TU WIEN Informatics

Automated Scheduling for Automotive Supplier Paint Shops and Teeth Manufacturing

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

Dipl.-Ing. Felix Winter
Registration Number 0825516

to the Faculty of Informatics

at the TU Wien

Advisor: Priv.-Doz. Dr. Nysret Musliu

The dissertation has been reviewed by:

Andrea Schaerf	Guido Tack

Vienna, 3rd November, 2021

Felix Winter

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Dipl.-Ing. Felix Winter

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 3. November 2021

Felix Winter

Acknowledgements

I would like to express my sincere gratitude to my advisor Priv.-Doz. Dr. Nysret Musliu. Writing this thesis would not have been possible without his ongoing support, encouragement and his deep knowledge in the field.

The financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development and the Christian Doppler Research Association is gratefully acknowledged. Furthermore, I would like to thank all people who were my project colleagues during my PhD studies. Their comments and feedback helped me to improve this thesis in many places.

Deepest gratitude goes to my wife Eni, my parents, and my brother, who have supported me throughout my life and aided me through many hard times. Without their help, finishing this work would not have been possible.

Kurzfassung

Viele Fabriken der heutigen Zeit verwenden einen hochautomatisierten Produktionsprozess um täglich große Stückzahlen an Artikeln effizient und ressourcenschonend produzieren zu können. Die dadurch entstehenden komplexen Produktionsabläufe machen eine manuelle Produktionsplanung zu einer schwierigen Aufgabe und erfordern oftmals den Einsatz von automatischen Lösungsverfahren. In der wissenschaftlichen Literatur wurden in der Vergangenheit viele industrielle Produktionsplanungsprobleme untersucht und algorithmisch gelöst. Durch die Vielzahl an unterschiedlichen Produktionsumgebungen die in den verschiedenen Industriesparten zu finden sind, gibt es allerdings nach wie vor NP-schwere Planungsaufgaben für die noch keine automatischen Planungsalgorithmen vorgeschlagen wurden.

Diese Dissertation stellt zwei wichtige neue Produktionsplanungsprobleme vor, welche in den Lackieranlagen der Automobilzuliefererindustrie sowie bei der Produktion von Zahnprothesen auftreten. Beide Probleme sind NP-schwer und unterscheiden sich von verwandten Aufgabenstellungen durch eine Menge von einzigartigen Restriktionen sowie Optimierungszielen, welche die Entwicklung von neuen effizienten Lösungsmethoden erfordern. Aus diesem Grund stellt diese Arbeit zusätzlich zu einer formalen Problemspezifikation und Komplexitätsanalyse erstmals eine Reihe von neuen exakten und heuristischen Problemlösungsverfahren für beide Probleme vor.

Außerdem identifiziert und löst die Dissertation wichtige Teilaufgabenstellungen der untersuchten Produktionsplanungsprobleme. Dabei stellt sich heraus, dass die vorgeschlagenen Lösungsmethoden der Teilprobleme auch für die Herangehensweise an andere NP-schwere Probleme aus der Literatur von Nutzen sein können. Zum Beispiel wird eine neue Nebenbedingung bezüglich der Editierdistanz zwei gegebener Zeichenketten in dieser Arbeit gemeinsam mit einem Lösungsverfahren vorgestellt, welche auch zur effizienten Lösung eines komplexen Problems außerhalb des Bereichs der Produktionsplanung verwendet werden kann.

Schließlich inkludiert diese Arbeit Ergebnisse einer umfassenden Evaluierung aller vorgeschlagenen Methoden. Dabei wurde zur Auswertung eine neu eingeführte Sammlung an Probleminstanzen verwendet, die realistische Planungsszenarien aus den Produktionsstätten der Automobilzuliefererindustrie und der Zahnprothesenherstellung abbilden. Die experimentellen Ergebnisse zeigen, dass die entwickelten exakten Lösungsverfahren in der Lage sind optimale Lösungen sowie untere Schranken für mehrere Instanzen zu

produzieren. Außerdem lässt sich aus der durchgeführten Evaluierung schließen, dass die vorgeschlagenen Heuristiken hochqualitative Lösungen für alle ausgewerteten realistischen Instanzen finden konnten.

Abstract

Nowadays, many modern day factories have migrated towards a highly-automated production process to efficiently create large quantities of products every day. Thus, production scheduling tasks are often challenging for human planners and there is a strong need for automated solution methods to find optimized schedules. Although various practical scheduling problems have been studied in the literature, still many novel NP-hard problems that originate from the industry remain to be investigated due to the unique requirements that arise from different application domains.

This thesis introduces two important scheduling problems that arise from real-life applications in the paint shops of the automotive supply industry and in the manufacturing of artificial teeth for dentures. Both investigated problems, which are called the paint shop scheduling problem and the artificial teeth scheduling problem, are NP-hard and include unique constraints as well as solution objectives that cause the need for efficient novel solution methods to solve large-scale problem instances. Therefore, the thesis proposes a range of innovative exact techniques, metaheuristics, hybrid methods, and hyper-heuristic solution approaches in addition to providing a formal specification and complexity analysis.

Moreover, this work identifies and solves important sub-problems that appear within the investigated real-life production scheduling problems and can also be useful when approaching other NP-hard problems. For example, the thesis introduces an important novel global constraint that captures restrictions on the edit distance between two strings and can be used to model the paint shop scheduling problem. The work further proposes an efficient propagation algorithm for this constraint which could be successfully used to improve state-of-the-art results on another challenging problem from the literature.

To experimentally evaluate all the proposed solution methods, the thesis provides a collection of benchmark instances that include real-life scheduling scenarios from factories of the automotive supply industry and teeth manufacturing. Computational results show that the introduced exact techniques could be successfully used to achieve several optimality results and can provide lower bounds for many instances. An extensive empirical evaluation further demonstrates that the proposed metaheuristics and hybrid techniques can be successfully used to produce high-quality schedules even for large real-life scheduling scenarios.

Contents

CHAPTER 1

Introduction

Production scheduling problems arise in many areas of industrial manufacturing and nowadays many companies have started to migrate towards a highly-automated production process where products need to be efficiently processed in a large-scale production. Therefore, the production scheduling tasks at modern production sites become more and more complex, which makes it challenging for human planners to design efficient schedules and creates a strong need for automated scheduling approaches to find optimized solutions.

Unfortunately, solving practical production scheduling problems is in general a challenging task and even basic problem variants have been shown to be NP-hard (e.g. job shop scheduling [GJS76] and parallel machine scheduling [GJ79]). In the literature a large variety of related problems has been studied and several methods from the areas of Operations Research and Artificial Intelligence have been successfully used to efficiently solve practical problems. These include methods from constraint programming (CP) [RvBW06], mixed integer programming (MIP) [JLN+09], Boolean Satisfiability (SAT) [BHvMW21], metaheuristics [GP19], hyper-heuristics [BGH+13, DKÖB20], and hybrid techniques [VHM10].

Although this field already has been the subject of intensive research in the past, a lot of novel scheduling problems originating from industry still remain to be investigated due to the plethora of diverse application domains and the recent trend towards full automation. The unique requirements of these novel problems often cause existing solution techniques to be inapplicable or inefficient and thus require the development of novel efficient solution approaches.

For example, in the paint shops of the automotive supply industry an innovative and highly-automated production process is utilized every day to paint large quantities of synthetic material pieces that are demanded by car manufacturers. There, the task of creating efficient production schedules includes making decisions not only about the

production sequence but also on how to efficiently group demanded items onto customized carrying devices while at the same time fulfilling various complex resource constraints. Furthermore, challenging solution objectives regarding carrier device setup costs and color changes in the production sequence should be minimized in high-quality solutions. These unique constraints and cost objectives arising in paint shops of the automotive supply industry cause the related scheduling problem to be substantially different from the many previously studied NP-hard problems from the automotive industry such as e.g. car sequencing [PKW86, SCNA08]. Therefore, there is still a strong need to develop innovative automated scheduling methods that are capable to create efficient paint shop schedules for the automotive supply industry.

Artificial teeth manufacturing is another important application domain which nowadays uses a highly-automated production process to produce large amounts of prosthetic teeth in various shapes and colors. The task of finding efficient production schedules that arises at real-life production sites in this area is related to single machine batch scheduling problems that originate from other industrial branches (such as e.g. [PTM20, TB20]) of which many variants have been shown to be NP-hard [PK00]. However, in contrast to previously investigated batch scheduling problems, in teeth manufacturing the set of jobs is not predetermined and the task of creating schedules requires making decisions about how to group customer demands efficiently into jobs while at the same time several resource- and eligibility constraints have to be considered. Thus, the investigation of novel efficient solution techniques is required in this area.

Finding efficient production schedules in both the paint shops of the automotive supply industry and artificial teeth manufacturing is a task that is currently usually done by human expert planners. However, the many requirements, constraints and minimization objectives that have to be considered make it very challenging to find efficient schedules manually. Therefore, the development of automated solution methods that can assist human planners in the scheduling process has a large potential to reduce the amount of required time spent on handling complex scheduling scenarios. Furthermore, efficient exact and metaheuristics techniques that are able to improve manually created schedules have a great potential to reduce costs as well as industrial waste by providing solution schedules that minimize machine setup costs, overproduction and the production makespan.

1.1 Aims of This Thesis

The main goal of this PhD thesis is to develop innovative automated solution methods that can efficiently solve novel real-life production scheduling problems as they appear in the automotive supply industry and artificial teeth manufacturing. In the course of the thesis we aim to identify and formally define new problems as well as sub-problems from these areas. Additionally, we aspire to analyze their complexity and study their relation to existing problems from the literature.

Furthermore, we aim to investigate novel problem formulations and modeling strategies from the area of constraint programming and mixed integer programming which can be

utilized as exact approaches with state-of-the-art solving technology from the literature. To provide solution methods for large real-life problem instances which cannot be tackled efficiently using exact techniques we aim to develop novel metaheuristic techniques as well as hybrid techniques and aspire to investigate the performance of automated algorithm selection techniques using hyper-heuristics.

The main aims of this thesis are:

- Mathematically specify new challenging production scheduling problems that originate from the automotive supply industry and artificial teeth manufacturing. In addition to providing a formal definition for these applications, we aim to perform a complexity analysis and analyze their relation with existing problems in the literature.
- Provide a collection of problem instances that include a set of challenging real-life scheduling scenarios and can be used as benchmarks to evaluate solution approaches for these domains.
- Develop and evaluate exact methods through the investigation of different modeling strategies from the area of constraint programming and mixed integer programming.
- Introduce novel metaheuristic methods for real-life problem instances that cannot be solved optimally within reasonable time due to a very large search space and complex constraints.
- Propose innovative hybrid approaches that combine the developed exact methods with the investigated metaheuristic techniques and are thereby able to further improve results on challenging real-life instances.
- Investigate and evaluate a hyper-heuristic approach that utilizes novel low-level heuristics for real-life problem instances from these areas.
- Identify and solve challenging sub-problems and constraints that appear in the investigated problems. Develop innovative solution methods for these sub-problems that can be useful also for other application domains.

1.2 Contributions

The following are the main contributions of this thesis:

- We introduce and formally specify the paint shop scheduling problem which arises in real-life paint shops from the automotive supply industry. In addition to a complexity analysis we provide a set of benchmark instances that include challenging real-life scheduling scenarios from the industry.

- We provide two constraint modeling formulations for the paint shop scheduling problem that can be used together with state-of-the-art constraint solving technology as an exact solution method. Using this approach we provide optimal solutions for 9 benchmark instances.
- To efficiently solve challenging large real-life instances of the paint shop scheduling problem, we develop several innovative construction heuristics and metaheuristic techniques.
- We identify an important novel global constraint on string distance metrics that appears not only in the paint shop scheduling problem, but also in another NP-hard optimization problem that arises in the area of computational biology. Additionally, we provide an efficient propagation algorithm for constraint programming solvers that utilize lazy clause generation. Experimental results show that this method can improve state-of-the-art results on instance sets for two applications.
- We identify and solve a challenging color change sub-problem that appears in the paint shop scheduling problem using heuristic as well as exact solution techniques. Furthermore, we introduce a hybrid approach for the paint shop scheduling problem that can utilize a combination of heuristic approaches together with exact approaches for this sub-problem within the framework of large neighborhood search.
- We introduce and define a novel challenging real-life machine scheduling problem from the area of artificial teeth manufacturing which we further call the artificial teeth scheduling problem. Furthermore, we provide a collection of benchmark instances which includes a set of real-life scheduling scenarios.
- As an exact solution method for the artificial teeth scheduling problem we introduce a constraint modeling approach, which we evaluate using state-of-the-art MIP and CP solvers. Using these techniques we can provide optimal results for 4 of the benchmark instances.
- To efficiently solve large realistic instances of the artificial teeth scheduling problem, we introduce an innovative construction heuristic together with a metaheuristic approach based on simulated annealing.
- We introduce novel low-level heuristic operators for the artificial teeth scheduling problem that can be utilized within hyper-heuristic solution approaches. Further, we perform an evaluation of the proposed low-level heuristics together with state-of-the-art perturbation based hyper-heuristics and show that these methods can improve results on several of the benchmark instances.
- We investigate a solver-independent constraint model for a challenging workforce scheduling problem that also arises in similar industrial manufacturing environments as they appear in the automotive supply industry or teeth manufacturing. We extensively evaluate our model using state-of-the-art MIP and CP solvers on a set

of benchmark instances and our experiments show that the modeling techniques can be successfully used to produce competitive results. Furthermore, we investigate an alternative MIP encoding for the modeling of working weekend constraints which improves the initial linear relaxation bound for the majority of the benchmark instances from the literature.

- Several benchmark instances provided in this thesis were further submitted to the 2019 and 2021 editions of the MiniZinc Challenge [SBF10]. This annually hosted competition is of high value for the constraint programming community as it provides the possibility to evaluate and improve state-of-the-art solving technology on challenging benchmark problems.

1.3 Publications

The contributions that are presented in this thesis have been included in the following publications:

- Journals
  - Felix Winter and Nysret Musliu. **Constraint-based Modeling for Scheduling Paint Shops in the Automotive Supply Industry**, ACM Transactions on Intelligent Systems 2021 [WM21a]
  - Felix Winter and Nysret Musliu. **A Large Neighborhood Search Approach for the Paint Shop Scheduling Problem**, Journal of Scheduling 2021 [WM21b]
  - Emir Demirović, Nysret Musliu, Andreas Schutt, Peter J. Stuckey, Felix Winter. **Solver-Independent Models for Employee Scheduling**, (currently under submission) [DMS+21]
- Conferences
  - Felix Winter, Christoph Mrkvicka, Nysret Musliu and Jakob Preininger. **Automated Production Scheduling for Artificial Teeth Manufacturing**, ICAPS 2021 [WMMP21]
  - Felix Winter, Nysret Musliu, Peter J. Stuckey. **Explaining Propagators for String Edit Distance Constraints**, AAAI 2020 [WMS20]
  - Felix Winter, Nysret Musliu, Emir Demirović and Christoph Mrkvicka. **Solution Approaches for an Automotive Paint Shop Scheduling Problem**, ICAPS 2019 [WMDM19]

The work presented in Chapter 8 is planned to serve as the basis for another publication in future work. Furthermore, during the time of his PhD studies the author of this thesis coauthored another publication on the paint shop scheduling problem as well as two

additional papers that investigate automated problem-solving methods for other real-life applications:

- Wolfgang Weintritt, Nysret Musliu, Felix Winter. **Solving the paintshop scheduling problem with memetic algorithms**, GECCO 2021 [WMW21]
- Tobias Geibinger, Lucas Kletzander, Matthias Krainz, Florian Mischek, Nysret Musliu, Felix Winter. **Physician Scheduling During a Pandemic**, CPAIOR 2021 [GKK+21]
- Johannes Vass, Nysret Musliu, Felix Winter. **Solving the Production Leveling Problem with Order-Splitting and Resource Constraints**, PATAT 2021 [VMW21]

1.4 Organization

In the following chapter we formally introduce the paint shop scheduling problem and review literature on related problems. Furthermore, we study the problem's complexity and provide a set of realistic benchmark instances. Afterwards, in Chapter 3 we model the problem using two constraint programming formulations and discuss the results of an extensive evaluation of the models.

Chapter 4 then proposes heuristic and metaheuristic methods to approach challenging large real-life instances of the paint shop scheduling problem. In addition to providing a constructive heuristic and local search based approach, we further identify and solve a NP-hard coloring sub-problem of the paint shop scheduling problem. Then, we utilize the proposed solution methods to introduce a large neighborhood search operator for the paint shop scheduling problem that is able to hybridize exact and heuristic techniques. At the end of Chapter 4, results of an extensive experimental evaluation with all heuristic solution methods for the paint shop scheduling problem are summarized.

We introduce a novel string-edit-distance global constraint which appears in the paint shop scheduling problem in Chapter 5, where we further propose an efficient propagator as well as an explanation strategy for this constraint. Additionally, we evaluate the efficiency of the novel global constraint based on experiments with two applications.

In Chapter 6 we introduce the artificial teeth scheduling problem, provide an overview on related problems from the literature, and give a set of real-life problem instances. Chapter 7 then introduces exact and heuristic solution methods for the artificial teeth scheduling problem which are evaluated based on a set of experiments at the end of the chapter.

Afterwards, in Chapter 8 we propose several low-level heuristic operators that can be used to solve the artificial teeth scheduling problem using hyper-heuristics. We further show empirically the effectiveness of state-of-the-art hyper-heuristics on the set of evaluated benchmark instances.

Chapter 9 investigates a solver-independent constraint modeling based solution approach for workforce scheduling. Afterwards, we describe constraint programming and mixed integer programming encodings of the model and discuss experimental results on a set of benchmark instances from the literature.

Finally, in Chapter 10 we give concluding remarks and discuss future work.

CHAPTER 2

The Paint Shop Scheduling Problem

In this chapter we introduce a novel and challenging scheduling problem originating from paint shops of the automotive supply industry which we call the paint shop scheduling problem (PSSP). First, we provide an informal problem description as well as some background on the problem. Afterwards, we give a formal problem specification followed by an overview of related literature. At the end of this chapter we prove that the decision variant of the PSSP is NP-complete and provide a collection of benchmark instances for the problem.

2.1 Problem Description & Background

A typical automotive supply company will serve not only one, but many car manufacturing companies and therefore produces a large variety of different products that need to be painted before delivery (e.g. bumpers and other exterior systems). Because of the short manufacturing cycles caused by the commonly used concept of just-in-time manufacturing in the automotive industry [SKCU77], it is of high importance to create production schedules that are able to fulfill all due dates requested by car manufacturers. Therefore, the main goal of the paint shop scheduling problem we introduce in this thesis is to determine a technically feasible production sequence that produces all ordered products within the given due dates.

Furthermore, two minimization criteria should be considered to reduce waste and save costs: First, the schedule should minimize the required color changes in the production sequences whenever possible. Second, the schedule should ensure the efficient utilization of the carrying devices used to transport the raw material components through the paint shop.

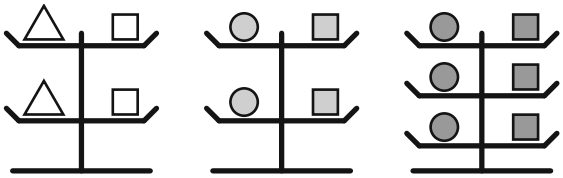

Figure 2.1: Schematic showing three carriers of two different carrier types.

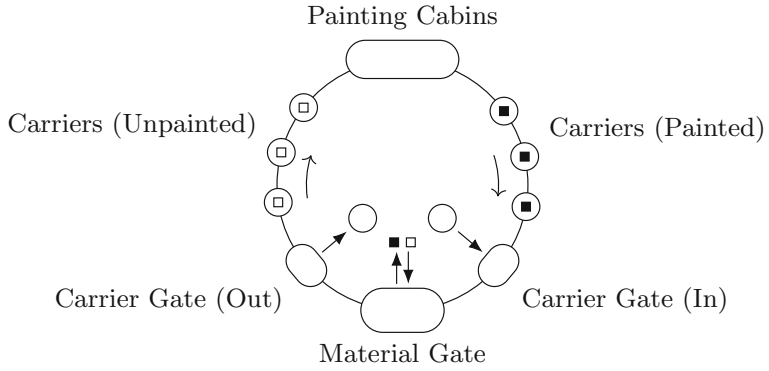

Figure 2.2: Schematic showing a paint shop layout that is commonly used in the automotive supply industry.

All items scheduled for painting need to be placed on customized carrier devices that move through the paint shop's painting cabins on a conveyor system. In each cabin, several painting robots apply paint on the carried automotive components. Carriers come in many types, each of which can transport certain configurations of demanded materials. Hence, different carrier device types need to be used in production. Although combinations of different raw material items may be transported by a single carrier, scheduling products that should be painted with different colors on a single carrying device is impossible. Figure 2.1 shows a schematic of two carrier types and three possible material configurations. The carrier shown on the left uses a material configuration that transports two triangular and two square components, the middle carrier transports two circular and two square components, and the carrier on the right side transports three circular and three square components. The figure illustrates how the same carrier type (the left and middle carriers are similar, whereas the right carrier is of a different type) can be used to transport different material type combinations through the paint shop as long as all pieces on a single carrier are painted with the same color (e.g., white, light-gray).

The paint shops of the automotive supply industry are designed to support an almost fully automated production process. Therefore, any scheduled carrying devices are automatically moved through the paint shop on a circular conveyor belt system. Carriers can be inserted into and removed from the conveyor belt at two carrier gates. One of the gates is used to insert carrying devices while the other one can be used to remove

carriers from the circular conveyor belt system. Once a carrier has been inserted, it moves through the cyclic paint shop system, wherein it repeatedly passes by the painting cabins, the carrier gates, and a material gate until the schedule selects it for ejection at the output gate. At the material gate, unpainted raw materials may be placed on any empty carrying device by paint shop employees. A loaded carrier then moves to the painting cabins, where the scheduled color is applied on all carried items. Whenever a loaded carrier arrives at the material gate after having completed a full round, another employee takes off the painted material pieces and may place new unpainted raw materials onto the carrier for painting in the succeeding round.

Figure 2.2 shows a schematic of a paint shop's layout and visualizes the movement of carriers through the paint shop. Carriers are displayed as circles in the figure, and some of them carry unpainted automotive components, which are visualized as small white squares. Carrying devices that have passed by the painting cabins are marked with black squares (which represent painted materials) in the graphic.

As the paint shop maintains a circular layout, the painting schedule is organized in rounds that are processed sequentially. Within each painting round, several carrier units are painted one after the other in a sequence that is predetermined by the schedule. However, the number of processed carriers per round and the exact sequence do not necessarily have to be equal for each round. A schedule therefore sets the painting sequences for multiple rounds and determines the raw material and color configurations for each scheduled carrying device. Note that in practice, the processing of a single paint shop round takes a fixed amount of time that does not depend on the number of carriers scheduled for the round (as long as the number of carriers per round stays within the specified boundaries). Therefore, the scheduling horizon of a problem instance is specified as the number of rounds to schedule, and due dates are expressed as due rounds in the problem input. The planner then needs to schedule carrier configurations into rounds so that all components are produced on time. In practice, schedules are usually planned weekly using a rolling horizon approach.

Note that this particular paint shop environment and production scheduling procedure is used at production sites all around the world from one of our industry partners. So far, we have not collaborated with other industry partners that could directly apply the solution methods that we propose in this chapter. However, the technical layout of the paint shop (including the painting robots, the circular conveyor belt, and the carrier devices) was implemented and designed in a similar way as it is used for different manufacturing processes from the automotive supply industry. Therefore, we believe that the paint shop scheduling problem that we model and solve in the following sections can be useful also for other problem variants from the industry that use similar automated paint shop environments.

To represent a candidate solution to the paint shop scheduling problem we derive a table where each column represents the scheduling sequence for a single round. Each table cell then assigns the carrier type, material configuration, and color that should be scheduled in the associated round sequence (from top to bottom). Figure 2.3 illustrates a

	R1	*R2*	*R3*
1	A1	A2	C1
2	A1	A2	C2
3	A2	C1	C3
4	B1	B2	B1
5	B2		B2

Figure 2.3: Example of a painting schedule with three rounds. Each column represents the scheduled carrier sequences scheduled within a single round.

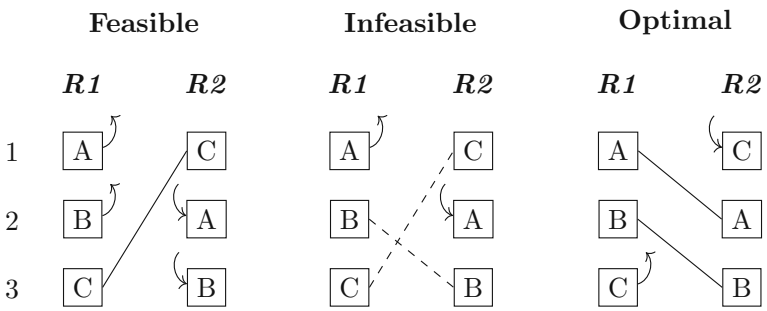

Figure 2.4: Three options to reuse carriers between two consecutive rounds.

toy problem solution for a scheduling horizon of three rounds. In the schedule shown in the figure, the carrier sequence (A1, A1, A2, B1, B2) scheduled for the first round (R1) includes five carriers. The first three carriers of this round sequence use a type A carrier with item configurations 1 and 2 and should be painted in a light gray color. Carriers 4 and 5 in R1 are of type B, and require a dark gray color and item configurations 1 and 2.

When all carrier configurations and colors that can be scheduled for production are considered, a large number of different schedules can be created. However, numerous constraints imposing due dates and technical requirements regarding feasible carrier sequences need to be fulfilled (Note that sequence constraints also apply to round overlapping sequences. In other words, sequence constraints must also hold between the last carriers of a round and the first carriers of the succeeding round.):

R.1 All material demands must be scheduled for production on time.

R.2 Carrier type availabilities must be considered in each round (e.g., if 10 physical instances of carrier type A are available, then this carrier can never be scheduled more than 10 times in a single round).

R.3 Minimum/maximum carrier capacities must be considered in each round: The number of carriers scheduled for each round needs to fall within the minimum and maximum

boundaries to ensure an efficient production cycle (empty carriers may be scheduled if necessary).

R.4 Forbidden carrier type sequences must not appear in the schedule: For logistical reasons at the material gate, certain carrier types are not allowed to directly follow another carrier type in the production sequence (e.g. a type A carrier must never directly follow a type B carrier in the production sequence).

R.5 Minimum and maximum carrier blocks must be considered: Similar to the consideration of the forbidden carrier type sequence constraints, logistical restrictions impose minimum and maximum block sequence constraints on scheduled carrier sequences.

In other words, whenever a carrier of type t is scheduled, the same carrier type needs to or may be used for the next consecutive carriers until the given minimum/maximum block length is reached (e.g., to illustrate the minimum block length, let the minimum block length for type t_1 be three and let the previously scheduled carrier type sequence be $\langle t_3, t_3, t_2, t_1 \rangle$; to satisfy the minimum block length, at least the next two carriers in the sequence need to be of type t_1).

R.6 Forbidden color sequences must be respected: For certain pairs of colors c_1, c_2, a number of carriers need to be painted in a different color before a switch from color c_1 to another color c_2 is legal in the production sequence.

For example, let this number for colors c_1 and c_2 be three. Then, the color sequences $\langle c_1, c_2 \rangle$ and $\langle c_1, y, c_2 \rangle$ would be illegal while the color sequence $\langle c_1, y, y, y, c_2 \rangle$ would be legal (assuming that $y \neq c_1$ and $y \neq c_2$).

A multi-objective function further includes two minimization criteria for the paint shop scheduling problem. The first optimization goal is to minimize color changes in the production sequence while the second optimization goal is concerned with the efficient utilization of carrying devices. In the following paragraphs, we further explain the second minimization goal.

As a paint shop schedule usually does not use the same carrier type sequence in each round, the carriers need to be removed from and inserted to the conveyor belt system between rounds. However, if carriers of the same type are scheduled in two consecutive rounds some of them may be reused as long as the sequence of the kept carriers is compatible with the scheduled carrier sequence in the succeeding round. As the insertion and removal of carrier units to and from the circular track might lead to delays and are generally not doable in parallel, the number of such operations should be kept as low as possible. Note that for any given two consecutive rounds, the minimum number of required carrier insertions and removals can be calculated by determining the minimum string edit distance (ED) [WF74] between two carrier type round sequences.

Two consecutive rounds of carrier type sequences may be viewed as two strings: the minimum number of required carrier changes corresponds to the ED with only the insertion and deletion operations considered. Figure 2.4 visualizes three alternative ways to reuse carriers between two consecutively scheduled round sequences, each of which

uses three carriers (R1: A, B, C and R2: C, A, B). The graphic shows how the ED determines the minimum number of required carrier changes.

The feasible option shown on the left side of the figure reuses only a single type C carrier and requires a total of two carrier insertions and two carrier removals. The infeasible option shown in the middle of the figure suggests retaining type B and C carriers between two consecutive rounds. However, this is technically not possible as C cannot be placed on an earlier position than B in the next round if it is reused (also due to the definition of edit distance no edge crossings are allowed). The option shown on the right side of the figure requires the fewest number of carrier insertions and removals, that is, one insertion and one removal; this requirement corresponds to the minimum string ED in this case.

Finally, note that in practice production scheduling is performed in a rolling horizon manner. Therefore, the last round that was processed at the production site before the current scheduling horizon (we from now on refer to this round as the history round) must also be considered regarding carrier changes, color changes and forbidden sequences. Figure 2.5 shows example schedules for three consecutive days to illustrate the concept of the history round.

At the top of Figure 2.5, we can see a schedule that is created for the first day of production, where exactly three rounds are scheduled that are labeled as $R1$, $R2$, and $R3$. Furthermore, there is also a history round labeled $History$ ($R0$) which displays the carrier configurations and colors that were processed before the scheduling horizon of the first day. Afterwards, in the middle of Figure 2.5, we see the schedule created on the second day which includes the rounds $R4$, $R5$, and $R6$ in its scheduling horizon. Note that in this example rounds $R1$–$R3$ were already processed on the first day, and therefore $R3$ is considered as the history round for the problem instance that occurs on the second day. Finally, on the third day another problem instance with a scheduling horizon of three rounds has to be solved, however, this time $R6$ is the history round.

2.2 Formal Specification

In this section we provide a formal specification of the paint shop scheduling problem.

2.2.1 Input parameters

The following parameters describe instances of the problem:

Set of carrier types: T

Set of colors: C

Set of materials: M

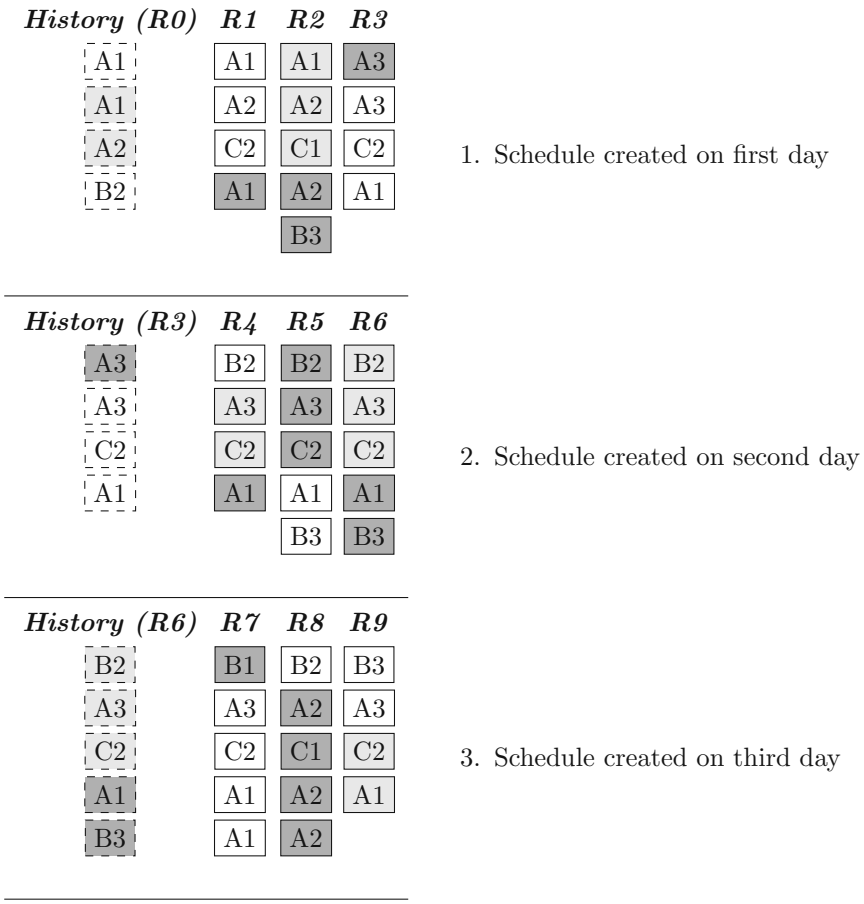

Figure 2.5: Example schedules created on three consecutive days, where on each day exactly three rounds are scheduled and processed.

Set of carrier configurations: K

A carrier configuration is always associated with a single carrier type and provides information about the materials that are placed on this carrier.

Number of rounds to schedule: n

Set of all rounds to schedule: $R = \{1, \ldots, n\}$

Maximum number of carrier slots per round: s

Set of carrier slots per round: $S = \{1, \ldots, s\}$

Minimum number of carriers that have to be scheduled in each round: q

Number of available carriers of type t in round r: $a_{r,t}, \forall r \in R, t \in T$

The number of available carriers is an input parameter because in practice some carriers will be scheduled for cleaning and maintenance from time to time (independently of the production schedule).

Set of demands: $D \subseteq \{(a, m, r, c) | a \in \mathbb{N}_{>0}, m \in M, r \in \mathbb{N}_{>0}, c \in C\}$

Each demand will ask for a number a of materials m in color c that have to be scheduled by round r. The set of demands may contain optional demands that are due by future rounds lying outside the scheduling horizon.

Number of pieces of material type m that can be placed on configuration k: $u_{k,m}, \forall k \in K, m \in M$

Carrier type of each carrier configuration: $v_k \in T, \forall k \in K$

Number of carriers scheduled in the round previous to the scheduling horizon (history round): p

Carrier type of the scheduled carrier at position i of the history round: $pt_i \in T, \forall i \in \{1, \ldots, p\}$

Used color at position i of the history round: $pc_i \in C, \forall i \in \{1, \ldots, p\}$

Set of forbidden carrier type sequences. All elements in F define forbidden carrier type sequences of length two that may not appear anywhere in the schedule: $F \subset \{(t_1, t_2) | t_1, t_2 \in T, t_1 \neq t_2\}$

As carriers are loaded manually by paint shop employees, mistakes are likely to happen when certain carrier types that use similar configurations appear directly after each other. In practice, it suffices to forbid carrier sequence of length two to prevent mistakes in the loading process.

Minimum block length for carrier type t: $b_t^{\min}, \forall t \in T$

Whenever a carrier of type t is scheduled, the same carrier type has to be used for the next consecutive carriers until the given minimum block length is reached. (For example let $b_{t_1}^{\min} = 3$ and the previously scheduled carrier type sequence be $\langle t_3, t_3, t_2, t_1 \rangle$, then to satisfy the minimum block length at least the next two carriers in the sequence have to be t_1).

Maximum block length for carrier type t: $b_t^{\max}, \forall t \in T$

The number of carriers that have to be painted in a different color before a switch from color c_1 to color c_2 becomes legal in the scheduled sequence: $o_{c_1,c_2} \in \mathbb{N}, \forall c_1, c_2 \in C$

For example let $o_{v,w} = 3$ for colors v and w. Then the color sequences $\langle v, w \rangle$ and $\langle v, y, w \rangle$ would be illegal while the color sequence $\langle v, y, y, y, w \rangle$ would be legal (assuming that $y \neq v$ and $y \neq w$).

Function that assigns color transition costs for all pairs of colors: $f_c : \{C \times C\} \to \mathbb{N}$

2.2.2 Decision variables

We define the following decision variables for the paint shop scheduling problem:

The carrier configuration scheduled in round i and position j, where λ_{j_1} is any configuration that is associated to the corresponding carrier type which is scheduled in the history round at position j_1:[1]

$$\begin{gathered} x_{i,j} \in K \cup \{\epsilon\}, \forall i \in \{0, \ldots, n\}, j \in S \\ x_{0,j_1} = \lambda_{j_1} \quad (\text{where } v_{(\lambda_{j_1})} = pt_{j_1}), \forall j_1 \in \{1, \ldots, p\} \\ x_{0,j_2} = \epsilon, \forall j \in \{p+1, \ldots, s\} \end{gathered}$$

If the value ϵ is assigned, the position is empty and no carrier will be scheduled at the position.

The color that is used in round i at position j:

$$\begin{gathered} c_{i,j} \in C \cup \{\epsilon\}, \forall i \in \{0, \ldots, n\}, j \in S \\ c_{0,j_1} = pc_{j_1}, \forall j_1 \in \{1, \ldots, p\} \\ c_{0,j_2} = \epsilon, \forall j \in \{p+1, \ldots, s\} \end{gathered}$$

If the value ϵ is assigned, the position is empty and will not be painted.

2.2.3 Helper Variables for Hard Constraints

To formulate the problem's hard constraints, we introduce the following helper variables and functions:

The number of carriers that are scheduled in round i ($i = 0$ refers to the history round): $p_i \in \{0, \ldots, s\}, \forall i \in \{0, \ldots, n\}$

p_0 refers to the number of carriers scheduled in the history round.

[1] The input parameters do not specify any information about the configurations used in the history round. For simplicity, we fix the corresponding decision variables for the history round to any configuration λ_{j_1} that is compatible with the used carrier type in the history round at position j_1.

The total number of carriers scheduled in the entire schedule, excluding the history round:

$pt \in \{0, \ldots, n \cdot s\}$

Sequence coordinate helper function:

$f_s(i, j) = p + \sum_{r \in \{2 \ldots i\}} p_{r-1} + j$

This helper function converts the two-indexed scheduling coordinates (round and position within rounds) into a one-indexed scheduling coordinate. For example let exactly 100 carriers be scheduled in round 1, then $f_2(2, 3)$ will be set to the value 103.

The carrier configuration that is scheduled at the one-indexed position coordinate i:

$$seqx_i \in K \cup \{\epsilon\}, \forall i \in \{1, \ldots, p + n \cdot s\}$$

The color that is scheduled at the one-indexed position coordinate i:

$$seqc_i \in C \cup \{\epsilon\}, \forall i \in \{1, \ldots, p + n \cdot s\}$$

2.2.4 Hard Constraints

1. Unplanned carrier positions must always be scheduled last in a round:

$$(x_{i,j} = \epsilon) \Rightarrow (x_{i,j+1} = \epsilon), \forall i \in R, j \in \{1, \ldots, s - 1\} \tag{2.1}$$

2. Any scheduled carrier position must also assign a color and any unscheduled position must not assign a color:

$$(x_{i,j} \neq \epsilon) \Leftrightarrow (c_{i,j} \neq \epsilon), \forall i \in R, j \in S \tag{2.2}$$

3. Force the correct number of scheduled carriers to the associated helper variables:

$$\begin{gathered} p_0 = p \\ p_r = |\{j \in \{1, \ldots, s\} | x_{r,j} \neq \epsilon\}|, \forall r \in R \\ pt = \sum_{r \in R} p_r \end{gathered} \tag{2.3}$$

4. Bind the values of the decision variables to the associated one indexed sequence helper variables:

$$
\begin{gathered}
seqx_j = x_{0,j} \wedge seqc_j = pc_j, \forall j \in \{1, \ldots, p\} \\
x_{i,j} \neq \epsilon \Rightarrow \\
(seqx_{(f_s(i,j))} = x_{i,j} \wedge seqc_{(f_s(i,j))} = c_{i,j}), \\
\forall i \in R, j \in S \\
(k > p + pt) \Leftrightarrow seqx_k = \epsilon, \\
\forall k \in \{p+1, \ldots, p + n \cdot s\}
\end{gathered}
\tag{2.4}
$$

5. All demands must be satisfied in time (overproduction is allowed):

$$
\begin{gathered}
\sum_{\{(d_a,d_m,d_r,d_c) \in D |\ d_m = m \wedge d_r <= r \wedge d_c = c\}} d_a \leq \\
\sum_{\{i \in \{1,\ldots,n\}, j \in \{1,\ldots,s\} | c_{i,j} = c\}} u_{(x_{i,j}),m} \\
\forall r \in R, m \in M, c \in C
\end{gathered}
\tag{2.5}
$$

6. Carrier availabilities must be respected in each round (X here refers to the set of all $x_{i,j}$ variables):

$$
|\{x_{i,j} \in X | i = r \wedge v_{(x_{i,j})} = t\}| \leq a_{r,t}, \forall r \in R, t \in T \tag{2.6}
$$

7. The minimum round capacity must be fulfilled in each round:

$$
p_r >= q, \forall r \in R \tag{2.7}
$$

8. Forbidden carrier type sequences must not appear in the schedule:

$$
\begin{gathered}
v_{(seqx_i)} \neq t_1 \vee v_{(seqx_{i+1})} \neq t_2, \\
\forall (t_1, t_2) \in F, i \in \{p, \ldots, (p + n \cdot s - 1)\}
\end{gathered}
\tag{2.8}
$$

9. Minimum carrier block length restrictions must be fulfilled:

$$
\begin{gathered}
(v_{(seqx_i)} \neq t \wedge v_{(seqx_{i+1})} = t) \Rightarrow \bigwedge_{j \in \{2,\ldots,b_t^{\min}\}} (v_{(seqx_{i+j})} = t), \\
\forall t \in T, i \in \{1, \ldots, (p + n \cdot s - b_t^{\min} - 1)\}
\end{gathered}
\tag{2.9}
$$

$$
\begin{gathered}
\neg(v_{(seqx_{p+n\cdot s-b_t^{\min}+1})} \neq t \wedge v_{(seqx_{p+n\cdot s-b_t^{\min}+2})} = t) \\
\forall t \in T
\end{gathered}
\tag{2.10}
$$

10. Maximum carrier block length restrictions must be fulfilled:

$$\bigvee_{j \in \{0,\ldots,b_t^{\max}\}} (v_{(seqx_{(i+j)})} \neq t), \quad \forall t \in T, i \in \{1,\ldots,(p + n \cdot s - b_t^{\max})\} \tag{2.11}$$

11. No forbidden color sequences should occur in the schedule:

$$(seqc_i = c_1) \Rightarrow \bigwedge_{j \in \{1,\ldots,o_{(c_1,c_2)}\}} (seqc_{(i+j)} \neq c_2), \quad \forall c_1, c_2 \in C, i \in \{1,\ldots,(p + n \cdot s - o_{(c_1,c_2)} + 1)\} \tag{2.12}$$

2.2.5 Helper Variables and Constraints for the Objective Function

To formulate the problem's minimization function, we introduce the following helper variables:

The amount of color change costs occurring in round r of the schedule:

$cc_r, \forall r \in R$

The number of required carrier type changes between round r and $r+1$:

$$sc_r, \forall r \in \{0,\ldots,n-1\}$$

The number of carriers that will not be changed after round r and reused in round $r+1$:

$sk_r, \forall r \in \{0,\ldots,n-1\}$

Edge helper variables:

$$e_{r,k,l} \in \{0,1\}, \forall r \in \{0,\ldots,n-1\}, k \in S, l \in S$$

Boolean edge variables are used to represent whether a carrier should be reused between two consecutive rounds. Previously, in Figure 2.4 an illustrated example of such edges between two rounds was presented and discussed in Section 2.1. The edge variables are set to true whenever a carrier from round r at position k is reused in round $r+1$ at position l.

The following hard constraints are used to assign values to the helper variables:

1. Sum up the color change costs per round in the associated helper variables. The value includes a potential color change cost that occurs between the last position of the previous round to the first position of the target round (We assume here that if the value ϵ is assigned to any parameter of f_c, the function will return 0):

$$cc_r = \sum_{j \in \{1,\dots,s-1\}} f_c(c_{(r,j)}, c_{(r,j+1)}) + f_c(c_{(r-1,(p_{r-1}))}, c_{(r,1)}), \forall r \in R \tag{2.13}$$

2. The number of necessary carrier changes between two given rounds are calculated with the helper variables sk_r that determine how many carriers can be kept after each round:

$$sc_r = p_r - sk_r + p_{r+1} - sk_r, \forall r \in \{0, \dots, n-1\} \tag{2.14}$$

3. The sk_r variables are assigned by summing up the number of associated edge variables that are set to 1. Note that each edge variable set to 1 will represent a carrier that is kept between two consecutive rounds:

$$sk_r = \sum_{k,l \in S} e_{r,k,l}, \forall r \in \{0, \dots, r-1\} \tag{2.15}$$

4. The following constraints enforce that edges between carriers of consecutive rounds (carriers connected by an edge will be reused) are only allowed if the carrier types at both positions are equal and not set to ϵ:

$$(e_{r,k,l} = 0) \Leftarrow (x_{r,k} = \epsilon \vee x_{r+1,l} = \epsilon), \quad \forall r \in \{0, \dots, n-1\}, k \in S, l \in S \tag{2.16}$$

$$(e_{r,k,l} = 1) \Rightarrow (v_{(x_{r,k})} = v_{(x_{r+1,l})}), \quad \forall r \in \{0, \dots, n-1\}, k \in S, l \in S \tag{2.17}$$

5. The following constraint forbids crossings between selected edges of two consecutive rounds. These crossings have to be forbidden to enforce the correct order of kept carriers.:

$$(e_{r,k,l} = 1) \Rightarrow \left(\bigwedge_{\substack{m \in \{1,\dots,k-1\}, \\ n \in \{l,\dots,s\}}} (e_{r,m,n} = 0) \wedge \bigwedge_{\substack{m \in \{k,\dots,s\}, \\ n \in \{1,\dots,l-1\}}} (e_{r,m,n} = 0) \right) \quad \forall r \in \{0, \dots, n-1\}, k \in S, l \in S \tag{2.18}$$

2.2.6 Objective function

The objective function aims to minimize the number of carrier changes (sc) and color change costs (cc). The sums are squared, since it is preferable to distribute the required changes over the scheduling horizon and to avoid peaks of many changes within a single round.

$$minimize \sum_{r \in \{0,\ldots,n-1\}} sc_r^2 + \sum_{r \in R} cc_r^2 \tag{2.19}$$

Note that the objective function is essentially a multi-objective function that combines the color change and carrier change objectives as a weighted sum of squared changes per round (using identical weights of value 1). This function was formulated with support from expert-practitioners to reasonably capture the cost factors in real-life automotive paint shop settings; thus, we do not consider additional alternative multi-objective functions in the current work.

We want to point out that color change costs and the number of carrier changes are in similar orders of magnitude for all practical problem instances used in our experimental evaluation. Therefore, we do not discuss the normalization of the two objectives.

2.3 Related Literature

Automated solution methods for production scheduling and sequencing problems in the automotive industry have been thoroughly studied. One of the earliest investigated problems from this area is the so-called car sequencing problem, which was first described in [PKW86]. The goal of the original formulation of this sequencing problem is to find an optimized production sequence for a given set of cars; the manufacturing process for each car may require different assembly operations depending on the installation options ordered (e.g., sun roof, air conditioning). As each of these options is installed at a different station, solutions to the car sequencing problem should ensure that the capacity of these stations is never exceeded during production. These constraints are usually expressed as ratio constraints that restrict the number of cars having a certain option appearing in subsequences of the solution.

In [Kis04] the car sequencing problem was identified as an NP-hard problem. Many heuristic and exact solution approaches have been investigated to solve this problem. One of the earliest exact approaches [DSVH88] successfully utilized a constraint logic programming approach to solve large practical problem instances. Other exact approaches for car sequencing have since been proposed, and they include MIP (e.g., [DK01]) and branch-and-bound algorithms (e.g., [DKM06]). Metaheuristic and hybrid approaches have also been investigated to tackle extremely large instances for variants of the car sequencing problem (e.g., [PG02, MCB19]) in a reasonable runtime. In [SCNA08], the authors provided an extensive survey of the solution approaches for the car sequencing problem and described a widely investigated problem extension used in the ROADEF'2005 challenge. The extended problem formulation used in this challenge additionally considers the painting process and includes the minimization of the costs caused by necessary color changes in the car sequence. The investigated solution methods for the ROADEF'2005 problem include exact approaches based on MIP (e.g., [PR08]) as well as metaheuristics such as ant colony optimization and local search (e.g., [GGP06, PR08]).

Another sequencing problem originating from automotive paint shops focuses solely on the minimization of costs induced by color changes [EHO04]. The main goal of this paint shop problem is to find an optimal coloring for a given sequence of jobs that minimizes the required color changes. An NP-hardness proof and an exact approach using dynamic programming (under bounded instance parameters) were investigated in [EHO04]. Linear programming and local search based approaches to tackling practically sized instances of the problem were studied in [MN12].

A sequential ordering problem in automotive paint shops was described in [SGV04]. Similar to other paint shop problems, this sequential ordering problem aims to find a production sequence that minimizes the necessary color changes. However, this variant considers the utilization of so-called selectivity banks, in which multiple cars are grouped together in banks, as is often the case in automotive paint shops. The authors of [SGV04] proposed a model as a sequential ordering problem and introduced an exact method based on a branch and bound algorithm. Further studies on the topic also investigated heuristic techniques to quickly produce efficient solutions (e.g., [SFSZ15, SH17]).

Similar to previous production scheduling problems from the automotive area, the paint shop scheduling problem we investigate is aimed at creating an optimized schedule for a paint shop that minimizes color changes in the production sequence. However, as the manufacturing process of this particular problem produces car components that are placed on carrier devices, novel carrier constraints need to be considered, and the number of carrier changes in the schedule should be minimized. In contrast to previous automotive paint shop problems, the proposed paint shop scheduling problem introduces due date constraints and therefore requires a prompt scheduling of the required components. These unique properties make the proposed paint shop scheduling problem considerably different from previous automotive sequencing problems.

Another scheduling problem from the automotive supply industry deals with component primer painting [SYK+20]. The problem was deemed to be NP-hard in [SYK+20], and the authors further proposed an exact solution approach that uses MIP and a metaheuristic approach that uses tabu search. Similar to the paint shop scheduling problem, the component primer painting problem deals with the placement of multiple automotive components on hanger devices. However, solution methods for this problem cannot be compared with the methods we investigate in this thesis, as they do not consider due dates, and they define different constraints and an objective function that includes capacity loss, mixing costs (when different item categories are placed on a hanger) and workload costs. Furthermore, the component primer painting problem does not consider the minimization of color changes or carrier changes and related constraints, all of which appear in the paint shop scheduling problem.

Table 2.1 presents an overview of the properties of the problems related to the paint shop scheduling problem investigated herein.

Columns 1–5 display from left to right the name of the related problem, considered constraints, considered solution objectives, example papers describing exact solution

Problem	Constraints	Objective Function	Exact Methods	Heuristics
Car Sequencing [PKW86, SCNA08]	Capacity Constraints, Color Batch Size	Color Changes, Capacity Violations	[DSVH88, DK01] [DKM06, PR08]	[PG02, PR08] [MCB19]
Paint Shop Problem [EHO04]	None	Color Changes	[EHO04, MN12]	[MN12]
Sequential Ordering in Paint Shops [SGV04]	Precedence Constraints	Color Changes	[SGV04]	[SFSZ15, SH17]
Component Primer Painting [SYK+20]	Hanger Eligibility, Hanger Capacity	Capacity Loss, Mixing Costs, Workload	[SYK+20]	[SYK+20]
Paint Shop Scheduling	Due Dates, Resource Capacity, Forbidden Sequences	Color Changes, Carrier Changes	–	–

Table 2.1: An overview on the literature on related problems to the paint shop scheduling problem.

methods, and example papers investigating heuristic solution methods. The final row of the table shows the properties of the paint shop scheduling problem investigated in this thesis.

2.4 Complexity Analysis

In this section, we show that the decision variant of the paint shop scheduling problem is NP-complete. This problem variant asks whether a feasible schedule with an objective value $\leq t$ can be found. We prove the following:

Theorem 1. *The decision variant of the paint shop scheduling problem is NP-complete.*

Proof. We provide a polynomial time reduction from the set cover problem [Kar72] to the paint shop scheduling problem.

Consider an arbitrary instance of the set cover problem consisting of the universe of elements $U = \{1, \ldots, w\}$, an integer k, and a set S that denotes the collection of z sets. The union of all sets in S is equal to the universe $U = \{1, \ldots, w\}$. The question is whether a set covering of size k or less is available or not.

We construct an instance of the paint shop scheduling problem as follows. We set the scheduling horizon to a single round ($R = \{1\}$) and set $C = \{1\}$ as only one color to be considered for scheduling. The set of materials M is set to match all items of the universe U, $M = \{1, \ldots, w\}$. The maximum number of allowed carrier devices per round is set to k while the minimum number of required carrier devices per round is set to $q = 1$. We only consider a single carrier type and therefore set $T = \{1\}$. The set of all demands D that need to be scheduled is given as $D = \{(1, i, 1, 1) | i \in \{1, \ldots, w\}\}$ (we schedule each material exactly once with a due date of round 1). We further create z carrier configurations, each of which corresponds to a single set in S: $K = \{1, \ldots, z\}$. All configurations belong to the same carrier type. Therefore, we set $v_k = 1, \forall k \in K$. The

materials contained within each configuration should be equal to all elements contained in the associated set. Hence, we set $\forall x \in K, m \in M$:

$$u_{x,m} = \begin{cases} 1, & \text{if } m \text{ is contained in the associated set of configuration } x \\ 0, & \text{otherwise} \end{cases}$$

Furthermore, we set the number of carriers in the history round to $p = 0$. Then, we set all color costs to 0 and disable all sequence-dependent hard constraints and the carrier availability constraint by setting $f_c(1,1) = 0, b_1^{\min} = 1, b_1^{\max} = k, o = 0, f = 0$ and $a_{1,1} = k$. Note that by setting color costs to 0 and not using sequence constraints, we do not interfere with the original specification but instead, we simply build legal instances to the problem by modifying the input parameters.

We now prove the following:

Theorem 2. *There exists a set cover of size k or less if and only if there exists a feasible paint shop schedule with total costs lower than or equal to k^2.*

Proof. As we set all color transition costs to 0 and the history round contains 0 carriers, the objective function of the paint shop scheduling problem becomes equal to the squared number of scheduled carriers in round 1 (any carrier needs to be inserted). As we have set the number of maximum carriers per round to k, any feasible paint shop schedule fulfilling the maximum round capacity hard constraint will have an objective value $\leq k^2$. Furthermore, because we have disabled all hard constraints, except the demand constraint, in the paint shop scheduling instance, any schedule that satisfies all demands becomes a feasible schedule.

Let S be a set cover using k' sets (where $k' \leq k$). Then, all elements of the universe U are contained in at least one of the selected sets. Through our reduction, the paint shop scheduling problem is constructed in such a way that each element of the universe has to be scheduled at least once in round 1. For each set $s \in S$, there exists a carrier configuration that carries each element in s exactly once. Therefore, there exists a set of k' carrier configurations that can be scheduled in any sequence to fulfill all demands.

At this point, we prove the opposite direction. Let P be a feasible paint shop schedule. Then, any material must be scheduled exactly once in round 1. Therefore, any repeatedly used configurations can be removed from P in such a way that each carrier configuration scheduled in P is used exactly once without violating the demand constraint. As we have set the maximum carriers per round to k in our reduction, we have k' (where $k' \leq k$) carriers in the schedule. Given such a schedule that uses k' carriers, we can construct a feasible set covering of size k' by using the sets corresponding to the configurations used in P.

We then show that the decision variant of the paint shop scheduling problem is in NP. Suppose that we have given a candidate solution P. We show below that we can verify in polynomial time whether P is a feasible solution to the problem.

The number of carriers in P cannot be larger than $|S| \cdot |R|$ (see input parameters in Section 2.2). We can simply check the carrier availability and round capacity constraints by counting the number of carriers in each round. Similarly, the sequence constraints (minimum/maximum block length of consecutive carrier types, forbidden carrier types and color sequences) can be checked by iterating over the scheduled sequence. Furthermore, we can check the demand constraint by verifying that Equation 3.5 holds. We have to perform not more than $|R| \cdot |M| \cdot |C|$ comparisons to check this equation. For each comparison, we need to consider at most d demands and $|S| \cdot |R|$ positions to calculate the sums.

To calculate the total color change costs of P, we iterate over the scheduled sequence, similar to our approach to the sequence-dependent hard constraints. Finally, we need to determine the maximum number of carriers that can be reused between any two consecutive rounds in the schedule to calculate the total carrier change costs. As mentioned previously, we can establish this number by solving the corresponding string ED problem for each pair of consecutive rounds (the polynomial time algorithms for calculating the minimum ED were described in [WF74]).

2.5 Benchmark Instances

We generated 24 instances for the paint shop scheduling problem based on actual planning scenarios from the automotive industry[2]. Instances 13–24 have been generated by processing scheduling scenarios as they have recently appeared at a real-life production site of our industrial partner. Those instances describe six different planning horizons of 7, 20, 50, 70, 100 and 200 rounds. For each horizon length we generated two instances: One instance which includes forbidden sequence constraints and one instance without forbidden sequence constraints. Early experiments with the instances showed that exact methods could not provide any solutions to these instances (the solvers ran out of memory on a machine with 48GB RAM), and we therefore decided to manually scale down Instances 13–24 by randomly selecting roughly 5% of the materials, colors, configurations and demands to create the smaller Instances 1–12.

Table 2.2 summarizes the size parameters for the 24 benchmark instances, where every row contains parameters for a single instance.

Columns 2 and 3 include information about the total number of rounds (Rounds) as well as the maximum capacity of carriers per round (Round Capacity), whereas Columns 4–6 display the total number of colors (Colors), carrier types (Carrier Types), and demands

[2]All instances are publicly available: `https://www.dbai.tuwien.ac.at/staff/winter/ps_instances.zip`

Instance	Rounds	Round Capacity	Colors	Carrier Types	Demands	Forbidden Seq.
I 1	7	19	6	2	2	no
I 2	7	19	7	2	0	yes
I 3	20	19	7	2	0	no
I 4	20	19	4	2	4	yes
I 5	50	19	4	3	22	no
I 6	50	19	6	2	0	yes
I 7	70	19	4	4	55	no
I 8	70	19	8	2	5	yes
I 9	100	19	6	2	39	no
I 10	100	19	6	2	35	yes
I 11	200	19	6	3	118	no
I 12	200	19	7	4	384	yes
I 13	7	480	20	46	108	no
I 14	7	480	20	46	108	yes
I 15	20	480	20	46	238	no
I 16	20	480	20	46	238	yes
I 17	50	480	20	46	1055	no
I 18	50	480	20	46	1055	yes
I 19	70	480	20	46	1743	no
I 20	70	480	20	46	1743	yes
I 21	100	480	20	46	2469	no
I 22	100	480	20	46	2469	yes
I 23	200	480	20	46	5907	no
I 24	200	480	20	46	6057	yes

Table 2.2: Overview of instance size parameters for the 24 publicly available benchmark instances for the PSSP.

(Demands) that are specified by the instances. Finally, Column 7 indicates whether forbidden carrier and color sequence hard constraints are imposed by the instance.

Note that instances 1–12 are considered to be small instances, as the round capacity as well as the number of colors, carrier types, and demands are much smaller than for instances 13–24. Instances 13–24 on the other hand represent real-life scheduling scenarios from a large-scale industrial paint shop, therefore they all use the same round capacity, colors and carrier types.

CHAPTER 3

A Constraint Programming Approach for the Paint Shop Scheduling Problem

In this chapter we propose an exact approach for the paint shop scheduling problem using constraint modeling. We first introduce a direct modeling approach, and afterwards propose an alternative modeling of the sequence constraints using deterministic finite automatons. Finally, we perform an in-depth evaluation of our models using several programmed search strategies together with state-of-the-art CP and MIP solvers.

3.1 Modeling the Paint Shop Scheduling Problem with CP

In this section, we briefly provide an overview on CP, prior to proposing a direct CP model for the paint shop scheduling problem.

Note that the direct CP formulation that we propose in this chapter models the problem similarly as it was done for the formal problem specification that was given earlier in Section 2.2. The main aim of the problem specification in the previous chapter was to mathematically specify the problem in a way that it is easy to understand for the reader. On the other hand, the constraint models given in this chapter are meant to serve as a high-level formulation of the problem that can be utilized by constraint solving technology. Therefore, some details about the input parameters and constraint specifications are formulated differently in the direct CP model than in Section 2.2 to allow an efficient encoding for CP solvers (especially regarding the formulation of sequence-dependent constraints and the carrier change objective).

In addition to the direct CP formulation we propose an alternative model for some of the problem's constraints in Section 3.2.

3.1.1 Preliminaries

CP is a paradigm for solving combinatorial search problems using a wide range of techniques from areas such as artificial intelligence, computer science, and operations research. CP has been successfully applied to solve problems from many domains, including scheduling, vehicle routing, and planning.

In CP, users declaratively state the constraints that restrict feasible solutions to a search problem. In the case of optimization problems, an associated objective function that determines the cost of a solution is declared.

Formally, a constraint optimization problem is defined as follows:

Given a set X of variables $(x_1, x_2, \ldots, x_n)$, a set D of domains for each variable $(D_1, D_2, \ldots, D_n)$, a set C of constraints $(c_i \subseteq (D_1 \times D_2 \times \cdots \times D_n), \forall c_i \in C)$, and an objective function $f : (D_1 \times D_2 \times \cdots \times D_n) \to \mathbb{R}$, a constraint optimization problem is the problem of finding an assignment $x_i = d_i \in D_i, \forall i \in \{1, 2, \ldots, n\}$ such that all constraints are satisfied and the objective function is optimized.

In solving constraint optimization problems with CP, standard methods utilize a combination of backtracking search and constraint propagation, in which users can specify customized problem-specific branching strategies. Further information about CP is available in [RvBW06].

High-Level Modeling In this thesis, we use a high-level CP modeling notation to propose CP models. Most parts of the models are directly solvable by CP. However, we implicitly make use of *constraint reification* to express conditional sums and logical implications. To illustrate how logical implications can be translated into low-level clauses, we consider a constraint of the form $(x_{1,1} = 0) \Rightarrow (x_{1,2} = 0)$. We can translate this constraint into the following clauses: $b_1 \Leftrightarrow (x_{1,1} = 0), b_2 \Leftrightarrow (x_{1,2} = 0), \neg b_1 \vee b_2$, where b_1, b_2 are Boolean variables.

We also implicitly make use of the *element constraint* to use variables as indices for array access in our models.

Further information on constraint reification and the element constraint is available in [RvBW06].

3.1.2 Input parameters

The following parameters describe instances of the problem:

Set of carrier configurations: K

Set of carrier types: T

Set of colors: C

Set of materials: M

Set of all rounds to schedule: R

Set of carrier positions per round (maximum number of positions per round): S

Minimum number of carriers that have to be scheduled in each round: $q \in \mathbb{N}_{>0}$

Number of available carriers of type t in round r: $a_{r,t} \in \{1, \ldots, |S|\}, \forall r \in R, t \in T$

Set of demands: D

Each demand will request a number a of materials m in color c that have to be scheduled by round r. The set of demands may contain optional demands that are due by future rounds (i.e. rounds lying outside the scheduling horizon).

Number of requested items per demand: $a_d \in \mathbb{N}_{>0}, \forall d \in D$

Material type of demand: $m_d \in M, \forall d \in D$

Due round of demand: $r_d \in \mathbb{N}_{>0}, \forall d \in D$

Color of demand: $c_d \in C, \forall d \in D$

Number of pieces of material type m that can be placed on configuration k: $u_{k,m} \in \mathbb{N}, \forall k \in K, m \in M$

Carrier type of each carrier configuration (v_0 will be set to 0):

$$v_x \in \{0, \ldots, |T|\}, \forall x \in \{0, \ldots, |K|\}$$

Number of carriers scheduled in the round previous to the scheduling horizon (history round): $p \in \mathbb{N}$

As already mentioned in the previous chapter, production in the paint shop will process one round of carriers after the other and therefore the conveyor belt system will never be empty. For this reason, whenever we need to create a new paint shop schedule we are given the carriers and colors used in the latest previous round of production as an input, so that the amount of carrier and color changes within the first round of the scheduling horizon can be determined.

Carrier type of the scheduled carrier at position i of the history round: $pt_i \in T, \forall i \in \{1, \ldots, p\}$

Used color at position i of the history round: $pc_i \in C, \forall i \in \{1, \ldots, p\}$

Forbidden carrier type sequences: F

First carrier type of forbidden sequence f: $t_f^1 \in T, \forall f \in F$

Second carrier type of forbidden sequence f: $t_f^2 \in T, \forall f \in F$

Minimum block size of consecutive carriers with type t: $b_t^{\min} \in \mathbb{N}_{>0}, \forall t \in T$

Whenever a carrier of type t is scheduled, the same carrier type needs to be used for the next consecutive carriers until the given minimum block length is reached. (e.g., let $b_{t_1}^{\min} = 3$ and let the previously scheduled carrier type sequence be $\langle t_3, t_3, t_2, t_1 \rangle$; to satisfy the minimum block length, at least the next two carriers in the sequence need to be of type t_1).

Maximum block size of consecutive carriers with type t: $b_t^{\max} \in \mathbb{N}_{>0}, \forall t \in T$

Set of forbidden color sequences: O

First color of forbidden color sequence o: $c_o^1 \in C, \forall o \in O$

Second color of forbidden color sequence o: $c_o^2 \in C, \forall o \in O$

The number of carriers that have to be painted in a different color before a switch from color c^1 to color c^2 becomes legal for sequence o: $j_o \in \mathbb{N}_{>0}, \forall o \in O$

For example let $j_o = 3$ for colors $c_o^1 = v$ and $c_o^2 = w$. Then the color sequences $\langle v, w \rangle$ and $\langle v, y, w \rangle$ would be illegal while the color sequence $\langle v, y, y, y, w \rangle$ would be legal (assuming that $y \neq v$ and $y \neq w$).

Color transition costs for all pairs of colors: $f_{c_1,c_2} \in \mathbb{N}, \forall c_1, c_2 \in C$

3.1.3 Decision Variables

Scheduled carrier configuration in round i and position j:

$$x_{i,j} \in \{0, \ldots, |K|\}, \forall i \in R, j \in S$$

If the value 0 is assigned, the position is empty and no carrier will be scheduled at the position.

Scheduled color configuration in round i and position j: $y_{i,j} \in \{0,\dots,|C|\}, \forall i \in \{0,\dots,|R|\}, j \in S$

If the value 0 is assigned, the position is empty and will not be painted.

3.1.4 Auxiliary Variables

Number of scheduled carriers per round: $p_i \in \{0,\dots,|S|\}, \forall i \in \{0,\dots,|R|\}$

Number of totally scheduled carriers: $ps \in \{0,\dots,|S| \cdot |R| + p\}$

Sequence variables that will convert a given round index i and position index j into a one dimensional position index: $seq_{i,j} \in \{0,\dots,|S| \cdot |R| + p\}, \forall i \in \{0,\dots,|R|\}, j \in S$

For example let exactly 100 carriers be scheduled in round 1 and the length of the history round p be 5, then $seq_{2,3}$ will be set to the value 108. $seq_{i,j}$ will be set to 0 if and only if no carrier is scheduled at position j in round i.

3.1.5 Hard Constraints

In the following we propose the set of hard constraints for our model. Note that all of them are essential and that we do not use any redundant constraints in our formulation.

(a) Bind the correct number of scheduled carriers to the associated helper variables:

$$\begin{aligned} p_0 &= p \\ p_i &= \sum_{\{j \in S | x_{r,j} \neq 0\}} 1 \quad \forall i \in R \\ ps &= \sum_{i \in \{0,\dots,|R|\}} p_i \end{aligned} \tag{3.1}$$

(b) Set correct values to sequence variables:

$$\begin{aligned} seq_{0,j} &= j \quad \forall j \in \{1,\dots,p\} \\ seq_{0,j} &= 0 \quad \forall j \in \{p+1,\dots,|S|\} \\ seq_{i,1} &= p + 1 + \sum_{z \in \{1,\dots,i-1\}} p_z \quad \forall i \in R \\ seq_{i,j} &= seq_{i,j-1} + 1 \quad \forall i \in R, j \in \{2,\dots,|S|\} \text{ where } x_{i,j} \neq 0 \\ seq_{i,j} &= 0 \quad \forall i \in R, j \in S \text{ where } x_{i,j} = 0 \end{aligned} \tag{3.2}$$

(c) Unplanned carrier positions should always be scheduled last in a round[1]:

$$(x_{i,j} = 0) \Rightarrow (x_{i,j+1} = 0) \quad \forall i \in R, j \in \{1, \ldots, |S| - 1\} \tag{3.3}$$

(d) Any scheduled carrier position must also assign a color and any unscheduled position must not assign a color:

$$(x_{i,j} \neq 0) \Leftrightarrow (c_{i,j} \neq 0) \quad \forall i \in R, j \in S \tag{3.4}$$

(e) All demands must be satisfied in time, where overproduction is allowed (this constraint models requirement **R.1** from Section 2.1):

$$\sum_{\{d \in D |\ m_d = m \wedge r_d <= r \wedge c_d = c\}} a_d \leq \sum_{\{x_{i,j} | i \in \{1,\ldots,n\} \wedge j \in \{1,\ldots,|S|\} \wedge y_{i,j} = c\}} u_{(x_{i,j}),m} \quad \forall r \in R, m \in M, c \in C \tag{3.5}$$

(f) Carrier availabilities must be respected in each round (models requirement **R.2**):

$$\sum_{\{j | j \in S \wedge v_{(x_{r,j})} = t\}} 1 \leq a_{r,t} \quad \forall r \in R, t \in T \tag{3.6}$$

(g) The minimum round capacity must be fulfilled in each round (models requirement **R.3**):

$$p_r >= q, \forall r \in R \tag{3.7}$$

(h) Forbidden carrier type sequences must not appear in the schedule (models requirement **R.4**):

$$\begin{gathered}(v_{(x_{i,j})} \neq t_f^1) \vee (v_{(x_{i,j+1})} \neq t_f^2) \quad \forall f \in F, i \in R, j \in \{1, \ldots, |S| - 1\} \text{ where } j < p_i \\ (v_{(x_{i,(p_i)})} \neq t_f^1) \vee (v_{(x_{i+1,1})} \neq t_f^2) \quad \forall f \in F, i \in \{1, \ldots, |R| - 1\} \\ (pt \neq t_f^1) \vee (v_{(x_{1,1})} \neq t_f^2) \quad \forall f \in F\end{gathered} \tag{3.8}$$

(i) Minimum carrier type block size restrictions must be fulfilled (models requirement **R.5**)[2]:

[1] This constraint breaks symmetric solutions that would be possible if unused carrier positions could appear anywhere in the variable arrays. However, it is still not a redundant constraint, as other parts of the model rely on this restriction.

[2] For simplicity, we omit an additional corner case that has to be regarded: The last carrier type and color that appears in the history round also needs to be checked regarding the sequence constraints. This can simply be modeled by adding additional constraints for the history round.

$$\bigwedge_{z\in\{j+2,\ldots,|S|\}} (seq_{i,z} = 0 \vee seq_{i,z} \geq seq_{i,j+1} + b_t^{\min} \vee v_{(x_{i,z})} = t) \wedge$$
$$\bigwedge_{y\in\{i+1,\ldots,|R|\},z\in S} (seq_{y,z} = 0 \vee seq_{y,z} \geq seq_{i,j+1} + b_t^{\min} \vee v_{(x_{y,z})} = t) \wedge$$
$$\left(\bigvee_{z\in\{j+1,\ldots,|S|\}} (seq_{i,z} = seq_{i,j} + b_t^{\min} \wedge v_{(x_{i,z})} = t) \vee \right. \tag{3.9}$$
$$\left. \bigvee_{y\in\{i+1,\ldots,|R|\},z\in S} (seq_{y,z} = seq_{i,j} + b_t^{\min} \wedge v_{(x_{y,z})} = t) \right)$$
$$\forall t \in T, i \in R, j \in \{1,\ldots,|S|-1\} \text{ where } j < p_i \wedge v_{(x_{i,j})} \neq t \wedge v_{(x_{i,j+1})} = t$$

$$\bigwedge_{z\in\{2,\ldots,|S|\}} (seq_{i+1,z} = 0 \vee seq_{i+1,z} \geq seq_{i+1,1} + b_t^{\min} \vee v_{(x_{i+1,z})} = t) \wedge$$
$$\bigwedge_{y\in\{i+2,\ldots,|R|\},z\in S} (seq_{y,z} = 0 \vee seq_{y,z} \geq seq_{i+1,1} + b_t^{\min} \vee v_{(x_{y,z})} = t)$$
$$\left(\bigvee_{z\in\{1,\ldots,|S|\}} (seq_{i+1,z} = seq_{i+1,1} + b_t^{\min} - 1 \wedge v_{(x_{i+1,z})} = t) \vee \right. \tag{3.10}$$
$$\left. \bigvee_{y\in\{i+2,\ldots,|R|\},z\in S} (seq_{y,z} = seq_{i+1,1} + b_t^{\min} - 1 \wedge v_{(x_{y,z})} = t) \right)$$
$$\forall t \in T, i \in \{1,\ldots,|R|-1\} \text{ where } v_{(x_{i,p_i})} \neq t \wedge v_{(x_{i+1,1})} = t$$

Equation 3.9 models the minimum carrier type block size requirement for all positions in the schedule that start a new carrier type block, except if this happens at the last carrier position of a round (Equation 3.10 models the same requirement similarly for carrier type blocks that start at the last position of a round).

The first big conjunction over $z \in \{j+2,\ldots,|S|\}$ ensures that succeeding positions within the same round are valid if they are unused (i.e. set to 0), if they lie outside the minimum block size, or if they are matching the carrier type of the block. The second big conjunction over $(y \in \{i+1,\ldots,|R|\}, z \in S)$ ensures the same requirements for positions in succeeding rounds.

Finally, the disjunctions enclosed in parentheses ensure that there must exist a position in the schedule that fulfills the minimum block size with the correct carrier type assignment to prevent the assignment of blocks that are too short at the end of the schedule.

(j) Maximum carrier type block size restrictions must be fulfilled (models requirement **R.5**):[2]

$$\bigvee_{z\in\{j+1,\ldots,|S|\}} (seq_{i,z} > seq_{i,j} \wedge seq_{i,z} \leq seq_{i,j} + b_t^{\max} \wedge v_{(x_{i,z})} \neq t) \vee$$
$$\bigvee_{y\in\{i+1,\ldots,|R|\},z\in S} (seq_{y,z} > seq_{i,j} \wedge seq_{y,z} \leq seq_{i,j} + b_t^{\max} \wedge v_{(x_{y,z})} \neq t) \quad (3.11)$$
$$\forall t \in T, i \in R, j \in S, \text{ where } j \leq p_i \wedge v_{(x_{i,j})} = t \wedge seq_{i,j} =\leq ps - b_t^{\max}$$

(k) No forbidden color sequences should occur in the schedule (models requirement **R.6**):[2]

$$\bigwedge_{z\in\{j+1,\ldots,|S|\}} (seq_{i,z} = 0 \vee seq_{i,z} > seq_{i,j} + j_o \vee y_{i,z} \neq c_o^2) \wedge$$
$$\bigwedge_{x\in\{i+1,\ldots,|R|\},z\in S} (seq_{x,z} = 0 \vee seq_{x,z} > seq_{i,j} + j_o \vee y_{x,z} \neq c_o^2) \quad (3.12)$$
$$\forall o \in O, i \in R, j \in S \text{ where } j \leq p_i \wedge y_{i,j} = c_o^1$$

3.1.6 Auxiliary Variables for the Objective Function

The amount of color change costs occurring in round r of the schedule:

$$cc_r \in \mathbb{N}, \forall r \in R$$

The number of required carrier type changes between round r and $r+1$:

$$sc_r \in \{0, \ldots, |S| \cdot 2\}, \forall r \in \{0, \ldots, |R| - 1\}$$

The number of carriers that will be reused between round r and round $r+1$:

$$sk_r \in \{0, \ldots, |S|\}, \forall r \in \{0, \ldots, |R| - 1\}$$

Information on the position of the kept carrier sequence in the next/previous round:

$$kept_{i,j}^1 \in \{0, \ldots, |S|\}, \forall i \in \{0, \ldots, |R| - 1\}, j \in S$$
$$kept_{i,j}^2 \in \{0, \ldots, |S|\}, \forall i \in R, j \in S$$

3.1.7 Hard Constraints for Objective Function

Calculate color changes per round: [3]

$$cc_i = \sum_{j\in\{1,\ldots,|S|-1\}} f_{(y_{i,j}),(y_{i,j+1})} + f_{(y_{i-1,p_i-1}),(y_{i,1})} \quad \forall i \in R \quad (3.13)$$

[3] We assume that color costs from and to 0 will always be 0

All kept carrier type sequences between consecutive rounds have to be legal: [4]

In Figure 2.4 in Section 2.1, we show which carrier types may be reused between two consecutive rounds by drawing edges that connect the associated positions. We initially experimented with a modeling approach that introduces variables for all possible edges and tries to maximize the number of selected edges without causing any edge crossings to capture the ED between two consecutive rounds. However, using this model in practical instances is not efficient. Therefore, we propose a modeling approach that introduces variables to store the positions of all reused carriers in equations 3.14, 3.15, and 3.16. For example, if the second reused carrier that is scheduled on position four in the current round sequence should be reused in the next round sequence at position three, the associated variables store the values three and four ($kept^1_{i,2} = 3, kept^2_{i,2} = 4$, where i could be any round index). A value of zero is assigned to a $kept^y_{i,x}$ variable when less than x carriers are kept between the corresponding round (where $y \in \{1, 2\}$). Table 3.1 shows example variable assignments that correspond to the edge examples that are showcased in Figure 2.4.

$$\begin{aligned} kept^1_{i,j} > kept^1_{i,j-1} \quad \forall i \in \{0, \dots, |R|-1\}, i \in \{2, \dots, |S|\} \text{ where } kept^1_{i,j} \neq 0 \\ kept^2_{i,j} > kept^2_{i,j-1} \quad \forall i \in R, i \in \{2, \dots, |S|\} \text{ where } kept^2_{i,j} \neq 0 \end{aligned} \tag{3.14}$$

$$\begin{aligned} kept^2_{1,j} \leq p \quad \forall j \in S \text{ where } kept^2_{1,j} \neq 0 \\ kept^1_{i,j} \leq p_{i+1} \quad \forall i \in \{0, \dots, |R|-1\}, j \in S \text{ where } kept^1_{1,j} \neq 0 \\ kept^2_{i,j} \leq p_{i-1} \quad \forall i \in \{2, \dots, |R|\}, j \in S \text{ where } kept^2_{1,j} \neq 0 \\ kept^1_{0,j} = 0 \wedge kept^2_{1,j} = 0 \quad \forall j \in \{p+1, \dots, |S|\} \\ kept^1_{i,j} = 0 \wedge kept^2_{i+1,j} = 0 \quad \forall i \in \{1, \dots, |R|-1\}, j \in S \text{ where } j > p_i \\ kept^1_{i,j} > 0 \Leftrightarrow kept^2_{i+1,j} > 0 \quad \forall i \in \{0, \dots, |R|-1\}, j \in S \end{aligned} \tag{3.15}$$

$$\begin{aligned} pt_{(kept^2_{1,j})} = v_{(x_{1,(kept^1_{0,j})})} \quad \forall j \in S \text{ where } kept^2_{1,j} \neq 0 \\ v_{(x_{i,(kept^2_{i+1,j})})} = v_{(x_{i+1,(kept^1_{i,j})})} \quad \forall i \in \{1, \dots, |R|-1\}, j \in S \text{ where } kept^1_{i,j} \neq 0 \end{aligned} \tag{3.16}$$

Calculate the number of reused carriers after each round:

$$sk_i = \sum_{\{j | j \in S \wedge kept^1_{i,j} \neq 0\}} 1 \quad \forall i \in \{0, \dots, |R|-1\} \tag{3.17}$$

[4] For simplicity, we omit a special condition that handles the corner case of an empty history round. In this case one can simply add a constraint that forces all carriers of round 1 to be inserted if $p = 0$.

Feasible		
x	$kept^1_{1,x}$	$kept^2_{2,x}$
1	1	3
2	0	0
3	0	0

Infeasible		
x	$kept^1_{1,x}$	$kept^2_{2,x}$
1	3	3
2	1	2
3	0	0

Optimal		
x	$kept^1_{1,x}$	$kept^2_{2,x}$
1	2	1
2	3	2
3	0	0

Table 3.1: Table shows the $kept^1_{i,j}, kept^2_{i,j}$ variable values that correspond to the three options to reuse carriers between two consecutive rounds as shown in Figure 2.4.

Count the total number of required carrier changes between two rounds:

$$sc_i = p_i - sk_i + p_{i+1} - sk_i \quad \forall i \in \{0, \ldots, |R| - 1\} \tag{3.18}$$

3.1.8 Objective Function

The objective function of the paint shop scheduling problem aims to minimize the number of carrier changes (sc) and color change costs (cc) per round.

The sums are squared because the required changes should be distributed over the scheduling horizon and peaks of many changes within single rounds should be avoided.

$$\min \sum_{i \in R} cc_i^2 + \sum_{i \in \{0,\ldots,|R|-1\}} sc_i^2 \tag{3.19}$$

Note that Equation 3.19 models the objective in the same way as it was done for the mathematical specification in Section 2.2.6.

3.2 Modeling the Problem with DFAs

In this section, we propose a different way to model the sequence constraints (requirements **R.4**, **R.5**, and **R.6** from Section 2.1) by using DFAs. For this variant of the model, we replace equations 3.8, 3.9, 3.10, 3.11, and 3.12 with automaton formulations. All automatons process either the total sequence of scheduled carrier types or the total sequence of scheduled colors. We can provide the total carrier type or color sequence in our model by simply concatenating the values of all two indexed decision variables ($x_{i,j}$ or $y_{i,j}, \forall i \in R, j \in S$) into a one-dimensional list. The automatons can then be used to check whether the total color or carrier type sequence can be accepted. Automaton-based models can be directly solved by CP using the *regular constraint* [Pes04].

3.2.1 Forbidden carrier type sequences

For each forbidden carrier type sequence $f \in F$, we model an automaton that processes the total sequence of scheduled carrier types (this model replaces Equation 3.8 from Section 3.1). One state accepts all carrier types, whereas the second state does not accept

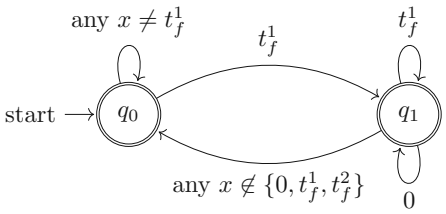

Figure 3.1: Automaton generated for each carrier type sequence to check the forbidden carrier type sequence constraint.

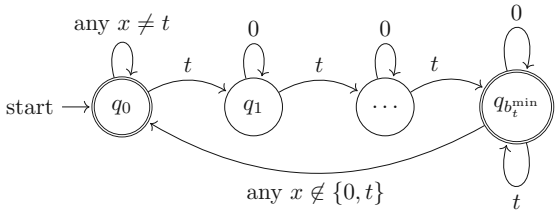

Figure 3.2: Automaton constructed for each carrier type $t \in T$ to check the minimum carrier block type size constraint.

any carrier type t_f^2 that immediately follows a carrier of type t_f^1. Figure 3.1 shows how automatons can be constructed to check that no forbidden carrier type sequence occurs in the schedule.

The first state q_0 in Figure 3.1 accepts any carrier type. State q_1 is entered whenever the first carrier type of the forbidden sequence (t_f^1) is encountered and does not accept the second type (t_f^2) before any other type is encountered. Both states are legal final states.

3.2.2 Minimum carrier type block sizes

For each carrier type $t \in T$, we model an automaton that processes the total sequence of scheduled carrier types (this model replaces Equations 3.9 and 3.10 from Section 3.1). Figure 3.2 shows how automatons can be constructed to check the minimum carrier type block size constraint.

The first state q_0 of Figure 3.2 accepts any carrier type that is different from t. States q_1–$q_{b_t^{\min}}$ are used to count the consecutive assignments of carrier type t. States q_0 and $q_{b_t^{\min}}$ are the only legal final states.

3.2.3 Maximum carrier type block sizes

For each carrier type $t \in T$, we model an automaton that processes the total sequence of scheduled carrier types (this model replaces Equation 3.11 from Section 3.1).

Figure 3.3 shows how automatons can be constructed to check the maximum carrier type block size constraint.

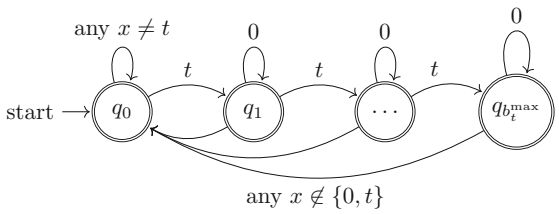

Figure 3.3: Automaton generated for each carrier type to check the maximum carrier block type size constraint.

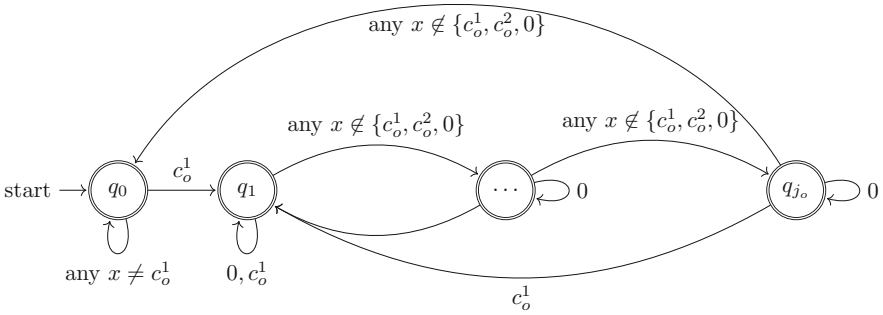

Figure 3.4: Automaton generated for each forbidden color sequence to check the forbidden color sequence constraint.

The first state q_0 in Figure 3.3 accepts any carrier type that is different from t. States q_1-$q_{b_t^{\max}}$ are used to count the consecutive assignments of carrier type t. All states are legal final states.

3.2.4 Forbidden color sequences

For each forbidden color sequence $o \in O$, we model an automaton that processes the total sequence of scheduled colors (this model replaces Equation 3.12 from Section 3.1).

Figure 3.4 shows how automatons can be constructed to check the forbidden color sequence constraint.

The first state q_0 in Figure 3.4 accepts any color assignment that is different from c_o^1. States q_1– q_{j_o} are used to assert that an assignment of color c_o^2 may only occur if color c_o^1 has not been encountered within the previous j positions. All states are legal final states.

3.3 Empirical Evaluation

In this section, we provide a detailed description of the experimental evaluation of the proposed CP models. First, we briefly describe the experimental setup and computational environment (Section 3.3.1). Second, we introduce the search strategies used to program

the search for the evaluated CP solvers (Section 3.3.2). Finally, we provide a summary of the experimental results and discuss the evaluation of the proposed models (Section 3.3.3).

3.3.1 Experimental Environment

To evaluate our models, we implemented the direct model proposed in Section 3.1 and the model using DFAs proposed in Section 3.2 by using the MiniZinc [NSB+07] modeling language, which provides interfaces to state-of-the-art CP and MIP solvers. All experiments were conducted on a computing cluster with 10 identical nodes, each having 24 cores, an Intel(R) Xeon(R) CPU E5–2650 v4 @ 2.20 GHz and 252 GB RAM.

We evaluated the proposed models by using the set of benchmark instances that we introduced in the previous chapter. This collection of instances includes 24 instances based on real-life planning scenarios. Instances 1–12 model problems for a small paint shop with a maximum capacity of 19 carriers per round. Instances 13–24 describe problems for large-scale paint shops that allow up to 480 carriers per round. The set of small instances and the set of large instances describe six different planning horizons of 7, 20, 50, 70, 100, and 200 rounds (two instances for each horizon: one that does not include forbidden carrier/color sequence constraints and another one that includes forbidden sequence constraints).

Table 3.2 presents an overview of the size parameters for all instances (For additional size parameters see Table 2.2). The columns show from left to right the instance ID, the number of rounds, the length of the planning horizon in days (in the industry, about five rounds are usually processed within 24-hour shifts), the maximum carrier capacity per round, whether forbidden sequence constraints are included, and the number of generated variables and constraints (# vars and # cs, respectively). To determine the number of generated variables and constraints, we analyzed the output of the MiniZinc compiler using the direct model (direct) and the model using DFAs (regular). A – indicates that the compiler ran out of memory on our benchmark machine or could not finish execution within 6 hours.

The size parameters displayed in Table 3.2 show that the model using DFAs generally leads to a lower number of variables and constraints for instances 1–10 in comparison with the direct model; hence, the model using DFAs has high efficiency. Furthermore, we can see that the MiniZinc compiler was not able to encode the large instances 11–24 on our machine within 6 hours of runtime, thereby indicating the tremendous size of the search space of large practical problem instances.

We conducted experiments with two state-of-the-art CP solvers: Chuffed [Chu11], which uses lazy clause learning; and Gecode [Gec19], which is a non-learning solver. In Section 3.3.2, we provide details about the 13 programmed search strategies that we evaluated for the CP solvers. Both solvers were run on each of the instances using both proposed models and search strategies within a runtime limit of 6 hours. Therefore, a total of 312 experiments were conducted for each CP solver.

	#rounds	days	capacity	forbidden seq	direct #vars	direct #cs	regular #vars	regular #cs
I 1	7	1.4	19	no	283845	289639	9237	12004
I 2	7	1.4	19	yes	399066	406964	9369	11499
I 3	20	4	19	no	3102108	3121429	25702	31801
I 4	20	4	19	yes	2891799	2912206	29312	37257
I 5	50	10	19	no	18374644	18435525	94066	124711
I 6	50	10	19	yes	19249180	19302056	66212	81411
I 7	70	14	19	no	42968894	43101797	210417	289019
I 8	70	14	19	yes	34183055	34258489	102820	129488
I 9	100	20	19	no	48323257	48439788	204091	282930
I 10	100	20	19	yes	84017220	84172576	232421	312301
I 11	200	40	19	no	-	-	-	-
I 12	200	40	19	yes	-	-	-	-
I 13	7	1.4	480	no	-	-	-	-
I 14	7	1.4	480	yes	-	-	-	-
I 15	20	4	480	no	-	-	-	-
I 16	20	4	480	yes	-	-	-	-
I 17	50	10	480	no	-	-	-	-
I 18	50	10	480	yes	-	-	-	-
I 19	70	14	480	no	-	-	-	-
I 20	70	14	480	yes	-	-	-	-
I 21	100	20	480	no	-	-	-	-
I 22	100	20	480	yes	-	-	-	-
I 23	200	40	480	no	-	-	-	-
I 24	200	40	480	yes	-	-	-	-

Table 3.2: An overview of the instance size parameters for the 24 instances that have been used for empirical evaluation.

Using the MiniZinc compiler we were able to automatically translate the proposed CP models for paint shop scheduling into MIP formulations, as MiniZinc provides its own linearization library. Therefore, we could also conduct experiments with the two state-of-the-art MIP solvers Gurobi [GO20] and CPLEX [Cor19] under the same runtime restrictions we used for evaluating the CP solvers.

3.3.2 Programmed Search Strategies

We evaluated the performance of the CP solvers Chuffed and Gecode by using several programmed search strategies, which are based on variable- and value selection heuristics. Such heuristics determine the order of the explored variable and value assignments for a CP solver and can play a critical role in reducing the search space that needs to be enumerated by the solver. For our experiments, we implemented the search strategies directly in the MiniZinc modeling language using search annotations.

Variable Selection

We defined the variable selection strategies for the decision variables that capture the scheduled carrier configurations $x_{i,j}, \forall i \in R, j \in S$ and the scheduled colors $y_{i,j}, \forall i \in R, j \in S$. Additionally, we investigated variable selection heuristics that set an order on the assignment of auxiliary variables that capture the number of required carrier changes between two rounds $sc_i, \forall i \in \{0, \ldots, |R|-1\}$. We also experimented with search strategies that focus on these sc variables as the calculation of the required carrier changes is one of the more complex parts of the models. Any variables that we do not explicitly mention

in the search strategies are selected on the basis of the CP solvers' default strategy after all the mentioned variables have been assigned.

- **default:** No variable and value selection is specified, and the solver uses its default strategy.
- **custom1:** All carrier configuration variables are selected first, followed by all color variables (i.e., $x_{1,1}, x_{1,2}, \ldots, x_{i,j}, y_{1,1}, y_{1,2}, \ldots, y_{i,j}$).
- **custom2:** The carrier change auxiliary variables are selected before the carrier configurations and colors are assigned (i.e., $sc_0, sc_1, \ldots, sc_{|R|-1}, x_{1,1}, x_{1,2}, \ldots, x_{i,j}, y_{1,1}, y_{1,2}, \ldots, y_{i,j}$).
- **custom3:** For each position in the schedule the associated carrier configuration variable is selected first, followed by the associated color variable (i.e., $x_{1,1}, y_{1,1}, x_{1,2}, y_{1,2}, \ldots, x_{i,j}, y_{i,j}$).
- **custom4:** The carrier change variables are selected first; then, the process continues with **custom3** (i.e., $sc_0, sc_1, \ldots, sc_{|R|-1}, x_{1,1}, y_{1,1}, x_{1,2}, y_{1,2}, \ldots, x_{i,j}, y_{i,j}$).
- **smallest:** The variables with the smallest possible domain value are chosen first (ties are broken on the basis of the order of **custom1**).
- **first fail:** The variables with the smallest domains are chosen first (ties are broken on the basis of the order of **custom1**).

Value Selection

We experimented with two different value selection heuristics:

- **min**: The smallest value is first assigned from a variable domain.
- **split**: The variable domain is bisected to first exclude the upper half of the domain.

Using the seven different variable selection strategies together with the two value selection heuristics we conducted experiments with a total of 13 search strategy configurations for each instance (the **default** variable selection uses the solver's default value selection).

3.3.3 Computational Results

In the following, we present the computational results for the paint shop scheduling problem that we obtained using the evaluated CP and MIP solvers. First, we provide an overview of the detailed CP results produced with Chuffed and Gecode. Second, we present the results produced by the MIP solvers Gurobi and CPLEX. Finally, we present an overall comparison of the best results.

search	t	#best	#fast	#opt	#prf	#sol	avg nodes	avg rt	std nodes	std rt
custom1 min	6h	2	0	2	2	**4**	287513.5	79.96	302407.65	73.54
custom1 split	6h	2	0	2	2	**4**	258341	68.06	260843.21	56.77
custom2 min	6h	3	0	3	**3**	**4**	972613.33	1547.57	1022208.46	2544.58
custom2 split	6h	3	0	3	**3**	**4**	1796283.33	2657.52	1146115.93	4176.63
custom3 min	6h	2	0	2	2	**4**	390482	78.61	237821.22	51.71
custom3 split	6h	2	0	2	2	**4**	520798	89.18	35434.54	20.68
custom4 min	6h	3	0	3	**3**	**4**	2033199.67	2092.07	1686560.49	3245.35
custom4 split	6h	3	0	3	**3**	**4**	2171945.33	4536.86	1933987.52	7501.61
default	6h	3	0	3	**3**	3	3221216.33	4856.34	4419678.13	6773.03
ff min	6h	2	0	2	2	**4**	2108669	470	608751.06	63
ff split	6h	2	0	2	2	**4**	2204912	481.15	472642.9	61.87
smallest min	6h	**4**	1	**4**	**3**	**4**	97822.67	45.25	157924.49	33.66
smallest split	6h	**4**	**2**	**4**	**3**	**4**	110939	43.81	182193.95	33.21

Table 3.3: Table summarizing the experimental results achieved with Chuffed and the direct model.

CP Results

We conducted experiments for instances 1–10 by using the introduced search strategies with Chuffed and Gecode (for Chuffed, we activated the free search parameter which allows the solver to alternate between the given search strategy and its default activity-based search heuristic on each restart). However, Gecode was not able to produce any feasible solution within the runtime limit. Thus, we only discuss the results produced with Chuffed in this section.

Tables 3.3 and 3.4 display an overview of results achieved with Chuffed using the direct model and the DFA model, respectively. Both tables present the summarized results for each of the evaluated search strategies and time limit configurations in a single row. The first seven columns display from left to right the used search strategy, the time limit, the number of best results achieved within its group ("group" refers to runs using the same model and runtime), the number of the fastest optimality proofs in the group (i.e. the number of instances where optimality could be determined the fastest over all experimental runs), the number of optimal solutions found, the number of proven optimal solutions, and the number of instances where a feasible solution could be achieved. Columns 8–11 present from left to right the average number of visited nodes in the search tree (only for instances that could be solved to optimality), the average runtime needed for the optimality proofs (considering only runs that could determine optimality within the runtime), the standard deviation of visited nodes for proofs, and the standard deviation of the optimality proof time (by optimality proof time we mean the total solving time until optimality was determined).

The results in both tables show that different search strategies do not exert a large effect on the number of solved instances. Nevertheless, the default search strategy seems to be slightly weaker than the other search strategies, and the smallest variable selection strategy leads the two models to the largest number of best solutions and optimality proofs.

Table 3.5 displays the overall best costs achieved for instances 1–10 by using Chuffed

search	t	#best	#fast	#opt	#prf	#sol	avg nodes	avg rt	std nodes	std rt
custom1 min	6h	9	1	**9**	**9**	**10**	3133111	1286.46	7235326.43	2274.45
custom1 split	6h	**10**	2	**9**	**9**	**10**	3481850.67	1109.3	6981431.78	1896.39
custom2 min	6h	9	0	8	8	**10**	1013865.75	709.87	1635642.59	1482.07
custom2 split	6h	7	0	7	7	**10**	9037592.57	1055.71	20273523.56	2518.71
custom3 min	6h	9	0	8	8	**10**	1013865.75	709.87	1635642.59	1482.07
custom3 split	6h	9	0	8	8	**10**	1246310.63	855.64	1966662.15	1929.19
custom4 min	6h	8	0	7	7	**10**	13717699.29	1177.45	29593979.97	2751.27
custom4 split	6h	8	0	7	7	9	21727481.14	1497.9	52229928.06	3715.43
default	6h	6	0	6	6	7	887576.83	226.43	1221199.62	458.27
ff min	6h	9	1	8	8	**10**	1025480.88	604.63	1841286.56	1314.26
ff split	6h	9	0	8	8	**10**	1608216.25	939.78	2789085.47	2002.13
smallest min	6h	**10**	2	**9**	**9**	**10**	3443121.67	1796.47	7117553.6	3624.84
smallest split	6h	9	**3**	**9**	**9**	**10**	2324348.44	1614.02	4194724.49	3787.35

Table 3.4: Table summarizing the experimental results achieved with Chuffed and the DFA model.

	chuffed-6h	proof time	chuffed-reg-6h	proof time
Instance 1	**775***	80.96s	**775***	3.10s
Instance 2	**842***	29.33s	**842***	0.70s
Instance 3	**961***	256.95s	**961***	1.84s
Instance 4	**918**	-	**918***	26.27s
Instance 5	-	-	**530***	33.68s
Instance 6	-	-	**842***	9.78s
Instance 7	-	-	**844***	2410.03s
Instance 8	-	-	**1237***	572.40s
Instance 9	-	-	**975***	3622.07s
Instance 10	-	-	**964**	-

Table 3.5: Table showing the best costs achieved for instances 1–10 using Chuffed.

with the direct model (direct) and the DFA model (reg) (we omitted the results for instances 11–24 as the solution process ran out of memory for these instances). The results formatted in boldface denote the overall best results, a * indicates that the solver could prove optimal costs within the time limit, and a – denotes the instances where no solution could be found.

We can clearly observe in Table 3.5 that the DFA-based model can solve more instances and prove more optimality results within the runtime limit in comparison with the direct model. For instances 1–3, for which both models were able to prove optimality, the DFA-based model could determine optimality in much shorter runtime, indicating that this model improves the performance of the solution process.

Integer Programming Results

As we implemented our models using the MiniZinc constraint modeling language, we could directly use the MiniZinc compiler to convert the direct model and the DFA-based model into MIP encodings. In the following, we present the experimental results produced

	cplex-6h	best bd	proof t	cplex-reg-6h	best bd	proof t
I 1	777	744	-	776	643	-
I 2	991	841	-	**842**	842	10992.47s
I 3	-	-	-	2761	961	-
I 4	-	-	-	12920	844	-
I 5	-	-	-	11085	225	-
I 6	-	-	-	1933	730	-
I 7	-	-	-	-	187.4	-
I 8	-	-	-	-	961	-
I 9	-	-	-	-	280.25	-
I 10	-	-	-	-	961	-

Table 3.6: Table showing the results achieved for instances 1–10 using CPLEX.

using these encodings together with the MIP solvers Gurobi and CPLEX.

Table 3.6 provides an overview of the best results for instances 1–10 with CPLEX (the solution process with encodings for instances 11–24 ran out of memory on our benchmark machine before any result could be produced). The columns on the left side of the table include the results achieved using the direct formulation: the cost of the best solution found within the runtime limit of 6 hours (cplex-6h), the best bound achieved using the direct formulation (best bd), and the required optimality proof time in seconds. The results in boldface denote the overall best results, and a – indicates that no solution could be found within the runtime limit. The right side of the table similarly shows the results achieved with CPLEX using the DFA formulation.

Table 3.6 shows that the direct formulation is only able produce solutions for the two smallest instances within the time limit, whereas the formulation using DFAs seems to be more efficient as it derived solutions for six instances and generated one optimality proof.

Table 3.7 provides the results achieved with Gurobi in our experiments. The results are presented in the same way as that in Table 3.6: The left side shows the best cost solutions produced with the direct model (gurobi-6h), followed by the best objective bound and the required optimality proof time. The right side of the table shows the best results achieved with the DFA-based problem formulation.

The results presented in Table 3.7 show that Gurobi can reach improved results relative to CPLEX for all instances, except instance 5. Moreover, Gurobi can provide optimality proofs for four instances. The DFA-based model leads to improved results relative to the direct formulation of the problem.

Comparison of Results

The summarized results of the experiments with the evaluated CP and MIP solvers are shown in Table 3.8. From left to right, columns 1–5 show the solver configuration (*6h*

	gurobi-6h	best bd	proof t	gurobi-reg-6h	best bd	proof t
I 1	**775**	768	-	**775**	775	6637.99s
I 2	**842**	842	9444.53s	**842**	842	127.56s
I 3	-	-	-	**961**	961	282.80s
I 4	-	-	-	967	862.49	-
I 5	-	-	-	**530**	530	13398.48s
I 6	-	-	-	-	842	-
I 7	-	-	-	904	841	-
I 8	-	-	-	-	964	-
I 9	-	-	-	-	582	-
I 10	-	-	-	-	961	-

Table 3.7: Table showing the results achieved for instances 1–10 using Gurobi.

solver	# best	# fast	# opt	# prf	# sol	avg nodes	avg rt	std nodes	std rt
chuffed-6h	4	0	4	3	4	96910	43.01	158722.61	32.53
chuffed-reg-6h	**10**	**9**	**9**	**9**	**10**	984148.89	735.66	1625073.26	1330.49
gecode-6h	0	0	0	0	0	-	-	-	-
gecode-reg-6h	0	0	0	0	0	-	-	-	-
cplex-6h	0	0	0	0	2	-	-	-	-
cplex-reg-6h	1	0	1	1	6	14375	10981.08	-	-
gurobi-6h	2	0	2	1	2	2149	9276.63	-	-
gurobi-reg-6h	4	0	4	4	6	110315.25	5071.33	213079.3	6269.04

Table 3.8: Table summarizing the experimental results achieved with all evaluated CP and MIP solvers using the proposed models.

indicates the time limit of 6 hours, the DFA-based model is compared with the direct model indicated by *reg*), the number of overall best cost solutions reached, the number of overall fastest optimality proofs, the number of solutions solved to optimality, and the number of optimal cost proofs. Columns 6–10 show the number of instances where a feasible solution could be obtained, the number of average expanded nodes (only for optimally solved solutions), the average proof time, the standard deviation of expanded nodes for optimally solved solutions, and the standard deviation of proof times.

The results shown in Table 3.8 indicate that Chuffed based on the DFA formulation clearly produces the best results for the most number of instances and provides the fastest optimality proofs in our experiments. By contrast, the Gecode solver could not reach any feasible solution in our experiments. Furthermore, the DFA formulation generally leads to improved results for the MIP solvers as the number of solved instances is higher than that of the results of the direct model.

Table 3.9 compares the overall best cost solutions achieved by the solvers within the time limit of 6 hours (6h). The results formatted in boldface indicate the best results per instance, and a * denotes proven optimal solutions.

The results presented in Table 3.9 show that Chuffed produced the best results in our experiments for instances 1–10 and that it can prove optimality for instances 1–9. By

	chuffed-6h	cplex-6h	gurobi-6h
I 1	**775***	776	**775***
I 2	**842***	**842***	**842***
I 3	**961***	2761	**961***
I 4	**918***	12920	967
I 5	**530***	11085	**530***
I 6	**842***	1933	-
I 7	**844***	-	904
I 8	**1237***	-	-
I 9	**975***	-	-
I 10	**964**	-	-
I 11	-	-	-
I 12	-	-	-
I 13	-	-	-
I 14	-	-	-
I 15	-	-	-
I 16	-	-	-
I 17	-	-	-
I 18	-	-	-
I 19	-	-	-
I 20	-	-	-
I 21	-	-	-
I 22	-	-	-
I 23	-	-	-
I 24	-	-	-

Table 3.9: Table showing the best results per instance produced by the exact methods.

contrast, Gecode was not able to solve a single instance. This result indicates that lazy clause learning together with an activity-based search heuristic is able to produce improved results for these instances relative to a non-learning solver that uses a fixed search. CPLEX was able to prove a single optimal solution (instance 2), and Gurobi was able to prove three optimal solutions (instances 2, 3, and 5) in our evaluation. Large instances (instances 11–24) could not be solved by the constraint modeling approach in our experimental setting as the MiniZinc compiler ran out of memory due to the excessive number of required variables and constraints.

CHAPTER 4

Heuristic and Hybrid Approaches for the Paint Shop Scheduling Problem

In the previous chapter we introduced an exact approach for the paint shop scheduling problem which was able to provide optimal results for several benchmark instances but could not provide feasible solutions for large realistic instances within reasonable space and runtime limitations. Therefore, we propose several heuristic approaches to tackle large instances for the PSSP in this chapter.

First, we introduce in Sections 4.1 and 4.2 a construction heuristic approach as well as a metaheuristic approach using local search and a set of novel neighborhood search operators. Afterwards, in Section 4.3 we introduce a large neighborhood search approach that can hybridize exact and heuristic techniques on an important color change sub-problem that arises in the PSSP. Finally, at the end of this chapter we evaluate all methods proposed on the set of benchmark instances we introduced in Chapter 2.

4.1 A Construction Heuristic Algorithm for Paint Shop Scheduling

As real-life instances of the paint shop scheduling problem occurring at the production sites of the automotive supply industry usually lead to a very large search-space, in practice often heuristic approaches are required. In this section, we therefore propose a greedy construction heuristic approach that is able to produce solution schedules in short runtime even for large-scale instances.

The method does not guarantee to always produce feasible solutions as some hard constraints may be violated. However, in practice the remaining violations can be repaired

by a human planner or can serve as an effective initial solution for the metaheuristic methods we describe later on in this chapter.

4.1.1 Construction Heuristic Phase 1: Constructing a round layout

A challenging property of the paint shop scheduling problem is that demanded materials and the associated carrier configurations have to be distributed over the rounds of the scheduling horizon. One strategy to keep the number of required carrier and color changes low in each round, is to minimize the number of scheduled colors per round while trying to reuse as many carrier types as possible between rounds. The first phase of our greedy construction heuristic algorithm follows this idea, while assigning carrier configurations and colors to each round without considering an exact round sequence at first.

Therefore, phase 1 of the construction heuristic inserts, given a customer demand, a carrier device with the configuration that maximizes the number of pieces for that demand. The color is specified by the demand. Insertion in one of the rounds is done greedily, i.e. minimize violations and costs with respect to the current state (where the current state consists of all previously inserted configurations plus the current insertion candidate).

Customer demands are processed one after the other, where the list of demands is sorted by the due dates.

Even without determining the exact carrier sequence, phase 1 can consider hard constraints that do not depend on the sequence and can calculate a lower bound of the objective. The calculation of the violations for a partial solution in the round layout phase of the construction heuristic includes violations of constraints that can be determined without knowing the final sequence. Therefore, round capacity violations, carrier availability violations, and minimum carrier type block size violations are evaluated by counting the number of excessive or missing carriers assigned to each round. Furthermore, carrier and color change costs are determined by calculating the lower bounds on the number of changes between two consecutive rounds (i.e. the least number of changes required for any sequencing based on the assignments to these rounds).

After all demands are fulfilled, the minimum round capacity requirement might still remain violated for some rounds. If this is the case, the construction heuristic continues to generate and perform carrier configuration insertions until the capacity requirements are met. In each iteration of this process the algorithm evaluates the violations and costs for each possible carrier configuration and color insertion to each of the rounds (which totals in $|K| \cdot |C| \cdot |R|$ possible combinations for $|R| \cdot q$ iterations in the worst case), and then selects the option that leads to the lowest costs (ties are broken lexicographically).

Algorithm 1 describes the details of Phase 1.

4.1.2 Phase 2: Determining the carrier sequence for each round

After the execution of Phase 1, the heuristic has decided which carrier configurations and colors should be scheduled within each round. Furthermore, Phase 1 has already consid-

Algorithm 1: Greedy Algorithm Phase 1

```
openDemands ← sort demands (earliest due round first)
schedule ← empty painting schedule
while openDemands not empty do
    demand ← next demand from openDemands
    configs ← all carrier configurations able to carry demand
    c ← color of demand
    r ← due round of demand
    while demand is unfulfilled do
        insertions ← []
        forall i ∈ {1, ..., r}, x ∈ configs do
            y ← carrier insertion with configuration x and color c to round i
            Add y to insertions
        z ← a ∈ insertions that leads to lowest violations and costs regarding the
          current state when assigned to schedule
        Perform z and update schedule
while minimum round capacity is unfulfilled do
    Calculate all possible carrier insertions
    Perform insertion leading to the lowest violations and costs in the current
      state
    Update schedule
return schedule = set of carrier assignments for each round
```

ered hard constraints that are not sequence-dependent (due round, carrier availability, and round capacity). Phase 2 therefore only determines the exact carrier sequence within each round while trying to fulfill sequence-dependent hard constraints and aiming for a low number of color and carrier type changes.

The main idea behind the second phase is to determine the carrier sequence one round at a time. It aims to keep the sequence from the previous round as closely as possible, i.e. carriers not used in the current round are removed. The remaining carriers are inserted greedily. Therefore, the algorithm will start with the first round and determine its sequence based on the scheduling sequence from the history round which is part of the input parameters.

After the sequence has been determined for round 1 the algorithm will continue to sequence round 2 and so on.

Further details regarding Phase 2 of the construction heuristic algorithm can be found in Algorithm 2.

Algorithm 2: Greedy Heuristic Phase 2

```
r ← number of rounds in the schedule
x ← set of carrier assignments per round from phase 1
y ← []
y[0] ← carrier sequence from history round
w ← []
for i ∈ {1, ..., r} do
    y[i] ← empty sequence
    y[i] ← copy carrier type assignments from y[i − 1] (remove color and
     configuration, only keep carrier type)
    forall c ∈ all colors used in x[i] do
        z ← set of all carrier assignments from x[i] with color c
        Place as many assignments from z as possible on compatible type
         assignments in y[i] that have not been assigned a color and configuration
         yet.
        Add any remaining assignments from z to w.
    Remove type assignments from y[i] that have not been assigned a color and
     configuration yet.
    forall u ∈ w do
        Calculate costs and violations for all possible insertion positions for u in
         y[i].
        Insert u at the position that leads to the lowest violations and costs.
return y = carrier sequences for each round
```

4.2 A Local Search Based Approach for Paint Shop Scheduling

In this section we introduce a local search based approach to solve the paint shop scheduling problem. We propose three different types of neighborhood moves, and several metaheuristic techniques to escape local optima.

4.2.1 Cost Function

We extend the objective function described in Equation 2.19 to also include a sum of all hard constraint violations hv that will be multiplied with a constant M that is guaranteed to be larger than the largest possible objective value. The sum of hard constraint violations will be calculated independently for each constraint in a way that captures the distance to a feasible solution. In the following, we define the distance function to a feasible solution for each hard constraint:

- *Material Demands:* All material demands must be fulfilled by their due round. Therefore, the distance function for this constraint simply counts the total number

of material pieces that are not produced in time.

- *Carrier Availabilities:* As the number of available carriers is restricted for each round, the distance function for this constraint sums up the total number of excess carriers.

- *Minimum/Maximum Carrier Capacities:* The distance function for this constraint counts the total number of missing carriers (regarding the minimum capacity) of each round that uses fewer carriers than the minimum. The maximum carrier capacity is always implicitly fulfilled by the solution representation.

- *Forbidden Carrier Type Sequences:* The distance function for this constraint counts the number of times that a forbidden carrier type transition occurs in the schedule.

- *Minimum/Maximum Carrier Type Block Lengths:* For each carrier type block that violates the minimum/maximum length requirement, the distance to the required minimum/maximum length is included in the total distance function for this constraint (E.g. let a carrier type block appear with length 2 in a candidate solution, and let the minimum block length for this carrier type be set to 5. Then a value of 3 is added to the total number of violations).

- *Forbidden Color Sequences:* The distance function for this constraint goes over all carrier positions in the schedule and adds the number of all subsequent forbidden color violations regarding the particular position to the total number of violations. For example, let color c_1 be assigned to a position in the schedule and let it be forbidden that color c_2 appears within the next 3 positions after c_1. Then, the number of times color c_2 appears within the next 3 positions after the position that used color c_1 is added to the number of total violations for this distance function. Note that the distance function considers all forbidden color sequences on each carrier position in the candidate schedule for the calculation of the total number of violations.

Using the total number of hard constraint violations hv, Equation 4.1 defines the extended objective function.

$$\begin{aligned} &minimize \sum_{r\in\{0,\ldots,n-1\}} sc_r^2 + \sum_{r\in R} cc_r^2 + hv \cdot M \\ &M = s^2 \cdot |R| + (maxColorCost)^2 \cdot |R| + 1 \\ &maxColorCost = \max\{f_c(c_1, c_2) | c_1, c_2 \in C\} \end{aligned} \tag{4.1}$$

4.2.2 Search Neighborhoods

We propose the following three neighborhood moves for local search:

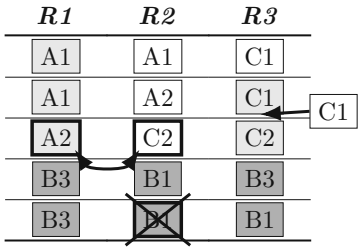

Figure 4.1: This figure shows a visualization of how the three neighborhood move types (swapping positions, delete positions, and insert positions) can make modifications to a paint shop schedule.

1. *Carrier removal*: Any carrier assignment that is placed in the schedule can be simply removed. Whenever a carrier is removed from a round, all carriers that have been planned after the removed position in the same round will be shifted forward by one position.

2. *Carrier insertion*: A new carrier assignment can be inserted in any round that has not reached its full capacity. Carriers that have been previously planned at or after the newly inserted carrier's position will be shifted backward by one position.

3. *Carrier swap*: Any two carrier assignments in the schedule can be swapped. In this case both the selected carrier configurations and colors are exchanged.

The local search approach we propose will also consider block moves where multiple consecutively scheduled carrier assignments may be inserted, deleted or swapped at once. In our experiments we used a maximum block move size that corresponds to the largest input parameter given with the minimum block length constraint ($\max\{b_t^{\min} | t \in T\}$), since block moves will be especially effective for repairing minimum block length violations. Figure 4.1 shows a visualization of the three neighborhood move types. In the following, whenever we say search move we refer to a single neighborhood move that can be either a carrier removal, insertion or swap.

4.2.3 Neighborhood Generation

Since early experiments revealed that generating the complete search neighborhood usually cannot be done within reasonable time for large instances, we propose to incorporate elements of a min-conflicts heuristic into our local search approach to focus on promising parts of the search neighborhood. Our algorithm will therefore track any carrier assignment in a candidate solution that is causing a constraint violation, a carrier-, or a color change. Furthermore, for constraints that require additional carrier configurations to be inserted into the schedule we track which carrier configurations are still missing. Our neighborhood generation routine will therefore consider two types of conflicts:

1. Position Conflicts: All positions in the schedule that are involved in at least one constraint violation will be considered to be in conflict.

2. Insertion Conflicts: Some constraints can be violated because of a number of missing carrier assignments in the schedule (e.g. demand constraint, min capacity). For those constraints we track information about what carrier configuration needs to be inserted to repair any violation. For some insertion conflicts it is irrelevant which color and configuration is inserted to repair the constraint violation (e.g. minimum round capacity constraint). Our algorithm will randomly select a configuration and color for insertion in such a case.

In addition to the conflict based moves, in each iteration we further generate a single random neighborhood move that is uniformly sampled from the complete search neighborhood. Thereby, the search has the chance to find improving moves that are not related to insertion or position conflicts. Algorithm 3 further describes our neighborhood generation routine.

Algorithm 3: Generate Neighborhood Moves

```
allMoves ← []
Calculate position and insertion conflicts
ic ← select random insertion conflict
pc ← select random position from position conflicts
sp ← generate random swap position
for i ← 1 to maxBlockMoveSize do
    Add insertion of size i based on ic to allMoves
    Add deletion of size i at pc to allMoves
    Add swap of size i for pc and sp to allMoves
Add randomly generated search move to allMoves
return allMoves
```

After Algorithm 3 has generated a collection of potential search moves, we propose two alternative methods to select the best neighborhood move. The first option always selects the neighborhood move that leads to the lowest cost value, while the second option uses a tabu list to prevent the repeated selection of recently performed moves (in this case the lowest cost move which is not tabu is selected). Note however, that the second option always selects a move that leads to a new unknown best solution even if it is contained in the tabu list.

4.2.4 Neighborhood Move Acceptance

We further propose to use an innovative simulated annealing based acceptance function that will decide whether a selected move should be accepted during a search iteration (if not accepted, no move will be performed in the current iteration). In addition to the

standard homogeneous simulated annealing temperature cooling scheme [KGV83], we incorporate a problem specific factor t' that will adjust the acceptance probability P based on the search progress (see Equation 4.2, where e and e' are the current- and the neighbor solution cost, and T is the current temperature).

$$P(e, e', T, t') = \exp(-(e' - e)/(T \cdot t')) \tag{4.2}$$

We included the factor t' into our algorithm due to the observation that the impact of unit improvements on the objective function (such as reducing the number of hard constraint violations by one or lowering the number of required color or carrier changes by one) will depend on the current objective value. The idea is to set t' to a value that roughly estimates the cost improvement that would occur to the current solution if the number of violations or required color and carrier changes is reduced by one. To calculate t' we do the following in each iteration: As long as the current solution violates any hard constraint, we simply set $t' = M$. However, if the current solution is feasible we instead calculate t' based on the current solution's cost as described in Algorithm 4 (The values *colorCosts* and *carrierCosts* store the sum of color- or carrier costs from the objective function.)

The rationale behind Algorithm 4 is to normalize the acceptance rate of moves that decrease the number of carrier or color changes during the overall search progress. This is done by calculating the average cost improvements that would occur to the current solution if the total number of color and carrier changes in the schedule is lowered by one. The calculated value depends on the quality of the current solution and significantly changes during the search progress.

Algorithm 4: Calculating t' if the current solution does not violate any hard constraints.

$a \leftarrow \frac{colorCosts}{numberOfRounds}$
$b \leftarrow \frac{carrierCosts}{numberOfRounds}$
$c \leftarrow (\sqrt{a} + maxColorCost)^2 - a$
$d \leftarrow (\sqrt{b} + 1)^2 - b$
$t' \leftarrow \min(c, d)$
return t'

4.3 A Large Neighborhood Search Approach for the Paint Shop Scheduling Problem

In the previous sections of this thesis we have proposed an exact solution method as well as a local search based approach for the paint shop scheduling problem. We now investigate a large neighborhood search approach that can hybridize exact and heuristic techniques to further improve results for many realistic benchmark instances.

We first identify an important NP-hard sub-problem of the paint shop scheduling problem regarding the minimization of color changes in the production sequence. Afterwards, we propose heuristic as well as exact approaches for this sub-problem. Then, we utilize the introduced techniques within a novel large neighborhood search operator that can be used to extend the metaheuristic methods we proposed in Section 4.2. Furthermore, we propose an innovative construction heuristic utilizing the large neighborhood search operator that serves as an alternative approach to the construction heuristic given in Section 4.1. At the end of the chapter we evaluate all proposed large neighborhood search methods using the proposed set of benchmark instances.

4.4 The Paint Shop Color Change Problem

In this section we describe the paint shop color change problem (PSCCP), which appears as a sub-problem within the PSSP. Afterwards, in Section 4.6, we will give a full specification of the PSSP and describe, how solution methods to the PSCCP can be utilized within a large-neighborhood search approach for the PSSP.

The main aim of the PSCCP is to find an optimized coloring of a carrier production sequence. Thus, a predetermined sequence of different carrier device types that are used to transport materials in the paint shop is given as input to the problem. In the paint shop, the materials which are transported on a single carrier have to be painted using one unique color, and therefore the aim of the PSCCP is to assign a single color to each individual carrier in such a way that the number of color changes in the production sequence is minimized.

A feasible coloring sequence has to ensure that all color demands are fulfilled, where color demands are given as part of the input in terms of carrier type quantities. To model a notion of time, the given carrier sequence which is part of the input is further grouped into several scheduling periods which are referred to as rounds since in industrial paint shops of the automotive supply industry carriers are usually moving on a cyclic conveyor belt system (for further information on the paint shop environment and its round layout see Section 2.1). In practice, processing of a single round is done within a fixed amount of time depending on the size of the paint shop, even though each round may schedule a different number of carriers. All color demands which are given as part of a problem instance set a due date (which is specified in terms of rounds) that needs to be fulfilled.

As an example, consider a simple instance of the PSCCP which is illustrated at the top of Figure 4.2.

The carrier sequence that is presented in Figure 4.2 contains three consecutive rounds called R1, R2, and R3; where the end of each period is visualized by tick marks. The letter in each of the cells denotes the carrier type in the sequence, so that within the first scheduling period R1 a sequence of four carriers is scheduled: C, B, C, B.

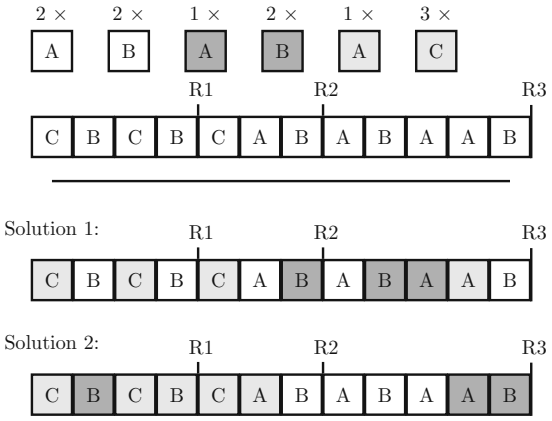

Figure 4.2: Illustration of a simple instance for the PSCCP together with two example solutions.

In addition to the predetermined carrier sequence for each scheduling period, an instance of the PSCCP also defines color demands which are illustrated at the very top of Figure 4.2. We can see that for this example two carriers of type A as well as two carriers of type B need to be painted using a white color, whereas one carrier of type A and two carriers of type B need to be painted in a dark gray color and so on. Note that each color demand usually also has a due date in terms of a scheduling period. For reasons of simplicity in the illustrated example all color demands are due until the end of the scheduling period R3 and therefore in this case a solution is feasible as long as all color demands are scheduled in the sequence.

The bottom of Figure 4.2 further illustrates two feasible solutions to the example instance. Solution 1 has been assigned using a naive approach where each carrier has been colored greedily from left to right and causes 10 color changes in total. Solution 2 on the other hand shows an improved solution that causes only four color changes. Note that the schedule actually contains a total of 12 carriers, whereas only 11 carriers are required to be painted. However, this is still a valid instance of the problem, as in such a case one of the carriers may be painted arbitrarily to minimize the overall number of color changes. A scenario like in this example where the number of carriers appearing in the production sequence is larger than the total number of demanded carriers can also occur in practice as technical requirements of the paint shop conveyor belt systems might not allow ejecting all carrier devices at once between rounds.

In the following, we provide a formal definition of PSCCP, where we for reasons of simplicity make use of the Iverson bracket notation[1].

[1] $[P] = 1$, if $P = true$ and $[P] = 0$ if $P = false$

4.4.1 Input parameters

An instance of the PSCCP specifies several parameters including a predetermined production sequence, information about carrier types, colors and demands. In the following we describe all parameters in detail:

A set of carrier types T is given as part of the input which includes all the different types of carrier devices that appear in the production sequence.

The set of colors C specifies all colors that can be assigned to the carriers in the schedule.

The demands for an instance are given as a set D that consists of 4-tuples, where each tuple defines the demand quantity $(a_d, d \in D)$, the carrier type of the demand $u_d, d \in D$, the demanded color $v_d, d \in D$ and the demand's due date $w_d, d \in D$. For example, consider a demand d_1 where $a_{d_1} = 10$, $u_{d_1} = t_1$, $v_{d_1} = c_1$, and $w_d = 5$. Then this demand d_1 would require 10 carrier devices of type t_1 in the schedule to be painted using color c_1 by round 5 (note that the due date is given in terms of paint shop rounds).

A history color h is given as part of the input which denotes the last color used in the previous production schedule (before the production sequence of the instance). The previous schedule is not part of the current problem, however the color assigned to the very first carrier of the problem might cause a color change regarding the history color.

The number of scheduling rounds $n, (R = \{1, \ldots, n\})$ denotes the number of rounds that are processed in the given production sequence. For each of these rounds, the number of scheduled carriers is given by a parameter $s_r, r \in R$. Finally, the detailed predetermined production sequence for the instance is determined by a list consisting of the scheduled carrier type sequence for each round $(l_{i,j}, \forall i \in R, j \in \{1, \ldots, s_i\})$.

The full list of formal input parameters is summarized in Table 4.1.

Set of carrier types:	T
Set of colors:	C
Set of demands:	D
Quantity of demand:	$a_d \in \mathbb{N}_{>0}, \forall d \in D$
Carrier type of demand:	$u_d \in T, \forall d \in D$
Color of demand:	$v_d \in C, \forall d \in D$
Due date of demand:	$w_d \in \mathbb{N}_{>0}, \forall d \in D$
History Color:	$h \in C$
Number of scheduling rounds:	$n\ (R = \{1, \ldots, n\})$
Number of carriers per round:	$s_r \in \mathbb{N}_{>0}, \forall r \in R$
List of scheduled carriers:	$l_{i,j} \in T,$ $\forall i \in R, j \in \{1, \ldots, s_i\}$

Table 4.1: The input parameters of the PSCCP.

4.4.2 Decision Variables

The set of decision variables for the PSCCP decide which color should be used for each scheduled carrier position in the production sequence:

Scheduled color in round i and position j

$$x_{i,j} \in C, \forall i \in R, j \in \{1, \ldots, s_i\} \tag{4.3}$$

4.4.3 Hard Constraints

In feasible solutions to the PSCCP, all color demands need to be fulfilled in time:

$$\sum_{d \in D} a_d \leq \sum_{i \in \{1,\ldots,w_d\}} \sum_{j \in \{1,\ldots,s_i\}} [x_{i,j} = v_d \wedge l_{i,j} = u_d] \quad \forall d \in D \tag{4.4}$$

4.4.4 Objective Function

The objective function builds a sum of all color changes in the production sequence including round overlapping changes, and a possible change from the last color used in the history schedule (history color):

$$\begin{aligned} \textit{minimize } [h \neq x_{1,1}] + \sum_{i \in R} \sum_{j \in \{1,\ldots,s_i-1\}} [x_{i,j} \neq x_{i,j+1}], \\ + \sum_{i \in \{1,\ldots,n-1\}} [x_{i,s_i} \neq x_{i+1,1}] \end{aligned} \tag{4.5}$$

4.4.5 Related Literature

A color change sequencing problem that is similar to the PSCCP has been studied under the name of the paint shop problem (PSP) in the literature [EHO04]. Indeed, one can view the PSCCP as an extension of the PSP that additionally includes due date constraints. However, as the PSP does not consider due date violations, solution methods for the PSP can in general not be used to find feasible solutions for instances of the PSCCP.

The PSP essentially asks to find an optimal assignment of a given set of colored letters to a predetermined word in such a way that color changes are minimized. In [EHO04] the authors provide a dynamic programming algorithm that can solve the PSP in polynomial time if the number of letters and colors are bounded and show that the decision variant of the problem is NP-complete otherwise. A local search approach using a swap neighborhood and an exact method based on linear programming for the PSP have been proposed in [MN12]. The authors randomly generate 15 benchmark instances to experimentally evaluate their methods. They conclude that the local search

approach overall produced better results than the linear programming approach in their experiments.

4.4.6 Complexity Analysis

If we consider instances of the PSCCP with only a single round in the sequence (i.e. the due date constraint is not violated as long as all demands are fulfilled at any time in the sequence) and further ignore the history color of the problem, the PSCCP is equivalent to the PSP. Thus, the PSCCP is a generalization of the PSP. As the decision variant of the PSP has been shown to be NP-complete in [EHO04] as long as the number of colors and different types in the sequence are not bounded, we can argue that the decision variant of the PSCCP also is NP-hard under the same assumptions, as we can solve the PSP with the PSCCP by simply creating an instance for the PSCCP with a single round in the sequence and no due date constraints.

4.5 Solution Methods

In this section, we propose two solution approaches to the PSCCP: An innovative heuristic approach as well as an exact approach based on constraint modeling.

4.5.1 A Heuristic Solution Approach for the PSCCP

The main idea behind this heuristic approach is to greedily determine the coloring of a single carrier in the given sequence one step at a time and thereby constructing a solution. To find out which carrier should be colored next during each iteration of the algorithm the heuristic essentially evaluates all possible single color assignments to the current partially colored carrier sequence.

Figure 4.3 illustrates the main execution steps of the construction heuristic we propose for the PSCCP.

At first, the algorithm starts with a fully unpainted production sequence and creates an ordered list of all color demands ordered by their due dates. As long as this list is not empty, the algorithm then iteratively processes a series of execution steps. These steps essentially go over all demands still contained in the list and calculate how much it would cost to color a single carrier in the production sequence to fulfill the associated demand. The single carrier color assignment that causes the lowest cost increase with the current state of the production sequence is selected and applied to the partial solution. Afterwards, the quantity of the demand affected by this color assignment is updated and in case the demand quantity is lowered to zero, the demand is completely removed from the demand list.

Algorithm 5 provides pseudocode to describe the proposed construction heuristic algorithm in further detail.

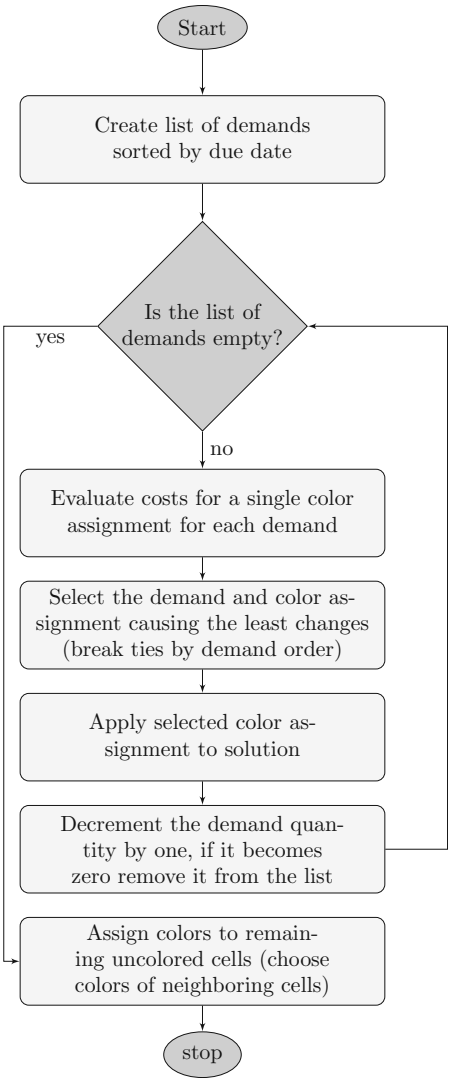

Figure 4.3: Main execution steps of the heuristic solution method for the PSCCP.

Algorithm 5: PSCCP Heuristic

```
fn GetBestColorPosition (demand d)
    bestCost = −1
    bestPos = null
    seq = reversed carrier sequence starting from w_d
    for position in seq do
        Color position with v_d
        cost = Evaluate costs for current sequence
        Uncolor position
        if bestCost == −1 or cost < bestCost then
            bestCost = cost
            bestPos = position
    return ⟨bestCost, bestPos⟩
fn PSCCPHeuristic (demandLookAhead)
    unfulfilledDemands = sort demands by due round
    while |unfulfilledDemands| > 0 do
        counter = 0
        bestDemand = null
        bestCost = ∞
        bestPosition = null
        for d in unfulfilledDemands do
            if counter == demandLookAhead then
                break
            ⟨cost, position⟩ = GetBestColorPosition(d)
            if bestCost > cost then
                bestCost = cost
                bestPosition = position
                bestDemand = d
            counter = counter + 1
        if bestDemand == null then
            break
        Apply color to bestPosition
        Update unfulfilledDemands
    for position in Sequence do
        Apply color of previously colored position
```

The function PSCCPHeuristic is the main entry point of the heuristic and takes in addition to the instance parameters a single positive integer parameter which is called *demandLookAhead*. In a first step, the list of all demands is sorted by their due dates. If there are multiple demands with the same due date, rare colors (which are determined by the total requested quantity for a color over all demands) are selected first. The idea here is that rare colors are potentially harder to group in the schedule and therefore should be scheduled earlier on.

After all demands have been ordered, the algorithm then enters its main loop that continuously applies single color assignments to the carrier sequence until all demands are fulfilled. Within the loop, the heuristic goes over the next *demandLookAhead* demands that are not already fulfilled by the current partially colored schedule. For each of these demands the costs caused for any possible single color assignment that can potentially fulfill the demand are calculated. Eventually, in each iteration the algorithm performs the single color assignment that causes the lowest costs (Ties are broken by selecting the best cost assignment that was encountered first).

For some instances, it can be the case that some carriers remain unpainted even after all demands have been fulfilled. This situation can occur in some real-life instances, for example when additional carriers are required in the schedule to fulfill minimum carrier capacity requirements. In such a case the heuristic simply goes over the sequence and tries to color those remaining unpainted carriers based on what colors have been assigned to neighboring carriers to keep the number of color changes as low as possible.

Although the heuristic can find candidate solutions quickly even for large instances (for realistic instances with several 100 demands the *demandLookAhead* parameter can be lowered if necessary), in general it does not guarantee to find a feasible solution as due round constraint violations might occur in the resulting schedule.

4.5.2 An Exact Approach for the PSCCP

To approach the PSCCP with state-of-the-art exact CP and mixed-integer programming (MIP) solvers, we propose to model the problem with the use of the high-level constraint modeling language MiniZinc [NSB+07]. This allows us to implement the problem definition from Section 4.4 in a declarative way and utilize the model with state-of-the-art CP and MIP solvers as an exact solution approach to the PSCCP.

In our constraint model, we define all the input parameters from Table 4.1 using integer value IDs in the ranges from 1 to $|T|$ for the carrier types, from 1 to $|C|$ for the colors, and from 1 to $|D|$ for the demands. We use an additional input $s = \max\{s_r | r \in R\}$, that is set to the maximum round length. The value s is used to define the input carrier sequence as a two-dimensional integer array of dimensions $|R| \times s$. Each position to the array is either set to the scheduled carrier id, or to 0 if the position is unused.

We model the decision variables from Equation (4.3) using a two-dimensional integer array and set the variable domain to $\{0, \ldots, c\}$, where 0 will be used to mark unused positions.

The due date constraint from Equation (4.4) is modeled with the use of a counting predicate that counts all occurrences of the associated color and carrier type combination for each relevant due date in the schedule. The resulting value is then constrained to be greater or equal to the required quantity by the due date.

Another constraint sets all unused positions in the decision variable array to 0.

Finally, the solution objective uses counting predicates to model the conditional sums from Equation (4.5).

Listing 4.1 displays the detailed MiniZinc model for the PSCCP. For details on the syntax of the MiniZinc language please refer to a recent version of the MiniZinc Handbook[2].

```
% INPUT
% carrier types
int: t;
set of int: T = 1..t;

% colors
int: c;
set of int: C = 1..c;

% history color
int: h;

% rounds to schedule
int: r;
set of int: R = 1..r;

% carrier sequence
int: s;
set of int: S = 1..s;
array[R,S] of 0..t: carriers;
array[R] of S: s_r;

% carrier demands
int: num_demands;
set of int: D = 1..num_demands;
array[D] of int: d_t;
array[D] of int: d_c;
array[D] of int: d_qty;
array[D] of int: d_r;

% VARIABLES
```

[2] https://www.minizinc.org/

```
array[R, S] of var 0..c: x;

% CONSTRAINTS
% All demands must be satisfied in time
constraint forall(d in D where d_r[d] <= r) (
  d_qty[d]
  <=
  count(i in 1..d_r[d], j in 1..s_r[i])(x[i,j] = d_c[d] /\
    carriers[i,j] = d_t[d])
);

% set unused positions to zero
constraint forall(i in R, j in s_r[i]+1..s) (
  x[i,j] = 0
);

% OBJECTIVE
solve minimize bool2int(h != x[1,1]) +
  count(i in 1..r, j in 1..s_r[i]-1)(x[i,j] != x[i,j+1]) +
  count(i in 1..r-1)(x[i,s_r[i]] != x[i+1,1]);
```

Listing 4.1: MiniZinc model code for the PSCCP

4.6 A Large Neighborhood Search Approach for the Paint Shop Scheduling Problem

In this section, we propose an innovative large neighborhood search operator for the Paint Shop Scheduling Problem (PSSP) that utilizes the solution methods for the PSCCP that have been introduced in the previous section. First, we describe how solutions to the PSCCP can be utilized to improve candidate schedules for the PSSP in Section 4.6.1. Afterwards, in sections 4.6.2 and 4.6.3, we introduce the large neighborhood search operator and describe how it can be used to improve local search for the PSSP.

4.6.1 Utilizing the PSCCP to Improve PSSP Solutions

Consider the example solution for an instance of the PSSP shown in Figure 4.4.

We can view the PSCCP as a sub-problem that appears within the PSSP, as once a carrier sequence has been determined, we can solve an associated instance of the PSCCP to find an optimized coloring for the associated carrier sequence. Figure 4.5 visualizes a solution to the PSCCP instance that corresponds to the PSSP example schedule shown in Figure 4.4.

We now describe how solutions methods to the PSCCP can be utilized to improve given candidate solutions to the PSSP without changing the predetermined carrier sequence.

	R1	*R2*	*R3*	...
1	A1	A1	C1	...
2	A1	A1	C2	...
3	A2	C2	C3	...
4	B1	B2	B1	...
5	B2	B3	B2	...

Figure 4.4: A simple paint shop schedule, which illustrates a candidate solution to the PSSP.

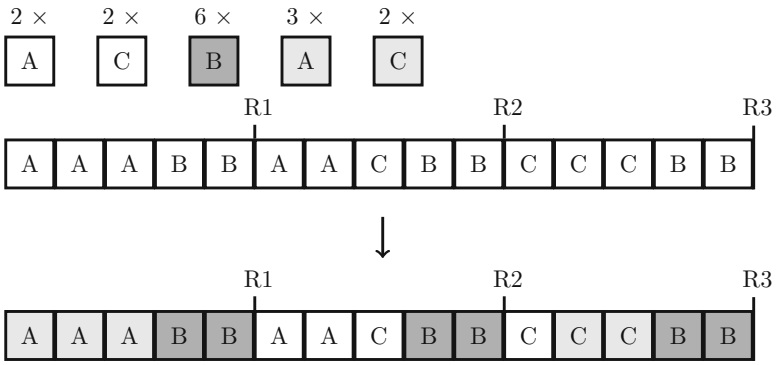

Figure 4.5: Visualization of a simple PSCCP instance that would be associated to the candidate schedule for the PSSP as it is shown in Figure 4.4.

This technique has the benefit that the number of required color changes in the production sequence can be improved without the need to consider the complex constraints of the PSSP that are related to the creation of feasible carrier type sequences. For this purpose, we take any candidate solution to the PSSP and create an instance for the PSCCP by simply removing the color assignments from all carriers in the sequence. We further generate demands for the PSCCP instance by looking at each individual carrier and by analyzing which paint shop demands are fulfilled by this carrier. The due round of the earliest demand that is processed by each individual carrier serves as the due round of the associated color demand in the PSCCP instance. Similarly, a solution to the generated PSCCP instance can be applied to the original candidate solution of the PSSP by applying the produced color sequence to the predetermined carrier sequence.

Note that in the original PSSP each carrier type actually can be used in a number of different configurations. Selecting a configuration can affect which and how many product types are placed on the carrier. When we create a PSCCP instance, we may ignore these configurations as long as we fulfill all the color demands, since we can safely remap the previously used configurations to the associated carrier type color pairs of the PSCCP solution later when we apply the PSCCP solution back to the PSSP schedule.

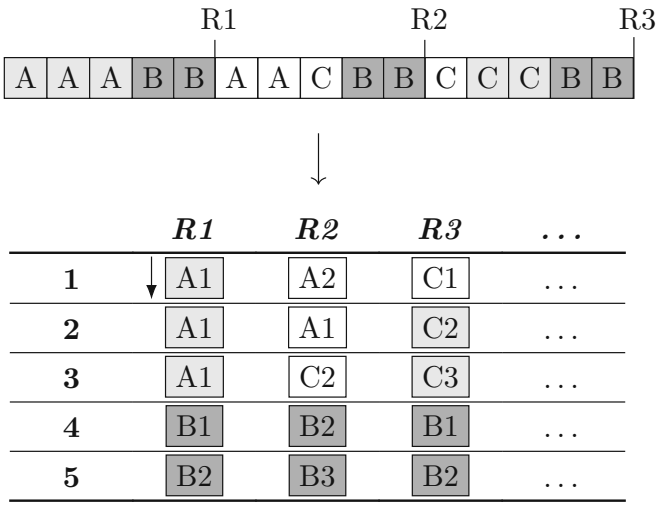

Figure 4.6: An example PSCCP solution is mapped back to the paint shop schedule for the original PSSP.

Figure 4.6 illustrates this process by displaying a solution to the PSCCP example that was previously shown in Figure 4.5 as well as a remapping to the PSSP schedule that was originally shown in Figure 4.4.

The top of Figure 4.6 visualizes the solution to the PSCCP, whereas the bottom shows the application of the solution to the PSSP schedule. Note that carrier types (which are represented by letters) have not changed compared to the original schedule shown in Figure 4.4. However, a carrier of type A with configuration number 2 ($A2$) that was previously scheduled in R1 at position 3 now is scheduled at position 1 of R2. Furthermore, position 3 in R1 now schedules a type A carrier using configuration number 1 ($A1$). The reason why the configuration numbers need to change in this example is that the demands for the associated PSSP instance require a white carrier of configuration $A2$ and a light gray carrier of configuration $A1$. This remapping of the configurations can be easily done algorithmically by going over all changed carrier positions and reassigning the configurations to fulfill any missing demands of the PSSP with an earliest demands first strategy.

Up to now, we have seen that the PSCCP solution methods cannot directly change the configurations used in the corresponding PSSP schedule, but only indirectly affect their positioning in the schedule. However, if we pass additional information about the configurations (i.e. what product types are associated with each configuration) to an instance of the PSCCP and specify the demands in a way to request product types instead of carrier types we could give the solution methods more possibilities to fulfill the demands by not only reassigning colors but also reassigning configurations in the carrier type sequence. We initially experimented with extended variants of the PSCCP that support the reassignment of configurations, but these variants turned out to be impractical for

most benchmark instances with exact methods due to the largely increased size of the search space. However, we could successfully adapt the heuristic approach to support a reassignment of configurations without a notable loss in performance and therefore implemented this variant for our experimental evaluation of the PSCCP heuristic. The procedure shown in Algorithm 5 is still used with the only difference that when a position is evaluated to be colored for a given demand, the heuristic tries to assign the best configuration (i.e. the one which produces the most pieces for the demand) to this position. After execution has finished, the assigned colors and configurations are then transferred back to the PSSP solution.

4.6.2 A Large Neighborhood Search Operator for the PSSP

We now describe how the state-of-the-art metaheuristic technique for the PSSP which we previously proposed in Section 4.2 can utilize solution methods for the PSCCP to improve candidate schedules during local search. Therefore, we propose a large neighborhood search (LNS) operator Φ_{LNS} for the PSSP that takes a candidate solution, solves the corresponding PSCCP problem to find an optimal coloring for the schedule, and then applies the optimal coloring to the PSSP candidate solution. Although Φ_{LNS} cannot be used to solve the complete PSSP problem, as it is unable to make any changes on the carrier sequence, it can effectively improve color change costs of the given schedule.

To implement Φ_{LNS} we can directly use the solution methods we proposed in sections 4.5.2 and 4.5.1. However, in practice we need an additional time limit parameter that causes the operator to stop if solving the associated PSCCP takes too long. In the case of an exact solver, we can still use any intermediate solution on a timeout, whereas for the heuristic approach we simply exit the main loop early and apply colors of adjacent positions for unpainted positions if time runs out.

In initial experiments with Φ_{LNS} we discovered that for many of the realistically sized instances of the PSSP the operator usually ran out of time before any improvement could be achieved. The main reason for this result is that the corresponding PSCCP (which is an NP-hard sub-problem on its own) has a search space that was simply too large to be effectively used within a local search neighborhood for the original PSSP.

Therefore, we further propose another LNS operator Φ_{LNS*} that destroys only parts of the colored carrier sequence and therefore reduces the search space for the associated PSCCP. The main idea behind Φ_{LNS*} is to leave large areas in the sequence that use only a single color intact, as they do not cause any color changes. In consequence, only color assignments from the remaining areas will be reassigned during the application of Φ_{LNS*}. Thus, many color assignments are predetermined in the corresponding PSCCP instances and only a smaller number of carriers needs to be colored which speeds up the solution process of the Φ_{LNS*}. The intuition behind Φ_{LNS*} is further illustrated in Figure 4.7.

At the top of the figure, we see a sequence that uses six different colors (for simplicity we omit carrier types here and assume they are similar). The carriers colored in shades

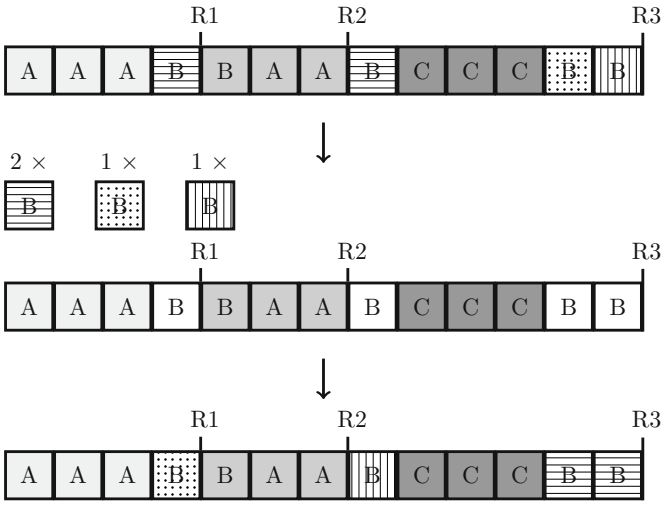

Figure 4.7: Example how Φ_{LNS*} can reduce the number of color changes by rearranging small color blocks in the overall sequence.

of gray are already arranged into "color blocks" of length 3, whereas the carriers using different pattern colors (horizontal stripes, dots, and vertical stripes) are not yet well arranged. The idea behind Φ_{LNS*} is to remove only small color block assignments from the sequence (which in this example are the pattern colors) while leaving larger color block assignments intact (which in this example are the gray colors). This is visualized in the middle of Figure 4.7.

Finally, solving the corresponding PSCCP instance in this example only needs to reassign four colors and can actually find an improved solution that reduces the number of required color changes in total (the corresponding sequence is illustrated at the bottom of Figure 4.7).

In addition to a time limit, we introduce a second parameter $k \in \mathbb{N}_{>0}$ for the operator Φ_{LNS*} to configure which areas of the sequence should have their colors reassigned. The parameter works in a way that any consecutive block of carriers that use a single color with a length that is greater or equal to k will be left intact. In Figure 4.7 for example, $k = 3$ as single color blocks of consecutive carriers with length ≥ 3 are not reassigned.

4.6.3 Integrating the LNS Operator into Local Search for the PSSP

The simulated annealing approach we proposed in Section 4.2 uses three search neighborhoods to solve the PSSP: *Carrier Insertion*, *Carrier Removal*, and *Carrier Swap*. These existing neighborhoods consider the insertion, removal or swapping of carriers in the schedule and can affect either single carriers or blocks of consecutive carriers. To incorporate the Φ_{LNS*} operator into the local search approach, we simply add a parameter $\alpha \in [0, 1]$ which defines the probability to call the LNS operator instead of a standard neighborhood move during a local search iteration. Furthermore, we only call

the Φ_{LNS*} operator if the current solution has no hard constraint violations as the LNS operator improves only the color change objective.

Figure 4.8 illustrates how the LNS operator is called within a local search iteration.

The candidate solution produced by application of the LNS operator is accepted by local search (i.e. the result will be used as the current solution for the next search iteration) if the costs of the candidate solution are either improved, or the result leads to a new minimum color change cost. The latter is the case if a new upper bound for color change costs has been achieved by LNS, but overall solution costs are not improved (e.g. because a hard constraint has been violated). If this is the case, the LNS result is still used for the next search iteration and local search may try to improve the overall best-known solution from this point within a limited number of iterations (which is determined by an additional parameter β). If no overall best cost solution can be achieved within β iterations, local search resets its current solution back to the previously known best solution. The rationale behind this fallback and the β parameter is to accept low color cost solutions that include some hard constraint violations (e.g. unfulfilled demands) to give local search a chance to repair these violations quickly and to potentially find a new best solution.

Algorithm 6 presents the pseudo-code with further details on how the LNS operator is called during local search.

4.7 A Novel Construction Heuristic for the PSSP

Previously in sections 4.1 and 4.2, we proposed a construction heuristic (CH) for the PSSP. In this section, we propose an alternative construction heuristic (CH*) that utilizes the solution methods for the PSCCP to color partially unpainted schedules during the creation of an initial schedule.

The pseudo-code in Algorithm 7 presents the details about the novel construction heuristic CH* that we propose in this chapter.

The main idea behind CH* is to first copy the unpainted carrier type sequence of the given history round to all rounds in the schedule, essentially keeping the number of carrier changes between each of the sequences at 0. In a second step, the construction heuristic solves an associated instance of the PSCCP to find an optimized coloring to the candidate schedule. After this coloring has been determined, there might still be remaining unfulfilled demands as not all required carrier types are necessarily included in the copied history sequence. If this is the case, the heuristic iteratively tries to insert a new carrier into each round of the schedule to handle remaining unfulfilled demands and again solves a partial instance of the PSCCP, where any colors that have been previously assigned stay fixed. The process is repeated until either no new carriers can be inserted due to resource limits or if all demands are fulfilled. Figure 4.9 illustrates the main steps of the construction heuristic.

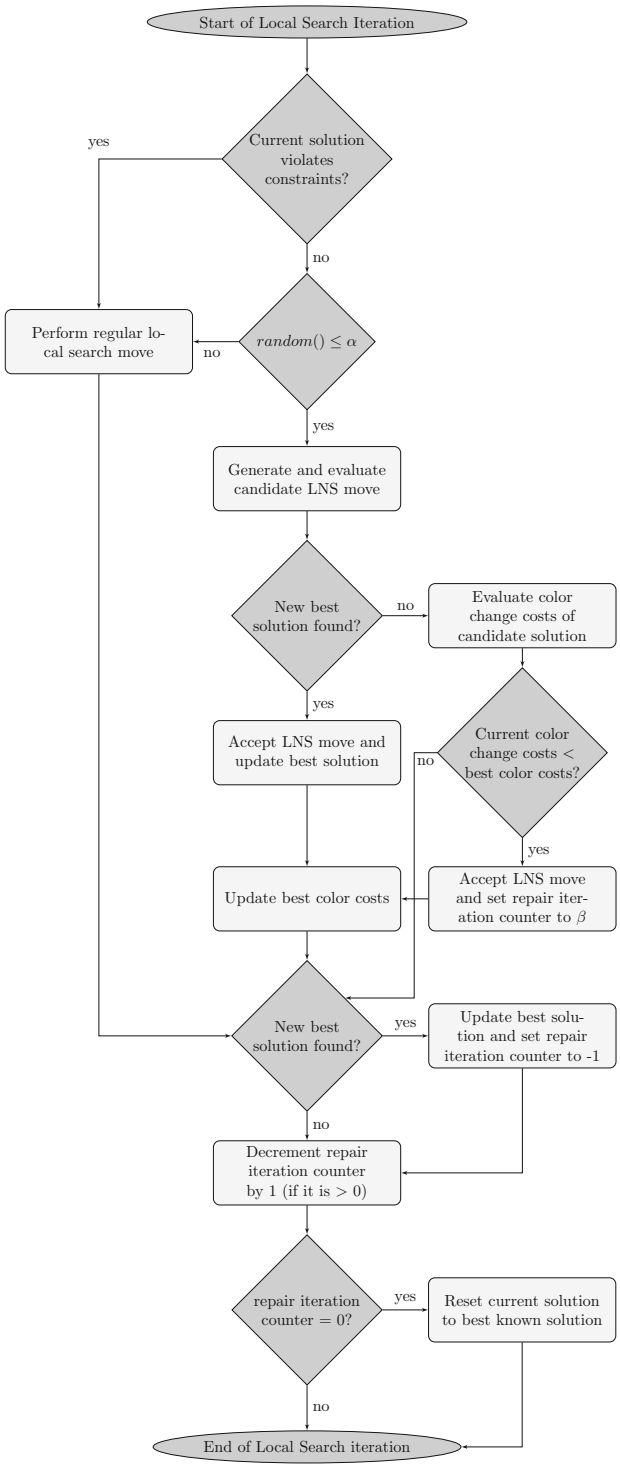

Figure 4.8: Main execution steps during a local search iteration for the proposed PSSP algorithm that utilizes the LNS operator.

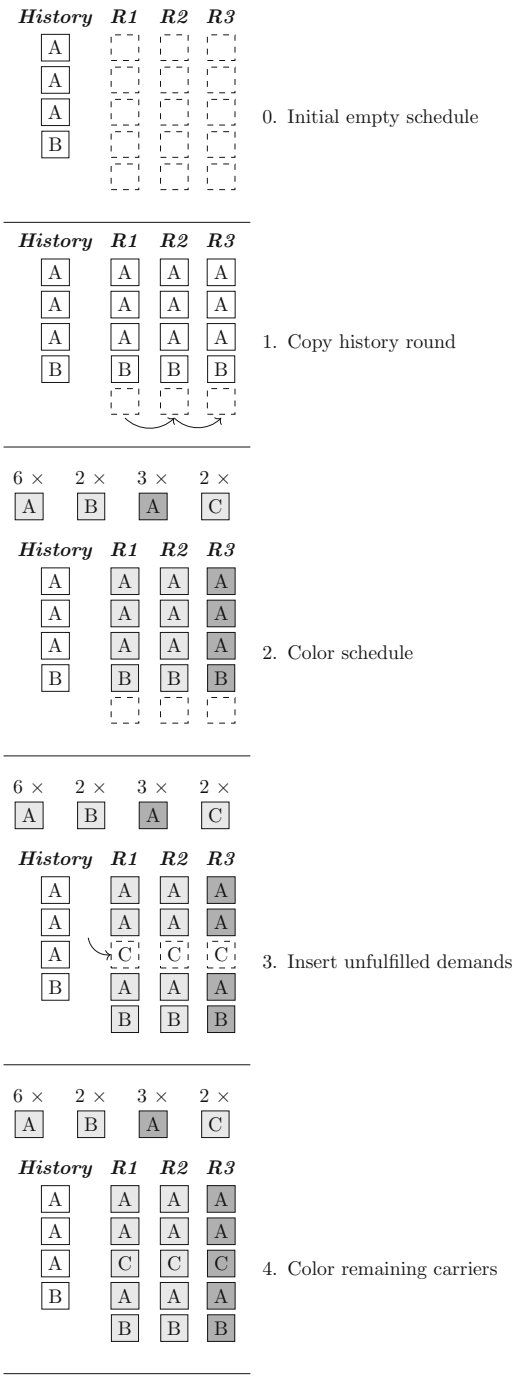

Figure 4.9: Illustration of the main processing steps of the proposed construction heuristic (CH*). Steps 3 and 4 are repeated until all demands are fulfilled or no more carriers can be inserted due to resource limits.

Algorithm 6: Incorporating Φ_{LNS*} within local search

```
lnsRepairIterationCounter = 0
while main local search loop do
    if currentSolution.Violations == 0 ∧ random.Next() ≤ α then
        lnsSolution = perform Φ_LNS*(k)
        if lnsSolution.Cost < bestSolution.Cost then
            currentSolution = lnsSolution
            bestSolution = currentSolution
        else
            if lnsSolution.Cost < currentSolution.Cost then
                currentSolution = lnsSolution
            else
                colorChgCount = CountColorChanges(lnsSolution)
                if colorChgCount < bestColorChgCount then
                    currentSolution = lnsSolution
                    lnsRepairIterationCounter = β
        bestColorChgCount = min {colorChgCount, bestColorChgCount}
    else
        perform standard local search iteration
    if currentSolution.Cost < bestSolution.Cost then
        bestSolution = currentSolution
        lnsRepairIterationCounter = 0
    if lnsRepairIterationCounter > 0 then
        lnsRepairIterationCounter = lnsRepairIterationCounter − 1
        if lnsRepairIterationCounter == 0 then
            currentSolution = bestSolution
```

The rationale behind CH* is to keep the number of carrier changes between the rounds as low as possible by copying the history round, while at the same time the number of color changes is minimized by solving corresponding instances of the PSCCP. CH* does not guarantee to produce a feasible schedule (note that CH also does not guarantee this), however, it is able to provide an initial schedule that is usually very low in costs compared to initial schedules produced by CH.

Algorithm 7 and Figure 4.9 describe and illustrate the main processing steps and the core idea of the alternative construction heuristic CH* we propose in this section. However, at this point we want to note that we included additional minor modifications for the construction heuristic in our implementation to consider minimum- and maximum-consecutive carrier block constraint violations when generating the initial carrier sequence. These constraints affect feasible carrier sequences such that numbers of consecutively

Algorithm 7: A novel construction heuristic for the PSSP

fn *CreateInitialSchedule*
1. copy history sequence to all rounds
2. Solve PSCCP problem to find coloring
while $\exists$*unfulfilled demands* $\vee$ *carrier limits reached* **do**
3. insert new carrier in all rounds
4. Solve PSCCP problem to find coloring

return *colored schedule*

scheduled carriers of the same type are restricted. We simply try to fix potential constraint violations by going over the carrier sequence and insert or remove single carriers. This is done twice in our implementation of CH*: First after the history round has been copied, and a second time before returning the initial schedule. Please refer to Chapter 2 for details about the minimum- and maximum-consecutive carrier block constraints.

4.8 Empirical Evaluation

In the following sections, we perform an experimental evaluation of all the heuristic methods for the PSSP that we propose in this chapter using the set of benchmark instances that we provided in Chapter 2.

First, we report and discuss in Section 4.8.1 experimental results produced by the construction heuristic and metaheuristic approach we proposed in sections 4.1 and 4.2. Afterwards, in Section 4.8.2 we perform an extensive evaluation of the large neighborhood search based approach as well as the advanced construction heuristic that we proposed in sections 4.6 and 4.7 before we finally compare the overall best heuristic results with results produced by exact methods.

4.8.1 Experiments with the Metaheuristic Approach

To evaluate the local search approach that we proposed in Section 4.2, we first conducted a set of benchmark experiments using an Intel Xeon E5345 2.33 GHz CPU with 48 GB RAM with a runtime limit of 60 minutes.

In early experiments we evaluated our search neighborhoods with a simple random walk move generation and standard simulated annealing techniques, however this approach could not produce feasible solutions for the larger instances.

Afterwards, we implemented the adaptive simulated annealing scheme as well as the conflict based neighborhood generation technique that we introduced in Section 4.2 and early experiments showed that this approach was able to produce feasible solutions for all but the four largest instances within the time limit. Although the greedy algorithm was not able to find any feasible solutions on its own, we could further utilize greedily

constructed initial solutions together with the metaheuristic approach to produce feasible solutions for all of our benchmark instances. We then decided to evaluate two variants of the combined greedy and local search approach: One variant that will always select the best move from the generated neighborhood and a second variant that will use a tabu list to prevent the repeated selection of recently performed moves. Based on manual tuning attempts in early experiments we set the following parameters: Initial temperature $t_1 = 0.25$, tabu list length $tl = 0.001$ (relative to the instance size), cooling rate $\alpha = 0.95$. Table 4.2 gives an overview of the results with the different evaluated metaheuristic approaches and compares them to the best results produce with the exact methods from Chapter 3.

Columns 2 and 3 of Table 4.2 show the results achieved with standard simulated annealing compared to the other metaheuristic approaches from Section 4.2 that make use of an adaptive simulated annealing acceptance without the greedy algorithm (in this case local search will start from an empty schedule). The results show that the proposed local search methods produced better results for the majority of the smaller instances (1–12) and most of the larger instances (13–20). Although the standard simulated annealing approach can process search iterations much faster and produced better results for two of the small instances, the results show that the proposed local search methods were more robust in our experiments especially when it comes to solving larger instances.

As discussed previously in Chapter 3, exact methods could produce optimal results for nine of the smaller instances and could provide good solutions for one additional instance. The methods we have proposed in this chapter were able to produce feasible solutions for all instances and could provide the best results for all the large practical sized instances. Starting from a greedily generated solution did not always have positive effects on the results for instances 1–12, however for the larger instances 13–24 methods incorporating greedily constructed solutions produced the best results. Adding a tabu list mechanism to our metaheuristic approach did not lead to improved results for most of the instances, although this technique could produce the best results for instances 15, 20 and 23.

4.8.2 Experimental Evaluation of the Large Neighborhood Search Approach

In this section, we provide an extensive experimental evaluation of the large neighborhood search operator and the alternative construction heuristic we proposed in sections 4.6 and 4.7. First, we describe the setup and computational environment we have used to conduct our benchmark experiments at the beginning of this section. Afterwards, we elaborate on how parameters for the heuristic algorithms have been selected using state-of-the-art automated parameter tuning software. Finally, the results of all our experiments are presented and discussed at the end of this section.

	SA	LS	LS/G	LS/G/T	EM
I1	—	1028	844	882	**775***
I2	896	868	932	927	**842***
I3	1011	990	992	994	**961***
I4	—	1016	975	1050	**918***
I5	618	616	593	599	**530***
I6	913	887	891	895	**842***
I7	1120	1084	1088	1137	**844***
I8	—	1871	1834	2553	**1237***
I9	—	1767	1735	2421	**975***
I10	1134	1262	1243	1269	**964**
I11	**5236**	6298	5476	6439	—
I12	6753	**5723**	7916	8274	—
I13	—	2097235	**116235**	123830	—
I14	—	1985513	**118628**	130552	—
I15	—	8159361	180863	**172679**	—
I16	—	8621490	**262252**	262897	—
I17	—	23320626	**421777**	455321	—
I18	—	23947097	**581021**	606917	—
I19	—	34294393	**555829**	576225	—
I20	—	34713814	930564	**927822**	—
I21	—	—	**917955**	957854	—
I22	—	—	**1128716**	1142530	—
I23	—	—	1889804	**1884125**	—
I24	—	—	**2086450**	—	—

Table 4.2: Summary of the achieved objective cost values (total solution cost as defined in Equation 2.19, — if no feasible solution could be achieved) for all instances produced with the standard simulated annealing (SA), the metaheuristic methods and adaptive simulated annealing acceptance (LS), the combined approach using the proposed local search methods and the greedy algorithm (LS/G), the combined approach that also uses a tabu list (LS/G/T), and the best results produced by exact methods (EM). The best result within each line is formatted in boldface. Results marked with a * denote proven optimal solutions.

Experimental Environment

To evaluate the performance of the proposed LNS operator Φ_{LNS*} and the novel construction heuristic CH* for the PSSP, we extended the code for the simulated annealing based metaheuristic approach (we chose the variant that was able to find solutions for all benchmark instances in the results shown in the previous section) to implement these methods.

As some benchmark instances for the PSSP define hard constraints that impose forbidden color sequences, we adapted our implementation of the PSCCP Heuristic (see Algorithm 5) to include violations to this constraint in its cost evaluation function (i.e. the best color position is the one which introduces the lowest number of forbidden color violations, ties are broken by color change costs). Furthermore, we used a performance efficient implementation of the cost evaluation function that utilizes incremental evaluation (i.e. only areas in the schedule that have been modified since the last evaluation call will be reevaluated).

To incorporate the forbidden color constraint in the MiniZinc model (Listing 4.1), we initially experimented with the deterministic finite automaton encoding we previously proposed in Chapter 3. However, early experiments showed that it was more effective to not include the forbidden color constraint in the model so that the solution time needed by the large neighborhood search operator is reduced. If forbidden color violations are caused by the operator, the local search process is usually able to quickly repair any forbidden color violations. We further used an up-to-date version of the MiniZinc software [NSB+07] to solve instances of the PSCCP with recent versions of the CP solver chuffed [Chu11] and the MIP solver gurobi [GO20].

As the PSSP actually uses squared color change costs per round in its objective function instead of a simple summation of the changes, we further incorporated such a solution objective in both the exact modeling approach and the heuristic approach for the PSCCP by slightly changing the solution objective to consider the squared color changes per round. Additionally, in some instances for the PSSP the costs for a single color change may in rare cases vary depending on the specific pair of colors which are involved. We considered these specific costs in the implementation of the heuristic PSCCP approach, however decided to not include it in the MiniZinc model as it drastically slowed down the model compilation and solution process in early experiments.

Initial experiments further showed that the PSCCP solution process of the exact solvers was too time-intensive for the repeated call in CH*, which caused an impractical runtime of the heuristic. Therefore, we evaluated only an implementation of CH* that uses the PSCCP heuristic in our experiments. The LNS operator on the other hand was still evaluated using both the heuristic and the exact solution methods for the PSCCP.

To compare the simulated annealing based approach from Section 4.2 as well as the PSSP construction heuristic from Section 4.1 with the large neighborhood search based methods, we evaluated a variety of different configurations of the LNS operator and the

Solver ID	Description
LS	Local search approach using simulated annealing from Section 4.2, starting from an empty initial solution
LS/C	As LS, but starting from an initial solution that was created by the construction heuristic from Section 4.1
LS/C*	As LS/C, but using the advanced construction heuristic from Section 4.7 to create an initial solution
LNS-H	As LS, but including the proposed LNS operator that uses the heuristic solution method for the PSCCP, starting from an empty solution
LNS-H/C*	As LNS-H but starting from a solution that is created from the advanced construction heuristic
LNS-CP	As LNS-H but using the exact MiniZinc approach with the chuffed CP solver
LNS-CP/C*	As LNS-CP but starting from a solution created by the advanced construction heuristic
LNS-IP	As LNS-H but using the exact MiniZinc approach with the gurobi MIP solver
LNS-IP/C*	As LNS-IP but starting from a solution created by the advanced construction heuristic

Table 4.3: Overview of the evaluated solution methods.

advanced construction heuristic in our experiments. Table 4.3 gives an overview of all the different evaluated solution methods.

Column 1 of the table displays an abbreviation that will be used to refer to this method later in this section, whereas Column 2 describes the configuration of this method. All evaluated approaches use local search and randomly select moves, therefore we conducted 10 repeated runs for each instance and used the arithmetic mean solution costs for our evaluations if not stated otherwise.

All experiments reported in this section were conducted on a computing cluster with 10 identical nodes, each having 24 cores, an Intel(R) Xeon(R) CPU E5–2650 v4 @ 2.20GHz and 252 GB RAM.

Parameter Configuration

To configure the heuristic methods we propose in this chapter, we need to select a number of parameters. In a first step we selected reasonable defaults for each parameter based on some manual tuning runs with a few realistically sized instances. Afterwards, we used a recent version of the state-of-the-art automated parameter configuration software SMAC [LEF+17], which we used to tune the parameters. Table 4.4 provides an overview of all configured parameters.

Parameter	Description	Range	Default	Tuning Result
initial temperature	The initial temperature used by simulated annealing	[0.1, 0.5]	0.25	0.1297
cooling rate	The cooling rate used by simulated annealing	[0.9, 0.99]	0.95	0.9462
k	Configures which color assignments are destroyed and repaired during a lns move	{2, 5, 10, 20, 40}	5	2
α	The probability to conduct a LNS operator move in a local search iteration	[0.00001, 0.001]	0.0001	0.000044
β	Configures how many iterations can be used by local search to find a new best result after a LNS move	{100, 200, 500, 1000, 10000}	1000	10000
demand look ahead	Configures how many demands are considered when searching for a color assignment in the novel construction heuristic	{1, 20, 50, 100, 150}	50	20
time limit	Configures the time limit (in seconds) for the PSCCP solution method within LNS	{60, 90, 120, 180}	60	120

Table 4.4: Overview of the configured parameters and tuning results.

The first column of Table 4.4 displays the parameter name, while the second column shows a brief description of the parameter. Column three presents the allowed parameter range for the tuning process, and column four displays the default value given to SMAC (we selected the ranges manually so that they include a reasonable range near the default value).

The approach from [WMDM19] used two manually tuned parameters: The initial temperature and cooling rate for simulated annealing (see the default values in Table 4.4). We decided to tune these two parameters in a first tuning process using the simulated annealing approach that uses the existing construction heuristic (LS/C). In a second tuning process we tuned the parameters for the LNS method and novel construction heuristic proposed in this chapter. Therefore, we handed the solution method that uses the novel construction heuristic and the heuristic PSCCP solution method (LNS-H/C*) to the second tuning process.

We executed both tuning processes with SMAC using the following settings: Instances 1–24 were used as the training set, and we set the cutoff time per instance to 30 minutes. The overall tuning process was given a wall clock time limit of 4 full days.

Note that we use the same set of instances for the training set which is later used to evaluate the final experimental results with the tuned parameters, as we are mainly interested in finding strong upper bounds for the benchmark instances and therefore aim

for an optimized algorithm configuration for these instances. However, we want to point out that using the full set of benchmark instances as a training set can cause overfitting of the parameters regarding unseen problem instances and thereby have a negative impact on the performance of the algorithms when dealing with instances that are not included in the training set. Therefore, a robustness analysis of the tuned parameters with the use of additional unseen instances (that could be generated for example with a random instance generator or by gathering novel real-life scheduling scenarios) is an important subject for future work.

The tuned results for all parameters are shown in Column 5 of Table 4.4. We used these parameter settings for all of our final experiments and set the runtime limit to 1 hour.

Computational Results

An overview of the final results for the 24 paint shop scheduling benchmark instances is presented in Table 4.5.

Columns 2–10 show the relative per instance results for each of the evaluated solvers (see Table 4.3 for an explanation about the solver IDs). To calculate relative per instance results, the mean solution costs (produced by 10 repeated experimental runs) were divided by the overall best mean solution costs per row. In other words this means that a value of 1 indicates an overall best mean cost result for an instance, and values greater than 1 display the mean costs relative to the best mean costs in our experiments.

We can see in the results shown in Table 4.5, that approaches which are starting the search from a heuristically generated solution are able to find feasible solutions for all 24 benchmark instances, whereas approaches that start from an empty initial solution cannot produce solutions for instances 23 and 24 within the time limit. Furthermore, the results show that starting from a heuristically constructed schedule leads to better results for the larger instances 13–24. However, for the small-to medium-sized instances 1–12, for some instances the best results were achieved by solution methods that started from an empty initial schedule. This indicates that generating a good solution quickly is beneficial especially for large-scale instances, where starting from an empty schedule would be too time-intensive.

We can see that results produced by the metaheuristic methods from Section 4.2 are improved for all instances by at least one of the solution methods that use the novel construction heuristic or the LNS operator. This shows the strength of the LNS techniques, especially for the large instances, where many results are improved by factors larger than 4.

By looking at the results produced with the LNS operator we see differences in the quality of the results depending on the utilized PSCCP solution approach. Overall, the approach that uses the heuristic PSCCP solution method produces the best results for the majority of the instances, however the exact PSCCP solution approach using the CP solver chuffed can reach the best results for many of the smaller instances. The approach

Inst.	LS/C	LS/C*	LS	LNS-H/C*	LNS-H	LNS-CP/C*	LNS-CP	LNS-IP/C*	LNS-IP
I 1	1.14	1.11	1.13	1.06	1.06	**1**	1.05	1.07	1.06
I 2	1.12	1.05	1.03	1.08	1.03	**1**	1.02	1.1	1.04
I 3	1.05	**1**	1.05	**1**	1.07	**1**	1.04	**1**	1.06
I 4	1.08	1.06	1.1	1.07	1.06	1.03	**1**	1.03	1.07
I 5	1.11	**1**	1.11	**1**	1.19	**1**	1.18	**1**	1.19
I 6	1.03	**1**	1.03	1.01	1.05	1.02	1.04	**1**	1.05
I 7	1.22	1.01	1.22	1.01	1.28	**1**	1.26	1.02	1.26
I 8	1.09	1.05	1.02	1.05	1.04	1.03	**1**	1.2	1.06
I 9	1.13	1.12	1.09	1.1	1.06	1.08	**1**	1.38	1.15
I 10	1.03	**1**	1.03	1.03	1.07	1.01	1.05	1.04	1.04
I 11	1.4	1.08	1.38	1.09	1.49	**1**	1.38	1.17	1.43
I 12	1.38	**1**	1.11	1.08	1.24	1.02	1.2	1.21	1.2
I 13	10.82	1.07	255.51	**1**	205.52	1.05	283.54	1.37	261.09
I 14	9.13	1.19	145.91	**1**	122.31	1.04	162.4	1.67	131.98
I 15	5.25	1.54	252.21	**1**	193.19	1.35	254.73	1.43	248.85
I 16	2.97	1.06	107.54	**1**	84.6	1.22	111.05	1.16	102.74
I 17	3.16	1.32	214.34	**1**	171.9	1.26	219.45	1.38	218.38
I 18	1.63	1.06	47.53	**1**	37.9	1.23	47.83	1.01	48.17
I 19	4.19	1.57	270.05	**1**	222.06	1.74	272.74	1.72	271.31
I 20	1.51	1.16	53.08	**1**	42.62	1.29	53.54	1.25	53.61
I 21	5.65	1.07	284.26	**1**	226.61	1.2	285.26	1.23	284.18
I 22	2.07	1.15	60.56	**1**	56.32	1.1	60.73	1.27	60.89
I 23	4.04	1.23		**1**		1.35		1.33	
I 24	1.58	1.02		1.12		1.08		**1**	

Table 4.5: Overview of the relative results for each of the evaluated solution methods.

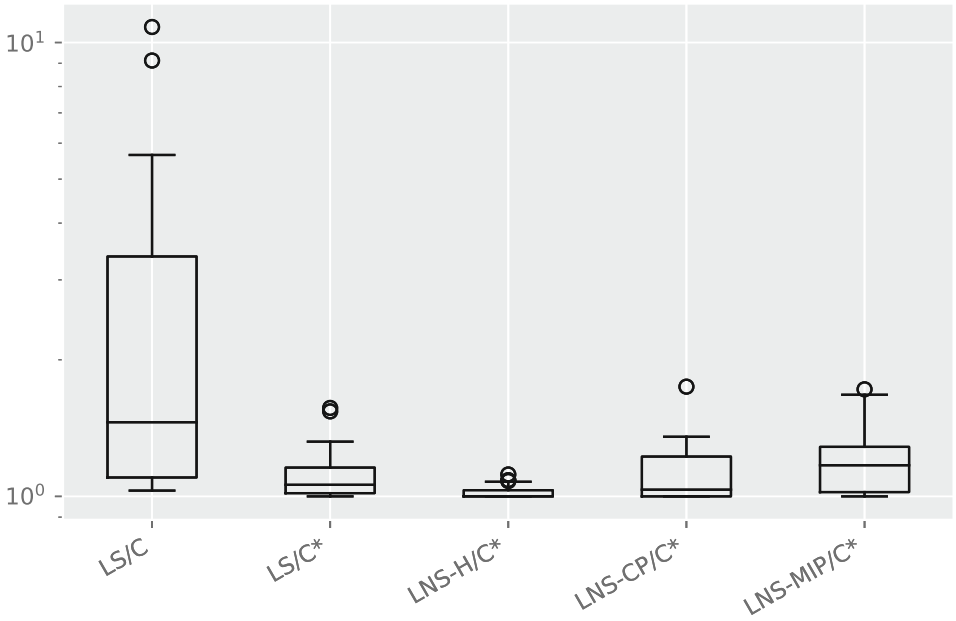

Figure 4.10: Box plots comparing the relative mean cost results produced by approaches that use a construction heuristic to generate an initial solution.

using the Gurobi integer programming solver for the PSCCP produces similar results, and reaches the best results for 3 of the smaller instances and the largest instance.

Table 4.6 presents the absolute best results for all evaluated approaches. The displayed costs are the best costs out of the 10 repeated experimental runs for each method and instance.

The best results per row are formatted in boldface. Overall, we can see that the best cost results show a similar outcome as the relative mean results. However, as this table presents the single best result out of ten repeated runs, we can see some outliers that do not match the best relative mean results. For example, we can see that the traditional local search approach is able to produce the best result for instance 8, although it did not achieve the best score in the mean cost comparison. Furthermore, the best solution for the largest instance in this case is produced by the LNS-CP/C* approach, although the best relative mean cost results were achieved using the LNS-IP/C* approach.

Figure 4.10 shows a comparison of all approaches that start from an initial solution that was generated by a construction heuristic.

The figure visualizes the relative mean cost results (using a logarithmic scale) for all 24 instances as box plots. We can see that approaches using C* overall lead to the best results in our experiments with a median value close to 1, whereas the C boxplot has a median value at roughly 1.5 and in general a much wider range with some outliers even

Inst.	LS/C	LS/C*	LS	LNS-H/C*	LNS-H	LNS-CP/C*	LNS-CP	LNS-IP/C*	LNS-IP
I 1	822	808	849	802	825	795	798	834	**794**
I 2	889	865	876	929	880	**844**	847	937	871
I 3	988	**961**	990	**961**	1009	**961**	971	**961**	1007
I 4	966	956	994	956	953	**930**	943	974	951
I 5	574	**530**	577	**530**	600	531	599	**530**	598
I 6	875	**845**	863	850	888	853	872	849	879
I 7	1032	867	1033	**856**	1089	858	1056	882	1063
I 8	1496	1480	**1426**	1489	1437	1459	1485	1770	1434
I 9	1345	1321	**1240**	1332	**1240**	1308	1272	1561	1316
I 10	1077	1058	1088	1085	1121	**1053**	1111	1082	1099
I 11	4318	3346	4357	3268	4608	**3030**	4353	3595	4407
I 12	5238	**3463**	4168	3699	4463	3697	4625	4501	4476
I 13	74236	7379	1683619	**6592**	1406142	6815	1577682	9023	1788255
I 14	110714	10341	1710055	9374	1503198	**9108**	1923650	12460	1329706
I 15	153728	28385	7291013	**25427**	5590121	29240	7481155	32572	7332157
I 16	213003	47097	7690901	**42765**	5995974	51953	8005671	61039	5898146
I 17	323829	105397	21716866	**68841**	17103946	107991	22135745	95567	21884939
I 18	597419	315377	22684889	**285572**	17709696	381400	22764282	311956	22865284
I 19	497486	108953	32447139	**96825**	25871047	144226	32631108	130235	32544683
I 20	913110	491845	33061150	**476506**	26323904	609803	33287708	568279	33573545
I 21	937094	**126750**	48199455	135757	37984334	146352	48381085	157073	47520813
I 22	1674595	615530	49354821	535846	40215243	**486698**	49219231	690072	49633330
I 23	1714000	495635		**357051**		506831		522690	
I 24	2609884	1209816		1290190		**1141429**		1327593	

Table 4.6: Overview on the best results produced by the evaluated methods.

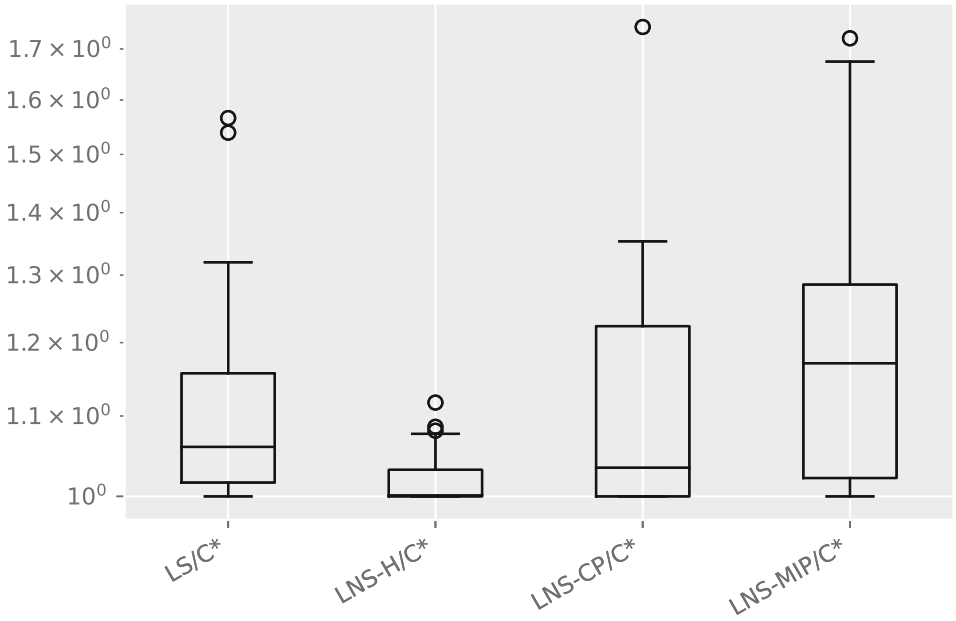

Figure 4.11: Box plots comparing the relative mean best results produced by approaches that use the C* construction heuristic to generate an initial solution.

lying above the value 9.

Figure 4.11 shows box plots only for such evaluated approaches that use the C* construction heuristic.

We can see that overall the approach using a large neighborhood search operator that utilizes the heuristic PSCCP solution method produces the best results, having a median value close to 1 and the smallest interquartile range. When comparing the approaches using exact PSCCP solution methods with the approach using no LNS operator, we can see that the LNS approach using chuffed has the lowest median, followed by the existing local search approach which has the second smallest interquartile range.

In Figure 4.12 we can see box plots for the mean costs results for instances 1–22 which were produced by the approaches that start from an empty initial schedule.

Again the LNS approach that uses the heuristic PSCCP solution method has the smallest median value, followed by the LNS approach using the CP solver to solve the PSCCP before the MIP-based LNS approach and the traditional local search approach. This indicates that the LNS techniques can improve results even when they are used without an initial construction heuristic.

Table 4.7 displays the relative mean costs produced by local search together with the basic construction heuristic approach in direct comparison with the LNS variant that uses the heuristic PSCCP solution method together with the advanced construction heuristic.

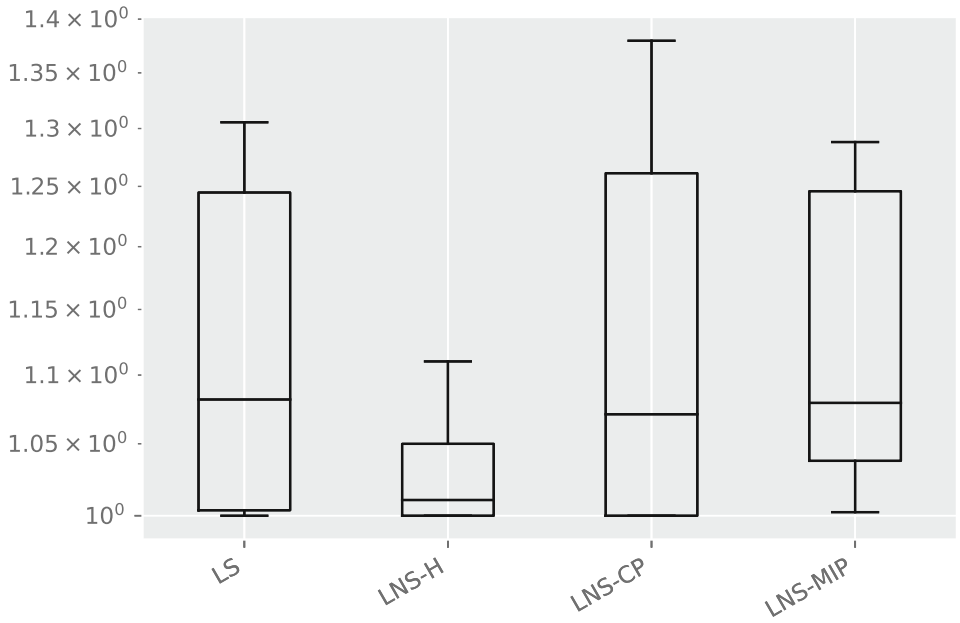

Figure 4.12: Box plots comparing the relative mean best results produced by approaches that start from an empty initial solution.

We selected these two methods for a direct comparison as the former was the overall best performing approach that does not utilize the LNS operator, and the latter is the overall best performing approach using LNS.

The results in Table 4.7 show that the LNS approach produced the best results for all instances in this comparison, and was even better than the existing approach in 23 out of 24 cases.

Finally, we compare the best upper bounds on the solution costs for each of the 24 PSSP benchmark instances produced by LNS with best upper bounds achieved by exact methods in Table 4.8.

Column 2 displays the best solutions produced with exact solution methods for the PSSP from Chapter 3. Note that exact methods could further prove that the best results for instances 1–9 are optimal. The best results produced by the LNS solution methods proposed in this section are shown in Column 3. Best results per instance are formatted in boldface.

The results in Table 4.8 show that the LNS approach is able to provide the best upper bounds for instances 11–24. Exact methods still produce the best results for instances 1–10, but the LNS approaches are able to reach optimal results for two of the instances and several additional nearly optimal results for small instances.

Instance	LS/C	LNS-H/C*
I 1	1.14	**1.06**
I 2	1.12	**1.08**
I 3	1.05	**1**
I 4	1.08	**1.07**
I 5	1.11	**1**
I 6	1.03	**1.01**
I 7	1.22	**1.01**
I 8	1.09	**1.05**
I 9	1.13	**1.1**
I 10	**1.03**	**1.03**
I 11	1.4	**1.09**
I 12	1.38	**1.08**
I 13	10.82	**1**
I 14	9.13	**1**
I 15	5.25	**1**
I 16	2.97	**1**
I 17	3.16	**1**
I 18	1.63	**1**
I 19	4.19	**1**
I 20	1.51	**1**
I 21	5.65	**1**
I 22	2.07	**1**
I 23	4.04	**1**
I 24	1.58	**1.12**

Table 4.7: Direct comparison of the relative mean cost results produced by the overall best traditional local search method with the overall best LNS method proposed in this section.

Instance	EM	LNS
I 1	**775**	794
I 2	**842**	844
I 3	**961**	**961**
I 4	**918**	930
I 5	**530**	**530**
I 6	**842**	845
I 7	**844**	856
I 8	**1237**	1434
I 9	**975**	1272
I 10	**964**	1053
I 11		**3030**
I 12		**3463**
I 13		**6592**
I 14		**9108**
I 15		**25427**
I 16		**42765**
I 17		**68841**
I 18		**285572**
I 19		**96825**
I 20		**476506**
I 21		**126750**
I 22		**486698**
I 23		**357051**
I 24		**1141429**

Table 4.8: Overview on the best upper bounds for all 24 PSSP benchmark instances that were produced by exact methods and the LNS methods proposed in this section.

CHAPTER 5

String Edit Distance Constraints

Previously, we introduced the formal problem definition of the paint shop scheduling problem in Chapter 2 and further explained that one of the problem's main objectives is to minimize carrier change costs between consecutive scheduling rounds. Furthermore, we identified that the minimum number of required carrier changes between two consecutive rounds can be calculated by using efficient string edit distance algorithms. Afterwards in Chapter 3, we additionally proposed constraint modeling techniques to model the carrier change objective function.

In this chapter, we investigate an alternative way to model such string edit distance constraints by using a novel global string edit distance constraint. We additionally propose an efficient constraint propagator for this constraint together with a strategy to explain the performed propagations. Thereby, we make it possible to use our propagator with powerful lazy clause generation solvers.

In the following sections, we first provide some preliminaries and related literature before we later introduce the novel global constraint together with an algorithm that can compute minimal explanations for the associated propagator. At the end of the chapter, we experimentally show that the proposed methods can be successfully used to improve results for the paint shop scheduling problem and another NP-hard problem from the literature.

5.1 Preliminaries

In this section we briefly provide some background on string edit distance and lazy clause generation, as we will later assume that the reader is familiar with these topics.

5.1.1 String Edit Distance

The notion of edit distance was first introduced in [WF74] and has been thoroughly studied in the literature ever since e.g. [Ukk85, Nav01]. In the following we give a short review of the definition as well as the traditional dynamic programming routine to calculate the edit distance.

The *edit distance* between two given strings s and t over alphabet Σ is defined as the minimum number of editing operations required to transform s into t (or vice versa). Let a given string s consist of n characters $s_1, s_2, \ldots s_n$ and another string t consist of m characters $t_1, t_2, \ldots, t_m$, then we distinguish between three different editing operations:

1. $s_i \to t_j$ denotes a change of the character at position i in string s to the character at position j in the second string t.
2. $s_i \to \epsilon$ denotes a removal of the character at position i in string s.
3. $\epsilon \to t_j$ denotes an insertion of the character at position j in string t.

The minimum edit distance between two strings s and t can be calculated with the use of dynamic programming (where ϵ denotes an empty string, $s(i)$ denotes the sub-string $s[1:i]$ of s, that is the first i characters of the string s or ϵ if $i = 0$, γ denotes the cost of an edit operation, and $d(i,j)$ denotes the minimum edit distance between $s(i)$ and $t(j)$):

$$\begin{aligned} d(0,0) &= 0 \\ d(i,j) &= \min \begin{cases} d(i-1,j-1) + \gamma(s_i \to t_j) \\ d(i,j-1) + \gamma(\epsilon \to t_j) \\ d(i-1,j) + \gamma(s_i \to \epsilon) \end{cases} \end{aligned} \tag{5.1}$$

The dynamic programming routine can be used to compute the edit distance between two strings as long as the following triangle inequality holds for the costs of the edit operations:

$$\gamma(s_i \to t_j) \leq \gamma(\epsilon \to t_j) + \gamma(s_i \to \epsilon) \tag{5.2}$$

We assume that if $s_i = t_j$ then $\gamma(s_i \to t_j) = 0$, that is the edit distance of making no change is 0.

Throughout this chapter in our examples we assume $\gamma(\epsilon \to c) = \gamma(c \to \epsilon) = 1$ for all $c \in \Sigma$ and $\gamma(c_1 \to c_2) = 2$ if $c_1 \neq c_2$ for all $\{c_1, c_2\} \subseteq \Sigma$. Nevertheless, the presented techniques can be applied also on any other cost assignments, as long as the triangle inequality holds.

Example 1: As an example, consider two given strings s and t where $s = ABBC$ and $t = ACB$ and assume that costs for insertion/deletion are set to 1 while substitution cost

is set to 2. We can use the dynamic programming approach shown in Equation 5.1 to find out that the minimum edit distance between s and t is 3. To visualize the calculation of the dynamic programming routine it is helpful to illustrate intermediate results for all recursion steps as a matrix, where each line in the matrix represents a letter of string s while each column represents a letter of string t (ϵ represents an empty string). Each cell of such a dynamic programming matrix will be set to the value of $d(i, j)$ where i is the associated row number and j is the associated column number. Figure 5.1 illustrates the full dynamic programming matrix for Example 1, and shows that finding the minimum edit distance corresponds to finding the shortest path through the dynamic programming matrix if one imagines a directed arc network that connects adjacent single cells of the matrix. Horizontal/vertical arcs will go in rightwards/downwards direction and have a length of one in our example, as they represent single character insertion/removal. Diagonal arcs represent single character substitution and have a length of two if the characters mismatch or a length of zero if two characters are equal. □

5.1.2 Lazy Clause Generation

In this chapter we propose an explaining propagator that can be used with a lazy clause generation (LCG) solver [OSC09]. A LCG solver tracks information about the reasons for any propagated domain changes during search and stores explanations for each propagation. In case of a failure, these explanations can then be used to compute so called *nogoods*, which record the reason for the failure in form of novel constraints. These *nogoods* can then prevent the search from making similar sets of faulty decisions later.

A LCG solver furthermore uses Boolean variables to represent integer variables. For example, a variable x with a domain $D(x) = [l \dots u]$ will be represented by Boolean variables $[\![x = d]\!], l \leq d \leq u$ and $[\![x \leq d]\!], l \leq d < u$. We use $[\![x \neq d]\!]$ to represent $\neg[\![x = d]\!]$ and $[\![x \geq d]\!]$ to represent $\neg[\![x \leq d - 1]\!]$. To explain a propagation, a LCG solver will define clauses over these Boolean variables. We will provide some examples later when we describe how to explain propagation for the constraint propagator that we propose in this chapter.

5.2 Related Literature

String comparison and matching are well studied topics in computer science which have spawned a large number of publications in the past e.g. [WF74, Ukk85, Nav01]. One of the most widely used methods to quantify the similarity of two given strings is the so-called string edit distance [WF74], that counts the number of required edit operations to transform a string into another given string.

Algorithms that can compute the edit distance between two given strings have been thoroughly studied, and several methods that use dynamic programming have been suggested to efficiently calculate the minimum edit distance in polynomial time e.g. [Ukk85]. However, there also exist NP-hard combinatorial optimization problems that aim to

	ϵ	A	C	B
ϵ	0	1	2	3
A	1	0	1	2
B	2	1	2	1
B	3	2	3	2
C	4	3	2	3

(a) DP Matrix

A	$-$	B	B	C
A	C	B	$-$	$-$

(b) Alignment of Strings

Figure 5.1: The dynamic programming matrix for calculating the edit distance between two given strings $s = ABBC$ and $t = ACB$ is shown in Figure 5.1a, where each insertion/deletion causes a cost of one and each substitution causes a cost of 2. It also shows the shortest path through the dynamic programming matrix that leads to the minimum edit distance of 3 in this case. The small number next to each arrow denotes the cost for a single edit operation (a diagonal move denotes keeping a single character, a downwards move will delete a single character from string s and a rightwards move will insert a single character to string t). Figure 5.1b shows the alignment corresponding to the edits.

minimize the edit distance between strings in the literature [NR03, WM21a]. In this chapter, we propose a novel global constraint that can be used to efficiently model and solve such problems, which require the repeated computation of the string edit distance, using CP.

A well known problem of this kind that has been extensively studied in the literature is the median string problem. An exact algorithm using dynamic programming for the median string problem has been proposed in [Kru83], however, this approach has a run-time complexity which is exponential on the instance size in the worst case. Therefore, several other techniques have been proposed in the literature to tackle practically sized instances e.g. [HK16, JABC03, OO08]. Although, most of these algorithms rely on approximations or heuristic techniques [JABC03, OO08] an exact approach using integer linear programming has been recently proposed in [HK16].

5.3 Propagating Lower Bounds on the Minimum Edit Distance

In this section, we propose a novel global constraint propagator that propagates lower bounds on the minimum edit distance between two strings that are represented by positive integer arrays. Such lower bounds on the minimum edit distance, that can be efficiently determined from a partial solution, can be especially useful when using CP solvers to tackle optimization problems that aim to minimize the edit distance.

Assuming two positive integer variable arrays X and Y of length n ($X = [x_1, \ldots, x_n]$, $[y_1, \ldots, y_n]$) and a positive integer variable ed, we introduce the edit distance constraint $ED(X, Y, ed)$ that will constrain ed to any value greater or equal to the minimum edit distance between X and Y ($ed \geq d(x, y)$). In the following we refer to the domain of any variable x as $D(x)$.

The constraint ED additionally constrains all values $x \in X$ and $y \in Y$ to be in the range $0..|\Sigma|$, where a value of zero represents the end of a string and a positive value c the c^{th} character in an alphabet Σ. We specifically allow the use of zero values so that the arrays X and Y can hold any string of length $\leq n$ including the empty string. For reasons of simplicity and to avoid symmetries, we further specify that the constraint ED (separately, by using a constraint decomposition into simple implication clauses on pairs of succeeding variables) enforces that whenever a variable x_i is set to 0, all variables $x_j, j > i$ have to be set to 0 as well, similarly for y_i.

Example 2: Let arrays $X = [1, 1, 2, 0, 0]$ and $Y = [1, 2, 2, 1, 1]$ represent two strings AAB and $ABBAA$. The ED constraint would then propagate $ed \geq 4$ and remove any values smaller or equal to 3 in $D(ed)$, as 4 is the minimum edit distance in this example. Another example array $X = [1, 0, 1, 2, 0]$ would violate the constraint independently of the values assigned to Y and ed, since the zero values of array X are not properly aligned at the end. □

We now propose an adaption of the standard dynamic programming routine to propagate lower bounds on the edit distance between two variable arrays. The idea is to build an "optimistic" dynamic programming matrix similar to the example shown in Figure 5.1, where we assume the best case for variables that are unfixed (in other words we will assume a zero cost diagonal move is possible when any character still appears in both corresponding variable domains). In addition to the standard dynamic programming routine previously defined in 5.1, we also have to include exceptional cases for insertion and removals of empty string characters as the variable arrays X and Y may contain less than n characters. Therefore, whenever a variable domain contains the value zero which denotes an empty string character, we will assume that the insertion or removal costs of an empty character will be 0, i.e. $\gamma(\epsilon \to 0) = \gamma(0 \to \epsilon) = 0$ and $\gamma(0 \to c) = \gamma(c \to 0) = 1, \forall c \in \Sigma$. Algorithm 8 describes the detailed propagation function and Figure 5.2 further shows how a dynamic programming matrix can be used to calculate a lower bound on the edit distance between two integer variable arrays.

Algorithm 8: Propagate Edit Distance Lower Bound

fn *PropagateEditDistance* (X, Y, ed)
- $d =$ CalculateDpMatrix (X, Y)
- $lb = d(length(X), length(Y))$
- $D(ed) = \{x | x \in D(ed) \wedge x \geq lb\}$

fn *CalculateDpMatrix* (X, Y)
- $n = length(X)$
- $m = length(Y)$
- $d(0,0) = 0$ ▷ d is a $(n+1 \times m+1)$ matrix
- **for** $j = 1$ *to* m **do**
  - $insCost = \min\{\gamma(\epsilon \to c) \mid c \in D(y_j)\}$
  - $d(0,j) = d(0,j-1) + insCost$
- **for** $i = 1$ *to* n **do**
  - $remCost = \min\{\gamma(c \to \epsilon) \mid c \in D(x_i)\}$
  - $d(i,0) = d(i-1,0) + remCost$
- **for** $i = 1$ *to* $n; j = 1$ *to* m **do**
  - $insCost = \min\{\gamma(\epsilon \to c) \mid c \in D(y_j)\}$
  - $remCost = \min\{\gamma(c \to \epsilon) \mid c \in D(x_i)\}$
  - $subCost = \min\{\gamma(c_x \to c_y) \mid c_x \in D(x_i) \setminus \{0\}, c_y \in D(y_j) \setminus \{0\}\}$
  - $$d(i,j) = \min \begin{cases} d(i,j-1) + insCost \\ d(i-1,j) + remCost \\ d(i-1,j-1) + subCost \end{cases}$$
- **return** d

The figure shows the dynamic programming matrix for calculating a lower bound for the edit distance between two given variable arrays $X = [x_1 = \{1\}, x_2 = \{2\}, x_3 = \{1\}, x_4 =$

	ϵ	$\{2,3\}$	$\{2,3\}$	$\{1\}$	$\{0,1\}$
ϵ	0	1	2	3	3
$\{1\}$	1	2	3	2	2
$\{2\}$	2	1	2	3	3
$\{1\}$	3	2	3	2	2
$\{3\}$	4	3	2	3	3

Figure 5.2: The dynamic programming matrix for calculating a lower bound for the edit distance between two given variable arrays $X = [x_1 = \{1\}, x_2 = \{2\}, x_3 = \{1\}, x_4 = \{3\}]$ and $Y = [y_1 = \{2,3\}, y_2 = \{2,3\}, y_3 = \{1\}, y_4 = \{0,1\}]$.

$\{3\}]$ and $Y = [y_1 = \{2,3\}, y_2 = \{2,3\}, y_3 = \{1\}, y_4 = \{0,1\}]$. Each row in the matrix represents a variable of array X while each column represents a variable of array Y (ϵ represents an empty string). In the first column/row of the matrix the domains of the corresponding variables are shown. All possible shortest paths through the matrix that lead to the lower bound for the edit distance of 3 in this case, are also shown on the figure as arrows. As not all variables are fixed, the best case (i.e. a possible match in characters, or a 0 cost insertion) is assumed several times in this example.

5.4 Explaining Propagation

In a LCG solver we need to provide an explanation clause which the solver can use to build an inference graph. When a conflict occurs during search, the solver can then find nogood constraints that are automatically created by analyzing the inference graph. In the following we will describe how inferences made by the edit distance propagator can be represented as an explanation clause.

Whenever a lower bound lb on the ed variable is propagated, essentially what we have to achieve is to enforce the corresponding Boolean variable $[\![ed \geq lb]\!]$ to be set to true. A correct explanation $expl$ therefore consists of a set of literals so that the following proposition is implied by the constraint (i.e. the proposition evaluates to true for any legal assignment to the variables):

$$\bigwedge_{l \in expl} l \rightarrow [\![ed \geq lb]\!] \tag{5.3}$$

Furthermore, an explanation is considered to be minimal, whenever it is not possible to remove any single literal l from $expl$ without invalidating Equation 5.3. In the following we show how a minimal explanation can be generated for inferences on the lower bound of the edit distance.

If we consider the example shown in Figure 5.2 we can see that in this case a lower bound of 3 has been determined for the edit distance and therefore the associated Boolean variable $[\![ed \geq 3]\!]$ would be set to true as a result of the propagation. To explain this inference, we can think about what could possibly be changed in the domains of variable arrays X and Y to allow an edit distance ≤ 2, and then negate such changes in our explanation.

If we look at the possible shortest paths in Figure 5.2, we can observe that we need to reduce the cost of at least one of the edges towards the end of the matrix that have a cost ≥ 1. For example if x_4 and y_4 would allow the same value assignment in their domain we could take another diagonal 0 cost move and improve the lower bound to 2. More generally speaking, improvements to the edit distance can be achieved if moves are possible that reduce the length of the shortest path through the matrix.

The algorithm for generating a minimal explanation is shown in Algorithm 9. It works backwards over the matrix of (i, j) values starting from (n, m) collecting the constraints C which must hold to ensure the lower bound lb. It stores in s the minimal edit cost from a position to reach (n, m) under the current set of constraint C in the explanation. The algorithm is based on a priority queue (heap) which stores node positions that are reachable under the current assumptions. We take the (lexicographically) largest node position (i, j) off the heap, and then consider what characters c in the domain of y_j would allow a smaller lower bound via a path from $(i, j-1)$. We add a constraint $y_j \neq c$ preventing this. The remaining characters are used to update the cost s for position $(i, j-1)$, and it is pushed onto the heap. Note if a node is pushed multiple times it only appears once on the heap. Afterwards, we consider paths via $(i-1, j)$ similarly. We only consider 0 edit operations $(\gamma(\epsilon \to 0), \gamma(0 \to \epsilon))$ as long as no constraint $[\![x_i \neq 0]\!], [\![y_i \neq 0]\!]$ has been added to C since the rules of correct string representation would not allow any additional 0 operations then. Finally, we consider paths via $(i-1, j-1)$. Here when a substitution $(c_x \to c_y)$ would lead to a path which is shorter than lb we can enforce $x_i \neq c_x$ or $y_j \neq c_y$. In practice, we make choices dependent on the current domains of x_i and y_j. If x_i is fixed and $D(x_i) = \{c_x\}$ then we choose the restriction on y_j and if $D(x_i) = \{c\}, c \neq c_x$ we choose the restriction on x_i. Similarly if y_j is fixed. If only one of the disequations holds in the current domain we choose that (note that it is impossible that both do not hold, otherwise lb would not be the lower bound). In the remaining cases we can choose arbitrarily.

The result of the code is to return C such that $C \to [\![ed \geq lb]\!]$. Note that we can simplify C by replacing a set of disequations $[\![x_i \neq c]\!], 0 \leq c < l$ by $[\![x_i \geq l]\!]$ and similarly a set of disequations $[\![x_i \neq c]\!], u < c < |\Sigma|$ by $[\![x_i \leq u]\!]$.

Example 3: Consider generating the explanation for the case shown in Figure 2. We

Algorithm 9: Generate disequalities for minimal explanation

```
fn GenerateExpl (X, Y, d, lb)
    n = length(X)
    m = length(Y)
    for i = 0 to n; j = 0 to m do
        s(i, j) = lb + 1
    s(n, m) = 0
    C = {}
    H = []
    H.push(n, m)
    while H is not empty do
        (i, j) = H.popMax()
        if s(i, j) ≥ lb then continue
        if j − 1 ≥ 0 then
            T = Σ ∪ {0 | ¬∃x_k ≠ 0 ∈ C, k ≥ j}
            for c ∈ T do
                if d(i, j − 1) + γ(ϵ → c) + s(i, j) < lb then
                    C.add([[y_j ≠ c]])
                else
                    s(i, j − 1) = min(s(i, j − 1), s(i, j) + γ(ϵ → c))
                    H.push(i, j − 1)
        if i − 1 ≥ 0 then
            T = Σ ∪ {0 | ¬∃x_k ≠ 0 ∈ C, k ≥ i}
            for c ∈ T do
                if d(i − 1, j) + γ(c → ϵ) + s(i, j) < lb then
                    C.add([[x_i ≠ c]])
                else
                    s(i − 1, j) = min(s(i − 1, j), s(i, j) + γ(ϵ → c))
                    H.push(i − 1, j)
        if i − 1 ≥ 0 ∧ j − 1 ≥ 0 then
            for c_x ∈ Σ, c_y ∈ Σ do
                if d(i − 1, j − 1) + γ(c_x → c_y) + s(i, j) < lb then
                    C.add([[x_i ≠ c_x]]) OR C.add([[y_j ≠ c_y]])
                else
                    s(i − 1, j − 1) = min(s(i − 1, j − 1), s(i, j) + γ(c_x → c_y))
                    H.push(i − 1, j − 1)
    return C
```

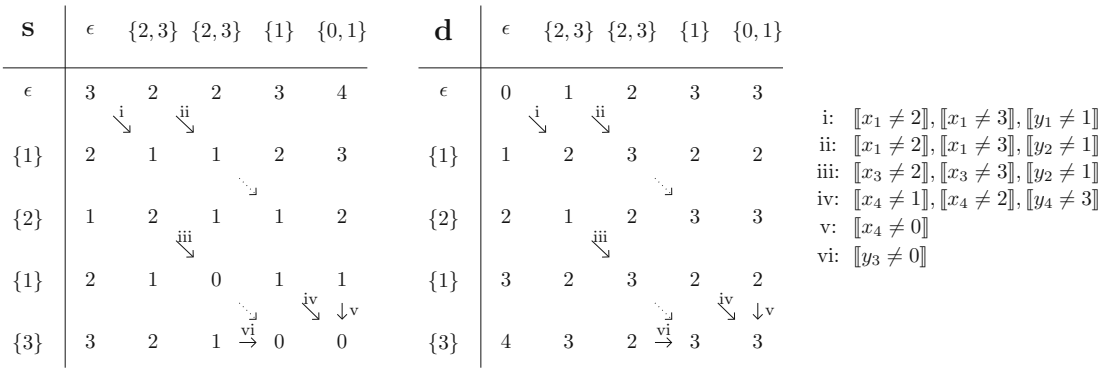

Figure 5.3: The matrices s and d at the end of the explanation algorithm (Algorithm 9) together with six determined disequalities for example variable arrays $X = [x_1 = \{1\}, x_2 = \{2\}, x_3 = \{1\}, x_4 = \{3\}]$ and $Y = [y_1 = \{2,3\}, y_2 = \{2,3\}, y_3 = \{1\}, y_4 = \{0,1\}]$.

start by setting $s(i,j) = lb + 1 = 4$ everywhere, then resetting $s(4,4) = 0$ and pushing (4,4). We pop off $(4,4)$. Since $d(4,3) = 3$ we cannot get a path of length ≤ 2 via it. We set $s(4,3) = 0$ (for the case that $y_4 = 0$ and push $(4,3)$). Since $d(3,4) = 2$ we can get a path of length ≤ 2 via it if $x_4 = 0$, so we add $[\![x_4 \neq 0]\!]$ to C. We set $s(3,4) = 1$ since all other deletions cost 1 and push $(3,4)$. Since $d(3,3) = 2$ we can get paths of length ≤ 2 via this position if the characters for x_4 and y_4 are the same. We need to add one of each pair $[\![x_4 \neq c]\!]$ or $[\![y_4 \neq c]\!]$ for all $c \in 1..3$. We choose $[\![x_4 \neq 1]\!]$, $[\![y_4 \neq 3]\!]$, because of the values in the current domains of x_4 and y_4. The remaining choice is arbitrary: say we add $[\![x_4 \neq 2]\!]$ to C. We set $s(3,3) = 2$ and push $(3,3)$. We pop off (4,3). Since $d(3,3) = 2$ we could get a path of length ≤ 2 if $x_4 = 0$, but we already inserted a constraint of the form $[\![x_4 \neq 0]\!]$, so we do not consider 0 edit operations in that direction any longer and do not change the cost to of $s(3,3)$. Since $d(3,2) = 3$ there is no path less than lb possible, we set $s(3,2) = 0$ and push it. Since $d(4,2) = 2$ we can get a path of length ≤ 2 via it if $y_3 = 0$, so we add $[\![y_3 \neq 0]\!]$ to C and push (4,2) as other insertions all cost 1. We pop off $(4,2)$ and its treatment is similar, followed by $(4,1)$ and $(4,0)$. Next we pop $(3,4)$ and will set $s(2,4) = 2$ and push $(2,4)$ as we cannot directly improve the lb. Similarly, we set $s(2,3) = 1$ and $s(3,3)$ will have its assigned value changed to 1, since we can reach (4,4) quicker via (3,4). The process continues eventually collecting $C = \{[\![x_1 \neq 2]\!], [\![x_1 \neq 3]\!], [\![x_3 \neq 2]\!], [\![x_3 \neq 3]\!], [\![x_4 \neq 0]\!], [\![x_4 \neq 1]\!], [\![x_4 \neq 2]\!], [\![y_1 \neq 1]\!], [\![y_2 \neq 1]\!], [\![y_3 \neq 0]\!], [\![y_4 \neq 3]\!]\}$. It can be simplified to $\{[\![x_1 \leq 1]\!], [\![x_3 \leq 1]\!], [\![x_4 \geq 3]\!], [\![y_1 \geq 2]\!], [\![y_2 \geq 2]\!], [\![y_3 \geq 1]\!], [\![y_4 \leq 2]\!]\}$. □

Figure 5.3 visualizes matrix s after generating the explanation for Example 3 and summarizes which constraints have been produced.

The matrix on the left of Figure 5.3 visualizes the contents of matrix s at the end of the explanation algorithm (Algorithm 9) that is called for two variable arrays $X = [x_1 = \{1\}, x_2 = \{2\}, x_3 = \{1\}, x_4 = \{3\}]$ and $Y = [y_1 = \{2,3\}, y_2 = \{2,3\}, y_3 = \{1\}, y_4 = \{0,1\}]$, an edit distance lower-bound of 3 and the matrix d which is shown in the middle.

Additionally, six sets of disequalities that are determined by the explanation algorithm are listed on the right. For each set of generated disequalities the corresponding edit operation is also visualized in the two matrices by solid arrows highlighting the operations that could lower the edit distance bound if their cost would be reduced. The dotted arrows on the other hand indicate operations that also would need to be lowered together with operations (ii or iii), to reach a reduced lower-bound. However, since the explanation algorithm aims to minimize the number of generated explanation clauses and it is sufficient to only produce disequalities for the operations that actually cause a shortest path lower than the current lower-bound in Algorithm 9, no sets of disequalities are inserted for the dotted arrows.

We can argue that the explanation produced by GenerateExpl is minimal since the only time we add constraints to C is when otherwise there would be a path to (n, m) of length less than lb. Removing any explanation would cause such a path to exist, thus making the explanation incorrect, hence it is minimal.

5.5 Experimental Evaluation

We implemented the constraint propagation and explanation algorithms proposed in this chapter for use with version 0.10.3 of the lazy clause generation solver Chuffed [Chu11]. Afterwards, we evaluated our constraint propagator on two NP-hard problems that utilize the edit distance constraint described in this chapter.

All of our experiments have been conducted on an Intel Xeon E5345 2.33 GHz CPU with 48 GB RAM, using a single CPU core.

5.5.1 Paint Shop Scheduling

The evaluation of setup costs for paint shop scheduling we introduced in Chapter 2 requires calculating the edit distance between two consecutive cycles, as the required change in production utilities corresponds to the minimum edit operations between the two scheduling sequences. The paint shop scheduling problem defines the following edit operation costs: $\gamma(\epsilon \to c) = \gamma(c \to \epsilon) = 1$ forall $c \in \Sigma$ and $\gamma(c_1 \to c_2) = 2$ if $c_1 \neq c_2$ forall $\{c_1, c_2\} \subseteq \Sigma$.

For our experiments we used the 12 smaller benchmarks instances that we previously introduced in Chapter 2. We used the CP model from Chapter 3 and replaced the edit distance constraint decomposition with the global constraint propagator proposed in this chapter. For each of the instances we used the same programmed search strategies that were used for the final experiments in Chapter 3. Afterwards, we ran the Chuffed solver with both the existing decomposition model and a model that uses the propagator proposed in this chapter on all 12 instances within a time limit of 1 hour. The results of these experiments are shown in Table 5.1. We see in the table that the model using the global constraint produces equally good or improved results for all the benchmark instances compared to the results produced by the previously proposed constraint decomposition.

	CP		CP+global	
Instance	Cost	Runtime	Cost	Runtime
I1	**775***	9.95	**775***	**3.63**
I2	**842***	1.30	**842***	**0.57**
I3	**961***	3.76	**961***	**1.75**
I4	**918***	178.46	**918***	**25.05**
I5	**530***	126.64	**530***	**51.03**
I6	**842***	9.76	**842***	**5.31**
I7	1046	∞	**1040**	∞
I8	**1237***	2915.83	**1237***	**445.43**
I9	1006	∞	**992**	∞
I10	973	∞	**966**	∞
I11	—	∞	—	∞
I12	—	∞	—	∞

Table 5.1: Results for the experiments conducted on benchmark instances 1–12 for the paint shop scheduling problem. Columns 2 and 3 show the best objective value achieved within one hour as well as the run-time needed to prove an optimal solution in seconds for the CP model from Chapter 3 (CP). Similarly, Columns 4 and 5 show the results achieved with the CP model that uses the global constraint propagator proposed in this chapter instead of the constraint decomposition for the edit distance constraint (CP+global). The best result within each line is formatted in boldface and results marked with a * denote proven optimal solutions. ∞ represent a timeout (1 hour), while — means that no solution at all could be found within the time limit.

Both models are able to prove optimality for 7 of the 12 instances and can produce 3 upper-bounds within 1 hour, however all the upper bounds produced with the global constraint propagator are improved compared to the upper bounds achieved with the decomposition. When we compare run-times for instances where both solvers could prove optimality, we can clearly see that the global propagator requires less run-time to find optimal solutions. Both methods are not able to produce any solutions for the two largest instances (I11 and I12) within one hour.

To investigate the effect of the global propagator without lazy clause learning, we further repeated all experiments with the paint shop scheduling instances without clause learning (we set the *nolearn* parameter for chuffed). The results produced without the explanation algorithm are shown in Table 5.2. Without clause learning, the constraint propagator produced improved results compared to the constraint decomposition for four instances and could further reduce the required runtime to prove optimality for two instances. The results indicate the effectiveness of the constraint propagator even without lazy clause generation.

	CP nolearn		CP+Global nolearn	
Instance	Cost	Runtime	Cost	Runtime
I1	3282	∞	**775***	**1617.15**
I2	**842***	13.24	**842***	**0.57**
I3	**961***	519.41	**961***	**1.76**
I4	—	∞	**918***	**79.77**
I5	—	∞	—	∞
I6	17234	∞	**842***	**6.25**
I7	—	∞	—	∞
I8	—	∞	—	∞
I9	—	∞	—	∞
I10	—	∞	**973**	∞
I11	—	∞	—	∞
I12	—	∞	—	∞

Table 5.2: Results for the experiments conducted on benchmark instances 1–12 for the paint shop scheduling problem without clause learning. Columns 2 and 3 show the best objective value achieved within one hour as well as the run-time needed to prove an optimal solution in seconds for the CP model from Chapter 3 (CP nolearn). Similarly, Columns 4 and 5 show the results achieved with the CP model that uses the global constraint propagator proposed in this chapter instead of the constraint decomposition for the edit distance constraint (CP+global nolearn).

5.5.2 The Median String Problem

To evaluate our constraint propagator we also consider the median string problem, which has been thoroughly studied in the literature (e.g. [Koh85, HK16]).

The median string problem is formulated as follows: Given a set of n strings S (all strings of length $\leq k$) over a finite alphabet Σ, find a string that minimizes the global edit distance to each of the given strings.

The *global edit distance* $D(s, S)$ between a string s and a set of strings S over the finite alphabet Σ is defined as follows:

$$D(s, S) = \sum_{s' \in S} d(s, s') \tag{5.4}$$

Then a median string is defined as any string m over Σ where $D(m, S) \leq D(w, S)$ holds for any string w over the alphabet Σ.

We generated many instances following the instance generation procedure that has been proposed in [HK16] to evaluate the performance of different exact solution approaches: Our instance generation routine considered different numbers of strings $n = [2, 4, 6, 10, 15]$ as well as different maximum string lengths $k = [3, 5, 8, 13, 20]$. We used a simple alphabet consisting of 4 different characters $\Sigma = \{1, 2, 3, 4\}$ to randomly generate 10

different instances for each of the possible $|n \times k|$ configurations, totaling 250 benchmark instances. To randomly generate 10 different instances per configuration we implemented two procedures: Five of the instances are generated by randomly assigning the letters s^i_j at positions $j \in \{1, \ldots, k\}$ for each string $i \in \{1, \ldots, n\}$. Each letter is assigned to $s^i_j = \min(1 + \lfloor |\alpha| \rfloor, |\Sigma|)$, where α follows a normal distribution with mean 0 and variance 1. The other five instances of each configuration are generated by performing 100 random single character edit operations on an initial string of length k that initially contains only the first letter of the alphabet Σ. In each of the 100 edit iterations we randomly select a feasible single character insertion, single character removal or single character substitution.

We compare the performance of our edit distance constraint propagator (CP+global) on the median string problem with an existing MIP formulation from [HK16] as well as a CP model that uses the same edit distance constraint decomposition as for the paint shop scheduling to solve the median string problem. In our experiments we use the edit operations costs that are known as *levensthein distance*, as these costs have often been used for median string problems e.g. [HK16]: $\gamma(\epsilon \to c) = \gamma(c \to \epsilon) = 1$ forall $c \in \Sigma$ and $\gamma(c_1 \to c_2) = 1$ if $c_1 \neq c_2$ forall $\{c_1, c_2\} \subseteq \Sigma$.

We performed experiments with all the 250 benchmark instances under a time limit of 10 minutes, using a recent version of the Chuffed solver [Chu11] for the CP model and the CP model that uses our propagator, as well as Gurobi 8.0.1 [GO20] for experiments with the MIP model that was previously proposed in [HK16].

Table 5.3 and Figures 5.4 and 5.5 summarize the results of our experiments with the median string benchmarks.

Looking at the results shown in Table 5.3, we can see that the CP model that uses the global edit distance propagator produces the largest number of best found solutions as well as the largest number of optimal solutions found. Furthermore, the approach can prove optimality faster than the MIP model and the CP model without the global propagator for all instances that can be solved to optimality. When comparing the decomposition based CP model with the MIP model, the results show that the MIP model requires less runtime to prove optimality and can find better solutions for a larger number of instances.

Figure 5.4 compares the quality of solutions where not all of the three considered approaches produced an equal result within 10 minutes. The results show, that except for a few outliers the approach using the constraint propagator always produced the best results, while the MIP model seems to overall produce better results regarding solution quality than the CP model which uses a constraint decomposition.

Figure 5.5 compares the run-time required to prove optimality for those instances where all three approaches could prove optimality within 10 minutes. The box plots show that the approach using the propagator always proved optimality within the shortest run-time in our experiments. Furthermore, it seems that the decomposition based CP model

	MIP	CP	CP+global
# opt	208	180	**227**
# proven opt	197	169	**227**
# best	213	180	**246**
# fastest proof	0	0	**227**
avg time	143.16	219.12	**61.61**
std dev time	243.37	277.52	**175.66**

Table 5.3: Summarized experimental results for the median string problem. Column 2 displays results achieved with the MIP formulation from [HK16], while column 3 displays results achieved with the edit distance constraint decomposition from Chapter 3. Column 4 shows results for the global constraint propagator proposed in this chapter. Line 1 shows the number of optimal solutions found, line 2 shows for how many instances optimality could be proven within the time limit, and line 3 shows the number of solutions that have the overall best found objective value. Line 4 displays for how many instances the method could provide the fastest optimality proof and lines 5 and 6 show the average required runtime, as well as the standard deviation of all required runtimes in the experiments.

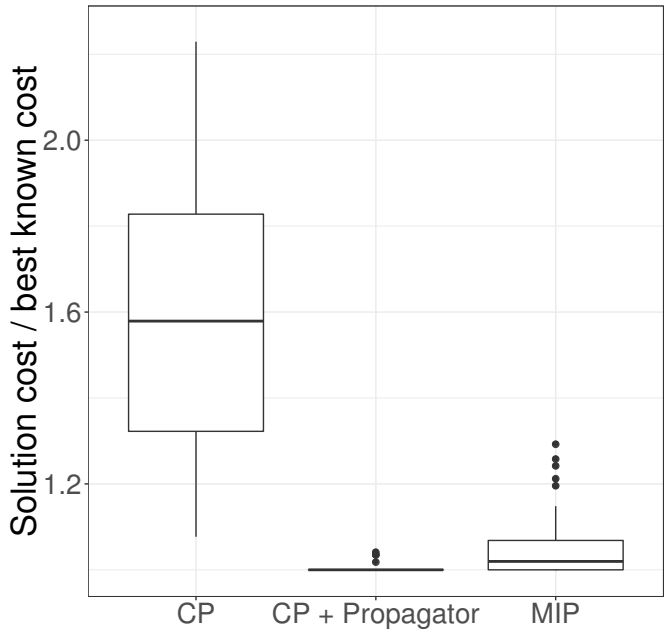

Figure 5.4: Box plot displaying the differences in quality of solutions for median string experiments, omitting 70 instances where all three approaches gave equal results. The vertical axis represents the relative objective value (objective value of solution divided by best found objective). Except for three outliers, the best solution cost was produced with the global propagator. Some outliers with values higher than 2.5 for the CP approach without the propagator have been omitted for a better visualization.

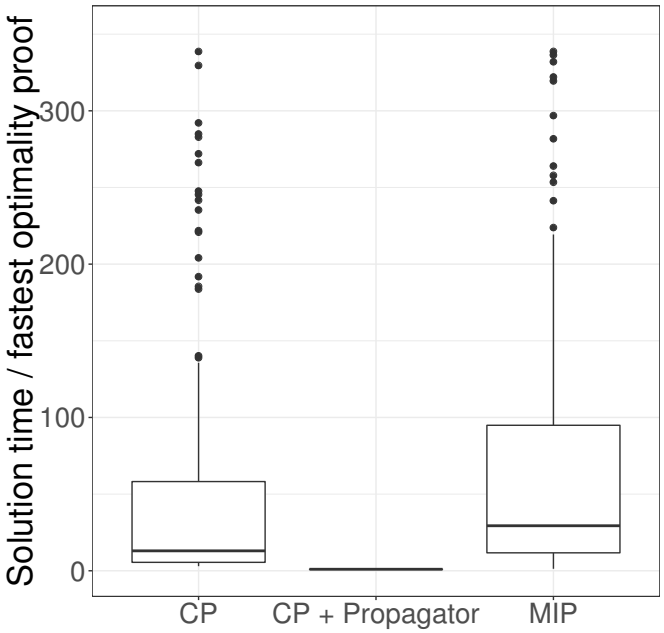

Figure 5.5: Box plot comparing the run-times for the 178 instances where all methods proved optimality. The vertical axis represents the required run-time divided by the overall shortest optimality proof time for each instance. The fastest optimality proof was in all cases achieved with the CP model that uses the constraint propagator proposed in this chapter. Some outliers with values higher than 300 for the CP approach without the propagator and the MIP approach have been omitted for a better visualization.

performs similarly to the MIP model when it comes to proving optimality, although the majority of instances have a shorter run-time with the CP model.

CHAPTER 6

The Artificial Teeth Scheduling Problem

In this chapter we introduce a novel single-machine batch scheduling problem arising from teeth manufacturing which we further call the artificial teeth scheduling problem (ATSP). We first present some background together with an informal description of the problem before we provide a detailed formal problem specification. Afterwards, we give an overview on related literature from the recent past and clarify what distinguishes the artificial teeth scheduling problem from existing single-machine batch scheduling problems. Finally, we introduce a set of benchmark instances that includes real-life problem scenarios from the industry.

6.1 Problem Description

Modern-day artificial teeth manufacturing uses an automated production process to produce large quantities of teeth in a variety of different shapes and colors. To efficiently handle a large-scale production, product moulds are usually grouped in batches which are then simultaneously processed on a single machine. However, due to resource constraints and the requirement that different machine programs have to be used depending on varying product families, creating cost-efficient batches becomes a very challenging task.

To efficiently produce a large number of artificial teeth, many teeth are usually processed simultaneously in batches. Therefore, each job in a production schedule uses a number of different product moulds to produce teeth. Such a mould essentially produces a certain tooth shape and is associated to a product line so that all moulds that belong to the same line form a family of related shapes. However, a job in the schedule needs to additionally decide which color should be applied to each of the produced teeth and therefore the final tooth product type is determined not only by its product line but also by the applied color.

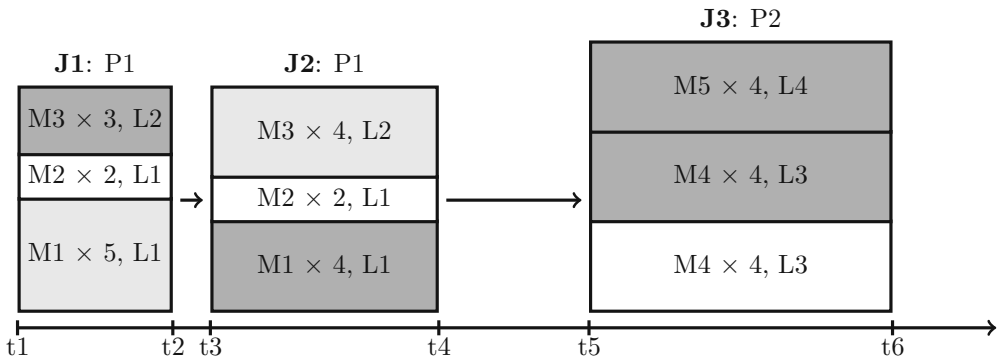

Figure 6.1: A small example schedule for the ATSP.

Each job is further configured by a length- and production program parameter. The length parameter sets the number of production cycles of the job that determines the total number of produced teeth. Note that each cycle produces the same teeth using the moulds which are assigned to the job. The production program parameter determines how many moulds are simultaneously processed by the job, which mould types are compatible, and the processing time of a single production cycle. As every production program requires a fixed amount of moulds to be processed per cycle, it might be necessary to produce more teeth than necessary in some job cycles. Usually this cannot be completely avoided, therefore one of the problem's goals is to minimize the amount of waste caused by excessively produced teeth. Consecutively scheduled jobs may either use different production programs or share the same program with a different set of mould and or color assignments. In any case a setup time is required between jobs, however if different production programs are used a longer setup time is required.

Finally, the main goal of the ATSP is to create a schedule that fulfills all given customer demands by creating jobs in a way that the makespan, total tardiness, and produced waste is minimized. Figure 6.1 further illustrates the problem, by visualizing a schedule with three jobs for a small example instance.

The figure shows three jobs J1, J2, and J3 being scheduled on the horizontal time line. Time points t1, t3, and t5 indicate the starting times of each job, whereas timepoints t2, t4, and t6 denote the corresponding job end times (in this case the total makespan is t6-t1). Jobs J1 and J2 both use the production program P1, whereas job J3 uses a different program P2. Note that the setup time between jobs J1 and J2 (visualized by the lengths of the horizontal arrows between jobs) is much smaller than it is between J2 and J3, as J1 and J2 both use program P1, but J2 and J3 use different programs. Furthermore, the horizontal length of the jobs indicates the number of assigned production cycles. Therefore, J2 uses more cycles than J1.

As the production program defines the total number of assigned moulds, we can see in Figure 6.1 that J1 and J2 both use a total of 10 moulds, whereas J3 uses a total of 12 moulds. Mould types M1, M2, and M3 are in this case compatible only with program

P1, and mould types M4 and M5 are associated to P2. We can further see in the figure, that each mould type is associated to a certain product line (e.g. M3 corresponds to line L2), and that the same mould type may be used with different colorings within the same job (e.g. in J3 mould type M4 is used in white color and gray color). Note that any two colors may only be used within the same job if they are compatible. Which pairs of colors are compatible is specified as part of the problem's input.

6.2 Formal Specification

We now provide the full formal specification of the ATSP in the following sections. For simplicity, we make use of the Iverson bracket notation[1].

6.2.1 Input Parameters

The following parameters describe instances of the problem:

Description	Parameter
Set of colors	C
Set of programs	P
Set of mould types	M
Set of product lines	L
Set of demands	D
Setup time between identical programs	$sj \in \mathbb{N}$
Setup time between different programs	$sp \in \mathbb{N}$
Max product types per job	$w \in \mathbb{N}_{>0}$
Min cycles per job	$c_{\min} \in \mathbb{N}_{>0}$
Max cycles per job	$c_{\max} \in \mathbb{N}_{>0}$
Number of available moulds per type	$a_m \in \mathbb{N}\ \forall m \in M$
Number of mould slots per program	$am_p \in \mathbb{N}\ \forall p \in P$
Cycle time per program	$t_p \in \mathbb{N}_{>0}\ \forall p \in P$
Admissible program per mould type	$p_m \in P\ \forall m \in M$
Product line of each mould type	$l_m \in L\ \forall m \in M$
Requested mould type per demand	$dm_d \in M\ \forall d \in D$
Requested mould quantity per demand	$dq_d \in \mathbb{N}_{>0}\ \forall d \in D$
Due date of each demand	$dd_d \in \mathbb{N}\ \forall d \in D$
Requested color for each demand	$dc_d \in C\ \forall d \in D$
Set of compatible colors per color	$comp_c \in 2^C\ \forall c \in C$

Table 6.1: Input parameters of the ATSP

[1] $[P] = 1$, if $P = true$ and $[P] = 0$ if $P = false$

6.2.2 Variables

We define the following variables for the ATSP:

- Number of assigned jobs: $j \in \mathbb{N} \quad J = \{1, \ldots, j\}$
- Program assigned to each job: $jp_i \in P \quad \forall i \in J$
- Length of each job (i.e. the number of cycles):

$$jl_i \in \mathbb{N}_{>0} \quad \forall i \in J$$

- The number of mould types (with color) assigned to each job:

$$jm_{i,m,c} \in \mathbb{N} \quad \forall i \in J, m \in M, c \in C$$

- The total number of mould types (with color) produced by each job:

$$totaljm_{i,m,c} = jm_{i,m,c} \cdot jl_i \quad \forall i \in J, m \in M, c \in C$$

6.2.3 Constraints

Several constraints impose restrictions on feasible schedules:

- The number of moulds assigned to each job must be equal to the number of mould slots of the job's program:

$$\sum_{m \in M} \sum_{c \in C} jm_{i,m,c} = am_{(jp_i)} \quad \forall i \in J$$

- The number of scheduled moulds per job must not exceed mould availability:

$$\sum_{c \in C} jm_{i,m,c} \leq a_m \quad \forall i \in J, m \in M$$

- The number of different product types within a single job must be less than or equal to the allowed maximum:

$$\sum_{c \in C} \sum_{l \in L} \sum_{m \in M} ([l_m = l] \cdot [jm_{i,m,c} > 0]) \leq w \quad \forall i \in J$$

- All demands need to be fulfilled:

$$\sum_{d \in D} [dm_d = m \wedge dc_d = c] \cdot dq_d \leq \sum_{i \in J} totaljm_{i,m,c} \; \forall m \in M, c \in C$$

- Job moulds must be compatible with the job's program:

$$\sum_{c \in C} jm_{i,m,c} \cdot [jp_i \neq p_m] = 0 \quad \forall i \in J, m \in M$$

- A single job must not use incompatible colors:

$$\left[\sum_{m \in M} jm_{i,m,c_1} > 0\right] \leq \left[\sum_{m \in M} jm_{i,m,c_2} = 0\right]$$
$$\forall i \in J, c_1 \in C, c_2 \in (C \setminus comp_{c_1})$$

6.2.4 Objective Function

For the formal definition of the objective function we introduce the following auxiliary variables:

- The processing time for each job: $jt_i \in \mathbb{N}_{>0} \quad \forall i \in J$
- The finishing time for each job: $je_i \in \mathbb{N}_{>0} \quad \forall i \in J$
- The finishing job for each demand (after completion the demand is fulfilled): $dj_d \in J \quad \forall d \in D$

Several constraints set the values of the auxiliary variables:

- Set the job processing times: $jt_i = jl_i \cdot t_{(jp_i)} \quad \forall i \in J$
- Set job finishing times:

$$je_i = jt_1 + \sum_{k=2}^{i} (jt_k + sj + [jp_k \neq jp_{k-1}] \cdot (sp - sj)) \; \forall i \in J$$

- For each demand, a constraint ensures that the corresponding demand finishing job auxiliary variable is set to a feasible value:

$$\sum_{i=1}^{dj_d} totaljm_{i,m,c} \geq \sum_{d' \in D'} dq_{d'} \quad \forall d \in D \text{ where } m = dm_d,$$
$$c = dc_d, D' = \{d' \in D | dd_{d'} \leq dd_d \wedge dm_{d'} = m \wedge dc_{d'} = c\}$$

  The left hand side of the constraint sums up the total number of moulds that have the correct type and color for the associated demand and are produced by all jobs that are scheduled before the end time of the demand finishing job (which includes the demand finishing job itself). This sum must be greater or equal to the total quantity of all demands that require the same mould type and color and have a due date that is smaller or equal to the demand which corresponds to the demand of the finishing job variable (this total quantity is specified on the right hand side of the constraint).

Using these auxiliary variables, the objective function aims to minimize three solution objectives:

1. The last job should be finished early to minimize the total **makespan** of the schedule: $ms = je_j$
2. The number of excess moulds which are not consumed by any demand are considered to be **waste** and should be minimized:

$$waste = \sum_{i \in J} \sum_{m \in M} \sum_{c \in C} totaljm_{i,m,c} - \sum_{d \in D} dq_d$$

3. The total **tardiness** of all demands in the schedule should be mimized:

$$tard = \sum_{d \in D} \max(0, je_{(dj_d)} - dd_d)$$

Finally, we aggregate all three objectives in a normalized weighted sum where the objectives marked with * denote the costs of a given reference solution and w_{1-3} are weight parameters:

$$minimize \quad \frac{w_1 \cdot ms}{ms*} + \frac{w_2 \cdot waste}{waste*} + \frac{w_3 \cdot tard}{tard*}$$

Parameters w_{1-3} are then used to configure the relative importance of the three individual objectives. In practice, the weight parameters and reference solution costs can be configured according to the practical use case.

6.3 Related Literature

A variety of batch scheduling problems, which share the goal to efficiently schedule batches of jobs onto machines, have been the subject of intensive study in the past. In [PK00] an overview and categorization of earlier NP-hard batch scheduling variants for several single machine and parallel machine environments was given. However, although the basic problem variants have been extensively studied in the past, there is still the need to investigate innovative solution methods for challenging real-life batch scheduling problems, due to the large variety of constraints and optimization objectives that arise from different application domains.

For example, a recent publication [PTM20] studied a just-in-time batch scheduling problem that aims to minimize tardiness and earliness objectives and was shown to be NP-hard in [HKS14]. Another practical single machine scheduling problem from the steel industry has recently been investigated in [ZLZ+20]. The problem considers sequence-dependent setup times, release time as well as due time constraints where batches of jobs are predetermined in advance.

Inst.	C	M	D	L	P	Vars	CS
I 1	5	38	20	4	2	12649	14243
I 2	4	28	24	3	1	15475	17442
I 3	4	16	7	1	1	3242	4045
I 4	5	38	4	4	2	5849	6048
I 5	4	28	9	3	1	6840	7646
I 6	3	16	1	1	1	1247	1447
I 7	22	153	799	4	2	621749	583209
I 8	18	114	390	3	1	372409	337268
I 9	18	64	285	1	1	135664	149647
I 10	22	153	190	4	2	373599	302936
I 11	18	114	224	3	1	294675	253199
I 12	13	64	36	1	1	50463	45298

Table 6.2: Size parameters of the used benchmark instances.

Further NP-hard single machine scheduling problem variants that do not include batching decisions but consider similar objectives as in artificial teeth manufacturing such as tardiness and setup time minimization, have been recently investigated for example in [NSD+19] and in [dWBH20].

In this thesis we introduce the artificial teeth scheduling problem (ATSP), which is a novel single machine batch scheduling variant that appears in real-life production plants of the artificial teeth manufacturing area. While previous single machine batch scheduling problem variants are given a predetermined set of jobs as an input and aim to efficiently group these jobs into batches, instances of the ATSP include customer demands but do not specify any job information. Therefore, solutions to the ATSP do not only need to design efficient batches, but are further required to create jobs that efficiently fulfill all customer demands. Furthermore, solution methods to the ATSP have to consider several constraints which impose restrictions on feasible schedules as well as an innovative cost objective that aims to minimize waste caused by overproduction.

6.4 Benchmark Instances

We received 6 problem instances from industry partners that represent real-life scheduling scenarios as they appeared at production sites of artificial teeth manufacturing. Early experiments with these instances showed that all of them have a very large search space, which makes it hard for exact methods to reach results within reasonable runtime. Therefore, we additionally generated 6 smaller instances by randomly selecting 25% of the colors and mould types together with associated demands for each of the realistic instances. Table 6.2 displays size parameters of all 12 benchmark instances, where instances I1–I6 form the small instance set and instances I7–I12 are the large real-life scheduling scenarios.

Columns 2–6 from Table 6.2 provide information about the number of colors, mould

types, demands, production lines and programs. Furthermore, columns 7 and 8 display the number of generated variables using the bilinear CP model that we introduce later in Section 7.1.

CHAPTER 7

Constraint Modeling and Heuristic Solution Methods for the Artificial Teeth Scheduling Problem

In this chapter, we first propose a CP formulation that we utilize as an exact approach for the ATSP. Afterwards, we introduce an innovative construction heuristic as well as a metaheuristic based on local search to quickly solve large-scale realistic instances that cannot be efficiently solved using the proposed exact techniques. At the end of the chapter, we evaluate all proposed solution methods using the set of benchmark instances that we introduced in the previous chapter.

7.1 Constraint Programming Approach

In this section, we provide a CP formulation for the ATSP using the input parameters that were given in Table 6.1 of the previous chapter.

7.1.1 Model Variables

The model we propose uses several arrays of decision variables, where the length of many arrays is dependent on the maximum number of jobs that can be scheduled. The problem instances do not set any restrictions on the number of jobs, however an arbitrary number of jobs can lead to an unnecessary blow up of the variables in the model. Therefore, we set the maximum number of possible jobs based on a user defined model parameter max_jobs and in the following refer to the set of possible job IDs as $J = \{1, \ldots, max\_jobs\}$. In practice the construction heuristic can be used to find reasonable values for the max_jobs parameter by simply taking the number of heuristically constructed jobs or increasing the number by a low value. We define the following variables:

- $jp_i \in \{0, \ldots, |P|\} \quad \forall i \in J$
- $jl_i \in \{c_{\min}, \ldots, c_{\max}\} \quad \forall i \in J$
- $jm_{i,m,c} \in \{0, \ldots, \max\{am_p | p \in P\}\} \quad \forall i \in J, m \in M, c \in C$
- $tjm_{i,m,c} \in \{0, \ldots, \max\{am_p \cdot c_{\max} | p \in P\}\}$
  $\forall i \in J, m \in M, c \in C$

Variable arrays jp and jl determine the selected program, as well as the number of selected cycles for each job. Note that the domain of jp includes 0 to indicate that a job variable should be ignored, allowing the formulation to use less than the maximum number of jobs. Variable arrays jm and tjm determine which moulds and colors are assigned to each job. The upper bound of these variable domains is calculated by the maximum number of possible program slots.

In addition to the mentioned variables, the model we propose uses a set of auxiliary variable arrays that are used in the formulation of the cost objectives. To efficiently set the domains of these auxiliary variables, we calculate several lower and upper bounds based on the input parameters:

- $lb\_end = c_{\min} \cdot \min\{t_p | p \in P\}$
- $ub\_end = max\_jobs \cdot c_{\max} \cdot \max\{t_p | p \in P\} + max\_jobs \cdot sp$
- $ub\_time = \max\{t_p | p \in P\} \cdot c_{\max}$
- $ub\_waste = \max\{am_p | p \in P\} \cdot c_{\max} \cdot max\_jobs$
- $ub\_tardiness = \sum_{d \in D} \max\{0, ub\_end - dd_d\}$

lb_end and ub_end define lower and upper bounds on the job end times in the schedule based on the minimum and maximum values regarding the number of cycles and cycle processing times. ub_time defines a bound on the maximum job processing time, whereas ub_waste and $ub\_tardiness$ provide upper bounds on the total waste and tardiness costs based on the maximum number of scheduled moulds and the maximum job end time. We then define auxiliary variables:

- $je_i \in \{lb\_end, \ldots, ub\_end\} \quad \forall i \in J$
- $jt_i \in \{0, \ldots, ub\_time\} \quad \forall i \in J$
- $de_d \in J \quad \forall d \in D$
- $ms \in \{lb\_end, \ldots, ub\_end\} \quad \forall d \in D$
- $waste \in \{0, \ldots, ub\_waste\} \quad \forall d \in D$

- $tard \in \{0, \ldots, ub\_tardiness\} \quad \forall d \in D$

The je, jt, and de variable arrays capture the job end times, job processing times, and demand end jobs. The ms, $waste$, and $tard$ variables capture the individual cost objectives.

7.1.2 Model Constraints

We use a high-level CP modeling notation to declare the constraints of the problem. Most parts of the model are directly solvable with CP solvers, however at some points we implicitly make use of constraint reification to express conditional sums and logical implications. Furthermore, we implicitly utilize the element constraint to use variables as indices for array access, and make use of the maximum global constraint.

The following constraints are used in our formulation:

- We break symmetrical job assignments by aligning unused jobs at the end of the schedule and setting the length of unused jobs to the minimum domain value:

$$(jp_i = 0) \Rightarrow (jp_{i+1} = 0) \quad \forall i \in \{1, \ldots, max\_jobs - 1\}$$
$$(jp_i = 0) \Rightarrow (jl_i = c_{\min}) \quad \forall i \in J$$

- Check that the amount of assigned job moulds is compatible with the program (we set $am_0 = 0$):

$$\sum_{m \in M} \sum_{c \in C} jm_{i,m,c} = am_{(jp_i)} \quad \forall i \in J$$

- The amount of available moulds must not be exceeded:

$$\sum_{c \in C} jm_{i,m,c} \leq a_m \quad \forall i \in J, m \in M$$

- The number of product types must not be larger than the allowed maximum:

$$\sum_{c \in C} \sum_{l \in L} \left[\sum_{m \in M} ([l_m = l] jm_{i,m,c}) > 0 \right] \leq w \; \forall i \in J$$

- Channel the tjm and jm variables:

$$tjm_{i,m,c} = jm_{i,m,c} \cdot jl_i \quad \forall i \in J, m \in M, c \in C$$

- As the channeling constraints for the tjm variables are bilinear, we additionally included an alternative linearized version of these constraints. We use an additional

set of auxiliary variables to achieve a binary encoding of the bilinear constraints as described in [GACD13]:

$$k = \lfloor \log_2(\max\{am_p | p \in P\}) \rfloor + 1$$
$$z_{i,m,c,x} \in \{0,1\}, v_{i,m,c,x} \in \{0,\ldots,c_{\max} - c_{\min} + 1\}$$
$$\forall i \in J, m \in M, c \in C, x \in \{1,\ldots,k\}$$
$$jm_{i,m,c} = \sum_{x \in \{1,\ldots,k\}} 2^{x-1} \cdot z_{i,m,c,x}$$
$$\forall i \in J, m \in M, c \in C$$
$$tjm_{i,m,c} = \sum_{x \in \{1,\ldots,k\}} 2^{x-1} \cdot v_{i,m,c,x}$$
$$\forall i \in J, m \in M, c \in C$$
$$(z_{i,m,c,x} = 1) \Rightarrow (v_{i,m,c,x} = jl_i) \wedge$$
$$(z_{i,m,c,x} = 0) \Rightarrow (v_{i,m,c,x} = 0)$$
$$\forall i \in J, m \in M, c \in C, x \in \{1,\ldots,k\}$$

- Ensure that all demands are fulfilled:

$$\sum_{d \in D} [dm_d = m \wedge dc_d = c] dq_d \leq \sum_{i \in J} tjm_{i,m,c} \forall m \in M, c \in \bigcup_{d \in D} dc_d$$

- Moulds have to be compatible with the job's program:

$$(jp_i \neq p_m) \Rightarrow (jm_{i,m,c} = 0) \quad \forall i \in J, m \in M, c \in C$$

- Only compatible colors may be assigned to the same job:

$$(\sum_{m' \in M} jm_{i,m',c_1} > 0) \Rightarrow (jm_{i,m,c_2} = 0)$$
$$\forall i \in J, m \in M, c_1 \in C, c_2 \in C \setminus comp_{c_1}$$

- Set the job time variables:

$$(jp_i = p) \Rightarrow (jt_i = jl_i \cdot t_p) \quad \forall i \in J, p \in P$$
$$(jp_i = 0) \Rightarrow (jt_i = 0) \quad \forall i \in J$$

- Set the job end time variables:

$$(jp_i > 0) \Rightarrow$$
$$\left(je_i = jt_1 + \sum_{k=2}^{i} (jt_k + sj + [jp_{k-1} \neq jp_k] \cdot (sp - sj))\right) \quad \forall i \in J$$
$$(jp_i = 0) \Rightarrow (je_i = 0) \quad \forall i \in J$$

- Set demand end job variables:

$$jp_{(de_d)} > 0 \quad \forall d \in D$$
$$\sum_{i=1}^{de_d} tjm_{i,(dm_d),(dc_d)} \geq \sum_{d' \in D'} dq_{d'} > \sum_{i=1}^{de_d - 1} tjm_{i,(dm_d),(dc_d)}$$
$$\forall d \in D, D' = \{d' \in D | dd_{d'} \leq dd_d \wedge dm_{d'} = dm_d \wedge dc_{d'} = dc_d\}$$

- Set the makespan: $ms = maximum(je)$
- Set the total waste:

$$waste = \sum_{i \in J} \sum_{m \in M} \sum_{c \in C} tjm_{i,m,c,} - \sum_{d \in D} dq_d$$

- Set total tardiness: $tard = \sum_{d \in D} maximum(\{0, je_{(de_d)} - dd_d\})$

7.1.3 Model Objective Function

The objective function aggregates the ms, $waste$, and $tard$ variables in a normalized weighted sum the same way as we have described it in the problem specification in Chapter 6.

7.2 Construction Heuristic Approach

In this section we propose a construction heuristic to quickly build solution schedules for instances of the ATSP.

The main idea is to consecutively create jobs by greedily fulfilling the demands which are ordered by their due dates. In other words, the next job is configured to fulfill the next most urgent demand as quickly as possible, using feasible mould type and color assignments.

Algorithm 10 presents the detailed procedure of the construction heuristic.

The algorithm first initializes an empty schedule and sorts the list of demands by their due date. Afterwards, the procedure creates jobs to fulfill demands in a loop until the list of sorted demands is empty. Within the outer while loop, the algorithm selects the next most urgent demand, and determines which program the job needs to use to fulfill the demand. Furthermore, the number of job cycles that are required to fulfill the demand is calculated based on the number of available moulds per cycle, as well as the minimum and maximum job length. The job is then created, and the number of used moulds within the job is updated accordingly. Additionally, the algorithm removes the processed demand from the list of remaining demands.

Algorithm 10: A construction heuristic for the ATSP

```
fn CreateSchedule
    schedule = Initialize empty schedule
    sorted_demands = sort demands by due date
    while sorted_demands.Count() > 0 do
        d = sorted_demands.GetNext()
        program = p_(dm_d)
        length = ⌈dq_d / a_m⌉
        length = min{c_max, max{length, c_min}}
        j = Create new job j
        Update used moulds of j to fulfill d
        Remove d from sorted_demands
        for d' ∈ sorted_demands do
            if j has no more unused moulds left then
                exit loop
            if d' is compatible with job j then
                Update used moulds of j to fulfill d'
                Update sorted_demands
        while j has free remaining mould slots do
            Update j to use any available mould
        schedule.AppendJob(j)
    return schedule
```

The newly created job is at this point likely to be only partially filled with moulds, and the selected program may require further moulds to be attached to this job. The inner for loop therefore goes over the complete list of sorted demands to look for other demands that could be fulfilled by this job. Thereby, for each demand it has to be checked if the required mould is still available in the job and the demanded product is compatible to the other already scheduled products so that no hard constraint would get violated. If the demand is compatible, the heuristic updates the remaining demand quantity as well as the assigned job moulds accordingly.

After the for loop, it can still be the case that free mould slots are left in the job, as no more compatible demands exist. In this case the algorithm simply fills any remaining unused mould slots by using any available mould types. The outer while loop ends by appending the newly created job at the end of the schedule. The overall job creating procedure continues until no more demands are left and afterwards the schedule is returned.

Note that this construction heuristic was developed in collaboration with domain experts to automatize the manual planning process to mimic decisions that would normally be

taken by a human planner. In the worst case, the inner for loop is executed $|D| \cdot |D|$ times, whereas the inner while loop is executed less than $|D| \cdot \max\{am_p | p \in P\}$ times. Therefore, the construction heuristic has a polynomial runtime complexity and can be efficiently used to quickly produce schedules for large-scale problem instances. However, the construction heuristic cannot guarantee feasible solutions as the procedure can run into a situation where not enough compatible mould types are available to fill a job with the required number of moulds. Such a case does not imply that a feasible solution does not exist for the particular problem instance, as the construction heuristic can get stuck by choosing color and mould assignments greedily.

For example, consider an instance where two demands that have different product types can only be fulfilled by 4 available moulds at the same time but the required number of mould slots for each job is 5. Furthermore, let the maximum number of product types per job be 2 and let the mentioned two demands be the demands with the earliest due dates. The heuristic would then try to build the first job using these two demands but could not fill the job with the required 5 moulds. If we further assume that there is a third demand that has a third product type with a late due date and many available moulds, the instance could still be solvable by not creating jobs that combine the two demands with the earliest due dates but using the third demand in combination with the first and second demand to create feasible jobs.

7.3 Metaheuristic Approach

In this section, we propose a local search based metaheuristic approach for the ATSP. We first describe the solution representation, cost function, and the generation of an initial solution. Afterwards, we propose several search neighborhoods for the problem, and describe how random neighborhood moves are generated in each search iteration. Finally, we present our neighborhood move acceptance criteria that is used to escape local optima.

7.3.1 Solution Representation and Cost Function

In our metaheuristic approach we represent solutions in a similar way as we did in the constraint model by using three variable arrays to store the assigned programs for each job, the length of each job, as well as the mould and color assignments assigned for each job. Therefore, we need to provide a parameter *max_jobs* that determines the length of these arrays and thereby limits the maximum number of jobs.

To determine the costs of candidate solutions, we use the previously defined normalized objective function but extend it in a way that it additionally captures potential hard constraint violations as follows:

$$cost(S) = \frac{ms}{ms*} + \frac{waste}{waste*} + \frac{tard}{tard*} + HC \cdot M$$

The function $cost(S)$ calculates the costs of a candidate solution S by adding the number of total hard constraint violations HC multiplied by a big constant M to the normalized objectives, where M should ideally be larger than the worst case normalized objective costs. As we normalize our objectives using a reference solution it suffices to set M to a very large integer in practice.

To determine HC we further need to define for each hard constraint how we actually count the number of violations. Regarding mould availability, we can simply count the number of assigned moulds that are unavailable. For unfulfilled demands, we count the number of missing moulds. If any incompatible colorings are assigned to a job, we count the total mould quantities that use any of the incompatible colors. We further count any mould quantities that are incompatible with the selected program. If the number of allowed product types is exceeded in a job, we first calculate the mould quantities for each product type assigned to the job. Afterwards, we count the n lowest product type quantities as violations, where n is the difference between the allowed maximum number of product types and the actual number of assigned product types. Finally, in case too many moulds are assigned to a job we simply count the excess mould quantities.

To generate an initial candidate solution for our metaheuristic approach we consider two options: We can either start search from an empty schedule or use our construction heuristic to produce an initial schedule.

7.3.2 Search Neighborhoods

In the following we propose seven search neighborhoods for our local search approach:

1. **Swap two jobs:** Swaps the positions of two existing jobs.
2. **Increment length:** Increments the cycles of a job by 1.
3. **Decrement length:** Decrements the cycles of a job by 1.
4. **Change single mould assignment:** Changes a single assigned mould type and/or color to a different mould type and/or color within the same job.
5. **Delete last job:** Deletes the last job in the schedule.
6. **Append new job:** Appends a new job at the end of the current schedule. Move parameters define the job program, as well as the mould quantities that should be used in the newly created job.
7. **Swap mould assignments between two jobs:** Swaps a single mould type and/or color assignment from a job with a single mould and/or color assignment from another job.

Note that neighborhoods 2–6 would suffice to reach all possible solution. However, the additional swap neighborhoods (1 and 7) have the advantage that they can swap mould

assignments and reposition jobs without violating any demand constraints in intermediate solutions.

We only allow the insertion and deletion of jobs at the end of the schedule mainly for the purpose of an efficient move generation. Note that the insertion of jobs is mainly motivated to handle unfulfilled demand violations, while the deletion of jobs is mainly motivated to lower the makespan and waste objective. Therefore, the purpose of these neighborhoods does not directly rely on the job position.

7.3.3 Neighborhood Exploration

Exploring the complete neighborhood easily becomes computationally expensive, especially when dealing with large real-life instances. Therefore, we do not explore the full neighborhood in our approach, but instead randomly select a single move out of the complete neighborhood in each iteration.

Which move is generated, is determined based on a random selection procedure that is configured by parameters N_1-N_7. Each parameter N_1-N_7 defines a real value between 0 and 1 that determines the probability to consider each of the seven neighborhoods in an iteration. We determine in each iteration a single random move in 3 steps: First, for each neighborhood we randomly decide based on the associated parameter whether it is considered for move generation. Afterwards, we randomly select one of the neighborhoods that have been selected in the previous step. Finally, we uniformly sample a single move from the chosen neighborhood.

7.3.4 Move Acceptance

Once a single random move has been generated, we evaluate the change of the current solution's quality that would be caused by the move. Based on the result we then decide whether the move should be applied to the current solution. We use a move acceptance function based on simulated annealing [KGV83]. The function ensures that a cost improving move is always accepted, whereas a non-cost-improving move is only accepted with a certain probability that depends on the change in solution quality as well as a temperature value. We set the temperature value at the beginning of local search to a user defined parameter. Afterwards, we use a geometrical cooling scheme that decreases the temperature value after each search iteration by multiplication with a user defined cooling rate parameter.

In our cooling scheme we further set the number of iterations per temperature to the total number of demanded moulds of the given instance. Therefore, the number of iterations between two consecutive cooling steps is determined relative to the instance size.

Algorithm 11 presents the full acceptance function, where $cost(S)$ is the cost of the current solution, $cost(S*)$ is the cost after the application of the randomly generated move, T is the current temperature value, and $random()$ is a uniformly sampled real value between 0 and 1.

Algorithm 11: Move acceptance function

```
fn AcceptMove (cost(S), cost(S*), T)
    result = True
    if cost(S) ≤ cost(S*) then
        p = e^(−(cost(S*)−cost(S))/T)
        if random() > p then
            result = False
return result
```

7.4 Computational Results

In this section we first describe the experimental environment and parameter configuration before we later present and discuss computational results.[1] All of our experiments as well as the parameter tuning were conducted on a computing cluster with 10 identical nodes, each having 24 cores, an Intel(R) Xeon(R) CPU E5–2650 v4 @ 2.20GHz and 252 GB RAM.

To evaluate the proposed solution methods for the set of real-life benchmark instances, we decided to set $w_{1-3} = 1$ as in the particular practical scheduling scenarios all three objectives are considered to be of similar importance. Furthermore, we use the construction heuristic approach to generate all reference solution costs. As this method was developed in collaboration with domain experts to automatize the manual planning process, it represents a baseline for the quality of current practical results.

Therefore, we used our construction heuristic to produce reference solutions for all the instances (i.e. the aggregated normalized solution cost is always exactly 3 for the construction heuristic) in a first series of experiments. The *max_job* parameter was set for the constraint model to 50 and for the metaheuristic approach to 500 (50 should clearly suffice for all benchmark instances, however for the metaheuristic we could even use a higher value without risking memory leaks).

To evaluate the metaheuristic approach we further have to configure several parameters. Based on some manual tuning attempts we selected a T value of 0.001 and a α value of 0.999 and set all neighborhood probabilities to 1 as the default. Starting from the default values, we further used the state-of-the-art parameter tuning software SMAC [LEF+17] to automatically tune all the parameters (Parameter value ranges were restricted to $T \in [0.0001, 2]$, $\alpha \in [0.9, 0.9999]$, and $N1 - N7 \in [0, 1]$). The tuning process was then started with the metaheuristic that starts from an initial reference solution and all 12 instances as the training set. Similar as with the parameter tuning for the paint shop scheduling problem that was previously described in Section 4.8.2, we use all instances for training in this case. Therefore, we want to note that a robustness analysis of the tuned

[1] Detailed results are publicly available online:
https://www.dbai.tuwien.ac.at/staff/winter/atsp.zip

parameters on a test set consisting of unseen instances is an important subject of future work. We further set the runtime limit for each individual run to 10 minutes and set the overall wallclock time limit to 4 days. The resulting parameter configuration which we used for our final experiments is as follows: $T = 0.4735, \alpha = 0.9274, N1 = 0.2042, N2 = 0.0407, N3 = 0.8522, N4 = 0.0632, N5 = 0.8630, N6 = 0.6250$, and $N7 = 0.3972$.

To evaluate exact approaches that utilize the proposed bilinear and linearized CP model we implemented both models using the modeling language MiniZinc [NSB+07], which provides interfaces to state-of-the-art CP and MIP solvers (for the latter MiniZinc automatically converts the constraint model into a MIP model).[2]

We then performed experiments with the MIP solvers gurobi [GO20] and cplex [Cor19], as well as the CP solvers gecode [Gec19] and chuffed [Chu11]. As chuffed is not able to handle floating point objectives, we simply used non normalized values in the objective for chuffed and normalized the final objective in a post-processing step.

For gecode and chuffed we further used a programmed search strategy that first selects all jm variables based on the smallest domain first heuristic where minimum values are assigned first. For the remaining variables, we use the solvers' default search and further activated the free search parameter for chuffed which allows the solver to alternate between the given search strategy and its default one on each restart. Further, we set a time limit of 1 hour for each run for all evaluated exact and metaheuristic methods. Numerical values have been rounded to two decimal places in all final results. Table 7.1 summarizes the final results produced by exact methods.

Note that Table 7.1 only displays results for instances 1–6 (I 1–6), as none of the exact approaches were able to reach feasible solutions within the time limit for instances 7–12. Each row shows the final normalized objective value reached for the corresponding instance with solvers cplex (Cpx), gurobi (Grb), gecode (Gce), and chuffed (Chu) using the bilinear channeling constraints (B) and the linearized channeling constraints (L) (note that for both versions we use the MiniZinc linearization library that automatically converts the proposed constraint model into a linearised formulation compatible with the MIP solvers so that only the linearized version of the channeling constraints is specified explicitly in its linearized form). Best results for each row are formatted in boldface and a - denotes that no solution could be found within the time limit.

The results presented in Table 7.1 show that gurobi produces overall the best results for all instances. All approaches are able to reach the best objective value of 3 for instance 6 which is equal to the reference solution cost. As instance 6 contains only a single demand (see Table 6.2) this is an expected result (clearly in such a case one cannot do better then building jobs of maximum length that utilize the maximum number of moulds to fulfill the single demand and therefore the construction heuristic provides an optimal result). The results further show that the CP solvers gecode and chuffed seem to be not competitive compared to the MIP solvers for instances 1–5. We see that there are only

[2]The models are publicly available online:
`https://www.dbai.tuwien.ac.at/staff/winter/atsp_minizinc.zip`

Inst.	Cpx L	Cpx B	Grb L	Grb B
I 1	2.96	4.66	2.54	**2.53**
I 2	2.01	3.24	**1.96**	2.01
I 3	**2.23**	**2.23**	**2.23**	**2.23**
I 4	**2.54**	**2.54**	**2.54**	**2.54**
I 5	2.11	2.12	2.11	**2.10**
I 6	**3.00**	**3.00**	**3.00**	**3.00**

Inst.	Gce L	Gce B	Chu L	Chu B
I 1	-	37.53	579.59	-
I 2	-	-	19.31	272.87
I 3	69.69	3.00	3.00	1123.83
I 4	37.87	37.87	99.09	619.50
I 5	43.85	45.11	10.32	-
I 6	**3.00**	**3.00**	**3.00**	**3.00**

Table 7.1: Summarized results for exact methods.

Inst.	Cpx L DB	Cpx B DB	Grb L DB	Grb B DB
I 1	1.13	1.34	1.87	**2.08**
I 2	0.83	0.67	0.99	**1.25**
I 3	**2.23**	**2.23**	**2.23**	**2.23**
I 4	**2.54**	**2.54**	**2.54**	**2.54**
I 5	1.29	1.12	1.38	**1.63**
I 6	**3.00**	**3.00**	**3.00**	**3.00**
I 7	0.47	0.47	**0.50**	0.49
I 8	**0.15**	0.14	**0.15**	0.14
I 9	0.58	0.56	0.58	**0.59**
I 10	**0.53**	0.51	**0.53**	0.51
I 11	**0.34**	**0.34**	**0.34**	0.32
I 12	**1.02**	0.96	0.96	0.96

Table 7.2: Final dual bounds achieved by MIP methods.

small differences between the bilinear and linearized models, especially for gurobi. We assume this is due to the fact that gurobi recently introduced improved techniques for bilinear constraints.

Table 7.2 provides an overview on the best dual bounds (DB) by the evaluated MIP solvers. The best dual bounds per row are formatted in boldface.

We can see that for the large instances, most of the best dual bounds can be obtained with the linearized model. For small instances on the other hand the bilinear model with

Inst.	LS B	LS M	LS S	LSC B	LSC M	LSC S
I 1	2.53	**2.53**	0.00	2.53	**2.53**	0.00
I 2	1.94	**1.95**	0.01	1.94	**1.95**	0.01
I 3	2.23	**2.23**	0.00	2.23	**2.23**	0.00
I 4	2.54	**2.54**	0.00	2.54	**2.54**	0.00
I 5	2.13	2.20	0.11	2.13	**2.14**	0.02
I 6	3.00	**3.00**	0.00	3.00	**3.00**	0.00
I 7	-	-	-	2.98	**3.00**	0.01
I 8	-	-	-	2.39	**2.42**	0.02
I 9	-	-	-	2.97	**2.99**	0.01
I 10	5.70	6.36	0.59	2.78	**2.87**	0.06
I 11	-	-	-	2.76	**2.77**	0.01
I 12	6.10	6.86	0.74	2.89	**2.97**	0.04

Table 7.3: Overview on computational results for local search.

gurobi produced the best results. This indicates that for large problems linearizing the constraints can be helpful to quickly obtain good dual bounds.

Table 7.3 summarizes results produced by the metaheuristic approach starting from an empty schedule (LS) and starting from an initial reference solution (LSC). Note that the proposed method is not deterministic, as neighborhood moves are randomly generated in each iteration. Therefore, these results were obtained by 10 repetitive runs on each instance and the table displays in addition to the overall best cost (B) also the mean costs (M) and the standard deviation (S). The best mean costs per instance are formatted in boldface and a - denotes that no feasible solution was reached.

The results presented in Table 7.3 show that starting from an initial schedule produces the best mean costs for all instances. Furthermore, we can see that for the majority of the instances starting from an empty solution cannot reach any feasible solution within the runtime limit. This indicates that starting from a construction heuristic is very effective to deliver robust and good results especially for large instances.

Finally, Table 7.4 summarizes the overall best lower bounds (LB) and overall best results produced with exact (Exact) and metaheuristic methods (LS).

We can see in the results that the exact methods could prove optimality for instances 3, 4, and 6 and that metaheuristics could also reach optimal results in these cases. It seems that overall the exact methods produce results of similar quality compared to the metaheuristic approach on the smaller instances. However, we can clearly see that the metaheuristic performed better for the large instances.

Inst.	LB	Exact	LS
I 1	2.08	**2.53**	**2.53**
I 2	1.25	1.96	**1.94**
I 3	2.23	**2.23**	**2.23**
I 4	2.54	**2.54**	**2.54**
I 5	1.63	**2.10**	2.13
I 6	3.00	**3.00**	**3.00**
I 7	0.50	-	**2.98**
I 8	0.15	-	**2.39**
I 9	0.59	-	**2.97**
I 10	0.53	-	**2.78**
I 11	0.34	-	**2.76**
I 12	1.02	-	**2.89**

Table 7.4: Summary of the overall best results.

CHAPTER 8

A Hyper-Heuristic Approach for Artificial Teeth Scheduling

In this chapter we investigate a hyper-heuristic approach for the artificial teeth scheduling problem. We first provide a literature review on hyper-heuristics and give some background on a well known flexible hyper-heuristic framework (HyFlex) that we utilize to develop a hyper-heuristic solution method for the artificial teeth scheduling problem. Afterwards, we propose several low-level heuristic operators for the artificial teeth scheduling problem which can be used together with state-of-the-art hyper-heuristic strategies to tackle large realistic benchmark instances. Finally, at the end of this chapter we evaluate the performance of our low-level heuristic operators with different hyper-heuristics in a set of computational experiments.

8.1 Background & Related Work

When developing efficient heuristic solution methods for optimization problems that arise from real-life applications, usually problem specific domain-knowledge is exploited to create efficient solution methods. However, traditional heuristics that strongly rely on domain specific strategies often do not generalize well on other application domains. The main idea behind hyper-heuristics, which is a field that has been the topic of intensive study in the last decades, is to develop problem-independent heuristic strategies which are able to generalize well on different problem domains.

In [PQ18], the authors differentiate between two main types of hyper-heuristic approaches: generation based approaches that generate low-level heuristics and selection based approaches that focus on an efficient selection of low-level heuristics. Low-level heuristics are problem-specific heuristics that are utilized by the hyper-heuristic approach to solve instances of a particular problem domain. In the case of selection based approaches

these low-level heuristics are designed by a domain expert, whereas generation based hyper-heuristics aim to generate new low-level heuristics by combining existing problem-dependent heuristics.

In this chapter we propose a set of low-level heuristics for the artificial teeth scheduling problem which can be utilized by selection based hyper-heuristics that choose and apply different low-level heuristics during an iterative search process. Over the recent decade this topic has been a subject of intensive study and a large amount of work has been reviewed in surveys such as [BGH+13] and [DKÖB20].

In the recent past several hyper-heuristic frameworks have been proposed that encourage the development of selection based approaches and allow researchers to measure and compare hyper-heuristics strategies on sets of low-level heuristics from different domains. For example, the HyFlex (Hyper-Heuristics Flexible) framework was introduced in [OHC+12] and is a software framework that clearly separates domain specific code such as solution representation, objective function and low-level heuristic operators from problem-independent code concerning the hyper-heuristic approach. This allows researchers to focus on the development of novel selection hyper-heuristic methods without the need of implementing low-level heuristics. At the same time, researchers that want to evaluate hyper-heuristic approaches on a particular problem domain can implement new low-level heuristics for their problem and use the hyper-heuristics that have been implemented in the HyFlex framework.

A hyper-heuristic that is implemented in HyFlex manages a pool of candidate solutions when solving an instance of a supported problem domain. New solutions can be added to this pool by calling problem specific low-level construction heuristics or by copying/overwriting existing solutions. Furthermore, Hyflex supports four different types of low-level heuristics which can be called on one or two candidate solutions: *Mutational*, *ruin-create*, *local search (hill climbing)* or *crossover* operators. *Mutational* heuristics perform a random perturbation on the given solution, whereas *ruin-create* operators destroy parts of the solution before repairing them again to create a feasible solution. *Local search* low-level heuristics iteratively perform modifications to candidate solutions and guarantee that the quality of the modified solutions is not reduced. Finally, *crossover* operators combine parts of two candidate solutions to form a new solution.

The HyFlex framework itself provides low-level implementations for the Boolean satisfiability problem, a one-dimensional bin-packing problem, personnel scheduling, a permutation flow-shop problem, the traveling salesperson problem, and vehicle routing. These problem domains together with associated sets of benchmark instances were used in the Cross-Domain Heuristic Search Challenge (CHeSC 2011[1]), where 20 different international teams participated with various hyper-heuristic approaches. To determine the winning teams, the results of all participants were ranked using a Formula 1 scoring system.

[1] http://www.asap.cs.nott.ac.uk/chesc2011/

The first place of the competition was achieved by an algorithm called AdapHH (also known as GIHH) from [MVDCVB12] which maintains subsets of low-level heuristics in different stages of the search process. These sets are dynamically adapted based on performance measures for each heuristic such as for example the number of the best solutions found or improvements over time.

Ranked second in the competition was a hyper-heuristic approach that is based on variable neighborhood search by [HCF12]. The approach conducts two phases, where the first phase operates on a population of solutions and the second phase only keeps a single best solution. Each of these phases starts with a shaking stage that focuses on diversification and a local search stage that focuses on intensification using different types of low-level heuristics.

A hyper-heuristic called ML [OHC+12] reached the third place of the competition. This method uses a self-adaptive metaheuristic that utilizes reinforcement learning and multi-cooperative agents. Furthermore, ML operates in three main stages: First a diversification phase, then an intensification phase, and finally a move acceptance phase where moves are accepted in case of an improvement or if no improvement could be found for a given number of iterations.

The remaining participants use a variety of different solution concepts such as for example iterated local search ([CXIC12]), evolutionary programming ([Mei11]), splitting the search into a single-point and a population based strategy ([LM12]), and many others.

HyFlex continues to be an important benchmark tool for the hyper-heuristic community even after the competition and several other approaches have been since developed within this framework [DKÖB20].

In addition to the HyFlex framework, several other selection-based hyper-heuristic frameworks have been proposed in the recent decade. Examples are the HyFlex 1.1 framework [AÖP13] which additionally allows the hyper-heuristics to handle batches of instances collectively and EvoHyp [PB17] which focuses on the development of hyper-heuristics that are based on evolutionary algorithms.

Recently, another hyper-heuristic approach that focuses on an automated combination of given neighborhood-based heuristics was proposed in [Chu20]. The main idea behind this work is to automatically find chains of different neighborhood operators that allow an efficient interaction of the individual heuristic strategies. This is realized by learning mechanisms that utilize data that is collected during the search process.

Furthermore, in [LG07] a self-adapting large neighborhood search approach for solving scheduling problems has been proposed that implements ideas that are similar to selection based hyper-heuristic approaches. The work uses sets of neighborhoods and completion strategies that play a similar role as the low-level heuristics from hyper-heuristic frameworks. Additionally, a machine learning technique is utilized to learn weights for an efficient selection of the different neighborhoods during the search process. The idea of self-adapting large neighborhood search has further been extended in [TS18],

where the learning mechanism not only watches the quality improvements achieved by the neighborhood operators, but also considers the required runtime of the individual heuristic calls.

We note that besides the related work discussed in this section, many additional selection based hyper-heuristic approaches have been investigated in the literature. For a comprehensive overview on this topic please refer to a recent survey (e.g. [DKÖB20]).

8.2 Low-Level Heuristics for the Artificial Teeth Scheduling Problem

In this chapter we propose low-level heuristics for the artificial teeth scheduling problem that can be used together with the HyFlex framework. We decided to develop our heuristics within this framework, as it has been widely used in the field and is publicly available. Therefore, we categorize the proposed heuristics into *mutational*, *local search*, and *crossover* operators as these are low-level heuristic types supported in HyFlex.

8.2.1 Mutational Heuristics

The idea behind mutational low-level heuristics is to perform small perturbative changes to the components of a candidate solution. In the HyFlex framework a mutational operator is further configured by an *intensity of mutation* parameter $\alpha \in [0, 1]$ that controls the level of the mutation, where a higher value leads to more modifications being performed.

We propose 7 low-level heuristic mutation moves for the artificial teeth scheduling problem that correspond to the search neighborhoods we proposed previously in Chapter 7: *Swap Two Jobs*, *Increment Job Length*, *Decrement Job Length*, *Change Single Mould Assignment*, *Delete Last Job*, *Append new Job*, *Swap Two Mould Assignments*.

The mutational low-level operator works as follows for all 7 move types: Given is a candidate solution that is mutated by iteratively applying uniformly random selected moves of the selected move type. For example, a mutation low-level heuristic which uses the *Swap Two Jobs* move, in its first iteration randomly selects one out of all possible swap moves for the given candidate solution. This move is then performed to modify the solution, and the process continues for a number of iterations.

To consider the *intensity of mutation* parameter we set the iteration limit to $\lfloor \alpha \cdot k \rfloor$ so that k moves will be performed in any mutational low-level heuristic move if $\alpha = 1.0$, where k is an additional parameter given to the mutational operator.

8.2.2 Local Search Heuristics

In this section we propose local search based low-level heuristics for the ATSP. The HyFlex framework configures local search low-level heuristics using a *depth of search*

parameter $\beta \in [0, 1]$ that controls how many search iterations will be performed to improve the candidate solution. All proposed heuristics guarantee to not reduce the quality of the given candidate solution.

Stochastic Hill Climber

The first low-level local search heuristic we propose randomly generates moves in each search iteration in a similar way as it has been described for the simulated annealing approach given in Section 7.3.

After a move has been randomly selected, this heuristic evaluates the change in costs that would be caused by this move to the current candidate solution. Then, the acceptance function we introduced in Section 7.3 is used to decide whether the move should be accepted using a given temperature parameter τ. Note that for this heuristic the temperature value that is given as a parameter will not change (like it is the case with simulated annealing), but instead is fixed to the given parameter for all search iterations.

The iteration limit for this low-level heuristic is determined by $\lfloor \beta \cdot k \rfloor$, where k is a given integer parameter. Additionally, a temperature parameter *tau* and a time limit t configure the fixed temperature value and a timeout. The heuristic stops if the iteration limit or time limit is reached.

Due to the nature of the acceptance function this low-level heuristic can produce solutions of reduced quality. However, the heuristic always returns the overall best solution found at the end of its execution (if no improvement was found, the operator simply returns the initial solution) and thereby can be considered as a local search operator.

Simulated Annealing

In addition to the *Stochastic Hill Climber* heuristic, we propose another similar low-level local search operator that uses the simulated annealing approach exactly in the same way as we introduced it in Section 7.3.

Similarly as with the *Stochastic Hill Climber* operator the initial temperature is given as parameter τ, but here we use an additional cooling rate parameter γ to configure the geometrical cooling schedule. Again we determine the iteration limit as $\lfloor \beta \cdot k \rfloor$ with k being an integer parameter, and the time limit for the simulated annealing low-level heuristic is set by parameter t.

Full Neighborhood Move Heuristics

In addition to the *Stochastic Hill Climber* and *Simulated Annealing* operators that both generate random moves in each search iteration, we introduce a set of local search low-level heuristics that consider the full search neighborhood for particular move operators.

For example, a full neighborhood move heuristic for the *Swap Two Jobs* move type generates moves for all possible pairs of jobs in the current candidate solution. Afterwards, the

change in solution quality for all generated moves is evaluated and the full neighborhood move heuristic applies the move that leads to the best solution quality (ties are broken by the order of the generated moves). However, in case none of the generated moves leads to an improvement no move is performed.

Following this idea, we propose four full neighborhood move heuristics:

1. *Full Change Mould Neighborhood Heuristic:* This low-level heuristic considers all possible instantiations of the *Change Single Mould Assignment* neighborhood operator on the given candidate solution.
2. *Full Job Length Neighborhood Heuristic:* This heuristic considers all length modifying neighborhood operators (i.e. *Increment/Decrement Job Length*) for all jobs in the given candidate solution.
3. *Full Swap Job Neighborhood Heuristic:* Considers *Swap Two Jobs* moves for all possible job pairs in the given solution.
4. *Full Swap Mould Neighborhood Heuristic:* This heuristic selects the best of all possible *Swap Two Mould Assignments* moves.

All four full neighborhood move low-level heuristics use no parameters as they only perform at most a single modification on execution (if no improving move is found, the current solution is not modified).

Note that we did not consider full neighborhood heuristics for the *Append Job* and *Delete Job* neighborhood moves. We decided to not include these heuristics as in the case of the *Append Job* operator an evaluation of all possible jobs would lead to an inefficient generation of a very large number of jobs, and in the case of the *Delete Job* there is only a single possible move (i.e. deletion of the last job) for each candidate solution anyway.

8.2.3 Crossover Heuristic

We now propose a crossover low-level heuristic that combines two given candidate solutions for the ATSP. The main idea behind this operator is to perform a one point crossover on the job sequences of the two given solutions. To achieve this, first a random job index is selected. Then the resulting job sequence is built by first scheduling all jobs from the first candidate solution that have a lower or equal job index. Afterwards, all jobs from the second solution that have a larger job index are appended on the resulting solution. Algorithm 12 shows the details of the crossover procedure where S_1 and S_2 are the given candidate solutions.

The algorithm starts by determining the maximum length of the two candidate job sequences. Afterwards, a random dividing job index is uniformly sampled. Finally, the result sequence is initialized as an empty job sequence and jobs from the first and second candidate solution are appended which are selected based on the position of the dividing job index.

Algorithm 12: Crossover Low-Level Heuristic for the ATSP

```
fn Crossover (S_1, S_2)
    maxNumJobs = max(length(S_1), length(S_2))
    dividingJob = random(1, maxNumJobs)
    result = new empty job sequence
    for i = 1 to min(dividingJob, length(S_1)) do
        result.append(S_1[i])
    for i = dividingJob + 1 to length(S_2) do
        result.append(S_2[i])
return result
```

8.3 Evaluated Hyper-Heuristic Approaches

We contacted the authors that achieved the first, second and third place of the CHeSC 2011 competition and asked them to share their implementation of the winning hyper-heuristics so that we could include them in our evaluation. Thereby, we received an up-to-date implementation of GIHH [MVDCVB12] which was the competition winner. The algorithms that were ranked second and third were unfortunately not available, but we also received an implementation of the HAHA hyper-heuristic [LM12] which scored sixth place in the competition.

In addition to algorithms that participated in CHeSC 2011, we also evaluated a recent hyper-heuristic approach from [Chu20] (CHUANG) as well as two hyper-heuristics which are based on self-adaptive large neighborhood search strategies from [LG07] (ALNS) and [TS18] (ALNS2). These three approaches have recently been adapted and implemented for use within the HyFlex framework by [MM21].

8.4 Computational Results

In this section we present computational results for the ATSP that were achieved by the selection based hyper-heuristic approaches mentioned in the previous section. For our experiments we implemented all low-level heuristics proposed in this chapter within the latest version of the HyFlex framework that is publicly available online[2].

To evaluate the proposed low-level heuristics with the mentioned hyper-heuristic strategies, we then performed a series of experiments using the benchmark instances for the ATSP we introduced in Chapter 6. Regarding the experimental setup, benchmark machine, and configuration of the objective function we used exactly the same configuration as in the computational results section of the previous chapter (please refer to Section 7.4 for details).

[2] http://www.asap.cs.nott.ac.uk/external/chesc2011/hyflex_download.html

Inst.	GIHH	HAHA	ALNS	ALNS2	CHUANG
I 1	2.53	2.53	2.53	2.53	**2.52**
I 2	1.95	**1.94**	**1.94**	1.95	1.95
I 3	**2.23**	**2.23**	**2.23**	**2.23**	**2.23**
I 4	**2.54**	**2.54**	**2.54**	**2.54**	**2.54**
I 5	**2.11**	2.12	2.12	2.13	**2.11**
I 6	**3**	**3**	**3**	**3**	**3**
I 7	2.86	**2.85**	2.95	2.88	2.89
I 8	2.47	2.46	2.47	**2.39**	2.49
I 9	2.87	**2.86**	2.91	**2.86**	2.9
I 10	2.68	**2.67**	2.76	2.72	2.74
I 11	2.77	**2.76**	2.81	2.8	2.79
I 12	2.85	**2.79**	2.95	2.96	2.9

Table 8.1: Overview on overall best results per instance.

Based on manual tuning trials we further set the parameters of the low-level heuristics as follows: For the mutational operators, we set the iteration limit $k = 100$. Additionally, we set the default intensity of mutation to $\alpha = 0.1$. However, note that this is only the initial value of α and this parameter can be controlled dynamically during search by each individual hyper-heuristic.

For the *Stochastic Hill Climber* local search heuristic we set the time limit t to 60 seconds. Further, we determine the iteration limit k dependent on instance size parameters by calculating the product of the number of colors, the number of mould types, the number of demands, and the total number of demanded moulds. Additionally, we set the temperature $\tau = 0.01$ and set the default depth of search $\beta = 0.1$

For the *Simulated Annealing* low-level heuristic, we use exactly the same parameters as for the *Stochastic Hill Climber* heuristic, except for the initial temperature τ, which we set to 0.4735. Further, we set the cooling rate parameter to 0.9274. Note that these two parameter values correspond to the tuning results from Section 7.4.

As some full neighborhood move heuristics can require a large processing time when computing all possible moves, we additionally imposed a maximum runtime limit of 10 minutes to each low-level heuristics in this category.

Using this parameter configuration we then performed 10 repeated experimental runs for each instance and hyper-heuristic approach, where every single run was given a time limit of 1 hour. Table 8.1 shows the overall best results per instance and hyper-heuristic.

The table shows in Columns 2–6 the best solution quality per instance produced with the evaluated hyper-heuristic approaches. Best results for each line are formatted in boldface. For the remainder of this section, we apply a similar formatting to all tables unless stated otherwise. The results displayed in Table 8.1 show that HAHA is able to

Inst.	GIHH M	HAHA M	ALNS M	ALNS2 M	CHUANG M
I 1	2.62	2.57	**2.53**	2.7	**2.53**
I 2	**1.95**	**1.95**	**1.95**	2.24	1.97
I 3	**2.23**	**2.23**	**2.23**	**2.23**	**2.23**
I 4	**2.54**	**2.54**	**2.54**	2.63	**2.54**
I 5	**2.12**	**2.12**	2.18	2.48	2.2
I 6	**3**	**3**	**3**	**3**	**3**
I 7	2.87	**2.86**	2.97	2.93	2.92
I 8	**2.48**	2.49	2.57	2.54	2.52
I 9	**2.89**	**2.89**	2.94	2.93	2.92
I 10	2.7	**2.69**	2.82	2.81	2.78
I 11	**2.78**	**2.78**	2.85	2.86	2.81
I 12	**2.93**	**2.93**	2.96	2.97	2.94

Table 8.2: Overview on mean objective costs per instance.

Inst.	GIHH S	HAHA S	ALNS S	ALNS2 S	CHUANG S
I 1	0.18	0.13	**0**	0.22	**0**
I 2	**0**	0.01	0.01	0.3	0.06
I 3	**0**	**0**	**0**	**0**	**0**
I 4	**0**	**0**	**0**	0.19	**0**
I 5	**0.01**	**0.01**	0.12	0.26	0.15
I 6	**0**	**0**	**0**	**0**	**0**
I 7	**0.01**	**0.01**	0.02	0.05	0.03
I 8	**0.01**	**0.01**	0.09	0.07	0.05
I 9	0.02	**0.01**	**0.01**	0.04	0.02
I 10	**0.01**	0.02	0.05	0.09	0.05
I 11	**0.01**	**0.01**	0.05	0.06	**0.01**
I 12	0.04	0.05	**0**	0.01	0.02

Table 8.3: Overview on standard deviation of the objective costs per instance.

produce the best results for 9 of the 12 instances, and therefore seems to be the overall best performing method in this comparison. However, we note that for instances 1, 5, and 8 other methods (CHUANG, GIHH or ALNS2) can produce slightly better results and GIHH provides solutions with only slightly higher costs than HAHA for the majority of the instances.

Tables 8.2 and 8.3 further show the mean objective costs (M) as well as the standard deviation (S) achieved over all 10 runs per instance.

Again we see that HAHA seems to be the overall best performing approach as it produces the best results for 10 out of 12 instances in Table 8.2. However, GIHH also provides the

Inst.	GIHH*	HAHA*	ALNS*	ALNS2*	CHUANG*
I 1	**2.52**	2.53	2.53	**2.52**	2.53
I 2	**1.94**	**1.94**	**1.94**	1.95	**1.94**
I 3	**2.23**	**2.23**	**2.23**	**2.23**	**2.23**
I 4	**2.54**	**2.54**	**2.54**	**2.54**	**2.54**
I 5	**2.11**	**2.11**	2.12	2.12	2.13
I 6	**3**	**3**	**3**	**3**	**3**
I 7	**2.88**	2.9	2.89	**2.88**	**2.88**
I 8	2.45	2.48	2.46	**2.4**	2.49
I 9	**2.87**	2.91	2.91	**2.87**	2.89
I 10	2.69	2.71	2.67	2.66	**2.59**
I 11	**2.79**	2.8	**2.79**	**2.79**	**2.79**
I 12	**2.9**	2.91	**2.9**	**2.9**	**2.9**

Table 8.4: Overview on the overall best results per instance achieved without the full neighborhood low-level operators.

best mean objective costs for 9 of the 12 instances and comes close to most of the results produced by HAHA. The standard deviation of the objective costs is close to 0 for most approaches and instances with only a few exceptions. This indicates that all evaluated hyper-heuristics are able to produce robust results regarding different random seeds.

The full neighborhood search heuristics may consume large amounts of execution time, and actually the evaluation of the full neighborhood required several minutes just for a single low-level heuristic call for some realistic instances in our experiments. Therefore, we decided to conduct a second set of experiments without using the full neighborhood search heuristics to investigate whether this has a negative effect on the overall performance of the hyper-heuristics. To mark results that have been produced without the full neighborhood low-level heuristics, we add a * after the identifiers of the evaluated methods in the following.

Table 8.4 summarizes the overall best results for all evaluated strategies without the full neighborhood low-level operators.

We can see in the results shown in the table that this time GIHH* produced the overall best results as it could provide best costs for 10 of the 12 instances. HAHA* could not reach best results for the majority of the instances. However, ALNS2* and CHUANG* were able to further improve costs for instances 8 and 9 respectively.

Furthermore, tables 8.5 and 8.6 display the mean objective costs (M) and the standard deviation (S) over all runs per instance achieved without the full neighborhood low-level operators.

Similar as with the best per instances results, GIHH* produces the best overall mean objective costs out of the 10 repeated runs for the majority of the instances. Only on

Inst.	GIHH* M	HAHA* M	ALNS* M	ALNS2* M	CHUANG* M
I 1	**2.53**	**2.53**	2.61	2.79	2.61
I 2	**1.95**	**1.95**	**1.95**	2.21	1.96
I 3	**2.23**	**2.23**	**2.23**	**2.23**	**2.23**
I 4	**2.54**	**2.54**	**2.54**	2.58	**2.54**
I 5	2.2	2.21	**2.14**	2.32	2.17
I 6	**3**	**3**	**3**	**3**	**3**
I 7	**2.89**	2.93	2.91	2.92	2.9
I 8	**2.49**	2.52	2.50	2.51	2.5
I 9	**2.91**	2.93	2.94	2.92	2.92
I 10	2.71	2.75	2.72	2.76	**2.69**
I 11	2.8	2.82	2.82	2.82	**2.79**
I 12	**2.93**	2.94	2.94	2.95	2.94

Table 8.5: Overview on mean objective costs per instance achieved without the full neighborhood low-level operators.

Inst.	GIHH* S	HAHA* S	ALNS* S	ALNS2* S	CHUANG* S
I 1	**0**	**0**	0.17	0.22	0.17
I 2	**0.01**	**0.01**	**0.01**	0.23	0.06
I 3	**0**	**0**	**0**	**0**	**0**
I 4	**0**	**0**	**0**	0.14	**0**
I 5	0.15	0.15	**0.03**	0.2	0.11
I 6	**0**	**0**	**0**	**0**	**0**
I 7	**0.01**	0.02	0.02	0.03	**0.01**
I 8	0.02	0.02	0.03	0.06	**0.01**
I 9	0.02	**0.01**	0.02	0.03	0.02
I 10	**0.02**	**0.02**	0.03	0.07	0.05
I 11	**0.01**	**0.01**	**0.01**	0.04	**0.01**
I 12	0.03	**0.02**	0.03	**0.02**	**0.02**

Table 8.6: Overview on standard deviation of the objective costs per instance achieved without the full neighborhood low-level operators.

Inst.	HAHA B	HAHA M	HAHA S	GIHH* B	GIHH* M	GIHH* S
I 1	2.53	2.57	0.13	2.52	**2.53**	0.00
I 2	1.94	**1.95**	0.01	1.94	**1.95**	0.01
I 3	2.23	**2.23**	0.00	2.23	**2.23**	0.00
I 4	2.54	**2.54**	0.00	2.54	**2.54**	0.00
I 5	2.12	**2.12**	0.01	2.11	2.20	0.15
I 6	3.00	**3.00**	0.00	3.00	**3.00**	0.00
I 7	2.85	**2.86**	0.01	2.88	2.89	0.01
I 8	2.46	**2.49**	0.01	2.45	**2.49**	0.02
I 9	2.86	**2.89**	0.01	2.87	2.91	0.02
I 10	2.67	**2.69**	0.02	2.69	2.71	0.02
I 11	2.76	**2.78**	0.01	2.79	2.80	0.01
I 12	2.79	**2.93**	0.05	2.90	**2.93**	0.03

Table 8.7: Comparison of results achieved with HAHA and GIHH*

instances 5, 10, and 11 ALNS* or CHUANG* achieve better results. The standard deviation over all runs are close to 0 for the large majority of the instances, again indicating the robustness of the hyper-heuristics on different random seeds.

As HAHA produced overall best results with inclusion of the full neighborhood low-level heuristics and GIHH* overall performed best without the consideration of full neighborhoods, we further directly compare the results produced by these two methods in Table 8.7. The table displays the best cost (B), mean cost (M), and standard deviation (S) over all 10 individual runs per instance for both HAHA and GIHH*. Here, best mean cost results per instance are formatted in boldface.

We clearly see that HAHA overall produces better results, as mean objective cost results are better compared to GIHH* for all instances except instance 1. Similarly, when comparing the best costs per instance HAHA can produce the best results for 9 instances, whereas GIHH* only reaches best costs in 7 cases. We conclude that the inclusion of the full neighborhood low-level heuristics overall do not have a negative impact on the performance of the evaluated hyper-heuristics but instead can improve results for the majority of the instances. A possible explanation for this result could be that many hyper-heuristic strategies such as GIHH perform an online selection and performance evaluation of the given low-level heuristics allowing them to automatically detect and remove inefficient heuristics during the search process.

In Table 8.8, we further compare the results produced with HAHA and the simulated annealing approach from the previous chapter.

The table displays the best cost (B), mean cost (M), and standard deviation (S) over all 10 individual runs per instance for both the simulated annealing approach from the previous chapter (SA), and the HAHA hyper-heuristic. Best mean cost results per instance are formatted in boldface.

Looking at the mean cost results shown in Table 8.8, we see that HAHA improves results for instances 5, 7, 9, 10, and 12 compared to simulated annealing. However, the simulated

Inst.	HAHA B	HAHA M	HAHA S	SA B	SA M	SA S
I 1	2.53	2.57	0.13	2.53	**2.53**	0.00
I 2	1.94	**1.95**	0.01	1.94	**1.95**	0.01
I 3	2.23	**2.23**	0.00	2.23	**2.23**	0.00
I 4	2.54	**2.54**	0.00	2.54	**2.54**	0.00
I 5	2.12	**2.12**	0.01	2.13	2.14	0.02
I 6	3.00	**3.00**	0.00	3.00	**3.00**	0.00
I 7	2.85	**2.86**	0.01	2.98	3.00	0.01
I 8	2.46	2.49	0.01	2.39	**2.42**	0.02
I 9	2.86	**2.89**	0.01	2.97	2.99	0.01
I 10	2.67	**2.69**	0.02	2.78	2.87	0.06
I 11	2.76	2.78	0.01	2.76	**2.77**	0.01
I 12	2.79	**2.93**	0.05	2.89	2.97	0.04

Table 8.8: Comparison of results achieved with simulated annealing and the GIHH hyper-heuristic.

Inst.	LB	Exact	SA	HAHA
I 1	2.08	**2.53**	**2.53**	**2.53**
I 2	1.25	1.96	**1.94**	**1.94**
I 3	2.23	**2.23**	**2.23**	**2.23**
I 4	2.54	**2.54**	**2.54**	**2.54**
I 5	1.63	**2.10**	2.13	2.12
I 6	3.00	**3.00**	**3.00**	**3.00**
I 7	0.50	-	2.98	**2.85**
I 8	0.15	-	**2.39**	2.46
I 9	0.59	-	2.97	**2.86**
I 10	0.53	-	2.78	**2.67**
I 11	0.34	-	**2.76**	**2.76**
I 12	1.02	-	2.89	**2.79**

Table 8.9: Comparison of the best bounds achieved with exact methods, simulated annealing, and the HAHA hyper-heuristic.

annealing approach reaches better mean costs for instances 1, 8, and 11. Comparing the best cost results per instance, we further see that HAHA is able to reach equal or better results than simulated annealing for all instances except instance 8. This is further illustrated in Table 8.9, where the best bounds for all instances achieved by exact methods, simulated annealing and HAHA are summarized.

Table 8.9 displays in Columns 2–5 from left to right, the best lower bound achieved by exact methods (LB), the best upper bounds achieved by exact methods (Exact), the best upper bounds achieved with simulated annealing (SA), and the best upper bounds

achieved with HAHA.

We conclude that the evaluated hyper-heuristics which utilize the proposed low-level heuristics could be successfully used to reach equal or improved results compared to simulated annealing for the large majority of the instances. Thereby, the methods could provide improved upper bounds for four realistic benchmark instances.

CHAPTER 9

Solver-Independent Modeling for Workforce Scheduling Problems

Besides the task of scheduling jobs onto machines that we have investigated in previous chapters, in many real-life factories and other areas there further arises the need of finding efficient workforce schedules. In this chapter we propose and evaluate a solver-independent model that can be utilized as an exact approach to a variant of the workforce scheduling problem. This particular variant can appear in a wide area of application domains such as industrial manufacturing, transportation or the medical sector.

At the beginning of this chapter we provide some background on the problem and provide a problem description as well as an overview of related work. Afterwards, we introduce a solver-independent high-level model for workforce scheduling which does not rely on solver specific low-level encodings and can therefore be directly used with solver-independent modeling languages such as MiniZinc [NSB+07]. However, to provide further insight on how the solver-independent model can be automatically processed for use with state-of-the-art CP and MIP solvers, we additionally describe the translation to low-level CP and MIP encodings. Finally, at the end of the chapter we extensively evaluate our models with the use of state-of-the-art CP and MIP solvers on a well known set of benchmark instances from the literature.

9.1 Background

Employee scheduling problems arise whenever there is a need for effective management and distribution of workforce over periods of time. Unfortunately, even basic variants have been proven to be NP-hard [CL96], and therefore it is a challenging task to find optimized workforce rosters in reasonable time.

Table 9.1: A two-week schedule for seven employees.

Employee	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su
1	*D*	*D*	*D*	*N*	—	—	—	*N*	*N*	*N*	—	—	*D*	*D*
2	*D*	*D*	*D*	—	—	*D*	*D*	*D*	—	—	*N*	*N*	*N*	*N*
3	*D*	—	—	*N*	*N*	*N*	*N*	—	—	—	—	*D*	*D*	*D*
4	—	—	—	—	*D*	*D*	*D*	*N*	*N*	*N*	*N*	*N*	—	—
5	*D*	*D*	*D*	*D*	*D*	—	—	*D*	*D*	*D*	*D*	*D*	—	—
6	*N*	*N*	*N*	*N*	*N*	—	—	*D*	*D*	*D*	—	—	*D*	*D*
7	*N*	*N*	*N*	—	—	*D*	*D*	—	—	—	—	*D*	*D*	*N*

The main aim of workforce scheduling (WS) is to determine a unique schedule for each employee over a fixed time period subject to shift constraints and working time constraints. The problem variant we study in this chapter tackles a standard problem arising in many areas (e.g. Nurse Rostering), where we have to schedule multiple different employees subject to different constraints. In our evaluation we concentrate on the employee scheduling instances introduced by [CQ14]. According to the authors, those instances were designed to describe realistic and challenging staff scheduling problems while still being straightforward to use. The included scheduling periods range from one week up to one year, requiring up to 180 employees and 32 shift types to be assigned. Furthermore, these instances have been used as a benchmark for many related publications in the literature (e.g. [BC14, DMW19, MW17, RAL17, Sme18]). Table 9.1 shows an example WS schedule for seven employees using night (N) and day (D) shift assignments over a scheduling horizon of two weeks.

We define two solver-independent model variants for WS, and evaluate them using a CP and a Mixed-Integer Programming (MIP) solver. One that is rather direct, where each constraint is separately stated and a second one that attempts to model as many as possible of the regulations using a single *regular* constraint ([Pes04]), which models the regulations as a deterministic finite automaton. We compare the variations of the models experimentally using the two solvers, and explore good search strategies to be used with the CP solver. To evaluate our models, we experimented with the well known standard benchmarks that have been used to evaluate state-of-the-art methods ([CQ14, BC14]). We show that our solver-independent models are able to achieve competitive results and can reduce the optimality proof runtime for several instances.

We first give an informal description of the workforce scheduling problem, before we discuss related literature in Section 9.3 and afterwards provide a direct solver-independent model in Section 9.4. Then, in Section 9.5 we give preliminaries on global constraints that we later use in Section 9.6 to introduce an alternative model using such constraints. Further details on how the proposed solver-independent models can be automatically translated to CP and MIP encodings are given in Section 9.7. Finally, we present computational results in Section 9.8.

9.2 Problem Description

To describe the requirements of workforce scheduling we assume the following parameters:

- A set of employees E.
- The number of days in a week w which is typically seven.
- A number of days in the scheduling horizon h and a set $D = \{1, \ldots, h\}$. We assume that h is a multiple of w, that is the length of the schedule is exactly $\frac{h}{w}$ full weeks.
- A set of all shift types T. We include another (artificial) shift type O representing a day off, and define $T^+ = T \cup \{O\}$. The length in minutes of each type t is specified as l_t.

We assume that schedules begin with the first normal working day of the week, and we is the length of the weekend, so the last we days of each week of length w are considered as weekend days. In the variant we investigate in this chapter we have $we = 2$ for 7-day weeks $w = 7$.

9.2.1 Single Employee Considerations

The principal decisions of workforce scheduling are assignments of shifts (including days off) to each employee e for each day d.

Common restrictions on the sequence of shifts for each employee are:

- A set of days that employee e must be assigned a day off: O_e
- A minimum number of minutes that employee e has to work in total: $b_e^{\min}$
- A maximum number of minutes that employee e can work in total: $b_e^{\max}$
- A maximum number of shifts of type t that employee e can work in total: $m_{e,t}^{\max}$
- A minimum number of consecutive working shifts that employee e must work: $c_e^{\min}$
- A maximum number of consecutive working shifts that employee e may work: $c_e^{\max}$
- A minimum number of consecutive days off that must be assigned to employee e: $o_e^{\min}$
- A maximum number of weekends that employee e may work on: $a_e^{\max}$
- A set of forbidden sequences of two consecutive shifts $(t_1, t_2) \in F$, that is, directly after a shift of type t_1 the employee cannot be assigned a shift of type t_2

Another consideration, not a constraint, is the notion of preferred shifts. We consider two kinds of preferences:

- A positive preference for employee e on day d to take shift type t is defined by a penalty $q_{e,d,t}$ which accrues if this is not the case.
- A negative preference for employee e on day d to *not* take shift type t is defined by a penalty $p_{e,d,t}$ if the employee is assigned that shift type on that day.

9.2.2 Group Considerations

The principal group restriction of workforce scheduling is that we have a desired number of shifts $u_{d,t}$ of each shift type t for each day d. In the case of the workforce scheduling problem variant we investigate in this chapter these restrictions are *soft* where we accrue an *undercover* penalty $v_{d,t}^{\min}$ for each employee under $u_{d,t}$ assigned to shift type t on day d, and we accrue an *overcover* penalty $v_{d,t}^{\max}$ for each employee over $u_{d,t}$ assigned to shift type t on day d.

9.3 Related Work

The workforce scheduling variant we model in this chapter was first described by [CQ14] where the authors also provide a number of challenging and realistic benchmark instances. The same authors propose an integer programming (IP) formulation in addition with a branch and price algorithm from [BC14] to find optimal solutions to small to medium-sized instances. An alternative IP formulation that uses network flows to model the minimum and maximum consecutive shifts and day off constraints has been proposed by [Sme18]. This approach has been successfully used to reduce the required runtime to prove optimality for the smaller instances. In [DMW19] a Max-SAT formulation is provided and several SAT encodings that can be used to approach the problem using maximum satisfiability solvers are described.

Several metaheuristic solution methods have been proposed to approach the problem. An approach from [BC14] which is based on variable neighborhood search and ejection chains was used to provide the first solutions for the largest instances. The upper bounds for the largest instances could be further improved by hybrid approaches [MW17, RAL17]. Recently, another approach based on simulated annealing has been applied [KM20]. In [KLSD18] the authors study how relaxed formulations of the problem influence the solution quality of a large neighborhood search based approach.

To the best of our knowledge we investigate for the first time an exact approach using CP based modeling for this particular variant of the workforce scheduling problem. However, as exact solution methods based on CP were extensively investigated for related nurse rostering problems in the past, we in the following give a short overview of previous literature on CP methods for nurse rostering.

One of the earlier constraint-based solution approaches to a real-life nurse scheduling problem was proposed in [DFM+95], where the aim is to find day to day shift assignments for each nurse considering several constraints such as the number of work hours per year, day-off requirements, and specific shift patterns on weekends. Another nurse scheduling problem variant imposing strict cover requirements and several soft constraints regarding shift patterns was investigated in [WHFP95]. The authors proposed a constraint model which they utilized with CP solving technology to tackle the problem.

Based on the experiences with nurse rostering problems that appear at german hospitals, an automatic rostering solution based on constraint optimization was studied in [MaH01].

The proposed approach integrated a branch-and-bound procedure together with local search to provide solutions that consider a variety of constraints such as rest time constraints and requirements regarding preferred sequences of shifts. A CP model and search strategy to solve staff scheduling problems appearing in the health-care area was studied in [BGP03]. The authors modeled several requirements regarding shift demands, workload, and feasible shift patterns with global constraints to efficiently solve a benchmark problem set originating from several hospitals with CP solvers.

Furthermore, a CP approach as well as a hybrid approach that combines CP and local search to solve a real-life hospital rostering problem that arises at the department of Neurology at the Udine University hospital was introduced in [CDGD06]. The work proposed a CP formulation to model a variety of domain-specific constraints which was then utilized by a CP solver to generate feasible solutions for the investigated problem instances. In [MBL09], an overview of modeling techniques for nurse rostering problems using soft global constraints was given, including several ways of expressing shift pattern requirements with deterministic finite automatons. Another variant of the nurse rostering problem was tackled using CP in [SCBM13]. This study modeled real-life problems that appear at chilean mid-size health care centers as constraint optimization problems and utilized CP solving technology in a series of computational experiments.

9.4 Direct Model

In this section we propose a direct solver-independent model that considers each of the previously described constraints individually.

The key decisions are:

- $S_{e,d} \in T^+$ for employee e which shift type is assigned for day d.

We now present the definition of each individual constraint:

Employees must have a day off on certain days.

$$S_{e,d} = O, \forall e \in E, d \in O_e \tag{9.1}$$

Minimum and maximum working time.

$$b_e^{\min} \leq \sum_{d \in D} l_{S_{e,d}}, \forall e \in E \tag{9.2}$$

$$\sum_{d \in D} l_{S_{e,d}} \leq b_e^{\max}, \forall e \in E \tag{9.3}$$

Note that the variable $l_{S_{e,d}}$ is indexed by the *variable* $S_{e,d}$ rather than a constant. This is realized using the `element` constraint (see next section on global constraints).

The maximum number of shifts for each type that can be assigned to an employee.

$$\sum_{d \in D} [S_{e,d} = t] \leq m_{e,t}^{\max}, \forall e \in E, t \in T \tag{9.4}$$

Note the use of reification here, the expression $S_{e,d} = t$ evaluates to 1 if in the solution $S_{e,d} = t$ and 0 otherwise.

Minimum consecutive working shifts.

$$(S_{e,d} = O \wedge S_{e,d+1} \neq O) \rightarrow S_{e,d+s} \neq O, \forall e \in E, s \in \{2, \ldots, c_e^{\min}\}, d \in \{1, \ldots, h-s\} \tag{9.5}$$

Maximum consecutive working shifts.

$$\sum_{j=d}^{d+c_e^{\max}} [S_{e,j} \neq O] \leq c_e^{\max}, \forall e \in E, d \in \{1, \ldots, h - c_e^{\max}\} \tag{9.6}$$

Minimum consecutive days off.

$$(S_{e,d} \neq O \wedge S_{e,d+1} = O) \rightarrow S_{e,d+s} = O, \forall e \in E, s \in \{2, \ldots, o_e^{\min}\}, d \in \{1, \ldots, h-s\} \tag{9.7}$$

Maximum number of working weekends.

$$\sum_{x \in \{1, \ldots, \frac{h}{w}\}} \exists_{s \in \{0, \ldots, we-1\}} [S_{e,xw-s} \neq O] \leq a_e^{\max}, \forall e \in E \tag{9.8}$$

Disallowed shift sequences.

$$S_{e,d} = t_1 \rightarrow S_{e,d+1} \neq t_2, \forall e \in E, d \in \{1, \ldots, h-1\}, (t_1, t_2) \in F \tag{9.9}$$

Shift Preferences The shift preferences objective is defined by

$$prefer = \sum_{e \in E} \sum_{d \in D} \sum_{t \in T} q_{e,d,t} [S_{e,d} \neq t] + p_{e,d,t} (S_{e,d} = t) \tag{9.10}$$

If there are no shift preferences we simply set $prefer = 0$.

Cover requirements. The cover requirements are defined by

$$\begin{aligned} y_{d,t} &= \max(u_{d,t} - \sum_{e \in E} [S_{e,d} = t], 0), \forall d \in D, t \in T \\ z_{d,t} &= \max(\sum_{e \in E} [S_{e,d} = t] - u_{d,t}, 0), \forall d \in D, t \in T \\ cover &= \sum_{d \in D} \sum_{t \in T} (v_{d,t}^{\min} y_{d,t} + v_{d,t}^{\max} z_{d,t}) \end{aligned} \tag{9.11}$$

Objective The objective of the direct model is simply

$$\text{minimize } \mathit{prefer} + \mathit{cover} \tag{9.12}$$

9.5 Global Constraints

Global constraints ([vHK06]) capture complex relationships between variables, typically by explicitly representing a subproblem. For example, the *alldifferent* global constraint takes as input a set of integer variables and imposes that no two variables may take the same value. Global constraints serve several purposes: First, they simplify the modeling process through more expressive constraints. Second, they explicitly state the relationship between variables, which is in some cases beneficial as specialized algorithms may be developed that provide stronger pruning power compared to *decomposing* the global constraint into a series of simpler constraints that capture the same meaning. In this way, the structure of the subproblem is preserved and may be exploited in CP solvers. Furthermore, global constraints enable the implementation of solver specific decompositions.

In the following text, we define global constraints that are relevant for this chapter, and discuss their use within the model to improve the solving process. We note that only the necessary syntax of the constraints will be described without detailing the underlying algorithms used within solvers.

9.5.1 Global Constraint Definitions

The `element` constraint

models accessing entries of an array using variables for the index rather than constants. This is written as follows:

$$\texttt{element}(y, B, x), \tag{9.13}$$

where y is an integer variable, B is an array of values, and x is a numeric variable. The `element` constraint enforces that x takes the y-*th* value of B, i.e., $x = B_y$. For example, given an `element` constraint $\texttt{element}(y, B = [5, 4, 10, 3, 5], x)$, if during search the *variable* assignment $y = 3$ takes place, the constraint enforces the assignment $x = B_y = B_3 = 10$.

The `global_cardinality` constraint

specifies that certain values must appear a given number of times among a set of variables. The syntax is as follows:

$$\texttt{global\_cardinality}(X, C, E), \tag{9.14}$$

where X and E are arrays of integer variables, and C is an array of integers (constants). The constraint requires that the value C_i must be present E_i times among the variables X_i. Formally, let $[x = a]$ represent the expression that evaluates to *one* if variable $x = a$ and *zero* otherwise, $count(X, a) = [\{|x = a| : x \in X\}]$, then `global_cardinality` constrains that $\forall i \in \{1, 2, \ldots, |C|\}$ $count(X, C_i) = E_i$.

The `global_cardinality_low_up`

constraint generalizes the previous global constraints by allowing an upper and lower bound on the number of times certain values may be assigned to variables. The syntax is as follows:

$$\texttt{global\_cardinality\_low\_up}(X, C, L, U) \tag{9.15}$$

Here X is an array of integer variables, and C, L, and U are arrays of integers. The `global_cardinality_low_up` constraint imposes that, for each i, the value C_i may appear at least L_i times and at most U_i times among the variable assignments of the variables $x \in X$. Formally, as before, let $|x = a|$ represent the expression that evaluates to *one* if variable $x = a$ and *zero* otherwise, $count(X, a) = |\{[x = a] : x \in X\}|$, then `global_cardinality_low_up` constrains that $\forall i \in \{1, 2, \ldots, |C|\}$ $L_i \leq count(X, C_i) \leq U_i$.

For example, for $\texttt{global\_cardinality\_low\_up}([x_1, x_2, x_3], [0, 1, 3], [0, 2, 1], [0, 3, 1])$ effectively states that the variables $x_i \neq 0$, at least two $x_i = 1$, and exactly one $x_i = 3$.

The `regular` constraint

defines constraints in the form of automata, i.e., state-transition tables. Traditionally, automata are defined over strings. In this case, each character of the string is represented by an integer variable. The constraint is considered violated if the string given by the integer variables is not accepted by the specified automata, and satisfied otherwise. The advantage of using the `regular` constraint is first, the ease of modeling complex constraints, and second, it has been shown that the representation is convenient for translation into integer programming. In the following sections we use the `regular` constraints by specifying the automata and make use of MiniZinc to translate the automata into the specification required by the target solver.

9.6 Modeling with Global Constraints

This section describes how shift requirements can be represented by global cardinality constraints and further presents how consecutive shift constraints can be modeled together with the use of deterministic finite automata.

9.6.1 Cardinality Constraints

Several parts of the workforce scheduling problem are counting the number of occurrences of various shift types. *Global cardinality* constraints can reason about the counts simultaneously. As this can give solving advantages, we use them to model shift- and cover-requirements.

Shift Requirements using Cardinality Constraints

We can enforce lower and upper bounds on the shifts of each type simultaneously, replacing Equation (9.4) by the following equation:

$$\texttt{global\_cardinality\_low\_up}([S_{e,d}|d \in D], T, [0|t \in T], [m^{\max}_{e,t}|t \in T]), \forall e \in E \tag{9.16}$$

Cover Requirements using Cardinality Constraints

The cover requirements can be modeled directly using *global cardinality* constraints. We use the cardinality constraint to count the number of shifts of type t on day d with auxiliary variables $C_{d,t}$ as

$$\texttt{global\_cardinality}([S_{e,d}|e \in E], T, [C_{d,t}|t \in T]), \forall d \in D \tag{9.17}$$

and redefine the soft cover penalty (Equation 9.11) as

$$cover = \sum_{d \in D} \sum_{t \in T} (v^{\max}_{d,t} \cdot \max(0, (C_{d,t} - u_{d,t})) + v^{\min}_{d,t} \cdot \max(0, (u_{d,t} - C_{d,t}))) \tag{9.18}$$

9.6.2 Modeling Shift Sequence Constraints with Finite Automata

Given that the principle constraints of the problem are defined on the possible shift sequences that an employee can take, finite automata are an obvious approach to model allowable shift sequences. Therefore, in this section we build separate automata for each employee e that can be used within `regular` global constraints.

We note that similar constraints for a related rotating workforce scheduling problem have been modeled using deterministic finite automata in [MSS18]. In the following we adapt the model from [MSS18] to be used for the workforce scheduling problem we study in this chapter.

The states of each automaton keep track of the last encountered shift type to enforce the forbidden shift sequence constraint. Additionally, information about how many consecutive shifts or day off assignments have been recently processed are encoded in the states to enforce the min/max consecutive shifts and min consecutive days off constraints. The idea is that the automaton only accepts a given shift sequence if all sequence-dependent constraints (Equations 9.5, 9.6, 9.7, 9.9) are fulfilled and thereby replaces these constraints from the direct model.

We now model a single automaton for each employee that will process the sequence of shift and day off assignments for the employee. For each employee $e \in E$ we define an automaton A_e with $Q = 1 + (c_e^{\min} - 1) \cdot |T| + o_e^{\min} + c_e^{\max} \cdot |T|$ states. In the following we enclose states with brackets to avoid confusion with other parameters. Each automaton defines an artificial start state $[S]$ as well as day off states $[O_i], i \in \{1, \ldots, o_e^{\min}\}$ and shift states $[N_{t,i}], t \in T, i \in \{1, \ldots, c_e^{\max}\}$. Additionally, we need for each shift type special states $[N^*_{t,i}], t \in T, i \in \{1, \ldots, c_e^{\min} - 1\}$ to handle corner cases with the minimum consecutive shift assignments constraint at the beginning of the scheduling horizon. All states except starting state $[S]$ are accepting states.

State $[O_i]$ represents that we have taken the last i shifts as days off while State $[N_{t,i}]$ represents that the last i shifts have been working shifts with the last shift of type t. Similar to state $[N_{t,i}]$, state $[N^*_{t,i}]$ represents that the last i shifts have been working shifts with the last shift of type t. However, a state $[N^*_{t,i}]$ is only entered in a block of working days that is scheduled at the immediate beginning of the scheduling horizon ($*$ states are necessary to ignore the minimum consecutive working days constraint at the beginning of the scheduling horizon).

We define the transition function d as follows (any missing transitions lead to a non-accepting error state):

- $[S]$: If a day off (O) is encountered go to state $[O_{o_e^{\min}}]$, on a shift $t \in T$ and $c_e^{\min} > 1$ go to state $[N^*_{t,1}]$. If $c_e^{\min} = 1$ go to state $[N_{t,1}]$ instead.
- $[N^*_{t_1,i}]$: If O is encountered go to $[O_1]$, on a shift $t_2 \in T$ go to either $[N^*_{t_2,i+1}]$ if $i + 1 < c_e^{\min}$ or otherwise to $[N_{t_2,c_e^{\min}}]$ unless $(t_1, t_2) \in F$ (if $(t_1, t_2) \in F$ neither a transition to $[N^*_{t_2,i+1}]$ nor a transitiion to $[N_{t_2,c_e^{\min}}]$ is possible).
- $[O_i]$: If O is encountered go to $[O_{i+1}]$ if $i + 1 \leq o_e^{\min}$ otherwise go to $[O_{o_e^{\min}}]$. On a shift $t \in T$ go to $[N_{t,1}]$ unless $i < o_e^{\min}$.
- $[N_{t_1,i}]$: If O is encountered go to $[O_1]$ unless $i < o_e^{\min}$. On a shift $t_2 \in T$ go to $[N_{t_2,i+1}]$ unless $i = c_e^{\max} \vee (t_1, t_2) \in F$.

Figure 9.1 shows an example automaton for an employee e and shifts $T = \{E, L\}$ where $c_e^{\min} = 2$, $c_e^{\max} = 4$, $o_e^{\min} = 2$, $F = \{(L, E)\}$. For simplicity, we write the shift accepting states as $[E_i]$ and $[L_i]$ instead of $[N_{E,i}]$ and $[N_{L,i}]$ in the example.

9.7 Translation for Solving

The high-level solver-independent models we have specified in previous sections can be directly used with solver-independent modeling languages such as MiniZinc [NSB+07]. Such high-level modeling tools automatically translate the solver-inpendent model into low-level encodings which are then solvable by an underlying solver technology. In this section we describe possible low-level translations for CP and MIP solvers.

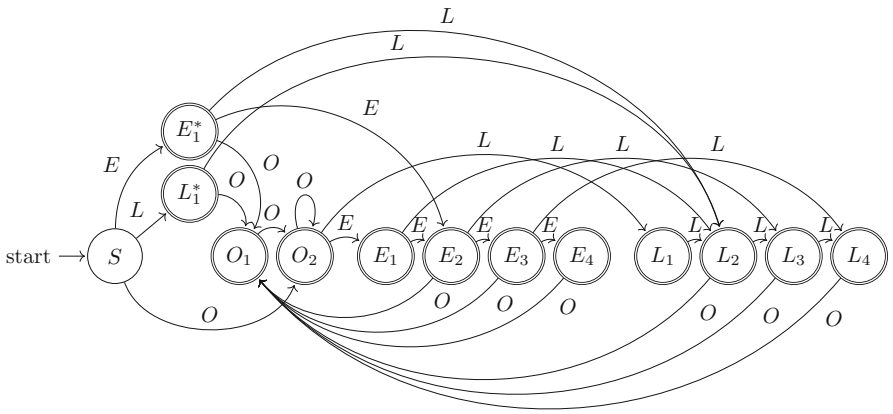

Figure 9.1: Example automaton for an employee e and shifts $T = \{E, L\}$ where $c_e^{\min} = 2$, $c_e^{\max} = 4$, $o_e^{\min} = 2$, $F = \{(L, E)\}$. An example sequence that would be accepted by this automaton would be for instance $EEELLLOOLLLL$ whereas $LLLOOEOOLE$ would be forbidden due to the minimum consecutive shift assignments and forbidden shift successor constraints.

9.7.1 Constraint Programming

The high-level model variants we have specified above are almost directly solvable by CP solvers, therefore we only need to examine the low level form of a number of equations.

Equations (9.2) and (9.3) use the ability to look up an array by a given variable index. The low level translation of Equation (9.2) for one $e \in E$ is:

$$\begin{aligned} &\texttt{element}(S_{e,d}, l, wt_d), d \in D \\ &wt = \sum_{d \in D} wt_d \\ &b_e^{\min} \leq wt \end{aligned} \tag{9.19}$$

For this translation we use an integer auxiliary variable wt which captures the total working time and auxiliary variables $wt_d, \forall d \in D$ that capture the working time on each individual day in the schedule. The translation for Equation (9.3) is done similarly.

Many constraints use *reification* of constraints to express the constraint. A constraint of the form $b \leftrightarrow (S_{e,d} = t)$ associates a 0/1 variable b with the expression. If the expression holds b is 1, otherwise it is 0. Note the use of common sub expression eliminations means that we will only create one 0/1 variable for each possible reified expression.

Equation (9.6) for one $e \in E, t \in T$ is for example translated as

$$\begin{gathered} b_j \leftrightarrow (S_{e,j} \neq O), d \leq j \leq d + c_e^{\max} \\ ss = \sum_{j=d}^{d+c_e^{\max}} b_j \\ ss \leq c_e^{\max} \end{gathered} \tag{9.20}$$

Implication constraints are translated into clauses. For example Equation (9.7) for some $e \in E$, $s \in \{2, \ldots, o_e^{\min}\}$, $d \in \{1, \ldots, h - s\}$ is translated as

$$\begin{gathered} b_1 \leftrightarrow S_{e,d} = O \\ b_2 \leftrightarrow S_{e,d+1} = O \\ b_3 \leftrightarrow S_{e,d+s} = O \\ b_1 \vee \neg b_2 \vee b_3 \end{gathered} \tag{9.21}$$

Existentially quantified expressions are also translated using clauses. For example Equation (9.8) for some $e \in E$ is translated as

$$\begin{gathered} b_d \leftrightarrow S_{e,d} \neq O, d \in D \\ eb_x \vee \neg b_s, x \in \{1, \ldots, \frac{h}{w}\}, d \in \{xw - we + 1, \ldots, xw\} \\ \neg eb_x \vee \bigvee_{d \in \{xw - we + 1, \ldots, xw\}} b_s \\ ss = \sum_{x \in \{1, \ldots, \frac{h}{w}\}} eb_x \\ a_e^{\max} \geq ss \end{gathered} \tag{9.22}$$

Cardinality constraints and automata constraints are directly representable in CP.

9.7.2 Integer Programming

In the integer programming model the decision variables $S_{e,d}$ are replaced by 0/1 variables $x_{e,d,t} \leftrightarrow (S_{ed} = t), t \in T$, very similar to how the reified expressions are treated in CP. The decision variables $S_{e,d}$ disappear from the model altogether. Note that we do not introduce $x_{e,d,O}$ to represent the fact that employee e on day d has a day off, but instead represent this implicitly as $1 - \sum_{t \in T} x_{e,d,t}$.

In order to assert that each person only takes a single shift on any day, the following constraints are added to the model:

$$\sum_{t \in T} x_{e,d,t} \leq 1, e \in E, d \in D \tag{9.23}$$

Equations (9.2) and (9.3) use the $x_{e,d,t}$ variables to encode the `element` constraint. The low level translation for one $e \in E$ then is

$$\begin{aligned} wt &= \sum_{d \in D} \sum_{t \in T} x_{e,d,t} \cdot l_t \\ b_e^{\min} &\leq wt \leq b_e^{\max} \end{aligned} \tag{9.24}$$

Implications are converted to inequalities. For example Equation (9.7) for some $e \in E$, $s \in \{2, \ldots, o_e^{\min}\}$, $d \in \{1, \ldots, h - s\}$ is translated as

$$(1 - x_{e,d,O}) + x_{e,d+1,O} \leq 1 + x_{e,d+s,O} \tag{9.25}$$

Existential quantifications are also translated to inequalities, though we need a new variable to encode the result of the existential quantification. For example Equation (9.8) for some $e \in E$ is translated as

$$\begin{aligned} eb_x \geq (1 - x_{e,d,O}), \forall x \in \{1, \ldots, \frac{h}{w}\}, d \in \{xw - we + 1, \ldots, xw\} \\ eb_x \leq \sum_{d \in \{xw - we + 1, \ldots, xw\}} (1 - x_{e,d,O}) \\ ss = \sum_{x \in \{1, \ldots, \frac{h}{w}\}} eb_x \\ a_e^{\max} \geq ss \end{aligned} \tag{9.26}$$

Global cardinality constraints are encoded using sums over the $x_{e,d,t}$ variables. For example, Equation (9.16) for $e \in E$ produces *exactly the same* translation as that of Equation (9.4).

Automata constraints are encoded to integer programming using network flow encodings that make use of the $x_{e,d,t}$ variables. For details on these network flow encodings see e.g. [CGR07, BSTW16].

Alternative Network Flow Encoding for Series Constraints

An alternative network flow based encoding of the minimum and maximum consecutive working days/day off constraints (Equations (9.5), (9.6), (9.7)) was recently proposed by [Sme18]. The encoding has similarities to the network flow encoding of the automata constraints, but does not encode the disallowed shift sequence constraints. However, it uses fewer arcs and nodes to encode the consecutive shift/day off constraints. In our experimental evaluation we examine this encoding as an alternative to the automata based constraint encoding.

9.7.3 Working Weekends Reformulation

The MIP translations of the solver-independent model from Section 9.7 are based on previously proposed models from [CQ14] and a network flow reformulation from [Sme18]. However, we slightly reformulate the working weekend constraint.

Using additional binary auxiliary variables $k_{e,i}, \forall e \in E, i \in \{1, \ldots, \frac{h}{w}\}$, we can reformulate the MIP translation of Equation (9.8) as:

$$\begin{gathered} k_{e,i} \geq \sum_{t \in T} x_{e,d,t}, \quad \forall i \in \{1, \ldots, \frac{h}{w}\}, d \in \{xw - we + 1, \ldots, xw\}, e \in E \\ \sum_{i \in \{1, \ldots, \frac{h}{w}\}} k_{e,i} \leq a_e^{\max}, e \in E \end{gathered} \tag{9.27}$$

This is different to the existing formulation of the working weekends constraint, as multiple constraints are defined for each weekend. Instead of Equation 9.27 previous formulations from [CQ14] and [Sme18] have used:

$$\begin{gathered} k_{e,i} \leq \sum_{t \in T, d \in \{xw - we + 1, \ldots, xw\}} x_{e,d,t} \leq 2k_{e,i} \quad \forall i \in \{1, \ldots, \frac{h}{w}\}, e \in E \\ \sum_{i \in \{1, \ldots, \frac{h}{w}\}} k_{e,i} \leq a_e^{\max}, e \in E \end{gathered} \tag{9.28}$$

9.8 Computational Results

In this section, we provide the results of our experimental study. We implemented all models using the solver-independent MiniZinc constraint modeling language ([NSB+07]) and evaluated the results with state-of-the-art CP and integer programming solving technology.

For the CP model we simply implemented the solver-independent high-level model and thereby utilized the automatic translation into low-level CP encodings provided by MiniZinc. However, for the MIP model we did not utilize the linearization library of MiniZinc to automatically linearize the high-level constraint model for use with MIP solvers in our final experiments. Instead, we manually performed the translations into a linear model that we described in previous sections, as some of the problem specific encodings are not included in the MiniZinc library and early experiments indicated that it was more efficient to manually encode the constraints on the evaluated benchmark instances.

All benchmark experiments on the workforce scheduling problem have been performed using an Intel Xeon E5345 2.33 GHz CPU with 48 GB RAM. We evaluated our models with the use of the 24 benchmark instances from [CQ14], which according to the authors describe realistic workforce scheduling instances, and have been used as a benchmark in

many related publications. In the following, we first present the results achieved with CP solving technology and afterwards give the detailed results produced with a MIP solver. For the CP experiments we used Chuffed 0.10.3 ([Chu11]), whereas the MIP results have been achieved with Gurobi 8.1.0 ([GO20]).

9.8.1 Search Strategies

We define several variable and value selection strategies that we use in our experiments:

Variable Selection Variable selection is critical for reducing the search space for any combinatorial problem. We need to balance the criteria of driving quickly towards failure, with getting the most possible inference from the solver. The key decisions of the model are the schedule variables $S_{e,d}$. We define our variable selection over these variables unless stated otherwise. Ties are broken by input order.

- **default**: The solver's default variable selection strategy.
- **rnd**: Randomly select a variable[1].
- **worker**: Select all variables in chronological order, or in other words assign the complete schedule for each employee before continuing with the next employee (i.e.: $S_{1,1}, \ldots, S_{1,h}, S_{2,1}, \ldots, S_{2,h}, \ldots, S_{|E|,h}$).
- **day**: Process one day after the other and assign shifts for each employee on this day before moving on to the next day (i.e. $S_{1,1}$, …, $S_{|E|,1}$, $S_{2,1}$, …, $S_{|E|,2}$, …, $S_{|E|,h}$).
- **wd**: First process assignments for day one (i.e. $S_{1,1}, \ldots, S_{|E|,1}$) and then continue with **worker**.
- **ff**: Use a first fail strategy (choose variables with the smallest domains first).

Value Selection

- **default**: The solver's default value selection strategy.
- **val1**: First assign an off shift, then assign all shifts chronologically.
- **val2**: First assign all shifts chronologically, then assign an off shift.
- **val3**: First assign an off shift, then assign all shifts in reverse chronological order.
- **val4**: First assign all shifts in reverse chronological order, then assign an off shift.

[1]The random variable selection is currently only supported with Chuffed in MiniZinc. Therefore, we evaluated this selection strategy only with Chuffed and could not use it to specify branching priorities for Gurobi.

Search Strategy	Direct Model	Automaton Model
solver default	10	9
day/val1	10	9
day/val2	17	16
day/val3	9	10
day/val4	13	12
ff/val1	11	10
ff/val2	11	10
ff/val3	10	11
ff/val4	12	11
rnd/val1	10	10
rnd/val2	9	10
rnd/val3	10	9
rnd/val4	9	10
wd/val1	11	9
wd/val2	11	12
wd/val3	11	11
wd/val4	12	12
worker/val1	10	10
worker/val2	10	12
worker/val3	11	11
worker/val4	12	11

Table 9.2: This table shows the number of instances for which a feasible solution could be found within a time limit of 10 minutes. The first column denotes which variable and value selection strategies have been used, while the second and third column display the number of acquired feasible solutions using the direct and the automaton based CP model.

Constraint Programming Results

In a first set of experiments we evaluated all possible combinations of the previously described variable and value selection heuristics (a total of 20 combinations without the default strategies) within a 10 minute time limit. Additionally, we activated the free search parameter for Chuffed which allows the solver to switch between the given search strategy and the activity based search.

Table 9.2 displays the number of instances for which a feasible solution could be found within 10 minutes using the direct model as well as the model that uses automata to capture shift constraints (both models use global cardinality constraints to state cover and shift requirements). As we can see both formulations give similar results and there is no clear winner for all search strategies.

As no variable/value selection stood out in the results of the initial experiments, we decided to perform additional experiments with those search strategy and model combinations that were able to produce at least 12 feasible solutions within 10 minutes.

Instance	D+day/val2	D+day/val4	D+ff/val4	D+wd/val4	D+worker/val4
I1	**607***	**607***	**607***	**607***	**607***
I2	2262	**957**	1670	1152	1064
I3	1943	1538	1454	2658	**1368**
I4	2801	2774	**2583**	3082	2871
I5	3914	4058	4151	3807	3874
I6	6608	6666	7082	7307	6384
I7	6672	6670	6764	**5955**	6401
I8	11479	**11295**	12215	11580	11888
I9	8108	9242	8156	8656	7927
I10	17539	17774	16731	18327	17358
I11	45147	30815	43244	45364	45469
I12	—	—	—	—	—
I13	—	—	—	—	—
I14	26969	—	—	—	—
I15	**38054**	—	—	—	—
I16	16758	18217	16987	29505	17606
I17	**66066**	—	—	—	—
I18	**43340**	47741	—	48800	48904
I19	**82128**	—	—	—	—
I20	**271968**	289287	—	—	—
I21	—	—	—	—	—
I22	—	—	—	—	—
I23	—	—	—	—	—
I24	—	—	—	—	—

Table 9.3: This tables shows the best results achieved for each instance using the direct CP model with Chuffed and selected search strategies. A * denotes proven optimal results. Results formatted in boldface highlight the overall best results achieved using Chuffed (see also Table 9.4).

Tables 9.3 and 9.4 present the final results achieved with Chuffed and the selected search strategies within a time limit of one hour.

The results show that among the evaluated search strategies the day variable selection strategy combined with the val2 value selection strategy is the most robust in our experiments as it can provide feasible solutions for 18 out of the 24 instances. Furthermore, this search strategy produced the best results in our CP experiments for 7 of the instances. However, for some problem instances other search strategies were able to produce better results. The model including the regular constraint produced the overall best results for 8 of the instances in our experiments, while experiments with the direct model produced 5 overall best results. For 6 of the instances both the regular- and the direct model produced the best results.

Integer Programming Results

In the literature two MIP models have been proposed to solve the workforce scheduling benchmark instances from [CQ14]: A direct formulation was described by [CQ14] (*Di-*

Instance	R+day/val2	R+day/val4	R+wd/val2	R+wd/val4	R+worker/val2
I1	**607***	**607***	**607***	**607***	**607***
I2	1054	1049	1150	1151	1058
I3	1874	1646	1762	1648	1759
I4	2789	2882	3174	3420	2968
I5	**3510**	3522	4215	4928	3708
I6	**6013**	6702	7614	6578	6272
I7	6534	6272	6573	6764	6797
I8	12373	11608	12373	12383	11861
I9	9138	8694	**7896**	8654	8760
I10	16968	17232	17622	17669	**16466**
I11	22774	23918	25537	26281	**21124**
I12	—	**30151**	31662	—	—
I13	—	—	—	—	—
I14	16195	**13336**	—	18199	17240
I15	**38054**	—	—	—	—
I16	14698	15169	14955	—	**14469**
I17	**66066**	—	—	—	—
I18	**43340**	47741	—	48901	—
I19	**82128**	—	—	—	—
I20	**271968**	289287	—	—	—
I21	—	—	—	—	—
I22	—	—	—	—	—
I23	—	—	—	—	—
I24	—	—	—	—	—

Table 9.4: This tables shows the best results achieved for each instance using the CP model that incorporates modeling the block series constraints as a regular constraint with the chuffed solver and selected search strategies. A * denotes proven optimal results. Results formatted in boldface highlight the overall best results achieved using chuffed (see also Table 9.3).

rect) and a network flow reformulation of the minimum/maximum consecutive working shift constraints was proposed by [Sme18] (*NetworkFlow*). We experimentally compare the existing models with our direct solver-independent model (*NewDirect*), the solver-independent model using the automata based global constraints (*NewAutomata*), and the solver-independent model that uses a network flow reformulation that is similar to the one from [Sme18] (*NewNetworkFlow*). Note that the main difference between the original formulations (*Direct*, *NetworkFlow*) and the formulations from this work (*NewDirect*, *NewNetworkFlow*) lies in the reformulation of the working weekends constraint as described in Section 9.7.3.

We implemented all five formulations and conducted experiments for instances 1–20 with Gurobi 8.1.0 ([GO20]) (Initial experiments with instances 21–24 caused out of memory exceptions on our benchmark machine).

In addition to the default search strategy, Gurobi supports partial programming of the search process using branching priorities. Therefore, we first evaluated all supported

Search Strategy	Direct	NewDirect	NetworkFlow	NewNetworkFlow	NewAutomata
solver default	12	10	12	12	5
day/val1	13	9	10	11	4
day/val2	9	9	9	8	5
day/val3	10	11	9	9	4
day/val4	13	10	11	11	4
ff/val1	12	10	10	11	4
ff/val2	9	8	9	8	4
ff/val3	10	10	9	8	4
ff/val4	13	9	10	11	5
wd/val1	13	9	10	11	6
wd/val2	9	9	9	8	4
wd/val3	10	10	9	8	5
wd/val4	13	9	11	11	4
worker/val1	13	9	11	11	4
worker/val2	9	9	9	8	4
worker/val3	10	10	9	10	4
worker/val4	15	11	13	14	4

Table 9.5: This table presents the number of best cost solutions found using the same MIP model with different branching strategies for Instances 1–20. Column 1 denotes the evaluated variable and value selection strategy, while columns 2–5 display the number of achieved best cost solutions.

variable- and value selection combinations from Section 9.8.1 using instances 1–20 using a 10-minute runtime limit in an initial set of experiments. Table 9.5 gives an overview on the number of best cost solutions produced with the different search strategies within a time limit of 10 minutes.

We can see in the results shown in Table 9.5 that the *worker/val4* strategy produced best results for the most instances when used with the *Direct*, *NetworkFlow*, and *NewNetworkFlow* MIP models, whereas the *NewAutomata* model produced the most best-cost solutions using the *wd/val1* strategy. Therefore, we evaluated these branching strategies in addition to the solver's default branching strategy in our final experiments with these models.

The *NewDirect* model achieved best cost solutions for 11 instances with both the *day/val3* and *worker/val4* branching strategy in the initial set of experiments. In this case we decided to evaluate the *day/val3* strategy in addition to the default branching strategy in the final experiments, as it produced on average a better solution quality than strategy *worker/val4* in the initial set of experiments.

The final experiments were then conducted using the selected branching strategies plus the default branching strategy of Gurobi under a runtime limit of one hour. Table 9.6 displays the final results achieved with the direct MIP formulations.

We can see in the results that overall the *Direct* and *NewDirect* models deliver similar results. However, in some cases the *Direct* formulation produced better bounds than the *NewDirect* formulation and vice versa (e.g. instances 8, 13, 14, 17). Using a custom branching strategy seems to improve results with the *Direct* model in some cases (e.g.

	Direct			NewDirect			Direct+worker/val4			NewDirect+day/val3		
Instance	LB	UB	Runtime	LB	UB	Runtime	LB	UB	Runtime	LB	UB	Runtime
I1	**607**	**607**	0.38	**607**	**607**	**0.35**	**607**	**607**	0.38	**607**	**607**	0.35
I2	**828**	**828**	6.85	**828**	**828**	4.30	**828**	**828**	4.45	**828**	**828**	7.20
I3	**1001**	**1001**	11.99	**1001**	**1001**	7.72	**1001**	**1001**	11.56	**1001**	**1001**	7.01
I4	**1716**	**1716**	201.01	**1716**	**1716**	172.82	**1716**	**1716**	414.07	**1716**	**1716**	432.69
I5	**1143**	**1143**	754.80	**1143**	**1143**	469.56	**1143**	**1143**	476.14	**1143**	**1143**	330.51
I6	**1950**	**1950**	218.30	**1950**	**1950**	405.31	**1950**	**1950**	513.09	**1950**	**1950**	324.96
I7	**1056**	**1056**	776.79	**1056**	**1056**	1042.95	**1056**	**1056**	1121.41	**1056**	**1056**	1315.08
I8	1284	1405	3600	1280	1500	3600	1279	1402	3600	1276	1407	3600
I9	406	**439**	3600	413	**439**	3600	**439**	**439**	**1165.44**	406	**439**	3600
I10	**4631**	**4631**	126.34	**4631**	**4631**	487.32	4631	**4631**	133.38	**4631**	**4631**	888.38
I11	**3443**	**3443**	50.08	**3443**	**3443**	**40.35**	**3443**	**3443**	53.41	**3443**	**3443**	216.27
I12	**4040**	4043	3600	**4040**	**4040**	1730.45	**4040**	**4040**	2690.11	**4040**	4141	3600
I13	1345	3094	3600	1343	**2957**	3600	1343	3308	3600	1336	—	3600
I14	1275	1279	3600	1275	1284	3600	**1278**	1280	3600	1274	1282	3600
I15	3811	5681	3600	3811	5695	3600	3812	6776	3600	3804	5585	3600
I16	3216	3322	3600	3215	3419	3600	3217	3228	3600	3213	3519	3600
I17	5734	5848	3600	5733	5757	3600	5734	5848	3600	5734	5848	3600
I18	4365	**4560**	3600	4355	4757	3600	4354	4661	3600	4350	4952	3600
I19	2923	4017	3600	2936	**3813**	3600	2935	5056	3600	2928	4208	3600
I20	4748	206698	3600	**4751**	209034	3600	4749	**202406**	3600	4735	—	3600
I21	—	—	3600	—	—	3600	—	—	3600	—	—	3600
I22	—	—	3600	—	—	3600	—	—	3600	—	—	3600
I23	—	—	3600	—	—	3600	—	—	3600	—	—	3600
I24	—	—	3600	—	—	3600	—	—	3600	—	—	3600

Table 9.6: Results produced using the original direct MIP formulation from [CQ14] (Direct) and the direct formulation we propose (NewDirect). Results formatted in boldface highlight the overall best lower bounds (LB), upper bounds (UB), and fastest proof times in seconds (Runtime) achieved using Gurobi (see also Tables 9.7 and 9.8). A — indicates that no solutions could be found within 1 hour.

instance 9, 20), however in case of the *NewDirect* model the branching strategy did not lead to improved results.

Table 9.7 displays the final results achieved with the network flow based MIP formulations.

We see in the results shown in Table 9.7 that for instances that could be solved to optimality, most of the times the *NewNetworkFlow* models provided the fastest proofs in our experiments. Regarding the achieved lower bounds and upper bounds, the *NetworkFlow* formulation and the *NewNetworkFlow* formulation overall produce similar results, although for some instances there are differences (e.g. instances 9, 18, 19). Further, the results show that using a custom branching strategy can sometimes improve the proof time (e.g. instances 3,4,7,16), but had only a little effect on the produced bounds in our experiments except for a few instances.

Table 9.8 displays the final results achieved with the solver-independent model that uses the automata based global constraints.

The results presented in Table 9.8 show that the models using automata global constraints could not produce competitive results when compared with the other evaluated models. Furthermore, the *NewAutomata* model could not benefit from a custom branching strategy in our final experiments.

Finally, Table 9.9 compares the overall best results achieved with existing MIP models from the literature (*Existing MIP*) with the best results achieved with the models from this work (*New MIP*).

The results show that the MIP models could successfully be used to reach 14 optimal solutions within a time limit of one hour. The *Direct* and *NetworkFlow* models from the literature could provide optimal results for 14 instances and further produce three additional upper bounds that could not be reached by the other models (Instances 15, 18 and 20). The *NewDirect* and *NewNetworkFlow* could provide optimal results for 13 instances and further achieved two additional upper bounds that could not be reached by the other models (Instances 13 and 19). Comparing optimality proof times, we can see that existing models provided the fastest proofs for 4 instances, whereas the *New MIP* models provided the fastest proofs for 10 instances.

The results further indicate, that for the majority of the instances the *Existing MIP* models and *New MIP* models produce bounds of similar quality. However, we can see that for solutions that could be solved to optimality the *New MIP* models provide faster proof times in 10 out of 14 cases. Furthermore, the results show that the models and custom search strategies from this work could produce improved lower and upper bounds in several cases (e.g. Instances 9, 13, 18, 19, 20).

As already mentioned in Section 9.3, the main difference between the existing models from the literature and models from this work lies in the reformulation of the working weekend constraint. To further investigate the effect of the reformulated working weekend constraint, we analyzed the optimal costs of the initial linear programming relaxations

	NetworkFlow			NewNetworkFlow			NetworkFlow+worker/val4			NewNetworkFlow+worker/val4		
Instance	LB	UB	Runtime	LB	UB	Runtime	LB	UB	Runtime	LB	UB	Runtime
I1	**607**	**607**	0.95	**607**	**607**	1.57	**607**	**607**	0.64	**607**	**607**	0.60
I2	**828**	**828**	2.59	**828**	**828**	**1.54**	**828**	**828**	2.28	**828**	**828**	1.88
I3	**1001**	**1001**	3.63	**1001**	**1001**	3.52	**1001**	**1001**	3.80	**1001**	**1001**	**2.61**
I4	**1716**	**1716**	35.72	**1716**	**1716**	34.96	**1716**	**1716**	40.77	**1716**	**1716**	**13.79**
I5	**1143**	**1143**	**18.95**	**1143**	**1143**	28.51	**1143**	**1143**	26.59	**1143**	**1143**	32.85
I6	**1950**	**1950**	20.72	**1950**	**1950**	**17.00**	**1950**	**1950**	22.49	**1950**	**1950**	28.56
I7	**1056**	**1056**	161.96	**1056**	**1056**	134.89	**1056**	**1056**	183.75	**1056**	**1056**	**127.30**
I8	**1297**	1301	3600	**1297**	**1300**	3600	1296	**1300**	3600	1295	1305	3600
I9	389	**439**	3600	406	**439**	3600	406	**439**	3600	406	**439**	3600
I10	**4631**	**4631**	205.37	**4631**	**4631**	307.67	**4631**	**4631**	697.61	**4631**	**4631**	401.07
I11	**3443**	**3443**	186.80	**3443**	**3443**	149.95	**3443**	**3443**	237.44	**3443**	**3443**	152.04
I12	**4040**	**4040**	468.77	**4040**	**4040**	**404.37**	**4040**	**4040**	895.11	**4040**	**4040**	664.70
I13	**1348**	—	3600	**1348**	—	3600	1329	—	3600	1130	—	3600
I14	**1278**	**1278**	1108.41	**1278**	**1278**	**924.98**	**1278**	**1278**	2268.40	**1278**	**1278**	2399.26
I15	**3816**	**4862**	3600	3815	5492	3600	3815	5769	3600	3815	5258	3600
I16	**3225**	**3225**	1398.90	**3225**	**3225**	1056.44	**3225**	**3225**	1024.45	**3225**	**3225**	**976.74**
I17	**5746**	**5746**	**1167.41**	**5746**	**5746**	1199.21	**5746**	**5746**	2904.20	**5746**	**5746**	2175.01
I18	4370	4675	3600	**4387**	4580	3600	4358	4863	3600	4365	4678	3600
I19	**3141**	119169	3600	3103	—	3600	2945	—	3600	2944	—	3600
I20	—	—	3600	—	—	3600	—	—	3600	—	—	3600
I21	—	—	3600	—	—	3600	—	—	3600	—	—	3600
I22	—	—	3600	—	—	3600	—	—	3600	—	—	3600
I23	—	—	3600	—	—	3600	—	—	3600	—	—	3600
I24	—	—	3600	—	—	3600	—	—	3600	—	—	3600

Table 9.7: This table shows the results produced using MIP formulation that formulates block series constraints as network flows [Sme18] (NetworkFlow) and the formulation from this work that uses a similar network flow encoding (NewNetworkFlow). Results formatted in boldface highlight the overall best lower bounds (LB), upper bounds (UB), and fastest proof times in seconds (Runtime) achieved using Gurobi (see also Tables 9.6 and 9.8). A — indicates that no solutions could be found within the time limit.

	NewAutomata			NewAutomata wd/val1		
Instance	LB	UB	Runtime	LB	UB	Runtime
I1	**607**	**607**	3.85	**607**	**607**	5.91
I2	**828**	**828**	19.66	**828**	**828**	38.98
I3	**1001**	**1001**	80.55	**1001**	**1001**	93.46
I4	**1716**	**1716**	71.73	**1716**	**1716**	310.80
I5	1040	**1143**	3600	783	1145	3600
I6	**1950**	**1950**	690.62	**1950**	**1950**	2690.25
I7	**1056**	**1056**	2428.77	1040	**1056**	3584.89
I8	1277	2208	3600	245	—	3600
I9	39	445	3600	39	539	3600
I10	1145	—	3600	50	—	3600
I11	—	—	3600	—	—	3600
I12	—	—	3600	—	—	3600
I13	—	—	3600	—	—	3600
I14	39	—	3600	20	—	3600
I15	—	—	3600	—	—	3600
I16	2950	3723	3600	2817	—	3600
I17	—	—	3600	—	—	3600
I18	1449	—	3600	927	—	3600
I19	—	—	—	—	—	3600
I20	—	—	3600	—	—	3600
I21	—	—	3600	—	—	3600
I22	—	—	3600	—	—	3600
I23	—	—	3600	—	—	3600
I24	—	—	3600	—	—	3600

Table 9.8: This table shows the results produced using the solver-independent formulation that uses automata global constraints (NewAutomata). Results formatted in boldface highlight the overall best lower bounds (LB), upper bounds (UB), and fastest proof times in seconds (Runtime) achieved using Gurobi (see also Tables 9.6 and 9.7). A — indicates that no solutions could be found within the time limit.

with the benchmark instances. Table 9.10 shows an overview of the optimal costs for the initial LP relaxation for the *Direct*, *NewDirect*, *NetworkFlow*, and *NewNetworkFlow* models for Instances 1–20 (The LP relaxation could not be solved for larger instances 21–24 within one hour on our benchmark machine).

The results show that for 19 out of 20 of the instances, the new working weekend formulation can provide a tighter lower bound to the optimal integer solution when compared to the existing working weekend formulation, which indicates the positive effect of the reformulation regarding this bound.

Comparison of CP and MIP Results

The summarized best results produced with CP and MIP solving technology within a time limit of one hour are presented in Table 9.11.

The results show that the best results could be reached with the MIP translations of the

	Existing MIP			Existing MIP+CB			New MIP			New MIP+CB		
Instance	LB	UB	Runtime	LB	UB	Runtime	LB	UB	Runtime	LB	UB	Runtime
I1	**607**	**607**	0.38	**607**	**607**	0.38	**607**	**607**	**0.345**	**607**	**607**	0.347
I2	**828**	**828**	2.59	**828**	**828**	2.28	**828**	**828**	**1.54**	**828**	**828**	1.88
I3	**1001**	**1001**	3.63	**1001**	**1001**	3.80	**1001**	**1001**	3.52	**1001**	**1001**	**2.61**
I4	**1716**	**1716**	35.72	**1716**	**1716**	40.77	**1716**	**1716**	34.96	**1716**	**1716**	**13.79**
I5	**1143**	**1143**	**18.95**	**1143**	**1143**	26.59	**1143**	**1143**	28.51	**1143**	**1143**	32.85
I6	**1950**	**1950**	20.72	**1950**	**1950**	22.49	**1950**	**1950**	**17.00**	**1950**	**1950**	28.56
I7	**1056**	**1056**	161.96	**1056**	**1056**	183.75	**1056**	**1056**	134.89	**1056**	**1056**	**127.30**
I8	**1297**	1301	3600	1296	**1300**	3600	**1297**	**1300**	3600	1295	1305	3600
I9	406	**439**	3600	**439**	**439**	**1165.44**	413	**439**	3600	406	**439**	3600
I10	**4631**	**4631**	**126.34**	**4631**	**4631**	133.38	**4631**	**4631**	307.67	**4631**	**4631**	401.07
I11	**3443**	**3443**	50.08	**3443**	**3443**	53.41	**3443**	**3443**	**40.35**	**3443**	**3443**	152.04
I12	**4040**	**4040**	468.77	**4040**	**4040**	895.11	**4040**	**4040**	**404.37**	**4040**	**4040**	664.70
I13	**1348**	3094	3600	1343	3308	3600	**1348**	**2957**	3600	1336	—	3600
I14	**1278**	**1278**	1108.41	**1278**	**1278**	2268.40	**1278**	**1278**	**924.98**	**1278**	**1278**	2399.26
I15	**3816**	**4862**	3600	3815	5769	3600	3815	5492	3600	3815	5258	3600
I16	**3225**	**3225**	1398.90	**3225**	**3225**	1024.45	**3225**	**3225**	1056.44	**3225**	**3225**	**976.74**
I17	**5746**	**5746**	**1167.41**	**5746**	**5746**	2904.20	**5746**	**5746**	1199.21	**5746**	**5746**	2175.01
I18	4370	**4560**	3600	4358	4661	3600	**4387**	4580	3600	4365	4678	3600
I19	**3141**	4017	3600	2945	5056	3600	3103	**3813**	3600	2944	4208	3600
I20	4748	206698	3600	4749	**202406**	3600	**4751**	209034	3600	4735	—	3600
I21	—	—	3600	—	—	3600	—	—	3600	—	—	3600
I22	—	—	3600	—	—	3600	—	—	3600	—	—	3600
I23	—	—	3600	—	—	3600	—	—	3600	—	—	3600
I24	—	—	3600	—	—	3600	—	—	3600	—	—	3600
# Best/Fastest	17	16	3	14	16	1	17	17	6	13	14	4

Table 9.9: Overview on the best results achieved by existing MIP formulations from the literature (Existing MIP) and the best results produced using the proposed formulations (New MIP) with and without custom branching strategies (CB). Results formatted in boldface highlight the overall best lower bounds (LB), upper bounds (UB), and fastest proof times in seconds (Runtime). The final row of the table counts the number of best lower bounds, upper bounds as well as the fastest optimality proofs. A — indicates that no solutions could be found within 1 hour.

Instance	Direct	NewDirect	NetworkFlow	NewNetworkFlow
I1	405.00	405.00	406.00	430.25
I2	715.00	716.00	718.67	725.50
I3	1000.00	1000.00	1000.00	1000.00
I4	1101.00	1202.50	1364.34	1488.15
I5	702.00	704.00	742.89	903.95
I6	1901.00	1904.00	1925.29	1938.08
I7	138.00	922.83	168.74	943.86
I8	1227.25	1228.42	1247.45	1259.70
I9	36.88	38.88	36.88	38.88
I10	4608.00	4614.50	4610.22	4622.23
I11	3412.50	3417.00	3413.50	3423.31
I12	2523.25	3622.00	2529.26	3630.48
I13	508.00	517.75	510.02	519.81
I14	1245.75	1254.50	1248.93	1260.46
I15	3737.63	3737.63	3751.50	3760.24
I16	3142.75	3150.83	3156.92	3172.56
I17	5689.50	5702.00	5699.00	5723.87
I18	4166.50	4173.75	4179.89	4211.06
I19	1968.38	2682.13	1981.66	2727.58
I20	728.88	4091.08	754.05	4127.87

Table 9.10: The optimal costs of the initial LP relaxation for instances 1–20 that have been determined using different integer programming formulations. Columns “Direct” and “NetworkFlow” show the relaxed costs of the direct model [CQ14] and the network flow model [Sme18] together with the original formulation of the working weekend constraint while columns “NewDirect” and “NewNetworkFlow” show the relaxed costs achieved using the new formulation for the working weekends constraint.

solver-independent model as well as existing MIP formulations for each of the instances.

Overall, the experimental results show that the proposed solver-independent models could be used to reach optimal solutions for several benchmark instances. While results produced with CP could not compete with state-of-the-art MIP models, experimental results indicate that Gurobi can improve the performance regarding computational speed and solution quality with the proposed high level models for several instances compared to the state of the art.

	CP	**MIP**	
Instance	UB	LB	UB
I1	**607**	607	**607**
I2	957	828	**828**
I3	1368	1001	**1001**
I4	2583	1716	**1716**
I5	3510	1143	**1143**
I6	6013	1950	**1950**
I7	5955	1056	**1056**
I8	11295	1297	**1300**
I9	7896	439	**439**
I10	16466	4631	**4631**
I11	21124	3443	**3443**
I12	30151	4040	**4040**
I13	—	1348	**2957**
I14	13336	1278	**1278**
I15	38054	3816	**4862**
I16	14469	3225	**3225**
I17	66066	5746	**5746**
I18	43340	4387	**4560**
I19	82128	3141	**3813**
I20	271968	4751	**202406**
I21	—	—	—
I22	—	—	—
I23	—	—	—
I24	—	—	—

Table 9.11: Column 2 shows the overall best upper bounds achieved using CP, whereas Columns 3–4 show the best lower and upper bounds achieved with the MIP formulations. Results in boldface mark the best found solutions per instance. A — indicates that no solutions could be found within the time limit.

CHAPTER 10

Conclusion

In this thesis we introduced two novel NP-hard real-life production scheduling problems arising from the automotive supply industry and teeth manufacturing. We performed an in-depth analysis of the problems and provided a formal specification as well as a collection of benchmark instances that include many real-life scheduling scenarios. Furthermore, we proposed several exact and heuristic solution methods that could successfully provide high-quality solutions even for the largest instances of the investigated realistic scheduling scenarios.

To efficiently approach the paint shop scheduling problem which appears in paint shops of the automotive supply industry with exact techniques, we investigated two alternative constraint modeling approaches together with various programmed search strategies. An extensive empirical evaluation showed that state-of-the art MIP and CP solving technology could provide optimal results for several benchmark instances using our models. However, solutions to the large instances that represent real-life scheduling scenarios could not be acquired by exact approaches within the space and runtime limits of our experimental environment.

Therefore, we further proposed an innovative metaheuristic approach based on simulated annealing that was able to achieve feasible solutions for all benchmark instances. Additionally, we investigated and solved an important sub-problem that aims to minimize the required color changes in the paint shop's production sequence. The exact and heuristic solution methods we proposed for this problem, could further be utilized within a novel large neighborhood search approach to the paint shop scheduling problem. Experimental results showed that this approach, which is able to hybridize the previously introduced exact and heuristic techniques, could produce the overall best results for the large majority of the benchmark instances.

Through an investigation of different constraint modeling techniques for the paint shop scheduling problem, we additionally discovered an important novel string edit distance

global constraint that is able to efficiently express string distance constraints that appear in NP-hard constraint optimization problems such as the paint shop scheduling problem. We proposed an innovative efficient constraint propagator for this constraint in addition to constraint decomposition modeling techniques that are utilized in the constraint model of the paint shop scheduling problem. Furthermore, we provided an algorithm that can provide minimal explanations for the propagator, thereby making it applicable for use together with state-of-the-art lazy clause generation solvers. A series of experiments clearly showed the effectiveness of the proposed constraint and propagation techniques as it could improve state-of-the-art results on benchmark instances for paint shop scheduling and the median string problem.

To approach the artificial teeth scheduling problem, which is the second real-life problem we introduced in this thesis, we proposed a CP model that can be utilized together with state-of-the-art MIP and CP solvers as an exact solution method. Experiments showed that this approach is able to produce optimal results for several instances and can further provide lower bounds for all benchmark instances. However, similar as with the paint shop scheduling problem the modeling approach could not provide solutions for large practical instances on our benchmark machine. Therefore, we further proposed a simulated annealing metaheuristic together with novel neighborhood operators that could successfully provide solutions for all benchmark instances. Additionally, we investigated the effectiveness of state-of-the-art hyper-heuristic strategies on the artificial teeth scheduling problem. We proposed several low-level heuristics that utilize local-search, mutation, and crossover operators within the well known selection-perturbation based hyper-heuristic framework HyFlex, which allowed us to evaluate our heuristics using several state-of-the-art problem independent heuristic strategies. Experimental results showed that the proposed low-level heuristics could be successfully utilized to improve the quality of solutions for many of the realistic benchmark instances when compared with the simulated annealing approach.

Finally, we further investigated solver-independent modeling techniques for workforce scheduling problems which in practice often arise together with production scheduling tasks in many industrial application domains. We performed an extensive experimental study using various search strategies on a set of benchmark instances from the literature and studied different CP and MIP encodings of the solver-independent model. The empirical results show that our model could produce competitive results when compared to the state of the art. Furthermore, the proposed MIP encoding could improve results regarding the optimality proof time for several instances.

Overall, we can conclude that exact solution methods based on constraint modeling and local search based metaheuristics can efficiently complement each other when approaching novel production scheduling problems like paint shop scheduling or artificial teeth scheduling. On the one hand, the high-level constraint models make it simple to utilize state-of-the-art solving technology from different areas such as CP and MIP to provide bounds and optimality results for small- to medium-sized instances, which can be used to evaluate the performance of heuristics techniques. On the other hand, local search

based metaheuristics and hybrid methods can efficiently obtain solutions even for real-life instances for which exact techniques cannot deliver results within reasonable space and runtime limitations, as it was the case for both investigated problems.

10.1 Future Work

Our experimental results showed that all of the proposed methods could produce high-quality solutions for real-life scheduling scenarios as they appear in modern day factories. For future work it could be interesting to additionally introduce random instance generators for paint shop scheduling and the artificial teeth scheduling problem to study the efficiency of the introduced approaches also on diverse artificially created instance sets. Such an extended instance set could then form the basis for an investigation of advanced algorithm selection methods that study the strengths and weaknesses of the different algorithms depending on various instance features.

Regarding the string edit distance constraint we have shown that our algorithms can efficiently propagate upper bounds on the minimum edit distance between given variable arrays. As a next step, it could be interesting to investigate if also the exact distance can be propagated efficiently. Furthermore, another possible topic of investigation could be backwards propagation that propagates restrictions to the domains of variables that represent the strings from a given bound on the edit distance.

Our evaluation of the proposed low-level heuristics together with hyper-heuristic approaches on the artificial teeth scheduling problem provided promising results on the benchmark instances. In future work it could be interesting to further extend the set of low-level heuristics for this problem with operators that implement a destroy-and-repair scheme. This could for example be achieved using the proposed constraint model within the framework of large-neighborhood search.

Another interesting topic could be the investigation of further advanced exact methods that utilize problem decomposition techniques such as column generation. As the artificial teeth scheduling problem for example considers an exponentially large number of mould- and color patterns for the creation of jobs, it could be interesting to investigate an approach that generates variables that select such patterns by using column generation.

Bibliography

[AÖP13] Shahriar Asta, Ender Özcan, and Andrew J. Parkes. Batched Mode Hyper-heuristics. In Giuseppe Nicosia and Panos Pardalos, editors, *Learning and Intelligent Optimization*, Lecture Notes in Computer Science, pages 404–409, Berlin, Heidelberg, 2013. Springer.

[BC14] Edmund K. Burke and Timothy Curtois. New approaches to nurse rostering benchmark instances. *European Journal of Operational Research*, 237(1):71–81, 2014.

[BGH+13] Edmund K Burke, Michel Gendreau, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and Rong Qu. Hyper-heuristics: a survey of the state of the art. *Journal of the Operational Research Society*, 64(12):1695–1724, December 2013.

[BGP03] Stéphane Bourdais, Philippe Galinier, and Gilles Pesant. hibiscus: A Constraint Programming Application to Staff Scheduling in Health Care. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming – CP 2003*, Lecture Notes in Computer Science, pages 153–167, Berlin, Heidelberg, 2003. Springer.

[BHvMW21] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2021.

[BSTW16] Gleb Belov, Peter J. Stuckey, Guido Tack, and Mark G. Wallace. Improved linearization of constraint programming models. In Michel Rueher, editor, *Principles and Practice of Constraint Programming – CP 2016*, pages 49–65, Cham, 2016. Springer International Publishing.

[CDGD06] Raffaele Cipriano, Luca Di Gaspero, and Agostino Dovier. Hybrid Approaches for Rostering: A Case Study in the Integration of Constraint Programming and Local Search. In Francisco Almeida, María J. Blesa Aguilera, Christian Blum, José Marcos Moreno Vega, Melquíades Pérez Pérez, Andrea Roli, and Michael Sampels, editors, *Hybrid Metaheuristics*, Lecture Notes in Computer Science, pages 110–123, Berlin, Heidelberg, 2006. Springer.

[CGR07] Marie-Claude Côté, Bernard Gendron, and Louis-Martin Rousseau. Modeling the regular constraint with integer programming. In Pascal Van Hentenryck and Laurence Wolsey, editors, *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming – CPAIOR 2007*, pages 29–43. Springer Berlin Heidelberg, 2007.

[Chu11] Geoffrey Chu. *Improving Combinatorial Optimization*. PhD thesis, Department of Computing and Information Systems, University of Melbourne, 2011.

[Chu20] Chung-yao Chuang. *Combining Multiple Heuristics: Studies on Neighborhood-base Heuristics and Sampling-based Heuristics*. thesis, Carnegie Mellon University, May 2020.

[CL96] Hoong Chuin Lau. On the complexity of manpower shift scheduling. *Computers & Operations Research*, 23(1):93–102, 1996.

[Cor19] IBM Corporation. *IBM ILOG CPLEX 12.10 User's Manual*. 2019.

[CQ14] Tim Curtois and Rong Qu. Computational results on new staff scheduling benchmark instances. Technical report, ASAP Research Group, School of Computer Science, University of Nottingham, NG8 1BB, Nottingham, UK, October 2014.

[CXIC12] C. Y. Chan, Fan Xue, W. H. Ip, and C. F. Cheung. A Hyper-Heuristic Inspired by Pearl Hunting. In Youssef Hamadi and Marc Schoenauer, editors, *Learning and Intelligent Optimization*, Lecture Notes in Computer Science, pages 349–353, Berlin, Heidelberg, 2012. Springer.

[DFM+95] S. J. Darmoni, A. Fajner, N. Mahé, A. Leforestier, M. Vondracek, O. Stelian, and M. Baldenweck. HOROPLAN: computer-assisted nurse scheduling using constraint-based programming. *Journal of the Society for Health Systems*, 5(1):41–54, January 1995.

[DK01] Andreas Drexl and Alf Kimms. Sequencing JIT Mixed-Model Assembly Lines Under Station-Load and Part-Usage Constraints. *Management Science*, 47(3):480–491, 2001.

[DKM06] Andreas Drexl, Alf Kimms, and Lars Matthießen. Algorithms for the car sequencing and the level scheduling problem. *J. Sched.*, 9(2):153–176, 2006.

[DKÖB20] John H. Drake, Ahmed Kheiri, Ender Özcan, and Edmund K. Burke. Recent advances in selection hyper-heuristics. *European Journal of Operational Research*, 285(2):405–428, September 2020.

[DMS+21] Emir Demirović, Nysret Musliu, Andreas Schutt, Peter J. Stuckey, and Felix Winter. Solver-independent models for employee scheduling. *(Under Submission)*, 2021.

[DMW19] Emir Demirovic, Nysret Musliu, and Felix Winter. Modeling and solving staff scheduling with partial weighted maxsat. *Annals OR*, 275(1):79–99, 2019.

[DSVH88] Mehmet Dincbas, Helmut Simonis, and Pascal Van Hentenryck. Solving the Car-Sequencing Problem in Constraint Logic Programming. In *Proceedings of the 8th European Conference on Artificial Intelligence*, ECAI'88, pages 290–295, USA, 1988. Pitman Publishing, Inc.

[dWBH20] Mathijs de Weerdt, Robert Baart, and Lei He. Single-machine scheduling with release times, deadlines, setup times, and rejection. *European Journal of Operational Research*, October 2020.

[EHO04] Th. Epping, W. Hochstättler, and P. Oertel. Complexity results on a paint shop problem. *Discrete Applied Mathematics*, 136(2):217–226, February 2004.

[GACD13] Akshay Gupte, Shabbir Ahmed, Myun Seok Cheon, and Santanu Dey. Solving Mixed Integer Bilinear Problems Using MILP Formulations. *SIAM Journal on Optimization*, 23(2):721–744, January 2013.

[Gec19] Gecode Team. *Gecode: Generic Constraint Development Environment*. 2019.

[GGP06] Caroline Gagné, Marc Gravel, and Wilson L. Price. Solving real car sequencing problems with ant colony optimization. *European Journal of Operational Research*, 174(3):1427–1448, November 2006.

[GJ79] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[GJS76] Michael R Garey, David S Johnson, and Ravi Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research*, 1(2):117–129, 1976.

[GKK+21] Tobias Geibinger, Lucas Kletzander, Matthias Krainz, Florian Mischek, Nysret Musliu, and Felix Winter. Physician Scheduling During a Pandemic. In Peter J. Stuckey, editor, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, Lecture Notes in Computer Science, pages 456–465. Springer International Publishing, 2021.

[GO20] LLC Gurobi Optimization. *Gurobi Optimizer Reference Manual*. 2020.

[GP19] Michel Gendreau and Jean-Yves Potvin, editors. *Handbook of Metaheuristics*. International Series in Operations Research & Management Science. Springer International Publishing, 3 edition, 2019.

[HCF12] Ping-Che Hsiao, Tsung-Che Chiang, and Li-Chen Fu. A VNS-based hyper-heuristic with adaptive computational budget of local search. In *2012 IEEE Congress on Evolutionary Computation*, pages 1–8, June 2012.

[HK16] Morihiro Hayashida and Hitoshi Koyano. Finding median and center strings for a probability distribution on a set of strings under levenshtein distance based on integer linear programming. In *BIOSTEC (Selected Papers)*, volume 690 of *Communications in Computer and Information Science*, pages 108–121. Springer, 2016.

[HKS14] Öncü Hazır and Safia Kedad-Sidhoum. Batch sizing and just-in-time scheduling with common due date. *Annals of Operations Research*, 213(1):187–202, February 2014.

[JABC03] Xiaoyi Jiang, Karin Abegglen, Horst Bunke, and János Csirik. Dynamic computation of generalised median strings. *Pattern Anal. Appl.*, 6(3):185–193, 2003.

[JLN+09] Michael Jünger, Thomas M Liebling, Denis Naddef, George L Nemhauser, William R Pulleyblank, Gerhard Reinelt, Giovanni Rinaldi, and Laurence A Wolsey. *50 years of Integer Programming 1958-2008: From the early years to the state-of-the-art*. Springer Science & Business Media, 2009.

[Kar72] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972.

[KGV83] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[Kis04] Tamás Kis. On the complexity of the car sequencing problem. *Operations Research Letters*, 32(4):331–335, July 2004.

[KLSD18] Minh Thanh Khong, Christophe Lecoutre, Pierre Schaus, and Yves Deville. Soft-regular with a prefix-size violation measure. In *CPAIOR*, volume 10848 of *Lecture Notes in Computer Science*, pages 333–343. Springer, 2018.

[KM20] Lucas Kletzander and Nysret Musliu. Solving the general employee scheduling problem. *Computers & Operations Research*, 113:104794, 2020.

[Koh85] Teuvo Kohonen. Median strings. *Pattern Recognition Letters*, 3(5):309–313, 1985.

[Kru83] Joseph B. Kruskal. An overview of sequence comparison: Time warps, string edits, and macromolecules. *SIAM Review*, 25(2):201–237, 1983.

[LEF+17] Marius Lindauer, Katharina Eggensperger, Matthias Feurer, Stefan Falkner, André Biedenkapp, and Frank Hutter. *SMAC v3: Algorithm Configuration in Python*. GitHub, 2017.

[LG07] Philippe Laborie and Daniel Godard. Self-Adapting Large Neighborhood Search: Application to Single-Mode Scheduling Problems. 2007.

[LM12] Andreas Lehrbaum and Nysret Musliu. A New Hyperheuristic Algorithm for Cross-Domain Search Problems. In Youssef Hamadi and Marc Schoenauer, editors, *Learning and Intelligent Optimization*, Lecture Notes in Computer Science, pages 437–442, Berlin, Heidelberg, 2012. Springer.

[MaH01] Harald Meyer auf'm Hofe. Solving Rostering Tasks as Constraint Optimization. In Edmund Burke and Wilhelm Erben, editors, *Practice and Theory of Automated Timetabling III*, Lecture Notes in Computer Science, pages 191–212, Berlin, Heidelberg, 2001. Springer.

[MBL09] Jean-Philippe Métivier, Patrice Boizumault, and Samir Loudni. Solving Nurse Rostering Problems Using Soft Global Constraints. In Ian P. Gent, editor, *Principles and Practice of Constraint Programming - CP 2009*, Lecture Notes in Computer Science, pages 73–87, Berlin, Heidelberg, 2009. Springer.

[MCB19] Ignacio Moya, Manuel Chica, and Joaquín Bautista. Constructive metaheuristics for solving the Car Sequencing Problem under uncertain partial demand. *Comput. Ind. Eng.*, 137, 2019.

[Mei11] David Meignan. An evolutionary programming hyper-heuristic with co-evolution for chesc11. In *The 53rd Annual Conference of the UK Operational Research Society (OR53)*, volume 3, 2011.

[MM21] Florian Mischek and Nysret Musliu. A collection of hyper-heuristics for the hyflex framework. Technical report, TU Wien, CD-TR, 2021/2, 2021.

[MN12] Frédéric Meunier and Bertrand Neveu. Computing solutions of the paintshop–necklace problem. *Computers & Operations Research*, 39(11):2666–2678, November 2012.

[MSS18] Nysret Musliu, Andreas Schutt, and Peter J. Stuckey. Solver independent rotating workforce scheduling. In *CPAIOR*, volume 10848 of *Lecture Notes in Computer Science*, pages 429–445. Springer, 2018.

[MVDCVB12] Mustafa Mısır, Katja Verbeeck, Patrick De Causmaecker, and Greet Vanden Berghe. An Intelligent Hyper-Heuristic Framework for CHeSC

2011. In Youssef Hamadi and Marc Schoenauer, editors, *Learning and Intelligent Optimization*, Lecture Notes in Computer Science, pages 461–466, Berlin, Heidelberg, 2012. Springer.

[MW17] Nysret Musliu and Felix Winter. A hybrid approach for the sudoku problem: Using constraint programming in iterated local search. *IEEE Intelligent Systems*, 32(2):52–62, 2017.

[Nav01] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[NR03] François Nicolas and Eric Rivals. Complexities of the centre and median string problems. In *CPM*, volume 2676 of *Lecture Notes in Computer Science*, pages 315–327. Springer, 2003.

[NSB+07] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard CP modelling language. In *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, pages 529–543, 2007.

[NSD+19] Shengsheng Niu, Shiji Song, Jian-Ya Ding, Yuli Zhang, and Raymond Chiong. Distributionally robust single machine scheduling with the total tardiness criterion. *Computers & Operations Research*, 101:13–28, January 2019.

[OHC+12] Gabriela Ochoa, Matthew Hyde, Tim Curtois, Jose A. Vazquez-Rodriguez, James Walker, Michel Gendreau, Graham Kendall, Barry McCollum, Andrew J. Parkes, Sanja Petrovic, and Edmund K. Burke. HyFlex: A Benchmark Framework for Cross-Domain Heuristic Search. In Jin-Kao Hao and Martin Middendorf, editors, *Evolutionary Computation in Combinatorial Optimization*, Lecture Notes in Computer Science, pages 136–147, Berlin, Heidelberg, 2012. Springer.

[OO08] Cristian Olivares-Rodríguez and José Oncina. A stochastic approach to median string computation. In *SSPR/SPR*, volume 5342 of *Lecture Notes in Computer Science*, pages 431–440. Springer, 2008.

[OSC09] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.

[PB17] Nelishia Pillay and Derrick Beckedahl. EvoHyp - a Java toolkit for evolutionary algorithm hyper-heuristics. In *2017 IEEE Congress on Evolutionary Computation (CEC)*, pages 2706–2713, June 2017.

[Pes04] Gilles Pesant. A Regular Language Membership Constraint for Finite Sequences of Variables. In *CP*, volume 3258 of *Lecture Notes in Computer Science*, pages 482–495. Springer, 2004.

[PG02] Markus Puchta and Jens Gottlieb. Solving Car Sequencing Problems by Local Optimization. In *EvoWorkshops*, volume 2279 of *Lecture Notes in Computer Science*, pages 132–142. Springer, 2002.

[PK00] Chris N. Potts and Mikhail Y. Kovalyov. Scheduling with batching: A review. *European Journal of Operational Research*, 120(2):228–249, January 2000.

[PKW86] Bruce D. Parrello, Waldo C. Kabat, and L. Wos. Job-shop scheduling using automated reasoning: A case study of the car-sequencing problem. *Journal of Automated Reasoning*, 2(1):1–42, March 1986.

[PQ18] Nelishia Pillay and Rong Qu. *Hyper-Heuristics: Theory and Applications*. Natural Computing Series. Springer International Publishing, 2018.

[PR08] Matthias Prandtstetter and Günther R. Raidl. An integer linear programming approach and a hybrid variable neighborhood search for the car sequencing problem. *European Journal of Operational Research*, 191(3):1004–1022, December 2008.

[PTM20] Sergey Polyakovskiy, Dhananjay Thiruvady, and Rym M'Hallah. Just-in-time batch scheduling subject to batch size. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, GECCO '20, pages 228–235, New York, NY, USA, June 2020. Association for Computing Machinery.

[RAL17] Erfan Rahimian, Kerem Akartunali, and John Levine. A hybrid integer programming and variable neighbourhood search algorithm to solve nurse rostering problems. *European Journal of Operational Research*, 258(2):411–423, 2017.

[RvBW06] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006.

[SBF10] Peter J. Stuckey, Ralph Becket, and Julien Fischer. Philosophy of the MiniZinc challenge. *Constraints*, 15(3):307–316, July 2010.

[SCBM13] Ricardo Soto, Broderick Crawford, Rodrigo Bertrand, and Eric Monfroy. Modeling NRPs with Soft and Reified Constraints. *AASRI Procedia*, 4:202–205, January 2013.

[SCNA08] Christine Solnon, Van Dat Cung, Alain Nguyen, and Christian Artigues. The car sequencing problem: Overview of state-of-the-art methods and industrial case-study of the ROADEF'2005 challenge problem. *European Journal of Operational Research*, 191(3):912–927, December 2008.

[SFSZ15] Hui Sun, Shujin Fan, Xianle Shao, and Jiangong Zhou. A colour-batching problem using selectivity banks in automobile paint shops. *International Journal of Production Research*, 53(4):1124–1142, February 2015.

[SGV04] S. Spieckermann, K. Gutenschwager, and S. Voß. A sequential ordering problem in automotive paint shops. *International Journal of Production Research*, 42(9):1865–1878, May 2004.

[SH17] Hui Sun and Jianming Han. A study on implementing color-batching with selectivity banks in automotive paint shops. *Journal of Manufacturing Systems*, 44:42–52, July 2017.

[SKCU77] Y. Sugimori, K. Kusunoki, F. Cho, and S. Uchikawa. Toyota production system and kanban system materialization of just-in-time and respect-for-human system. *International Journal of Production Research*, 15(6):553–564, 1977.

[Sme18] Pieter Smet. Constraint reformulation for nurse rostering problems. In *Proceedings of the 12th international conference on the practice and theory of automated timetabling*, pages 69–80. PATAT, 2018.

[SYK+20] Ivan Kristianto Singgih, Onyu Yu, Byung-In Kim, Jeongin Koo, and Seungdoe Lee. Production scheduling problem in a factory of automobile component primer painting. *Journal of Intelligent Manufacturing*, January 2020.

[TB20] Tanya Y. Tang and J. Christopher Beck. CP and Hybrid Models for Two-Stage Batching and Scheduling. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, Lecture Notes in Computer Science, pages 431–446, 2020.

[TS18] Charles Thomas and Pierre Schaus. Revisiting the Self-adaptive Large Neighborhood Search. In Willem-Jan van Hoeve, editor, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, Lecture Notes in Computer Science, pages 557–566, Cham, 2018. Springer International Publishing.

[Ukk85] Esko Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64(1-3):100–118, 1985.

[vHK06] Willem-Jan van Hoeve and Irit Katriel. Global constraints. In *Foundations of Artificial Intelligence*, volume 2, pages 169–208. Elsevier, 2006.

[VHM10] Pascal Van Hentenryck and Michela Milano. *Hybrid optimization: the ten years of CPAIOR*, volume 45. Springer Science & Business Media, 2010.

[VMW21] Johannes Vass, Nysret Musliu, and Felix Winter. Solving the Production Leveling Problem with Order-Splitting and Resource Constraints. *Proceedings of the 13th International Conference on the Practice and Theory of Automated Timetabling*, I:261–284, 2021.

[WF74] Robert A. Wagner and Michael J. Fischer. The String-to-String Correction Problem. *Journal of the ACM (JACM)*, January 1974.

[WHFP95] G. Weil, K. Heus, P. Francois, and M. Poujade. Constraint programming for nurse scheduling. *IEEE Engineering in Medicine and Biology Magazine*, 14(4):417–422, July 1995.

[WM21a] Felix Winter and Nysret Musliu. Constraint-based Scheduling for Paint Shops in the Automotive Supply Industry. *ACM Transactions on Intelligent Systems and Technology*, 12(2):17:1–17:25, January 2021.

[WM21b] Felix Winter and Nysret Musliu. A large neighborhood search approach for the paint shop scheduling problem. *Journal of Scheduling*, 2021.

[WMDM19] Felix Winter, Nysret Musliu, Emir Demirović, and Christoph Mrkvicka. Solution Approaches for an Automotive Paint Shop Scheduling Problem. *Proceedings of the International Conference on Automated Planning and Scheduling*, 29:573–581, July 2019.

[WMMP21] Felix Winter, Christoph Mrkvicka, Nysret Musliu, and Jakob Preininger. Automated Production Scheduling for Artificial Teeth Manufacturing. *Proceedings of the International Conference on Automated Planning and Scheduling*, 31:500–508, May 2021.

[WMS20] Felix Winter, Nysret Musliu, and Peter Stuckey. Explaining Propagators for String Edit Distance Constraints. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(02):1676–1683, April 2020.

[WMW21] Wolfgang Weintritt, Nysret Musliu, and Felix Winter. Solving the paintshop scheduling problem with memetic algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '21, pages 1070–1078. Association for Computing Machinery, June 2021.

[ZLZ+20] Z. Zhao, S. Liu, M. Zhou, X. Guo, and L. Qi. Decomposition Method for New Single-Machine Scheduling Problems From Steel Production Systems. *IEEE Transactions on Automation Science and Engineering*, 17(3):1376–1387, July 2020.

Felix Winter

Curriculum Vitae

Personal Data

Place and date of birth:	Vienna, 29 July 1990
email:	winter@dbai.tuwien.ac.at

Education

since December 2017	PhD student at Vienna University of Technology, Austria Thesis Title: *Automated Scheduling for Automotive Supplier Paint Shops and Teeth Manufacturing*
March 2013 - October 2016	Master of Science in Software Engineering & Internet Computing, Technical University Vienna Thesis: *MaxSAT Modeling and Heuristic Solution Methods for the Employee Scheduling Problem* pass with distinction
September 2012 - July 2013	Preliminary studies for Popular Music and Jazz Guitar, Franz Schubert Konservatorium Vienna
September 2008 - March 2013	Bachelor of Science in Software & Information Engineering, Technical University Vienna pass with distinction

Work experience

since December 2017	Project assistant in the *Christian Doppler Laboratory for Artificial Intelligence and Optimization for Planning and Scheduling*, Technical University Vienna
October 2016 - November 2017	Consultant and Software Developer for *Automated Planning and Scheduling Software*, MCP GmbH, Vienna
August 2015 - October 2016	Project assistant in the research project *Artificial Intelligence in Employee Scheduling*, Technical University Vienna
March 2014 - January 2016	Tutor for *Distributed System Technologies* and *Advanced Internet Security*, Technical University Vienna
October 2011 - February 2012	Junior Data Warehouse Consultant, pmOne GmbH, Vienna
Juli-September 2009	Software developer, xS+S,*x Software und Systeme, Vienna

Languages

English:	Fluent
German:	Mother Language

Publications

Felix Winter*, Nysret Musliu.
A Large Neighborhood Search Approach for the Paint Shop Scheduling Problem.
Journal of Scheduling, 2021

Felix Winter*, Christoph Mrkvicka, Nysret Musliu and Jakob Preininger.
Automated Production Scheduling for Artificial Teeth Manufacturing
31st International Conference on Automated Planning and Scheduling (ICAPS 2021)

Wolfgang Weintritt, Nysret Musliu and Felix Winter.
Solving the Paintshop Scheduling Problem with Memetic Algorithms
Genetic and Evolutionary Computation Conference (GECCO 2021)

Tobias Geibinger, Lucas Kletzander, Matthias Krainz, Florian Mischek, Nysret Musliu and Felix Winter.
Physician Scheduling During a Pandemic.
18th International Conference on the Integration of CP, AI, and OR (CPAIOR 2021)

Felix Winter*, Nysret Musliu.
Constraint-based Modeling for Scheduling Paint Shops in the Automotive Supply Industry.
ACM Transactions on Intelligent Systems and Technology, 2021

Johannes Vass, Nysret Musliu, Felix Winter.
Solving the Production Leveling Problem with Order-Splitting and Resource Constraints.
13th International Conference on the Practice and Theory of Automated Timetabling (PATAT 2021)

Felix Winter*, Nysret Musliu, Peter J. Stuckey.
Explaining Propagators for String Edit Distance Constraints.
34th AAAI Conference on Artificial Intelligence (AAAI 2020)

Felix Winter*, Emir Demirovic, Nysret Musliu and Christoph Mrkvicka.
Solution Approaches for an Automotive Paint Shop Scheduling Problem.
29th International Conference on Automated Planning and Scheduling (ICAPS 2019)

Emir Demirovic, Nysret Musliu, Felix Winter*.
Modeling and Solving Staff Scheduling with Partial Weighted maxSAT.
Annals of Operations Research, 2017

Nysret Musliu, Felix Winter*.
A Hybrid Approach for the Sudoku problem: Using Constraint Programming in Iterated Local Search.
IEEE Intelligent Systems, 2017

A * indicates that I am the corresponding author of the publication.

Scholarships, Awards and additional info

2009-2011	Performance scholarship from TU Vienna
2014-2015	Participation at TUtheTOP, the High Potential program from TU Vienna. TUtheTOP participants are selected among the best 20% of the students through a multi-stage selection process.
2016	Won the *Distinguished Young Alumnus Award* of the faculty of informatics for the best diploma thesis.
August 2018	Organizing committee member of the *12th International Conference on the Practice and Theory of Automated Timetabling (PATAT 2018)*.
July 2019	Participated in the *ACP Summer School on Constraint Programming* and won the hackathon in the medium sized instance category as a member of the winning team.
	Won the *Best Doctoral Consortium Poster Award* at the *29th International Conference on Automated Planning and Scheduling (ICAPS 2019)*.
September 2020/July 2021	Organizing committee member of the *17th and 18th International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR 2020/2021)*.

Interests and activities

Research interests: Metaheuristic algorithms, Constraint Programming, SAT & maxSAT solving, Hybrid approaches for optimization problems, Constraint satisfaction problems, Automated Algorithm Selection and Configuration

Activities: Playing music with my band, Swimming, Hiking