Universiteit Utrecht

Department of Information and Computing Sciences

Scheduling mechanics on a Shunting Yard: skills, synchronization and train movements

Master's Thesis

Author:
G. C. (Kees) Szabó
Utrecht University

First Supervisor:
dr. J.A. (Han) Hoogeveen

Second Supervisor:
dr. ir. J.M. (Marjan) van den Akker

Supervisor NS:
dr. R.W. (Roel) van den Broek

January 23, 2023

Abstract

When trains are not in use, they reside on shunting yards. On these shunting yards, the trains move around following a predetermined shunting plan to ensure an efficient use of the shunting yards. The trains have to be checked and small repairs have to be carried out by mechanics working at the shunting yard. Obviously, tasks cannot be carried out when a train is moving. Therefore, trains can have multiple moments at different times and locations where they are stationary and a mechanic is allowed to work on them. As some tasks require certain skills, there are restrictions on which mechanics can do which tasks. Some of these tasks require two mechanics to work on at the same time, and therefore the time when these mechanics start working on this job has to be synchronized. At the moment, scheduling is done ad hoc by the mechanics at the start of their shifts. The manual planning takes time out of a possibly busy schedule while not always resulting in a desirable, robust solution. Therefore, in this thesis, we propose a local search approach for scheduling mechanics on a shunting yard subject to these constraints. We aim to create robust schedules for the mechanics which can be generated before their shifts start and are easy to update if a disruption in the shunting plan were to occur. Our method is able to generate good results on a variety of realistic problem instances.

Contents

Chapter 1

Introduction

The Nederlandse Spoorwegen[1] (NS) is responsible for a large part of the passenger transport by train in the Netherlands. To guarantee a smooth and safe journey for all passengers, making sure the trains are well-maintained is of the upmost importance. While large maintenance such as wheel replacement require the train to be taken out of service for some time, small maintenance and safety checks need to happen in between rides when the train is temporarily parked at a shunting yard.

One might be quick to compare shunting yards for trains to what parking lots are for cars. However, the only similarity between a shunting yard and a parking space is that they accommodate vehicles to be stationary for a while. In a car park, cars are free to drive around at the roads between the parking spots. At shunting yards, the same rail infrastructure which is used for the trains to drive on can also be used to park the trains. The dual usage of tracks to both park and move trains leads to a complicated puzzle when deciding which trains are parked at which spot as a train which is required to leave might be blocked by another one. At the shunting yards, the trains also need to be washed at the trainwash which is located somewhere at the yard which means that even if a train could be parked at one spot, it still has to move during its duration at the shunting yard. The problem of deciding how to move the trains around the shunting yard in order for them to be cleaned and ready to leave at the right time is studied and solved by Roel van den Broek (2022); see also Roel van den Broek et al. (2022).
This thesis picks up from the moment Roel's solution has been generated. Given the movements of the trains, a set of jobs, and a crew of mechanics, how can we create a robust schedule for each mechanic?

As trains only spend a few hours at the shunting yard, each (maintenance) job which has to be done has a fixed time limit where the train is available. Furthermore, the trains move around on the shunting yard, thus splitting one big time window at one location for a job into multiple smaller time windows at different locations. The jobs have to be carried out by mechanics, but certain jobs require certain skills, which not mechanic possesses. As an example, the mechanic who is qualified to replace broken furniture may very well not have the qualification to work on the high-voltage systems which power the train. Some jobs may require two mechanics to work on them at the same time; if this is the case, start times of these mechanics for such a job have to be synchronized. We refer to the problem of scheduling the mechanics at a shunting yard while taking skills, train movements, and synchronization as the Shunting Yard Mechanic Scheduling Problem (`SYMSP`).

[1] Dutch railway company for passenger transport

In this thesis, we present a solution method for the Shunting Yard Mechanic Scheduling Problem (`SYMSP`) which we describe in Chapter 2. Note that as the number of mechanics assigned to each shift is predetermined along with their skill sets, this thesis focuses on the assignment of jobs to the mechanics within their shifts. It is of course possible to run the algorithm with a different assignment of mechanics to see if the change in results justifies assigning more, fewer, or other mechanics to one or more shifts.

In Chapter 3 we look at related problems described in the literature, and we explore a variety of solution methods for these problems and their possible usefulness for the `SYMSP`. Our quality metric used to determine the quality of a given solution is explained in Chapter 4 after which we introduce the exact and heuristic solution methods in Chapter 5 and Chapter 6 respectively. In Chapter 7, we test our model's ability to generate new solutions if disruptions occur when the generated schedule is already partially executed. Chapter 8 introduces the problem instances we try to solve along with a brief description of their characteristics. The results and our interpretation of them can be found in Chapter 9 after which we will conclude this thesis with our conclusion and a look into the future in Chapter 10.

Chapter 2

Problem Description

In our paper we aim to find a solution method for the Shunting Yard Mechanic Scheduling Problem (SYMSP). In this problem, trains arrive at a shunting yard and move around this yard in a predetermined route. Before leaving the shunting yard, the train must have undergone some checks and possibly small repairs. These repairs and checks are carried out by a set of mechanics which work 8-hour shifts each day with three consecutive shifts each day. The mechanics have to be assigned to jobs (consisting of checks and repairs) to which they travel from their previous job. Once started, a mechanic has to finish the job; preemption of jobs is not allowed. Some jobs require certain skills which lead to a subset of the total set of mechanics being allowed to service them. We assume that for the entire duration of each shift, there is at least one mechanic present with the skills to do any job that must be done in that shift. Some jobs require two mechanics to work on them at the same time. In that case, the mechanics need to synchronize their start-times of that job.

In a mechanic's schedule, there can be periods where the mechanic is not working nor moving. These periods of 'free' time might seem like wasted productivity, but they are very useful to prevent disruptions from cascading through the entire schedule. These periods of time give some buffer where a job can be extended into when a train arrived late, or when a job takes more time than initially planned. Therefore, we call all the time in between jobs where a mechanic is not working or walking the total available buffer time B_{total}. Our overall goal is to create a robust schedule for each mechanic by distributing B_{total} over the buffers as evenly as possible between the scheduled jobs in the schedule. Each job b, except for the first job, has a buffer protecting it from delays during the previous job a in the schedule which is defined as the start time of job b minus the start time and duration of job a minus the travel time from the location of job a to the location of job b. When talking about job b's buffer from the previous visit, we refer to it as the buffer of job b. For all intents and purposed, it could also be referred to as the buffer of job a, but we choose to call if the buffer of b as it protects job b's placement. An illustration of the buffer in front of a job b can be seen in Figure2.1.

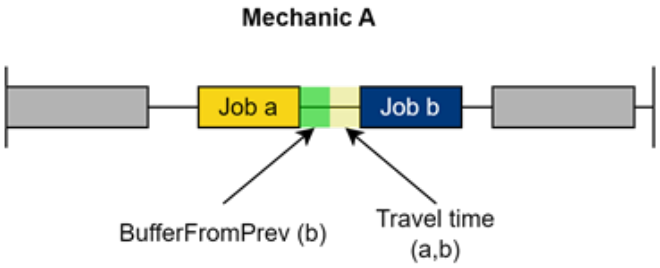

Figure 2.1: The buffer (green area) for job b in the schedule of mechanic A.

When mechanics get to work with a schedule we have generated, they are free to completely ignore the buffers. After finishing one job, they are free to move onto the next job in their schedule and start as soon as its time window opens. This not only prevents wasting time, but it also increases the buffers from that job on.

The `SYMSP` is specific to railway applications. However, at the core, this problem is a generic vehicle routing problem with additional components. In the remainder of this chapter, we will look at the generic components which together form the problem at hand when taken in the context of mechanic scheduling for shunting yards. Along with the description of the components, there is a short summary of the challenges arising from including each component. We will include an example of additional real-world uses for each component. The notations we will use when referring to the literature can be found in Table 2.1.

Notation	Description
M	set of all vehicles/mechanics/caregivers $m \in M$
J	set of all jobs/tasks/places to visit $j \in J$
S	set of all skills $s \in S$
$Q_{m,s}$	$\|V\| \times \|S\|$ matrix containing the level of each skill $s \in S$ for each $m \in M$
$R_{j,s}$	$\|C\| \times \|S\|$ matrix containing the required level of each skill $s \in S$ for each $j \in J$
$T_{h,i}$	travel time from $h \xrightarrow{to} i$ with $h \in J, i \in J$
t_j	the time when the visit to $j \in J$ starts
$U_{h,i}$	indicator if visits to h and i are synchronized with $h \in J, i \in J$
W_j	time window for each $j \in J$ spanning the period $W_j(open) \rightarrow W_j(close)$
d_j	duration it takes to complete $j \in J$
S	set of all skills $s \in S$

Table 2.1: An overview of the notation used in our paper.

2.1 Vehicle routing problem (VRP)

Package delivery companies are in the business of getting the packages from the main depot to the customers. In order to maximize profits, these companies want to reduce costs. As the main sources of operational costs are the driver's salaries and fuel expenses, the companies want to find

the shortest routes which cover all clients. The problem of optimally routing a set of vehicles so that all locations are visited is called the vehicle routing problem (VRP).
In the VRP the goal is to route a set of m vehicles[1] M from a depot such that they visit the locations in J once whilst minimizing the time/distance travelled. The purpose of a visit is application-specific and can include things such as delivering a product to a consumer or providing a certain service like cleaning the windows. Another example of a VRP-based problem is the Home Healthcare (HHC) scheduling problem where nurses travel to the homes of patients to deliver the required care.
The times which need to be taken into account for the VRP are the travel time to location i from the vehicle's previous position h (either another location or the depot) denoted as $T_{h,i}$ and the service time d_j for each location/job $j \in J$ which indicates how long the vehicle has to spend at the location.

In the case of `SYMSP`, the vehicles are not the trains, but the mechanics as they need to visit the trains at certain locations to perform the necessary tasks. We assume that in the `SYMSP`, the jobs can have varying predetermined durations denoted by the service time d_j for each job $j \in J$. Trains can be located at varying locations on the shunting yard, therefore, the travel time between jobs varies based on the locations of the jobs.

2.2 Time windows (TW)

When delivering packages, you cannot deliver (and especially install) them if the customers are not available at the location. As customers are not always available, you make an appointment for a certain period of time in which the delivery can occur. The same goes for the maintenance tasks at the shunting yard: the train is only available at certain locations for a limited amount of time.

These periods where a visit to a location can occur are called *Time Windows*. There are many application's of the `VRP` where visits have a visit-specific time window defined by $W_j[minStart, maxEnd]$, during which the visit is allowed. A vehicle executing the visit may arrive before the start of the time window and wait until it commences, but it must have visited (and spent the service time at) the location before the end of the window.
In the *VRP*, any solution is feasible (albeit possibly terrible) as it does not have any restrictions. The introduction of time windows imposes a restriction which can render solutions infeasible which complicates the problem. The added restrictions also add complexity to whichever model is used to solve the problem.

In the `SYMSP`, an additional challenge arises from the use of multiple time windows for the same job. As a train may be available at different locations during its time on the shunting yard, the job has to be executed at one of the possible locations which adds the complexity of choosing an option instead of just determining the time.

[1] Vehicles are entities which can move autonomously, the type of entity (i.e. Vehicle driven by a human, self-driving vehicle, UFO, chimpanzee on a tricycle) does not matter for the definition.

2.3 Skills (S)

Skills are a common addition to VRP-based problems. In our package delivery example, a package may need to be delivered in a zone with strict environmental regulations for vehicles. It is highly likely that only a subset of the delivery company's vehicles meet the standard, so there will be a constraint restricting the set of vehicles which may deliver the package. In the HHC example mentioned earlier, certain actions such as administering medicine require additional certifications which not all nurses will have.

These constraints on which vehicle may perform a visit are generally referred to as constraints on *Skills*. Instead of allowing each vehicle to perform any visit, each vehicle has its own properties which need to at least meet the requirements of a location in order to be allowed to visit that location. The addition of skills does not only complicate the model by adding many parameters and constraints, it also fundamentally complicates the process of finding a solution as many previously feasible solutions will become infeasible due to the vehicles not containing the right skills for their assigned visits. Another complication is that without skills, vehicles could be assumed to be homogeneous which allowed for (relatively) easy decision-making. When considering a cost-minimizing objective, homogeneous vehicles cost the same to operate and therefore the minimization of costs becomes a minimization of the time and distance travelled. Heterogeneous vehicles on the other hand can have different costs which are often correlated with the skill level. Minimizing the total cost with heterogeneous vehicles requires these costs to be taken into account. An example is a situation where two locations are near each other but a fairly long distance from the others. In the homogeneous case, it would make sense (if possible) to have one vehicle visit both back-to-back as this minimizes the travel time. In the heterogeneous case, if one of the locations requires a high-skilled visit vehicle the other does not require any skill, it may be cheaper to have an extra travel action for a low-skilled vehicle to service the skill-less visit instead of having the high-skilled vehicle perform it. A vehicle $m \in M$ can perform a visit to a location/job $j \in J$ as long as its level $Q_{m,s}$ for each skill $s \in S$ is the same as or higher than the level required by j defined as $R_{j,s}$ for each respective skill.

In the `SYMSP`, skills are present. While some jobs require no additional skills, jobs such as working on the high voltage equipment or safety systems can require certain skills/qualifications. As the availability of the mechanics is predetermined and the focus lies on creating a robust schedule, we do not take varying costs such as higher wages for higher-skilled mechanics into account. A mechanic is deemed qualified to serve a job if his skill levels for each skill at least satisfy the required level of that skill.

2.4 Synchronization

Sometimes when delivering/installing packages, the installation can be a two-person job. The intuitive solution would be to have two people travelling together in the same van delivering packages together. However, this method is rather wasteful of the second person's time and company money when not all packages require multiple people. A better solution would be for people to travel separately and then have them meet at the customer location where multiple people are required. This is quite similar to the nurses travelling by themselves and meeting at a patient's home when that patient's care requires multiple nurses. Requiring vehicles to meet at a location at the same time is generally referred to as a *synchronization constraint*. When scheduling mechanics, some jobs can require two mechanics. This can be due to labour rules about lifting heavy objects, or the

nature of the job which makes it impossible for just one person. For these jobs, mechanics should be present at the right location at the same time to do the job together. Synchronization of more than 2 mechanics is outside of the scope of the `SYMSP` as this is not required by the NS.

Adding synchronization to the `SYMSP` adds a new layer of complexity. Without synchronized visits, vehicles are independent of each other, that is, a change in the schedule of one vehicle does not directly impact the schedules of other vehicles. With the introduction of synchronization comes the addition of vehicle interdependence and changing the schedule of one vehicle m_a could mean that another vehicle m_b executing a visit synchronized to a visit of m_a also has to be rescheduled which in turn might require other vehicles to change their schedules, etc.

Chapter 3

Literature

In this chapter, we explore the literature on the SYMSP and the related problems along with their solutions. This paper is not a literature survey, therefore we will only mention relevant problems, papers, and details in a possibly non-exhaustive manner.

3.1 Related fields

Before diving into the existing literature about problems related to the SYMSP, we take a look at the settings in which they are frequently studied. Some of these settings/problems have already been mentioned in Chapter 2. These problems are all based on finding the best sequences to perform jobs which may have some additional constraints.

3.1.1 Home Healthcare

The scheduling of heterogeneous vehicles with skills to visit time-window constrained locations has been thoroughly studied in Home Healtcare (HHC) settings; see for example Haitam et al. (2021). The scheduling and routing of HHC caregivers to patients closely matches the SYMSP. Patients have to be helped at certain times like in the morning when getting dressed, this is where the time windows come in. The synchronization is present when patients need more than one caregiver to complete a certain task. Skills are present as well as a caregiver might need certain skills/certifications to get a patient out of bed and administer some medication whereas household help requires few to no qualifications. Unlike the trains in the SYMSP, patients receive their care at a predetermined location with one single time window for each care task which has to be done. Furthermore, costs are a big factor in HHC settings whereas the focus of the SYMSP lies on robustness.

3.1.2 Airport operations

In aviation, scheduling is a vital part of operations. Each minute an aircraft is not flying is a minute where it is not generating profits. When an aircraft arrives at an airport, a complicated series of operations in a specific order using specialized vehicles and workers has to occur within the period that the aircraft is at the airport; see for example Fink (2016). This problem is similar to the SYMSP as they both deal with relatively short time windows where a set of specialized tasks has to be done before the plane/train can be put back into service. However, while the location of a plane

is not known days in advance, once it has landed, all operations tend to happen at the same gate with just one time window per operation.

3.1.3 Technician routing and scheduling

The observant reader has by now noticed that the terms *vehicle* and *location* cover a wide variety of terms. A location can be an aircraft, train, patient, or in the case of technician routing and scheduling: a place where maintenance/hardware installation is needed. The SYMSP is closely related to the technician routing and scheduling problem described by Pillac et al. (2011). For any repair/installation, certain tools are required which are not present in every technician's toolbox, just like specific knowledge and certifications are dependent on the job. Jobs sometimes need multiple people to be completed thus requiring synchronization. Time windows can be present in the form of deadlines and locations being open for the technician to work on the job (e.g. people being home).

3.2 Related literature

As mentioned, the NP-hard VRP forms the core of the SYMSP . Over the last two decades, the extensions to the VRP which are present in the SYMSP described in Chapter 2 have been introduced in papers which each focused on a subset of these extensions.

When looking at the *synchronization constraints*, Paraskevopoulos et al. (2017) and Haitam et al. (2021) conclude that our version described in Chapter 2.4 has not been widely studied yet. Our definition of synchronization is based on the work of Bredstrom and Rönnqvist (2007) and Rasmussen et al. (2012) where nurses need to provide synchronized visits in HHC applications. However, other types of synchronization exist in the literature as well. Instead of synchronizing visits, operations and movements can also be synchronized. One such problem, described by Van Twist et al. (2021), is trying to find a way to move passengers with reduced mobility around airports where passengers have to be handed over (which needs to be synchronized to minimize waiting times) and passenger's transports can be synchronized to allow for better efficiency. Drexl (2012) states that synchronization of movements can also arise from a difference between trucks and trailers where a trailer can only be moved when attached to a truck. Drexl distinguishes between different types of synchronization: The aforementioned synchronization of visits and handover of passengers fall under *operation synchronization* whilst the (non)-autonomous vehicles and passengers moving together fall under *movement synchronization*. The other types of synchronization mentioned by Drexl (regarding load and resources) are not relevant to the SYMSP discussed in our paper. Synchronization of load and resources can be very relevant when cargo transportation is involved which is not the case in the SYMSP considered in our paper. Movement synchronization can be interesting if non-autonomous vehicles which need to be handed over between autonomous vehicles are considered. However, this is a very niche application. For non-autonomous vehicles which do not have to switch between their autonomous vehicle, multiple solution methods exist. These methods include building teams consisting of multiple vehicles which travel together, and grouping vehicles in a team as a single vehicle which is used as a regular vehicle when planning as described by Zamorano and Stolletz (2017).

When discussing synchronization, the literature sometimes mentions temporal dependencies as well. Temporal dependencies put restrictions on the order of operations and the time between these operations. Sometimes, temporal dependencies are also called precedence constraints if the order of jobs matters but the time between them does not. While not relevant for the SYMSP, temporal

dependencies are sometimes used to discuss synchronization and thus deserve to be mentioned.

Unlike the synchronization, the skills component of the SYMSP is more common in the literature. Golden (1975) describes an extension to the VRP where the fleet of vehicles V consists of multiple vehicles, each with their own properties which at the time mostly came down to different load capacities. This can be seen as the introduction of skills in the literature. In more recent studies, there are multiple properties which a vehicle might need to comply with in order to be able to execute a certain visit like environmental restrictions that apply at the location which have become more abundant in recent years like in the city of Gemeente Utrecht (2021). Our skills example in Chapter 2.3 is derived from Fikar and Hirsch (2017) where they study nurse scheduling in a HHC setting with requirements on nurse qualifications. Pillac et al. (2011) studies the technician routing and scheduling problem where the technicians have different qualifications. Cappanera et al. (2020) show that apart from certifications and procedural skills, jobs can also require the presence of some tool/special device which may not be present in every vehicle.

In the literature, skills are modelled both with as well as without levels (Zamorano & Stolletz, 2017). Skill requirements without levels are usually modelled by having an indicator for each skill in a $skill \times vehicle$ matrix indicating if it possesses said skill and a $skill \times location$ matrix indicating which skills each location requires. Whilst simple, this method has the drawback of having to add $|S| * (|J| + |M|)$ parameters when a skill (i.e. being certified to work at certain heights) has multiple gradations (i.e. grade 1 for anything up to 10 meters, grade 2 for anything up to 25 meters etc.). As each skill indication only represents the skill being present/required, each level will require a new skill. Using a representation with skill levels allows for skills to have gradations without having to expand the number of parameters and constraints. As skill levels are present in the real world and are (probably) not any harder to implement than regular skills whilst possibly reducing the number of variables, we use the representation of skills with levels in our problem. Yan et al. (2019) described how the variable costs for workers with different skills can be included in the problem formulation.

3.3 Solution methods for similar problems

Multiple solution methods, both exact variants as well as heuristics or hybrids[1], have been proposed for problems related to the SYMSP such as the problems described earlier in this chapter. In the remainder of this chapter, we explore the solution methods that exist, the possible applicability of the solution method to the SYMSP described in Chapter 2, and the performance of the methods [2]. Not all proposed solution methods for related problems will be discussed in this chapter. For an extensive overview we refer to Paraskevopoulos et al. (2017). This chapter serves to explore valuable strategies which can be included in the solution method for the SYMSP.

3.3.1 Exact methods

Dohn et al. (2011) present a Branch-And-Price solution approach for optimally solving the capacitated vehicle routing problem with time windows and temporal dependencies requiring a minimum amount of time between some jobs. The proposed model is tested with three slightly different formulations on a subset of the well-known benchmark set by Solomon (1987) with added temporal

[1] Heuristics using an exact method somewhere in their solution method.
[2] The performance is taken from the papers in which the methods are described.

dependencies. The problem of finding the minimal total travel cost is turned into a set-partitioning master problem with a shortest path subproblem using a Dantzig-Wolf decomposition where the master problem consists of selecting the best combination of routes which are generated in the subproblem. In their model, they propose an interesting branching strategy which is adapted in another paper published in the same period by the same authors, namely Rasmussen et al. (2012). The proposed method uses branching on reducing time window sizes of visits with precedence constraints. With the reduced time windows, they enforce the precedence constraints. For a more detailed explanation, we refer you to Rasmussen et al. (2012). While their approach is not entirely applicable to the `SYMSP` as we do not have precedence constraints, using time windows to deal with synchronization may be an interesting tactic to employ. Apart from the precedence constraints, integrality is also handled in the branching without further explanation as to how. The method proposed by Dohn et al. (2011) manages to find a feasible (optimal) solution for the instances with 50 locations in about 69.7% (83% if the temporal dependencies are only of the synchronization type) of cases on average before the time-out of one hour.

Rasmussen et al. (2012) focuses on the HHC application with soft preference constraints indicating the patients' preferences for the nurses. The used method and branching strategy are very similar to Dohn et al. (2011) but differ in a few ways. Now, visits are allowed to be uncovered at a penalty cost as, according to the authors, these uncovered visits can be dealt with manually. Unlike Dohn et al. (2011), integer branching to handle fractional visits is an essential part of the branching strategy. While trying to solve the problem to optimality, Rasmussen et al. (2012) also introduces two heuristic improvements for the processing speed based on pre-possessing for larger instances. Their first heuristic improvement is visit clustering based on preferences. By clustering patients into groups and having caregivers only operate in certain groups, large problems which were previously unsolvable can be solved. Their second prepossessing improvement, the removal of idle arcs, aims to eliminate travel arcs where the minimum amount of time a caregiver has to wait for the next time window to open is significantly high with regard to the travel time. These heuristic prepossessing steps are an example of hybrid methods which we discuss in Chapter 3.3.3. The authors state that although proof of optimally may be sacrificed, in good solutions high idle times would probably be a rare occurrence.

In Rasmussen et al. (2012), a combination of real-world and generated data was used to benchmark their model. Without clustering, the largest instance for which they could find a feasible result within one hour had 16 caregivers, 80 patients and 16 synchronized visits (or 7 caregivers, 60 patients and 18 temporal dependencies which were not merely synchronized visits). This is similar to the results of the hybrid method by Bredström and Rönnqvist (2008) on the same instances[3], but the time needed to reach the solution in Rasmussen's case was significantly less. When enabling clustering, the largest instance for which a feasible solution was found had 15 caregivers, 150 patients, and 46 temporal dependencies. Removing idle arcs provided a significant speedup to the average running time over the test cases. Unfortunately, removing idle arcs clashes with our objective to include significant buffers to create a robust solution for the `SYMSP`. Although preferences are omitted in the `SYMSP`, clustering may be interesting when applied to skills. The results by Rasmussen et al. (2012) also show the difference between different clustering schemes where, depending on the scheme, uncovered visits may be added to additional clusters. These clustering schemes are not applicable to the `SYMSP` as skills are hard constraints, unlike the soft preference constraints.

[3]The same instances were used, but Rasmussen et al. (2012) doubled the number of synchronized visits

3.3.2 Non-exact methods

As seen in Chapter 3.3.1 and in the overview by Paraskevopoulos et al. (2017), exact solution methods for problems similar to the SYMSP in type and complexity seem to have a limit of around 100 jobs which can, with some efficiency improvements, be stretched to around 150 jobs at most. This performance is not terrible, but one can think of many situations where a solution method capable of handling more locations is desirable. Not only does enabling more locations let you handle larger instances with the immediate benefit of scalability for larger organisations, but it also allows for planning over a longer period. Unlike exact methods, approximation methods are oftentimes capable of finding relatively good solutions for large instances and are therefore worth considering. Results generated by a metaheuristic strategy may not be provably optimal, but optimality is also sacrificed for efficiency with the idle arc removal optimization in the exact method by Rasmussen et al. (2012) which turns it into a heuristic method whilst, on average, hardly impacting the solution quality.
Recent surveys by Paraskevopoulos et al. (2017) and Haitam et al. (2021) showed only one heuristic solution method for a problem resembling the SYMSP which was proposed by Redjem and Marcon (2016). Their heuristic for Home Healthcare scheduling involves generating all possible routes for each caregiver, selecting the fastest route for each one and adjusting the routes to correctly cover all patients and adhere to the temporal dependencies. The caregivers are divided into multiple subsets which each start at their own location and can serve patients within a certain radius of that location. According to the authors, this is possible as they claim that the number of patients visited per caregiver each day is at most 9. However, this is not an assumption which can be made for the SYMSP. Their proposed strategy also requires $\#Locations!$ iterations which they found to become problematic for more than 15 caregivers which is logical as calculating all possible routes for each caregiver is quite a slow process.

Metaheuristic approaches are more common in the literature and predominantly focus on neighbourhood searching algorithms according to the surveys. Polnik et al. (2021) proposed a three-stage solution method for a VRP with time windows, skills and synchronization. At each stage, an initial solution is generated and then the Google OR-tool framework (using a combination of local search, large neighbourhood search, and constraint programming) is used to optimise the result. Their method first tries to find an initial feasible solution by following the same strategy used by human planners: creating pairs of two workers which maximizes the number of skills and working hours the workers share in common and assigning them to as many synchronized visits as possible. Maximizing the skill overlap is useful in their algorithm as in their problem, synchronized visits require the same set of skills. It is noteworthy that this assumption does not hold for the skills part of the SYMSP described in Chapter 2.3. In the second stage, the result of the first stage is used as a starting point to plan all visits. First, the teams are split whilst keeping their routes after which the Google optimization algorithm is run with operations for both synchronized as well as unsynchronized visits enabled to try to plan all the visits. Finally, the third stage uses a guided local search algorithm which adds an adaptive penalty per used arc on a route. At each step, the Google optimization is used to determine the new solution. This method was able to solve instances with up to 70 caregivers and around 600 visits (split 380-220 between unsynchonized and synchronized) all within one hour and significantly better than the results obtained by human planners. Whilst the solution approach is not directly applicable to the SYMSP, the idea of a multi-stage approach to plan the most restricted visits first is a possibly interesting idea.

Another combination of metaheuristics can be found in the $GRASP{\times}ILS$ method proposed by

Haddadene et al. (2016) to solve a VRP with time windows and temporal dependencies. Their method is based on the *Greedy Randomized Adaptive Search Procedure (GRASP)* method where a trial solution is constructed using a greedy heuristic which is then used as the starting solution for a local search which terminates once a local optimum is found. This is repeated multiple times with different starting solutions due to the randomness. At the timeout, the best solution is selected. *Iterative Local Search* works by iteratively applying local search to the current best solution for a given number of iterations. In order to escape from local optima, the current best solution is first disturbed before applying a local search iteration. Haddadene et al. (2016) replace the local search used in *GRASP* with the *ILS* method. Unlike Polnik et al. (2021), they do not use any special methods for dealing with temporal constraints, which leads to a very complicated procedure for finding allowed insertions during the ILS phase. While the results are comparable to those obtained by a MIP formulation solved with the standard CPLEX solver, the findings in the paper show that there is no significant improvement of maximum instance sizes gained over exact methods. With a largest instance of 73 locations, 16 vehicles and around 7 temporal dependencies, the scalability is worse than some exact methods such as Rasmussen et al. (2012).

A memetic (combination of a genetic algorithm and a local search algorithm) is proposed for the VRSP with one skill with two skill levels, and soft constraints for time windows and synchronization by Decerle et al. (2018). The soft synchronization constraint with a penalty for the time between a synchronized pair of visits allows for a simpler local search where an operation is applied if its effect on the objective is strictly positive. Although the results were slightly worse than the best-known solution, their approach successfully handled all test instances of Bredstrom and Rönnqvist (2007) with comparable results in 10 minutes instead of the 60-minute timeout on the branch-and-price method proposed by Bredström and Rönnqvist which only solved[4] 44 out of 60 instances.

3.3.3 Hybrid methods

In their paper Bredström and Rönnqvist (2008) propose a hybrid method for a VRP with time windows, precedence constraints, and skills by iteratively utilizing a MIP in their heuristic method. They theorized that heavily restricting the problem by only allowing a subset of visits to be considered per vehicle at each iteration would drastically reduce the *B&B* tree size. Their proposed heuristic works by generating visit subsets for each vehicle which are the only visits that vehicle may perform. Then, they use their MIP to find an initial solution using these subsets. As visits are allowed to be left unserviced at a penalty cost, and as visits can be part of multiple visits, not all visits in a vehicle's subset will be visited by that vehicle. These visits which are included in a vehicle's subset but are not visited by that vehicle are deemed unnecessary and are therefore removed from the subset. With these reduced subsets, the best solution which has been found thus far can still be created. However, there might be subsets which might result in a better solution. As determining which visits to add to which subset is the same problem as the main problem at hand, randomness is used to add visits to the subsets. With these new subsets, the MIP is solved again and the best solution is updated if an improvement is found and the cycle restarts with reducing the subsets. In order to compare the result quality of this hybrid method, they solved a number of smaller benchmark instances (using the MIP from the heuristic) to optimality using the CPLEX solver. With their proposed heuristic, they managed to find good feasible solutions for all instances. For the smaller benchmark instances, the results were close to (and sometimes equal to and thus optimal) those from the CPLEX solver on their MIP whilst running in a fraction of the time. For the smaller instances, the hybrid method had a timeout of two minutes while the larger instances

[4] To optimality.

had a timeout of 10 minutes. By using this method, Bredström and Rönnqvist (2008) were able to solve instances[5] with 80 locations, 16 vehicles, and 8 temporal dependencies.

[5]These instances are the same ones used by Bredstrom and Rönnqvist (2007).

Chapter 4

Solution quality definition

We want to define a metric to determine the quality of the solutions created by our solution methods. For this, there are several considerations to be taken into account. Firstly, since there are no costs involved, we want to create a robust solution instead of a cheap solution. We define a robust solution as a solution where a delay on any given job has a small chance of disturbing jobs later on in the schedule. As it is unknown which job might experience a delay, our aim is to distribute the available buffer space as evenly as possible over all the jobs as described in Chapter 2. For each solution, the number of available mechanics is known in advance. As costs are not considered in the solution quality, we do not value solutions with unused mechanics higher than solutions where all mechanics are used.
While we do not take costs into account, steering towards a solution with very large buffers is wasteful of the mechanics' time when they still have to wait for their next scheduled job to become available. Furthermore, such a solution could create the illusion that planning fewer mechanics would lead to a vastly lesser solution, while the buffers might still be big enough. Instead of allowing all buffer values to be used when calculating the score, we work around this problem by setting a buffer cap for the score calculation. While buffers may exceed the buffer cap, all buffers which match or exceed the size of the cap receive the same score. After speaking to the NS and looking at the duration of the jobs in the problem instances from Chapter 8, we decided to use a buffer cap of 20 minutes as this is realistic for the instances and a solution with the smallest buffer at 20 minutes constitutes a good solution according to the NS.

Another consideration is the valuation of buffers. When a scenario arises where not every buffer can reach the buffer cap, we need to be able to valuate a buffer based on its size. The simplest option would be a linear valuation where a buffer of 10 minutes would be 5 times as valuable as a buffer of 2 minutes. Such a linear valuation clashes with our aim of a uniform distribution of buffer time over the buffers. It would also prefer increasing a buffer of 12 minutes with 5 minutes over increasing a buffer of just 1 minute by 4 minutes. A good buffer valuation should prefer increasing small buffers by a smaller amount of time over increasing large buffers by a slightly larger amount of time. Such a function can be found in the mathematical concept of the arc tangent function. This function has already proven to be valuable for other planning problems involving robustness, see Diepen et al. (2012). The need for this scoring function is illustrated by Figure 4.1 where we see that, while schedule A has a larger combined buffer, schedule B is more desirable due to the more even distribution of the buffers and the larger smallest buffers.

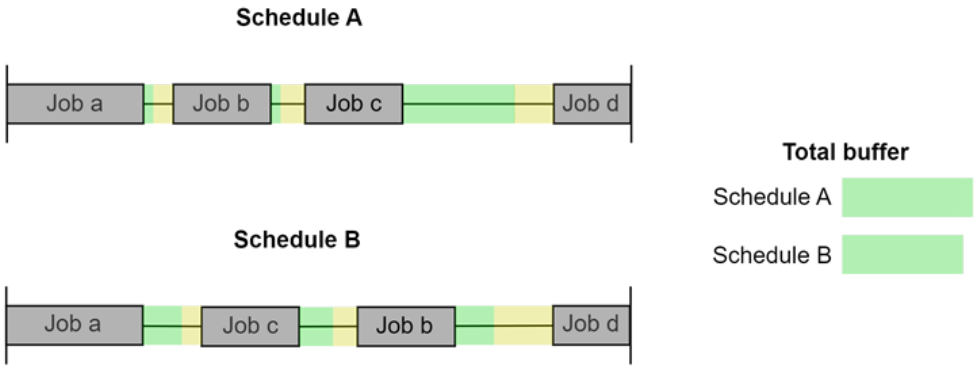

Figure 4.1: Two schedules with the same jobs in different configurations with their buffer times (green) and travel times (yellow).

When we combine our non-linear scoring with the buffer cap for scores, we get the following score function for a given buffer:

$$score(b) = \arctan(\frac{Min(b, buffercap)}{360}) \tag{4.1}$$

We divide the buffers by 360 as this leads to a scoring function which can make a significant distinction between higher buffer values and provides a smoother curve. In Figure 4.2, the advantage of dividing by 360 becomes obvious when one takes into account that buffers, in order to be useful, should at the very least be a few minutes long.
It is noteworthy that the first job in each schedule does not have a preceding job and thus no buffer. For these jobs, we use an artificial buffer with the size of buffer cap as the buffer value when calculating the solution scores. This effectively means that the fist job in a schedule always has the highest possible score. Currently, there is no defined planning method for, or data on the mechanics and the quality of their schedules which are currently employed by the NS. Therefore, we are unable to compare our solutions to the current situation.

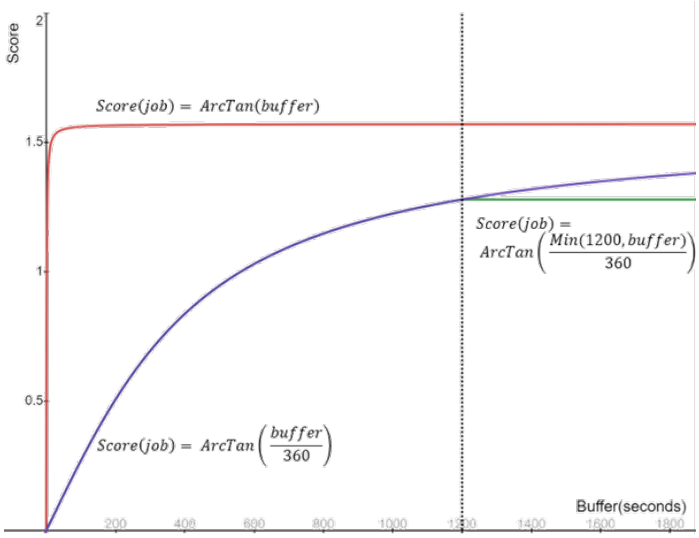

Figure 4.2: A comparison of the scoring functions using just the Arctangent (red), the Arctangent with the buffer divided by 360 (purple) and the scoring function we use with the buffer cap included (green). The dotted line on 1200 seconds represents the buffer cap.

Chapter 5

An exact method for the SYMSP

In order to try to find the optimal solution for the SYMSP, we have created a MIP formulation which we solve using Gurobi. Solving instances of the SYMSP using this model is not fast, in fact, for instances with no tight time windows and few mechanics, it struggles to find a feasible solution. Therefore, we have decided to look at the simpler problem without synchronization.
Each job has at least one option where it can take place. Jobs with multiple time windows have multiple options. For each job $j \in J$, the set of options for each job is denoted as K_j with individual options for job $j \in J$ being referenced to as $k \in K_j$. All options reside within the set K. For each job, exactly one[1] job option has to be assigned to the schedule of a mechanic $m \in M$. The variable indicating for each mechanic m if he/she performs job option k is denoted with the variable $x_{m,k}$ which is 1 if m performs k and 0 otherwise. The time when j is serviced is determined with the variable t_j, as we choose exactly one option per job, we can assign the start time to the job instead of having it per job option.
To choose a job option for each job and assign the option to a mechanic who carries out the job within the interval specified for the chosen option and at the location of the chosen option, we need to take multiple factors into consideration. For instance, the amount of time it takes to travel between options is dependant on which options we look at. We also need to make sure that jobs and travel times do not overlap, so a buffer of at least zero seconds should exist between the completion of option k and the moment the mechanic starts walking to the next option k' in his schedule.
As we are planning over a period of time, we need to know the order in which each mechanic executes the jobs in his/her schedule. The order of the jobs is strictly increasing in start time. Ideally, we would assign jobs to positions in the mechanics's schedules, but unfortunately, we do not know the optimal number of jobs per mechanic in advance. In order to define this order, each job, except for the first job is assigned a predecessor which is the job that directly proceeds it. in the mechanic's schedule The job to which the first job option in the schedule of mechanic m is indicated with the variable $F_{m,j}$ which has the value of 1 when job j is the first job in the schedule of mechanic m. We have this indicator per job as only one job option per job can be chosen and because the order and buffers are also determined for the jobs, not the job options. For each combination of two jobs j_a&j_b, the variable P_{j_a,j_b} indicates if j_a is the direct predecessor to j_b.
When planning the jobs, we need to respect the time windows of the chosen options and make sure that the jobs fit inside of the mechanic's time windows. We also have to ensure that jobs can only be assigned to mechanics who are sufficiently skilled for the job. For each job option k, there is a time window where it can occur. This time window opens at $w_{k,open}$ and closes at $w_{k,close}$ and is

[1] Jobs requiring a synchronized visits are modelled as two identical jobs.

at least as big as the duration of the job j to which option k belongs. The duration of a job j is denoted as d_j which indicates how long it takes to complete the job after starting it. As mechanics work in shifts spanning a given time period, a mechanic m can only start working on jobs when their shift has started at $m.start$. The mechanics stop working after their shift has ended, therefore, they should be finished with the last job in their schedule at the end of their shift at $m.end$.
Apart from the time windows, we also have to keep track of the travel times between jobs. Each job option has a specific location where the train unit is at the time of the option. Therefore, each option is linked to one location $l \in L$. Based on which job option is linked to which location, we determine the travel times between all job options and to state that the time it takes to travel from job option $k \in K$ to option $k' \in K$ is denoted as $e_{k,k'} \in E$ for each pair of options which do not belong to the same job.
It is important to use the correct traveling times as we need to know which time between jobs is spent walking so we can subtract it from the buffer time.
This buffer time is what is used in the objective function. We define this buffer per job as $b_j \in B$. As we want to prevent a mechanic from working at more than one job at a time, this buffer needs to be at least zero.
In Chapter 2.3, we explain how skills are handled in the `SYMSP`. In order to include skills in our MIP formulation, we say that each job $j \in J$ requires the given level $r_{j,s} \in R$ for each skill $s \in S$. Mechanics have the qualification of level $q_{m,s}$ for each skill $s \in S$.

In our formulation, All in all, this description of our MIP formulation yields the following list of constants and variables:

(constant) Mechanics: $m \in M$
(constant) Jobs: $j \in J$
(constant) A job option for job j: $k \in K_j$
(constant) The set of job options for job j: K_j
(constant) The set of all job options: K
(constant) Time windows: $w_{k,\{open/close\}} \in W$
(constant) Job durations: $d_j \in D$
(constant) Travel times: $e_{k,k'} \in E$
(constant) Skills: $s \in S$
(constant) Skill requirements per job and skill level: $r_{j,s} \in R$
(constant) Mechanic qualification level per skill: $q_{m,s} \in Q$

(Variable, binary) mechanic m performs job option k : $x_{m,k} \in X$
(Variable, integer) job j has a buffer from its predecessor of size b_j : $b_j \in B$
(Variable, integer) time when job j starts $t_j \in T$
(Variable, binary) Indicator if job h is the direct predecessor to job j (and therefore they have the same mechanic): $p_{h,j} \in P$
*(Variable, binary)*Job j is the job to which the first job option in the schedule of mechanic m belongs: $F_{m,j}$

5.1 Objective

In Chapter 4, we explain how the quality of a solution is evaluated. With this scoring method, the objective function becomes:

$$MAXIMIZE : \sum_{j \in J} \arctan(\frac{Math.Min(b_j, buffercap)}{360}) \tag{5.1}$$

This objective function takes the buffer of each job, applies the scoring function to it, and then sums it together with the other scores to determine the total score. The arc-tangent is not a linear function, and we therefore calculate the arc-tangent value for each possible integer buffer value from 0 to the value of buffer cap before running the solver and retrieve the calculated values using piece-wise linear approximation. As the start times and buffer values are all integral, we do not have to apply any rounding for fractional values.

5.2 Constraints

Solving the `SYMSP` while accounting for the constraints mentioned in Chapter 2, we need to define a set of constraints to get to a valid solution.

For each job, we want to plan exactly one job option as each job has to be carried out once. This requirement leads to Constraint (5.2).

$$\forall j \in J : \sum_{k \in K_j} \sum_{m \in M} x_{m,k} = 1 \tag{5.2}$$

Now that we can assign job options to mechanics, we need to make sure that the time when the job option is carried out lies within the allowed time window for the specified option. Constraint 5.3 specifies that the start time of a job option must be at least the start of its time window while Constraint 5.4 specifies that the start time of a job option must at least leave the duration of the job as room before the end of the time window so it can be completed in time. Constraints 5.5 and 5.6 specify the same, but in their case, we use the mechanic's time windows as the bounds for the start times.

$$\forall j \in J : \sum_{k \in K_j} \sum_{m \in M} (x_{m,k} * w_{k,open}) - t_j \leq 0 \tag{5.3}$$

$$\forall j \in J : \sum_{k \in K_j} \sum_{m \in M} (x_{m,k} * w_{k,close}) - (t_j + d_j) \geq 0 \tag{5.4}$$

$$\forall j \in J : \sum_{k \in K_j} \sum_{m \in M} (x_{m,k} * m.start) - t_j \leq 0 \tag{5.5}$$

$$\forall j \in J : \sum_{k \in K_j} \sum_{m \in M} (x_{m,k} * m.end) - (t_j + d_j) \geq 0 \tag{5.6}$$

In order to ensure that each mechanic only works on one job at a time and does not need teleportation to move around, we want to add a constraint specifying that between two consecutive jobs, there is enough time to walk from the location of the first job to the location of the second job.
Another requirement with regard to the time between two consecutive jobs is that we determine the buffer time between these jobs. As the buffer time is the time between the end of a job and the start of the next job minus the travel time, we can use the buffer time to require sufficient time to travel between jobs. For this, we first need to specify that the buffer time should be at least 0. A negative

buffer time would mean that there is not enough time between jobs to travel between them. We therefore introduce Constraint 5.7. As the value of b_j is the buffer value used to determine the score, it can be at most the buffer cap used to determine the score. The actual buffers in the schedule may be bigger than the buffer cap, but this is accounted for in the Constraint 5.8 by stating that b_j can be at most the actual buffer time instead of requiring b_j to be equal to it.

$$\forall b_j \in B : 0 \leq b_j \leq buffercap \tag{5.7}$$

To ensure that buffer times in B do not exceed the actual buffer times from the solution and to require this time to be sufficient for the travel time between consecutive jobs, we introduce Constraint 5.8 where we require the buffer b_h to be at most the time between the start time of $h \in J$ and its predecessor $g \in J$ minus the travel time between their locations and the duration of g. The second line of Constraint 5.8 deals with the start times, duration and travel times. While this is fine for jobs which are actually succeeding each other, using just the first two lines would mean that two jobs which are not directly succeeding would require the start time of the second job to be greater than that of the first job. This leads to infeasible solutions at worst, and very bad solutions at best. In order to combat this, we add the third line of the equation where we make use of a *bigM* constraint. For each pair of non-succeeding jobs, we add a very big number (∞) to the result of the second line to ensure that the result is always bigger than 0. The second half of the third line handles cases where the jobs are directly succeeding, but the specific options are not both selected. Constraint 5.8 also contains an and-operator which is not linear. While we use the formulation:

$$\sum_{m \in M} AND(x_{m,k}, x_{m,k'})$$

in Constraint 5.8 to make it more readable, in practice we introduce a variable which is assigned the correct value for this part of the formulation based on a set of constraints which keep the model linear for each pair $k,' k$.

$$\begin{aligned} &\forall k \in K_g, \forall k' \in K_h : \\ &t_h - (t_g + d_g) - (e_{k,k'})) \times p_{g,h} \\ &+((1 - p_{g,h}) \times \infty) + (1 - \sum_{m \in M} (AND((x_{m,k}, x_{m,k'}))) \times \infty) \\ &\geq b_g \end{aligned} \tag{5.8}$$

Because Constraint 5.8 is only restrictive if the value for $p_{g,h}$ is 1, an optimal solution at this point could be reached by simply not assigning a predecessor to jobs and giving all jobs a false buffer the size of the buffer cap. The only jobs without a predecessor are the jobs at the first position of each mechanic's schedule. Therefore, we need to require all jobs to have a predecessor, or to be the first job in the schedule. We achieve this by adding Constraint 5.9.

$$\forall j \in J : \sum_{h \in J} (p_{h,j}) + \sum_{m \in M} (F_{m,j}) = 1 \tag{5.9}$$

Now, each job must be the first job or have a predecessor. Unfortunately, a solution where one job is chosen to be the predecessor of all jobs is not prohibited. As a job can only be the direct predecessor to at most one other job, we add Constraint 5.10 which requires each job to be the predecessor to at most one job. The reason we set this to at most one is that the last job in each schedule does not have a successor and adding another variable just to identify the last job would be wasteful.

$$\forall j \in J : \sum_{h \in J} (p_{j,h}) \leq 1 \tag{5.10}$$

Another possibility for invalid optimal solutions can be achieved by making each job a first job, thus circumventing the restrictions which are applied on predecessors and successors. Therefore, we want to specify that for each schedule, there can only be one job which is designated as the first job. We achieve this by adding Constraint 5.11 which states that for each mechanic, only one job out of the set of all jobs can be the first job for that mechanic.

$$\forall m \in M : \sum_{j \in J} (F_{m,j}) \leq 1 \tag{5.11}$$

The final scenario for invalid predecessor allocations is cross-linking jobs which are in different schedules (i.e. saying $P_{g,h} = 1$ while jobs g and h reside in different mechanic's schedules). Cross linking might lead to a better solution as it can implicitly put a job in the schedule of a mechanic who is not qualified to execute that job, or has conflicting time windows with the time windows of the job. Therefore, we introduce Constraint 5.12 where we require for each combination of jobs that if the indicator variable that they are directly succeeding has a value of 1, their selected options must be assigned to the same mechanic in order to satisfy the constraint.

$$\forall g \in J, \forall h \in J, \forall m \in M : 2 \times p_{g,h} \leq \sum_{k \in K_g} (x_{m,k}) + \sum_{k \in K_h} (x_{m,k}) \tag{5.12}$$

Lastly, we need to make sure that jobs can only be assigned to mechanics with the right skills. As mentioned, the skill level of a mechanic for each skill must be at least the level that is required by the job. We use Constraint 5.13 to say that each job option $k \in K_j$ must be assigned to a mechanic which is qualified for every skill required by job j.

$$\forall k_{j,i} \in K, \forall s \in S : \sum_{m \in M} (x_{m,k}) * Q_{m,s}) \geq R_{k,s} \tag{5.13}$$

While Gurobi automatically restricts binary variables to have the value of $0/false$ or $1/true$, we want our formulation to be as complete as possible. Therefore, we add the following constraints our binary variables to only be allowed to have the value of either 0 or 1.

$$\forall k \in K, \forall m \in M : x_{m,k} \in \{0, 1\} \tag{5.14}$$

$$\forall j \in J, \forall m \in M : F_{m,j} \in \{0, 1\} \tag{5.15}$$

$$\forall h \in J, \forall j \in J : p_{h,j} \in \{0, 1\} \tag{5.16}$$

5.3 Column generation

One method for creating a solution with mechanic schedules is column generation where multiple mechanic schedules consisting of job options are made and given a score. These schedules have to adhere to time windows and can be given skill sets which match the available skill sets of the specific mechanics. Then, creating the complete solution is a matter of assigning one schedule to each mechanic[2] such that each job has one option in a schedule in the final solution. While a similar method might be possible for the `SYMSP`, it is unlikely to be successful. Unfortunately, synchronization adds constraints which link schedules to each other as the two jobs of each synchronized job pair must

[2]With the correct skills to complete the schedule.

be executed with the same option and at the same start time while they both have to be assigned to a different schedule. These constraints heavily restrict which schedules are allowed to be selected and make the problem much more complex. An example of this method can be found as solution to parallel machine scheduling problems proposed by Van den Akker et al. (2012) where they create the mechanic schedules using a local search algorithm.

Chapter 6

Local Search for the SYMSP

In order to find a reasonably good solution in an acceptable amount of time, a heuristic solution method is needed. We decided to use a simulated annealing local search approach to build and modify a solution.

6.1 Local Search Loop

The local search method we have developed uses a loop with four distinct phases in order to find a solution. The loop automatically terminates when the best solution so far has the theoretical best possible score. Otherwise, it terminates after a predefined number of cycles which is currently set to two billion. Solutions are scored with the scoring formula described in Chapter 4 where we put a maximum on the buffer values used to determine the score: the buffer cap. By using this buffer cap, we prevent excessively large buffers from getting valued higher than buffers which are large enough. In our local search, we also use this buffer cap. While the buffer cap for our exact approach from Chapter 3.3.1 is set to 1200 seconds, this could be disadvantageous for our local search where a few large buffers within the buffer cap range could create a local optimum when the other buffers are still small. Therefore, we want to use a buffer cap which can still limit excessively large buffers at 1200 seconds, while also not being so large that it restricts the exploration of neighboring solutions. To achieve this, we use a dynamic buffer cap which starts at 360 seconds but is increased by 60 seconds each time all buffers in the solution are at least 80% as big as the buffer cap and no jobs are left unplanned. By doing this, we avoid a situation where we choose a buffer score ceiling which is too big to be reached by all buffers which could result in the smallest buffers being shrunk to gain an advantage in the already large buffers. This approach can cause problems for scenarios where the maximum buffers for the jobs have a big variance in size. In this case, the cap could be too low for some jobs because a single job cannot get a bigger buffer than it currently has. We combat this by increasing the buffer cap by 61[1] seconds every time we let the temperature increase (as described in Chapter 6.1.2) when during the current run with a decreasing temperature, the buffer cap has not been increased. As per NS specifications, the buffer cap will never exceed 20 minutes. For other applications, the buffer cap can be adjusted to suit the specific requirements.
In our local search, we use an initial solution where no jobs are planned and all schedules are empty.

[1] We use a value of 61 seconds as one second more does not impact the results, but can be used to keep track of how many times the cap was raised without meeting the requirements to raise it (i.e. when a buffer cap of 602 is reached, we know two of the raises were done with 61 seconds without meeting the requirements for a raise).

The working of the local search loop is described below.

6.1.1 Neighborhood exploration

For every cycle in our local search, a neighborhood operator is randomly chosen with all operators having the same chance to be chosen. The algorithm checks if this operator can possibly be applied to the current solution by checking if there are unplanned jobs left in the case of insertions, or if there is at least one schedule with enough planned jobs for the other chosen operator to be applied. If the operator can be applied, the schedule(s) on which it is applied, the necessary parameters such as an unplanned visit, and indices on which it is applied, can be selected while always respecting the bounds of allowed solutions. For instance, only schedules of qualified mechanics can be selected for an insertion.
After the selection of an operator, the schedule(s), and parameters, the chosen operator is applied. If the application is successful, the operator will return with status *true*. If the operator can not be applied for the given parameters, like when a schedule is full, the status $false$ is returned. When successful, the operator updates the score of the schedule. We recognize that some people disagree with applying an operator before determining if the resulting solution would be accepted as it requires a possible reversal of the operator. While we do admit that undoing operators is not ideal, we would like to point out that our algorithm runs in a reasonable time and that some operators which require a forced insert would be just as expensive to undo as it would be to first calculate the score and then update them.
After a successful operator application, the resulting change in solution score is examined. If the change in score is positive, the alteration is accepted. When the change in score is negative, there is a chance of $p = Math.Exp(\frac{changeinscore}{temperature}) * 100\%$ that the change will be accepted. If the change is not accepted, the operator is undone and the solution returns to the state it had before the operator was applied. When the score of the current solution is better than that of the best solution known thus far, the best solution is updated to the current solution.

6.1.2 Parameters and Temperature Updates

When starting the local search, we start with a temperature of value 5 and an empty solution. At each interval of $500k$ cycles, the temperature is multiplied by a factor of 0.99. If a new best solution has been found in this cycle, we use the optimizer described in Chapter 6.3 to determine the best score for the given job-to-mechanic assignment in that solution. When the temperature is lower than 0.1, the starting temperature is multiplied by 0.75 after which we set the current temperature to the new value for the starting temperature.
After each interval, we also check how many interval updates have passed since last finding an improvement to the current best solution. If there have been 2 or more intervals without improvement, we allow insertions to be done by force (which we explain in Chapter 6.2.1) until an improvement is found. An interval without improvements can, when combined with a low temperature, indicate a local minimum. In order to escape this local minimum, we allow 100 operations which would be denied due to their negative score impact to be accepted despite their bad impact on the score in the next interval when the temperature has a value smaller than 2.

6.2 Neighborhood Operators

For the local search, we use a number of different neighborhood operators. Below, we will explain which operators we have used. We use the following operators:

- Smart Insert Operator (SIO): inserts a job visit in a schedule.
- Remove Operator (RO): removes a job visit from a schedule.
- Move Beteen Schedule Operator (MBSO): moves a visit from the schedule of mechanic A to the schedule of mechanic B.
- Chance Chosen Option Operaotr (CCOO): change the chosen option for a planned job.
- Optimize Pair In Schedule Operator (OPISO): optimize the placement of two consecutive jobs within a schedule.

6.2.1 Smart Insert Operator (SIO)

The Smart Insert Operator (SIO) is used to select an unplanned job and add it to a schedule by inserting one of the job's possible visit options in the schedule of a mechanic with the right qualifications to execute the job. The selection of an unplanned job (including which visit option is used) and a qualified mechanic are randomized. SIO is called smart because unlike the selection of a job and a mechanic, the placement of the visit in the mechanic's schedule is not random. To find the best position for the job, SIO iterates over the schedule and checks each possible placement from inserting at the start of the schedule, between two adjacent visits in the schedule, and at the end of the schedule. When checking for possible placements, the start times of the already planned jobs cannot be changed. A placement is deemed viable when there is enough time between the end of the previous visit[2] and the start of the next visit[3] to walk to the unplanned visit, complete it, and walk to the next visit if there is one. When a placement is deemed viable, we calculate the change in score which would result from placing the new visit at this position with the best possible buffer allocation. The best possible buffer allocation is achieved by placing the new visit at the chosen position with its start time set such that the buffers between the visit directly preceding the new visit and the visit directly succeeding the new visit are distributed as equally as possible.
After looking at all possible placements, we compare the best placement, which has the highest new score for the schedule, with the second best placement, which has the second highest new score for the schedule, if these placements exists. When the second best placement's score is within a certain temperature based percentage of the best placement's score, we choose the second best placement over the best placement. By sometimes choosing the second best placement, we aim to leave more open space for other jobs to be inserted as leaving space for other operations can be more valuable than the difference in scores between the best and second best option. However, leaving space for other operators is mostly a concern when filling the schedule and the temperature is high, not when the temperature is low and we are trying to find the (local) optimum. Therefore, as the temperature decreases, the percentage of the best scorechange needed to accept the second best scorechange increases until it becomes 100 percent.

As our model tries to optimize[4] the buffers, finding a viable position for an insertion does not

[2] Or the start time of the schedule when inserting at the first position of the schedule.
[3] Or the end time of the schedule when inserting at the last position of the schedule.
[4] By trying to maximize two buffers between visits while prioritizing growing smaller buffers over larger buffers.

pose a problem for a scenario where the median buffer size roughly equals the average duration of a job to be completed. Unfortunately, this scenario is not reflected in many real-world instances. Therefore, instead of politely placing a job where there is place without moving other visits, we need to be able to apply some elbow-grease and squeeze a job somewhere in a schedule which requires other visits to move to make room for the insertion.
When the SIO does not find a viable position to insert a job, it has the option to try to forcibly insert it anyway. When applying forced insertions, the schedule is iterated once more, but this time we look for a position with neighboring visits that have enough possible movement to make space to insert the new visit. As soon as such a position is found, a gap is created by moving the neighbors and their neighbors if necessary and the new visit is inserted in the gap. Forced insertions are only allowed if after a certain number of operations, no improvement on the best solution was found. Forced inserts require significantly more steps than smart inserts, therefore they are only allowed as a last resort when normal inserts do not seem to improve the solution anymore. Even though forced inserts do not always happen at the most optimal position, the possible viable positions for an insertion in a situation where forced inserts are required are likely to be close to each other or even adjacent. We have other operators such as OPISO, described in Chapter 6.2.5, which can improve the placement by shifting the visit around within the schedule to other nearby placements. After applying a forced insertion, some buffers might be set to zero while in theory, they could be bigger by shrinking buffers elsewhere. However, a method which could evenly allocate the remaining buffer space over the possible n visits while respecting the time windows is too expensive to use in a local search.

Algorithm 1 Smart Insert Operator

```
Require: visit v, schedule s
  [bestPlacement, secondBestPlacement] ← GetOptimalPlacement(v,s)
  if bestPlacement == null then
      bestPlacement ← GetForcedPlacement(v,s)
     if bestPlacement == null then
      return false
     end if
  end if
  if secondBestPlacement.scoreChange ≥ p × bestPlacement.scoreChange then
     bestPlacement ← secondBestPlacement
  end if
      s.Insert(v, index: bestPlacement.Index);
      return true
```

6.2.2 Remove Operator (RO)

The Remove Operator (RO) is used to remove a visit from a schedule at a given index. The RO only removes the visit at index i and updates[5] the travel time and buffer time at the visit that was at index $i + 1$. When removing a visit, the start times of other visits in the mechanic's schedule are not updated to leave a gap for a possible insertion. There is also an OPIS-operator, described in Chapter 6.2.5, which might be applied to a preceding and succeeding visits which to their start times.

[5] Only if $i <= schedule.count() - 2$

Algorithm 2 Remove Operator

Require: index i, schedule s
  if $0 \leq i \leq s.count - 1$ **then**
    s.RemoveAt(i);
    return $true$
  end if
    return $false$

6.2.3 Move Between Schedules Operator (MBSO)

For any given solution, a logical neighboring solution would be the same solution, but with one visit transferred to a different mechanic's schedule. Moving a visit from schedule s_a to schedule s_b is done by the Move Between Schedule Operator (MBSO) which utilizes the RO to remove the visit at s_a and insert it in schedule s_b.
Moving a visit between schedules might pose a problem with the implementation of skills as the selection of qualified mechanics and their schedules for the visit might be limited to just one mechanic. We have therefore implemented the MBSO such that is also allows an insertion in originating schedule if this is the only allowed schedule. Allowing a move within the original schedule when there is no allowed other schedule to move could prove to be beneficial as changes in the schedule may have made the current placement unfavourable. It is important to note that when allowing inserts in the original schedule, the removal of the visit from the schedule should be done **before** the insertion as it might block its own placement otherwise. When inserting in the original schedule, the original placement is forbidden to encourage the exploration of different solutions.

Algorithm 3 Move Between Schedules Operator

Require: index i, schedule s_a, schedule s_b
    $v \leftarrow s_a[i]$
    s_a.RemoveAt(i);
  if !SmartInsert(v, s_b) **then**
    Undo(RemoveAt(i), s_a)
    return $false$
  end if
    return $true$

6.2.4 Change Chosen Option Operator(CCOO)

As discussed, some jobs have multiple visit options. In order to explore solutions where a different visit was chosen than the current one for a job j, we introduced the Change Chosen Option Operator (CCOO) which is a combination of the RO and SIO similar to the MBSO. Unlike MBSO, CCOO only works on jobs with multiple visit options and chooses a different option instead of the current option used by MBSO. Just as with the MBSO, the CCOO is a combination of two other neighborhood operators and thus it could occur without being implemented specifically. However, the removal of a visit when the temperature is low would almost certainly lead to an unaccepted score change and thus an undo of the operator. Especially in tightly filled schedules which mostly occur at low temperatures, CCOO allows for option changes which might not occur naturally.

Algorithm 4 Change Chosen Option Operator

Require: index i, schedule s_a
    $v \leftarrow$GetOtherOptionForJob($s_a[i]$)
  if $v \neq null$ **then**
    $s_b \leftarrow v$.QualifiedMechanics.GetRandom();
    s_a.RemoveAt(i);
    if !SmartInsert(v, s_b) **then**
      Undo(RemoveAt(i), s_a)
      return $false$
    end if
    return $true$
  end if
    return $false$

6.2.5 Optimize Pair In Schedule Operator (OPISO)

In the description of the SIO and RO, we mention that a forced placement and a removal can both lead to a solution where an improvement in a schedule's score could be achieved by resizing the buffers within the bounds of the schedule and time windows. We have also noted that optimally resizing the buffers for an entire solution is too computationally expensive to be applied constantly. Therefore, we have introduced the Optimize Pair In Schedule Operator (OPISO) which, given a schedule with at least two visits and index of the first element of the pair, optimizes the placement of that pair of visits while not altering the rest of the schedule.
For a schedule $s = [..., v_a, v_b, v_c, v_d, ...]$ and an index pointing to v_b, OPISO considers two scenarios. In the first scenario, the order of v_b before v_c is preserved. The second scenario reverses this order and assumes that v_c happens before v_b. For both scenarios, the potential buffer time is calculated by taking the time between the planned end of v_a and the planned start of v_d and subtracting the duration of v_b and v_c along with the total travel time from v_a to v_d trough v_b and v_c in the order reflecting the scenario. By moving v_b and v_c, their buffers along with the buffer of v_d can be affected. After determining the total potential buffer time for both scenarios, OPISO tries to find the best allocation of the potential buffer time over the three buffers for both scenarios. The OPISO then chooses the best placement and applies it. In the case that there is no improvement by keeping the original order, but a worsening by changing it, the algorithm changes the order and presents the worsened score to be either accepted or rejected. By doing this, we hope to explore more possible solutions. When OPISO operates at the first index of a schedule, it places the first visit in both scenarios at the earliest opportunity where it can be planned within the schedule and uses the placement of the other visit from the pair to evenly distribute the buffer time[6]. When operating at the last index of a schedule, OPISO takes a similar approach by placing the last visit at late as possible in the schedule. An interesting aspect of OPISO is that, unlike other operators, it does not have the possibility to lead to a worse score as when the current placement and buffers are optimal for the current schedule, OPSIO will return these.

[6]In this case, there are at most two affected buffers.

6.3 Optimizer

Heuristic methods are not known for their ability to find optimal solutions, whereas optimal solution methods are not known to be fast. At some point, our local search algorithm will reach a local optimum given the temperature and the neighborhood that our operators can reach. The fact that we have reached a local optimum does not mean that the current set of visits per mechanic is arranged in an optimal schedule. Creating an optimal schedule for his allocated job options in the solution is something which is possible with an LP optimizer. We have implemented such an optimizer where we set the buffer cap to be 120 seconds larger than the current cap. Inspired by Bredström and Rönnqvist (2008), we use the current allocation of visits per mechanic and use a LP formulation, which we solve using the Gurobi solver to get the optimal route for each mechanic. The optimizer is free to change the order and start times of visits within a mechanic's schedule as long as all constraints are respected. Which options are chosen and to which mechanic each option is allocated cannot be changed by the optimizer. Without synchronized visits, we could solve each schedule individually. However, we will later need to account for synchronization and therefore optimize all schedules all at once to ensure that the start times of synchronized visits are the same. Therefore, we have written this model with the addition of synchronization in mind. In Chapter 6.4.1.6 we will explain how to adapt this model to include synchronization constraints.
We apply the optimization step every time that the best solution found by the local search has been improved in order to try to improve the new best solution. We never use the optimized solutions in our local search, instead, we save them as a separate output. The optimizer pulls the solution into a local optimum which would reduce the effectiveness of the local search, therefore we choose not to use the optimized solutions in our local search.

As visits are pre-allocated to a mechanic and each planned job has a chosen option, this model is less complex than our exact solution method which also has to deal with visit allocation and option selection. The optimizer uses the following LP formulation:
Just like with out exact method described in Chapter 5, the objective of the optimizer is to get the highest possible score defined as the sum of the score function described in Chapter 4 applied over all buffers. However, as the assignment of visits to schedules is predetermined for the optimizer, we can get rid of the variables linking jobs to schedules. As we know the number of jobs in each schedule, we can assign jobs to positions in the schedule rather than needing variables to link jobs to a predecessor and successor. The chosen options are also predetermined, so we can also get rid of these.

This leaves us with the following constants:

(constant) Mechanic schedules: $m \in M$
(constant) Visits for each schedule: $v \in V_m$
(constant) Time windows: $W_{v,\{open/close\}} \in W$
(constant) Durations: $d_v \in D$
(constant) Travel times: $e_{v',v} \in E$

As we are linking visits to positions in the schedule, we need a variable to indicate if a visit is assigned to a certain spot. These variables are grouped per mechanic:

(Variable, Binary) In schedule $m \in M$, visit $v \in V_m$ occurs at position p: $P_{m,p,v}$.

Because time windows still have to be adhered to, and buffers have to be calculated, we need to know the start time of the job at each position. For this, we introduce the following variables:

(Variable, Integer) In schedule $m \in M$, the visit at position p starts at time (in seconds): $t_{m,p}$.

Consequently, each position other than the first one has a buffer time from the previous position defined with the variable:

(Variable, Integer) The buffer (in seconds) between positions $p-1$ and p in schedule $m \in M$ is: $b_{m,p}$.

With this set of variables, our objective to maximize the score becomes:

$$MAXIMIZE: \sum_{m}^{M} \sum_{p=1}^{m.visitcount} \arctan(\frac{b_{m,p}}{360}) \tag{6.1}$$

To generate a valid solution, we need to add a constraint which enforces that the time windows are adhered to. We therefore have to add a constraint to each positing stating that the start time of that position needs to lie within the time window of the visit assigned to that position. This leads us to the following constraint:

$$\begin{gathered}\forall m \in M, \forall p \in \{1, 2, ..., m.visitcount\}: \\ \sum_{v \in V_m} (P_{m,p,v} * W_v(open)) \leq T_{m,p} \leq \sum_{v \in V_m} (P_{m,p,v} * (W_v(close) - d_v)).\end{gathered} \tag{6.2}$$

As the order of the positions is increasing in each schedule, we can define the buffer between positions as the time between the start times of the positions minus the duration of the job on the $(p-1)$'th position and travel time from the $(p-1)$'th to p'th position. As our objective score is increasing with the buffer size and we are maximizing, we can require the buffer to be smaller or equal to the actual buffer time and bigger than zero. The reason we do not require the buffer for a position to be equal to the actual buffer time is that the buffer which we use for the score can be capped by the buffer cap. Restricting the buffer which is used for the score to the actual buffer size is done by Constraint 6.4. To determine which travel time to use, we need to know which consecutive pair of visits is planned on each pair of positions $p-1$ and p in each mechanic's schedule. Therefore, we want to determine for all pairs of visits allocated to each mechanic if, and on which positions they appear as a consecutive pair. For this, we introduce the variable $c_{m,v',v,p} \in C$.

(Variable, Binary) $c_{m,v',v,p} \in C$:
For mechanic $m \in M$, visit $v' \in V_m$ is placed on position $p-1$ while visit $v \in V_m$ is placed on position p.

We use Constraint 6.3 to ensure that $c_{m,v',v,p}$ can only have the value 1 when both $P_{m,p-1,v'}$ and $P_{m,p,v}$ have a value of 1.

$$\begin{gathered}\forall m \in M, \forall p \in \{2, 3, ..., m.visitcount\}, \forall v' \in V_m, \forall v \in V_m: \\ 0 \leq (P_{m,p-1,v'} + P_{m,p,v}) - (2 * c_{m,v',v,p}) \leq 1\end{gathered} \tag{6.3}$$

With this constraint, we can now determine the travel times and with them, Constraint 6.4 to ensure the correct buffer times.

$$\begin{gathered}\forall m \in M, \forall p \in \{2, 3, ..., m.visitcount\}: \\ 0 \leq b_{m,p} \leq T_{m,p} - T_{m,(p-1)} - \sum_{v' \in V_m} (P_{m,p-1,v'} * d_{v'}) \\ - \sum_{v' \in V_m} \sum_{v \in V_m} (c_{m,v',v,p} * e_{v',v});\end{gathered} \tag{6.4}$$

As the buffer sizes used to determine the solution's score are limited to the buffer cap, Constraint 6.5 limits the buffer which is used for the score to be at most the buffer cap.

$$\forall m \in M, \forall p \in \{2, 3, ..., m.visitcount\} : (B_{m,p}) \leq buffercap; \tag{6.5}$$

To ensure a valid solution, we require each position in each schedule is chosen exactly one. This is done with the following constraint:

$$\forall m \in M, \forall p \in \{2, 3, ..., m.visitcount\} : \sum_{v}^{m.visits} (P_{m,p,v}) == 1 \tag{6.6}$$

We also want to specify that each job can only be chosen once, and that each job has to be planned. Therefore, the following constraint requires each job to be put on exactly one position:

$$\forall m \in M \forall v \in m.visits : \sum_{p=1}^{m.visitcount} (P_{m,p,v}) == 1 \tag{6.7}$$

In our optimizer model, Gurobi automatically restricts binary variables to have the value of $0/false$ or $1/true$ as in the case in our exact model from Chapter 5. Once again, we want our formulation to be as complete as possible and therefore, we add the following constraints our binary variables:

$$\forall m \in M, \forall p \in \{1, 2, ..., m.visitcount\}, \forall v \in V_m : P_{m,p,v} \in \{0, 1\} \tag{6.8}$$

$$\forall m \in M, \forall v' \in V_m, \forall v \in V_m, \forall p \in \{2, 3, ..., m.visitcount\} : c_{m,v',v,p} \in \{0, 1\} \tag{6.9}$$

6.4 Dealing with synchronization

Dealing with synchronization is the main challenge in creating a local search algorithm for the `SYMSP`. A good local search algorithm is able to quickly and easily explore neighboring solutions. Unfortunately, the two most logical synchronization implementations all violate these principles. The first idea one might suggest for planning synchronized jobs is pairwise insertion and deletion: when inserting or deleting one job of a synchronized pair, the other one also has to be inserted/deleted. Pairwise insertion is a bad idea as it not only complicates insertions by having to do some expensive calculations to find two parallel insertion options in two different schedules, but it also can lead to some synchronized visits never being inserted because a free spot in two schedules is never found. The second idea provides a solution for this problem: separate insertion. With separate insertion, the first visit is inserted after which it is able to be moved around just like a normal visit. At some point, the second visit is chosen to be inserted. When inserting the second visit, the start time has to match the current start time of the first visit. While this gives a slightly better chance of one of the visits to be inserted, it might be hard to insert the second visit at the exact start time of the first visit.
With both of these methods, once both visits for a synchronized job pair have been planned, their start times have to remain the same. This is a heavy burden on the algorithm's ability to explore neighboring solutions as it either has to deal with fixed visits which prohibit movement, or it has to perform very time-expensive calculations over multiple schedules to see if both visits might be able to be moved.

It is clear that we need a better approach to handle synchronized visits with hard synchronization constraints. One that provides easier insertions and allows for (some) movement once both visits

have been planned. This is why we have introduced our own method: `Guaranteed OverLapping Duration Synchronization (GOLDS)`. As the name suggests, GOLDS makes use of overlapping durations to ensure visits can be synchronized.

6.4.1 Guaranteed OverLapping Duration Synchronization (GOLDS)

A different way to look at synchronization is to not focus on synchronized start times, but instead focus on a period of a minimum duration where two mechanics are present at the same location. When looking at separate insertion from this perspective, the second insertion has to happen within a small time window of $\{starttime(first), starttime(first) + duration(first)\}$ to ensure that there is an overlap for the entire duration of both visits. When working with the original duration, the duration of the small time window equals the duration of the jobs. While this holds for the original durations, we have found that in order to ensure an overlap of the original duration, you can expand the small time windows as long as the duration of the visits is increased as well. We have found that, as long as the time windows for both visits are the same, we can guarantee an overlap which is at least as long the original duration $d_{original}$ between the visits regardless of the placement of the jobs within the expanded time window TW' as long as they both have the correct expanded duration d'. The guaranteed overlap ,which is at least a long as the original duration, means that there will always be a period where both mechanics are present at the location of the job for long enough to synchronously complete the job. For two synchronized jobs with a an expanded time window which spans from 0 tot T, Figure 6.1 illustrates that even if both expanded jobs are put on start and end of the time window, there is always an overlap of at least the original duration $d_{original}$.

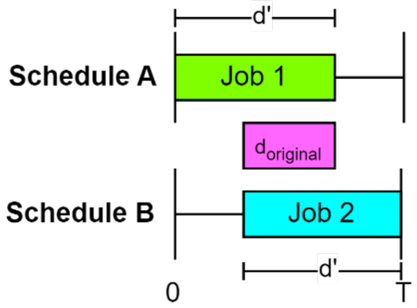

Figure 6.1: Jobs 1 and 2 have an overlap (pink) of at least $d_original$ regardless of their placement in the time window.

To guarantee the overlap of the original duration while keeping the time window as large as possible and the duration as short as possible to allow for the most possible movement, the time window TW' and duration d' have to adhere to the following equations:

$$TW' = 2 * d' + d_{original} \tag{6.10}$$

$$d' = \frac{TW' - d_{original}}{2} \tag{6.11}$$

Equations 6.10 and 6.11 require that if we expand the time window of a synchronized pair by 2 time units, we only have to expand the duration by 1 time unit to guarantee an overlap of $d_{original}$. Therefore, by expanding the time window with two time units, the jobs will be granted one time unit of free movement. We have illustrated this behaviour in Figure 6.2 where we show two synchronized jobs and their durations for three different time window sizes along with their guaranteed overlap. In this figure, we see that a larger time window allows for more freedom of movement for the jobs. By expanding time windows, we can create some movement for the visits while the guaranteed overlap of the original duration means that we can move the visits independently of each-other. This independent movement is possible as there will always be a period (during the overlap) where both mechanics have been scheduled to work at the synchronized jobs which is at least as long as the original duration. While some movement is allowed, GOLDS may still restricts movement more than would be the case when calculating the possible movement for each synchronized pair every time a movement has to be done as the time windows determined by GOLDS can be smaller than the original time windows. Working with larger durations can also lead to misrepresented buffer times or overly full schedules. However, we believe GOLDS is a good compromise between flexibility and quality. In order to prevent overly inflated durations from filling up the schedules, we set a temperature-based limit on how big they can become. Then, at every temperature update down to a predefined temperature, we gradually shrink the time windows and durations until the original duration has been reached. In the subsections below, we will explain how GOLDS works with respect to the operators described in Chapter 6.2. As one of the main goals of GOLDS is to prevent the need to update visits which do not have to be updated, we do not update the time windows and durations of synchronized visits in the case where an operation on a different visit could merit an expansion by creating more free time.

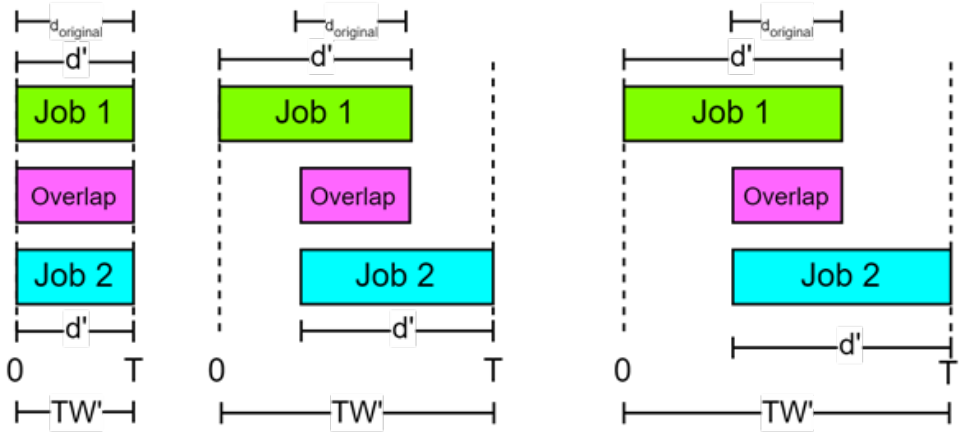

Figure 6.2: Synchronized jobs 1 and 2 with varying durations and corresponding time windows. On the left example, the time window is the size of the duration to ensure synchronization. The middle and right examples, we see the expanded time windows in accordance with the increased duration. The overlap (pink) is always as least as big as the original duration.

6.4.1.1 Insertions (SIO)

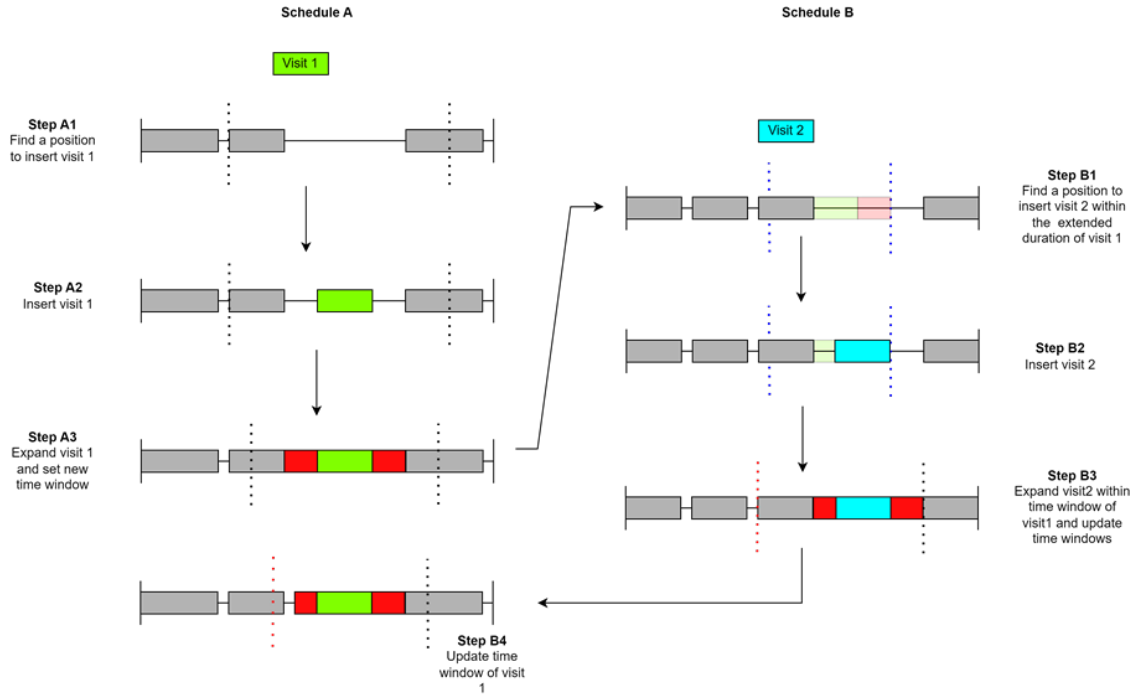

Figure 6.3: Insertions with GOLDS. The order in which the steps occur is: A1, A2, A3, possible other steps for other operations, B1, B2, B3, and B4. The red areas represent the expanded durations. The time windows are represented as dotted lines.

With GOLDS, visits which need to be synchronized are inserted just like normal visits, one at a time. It is after and before the insertion where GOLDS changes visit parameters to handle the synchronization. The calculations and parameters depend on the current status of the synchronized visits. In Figure 6.3, we show the steps which are taken when inserting a synchronized visit pair. We will refer to these steps in the explanation below.

First insertion: no visit planned
When none of the two visits v_a for job a and v_b for job b are planned in the current solution, the insertion of v_a happens (Fig. 6.3 steps A1 and A2) like any other visit without any additional pre-processing steps and with the original duration and time windows that come with the chosen option for the job. After the insertion is applied and accepted[7], GOLDS will try to expand the v_a's duration and update the time windows of this visit[8] (Fig. 6.3 step A3) to guarantee an overlap for the new duration. The new duration d'_a is determined as the minimum of the following three options:

[7]Calculating the change in score which is used to accept or reject the operation is done with the original duration and without considering the other unplanned job from this pair.

[8]We explicitly expand the time windows for this planned visit which is the object representing a planned job option in the solution. We do not alter the time windows of job options in the problem instance.

- The duration size v_a can grow to on the current position in the schedule whilst keeping an allowed solution where all buffers have a size of at least 0.
- The maximum time allowed given the original time window for v_a defined by: $d'_a = \frac{TW_a + d_a}{2}$.
- The number of temperature update intervals until a specified temperature (0.5 in our case) is reached times 10, with a maximum of 2 times the original duration added to the original duration.

After determining the new duration, we update the duration in the schedule along with the buffers around v_a. If the time window of v_a is too large to guarantee an overlap for the new duration, we shrink it to the correct size. When shrinking the time window, we make sure not to violate the start time and end time of the current visit. When possible, we try to shrink the time window as uniformly as possible (i.e. we try to move the start backwards the same amount as we move the end forwards).
We expand the visit after the first insertion in order to 'reserve' a space in the schedule for the visit to actually occur in. While there is no overlap without the second visit, the bigger visit allows for a more flexible placement of the second visit (which is inserted with its original duration) as instead of a fixed start time, any start time which allows for a big enough overlap with the first visit is allowed. As the duration is directly linked to the time window, we also have to update these after the first insertion[9].

Second insertion: one visit planned
When one of the two visits has been planned, there is a stricter time window imposed on the second visit v_b . Also, as synchronized visits require the same job option to be chosen, the chosen job option for job b is set to be the same as the one chosen for job a which is planned as v_a. As synchronized visits require two mechanics, v_b cannot be inserted in the same mechanic's schedule where v_a is planned. In order to not miss any insert options, the second visit is inserted with its original duration. The time window for the second visit is defined as $\{start(v_a), start(v_a) + d'_a\}$ (Fig. 6.3 step B1). This time window guarantees that regardless of where v_b is inserted within it, the schedule where v_a is planned has the time reserved to let v_a occur synchronized with v_b.
After v_b has been inserted with its original duration (Fig. 6.3 step B2), GOLDS checks to what size it can be expanded based on the same criteria as the first insertion (Fig. 6.3 step B3), but with the added maximum of d'_a. If $d'_a == d'_b$, v_b gets assigned the same time windows as v_a as we have already calculated that it is the correct time window for this duration and d'_b has been calculated to fit within this time window. However, when v_b has a smaller maximum duration in the current schedule, the algorithm calculates the new time window for v_b and updates the duration of v_a after which v_a is assigned the time window of v_b (Fig. 6.3 step B4). In Figure 6.4, we see that this insertion method has created a scenario with an overlap in the durations of v_a and v_b which is at least as big as the original duration where the synchronized visits can occur. While v_a cannot move to a later moment in schedule A as it is blocked by another (unrelated) job, if this job were to be removed, v_a could move to a later time in schedule A as long as it stays within time window TW'.

[9]While the end of the time window always has to be updated due to the increased duration, with a more complex model, it might be possible to put off updating the start of the time window until after the second insertion.

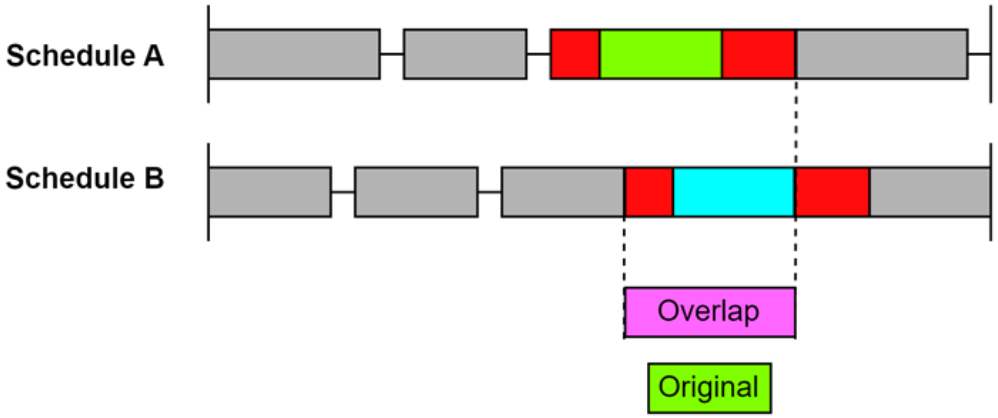

Figure 6.4: The resulting situation with the resulting overlap from the insertion using GOLDS.

6.4.1.2 Deletions (RO)

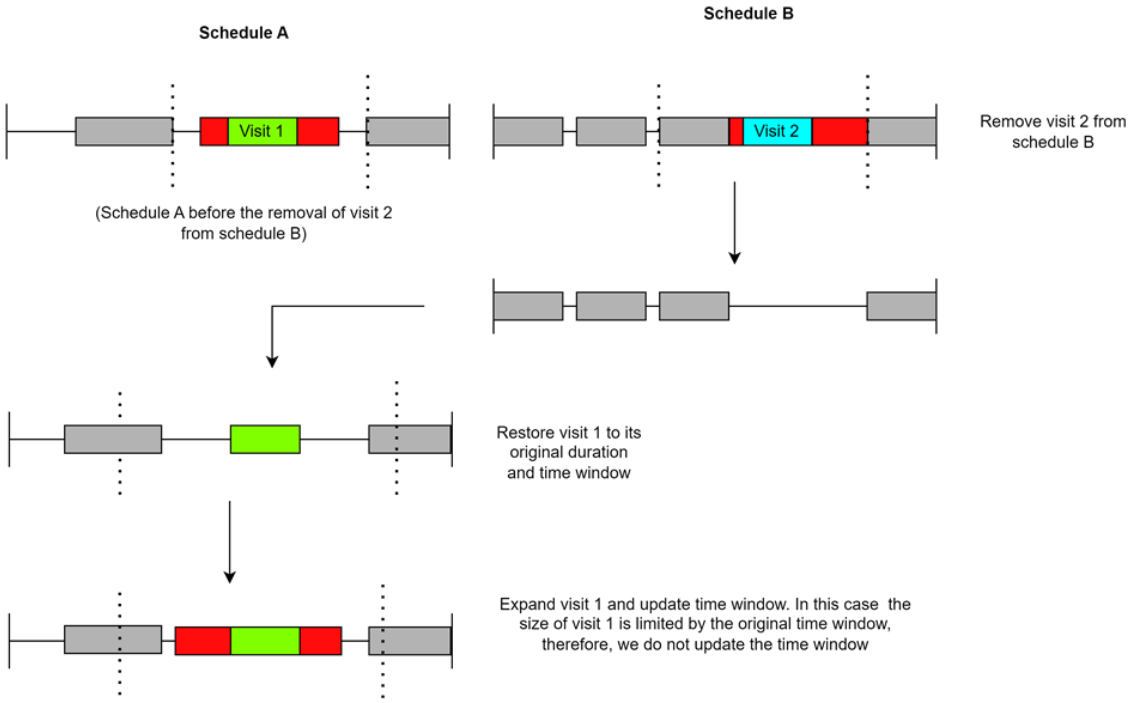

Figure 6.5: Deleting a synchronized visit while the other synchronized visit remains planned.

When deleting a planned synchronized visit, we have two possible scenarios. In the first scenario, the deleted visit was the only one of the synchronized pair which was planned. In that case, we can

just delete the visit like any other visit without further actions.
In the second scenario, both visits were planned, and thus one remains. As we do not know if the remaining visit's duration could be made bigger in the current solution, and because we would like it to be as big as possible to make the insertion of the freshly removed visit a bit easier, we reset the remaining visit to its original duration and time windows and handle it as an insert on the place in the schedule where it already resides. This will determine the correct duration and time window for the remaining visit. This second scenario is illustrated in Figure 6.5.

6.4.1.3 Movements (MBSO)

For synchronized visits, movements within and between schedules are allowed as long as they respect the determined time window and duration for the visit. The only extra added constraint is that a synchronized visit cannot be moved to a schedule where its paired synchronised visit resides. After a movement operation, we do not update time windows and durations as this could interfere with the gradual shrinkage of the duration as the temperature decreases by either expanding or reducing the duration. Future experiments could look into the possibility of resizing the jobs after movements.

6.4.1.4 Option changes (CCOO)

Option changes are allowed for synchronized visits, as long as only one of the jobs from the synchronized pair has a visit planned. For the planned job, the option which is chosen for the visit can be changed just like any other job. When trying to change the option, the original duration and time windows for the new option are used after which the same procedure as for an insertion is followed to update the duration and time window.
As mentioned synchronized visits are used to plan jobs which require two mechanics to work on them at the same time. Therefore, apart from the start time, the selected job option has to be the same for both synchronized visits related to the synchronized job pair. Changing which of the job options for these jobs are planned would require a change for both planned visits which is not only a complicated expensive operation, but we are also unlikely to randomly find a feasible solution. Therefore, we do not allow job option changes for synchronized jobs when both visits are planned. In order to change the chosen option for a synchronized job pair with both visits planned, one of the visits will have to be removed first before the option can be changed. We have tested our algorithm and determined that this sequence of operations does naturally occur in a high enough frequency to not merit a special operator to encourage this behaviour.

6.4.1.5 Optimizing an adjacent pair of visits in a schedule (OPSIO)

As optimizing an adjacent pair in a schedule using the OPSIO operator described in Chapter 6.2.5 is just moving two visits within a schedule, this operator works unaltered on synchronized visits, albeit with some more restrictive time windows than for visits which do not require synchronization.

6.4.1.6 Optimizer

As discussed in Chapter 6.3, we use an exact solver to optimize the current best solution. When dealing with synchronized visits, the optimizer uses the original time windows and durations for these visits. Just like any other visits, the synchronized visits are assigned to the mechanic's schedule where they reside in the solution that is about to be optimized. We have added the following constraints to enforce that the visits are synchronized. While these constraints do link mechanics' schedules in a way that makes the problem more complex as explained in Chapter 2.4, the

pre-assigned visits reduce the complexity enough for the algorithm to quickly come to an optimal solution for the current assignment.

For each set of mechanic's schedules which are linked by a synchronized visit pair (i.e. they both contain one visit of this pair), we need to add a constraint which ensures that the synchronized jobs have the same start time. We add these constraints per pair of mechanics as doing it for all mechanics would lead to a lot of unnecessary added constraints. The added constraint serves to link the start times of two positions across the two schedules where the synchronized visits are placed, while not interfering with the start times of unrelated positions. In order to do this, we use a *big M* notation in the constrains where for two schedules which share a synchronized visit pair, each position has to have a start time which is less than equal than every position from the other schedule plus a big number which allows all start times for unrelated visits.. However, for synchronized visits, we do not add the big number. Therefore, the start times of the synchronized positions have to be equal to one-another thus ensuring synchronization of the jobs. The constraint we add to the model is Constraint 6.14. This constraint goes over all pairs of mechanic schedules $\{m, m'\}$ which are linked by a synchronized visit and requires that the start time of (the visit on) each position in schedule m is less than or equal to that of all positions in schedule m'. This constraint would create an infeasible solution, so therefore we add the *big M* value to the right hand side of the equation for all combinations of positions which do not share a synchronized visit pair which ensures the constraint is only restrictive on the position combinations which share a synchronized visit pair. As we know which visits are synced, we have a constant $s_{v,v'}$ which indicates for each combination of visits v and v' if they form a synchronized visit pair. In order to determine if two positions in two different mechanic's schedules share a synchronized visit pair, we introduce the variable $y_{m,p,m',p'} \in Y$.

(Variable, Binary) $y_{m,p,m',p'} \in Y$:
Indicates with value 1 if position p in schedule m and position p' in schedule m' share a synchronized visit pair and has a value of 0 otherwise.

To ensure this variable has a binary value, we add the following constraint:

$$\begin{gathered}\forall m \in M, \forall p \in \{1, 2, ..., m.visitcount\}, \forall m \in M, \forall p' \in \{1, 2, ..., m'.visitcount\} : \\ y_{m,p,m',p'} \in \{0, 1\}\end{gathered} \tag{6.12}$$

We use Constraint 6.13 to ensure that $y_{m,p,m',p'}$ can only have the value 1 when a synchronized visit pair $\{v, v'\}$ exists for which both $P_{m,p,v'}$ and $P_{m,p,v}$ have a value of 1.

$$\begin{gathered}\forall m \in M, \forall p \in \{1, 2, ..., m.visitcount\}, \forall m' \in M, \forall p' \in \{1, 2, ..., m'.visitcount\}, \\ \forall v \in V_m, \forall v' \in V_{m'} : \\ 0 \leq ((P_{m,p,v} + P_{m',p',v'})) * s_{v,v'} - (2 * y_{m,p,m',p'}) \leq 1\end{gathered} \tag{6.13}$$

Now that we can linearly determine if two positions share a synchronized visit pair, we can add the constraint which requires the synchronized visits to have the same start time:

$$\begin{gathered}\forall \{m, m'\} \text{which share a synced visit :} \\ \forall p \in \{1, 2, ..., m.visitcount\}, \forall p' \in \{1, 2, ..., m'.visitcount\} : \\ T_{m,p} \leq T_{m',p'} + (1 - y_{m,p,m',p'}) * BigM\end{gathered} \tag{6.14}$$

Note that if two schedules m_a and m_b are linked by a synchronized visit pair, the constraint will be added for both pairs $\{m_a, m_b\}$ and $\{m_b, m_a\}$.

6.4.1.7 Parameters and Interval updates

At each temperature interval update, we determine what the maximum duration for each pair of synchronized visits can be based on the original duration and new temperature. After each interval of $500k$ operations, we shrink the maximum duration of all synchronized jobs by 10 seconds until the original duration is reached. We want the original duration to be reached when the temperature has a value of 0.5, so to determine the maximum duration we check how many [10] interval updates are left to reach the temperature of 0.5 and multiply this value by 10 which we then add to the original duration. If the current duration exceeds the maximum duration, we shrink the current duration and time window. Shrinking the time window is done in increments of two time units to allow the duration to be shrunk by exactly half of the time window shrinkage to preserve their relation which guarantees an overlap. As soon as the original duration is reached, the time windows and duration are not decreased any further.
When shrinking the time windows, we try to shrink them such that the parts of the time window without overlap are reduced. Figure 6.6 shows two examples of how time windows and job durations are shrunk during the interval updates where they shrink by 10 seconds at a time.

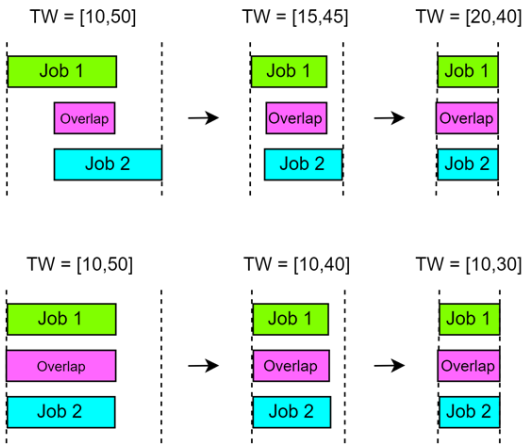

Figure 6.6: Two examples of how time windows shrink.

6.5 Effectiveness of operators

To compare the effectiveness of the operators, we will run the local search algorithm for 500.000.000 cycles or until a buffer of 1200 seconds is reached for each job. We will do this for each problem instance described in Chapter 8 for all situations where one operator (except for SIO and RO) is disabled.

[10] With a minimum of 0 updates when the temperature has a value at or below 0.5.

Chapter 7

Disruption of train movements

The solutions generated by our model are robust to jobs taking more time than specified and trains arriving later than expected. This robustness comes from the built-in buffers where a disruption should not propagate further in the schedule. While the impact of these disruptions is mostly contained to the disrupted job, the job itself might have to be rescheduled. Another possibility is that, for some trains, a change in their movement schedule at the shunting yard is announced during the period for which the mechanic's schedules have been generated. These new movement schedules could be incompatible with the current mechanic's schedules as the job options for these trains will have to be redetermined.
In order to deal with these disruptions, our model should be able to quickly generate an altered schedule based on the new job options for the disrupted jobs while not altering the planning of jobs which have already been completed or are currently being worked on.
We have tested our model's ability to handle disrupted schedules by simulating how the schedules would be executed in real life until a point in time where we make the disruption announcement. The simulation works by letting the mechanics work on their schedule where they randomly use some buffer time and start working on the next job in their schedule as soon as the previous job is done and the next job is available while of course taking travel times into account. In case of synchronized jobs, a mechanic who can start the job earlier than the other synchronized mechanic for that job waits for that mechanic to arrive.
When the simulation reaches the disruption time, all jobs which are completed or in progress are locked at the times they were started in the schedule with their normal duration. Then, we remove the disrupted jobs from the schedules and add them with their new randomly generated options (which all lie in the future) and let the local search algorithm try to put them in the schedule.

As buffers are compressed/used until the disruption time, we cannot compare the score quality of the altered solution which includes the altered job option to the generated solution before the disruption. We use the same neighborhood operators and temperature update intervals while starting with a temperature value of 1.

On the solutions our local search generated for the problem instances described in Chapter 8, we applied a disruption when 10 hours of the schedule had already been completed. We made 25 percent of the already executed jobs use between 0 and 60% of their buffer and disturbed/created new options for 5 non-synchronized jobs.
With these simulation and disruption parameters, our local search was able to find a new feasible solution with all jobs planned within one second for the instances from E, F, G, H, I, J, K, and L.

For instances A and B, five seconds were needed while the solution instance D needed 15 seconds to recover to a feasible solution with all jobs planned. The solution for instance C, our instance with some of the smallest buffers and narrowest time windows, the local search was not able to find a solution with all jobs planned within a few minutes, one job remained unplanned. We do not know if the disruption we have created could have lead to a feasible solution. When running a simulation where only 15 percent of jobs use between 0 and 20 percent of their buffer, a feasible solution with all jobs planned was found in 15 seconds.

Chapter 8

Problem instances

At the moment of writing, the NS did not have problem instances with the required elements to evaluate our model as currently, the scheduling is done ad-hoc and by hand and instances are not kept track of. Therefore, we have created our own problem instances according to the descriptions specified by the NS and based on the data we were able to obtain from the NS.
Our problem instances use the realistic walking distances from the *Grote Binkhorst* shunting yard, the same shift durations (albeit with more mechanics), and similar job durations to what the NS would use. The locations for job options are randomly chosen, as are the number of options where each job can have one, two, or three options. The durations are randomly chosen as either 20, 30, or 40 minutes.

When generating problem instances, we can specify the range in which the buffer for each job must be. Then, the problem generator algorithm will generate a scenario where for each job, a designed minimum buffer is chosen within the range which is guaranteed to be obtainable as we make sure to leave enough room between jobs to have this buffer whilst taking skills and synchronization into account. We can also vary the size of time windows by specifying the size range of the time windows with respect to the job durations (i.e. with a range of $[120\%, 180\%]$, a job with a duration of 20 minutes will get a time window with a size between 24 and 36 minutes.
For each problem instance, we can tune the number of synchronized visits. We also define the number of mechanics per shift which will get assigned a skill set based on their index within the current shift. These skill sets are then used to randomly assign skill requirements to jobs which are intended to be done by the specified mechanic while making sure that the skill requirements no not exceed the skills of the mechanic.

In Table 8.1, we provide an overview of the problem instances we use to evaluate our model.

Table 8.1: Overview of problem instances used to evaluate our model. The column TW-range shows the size range of the time windows with regard to the job duration. #Synced contains the number of synced job pairs which is half of the total number of synced jobs. #Mechanics (S1-S2-S3) contains the number of mechanics available for each of the three shifts (S1-S2-S3). The total number of jobs and job options for each instance can be found in #Jobs and #Options respectively.

Instance	**Buffer range**	**TW-range**	**#Synced**	**#Mechanics**	**#Jobs**	**#Options**
A	500-700	120% - 180%	5	3-2-3	75	130
B	500-700	200% - 400%	5	3-2-3	72	124
C	1-1199	120% - 180%	5	3-2-3	73	136
D	1-1199	200% - 400%	5	3-2-3	75	136
E	1100-1300	120% - 180%	5	3-2-3	62	110
F	1100-1300	200% - 400%	5	3-2-3	61	106
G	601-1799	120% - 180%	5	3-2-3	61	115
H	601-1799	200% - 400%	5	3-2-3	57	103
I	1700-1900	120% - 180%	5	3-2-3	54	94
J	1700-1900	200% - 400%	5	3-2-3	51	90
K	1201-2399	120% - 180%	5	3-2-3	53	94
L	1201-2399	200% - 400%	5	3-2-3	53	93

Chapter 9

Results

In this chapter, we discuss the results from using our model to solve the problem instances we describe in Chapter 8. Unfortunately, even the simplest problem instances which would still be representative of the problem we are trying to solve proved to be too complex to be solved using the exact model we described in Chapter 5. While this proved the need for an alternative method, this leaves us without a baseline to compare our results to. In order to still perform a meaningful analysis, we will use a buffer ratio to compare the performance of our model between the varying problem instances. The buffer ratio is a measure of how well the model was able to assign each job its designed minimum buffer which, as described in Chapter 8 can be reached for all jobs in at least one solution. As we have set our maximum buffer cap at 1200 seconds, designed minimum buffers larger than 1200s will be capped to 1200s. For each test case, we report the percentage of jobs that was able to reach a buffer of at least 60%, 70%, 80%, 90%, 100%, 150%, 200%, and 300% of their designed minimum buffer. The formula for the buffer ratio can be found in Equation 9.1. We have split the results into three tables. Table 9.1 contains the results for the tests on the problem instances A, B, C, and D with a minimum designed buffer lower than 1200 seconds, Table 9.2 contains the results of the tests on problem instances E, F, G, and H with designed minimum buffers which lie between 601 and 1799 seconds. Lastly, the results from the tests on instances I, J, K, and L with designed minimum buffers of at least 1201 seconds.

$$BufferRatio(job) = \frac{Buffer(Job)}{Min(DesignedMinimumBuffer(Job), 1200)} \tag{9.1}$$

When comparing the effects of disabling operators within the same problem instance, we also use the reached buffer cap and score to further evaluate the results. As the score is determined by the buffer size over all jobs, and because the number of jobs varies between instances, the score cannot be used to compare the performance on different problem instances. However, we can use the scores within the instances to compare the impact of disabling operators. For this, we calculate all scores with a buffer cap of 1200 to make sure buffers of the same size are valued equally within the instances over the scenarios with disabled operations as during the solution, different buffer caps can be found.
When we talk about the performance of our model, we are always referring to the quality of the generated solutions.

Table 9.1: The results of our tests using local search on instances A, B, C, and D.

	Disabled	bCap	Score	Percentage of jobs with a buffer ratio of at least:							
				0.6	0.7	0.8	0.9	1.0	1.5	2.	3.0
A	None	721	83,41	97%	97%	97%	95%	86%	22%	11%	4%
A	CCOO	481	82,76	99%	95%	92%	84%	77%	27%	16%	3%
A	MBSO	541	81,41	99%	93%	93%	86%	71%	27%	12%	4%
A	OPISO	541	83,06	100%	99%	96%	90%	79%	32%	15%	4%
B	None	660	85,46	100%	100%	100%	96%	88%	34%	11%	3%
B	CCOO	481	83,70	100%	100%	97%	86%	77%	30%	19%	10%
B	MBSO	541	84,28	100%	100%	96%	90%	78%	33%	16%	7%
B	OPISO	482	84,09	100%	96%	93%	92%	84%	27%	18%	7%
C	None	482	80,61	88%	82%	72%	65%	64%	35%	27%	22%
C	CCOO	482	81,88	91%	84%	76%	68%	59%	38%	30%	19%
C	MBSO	482	79,64	86%	81%	72%	68%	61%	34%	27%	16%
C	OPISO	482	79,82	92%	82%	73%	69%	61%	32%	20%	16%
D	None	661	86,13	96%	91%	86%	80%	79%	43%	32%	24%
D	CCOO	541	84,10	96%	87%	82%	79%	74%	38%	32%	22%
D	MBSO	600	85,14	91%	87%	80%	75%	71%	47%	38%	26%
D	OPISO	481	86,45	95%	89%	84%	80%	70%	45%	33%	24%

Table 9.2: The results of our tests using local search on instances E, F, G, and H.

	Disabled	bCap	Score	Percentage of jobs with a buffer ratio of at least:							
				0.6	0.7	0.8	0.9	1.0	1.5	2.	3.0
E	None	780	79,10	98%	95%	89%	81%	65%	11%	5%	0%
E	CCOO	901	77,52	97%	89%	83%	54%	38%	6%	5%	5%
E	MBSO	840	78,79	98%	92%	83%	78%	70%	14%	5%	0%
E	OPISO	901	77,82	97%	89%	81%	67%	46%	10%	3%	2%
F	None	1200	79,20	98%	98%	98%	98%	90%	15%	10%	2%
F	CCOO	1140	78,96	100%	100%	98%	98%	81%	16%	11%	5%
F	MBSO	1140	79,19	100%	100%	98%	98%	94%	19%	11%	3%
F	OPISO	960	78,86	100%	100%	100%	92%	84%	19%	13%	6%
G	None	1141	79,18	100%	100%	100%	100%	92%	29%	10%	6%
G	CCOO	1141	79,08	100%	100%	100%	100%	90%	32%	15%	5%
G	MBSO	1141	79,02	100%	98%	98%	98%	90%	27%	8%	5%
G	OPISO	960	78,37	100%	98%	98%	85%	82%	34%	15%	6%
H	None	1200	74,20	100%	100%	100%	100%	100%	38%	22%	16%
H	CCOO	1141	73,91	100%	100%	100%	93%	88%	29%	21%	14%
H	MBSO	1140	74,15	100%	100%	100%	100%	93%	40%	16%	10%
H	OPISO	1080	73,90	100%	100%	100%	100%	83%	40%	24%	12%

Table 9.3: The results of our tests using local search instances I, J, K, and L.

	Disabled	bCap	Score	Percentage of jobs with a buffer ratio of at least:							
				0.6	0.7	0.8	0.9	1.0	1.5	2.	3.0
I	None	1200	70,36	100%	100%	100%	100%	100%	38%	20%	0%
I	CCOO	1200	70,33	100%	100%	100%	100%	96%	40%	13%	4%
I	MBSO	1200	70,36	100%	100%	100%	100%	100%	47%	13%	0%
I	OPISO	1200	70,36	100%	100%	100%	100%	100%	40%	13%	2%
J	None	1200	66,53	100%	100%	100%	100%	100%	52%	37%	19%
J	CCOO	1200	66,53	100%	100%	100%	100%	100%	38%	31%	15%
J	MBSO	1200	66,53	100%	100%	100%	100%	100%	31%	23%	15%
J	OPISO	1200	66,53	100%	100%	100%	100%	100%	29%	27%	15%
K	None	1200	69,08	100%	100%	100%	100%	100%	52%	28%	9%
K	CCOO	1200	69,08	100%	100%	100%	100%	100%	39%	22%	15%
K	MBSO	1140	69,08	100%	100%	100%	100%	98%	43%	22%	11%
K	OPISO	1200	69,08	100%	100%	100%	100%	100%	48%	24%	7%
L	None	1200	69,08	100%	100%	100%	100%	100%	28%	24%	17%
L	CCOO	1200	69,08	100%	100%	100%	100%	100%	39%	22%	15%
L	MBSO	1200	69,08	100%	100%	100%	100%	100%	33%	30%	15%
L	OPISO	1200	69,08	100%	100%	100%	100%	100%	30%	24%	17%

9.1 Experimental setup

All results were generated on a *11th Gen Intel(R) Core(TM) i7-1185G7 @ 3.00GHz* processor with 16 GB of DDR4 memory. With this hardware, running the local search for $500M$ cycles on one of the problem instances takes about 10 minutes to complete when no other processes are active.
We ran every problem instance for at most $500M$ cycles but early termination was possible when every buffer was at least 1200 seconds. As described in Chapter 6.5, we have run 3 cases where a operator was disabled for each problem instance along with the scenario where all operators were enabled.

9.2 MIP results

Unfortunately, the MIP we have formulated in Chapter 5 did not find any feasible solutions. Even with Gurobi configured to find and output a feasible solution as soon as possible, after 10 hours of computing, no solution was found. Therefore, we are not able to compare the results generated by our local search to optimal results.

9.3 Analysis of results

We provide an overview of all the results in Tables 9.1, 9.2, and 9.3. The buffer ratios are notated as $BR(number)$ where, for example, the column $BR0.8$ contains the percentage of jobs for each experiment with a buffer ration ≥ 0.8. The column *bCap* contains the buffer cap, which is the highest value a buffer may have for the score calculation, that was reached for the instance. For

each problem instance and with every test, our method was able to find a feasible solution where all jobs were planned.

9.3.1 Effectiveness of operators

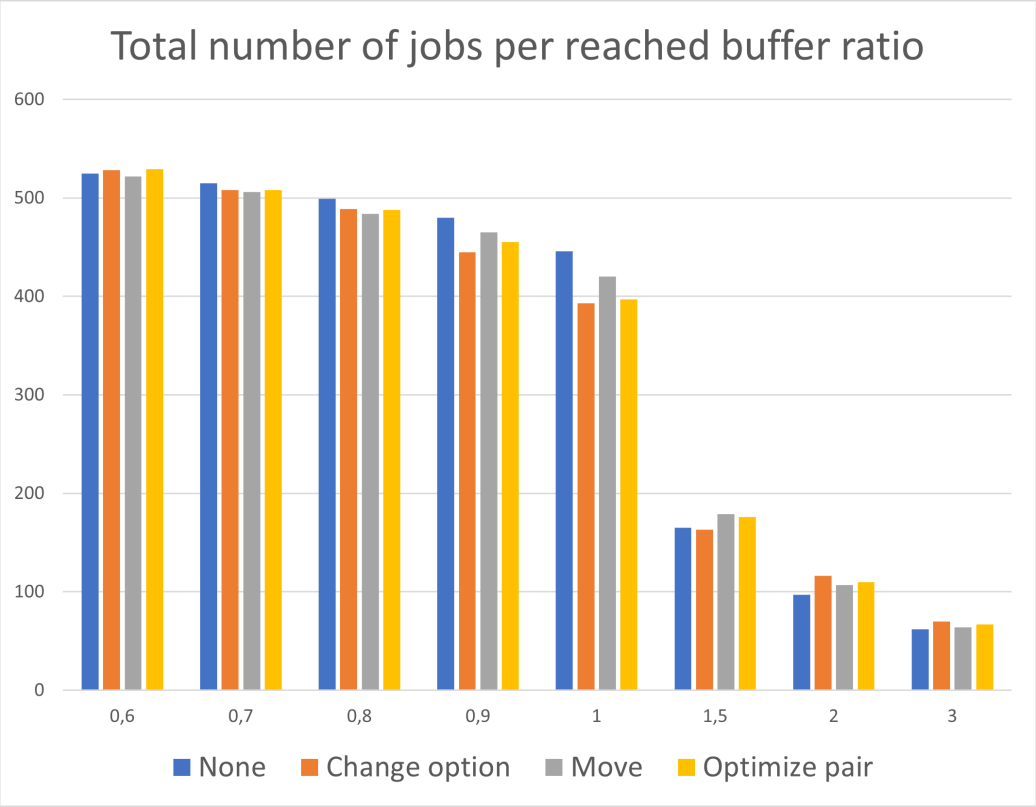

Figure 9.1: The total number of jobs with a buffer ratio of at least the buffer ratio on the x-axis grouped per disabled operator from a total of 541 jobs over problem instances A, B, C, D, E, F, G, and H.

As discussed in Chapter 6.5, we want to evaluate the usefulness and thereby effectiveness of our operators. We will now look at the results which were achieved when disabling an operator and how this compares to the situations where it was not disabled. In Figure 9.1, we plot the total number of jobs which reached at least the buffer ratio on the x-axis for the problem instances A, B, C, D, E, F, G, and H grouped per disabled operator to compare the overall effectiveness of the operators. We have excluded problem instances I, J, K, and L as their very large designed minimum buffers cause them to be easily solved. In the cases where a difference was observed for these instances, the difference is too small to attribute it to the disabled operator rather than to randomness.
In Figure 9.1, we see that for buffer ratios in the range from 0.7 up to and including 1.0, the best results came from the tests where all operators were enabled. While the tests with all operators enabled yielded slightly worse results for the buffer ratio of 0.6, the difference is marginal. While for the buffer ratios of 1.5 and higher, the tests with an operator disabled yielded more jobs which

reached these ratios, this came at the cost of worse performance on all but one of the lower buffer ratios which makes these results undesirable.

9.3.1.1 CCOO

Based on our results, the impact of disabling the CCOO seems to be highly dependant on the problem instance without a clear indication of which aspect of the problem instance causes these differences. With the exception of problem instance C[1], the tests where the CCOO was disabled all yielded a worse result, it seems like the CCOO has a meaningful positive impact on the result at best and no impact at worst. This is further supported by the graph showing the total number of jobs per reached buffer ratio in Figure 9.1.

9.3.1.2 MBSO

In our results, we notice that for problem instances with designed minimum buffer of at most 1200 seconds, not having the MBSO operator to move visits between schedules seems to almost universally lead to a less desirable result. While for problem instances E and F, disabling the MBSO resulted a slightly better performance on the buffer ratios, having it enabled yielded a higher scores based on our robustness functions. Furthermore, on the other problem instances with minimum designed buffers which can be smaller than 1200 seconds, disabling the MBSO resulted in a noticeably worse result. We therefore conclude that, in general, disabling the MBSO yields a worse result and therefore, the MBSO does contribute to a more desirable solution. This is further supported by the graph showing the total number of jobs per reached buffer ratio in Figure 9.1 where we see that, for buffer ratio between 0.7 and 1.0, disabling the MBSO yields less jobs which reached each buffer ratio.

9.3.1.3 OPISO

The results for test runs with the OPISO disabled show that for all instances, with the exception of instance E where a marginally better result is obtained over the run with all operators disabled, disabling the OPISO leads to a similar solution a best and a quite worse solution at worst. When looking at Figure 9.1, we see that over all, disabling the OPISO yields to less jobs reaching each of the buffer ratios compared to the tests with all operators enabled.

9.3.2 Differences between instances

As we have used a number of varying problem instances to evaluate our model, we will now explore the differences in performance of our model over the variations in our problem instances. For this evaluation, we will use the results generated with all operators enabled. As scores are meaningless for comparing different test cases, we will instead use the percentage of visits which used the different buffer ratios to compare the results

9.3.3 Tight VS loose buffer ranges

In our experiments, we have used tight buffer ranges where all designed minimum buffers lie within 200 seconds of each other as well as loose buffer ranges where designed minimum buffers within an instance can vary by as much as 1199 seconds. When looking at our results, we see that for

[1] And the test cases J, K, and L with designed minimum buffers larger than 1200 seconds which we do not consider for this .

all instances with a designed minimum buffer greater than 1200 seconds, there seems to be no noticeable difference in the quality of the outcome. An interesting observation is that for designed minimum buffers smaller than 1200, the performance of our model on instance A (tight) is better than that on instance C (loose) which was generated with the same parameters as instance A except for the designed minimum buffer range. The same pattern can be seen between instance B (tight) and instance D (loose) which are also generated with the same parameters except for the designed minimum buffer range.

The problem instances where the designed minimum buffer range is centered around 1200 seconds have results which tell us the exact opposite. For instance G (loose), we have found a higher-quality solution than for the corresponding tight problem instance E. We have made the same observation for instance H (loose) and corresponding instance F (tight).

As expected, the buffer cap for the visits with loose designed minimum buffer size ranges was limited by the smaller buffers, although it ended up higher than we anticipated as our heuristic found better buffers for the visits with the smallest designed minimum buffers than our instance generator had set as a minimum designed buffer.

9.3.4 Small VS large time windows

In our experiments, instances A, C, E, G, I, and K have jobs with small time windows with are at most 180% of the job durations. Their corresponding respective instances with large time windows which are between 200% and 400% of the job durations are instances B, D, f, H, J, and L. For the instances with all designed minimum buffers greater than 1200 seconds, we did not observe any difference in the performance between small and large time windows. For the other instances, the performance, measured by the percentage of jobs which achieved the buffer ratios and compared to the problem instances with the same parameters except for the time window sizes, of our model was better on the large time windows while still generating acceptable feasible solutions for the smaller time windows. As larger time windows allow for more movement and as GOLDS benefits from larger time windows to allow movement of synchronized jobs, this difference can easily be explained. We will discuss the impact of these findings on our evaluation of GOLDS in Chapter 10.

9.3.5 Impact of buffer size

While we have used varying designed minimum buffer sizes, we do not see any patterns in how the solution quality over problem instances with varying designed minimum buffers vary. From the results, it seems that the designed minimum buffer size has an impact on the performance in combination with the other variations, but not merely on its own. As with the other variations, for instances with all designed minimum buffers greater than 1200 seconds, there are no interesting observations to report on.

Chapter 10

Conclusion and future work

From the results we have seen in Chapter 9, we can draw some conclusions about our solution method and our innovation on synchronized job planning. Based on these conclusions, we will point out some directions for future research and possible improvements.

We have set out to create a solution method which can create a robust schedule for mechanics at shunting yards while taking skills, time windows, and especially synchronization into account. Based on the results we have gathered, we are confident in saying that we achieved our goal and have created this solution method. Using the robustness measure we have defined, all our problem instances have a buffer for each visit which is at least 100% of the designed minimum buffer for that visit in 82% of all instances where not all jobs were designed to have a buffer of at least 1200 seconds. For the instances with designed minimum buffers of at least 1200 seconds, our approach was able to assign each job a buffer of 1200 seconds. Furthermore, we have found a feasible solution with all visits planned for each instance.

In Chapter 7, we have studied our model's ability to handle disruptions during the executing of the schedule which require some jobs to be rescheduled. We have seen that in all but one case, new schedules where all jobs were planned could be created within seconds. Future research could extend this part of our research by looking into both disruptions which lead to feasible schedules and the performance on these disruptions as well as infeasible disruptions how the model should handle them.

An improvement which could be made in future research is implementing a better robustness measure which, for example, takes the size of a job and the probabilistic buffer use for a job of this size and time into account. There are many different ways to measure robustness, therefore finding the best robustness measure for the `SYMSP` is something which we kept out-of-scope, but could be an interesting continuation of our research. A good starting point for this research would be the work by Roel van den Broek et al. (2018) which looks at the robustness of shunting plans.
More advanced simulations can be used to determine the usefulness of different robustness measures. These simulations could also include mandatory break times which we have excluded as these are basically jobs which can only be done by one mechanic of the mandatory break duration with a time window set by the break regulations. Along with breaks, mechanics could have preferences with regard to how much they have to walk each day. Future research could take these preferences into account.

Our synchronization method GOLDS makes use of artificially increased job durations and corresponding time windows to guarantee synchronization. As all of the jobs in our instances have been planned and synchronized when needed, we can conclude that our method is effective. As our method makes use of time windows and job durations for the synchronization while always guaranteeing a feasible solution, small time windows for the jobs can pose a restriction on the effectiveness of our method. We believe that the difference in performance on small time window instances compared to large time window instances observed in Chapter 9.3.4 can be explained partially by this property of GOLDS. As GOLDS is less effective on smaller time windows, future research could look into adapting the train schedules on the shunting yards to ensure larger time windows. This is something that the NS could look into when possibly integrating our method into their shunting yard planning method. For a more theoretical study or for another application, methods which use a dynamic scoring function to 'protect' smaller buffers for cases with a large variance in buffer size could be explored which would allow the buffer cap to be heightened even when smaller buffers exist which cannot be increased any further.

The operators we have introduced for our local search in Chapter 6 are all useful for finding a good solution, but there are some problem instances where disabling an operator yielded a better result. We believe this might have to do with the operators being chosen randomly with an equal possibility where some problem instances have little use for specific operators. A disabled operator would therefore allow the other useful operators to be chosen more often and thus lead to a better result. We think that a future study could look into better ratios for the operator selection or possibly dynamic operator selection based on the current state of the solution.
This leads us to the final recommendation for any future research which looks into improving our method: we suggest generating more tests and preferably using real world data which the NS can hopefully provide by then. We have proven that our method generates good results on a variety of sets, but in order to do more fine-tuning and optimizations, more data is needed.

It would still be nice to compare our results to an exact solution, but we do not think that this will be realistic to do anytime soon without compromises on the complexity of the problem or significant advances in computing power.

While precedence constraints were not part of our scope, they could be added in future research where jobs would have to be executed in a certain order. When adding precedence constraints, time window manipulations might provide the same compromise between freedom of movement and complexity as is offered by GOLDS for synchronization.

Bibliography

Bredstrom, D., & Rönnqvist, 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*, (2007/7).

Bredström, D., & Rönnqvist, M. (2008). Combined vehicle routing and scheduling with temporal precedence and synchronization constraints. *European journal of operational research*, *191*(1), 19–31.

Cappanera, P., Requejo, C., & Scutellà, M. G. (2020). Temporal constraints and device management for the skill vrp: Mathematical model and lower bounding techniques. *Computers & Operations Research*, *124*, 105054.

Decerle, J., Grunder, O., El Hassani, A. H., & Barakat, O. (2018). A memetic algorithm for a home health care routing and scheduling problem. *Operations research for health care*, *16*, 59–71.

Diepen, G., Van Den Akker, M., Hoogeveen, H. a., & Smeltink, J. (2012). Finding a robust assignment of flights to gates at amsterdam airport schiphol. *Journal of Scheduling*, *15*(6), 703–715.

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

Drexl, M. (2012). Synchronization in vehicle routing—a survey of vrps with multiple synchronization constraints. *Transportation Science*, *46*(3), 297–316.

Fikar, C., & Hirsch, P. (2017). Home health care routing and scheduling: A review. *Computers & Operations Research*, *77*, 86–95.

Fink, M. (2016). *The vehicle routing problem with worker and vehicle synchronization: Metaheuristic and branch-and-price approaches* (Dissertation). Technische Universität München. München.

Gemeente Utrecht. (2021). Milieuzone. https://www.utrecht.nl/wonen-en-leven/gezonde-leefomgeving/luchtkwaliteit/milieuzone/

Golden, B. L. (1975). *Vehicle routing problems: Formulations and heuristic solution techniques* (tech. rep.). Massachusetts inst of tech and Cambridge operations research center.

Haddadene, S. R. A., 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.

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.

Paraskevopoulos, D. C., Laporte, G., Repoussis, P. P., & Tarantilis, C. D. (2017). Resource constrained routing and scheduling: Review and research prospects. *European Journal of Operational Research*, *263*(3), 737–754.

Pillac, V., Guéret, C., & Medaglia, A. (2011). On the technician routing and scheduling problem. *Metaheuristics International Conference (MIC) 2011*, S2–40.

Polnik, M., Riccardi, A., & Akartunalı, K. (2021). A multistage optimisation algorithm for the large vehicle routing problem with time windows and synchronised visits. *Journal of the Operational Research Society*, *72*(11), 2396–2411.

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.

Redjem, R., & Marcon, E. (2016). Operations management in the home care services: A heuristic for the caregivers' routing problem. *Flexible Services and Manufacturing Journal*, *28*(1), 280–303.

Roel van den Broek. (2022). *Towards a robust planning of train shunting and servicing* (Doctoral dissertation). Utrecht University.

Roel van den Broek, Hoogeveen, H., & Van den Akker, M. (2018). How to measure the robustness of shunting plans. *18th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2018)*.

Roel van den Broek, 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.

Solomon, M. M. (1987). Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations research*, *35*(2), 254–265.

Van den Akker, M., Hoogeveen, H., & Van Kempen, J. W. (2012). Using column generation to solve parallel machine scheduling problems with minmax objective functions. *Journal of Scheduling*, *15*(6), 801–810.

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.

Yan, X., Xiao, B., Xiao, Y., Zhao, Z., Ma, L., & Wang, N. (2019). Skill vehicle routing problem with time windows considering dynamic service times and time-skill-dependent costs. *IEEE Access*, *7*, 77208–77221.

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.