number of arcs by the number of nodes and allowing each node to have only that many outgoing arcs. Additionally, arcs are created only if they are deemed logical, which is determined based on whether or not one of the below conditions is true for a possible arc from job i to job j:

1. There is no arc yet from i to j
2. The buffer using this arc is bigger than the buffer using the other arcs going from i to j
3. The times for starting the jobs in this arc are closer to their preferred start times than previous arcs from i to j
4. job j is either a main job or an assisting job
5. One of the predecessors or successors of i and j is a main or an assisting job
6. Using this arc j can be started immediately on arrival
7. Part of the buffer time is maintained
8. If j is i's assisting job and the time window of j starts once i is finished in this arc.

4 Branch and Price

In this section, we will present the algorithm, a branch and price variation we use to solve the problem. It is split up into four parts. First, we explain the decomposition we use. Then we elaborate on our solution to the pricing problem. Then, we explain the branching rules. Finally, we present some additional strategies we use to potentially speed up the algorithm.

4.1 Decomposition

We will use a slight modification of the Dantzig-Wolfe decomposition described in the set covering model, modelC, of the previous section. We use a branch-and-price framework with column generation to solve the problem. Typically, a branch-and-price framework splits the problem into two smaller problems: a master problem and a subproblem. The subproblem focuses on generating feasible driver schedules, while the master problem selects a combination of the generated schedules to form a solution to the problem. We iteratively perform branching on the time windows for the jobs until the starting times for all jobs are consistent across the solution.

4.1.1 Master Problem

In this section, we describe the restricted master problem (RMP) in which we select a duty for each driver out of a large set of duties. In Section 3, we introduced a set partitioning-based formulation (model C, which we repeat below) that serves as the foundation of our master problem. Model C is an ILP model that assumes that every feasible duty can be selected. Due to the vast number of feasible duties, an exhaustive enumeration becomes infeasible for solving the model. Therefore, we try to solve the model with only a small number of simple duties instead, creating the RMP.

$$
\begin{aligned}
min \quad & \sum_{r \in R \cup \overline{R}} c_r \cdot \lambda_r, && && (C.1)\\
st. \quad & \sum_{r \in R} a_{ir} \cdot \lambda_r + \sum_{r \in \overline{R}} a_{ir} \cdot \lambda_r = 1, && \forall i \in T && (C.2)\\
& \sum_{r \in R} b_r^d \cdot \lambda_r \leq q_d, && \forall d \in D && (C.3)\\
& \sum_{r \in R \cup \overline{R}} t_{ir} \cdot \lambda_r - \sum_{r \in R \cup \overline{R}} t_{jr} \cdot \lambda_r = p_{ij}, && \forall (i,j) \in P && (C.4)\\
& \sum_{r \in R} a_{ir} \cdot \lambda_r \geq \sum_{r \in R} a_{jr} \cdot \lambda_r, && \forall (i,j) \in P && (C.5)\\
& \lambda_r \in \{0,1\}, && \forall r \in R && (C.6)
\end{aligned}
$$

To generate additional duties (or columns), we relax the integrality constraints (C.6). This allows us to compute the dual variables for the remaining constraints. We also remove the upper limit of 1 of constraints (C.6) as that is

already enforced by constraints (C.2). We also relax the synchronization constraints (C.4) by removing them entirely from the LP, resulting in the Relaxed Restricted Master Problem (RRMP). Using the dual variables, we can generate additional columns with negative reduced cost iteratively until no columns with negative reduced costs can be found anymore. The columns we find are allowed to contain a single job more than once. This allows us to generate columns more quickly as we can use a dynamic programming approach that does not need to keep track of all jobs done as a part of the state. Finally, we remove all dummy schedules containing only one job and add negative costs for doing a job in the costs for the schedules.

Below we describe the RRMP. We use the variable λ_r, representing the selection of schedule r. Additionally, we introduce the integral parameter a_{ir}, which represents the number of times job i is included in schedule r (note that due to our method for solving the pricing problem, this can exceed 1). The cost for selecting schedule r is denoted by c_r^p. You can find the calculation of this cost in Equation CR, with a minor modification: this time we incorporate negative costs for completing a job, eliminating the need to separately add costs for not performing jobs. Finally, we define the parameter b_r^d, which is 1 if driver d can complete schedule r.

$$
\begin{aligned}
min \quad & \sum_{r \in R} c_r^p \cdot \lambda_r & & & (M.1)\\
st. \quad & \sum_{r \in R} a_{ir} \cdot \lambda_r \leq 1, & \forall i \in T & & \pi_i (M.2)\\
& \sum_{r \in R} b_r^d \cdot \lambda_r \leq q_d, & \forall d \in D & & \omega_d (M.3)\\
& \sum_{r \in R} a_{ir} \cdot \lambda_r \geq \sum_{r \in R} a_{jr} \cdot \lambda_r, & \forall (i,j) \in P & & \gamma_{ij} (M.4)\\
& 0 \leq \lambda_r, & \forall r \in R & & (M.5)
\end{aligned}
$$

The objective function (M.1) is equal to the costs of the selected schedules. The constraints (M.2) ensure that each job is selected at most once, with π_i the dual variable for job i. Constraints (M.3) ensure that each driver has at most one selected schedule, with ω_d the dual variable for driver d. Constraints (M.4) ensure that each assisting job is only planned if its main job is planned, with $-\gamma_{ij}$ the dual variable for job i and γ_{ij} the dual variable for job j. Constraints (M.5) make sure the variables λ are at least 0.

Ideally, the solution found in the RRMP satisfies the integrality and synchronization constraints, with each job being planned in at most one schedule. However, in practice, this is not always the case. To address violations of the synchronization constraints and jobs occurring more than once, we branch on the time windows of the jobs that violate these constraints. We additionally branch on time windows if a job starts at different times in the (possibly partly)

selected schedules. This ensures that every job starts at a single time in the solution we find (even if it is not integral). To restore integrality to the solution, we employ a flow algorithm, which will be elaborated in Subsection 4.4.

After finding the primal solution for this LP we also obtain the dual solution $[\pi_i, \omega_d, \gamma_{ij}]$ where $\pi_i, \omega_d, \gamma_{ij}$ are the dual variables of Constraints (M.2), (M.3) and (M.4) respectively. Using these dual variables we can compute the reduced cost (Equation 2) for adding a new schedule 0 with variable λ_0.

$$\sum_{d \in D} b_0^d \cdot c_0^p - \sum_{i \in T} \pi_i \cdot a_{i0} - \sum_{d \in D} \omega_d \cdot b_0^d - \sum_{(i,j) \in P} \gamma_{ij} \cdot (a_{i0} - a_{j0}) \tag{2}$$

4.1.2 Pricing Problem (formulation)

In the pricing problem, we attempt to generate a feasible schedule with the biggest negative reduced cost. As the schedules we add to the master problem correspond to a driver, we can split this problem into finding a schedule for an individual driver d. This means we can simplify Equation 2 into Equation 3 by changing the summations over D to only use the single driver d (this also means b_0^d is equal to 1). We can then use this to solve the pricing problem.

$$c_0^p - \sum_{i \in T} \pi_i \cdot a_{i0} - \omega_d - \sum_{ij \in P} \gamma_{ij} \cdot (a_{i0} - a_{j0}) \tag{3}$$

To be more detailed, we have listed the ILP formulation for the pricing problem below. This is not used to solve the pricing problem but does give an exact overview of what we are trying to solve. For the formulation of the pricing problem, we adopt the graph-based approach first described in model A. Vertices represent each possible combination of job and time. For instance, a vertex might be infeasible if the driver lacks the necessary skills for a task, if it is a fixed job meant for another driver or if the time is outside of that driver's time window or outside of the job's time window. An arc is created for every pair of vertices for which it is possible to do the jobs consecutively.

To formulate the problem mathematically, we define the binary decision variable x_{ijm} as 1 if driver d starts job i at minute m and then proceeds to job j. By doing so, we select an edge between the vertex representing driver d starting job i at minute m and one of the vertices for job j and driver d. Note that the exact time at which job j will be done is still undecided at this point. On top of that, we generalize the objective and constraint function to sum over M for each job, while many jobs only have one possible starting time equal to the start of their time window. We also introduce the binary coverage variable y_i (this will be relaxed to an integer variable later), which is equal to 1 if we plan to execute job i and 0 otherwise. Finally, we define the buffer cost parameter $c_{ijmm'}$, which reflects the additional cost incurred when starting job i at minute m and starting job j at minute m'.

Objective Function

minimize

$$\sum_{i \in T} (c_i \ + \pi_i) \cdot -y_i + \tag{P.1}$$

$$\sum_{i \in T_r} \sum_{j \in T_S} \sum_{m \in M} c^r \cdot l_i \cdot x_{ijm} + \tag{P.2}$$

$$\sum_{i \in T_S} \sum_{j \in T_S} \sum_{m \in M} c_{ij} \cdot x_{ijm} + \tag{P.3}$$

$$\sum_{i \in T} \sum_{j \in T_S} \sum_{m \in M} c_i^d \cdot x_{idm} \cdot a_{im} + \tag{P.4}$$

$$\sum_{(i,j) \in S} \sum_{m \in M} -c^s \cdot x_{ijm} + \tag{P.5}$$

$$\sum_{(i,j) \in P} \gamma_i \cdot (y_i - y_j) + \tag{P.6}$$

$$\sum_{h \in T_S} \sum_{i \in T_S / \{h\}} \sum_{j \in T_S / \{h,i\}} \sum_{m \in M} \sum_{n \in M} c_{ijmn} \cdot x_{ijm} \cdot x_{jhn} + \tag{P.7}$$

$$-\omega_d \tag{P.8}$$

The Pricing Problem has an objective built up from several smaller goals. The objective (P.1) subtracts the costs for not planning a job for all planned jobs. It also subtracts the dual variable for that job. Objective (P.2) adds the costs for riding along with the train. Objective (P.3) adds the costs for walking between jobs. Objective (P.4) adds the costs for deviating from the preferred time for jobs. Objective (P.5) subtracts the costs for deviating from the preferred succession for a pair of jobs for each succession pair that **is** selected. This means we get a lower objective score if a pair from S are planned to start consecutively. Objective (P.6) adds the dual variables for constraints (M.4). Objective (P.7) adds the costs of not having the maximum buffer time. This objective is not linear but can be linearized using the big M method (which is not necessary since we are not solving it as an ILP). Objective (P.8) adds the dual variable for the driver d.

Constraints

$$\sum_{j \in T} \sum_{m \in M} x_{ijm} = 1, \qquad \forall i \in T_X \qquad (P.9)$$

$$\sum_{j \in T} \sum_{m \in M} x_{pjm} = 1, \qquad (P.10)$$

$$\sum_{j \in T} \sum_{m \in M} x_{ijm} = y_i, \qquad \forall i \in T/T_X \qquad (P.11)$$

$$\sum_{j \in T_S} \sum_{m \in M} x_{0jm} = \sum_{j \in T_S} \sum_{m \in M} x_{jnm} = 1, \qquad (P.12)$$

$$\sum_{i \in T} \sum_{m \in M} x_{ihm} - \sum_{j \in T} \sum_{m \in M} x_{hjm} = 0, \qquad \forall h \in T, \qquad (P.13)$$

$$x_{ijm} \in \{0, 1\}, \qquad \forall i \in T, \forall j \in T, \forall m \in M, \qquad (P.14)$$

$$y_i \in \{0, 1\}, \qquad \forall i \in T \qquad (P.15)$$

Constraints (P.9, P.10) ensure that this driver has been assigned all its fixed jobs and its break job. Constraints (P.11) ensure the remaining jobs are assigned at most once. Constraints (P.12, P.13) guarantee that the duty begins at its beginning job and ends at its end job, and every vertex that has an outgoing edge has an incoming edge. Constraints (P.14-P.15) make sure the variables x and y are binary.

Before we start solving this pricing problem, we relax some of its constraints. We relax constraints (P.15) to not have an upper bound (this transforms y_i into an integer variable). This means that jobs can be completed more than once. This is usually solved implicitly through the dual variables, as selecting a job twice in a single variable will increase its dual variable, making it less likely to be picked twice in the next round. It may be that despite of this, schedules will still be selected with a single job planned twice. However, this will be fixed later using the branching rules. We remove Constraints (P.9) and add fixed jobs to constraints (P.11). To ensure that fixed jobs will still be completed, we add a substantial negative cost for completing them. Due to fixed jobs having a time window in which they can only be started at one time point, it is impossible to select a fixed job twice in one schedule as the two occurrences would have to overlap. Thus, we can easily increase the cost for them without fear of it being highly likely to be in the resulting schedule more than once. For this reason, we still keep constraints (P.10), as simply giving the break high negative costs will result in it always getting selected multiple times. This results in the updated constraints below (the objective function is unchanged).

Constraints

$$\sum_{j \in T} \sum_{m \in M} x_{pjm} = 1, \tag{P.16}$$

$$\sum_{j \in T} \sum_{m \in M} x_{ijm} = y_i, \qquad \forall i \in T \tag{P.17}$$

$$\sum_{j \in T_S} \sum_{m \in M} x_{0jm} = \sum_{j \in T_S} \sum_{m \in M} x_{jnm} = 1, \tag{P.18}$$

$$\sum_{i \in T} \sum_{m \in M} x_{ihm} - \sum_{j \in T} \sum_{m \in M} x_{hjm} = 0, \qquad \forall h \in T, \tag{P.19}$$

$$x_{ijm} \in \{0, 1\}, \qquad \forall i \in T, \forall j \in T, \forall m \in M, \tag{P.20}$$

$$y_i \geq 0, \qquad \forall i \in T \tag{P.21}$$

4.2 Pricing Problem (solution)

To solve the pricing problem, we adopted a dynamic programming approach. We aim to find the schedule with the largest negative reduced cost separately for each driver. The driver d for which we are trying to find a schedule has to start with the dummy job 0 and finish with the dummy job n. We define job p as the break job for driver d if driver d has to have a break in his schedule. Each job i has time window $[\alpha_i, \beta_i]$ and duration l_i. If two jobs require a buffer between them b_{ij} is equal to $true$; otherwise, it is $false$. In our DP algorithm, we denote D_i^m as the reduced cost for the best possible schedule that starts at job 0 and ends at job i, which is started at minute m. This schedule incorporates the relaxed model's constraints (P.16-P.21).

We initialize the DP by setting the cost for $D_0^{\alpha_0}$ as the dual variable ω_d, and all other costs are set to infinity. We then iterate over each minute between the end of job 0 ($\alpha_0 + l_0$) and the start of job n (α_n). For each minute m, we iterate over the jobs to calculate the cost for D_i^m. Finally, we calculate $D_n^{\alpha_n}$ to find the best schedule. In case the driver has to have a break in his schedule, we denote $\underline{D}_i^m$ as the best possible schedule that starts at job 0 and ends at job i, which is started at minute m before having had the break and $\overline{D}_i^m$ after having had the break.

Before presenting the complete DP calculation, we first define two auxiliary functions. The first function $p(i, j, m)$ returns the possible starting times for i if job i is the job started before job j and job j starts at time m. This function considers the context of the dynamic programming, meaning that it only returns starting times that could potentially be the best. This function is based on the fact that starting a job later will allow more jobs to be started before it and can, thus, potentially be the best. However starting a job earlier might improve the buffer and preferred time costs, which could also lead to a better solution. On the other hand, it is never better to start a job earlier while not improving buffer or preferred time costs, and we can avoid returning these starting times.

To aid in the calculation of $p(i, j, m)$, we define the latest possible starting time of starting job i while being on time to start job j at minute m as $w_{ijm} = \min\{\beta_i - l_i, m - e_{ij} - l_i\}$, with e_{ij} being the walking time between job i and

j and l_i being the duration of job i. Additionally, the parameter h_i represents the preferred starting time for job i.

$$p(i,j,m) = \begin{cases} \{\} & \text{if } \alpha_i > w_{ijm}(1) \\ \{m - \alpha_j + \alpha_i\} & \text{if } (i,j) \in P(2) \\ \{w_{ijm}\} & \text{if } b_{ij} \wedge w_{ijm} < m - b_{max} \wedge h_i \geq w_{ijm}(3) \\ \{h_i, ..., w_{ijm}\} & \text{if } b_{ij} \wedge w_{ijm} < m - b_{max}(4) \\ \{\max\{\alpha_i, \min\{m - b_{max}, h_i\}\}, ..., w_{ijm}\} & \text{if } b_{ij}(5) \\ \{w_{ijm}\} & \text{if } w_{ijm} < h_i(6) \\ \{h_i, ..., w_{ijm}\} & \text{otherwise}(7) \end{cases}$$

The cases considered in the function $p(i,j,m)$ are as follows (note that the cases are checked according to their order as if it were a large if-else, e.g. case (1) is selected if both case (1) and case (2) are true): (1) it is not possible to start job j at minute m and complete job i before it. (2) If jobs i and j form a pair of assisting jobs, there is only one possible starting time that maintains the synchronization constraints as we already know the second job's starting time (m). (3) A buffer between the jobs is desired, but the latest starting time already has the maximum buffer and moving further back in time does not improve its preferred starting time. (4) A buffer between the jobs is desired, but the latest starting time already has the maximum buffer and moving further back in time will improve its preferred starting time. (5) A buffer between the jobs is desired, and moving further back in time will improve the buffer and possibly the preferred starting time. (6) No buffer is desired and moving further back than the latest starting time does not improve the preferred time. (7) No buffer is desired and moving further back than the latest possible starting time improves the preferred starting time.

The second auxiliary function $f(i,j,m,m')$ returns the additional costs for adding job j planned to start at minute m to the end of the schedule that previously had job i planned to start at minute m'. The calculation of the function f listed below uses several variables defined below.

$$f(i,j,m,m') = v_j + v^s_{ij} + v^b_{ijmm'} + v^r_j$$

The reduced cost for doing task j is defined as:

$$v_j = \begin{cases} c_j - \pi_j + \gamma_{ji} & \text{if } \exists i \in T \rightarrow (j,i) \in P \\ c_j - \pi_j - \gamma_{ij} & \text{if } \exists i \in T \rightarrow (i,j) \in P \\ c_j - \pi_j & \text{otherwise} \end{cases}$$

The bonus for not deviating from the preferred succession of (i,j) is defined as:

$$v^s_{ij} = \begin{cases} -c^s & \text{if } (i,j) \in S \\ 0 & \text{otherwise}o \end{cases}$$

The buffer cost for doing task i at minute m and then doing task j at minute m' is defined as:

$$v^b_{ijmm'} = \begin{cases} c_{ijmm'} & \text{if } b_{ij} \\ 0 & \text{otherwise} \end{cases}$$

Finally, using these functions, the DP can be described as follows:

Initialization:

For drivers who do not need a break, we initialize everything to infinity except for the start of the schedule, which is starting job 0 at minute α_0.

$$D^m_i = \begin{cases} -\omega_d & \text{if } i = 0 \text{ and } m = \alpha_0 \\ \infty & \text{otherwise} \end{cases}$$

For drivers who do need a break, we split the DP into two parts. We first initialize $\underline{D}$ as we did for drivers who do not need a break by initializing everything to infinity except for the start of the schedule, which is starting job 0 at minute α_0. Then, we solve $\underline{D}$ before initializing the $\overline{D}$ with entries from the first DP containing the break job ($\underline{D}^m_p$).

$$\underline{D}^m_i = \begin{cases} -\omega_d & \text{if } i = 0 \text{ and } m = \alpha_0 \\ \infty & \text{otherwise} \end{cases}$$

$$\overline{D}^m_i = \begin{cases} \underline{D}^m_i & \text{if } i = p \\ \infty & \text{otherwise} \end{cases}$$

Recurrence Relation:

We then iterate over each minute between the end of job 0 ($\alpha_0 + l_0$) and the start of job n (α_n). For each minute m, we iterate over all the jobs to calculate the cost for D^m_j. To do this we find the minimum of combining a job j starting at minute m with a job i before it starting at minute m'. For drivers who do not need a break, we calculate the DP in one go, starting with job 0 and finishing with job n.

$\forall m \in \{\alpha_0 + l_0, ..., \beta_n - l_n\}$, $\forall j \in T \cup \{n\}$:

$$D^m_j = \min_{i \in T \cup \{0\}/\{j\}} \{ \min_{m' \in p(i,j,m)} \{D^{m'}_i + f(j, i, m, m')\}\}$$

For drivers who need a break, we first calculate the part of the schedule preceding the break job, starting with job 0 and finishing with the break job b.

$\forall m \in \{\alpha_0 + l_0, ..., \beta_b - l_b\}$, $\forall j \in T$:

$$\underline{D}^m_j = \min_{i \in T \cup \{0\}/\{j,b\}} \{ \min_{m' \in p(i,j,m)} \{\underline{D}^{m'}_i + f(j, i, m, m')\}\}$$

We then calculate the second part of the schedule starting with the break job b and ending with the final job n.

$\forall m \in \{\alpha_b, ..., \beta_n - l_n\}$, $\forall j \in T \cap \{b\} \cup \{n\}$:

$$\overline{D}_j^m = \min_{i \in T/\{j\}} \{ \min_{m' \in p(i,j,m)} \{\overline{D}_i^{m'} + f(j,i,m,m')\}\}$$

4.2.1 Case exclusions

To speed up the dynamic programming we employ two strategies that exclude many cases we would otherwise need to consider while calculating D_j^m. In the first strategy, we try to exclude jobs that will never fit before j. To do this, we create a set of jobs for each job that contains all the jobs that can be planned right before job j. We exclude jobs from this set if any of the equations 4 are true. In equation 4a, we check whether we can still start job j on time after starting job i at its earliest starting time. In equation 4b, we check whether we can start with the beginning job at its earliest starting time, then do job i and still be on time for job j. In equation 4c, we check whether we can start with job i at its earliest starting time, then do job j and still be on time to start the ending job n. In case the driver has to do a break job, we do this preprocessing step twice: once from the start until the break and once from the break until the end.

$$\alpha_i + l_i + e_{ij} > \beta_j - l_j \tag{4a}$$
$$\alpha_0 + l_0 + e_{0i} + l_i + e_{ij} > \beta_j - l_j \tag{4b}$$
$$\alpha_i + l_i + e_{ij} + l_j + e_{jn} > \beta_n - l_n \tag{4c}$$

In the second strategy, we leverage the fact that the majority of the extra costs added when planning job j right after job i can already be calculated without knowing the exact time job i starts (m'). In the function $f(i,j,m,m')$, we only use m' to calculate the added buffer costs (which can only increase the result of f). This means we can already estimate the other costs v_j, v_{ij}^s, v_j^r. To use this, we keep track of the lowest $D_i^{m''}$ with $m'' < m$ found so far for each job j using variable r_i. If we then find that $r_i + v_j + v_{ij}^s + v_j^r$ is already higher than the best previous job we found so far while calculating D_j^m, we can skip job i and continue to the next job.

4.3 Branching

Once we can no longer find any columns with a negative reduced cost to add to the master problem, we stop generating columns and continue with the branch and bound. For the current node, we check whether the resulting cost of its solution is lower than the cost of the best feasible solution found so far. If its cost is not lower, we can discard this node as branching further on this node can only result in worse solutions, meaning we will not find any node better

than the best node. If its cost is lower, we check whether the starting times are consistent across the solution for each job (e.g. it is feasible apart from integrality); if we find an inconsistency, we use one of the branching rules to create new nodes. Otherwise, if the solution is consistent, we update it as the lowest-cost solution found thus far.

The relaxed constraints (P.9, P.15) are handled in the branching part of the branch-and-price algorithm. We employ three branching rules that are quite similar. The first branching rule enforces that one job is started at a single time point, the second rule enforces synchronization between two jobs (if required), and the third branching rule ensures the planned starting times for breaks are consistent. We select branches according to a specific order: firstly, we try to branch on synchronization; secondly, we try to branch on single starting time; and finally, we try to branch on break jobs. Below, we will present these branching rules in an easier-to-explain order (note that this is not equal to the order in which they are applied).

4.3.1 Single starting time

If, for any job j we find that it is planned to start at a number of different time points W, we branch on the time window of that job. Let job j have a time window $[\alpha_j, \beta_j]$ and a duration l_j. Our branching rule splits the time window at the midpoint between all the time points in W. We define the midpoint as $t_m = \lfloor \sum_{t \in W} t/|W| \rfloor$, and we create two new nodes: the left node with an updated time window for job j of $[\alpha_j, t_m + l_j]$, and the right node with an updated time window of $[t_m + 1, \beta_j]$. The duration of the job is added to the end of the time window in the left node to ensure that no possible starting times are lost, and they are all contained within the two child nodes. When job j belongs to a pair of synchronized jobs, such as $(i, j) \in P$ or $(j, i) \in P$, we must also modify the time window of its counterpart, job i. Let job i have a time window $[\alpha_i, \beta_i]$ and a duration l_i, the left node will use time window $[\alpha_i, t_m - \alpha_i + \alpha_j + l_j]$ and the right node will use time window $[t_m + 1 - \alpha_i + \alpha_j, \beta_i]$. The job we choose to branch on depends on two things: we prefer to choose a job with as many different time points as possible; if no job has more different time points than any other, we choose the job where the time points are furthest apart.

4.3.2 Break jobs

As we treat drivers who are equal as one in the master problem the situation for break jobs is slightly different. In case there is a group of equal drivers of size at least two, it is possible for their break jobs to be planned to start at two different time points and still be a feasible solution. It only becomes a problem once their breaks have been planned to start at more different time points than there are drivers in the group. In this case, we split on the time window of the break job for one of the drivers, which means that this driver is consequently separated from the rest of the group as it is no longer equal.

4.3.3 Synchronization

If any pair of jobs $(i, j) \in P$ are both selected in the solution with two time points t_i and t_j such that $t_i \neq t_j - \alpha_i + \alpha_j$, they are not properly synchronized in the solution. In case a job itself is already planned to start at multiple time points we take t_i and t_j as the average of its starting times. To get closer to having these jobs synchronized in the solution, we branch on the time windows for both jobs. We define $t_m = (t_i + \alpha_j - \alpha_i + t_j)/2$. Similarly to the previous section we create two new nodes: the left node with time window $[\alpha_j, t_m + l_j]$ for job j and time window $[\alpha_i, t_m - \alpha_i + \alpha_j + l_j]$ for job i, while the right node will have time windows $[t_m + 1, \beta_j]$ and $[t_m + 1 - \alpha_i + \alpha_j, \beta_i]$. For this branching rule, we prefer a pair of time points that start as far from each other as possible.

4.4 Restoring Integrality

After all the nodes following the branching rules have been explored, we have a lowest-cost solution that is still possibly infeasible due to lack of integrality. In the solution we find, every job is only planned to start at a single starting time. In van den Akker et al. (1996) they have a similar situation for which they prove they can create an integral solution with the same cost as the fractional one, but we have not been able to find a proof for our case due to the extra complexity of the problem. Still, so far, we have been able to find an integral solution with the same cost as the fractional solution for all our instances. Therefore, we think that likely a similar proof exists for our problem. To find this (likely) integral solution, we use the ILP formulation described in model D of the previous chapter. While normally it would be very slow, we only add nodes and arcs that are at least partially selected in the master problem of the lowest-cost solution that we found. This means that for every selected schedule, we add a node for each job with only the starting time it is planned at for only the drivers whose schedules contain the job. Similarly, we only add arcs that already exist in the schedule. This method can find an integral solution in only a fraction of the time it took to find the fractional solution.

4.5 Acceleration Strategies

To accelerate the branch-and-price algorithm, we employ several strategies. These strategies can be split into two parts: optimal strategies that still find an optimal solution and heuristic strategies that do not necessarily find an optimal solution. Each of these strategies has upsides and downsides. Generally, they speed up one part of the algorithm while slowing down another part, or in the case of heuristic strategies, they may worsen the solution while likely speeding up the algorithm.

4.5.1 Optimal Strategies

First, we present optimal strategies. These strategies may or may not increase the algorithm's speed but will still result in an optimal solution.

Adding Multiple Columns

Typically, a prevalent acceleration strategy for similar problems is to add multiple columns every time the pricing problem is solved. However, in our case, it did not seem to work well. Generally, only a few fewer pricing problems had to be solved while dramatically increasing the time spent on solving the master problem. Instead, we choose to use a different way of adding multiple columns. To do this, we group drivers based on their time windows; specifically we create groups of drivers as large as possible that do not have any overlap in their time windows. With no overlap in their time windows, these drivers will likely have minimal overlap in terms of jobs they can do. This means that, ideally, choosing a schedule for one of the drivers in a group will not cause a major shift of dual variables for the jobs available for the other drivers in the group. Once we have created these groups, we solve the pricing problem for each group member before adding each of the discovered columns and solving the master problem. In terms of running time, this generally means that slightly more columns have to be generated as they are not perfectly accurate. However, it also means that the number of times the master problem has to be solved is lower. We also add a parameter that sets the minimum time between the time windows of drivers before they are allowed to be in the same group. Having a larger minimum time makes sure that jobs with a larger time window will not be available for multiple drivers in the same group. This parameter therefore allows us to find a balance between having larger groups and less overlap within these groups.

Cooldown

During every iteration of the master problem, we attempt to find new columns for each driver until no driver has a column with a negative reduced cost left. Due to differences among drivers, it is possible that after several iterations, we cannot find columns with a negative reduced cost for some drivers. Continuing to calculate the entire pricing problem for these drivers is a huge waste of time. To address this, we introduce the parameter "cooldown", which determines the number of iterations we skip solving the pricing problem for a driver after failing to find a column with a negative reduced cost.

Initial columns

Initializing the branch and bound with good columns can decrease the time it takes to find an optimal solution to the master problem. We use a local search method (simulated annealing) inspired by Szabó, 2023 that can quickly find a decent solution. Our method has two major differences from his method. The first difference is that we only try to find a decent solution to start the branch and bound with, which means finding a local optimum is relatively okay. Therefore, we can decrease the temperature much faster and place less emphasis on operators that get us out of a local optimum. The second main difference is that we do not consider the synchronization. We run the local search multiple times for a very short time to find different local optimums and add all the

resulting columns to the first master problem. In our local search, we have five neighbourhood operators listed below. When the operators use 'a schedule', this schedule corresponds to one of the available drivers.

Smart Insert Operator This operator takes a random, unplanned job and a random schedule. It then tries to fit the job somewhere in the schedule and chooses the best possible location (e.g. with the lowest cost).
Forced Insert Operator This operator takes a random, unplanned job and a random schedule. It then tries to fit the job somewhere in the schedule and chooses the best possible location (e.g. with the lowest cost). This operator is allowed to move the job before and the job after the inserted job.
Remove Operator This operator takes a random schedule and a random index and tries to remove the job at that index. This operation fails if this job is a fixed or a break job.
Move Operator This operator takes two random schedules and one random index and tries to remove the job at the index of the first schedule. Then, using the smart insert operator, the job is added to the second schedule.
Optimize Pair Operator This operator takes a schedule and a pair of jobs next to each other in that schedule. It then optimizes the starting time for these two jobs, where we try to minimize the costs for buffers and deviation from preferred starting times.

4.5.2 Heuristic Strategies

Look back depth

In our DP, we encounter situations where a job may appear multiple times in a schedule we find. To address this issue, we introduce a parameter, *look_back*, which allows us to prevent such occurrences by looking back in the best solution found so far; if we come across the job in this solution, we stop considering this option. In other words, while trying to calculate $\underline{D}_j^m$, if we find a $\underline{D}_i^{m'} + v(i, j, m, m')$ that is lower than the current lowest, we look at the last *look_back* jobs to see if job j is among them. The advantage of this strategy is that we find fewer schedules that will not be in the final solution anyway. However, we might also slow down the master problem due to a slower convergence of the dual costs for the individual jobs. While this strategy is not necessarily optimal, the experiments have shown that it usually finds the optimal solution.

Preferred Succession

Pairs of preferred successions are chosen based on three main properties: firstly, both jobs operate on the same train; secondly, the end of the first job and the start of the second job are at the same location; and thirdly, it is possible (and desired) to immediately do the second job once the first is finished. While the first property is only relevant for driver satisfaction, the second and third properties are very commonly found in optimal solutions. While the objective function already includes a cost for deviating from the preferred succession, optimal solutions that do not adhere to the preferred succession may exist. For

example, if another job is available only right before the second job and it is possible to do the first job, then the other job and then the second job. However, in practice, the preferred succession often indicates a strong likelihood of two jobs succeeding each other in the optimal solution. To leverage this insight, we introduce a parameter that determines whether we only consider the preferred succession for jobs that are part of it. This means that if there is a pair of jobs $(i, j) \in S$, and we are trying to calculate D_j^m, then we only look at job i (and the starting dummy job 0). Thus, we allow the solution to break the preferred succession only right after a driver's shift starts. This approach significantly speeds up the algorithm for two reasons: first, it reduces the number of options in the pricing problem calculations, resulting in faster generation of new columns; second, it eliminates many possible columns, reducing the number of iterations required before no more columns with negative reduced cost can be found.

Stopping Criteria

At the beginning of the column generation process, nearly every column significantly improves the solution. However, as the selection of columns available for the master problem narrows and new columns contribute to increasingly smaller improvements, many iterations may no longer enhance the solution. To expedite this process, we considered stopping looking for improving columns prematurely. However, in practice, this did not work very well; when stopping earlier, the solution was more fractured which in turn resulted in many more nodes having multiple schedules starting times.

Time skips

In the case of large problems, narrowing the search space becomes valuable. We achieve this by restricting the number of starting times considered for a job. With the parameter *time_skip*, we only look at times which are a multiple of *time_skip*. This means that in the DP we only calculate D_i^m if *time_skip* is divisible by $m - a_i$. This might exclude some very good options from the search space, but it also speeds up the algorithm with the smaller search space.

Prepossessing riding along

The possibility of riding along with jobs makes the problem take a considerable amount of extra time to solve while generally not providing a large benefit in terms of the total cost of the solution. To leverage this fact, we introduce three strategies to deal with riding along. The first strategy is dealing with them as if they are jobs that do not have any cost with not doing them; this strategy is optimal but slightly slow. The second strategy is not to allow any riding along at all. Obviously, this strategy is not optimal, but it is the fastest. For the third strategy, we calculate where we ride along as a preprocessing step. Typically, jobs that have a time window are easier to fit into a schedule somewhere. Therefore, we only allow riding along between two jobs that have a fixed starting time. This is also due to only having to calculate the perfect riding along job for one time point, as different riding along jobs may be available at

different time points in the time window. Between jobs with a fixed starting time, we can easily calculate the riding along job that results in the lowest costs. While solving the pricing problem, we always select this riding along job if the two jobs follow each other in the schedule.

5 Results

This section aims to assess the performance of the branch and price algorithm and compare it to the currently used algorithm. To achieve this, we experiment with different settings for the parameters for the accelerating strategies discussed in the previous chapter. First, we will elaborate on the instances we will experiment with. Then, we will experiment with finding the best parameters for the algorithm using some of the smaller instances. Finally, we present the results for the larger instances using the previously found parameters. The experiments are executed on an Intel(R) Core(TM) i5-8300H CPU with a clock speed of 2.30 GHz.

5.1 Instances

To evaluate our implementation, we use thirteen real-life instances representing various shunting yards in the Netherlands. Table 4 provides details about each instance, indicating whether riding along is allowed (1) or not allowed (0) and specifying the number of various job types present in the instance. Instances are categorized into 'small' and 'large' based on whether or not the branch and price can quickly solve the problem optimally. Notably, "Rotterdam" and "Den Haag" are classified as large instances, while the rest fall under the easy category. Consequently, this means Rotterdam and Den Haag are more important to solve efficiently as the difference between fast and very fast is negligible. On the other hand, the difference between slow and very slow is more noticeable.

5.2 Parameters

For each strategy explained in the accelerating strategies section of the previous chapter, there is an accompanying parameter that can take different values. While for some of the parameters, this is a simple on or off switch, other parameters can take an integer value. For such parameters that take an integer value, it is impossible to test every different option, especially since the parameters often interact with each other. Testing every feasible combination of parameter settings would simply take too long. Therefore we have selected a couple of different options for each numerical parameter. We will present the results for these parameters in two ways. We first show the results of the different values a parameter can take independent of other variables (e.g. taking the average over all runs with that parameter set to that value), including showcasing parameter-specific effects for some parameters. In the next section, we will show the best combinations of parameters and their performance.

Table 4: Instances used

Location	Riding Along	Assisting Pairs	Drivers	Jobs	Time Windows
Amersfoort	0	0	14	174	22
Arnhem	0	1	3	19	1
Deventer	1	1	2	106	1
Eindhoven	1	1	10	253	8
Enschede	0	7	9	108	10
Groningen	1	0	11	291	0
Leeuwarden	0	0	15	139	25
Nijmegen	0	1	3	33	3
Zutphen	0	0	7	67	4
Zwolle	0	2	7	85	2
Utrecht	1	12	69	499	43
Rotterdam	1	47	37	800	88
Den Haag	1	84	53	900	169

The results for each of the parameters can be found in the Tables 5 - 10 below. For each parameter that does not affect optimality, we do not show the cost which does not change with different parameter settings. Table 5 shows the results of the cooldown strategy. For this parameter, we have chosen to test with a cooldown of 0, 5, 10 and 15, meaning a driver is skipped respectively 0, 5, 10 and 15 iterations after not producing a column with reduced cost. From these results, we can see that for most instances, a cooldown between 5 and 15 performs best. However, it should be noted that this likely scales with the number of iterations necessary in general, meaning that skipping 15 iterations while only needing 25 iterations makes less sense than skipping 15 iterations while 300 are done. Table 6 shows the results of the strategy for adding multiple columns. For this strategy, we have chosen to test with the settings 0, 4 and 1000, meaning drivers can be grouped if their time windows are apart by at least 0, 4 and 1000 hours (where 1000 means drivers are never grouped). In this table, we left out all instances that would not result in grouped columns due to drivers having similar time windows. The results show that for most instances, grouping with a distance of 0 hours is the fastest. Table 7 shows the results for starting the problem with initial columns found using a simulated annealing approach. For this strategy, we run the simulated annealing 4 times with 300000 iterations each (or stop if each job has been planned), adding each schedule to the first node. This means that for smaller instances that did not need as many iterations the simulated annealing took up a larger part of the used running time. Therefore it performs much better on larger instances (or small instances where every job can be planned). From the results, we can see that for larger instances it is slightly faster when adding initial columns. Table 8 shows the results of using the look back strategy. We left out each instance where there are no time windows that allow for a job to be done twice in the same schedule.

The results show that it is highly dependent on the instance whether or not using this strategy pays off. For some instances, it is slightly faster when using the parameter, while for some other instances, it is slightly slower. Table 9 shows the results of using the preferred successor strategy. From these results, we can see that it is nearly always much faster to use this strategy. For most instances the preferred successors are also in the optimal solution, meaning that using this strategy will not cause worse results. However, for some instances, the results are much worse. One possible workaround could be to first run the program with preferred successors before running it without. Table 10 shows the results for different riding along strategies. From these results, we can see that including riding along will only cause a small dip in solution quality while providing marginal speedups. Therefore it could be useful for larger instances to not use riding along as a possible option.

Table 5: Results for Cooldown strategy with settings 0, 5, 10 and 15

Location	Time (ms)			
	0	5	10	15
Alkmaar	939	768	775	783
Amersfoort	113970	96194	98098	89958
Arnhem	282	270	271	280
Deventer	914	902	877	877
Eindhoven	3173	3081	3190	3257
Enschede	1782	1652	1636	1655
Groningen	245	227	227	228
Leeuwarden	1602	1023	971	976
Nijmegen	86	90	92	82
Utrecht	10139	8188	8018	8273
Zutphen	403	347	357	349
Zwolle	613	600	593	604

Table 6: Results for Adding Multiple Columns, with a minimum group distance of 0, 4 and 1000 hours

Location	Time (ms)		
	0	4	1000
Alkmaar	820	811	818
Amersfoort	91797	103961	102907
Enschede	1571	1721	1751
Groningen	233	230	233
Leeuwarden	1122	1180	1128
Utrecht	8396	8568	9000
Zutphen	363	377	353

Table 7: Results for using Initial Columns generated using simulated annealing

Location	Time (ms)	
	no Initial Columns	Initial Columns
Alkmaar	663	1098
Amersfoort	179507	172052
Arnhem	25	515
Deventer	529	1289
Eindhoven	3683	3172
Enschede	1338	2087
Groningen	237	228
Leeuwarden	801	1615
Nijmegen	93	78
Utrecht	9603	8794
Zutphen	76	661
Zwolle	329	858

Table 8: Results for Looking Back with a depth of 0, 4, 7

Location	Time (ms)		
	0	4	7
Alkmaar	867	798	784
Amersfoort	109008	91456	98201
Eindhoven	3102	3196	3229
Enschede	1678	1692	1672
Leeuwarden	1260	1111	1059
Nijmegen	88	86	88
Utrecht	8573	8816	8575

5.3 Small Instances

Showing the performance results for the small instances using every combination of values for the parameters is impractical, even when only considering the smaller subset of possible values. This would result in a table with over 1000 rows. Therefore, we select a subset of the possible combinations, including the best individual combinations for each instance and the overall best parameter values. While this may result in somewhat cherry-picked results, we can reasonably assume that repeating requests for the same location will also look similar. Therefore the results are still likely to be representative of the actual performance. In Table 11 you can see all the combinations of parameter settings we will use. These settings are made up of two situations. They are either a combination for the best time (with optimal cost) for one of the instances in

Table 9: Results for Only Successor strategy with settings true and false

Location	Cost		Time (ms)	
	false	true	false	true
Alkmaar	5648300	6126700	663	489
Amersfoort	107801	107801	179507	15460
Arnhem	702986	702986	25	26
Deventer	908124	908124	529	437
Eindhoven	25504	25504	3683	3051
Enschede	703327	703327	1338	1237
Groningen	5070	5070	237	235
Leeuwarden	229524	407024	801	614
Nijmegen	2369	2369	93	96
Utrecht	6739375	7017186	9603	8388
Zutphen	1112780	1112780	76	83
Zwolle	108626	108626	329	354

Table 10: Results for different riding along settings with settings No Riding Along (NRA), Preprocessed(PP) and As Jobs(AJ)

Location	Cost			Time (ms)		
	NRA	PP	AJ	NRA	PP	AJ
Deventer	908124	908124	908124	792	880	1005
Eindhoven	25677	25578	25257	2800	2833.0	3893
Groningen	5070	5070	5070	176	327	192
Utrecht	6878539	6878201	6878139	7656	8888	9419

which case the Origin column will show the location for which it was the best combination. Or, in this case, the Origin column is empty, it is a combination that fills the gap between the other combinations. The table is sorted on the settings from left to right.

Table 12 shows the time results for all the parameter combinations. In this table entries that are not optimal are followed by * if it is due to the successors only strategy and they are followed by ** if it is due to the riding along strategy. In both cases, you can find the effect of using the strategy on the cost in the respective tables for both of the strategies (Table 9 and Table 10). For example, for Utrecht, using the preprocessed riding along setting will result in a solution with a cost 82 worse than using the As Jobs strategy. For each location, the fastest optimal entry is made bold. From these results, we can see that even though the parameter combinations influence the speed of the algorithm somewhat, many combinations still give close results.

Table 11: All parameter settings used and by which id they are accompanied

id	Origin	Cooldown	Multiple	Initial	Look Back	Successor	Riding Along
1	Arnhem	0	4	0	0	1	No_Passagieren
2	Deventer	0	10000	0	7	1	No_Passagieren
3	Nijmegen	0	10000	1	0	0	No_Passagieren
4	Eindhoven	0	10000	1	0	1	Passagieren_As_Jobs
5		5	0	0	7	1	No_Passagieren
6		5	10000	1	7	0	No_Passagieren
7	Groningen	5	10000	1	7	1	No_Passagieren
8	Utrecht	10	0	1	7	0	Passagieren_As_Jobs
9	Alkmaar	10	4	0	7	0	No_Passagieren
10		10	4	1	7	0	No_Passagieren
11	Zutphen	10	10000	0	7	0	No_Passagieren
12	Enschede	15	0	0	0	1	No_Passagieren
13		15	0	1	7	0	No_Passagieren
14		15	4	1	7	0	No_Passagieren
15	Zwolle	15	4	0	7	0	No_Passagieren
16	Amersfoort	15	4	0	7	1	No_Passagieren
17	Leeuwarden	15	10000	0	7	0	No_Passagieren

Table 12: For each parameter combination (show in Table 11) and location the time it takes to solve the problem. Entries with * are not optimal due to the successor-only strategy. Entries with ** are not optimal due to not using riding along

location	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Arnhem	**20**	23	490	X	23	529	519	X	31	511	25	24	518	524	33	23	25
Deventer	366	**333**	1226	1376	370	1212	1176	1355	421	1190	401	354	1195	1187	428	367	352
Nijmegen	85	91	**65**	X	86	76	81	X	115	74	115	79	66	85	106	88	86
Eindhoven	2184**	2527**	3109**	**3110**	2790**	2755**	2431**	3779	3584**	2731**	3500**	2793**	2726**	2812**	3508**	2723**	3529**
Groningen	189	199	185	201	175	174	**157**	176	163	169	184	175	179	177	179	170	184
Utrecht	6034**	8012*	10537**	8512	7042*	7570**	6420*	**6646**	6207**	6314**	8645**	6069*	7466**	7121**	7194**	7278**	9325**
Alkmaar	659*	516*	1198	X	435*	967	952*	X	**537**	1045	637	445*	1013	1000	589	435*	653
Zutphen	91	87	648	X	69	634	600	X	61	632	**56**	74	612	639	62	67	62
Enschede	1335	1343	2194	X	1113	2042	2081	X	1395	2001	1414	**1019**	1899	2103	1246	1195	1564
Zwolle	453	348	845	X	340	860	853	X	311	874	317	355	872	866	**309**	339	333
Amersfoort	16640	19980	215935	X	14908	143726	22442	X	173325	155977	129319	10281	128985	155498	162485	**9170**	151715
Leeuwarden	2006*	821*	2396	X	445*	1502	1332*	X	523	1346	514	482*	1333	1363	485	422*	**478**

5.4 Large Instances

To run the algorithm for the larger instances, we take the best combination of parameters for the five small locations that take the longest to solve, as they are likely the most representative of the large locations, and change those combinations to suit the large instances. First, we change the instances to always use no riding along, as the change in cost is small while having a very clear effect on the speed. Secondly, we add an extra parameter for the strategy time skips that can have the values 1 or 2, meaning it only skips no cells in the DP or skips every other cell, respectively. This parameter is far from optimal as it skips large parts of the solution space, but does speed up the program significantly. Finally, because we cannot solve these instances in a reasonable time we put a limit of fifteen minutes on the running time before we stop. Once the program reaches this limit, we take the best possible solution found so far, if one has been found. In Table 13, you can see the results for the large instances. The column parameter shows the original parameter combination coupled with a setting for the time jump strategy. If we were not able to find a feasible solution within the time limit we placed 900000 as the time and an X as the cost. From these results, we can see that for Rotterdam we can find decent results sometimes when using the time jump 1 parameter and also when using time jump 2 with initial columns. For Den Haag, we are never able to find a feasible solution.

Table 13: Results for different parameter combinations for the large locations

Parameters	Rotterdam		Den Haag	
	time (ms)	cost	time (ms)	cost
4-1	900000	X	900000	X
4-2	900000	2788265	900000	X
8-1	900000	X	900000	X
8-2	900000	2788220	900000	X
9-1	900000	1489402	900000	X
9-2	900000	1489902	900000	X
12-1	900000	1489365	900000	X
12-2	900000	2788220	900000	X
16-1	900000	1489367	900000	X
16-2	900000	2698670	900000	X

5.5 Performance

In Table 14, you can see a comparison between the current algorithm (ILP) and our algorithm (Branch). For each location, the best time is made bold, and if there is a difference in the solution value, the lower cost has also been made bold. What we can see from these results is that the branch and price works much better for small locations; however, for large locations branch and

price finds solutions much worse while taking a lot more time than the current algorithm. Finally, in Table 15, you can see the distribution of time spent on the branch-and-price. The column total displays the total amount of time spent on the instance, the column root node displays the time spent on the first node of the algorithm, and the columns Col Gen and LP show how much time was spent on creating additional columns and solving the LP after adding them, respectively. Note that the columns Col Gen and LP are calculated over all the nodes, including both the root node and subsequent nodes. Additionally, you can see how often the different branching rules are applied. What we can see from this is that for smaller instances, column generation takes a very large portion of the time. For the larger instances, by far, the majority of the time was spent on solving the LP.

Table 14: Comparison of results between branch and price and the current in-use ILP technique

Location	Branch		ILP	
	time (ms)	cost	time (ms)	cost
Arnhem	**20**	702986	79	702986
Deventer	**333**	908124	680	908124
Nijmegen	**65**	2369	218	2369
Eindhoven	**3110**	**25257**	10280	25557
Groningen	**157**	5070	884	5070
Utrecht	**6646**	**6739233**	150324	6739333
Alkmaar	**537**	5664450	611	5664450
Zutphen	**56**	1112780	348	1112780
Enschede	1019	803327	**594**	803327
Zwolle	**309**	419660	828	419660
Amersfoort	9170	107801	**7020**	107801
Leeuwarden	**478**	229524	1240	229524
Rotterdam	900723	1489365	**133463**	**1210606**
Den Haag	900000	X	**322711**	**1680635**

Table 15: Comparison of results between branch and price and the current in-use ILP technique

Location	Time spent(ms)				Branching Rules		
	Total	Root node	Col Gen	LP	Synchronization	Single	Breaks
Arnhem	20	17	4	1	0	0	0
Deventer	333	298	178	99	0	0	0
Nijmegen	65	60	33	4	0	0	0
Eindhoven	3110	1490	1995	150	1	0	0
Groningen	157	154	101	4	0	0	0
Utrecht	6646	4259	3456	680	1	0	7
Alkmaar	537	276	410	60	6	0	0
Zutphen	56	53	46	3	0	0	0
Enschede	1019	1000	496	435	0	0	0
Zwolle	309	308	211	41	0	0	0
Amersfoort	9170	8303	6022	3023	0	1	1
Leeuwarden	478	471	430	40	0	0	0
Rotterdam	900723	414843	50373	828681	7	0	11
Den Haag	900000	895884	21578	874191	0	0	0

6 Conclusion and Future Work

We have formally introduced the Shunting Yard Driver Scheduling Problem. This is a problem currently solved by the NS, which is in need of a more efficient algorithm. We have formulated the problem as a set partitioning problem with side constraints and developed a branch-and-price solution algorithm. Our branch-and-price algorithm contains a pricing problem where we have to find schedules for individual drivers. We have adopted a dynamic programming approach to generate these columns. To increase the speed of the column generation, we allow schedules that contain a job more than once. To avoid this in the final solution, this is addressed implicitly by the master problem, which will likely choose schedules in the end that only contain each job once due to converging reduced costs. To ensure the starting times for jobs are consistent across the selected columns, we branch on the time windows of jobs. Additionally, a similar branching rule ensures that synchronized jobs have equidistant starting times. Finally, we use a flow algorithm to create a final solution that does not contain fractional schedules. Several acceleration techniques have been developed to further speed up the algorithm, each with advantages and disadvantages. The algorithm is experimented with using many different parameter combinations for the acceleration strategies to discover the combinations that provide the best balance between their ups and downsides.

To test the algorithm, we use real-life instances for shunting yards in the Netherlands provided by the NS. To compare the algorithm, we tested it against the currently used algorithm by the NS. We have found that the branch-and-

price algorithm is much faster when used for small instances. We can find a solution marginally faster and sometimes with a slightly lower cost for nine of the eleven smaller instances. For the remaining two instances, we can find solutions nearly as fast. However, branch-and-price gets outperformed for the largest instances. We cannot solve one of the largest two instances within a time limit of fifteen minutes, and we cannot find a competitive solution in both time and cost for the other large instance when compared to the currently used algorithm. Unfortunately, this means it is impractical for actual use, as better performance on larger instances is more critical than performance on smaller instances.

In future research, a couple of areas of improvement could be made. One critical area lies in the optimization of the tail end of the master problem. The traditional strategy of prematurely halting column generation proves ineffective in this problem. Therefore, we need to continue creating columns until no negatively reduced cost can be found. This results in an extreme slowdown at the end, where barely any columns improve the solution cost. This makes it very difficult to tackle larger problems where too much time is spent on columns that do not improve the objective. To tackle this, looking at stabilized column generation techniques that result in a more linear improvement of the solution could be usable.

One other possibility is to look at more elaborate ways to choose a time window and time to branch on. The article Bredstrom and Ronnqvist (2007) shows a way to branch in the most optimal way. In practice, however, this will likely not cause a large enough improvement as, for the larger instances, the algorithm is already too 'slow' before even reaching a branching point. If it is possible to improve the efficiency before reaching a branching point, it could be a way to improve the algorithm even further.

Another issue that might currently slow down the algorithm is the master problem LP slowing down once a lot of columns have been added. The first idea would be to try to remove columns with a high reduced cost to maintain fewer columns. In practice, this did not work very well, as removing these columns required many more iterations to find additional columns. In other words, the speedup of solving the LP was not enough to balance the need for additional iterations. A potential cause for the large number of discovered columns is likely that it finds columns that contain a job more than once. This can be seen in the effect of the look back parameter. This parameter usually speeds up the algorithm despite not necessarily finding optimal columns (within the dynamic programming approach). Another possible way to do this would be to increase the number of dimensions of the DP, keeping track of more previous jobs. Once a number of iterations have passed, jobs become unlikely to appear more than once in a schedule. From then on, returning to the current DP dimensions would be beneficial.

Another area that may be improved upon is the breaks. For larger instances, breaks are the jobs that need to be branched on by far the most. It may be worth trying to discover techniques that can look at ways to unify the break times to avoid having to branch on them.

References

Ait Haddadene, S. R., Labadie, N., & Prodhon, C. (2016). A grasp × ils for the vehicle routing problem with time windows, synchronization and precedence constraints. *Expert Systems with Applications*, *66*, 274–294. https://doi.org/10.1016/j.eswa.2016.09.002

Boschetti, M. A., Mingozzi, A., & Ricciardelli, S. (2004). An exact algorithm for the simplified multiple depot crew scheduling problem. *Annals of Operations Research*, *127*(1-4), 177–201.

Bräysy, O., & Gendreau, M. (2005). Vehicle routing problem with time windows, part i: Route construction and local search algorithms. *Transportation science*, *39*(1), 104–118.

Bredstrom, D., & Ronnqvist, M. (2007). A branch and price algorithm for the combined vehicle routing and scheduling problem with synchronization constraints. *NHH Dept. of Finance & Management Science*, *Discussion Paper No. 2007/7*(1), 55–68. https://doi.org/http://dx.doi.org/10.2139/ssrn.971726

Chen, S., Shen, Y., Su, X., & Chen, H. (2013). A crew scheduling with chinese meal break rules. *Journal of Transportation Systems Engineering and Information Technology*, *13*(2), 90–95. https://doi.org/10.1016/S1570-6672(13)60105-1

Dohn, A., Rasmussen, M. S., & Larsen, J. (2011). The vehicle routing problem with time windows and temporal dependencies. *Networks*, *58*(4), 273–289.

Ernst, Jiang, Krishnamoorthy, & Sier. (2004). Staff scheduling and rostering: A review of applications, methods and models. *European journal of operational research*, *153*(1), 3–27.

Gottenbos, M. (2022). *Balancing costs and driver satisfaction in cargo train driver scheduling* (Master's thesis). https://studenttheses.uu.nl/handle/20.500.12932/41930

Haitam, E., Najat, R., & Jaafar, A. (2021). A survey of the vehicle routing problem in-home health care services. *Proceedings on Engineering*, *3*(4), 391–404.

Hanafi, R., & Kozan, E. (2014). A hybrid constructive heuristic and simulated annealing for railway crew scheduling. *Computers & Industrial Engineering*, *70*, 11–19. https://doi.org/10.1016/j.cie.2014.01.002

Heil, J., Hoffmann, K., & Buscher, U. (2020). Railway crew scheduling: Models, methods and applications. *European Journal of Operational Research*, *283*(2), 405–425. https://doi.org/10.1016/j.ejor.2019.06.016

Kasirzadeh, A., Saddoune, M., & Soumis, F. (2017). Airline crew scheduling: Models, algorithms, and data sets. *EURO Journal on Transportation and Logistics*, *6*(2), 111–137. https://doi.org/10.1007/s13676-015-0080-x

Li, J., Qin, H., Baldacci, R., & Zhu, W. (2020). Branch-and-price-and-cut for the synchronized vehicle routing problem with split delivery, proportional

service time and multiple time windows. *Transportation Research Part E: Logistics and Transportation Review*, *140*, 101955.

Liu, R., Tao, Y., & Xie, X. (2019). An adaptive large neighborhood search heuristic for the vehicle routing problem with time windows and synchronized visits. *Computers & Operations Research*, *101*, 250–262. https://doi.org/10.1016/j.cor.2018.08.002

Liu, W., Dridi, M., Fei, H., & El Hassani, A. H. (2021). Hybrid metaheuristics for solving a home health care routing and scheduling problem with time windows, synchronized visits and lunch breaks. *Expert Systems with Applications*, *183*, 115307.

Rasmussen, M. S., Justesen, T., Dohn, A., & Larsen, J. (2012). The home care crew scheduling problem: Preference-based visit clustering and temporal dependencies. *European journal of operational research*, *219*(3), 598–610.

Shen, Y., Peng, K., Chen, K., & Li, J. (2013). Evolutionary crew scheduling with adaptive chromosomes. *Transportation Research Part B: Methodological*, *56*, 174–185. https://doi.org/10.1016/j.trb.2013.08.003

Szabó, K. (2023). *Scheduling mechanics on a shunting yard: Skills, synchronization and train movements* (Master's thesis). https://studenttheses.uu.nl/handle/20.500.12932/43524

Van den Akker, M., Hoogeveen, H., & van Kempen, J. (2012). Using column generation to solve parallel machine scheduling problems with minmax objective functions. *Journal of Scheduling*, *15*, 801–810.

Van den Broek, R. (2022). *Towards a robust planning of train shunting and servicing* (PhD thesis). Utrecht University.

Van den Broek, R., Hoogeveen, H., & Van den Akker, M. (2020). Personnel scheduling on railway yards. *20th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2020)*.

Van den Broek, R., Hoogeveen, H., Van den Akker, M., & Huisman, B. (2022). A local search algorithm for train unit shunting with service scheduling. *Transportation Science*, *56*(1), 141–161.

van den Akker, J., Hoogeveen, J., & van de Velde, S. (1996). *Parallel machine scheduling by column generation (april 1996 workshop). technical publication* (tech. rep.). National Aerospace Lab., Amsterdam (NL); Univ. Catholique de Louvain ...

van Twist, R., van den Akker, M., & Hoogeveen, H. (2021). Synchronizing transportation of people with reduced mobility through airport terminals. *Computers & Operations Research*, *125*, 105103.

Verhave, M. M. (2015). *Column generation with a resource constrained shortest path algorithm applied to train crew scheduling.* (Master's thesis). https://thesis.eur.nl/pub/32478/Verhave.pdf

Zamorano, E., & Stolletz, R. (2017). Branch-and-price approaches for the multiperiod technician routing and scheduling problem. *European Journal of Operational Research*, *257*(1), 55–68. https://doi.org/10.1016/j.ejor.2016.06.058