FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Large Neighborhood Search for Break Scheduling

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Information & Knowledge Management

eingereicht von

Maria-Elisabeth Züger, B.A.
Matrikelnummer 0926766

an der Fakultät für Informatik
der Technischen Universität Wien
Betreuung: Privatdoz. Dipl.-Ing. Dr.techn. Nysret Musliu

Wien, 1. August 2016

Maria-Elisabeth Züger	Nysret Musliu

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Large Neighborhood Search for Break Scheduling

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Information & Knowledge Management

by

Maria-Elisabeth Züger, B.A.
Registration Number 0926766

to the Faculty of Informatics

at the Vienna University of Technology

Advisor: Privatdoz. Dipl.-Ing. Dr.techn. Nysret Musliu

Vienna, 1st August, 2016

Maria-Elisabeth Züger	Nysret Musliu

Erklärung zur Verfassung der Arbeit

Maria-Elisabeth Züger, B.A.
Rütistraße 38
CH-8134 Adliswil

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, 1. August 2016

Maria-Elisabeth Züger

Acknowledgements

I am deeply grateful to my supervisor Nysret Musliu for guiding my master thesis. If not for him and his captivating lecture I might have never walked down this path. He supported me with useful remarks and discussions that always proofed to be fruitful. Further, I am grateful to Toni Pisjak, the system administrator of the DBAI group, for his technical support and prompt responses.

Finally, I would like to thank my whole family, who always provided me with everything I needed. A special thanks goes to my soon-to-be husband, who stood by me all the time and always had the right words to keep me motivated.

Kurzfassung

In gewissen Arbeitsbereichen, wie etwa in der Flugverkehrskontrolle oder bei Fließbandarbeit, ist ein hohes Maß an Konzentration erforderlich. In solchen Bereichen sind regelmäßige Pausen verpflichtend um fatale Fehler zu vermeiden. Pausen sind streng geregelt durch etwa Sicherheitsregeln oder rechtlichen Forderungen. Das Break Scheduling Problem (Bsp) befasst sich mit diesen Regelungen. Das Bsp hat zum Ziel Pausen in einem gegebenen Schichtplan einzuplanen, sodass alle Pausenregeln erfüllt sind während Abweichungen vom Personalbedarf minimiert werden.

In dieser Arbeit stellen wir eine Mixed Integer Programming Formulierung für das generelle Bsp vor. Um das Bsp zu lösen schlagen wir einen Large Neighborhood Suchalgorithmus (LNS) vor. Er besteht aus einer Initialisierungsphase und zwei Unteralgorithmen: ein Local Search Algorithmus und ein Mixed Integer Programming (MIP) Algorithmus. Um die MIP-Formulierung zu lösen kommt der Constraintlöser CPLEX zum Einsatz. Der Local Search Algorithmus verwendet zwei Nachbarschaftsoperationen: Swap und Assignment. Zusätzlich wird eine Random-Walk Methode genutzt um lokalen Optima zu entkommen. Local Search fokussiert sich auf einzelne Pausen. Der MIP Algorithmus entfernt alle Pausen einer gesamten Schicht um sie dann optimal wieder einzuplanen.

Die Unteralgorithmen werden abwechselnd eingesetzt und bei jeder Iteration durch einen Selector gewählt. Wir testeten drei Auswahlverfahren: Random-Selector, Timebound-Selector und Probability-Selector. Der Probability-Selector, welcher die Wahrscheinlichkeit mittels einer Funktion regelt, erwies sich als überlegen. Zudem wurden Parameter, welche einen Einfluss auf die Leistung des Algorithmus haben, evaluiert. Insgesamt wurden 56 Experimente durchgeführt, welche in Summe eine Laufzeit von 2.736 Stunden hatten. Wir berechneten Endergebnisse zu 30 Fallbeispielen, 20 von einem realen Szenario und 10 zufällig generierte. Der LNS Algorithmus übertrifft in den meisten Fällen unsere Implementierung von Local Search. Jedoch konnte er noch nicht an die besten bislang bekannten Resultate herankommen.

Abstract

A high a level of concentration is essential in certain working areas such as in air traffic control, assembly line works or supervision. In such areas breaks are mandatory to avoid fatal errors. Breaks are regulated due to safety rules or legal demands. The break scheduling problem (Bsp) deals with these kind of regulations. The aim of the Bsp is to assign breaks to a given shift plan so that all regulations regarding breaks are fulfilled while violations of staff requirements are minimized.

We give a mixed integer programming formulation for the general Bsp. To solve the Bsp we propose a large neighborhood search (LNS) algorithm. It is made up of an initialisation phase and two sub-algorithms: a local search algorithm and a mixed integer programming (MIP) algorithm. To solve the MIP formulation the solver CPLEX is used. The local search sub-algorithm uses two moves to optimize the solution: swap and assignment. In addition, a random-walk procedure is used to escape local optima. Local search focuses only on single breaks. The MIP sub-algorithm removes the break assignment of an entire shift and reassigns the breaks optimally.

Sub-algorithms are applied alternately and are chosen by a selector at each iteration. We tested three different selection procedures: random-selector, timebound-selector and probability-selector. The probability-selector, using a function to regulate the probability, has shown to be superior. Further, different parameter settings, which influence the performance of the algorithm, are evaluated. In total 56 experiments were performed taking a total runtime of 2,736 hours. We computed final results for 30 instances, 20 obtained from a real-life scenario and 10 randomly generated. The LNS algorithm outperforms our local search implementation in most of the cases. However, it did not yet reach the upper bounds of the best known results so far.

Contents

CHAPTER 1

Introduction

There are many working areas where relief and rest breaks are mandatory to maintain a high level of concentration. In security areas such as air traffic control or security checking, a loss in concentration can lead to dangerous situations. Assembly line workers performing monotonous, yet demanding work need to have breaks regularly in order to avoid fatal errors. Other factors that are relevant are health and safety regulations or legal obligations. Usually each area has their own rules on how to schedule breaks. The break scheduling problem (Bsp) deals with these kind of regulations when scheduling breaks into shift plans. Besides the strict rules and regulations on when, how many and how long breaks need to be, varying staff requirements need to be considered. Call centers for example face different customer demands during the day. The challenge of break scheduling is to satisfy break regulations while the optimum number of staff is present.

Compared to shift scheduling, break scheduling is a rather young topic. It was formaly first introduced by Beer et al. [BGM+10] in 2010. Designing shift plans is a well established task and managers recognize the benefits of automated scheduling. Break scheduling on the other hand was not addressed as a separate problem until recently. According to Thompson and Pullman [TP07] managers tend to schedule breaks roughly in advance and adjust them during the day in accordance with changing demands. However, this can lead to non-optimal or even illegal shift plans according to staff requirements and break regulations. Thompson and Pullman [TP07] showed that scheduling relief or rest breaks in advance is preferable over assigning or adjusting those breaks in real-time. Break scheduling is especially crucial for shift plans where many breaks need to be assigned.

The aim of the Bsp is to assign breaks to a given shift plan so that violations of staff requirements are minimized and all regulations - given as hard constraints - are fulfilled. The particular problem addressed in this thesis comes from a real-world scenario in the area of supervisory personnel. It comprises rules regarding the length of breaks and working periods, the time window breaks need to be taken in, or the positioning of breaks.

These rules are formulated as so called temporal constraints. Working periods are the time periods between breaks in which an employee is considered working. The inputs of the Bsp are a shift plan, the staff requirements and the constraints. Each shift is assigned to an employee indicating that the employee is on duty. For each shift the start, end and length are given, which are varying from shift to shift. The total number of breaktime an employee needs to take during a shift is based on the duration of the shift. The feasibility of a solution is given under certain conditions such as that breaks of the same shift do not overlap in time.

The first work solving the Bsp proposed a local search algorithm combined with a minimum-conflicts-based heuristic and a tabu search for call centers [BGM+08]. And extended version of this algorithm was presented for supervisory personnel in [BGM+10]. Beer et al. [BGM+10] further introduced benchmark instances for the Bsp - randomly generated and real-life instances . A memetic algorithm was presented in [MSW09], [Wid10] and [WM14]. Di Gaspero [GGM+10] performed shift and break scheduling within a single task using a hybrid local search-constraint programming (LS-CP) method. Recently Kocabas [Koc15] presented an exact method using mixed integer programming for a special case of the break scheduling problem. He limited the number of breaks per shift combinations to confine the search space.

The optimum for the randomly generated instances is known and not reached so far by previous works. For the real-life instances the optimum is unknown, yet we believe there is still room for improvements. In this thesis we investigate new methods for solving the Bsp. We are the first to present a mixed integer programming formulation for the general Bsp. Further, we present a large neighborhood search algorithm based on local search and a mixed integer programming procedure.

1.1 Objectives

The objectives of this thesis are:

- Definition of a mixed integer programming formulation for the general Bsp.
- Design and implementation of a large neighborhood search algorithm to optimize the Bsp. It is based on an existing local search algorithm and a mixed integer programming procedure.
- Conduction of experiments to evaluate parameters and procedures influencing the outcome of the proposed algorithm.
- Comparison of our results to results of past publications on the Bsp.

1.2 Results

The main contributions of this thesis are as follows:

- We give a mixed integer programming formulation for the Bsp without restrictions.
- A large neighborhood search algorithm to solve the Bsp is proposed. It is made up of an initialisation phase and two sub-algorithms: a local search algorithm based on [BGM+10] and a mixed integer programming algorithm. To solve the MIP formulation the solver IBM ILOG CPLEX Optimization Studio (CPLEX) [IBM16] is used.
- Different selection procedures for the choice of sub-algorithms at each iteration are tested. An elaborated procedure using a function to regulate the probability has shown to be superior.
- We evaluate different parameter settings which have a significant influence on the performance of the LNS algorithm. In total 56 experiments were performed taking a total runtime of 2,736 hours.
- The final results are discussed and compared to previously published results. The LNS algorithm outperforms our implementation of local search in most of the cases. However, it did not yet reach the upper bounds of the best known results so far.

This thesis is organized as follows: In chapter 2 we give the formal problem description of the Bsp (section 2.1) followed by a literature review on related work (section 2.2). In chapter 3 the mixed integer programming formulation for the Bsp is provided. Following this, the implementation of the large neighborhood search algorithm to solve the Bsp is presented. We give the details on our computational experiments and final results in chapter 5. Finally, we conclude and show an outline in chapter 6.

CHAPTER 2

The Break Scheduling Problem

The Break Scheduling Problem (BSP) deals with the problem of strict regulations and complex rules regarding break times during a shift. Those regulations are of different nature and determined by legal demands, ergonomic criteria and staffing requirements. They can, as for instance, refer to the duration of breaks, the time window breaks need be taken in, or the amount of break time there is per shift. Depending on the area applied to, some regulations are mandatory and considered to be hard constraints, others are mere preferences and thus soft constraints. For example security personnel working at an airport need to maintain a high level of concentration while in charge of the baggage screening monitoring. Thus it is obligatory for them to take a break after a certain period of time. In other areas on the other hand, it might be of greater importance that the number of required employees working is met optimally. There might be no security criteria affecting the break scheduling, hence breaks can be scheduled with more relaxed regulations. No matter the type of regulations, when scheduling breaks the aim is to minimise violations in staff requirements (soft constraint) whereas all hard constraints are fulfilled.

2.1 Problem Description

The original problem definition for the break scheduling problem investigated in this thesis was introduced by Beer et al. [BGM+10]. Later on Widl [Wid10], and Widl and Musliu [WM14] presented a more formal problem statement. The problem description of this work is based on these three references.

The break scheduling problem takes as input a shift schedule containing multiple *shifts* of varying length, each with a given start and end. Those shifts may overlap in time and are assigned to one employee on duty. The time frame (e.g. a week) for which the breaks are scheduled is called *planning period*. It consists of consecutive *timeslots*, which are time sequences of a fixed length (here five minutes). A timeslot paired with a specific

shift is denoted *slot*. One of three states can be assigned to a *slot*: working, on break or reacquainting (meaning adapting to a modified work situation after returning from a break). The time periods an employee is considered working (segmented by breaks) are referred to as *working periods*. The *breaktime* is a depended variable and computed based on the length of the related shift. It states the amount of slots per shift which need to be of type 'on break'.

Besides the shift schedule the *staff requirements* and *temporal constraints* are given as input. The staff requirements define the required number of working employees at each timeslot. Temporal constraints comprise regulations regarding the arrangement of breaks within a shift. In particular the duration of breaks and working periods, break positions, and the need for a lunch break are specified. A feasible solution is obtained when all temporal constraints are satisfied and the necessary amount of breaks is assigned to each shift, whereas breaks associated with the same shift are distinct from one another and not overlapping in time.

The objective is to find a feasible solution which minimizes the violation degree of the staff requirements. The violation degree is derived from the over- and undercover resulting from more, or less employees working then required.

In the following, a formal definition of the Bsp is given. Most of the definitions are based on Beer et al. [BGM+10], Widl [Wid10], and Widl and Musliu [WM14].

Definition 2.1. (Planning Period T). A set T formed by consecutive time slots $\{[a_1, a_2), [a_2, a_3), ..., [a_T, a_{T+1})\}$, all having the same length. The time points a_1 and a_{T+1} refer to the beginning and end of the planning period, respectively. In case the problem is of cyclic nature, the first time point a_1 and the last a_{T+1} are equal.

Definition 2.2. (Timeslot t). An element of the planning period T. t is a time period of fixed length (typically 5 minutes).

Definition 2.3. (Shift S). A set $S = \{t_i, t_{i+1}, ..., t_{i+m}\}$, $S \subseteq T$, of consecutive timeslots, i.e. $t_{j+1} - t_j = 1$ for $i \leq j \leq i + m$. The first timeslot t_i corresponds to the shift start S_s, and the last timeslot t_{i+m} to the shift end S_e. Shifts can overlap in time.

Definition 2.4. (Shifts $\mathcal{S}$). A set $\mathcal{S} = \{S_1, S_2, ..., S_n\}$ of n shifts. Each shift represents an employee on duty within the planning period.

Definition 2.5. (Staff Requirements $\rho(t)$). Function $\rho : T \to \mathbb{N}_0$ assigns a non-negative integer number indicating the required number of working employees to each timeslot in the planning period.

Definition 2.6. (Slot). A slot denotes a timeslot in a particular shift. A slot can be set to one of three different states:

- working (w-slot): an employee is working in this slot
- on break (b-slot): an employee is on break

- reacquainting (r-slot): an employee is taking a reacquaintance pause

A reacquaintance pause or r-slot is assigned exclusively to a slot directly following a break (sequence of b-slots). It is necessary to give the employee one timeslot to become reacquainted with the current situation at the workplace after a break. The employee is neither considered working regarding staff requirements nor being on break.

Definition 2.7. (Break B). A set B of consecutive b-slots corresponding to a particular shift. Each break has a start B_s and end B_e. There are two different types of breaks: monitor and lunch breaks.

Definition 2.8. (Breaks $\mathcal{B}$). A set $\mathcal{B} = \{B_1, B_2, ..., B_m\}$ of m breaks. Each particular shift S_i a set $\mathcal{B}_{S_i}$ is assigned.

Definition 2.9. (Working Period W). A set W of one r-slot and consecutive w-slots in a particular shift. Each working period has a start W_s and end W_e.

Definition 2.10. (Working Periods $\mathcal{W}$). A set $\mathcal{W} = \{W_1, W_2, ..., W_k\}$ of k working periods. Each particular shift S_i a set $\mathcal{W}_{S_i}$ is assigned.

Definition 2.11. (Duration). The duration of a shift $|S|$, break $|B|$ or working period $|W|$ in timeslots can be calculated by counting the number of timeslots within the interval from start to end $[start, end]$, endpoints included.

Definition 2.12. (Breaktime $\tau(|S|)$). A function τ:$\{|S_1|, ..., |S_n|\} \rightarrow \mathbb{N}$ specifying the required amount of break time in timeslots (b-slots) for each shift S_i according to its duration $|S_i|$.

Definition 2.13. (Temporal Constraint):

C_1 **Break Positions (p_1,p_2).** In each shift S_i a break may start earliest p_1 timeslots from the shift start S_{is} and end latest p_2 timeslots to the shift end S_{ie}. See figure 2.1 for a graphical example.

$$\begin{aligned} &\forall B_j \in \mathcal{B}_{S_i} \ \Big(S_{is} + (p_1 - 1) \leq B \ \wedge \ B_{je} \leq S_{ie} - (p_2 - 1)\Big) \\ \text{or} \quad &\forall B_j \in \mathcal{B}_{S_i} \ \Big(S_{is} + (p_1 - 1) \leq B_{js} \leq S_{ie} - (p_2 - 1) - (|B_j| - 1)\Big) \end{aligned} \tag{2.1}$$

C_2 **Lunch Break (h,g,l_1,l_2).** Each shift S_i with a minimum duration h must contain at least one lunch break. A lunch break has a minimum length g and is located within a certain time window (earliest possible start: l_1 timeslots from the shift

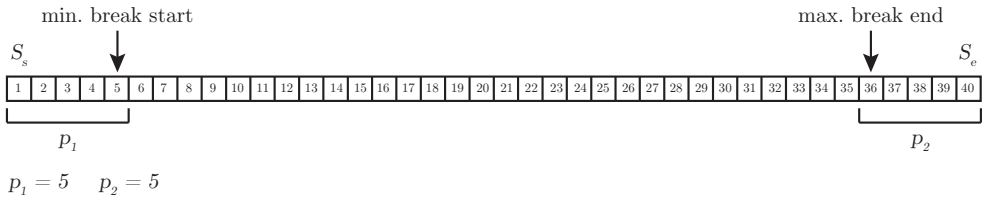

Figure 2.1: A graphical example of the temporal constraint C_1 (Break Positions). A break may start earliest at slot 5 and end latest at slot 36 where the shift length is 40 slots, and p_1 and p_2 are each 5 slots.

start S_{is}; latest possible end: l_2 timeslots from the shift start S_{is}). See figure 2.2 for a graphical example.

$$\begin{aligned} &\forall S_i \left(|S_i| \geq h\right) \exists B_j \in \mathcal{B}_{S_i} \\ &\Big(|B_j| \geq g \ \wedge \ S_{is} + (l_1 - 1) \leq B_{js} \ \wedge \ B_{je} \leq S_{is} + (l_2 - 1)\Big) \\ &\text{or} \quad \forall S_i \left(|S_i| \geq h\right) \exists B_j \in \mathcal{B}_{S_i} \\ &Big(|B_j| \geq g \ \wedge \ S_{is} + (l_1 - 1) \leq B_{js} \leq S_{is} + (l_2 - 1) - (|B_j| - 1)\Big) \end{aligned} \tag{2.2}$$

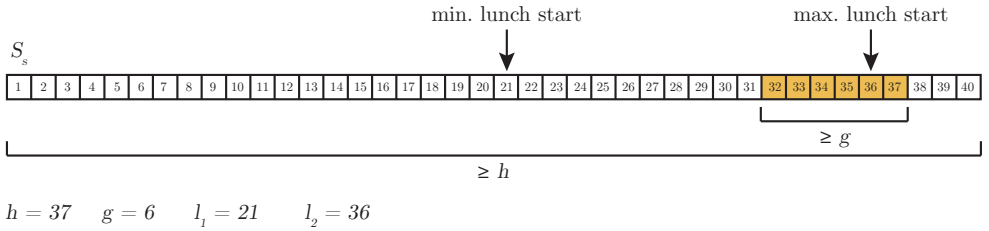

Figure 2.2: A graphical example of the temporal constraint C_2 (Lunch Break). A shift with the length greater or equal 37 slots (h) must have a lunch break, which may start earliest at slot 21 (l_1) and latest at slot 36 (l_2), and has a length of greater or equal 6 slots (g).

C_3 **Duration of Working Periods (w_1,w_2).** The duration of each working period $|W_j|$ must range between w_1 and w_2.

$$\forall W_j \in \mathcal{W}_{S_i} \ \ (w_1 \leq |W| \leq w_2) \tag{2.3}$$

C_4 **Minimum Break Durations (w,b).** In case a working period W_j has a duration greater or equal w, the following break B_{W_j} must last at least b timeslots. See figure 2.3 for a graphical example.

$$\forall W_j \in \mathcal{W}_{S_i} \ \ (|W_j| \geq w) \implies \exists B_{W_j} \ \ (|B_{W_j}| \geq b) \tag{2.4}$$

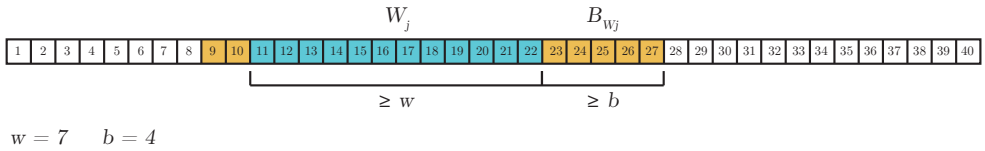

Figure 2.3: A graphical example of the temporal constraint C_4 (Minimum Break Durations). If the length of a working period exceeds 7 slots (w), the following break (B_{W_j}) must last at least 4 slots (b).

C_5 **Duration of Breaks (b_1,b_2).** The duration of a break $|B_j|$ must range between b_1 and b_2.

$$\forall B_j \in \mathcal{B}_{S_i} \ \ (b_1 \leq |B_j| \leq b_2) \tag{2.5}$$

Feasibility of Solution:

A solution is feasible when C_1 to C_5 are fulfilled and the following three conditions are met:

F_1 **Break Boundaries.** Each break B_j associated with one shift S_i lies entirely within the shift.

$$\forall B_j \in \mathcal{B}_{S_i} \ \ (S_{is} \leq B_{js} \leq B_{je} \leq S_{ie}) \tag{2.6}$$

F_2 **Break Succession.** Two distinct breaks (B_j, B_k) associated with the same shift S_i are successive and not overlapping in time.

$$\forall B_j, B_k (j \neq k) \in \mathcal{B}_{S_i} \ \ \Big((B_{js} \leq B_{je} \leq B_{ks} \leq B_{ke} \ \vee \ B_{ks} \leq B_{ke} \leq B_{js} \leq B_{je})\Big) \tag{2.7}$$

F_3 **Breaktime.** In each shift S_i the sum of lengths of its associated breaks (sum of b-slots) is equal to the required amount of breaktime $\tau(|S_i|)$.

$$\forall S_i \ \ \Big(\sum_{B_j \in S_i} |B_j| = \tau(|S_i|)\Big) \tag{2.8}$$

Objective:

The objective is to find an assignment of b- and r-slots for each shift such that the solution is feasible as stated above and the following objective function F is minimised:

$$F = \sum_{t \in T} (w_o * overcover_t + w_u * undercover_t) \tag{2.9}$$

where

- $overcover_t$ is the calculated overcover at timeslot t; overcover = if (staff requirement at t - number of b-slots at t - number of r-slots at t) is greater than 0
- $undercover_t$ is the calculated undercover at timeslot t; undercover = if (staff requirement at t - number of b-slots at t - number of r-slots at t) is less than 0
- w_o is the weight for overcover
- w_u is the weight for undercover

2.2 Related Work

Previously the break scheduling problem was covered by the related shift scheduling problem and only a small number of breaks were considered. The break assignment is carried out in the course of the shift scheduling. Break scheduling as autonomously problem distinguishes itself hereof by the larger number of breaks and that scheduled shifts are given as input in the majority of cases.

So far no indications were given whether breaks scheduled along side with shifts or scheduled in a subsequent phase is the better practise. The decision to combine both tasks into one is based on many factors such as the complexity of scheduling rules or the trade-off between computation time and memory needed. Considering both problems at once deals with a considerably larger search space. Still this procedure might result in more optimal solutions. In the following, a literature overview of shift scheduling with breaks and break scheduling is given.

Shift Scheduling with Breaks

Dantzig presented the original set-covering formulation [Dan54] for the shift scheduling problem. He considered shift planning for toll booth staff with up to three breaks. This formulation requires an integer variable for each shift, shift start, duration, break and time window for breaks, which makes it impractical for large numbers of shifts. Bechtold and Jacobs [BJ90] proposed an implicit integer linear programming formulation with flexible break assignment. It is meant to be superior to Dantzig's formulation in terms of execution time, memory requirements and being able to generate optimal solutions for larger problems. Based on Bechtold and Jacobs' model, Thompson [Tho95] developed an integer programming model handling breaks and shifts implicitly, which was able achieve improvements. Thompson further applied his approach to the special case of scheduling shifts and breaks where employees have limited time availability [Tho96]. Aykin proposed another implicit integer programming model for shift scheduling [Ayk96], which uses variables for every shift-break combination to represent break placement in a certain shift. It requires a much smaller number of variables then the previously presented set-covering formulations and has shown to be useful in solving large shift scheduling problems optimally. Aykin [Ayk00] compared Bechtold and Jacobs' model to his concluding Bechtold and Jacobs' formulation requires a smaller number of variables although contains more constraints and needs more time to compute an optimal solution.

More recently Gärtner et al. [GMS05] introduced a heuristic algorithm for solving shift scheduling with breaks. Experiences with a real-life problem - a large European airport - are given. Breaks are scheduled separately from shifts, which makes the problem related to BSP. Tellier and White [TW06] proposed a tabu search algorithm for a shift scheduling problem which occurs in contact centres. Thompson and Pullman [TP07] address the significance of scheduling breaks. They compare three different approaches to schedule relief breaks: scheduling relief breaks alongside with shifts, no relief break scheduling at all and scheduling breaks subsequent to shifts. Their investigation showed that scheduling relief or rest breaks in advance is preferable over assigning those breaks in real-time.

Rekik et al. [RCS10] presented another implicit model, however with a similar problem formulation to BSP. Breaks can be fractioned and distributed under some conditions within the shift, comparable to the break time and placing of breaks with variable lengths in BSP. Restrictions in work stretch duration - similar to working periods in BSP - controls the positioning of breaks. This approach has produced solutions with reduced numbers of employees for several instances. Quimper and Rousseau [QR10] proposed a large neighborhood search approach for a multi-activity shift scheduling problem. The scheduling rules are modelled with formal languages such as regular languages and context-free grammars. Côté et al. presented another grammar-based model for multi-activity shift scheduling [CGQR11] [CGR11]. They generated a mixed integer programming model with an implicit problem formulation and using context-free grammars.

Break Scheduling

As the need arose (e.g. from call centres or airports) to perform break scheduling with a much higher number of breaks and more complex requirements, break scheduling was considered detached from shift scheduling by researchers. Break scheduling was formally first introduced by Beer et al. [BGM+10] and considered as a constraint-satisfaction optimization problem. They present a scheduling tool named Operating Hours Assistant which performs shift and break scheduling in two different phases. The break scheduling system uses local search combined with a minimum-conflicts-based heuristic, a tabu search and a simulated-annealing algorithm. For the generation of the initial solution a formulation for a simple temporal problem is given, which is solved by a randomized version of the Floyd-Warshall shortest-path algorithm [PS82]. To escape local optima they apply the random walk strategy. Beer et al. further introduced real-life benchmark instances and randomly generated datasets. A similar break scheduling problem which occurs in call centres was presented by Beer et al. [BGM+08] and Schafhauser [Sch10].

A first memetic algorithm was presented in 2009 [MSW09]. An improved version of it was proposed in 2010 [Wid10] and most recently in 2014 including a proof of complexity [WM14]. For initialisation a set of feasible break patterns is created making use of the same simple temporal problem as Beer at al. [BGM+10]. Then a simple local search is executed on the initial solution. Two variations of a memetic algorithm are proposed, both combining a genetic algorithm and a local search procedure based on three different neighborhoods. For the selection of memes a penalty system is used to avoid local optima.

Widl and Musliu reached new upper bounds for almost all instances. Further, they showed that Bsp - under the condition that all possible break patterns for each shift are given explicitly as part of the input - is NP-complete. A proof of NP-completeness is given by re-formulating Bsp as a decision problem.

At about the same time in 2010 Di Gaspero et al. [GGM+10] proposed a hybrid local search-constraint programming (LS-CP) method for performing shift design and break scheduling together within a single task. Local search is used to schedule shifts and a constraint programming model is applied to assign breaks. This approach did not result in improvements. A literatur survey of break and shift scheduling, two approaches based on local search techniques and a real-life case study are given in Di Gaspero et al. [DGGM+13].

Recently Kocabas [Koc15] presented an exact method using mixed integer programming for a special case of the break scheduling problem with a limited number of break combinations. To confine the search space the number of possible break patterns is reduced. His approach outperforms Widl and Musliu [WM14] for most of the random generated datasets, but his model could not be used for the real-life instances introduced by Beer et al. [BGM+10].

CHAPTER 3

Mixed Integer Programming Formulation for Break Scheduling

The nature of the break scheduling problem (Bsp) makes it well suited for being formulated as a mixed integer programming problem. Start and end of shifts and breaks as well as durations measured in timeslots can be easily expressed as integer variables. Binary decision variables can indicate whether a slot is assigned a break or not. In the past integer programming was applied successfully to shift scheduling problems, such as [BJ90], [Tho95], [Ayk96], [CGQR11] and [CGR11]. To the best of our knowledge an integer programming fomulation for the break scheduling problem (Bsp) was only given by Kocabas [Koc15]. However, in order to narrow down the search space he modelled a restricted version of the Bsp. He proposed an explicit formulation of the problem, but with a reduced number of possible break combinations. Two assumptions are made in the course of the restrictions: 1. Every monitor break has a duration of exactly 2 timeslots; 2. Three monitor breaks are to be positioned before the lunch break and the remaining monitor breaks are after the lunch break.

In the following sections, our mixed integer programming formulation for Bsp without restrictions is given. First we discuss our implementation. Second we introduce important variables and constants, then the constraints are given in the form of expressions. Finally we specify the fitness function and the objective.

3.1 Mixed Integer Programming Implementation

In contrast to previous work, in our formulation breaks are not restricted, but free to be positioned anywhere as long as all constraints are fulfilled. Single monitor breaks can be merged into one larger break. Thus, additionally to the duration of breaks the number of breaks within a shift is variable, although the total breaktime is fixed. This results in

a much higher flexibility in generating solutions but also in a considerably larger search space.

For the actual implementation the planning period T is set to be one week and is of cyclic nature. Each timeslot t has a length of 5 minutes. Shifts $\mathcal{S}$ are given as input including the breaktime. Further input are the staff requirements and all temporal constraints. A solution of the problem is represented by a set of break starts and break durations for each shift (see Definition 3.1 and 3.2). The fitness is calculated using the auxiliary variables $BreaksPerTimeslot$ and $rSlotsPerTimeslot$ (see Definition 3.3 and 3.4), which indicate whether a employee is on break or reacquainting at a certain timeslot. To compute the fitness first the coverage at each timeslot is calculated, then the overcover and undercover is determined (Definition 3.4). This model was written for the solver IBM ILOG CPLEX Optimization Studio (CPLEX) [IBM16].

3.2 Variables and Constants

For each shift S_i the following variables and constants are declared:

Definition 3.1. ($Breaks$). An integer array storing the timeslot of each break start. Values are restricted to possible timeslots within the shift leaving out timeslots before the earliest break start and after the latest break end. The length of the array is derived from the maximum possible number of breaks for the given breaktime.

Definition 3.2. ($Lengths$). An integer array storing the duration (in number of timeslots) of each break. The location in the array corresponds to the break in the same location in $Breaks$. Combining the break start in $Breaks$ and the break duration the timeslot of the break end can be computed. The maximum allowed value is the maximum break length according to the Bsp formulation. A break of length 0 would indicate the non-existence of a break, no matter which value is stored in $Breaks$.

Definition 3.3. ($BreaksPerTimeslot$). A boolean array storing 1 if there is a break or 0 if there is no break for each timeslot in the shift. $BreaksPerTimeslot$ is an auxiliary variable for the fitness calculation. Only timeslots for possible break positioning are considered.

Definition 3.4. ($rSlotsPerTimeslot$). A boolean array storing 1 if a slot is assigned the state reacquainting (r-slot), or 0 in all other cases. $rSlotsPerTimeslot$ is an auxiliary variable for the fitness calculation. Only timeslots for possible break positioning (plus one for the succeeding r-slot) are considered.

Definition 3.5. ($LunchBreak$). A boolean array storing 1 if a break is considered a lunch break, or 0 otherwise. The location in the array corresponds to the break in the same location in $Breaks$. If a shift requires a lunch break is depending on the duration of the shift as stated in the constraint C_2 (Lunch Break).

Definition 3.6. (*lunchMinStart*). A constant holding the earliest possible timeslot at which the lunch break of a shift is allowed to start. Its value is depending on the shift start.

Definition 3.7. (*lunchMaxStart*). A constant holding the latest possible timeslot at which the lunch break of a shift is allowed to start. Its value is depending on the shift start.

Definition 3.8. (*breakMaxEnd*). A constant holding the latest possible timeslot at which a break of a shift is allowed to end. Its value is depending on the shift end.

A graphical illustration of Definitions 3.1 to 3.5 is given in figure 3.1.

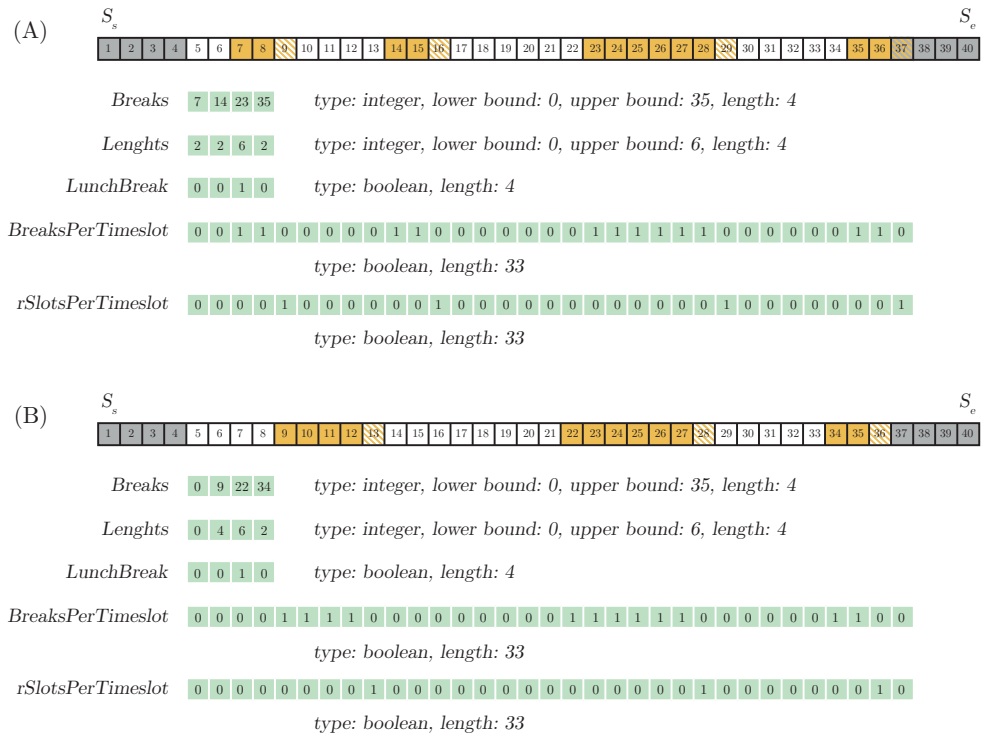

Figure 3.1: Two examples (A) and (B) showing the mixed integer programming variables and their settings for a given break assignment of a shift with length 40 and a possible break positioning from slot 5 to 35. (A) shows four breaks. (B) shows three breaks. Note that in (B) the first position of *Breaks* and *Lengths* is set to 0 indicating a non-existing break.

3.3 Expressions

Break Succession. In each shift S_i breaks need to be succeeding and not overlapping in time. Let x be a position in an array (e.g. $Breaks_{S_i}$) ranging from 1 to the length of the array, denoted as $length()$. And let $Breaks_{S_i}[x]$ be the value stored at the position x.

$$\forall x \in \{1, .., length(Breaks_{S_i})\} \left(Breaks_{S_i}[x] + Lengths_{S_i}[x] \leq Breaks_{i+1} \right) \quad (3.1)$$

Break Position. A break is allowed to start earliest p_1 timeslots after the shift start and ends latest p_2 timeslots before the shift ends. The restriction of the earliest possible break start is already considered in $Breaks$ due to value restrictions. The latest possible break end is covered by expression 3.2. Only breaks with a minimum length of 2 (else they are non-existing) are considered.

$$\forall x \in \{1, .., length(Breaks_{S_i})\} \Big(Lengths_{S_i}[x] \geq 2 \implies Breaks_{S_i}[x] + Lengths_{S_i}[x] \leq breakMaxEnd_{S_i} \Big) \quad (3.2)$$

Lunch Break. Each shift S_i exceeding a specific duration must contain at least one lunch break. This lunch break must be positioned within a specified time window based on the shift start. In expression 3.3 a reference in $LunchBreak$ is made whether a break is a lunch break or not. A break is a lunch break if it is positioned within a certain time window and has a length equal to or greater g. Expression 3.4 is needed to ensure that at least one lunch break exists.

$$\forall x \in \{1, .., length(Breaks_{S_i})\} \left(LunchBreak_{S_i} = \begin{cases} 0 & \text{if } Lengths_{S_i}[x] < g \\ 0 & \text{if } Breaks_{S_i}[x] < lunchMinStart_{S_i} \\ 0 & \text{if } Breaks_{S_i}[x] > lunchMaxStart_{S_i} \\ 1 & \text{otherwise (by default)} \end{cases} \right) \quad (3.3)$$

$$\sum_{x=1}^{length(Breaks_{S_i})} LunchBreak_{S_i}[x] \geq 1 \quad (3.4)$$

Duration of Working Periods. The duration of each working period must range between a lower bound w_1 and upper bound w_2. Only breaks with a minimum length of 2 (else they are non-existing) are considered for deriving the working periods. A working period lies within the break end of the previous break and the break start of the following one. Expression 3.5 deals with working periods between breaks. The first (from shift start to first break) and last working period (from last break to shift end) are addressed separately. Expression 3.6 ensures the maximum duration

of the first working period and expression 3.7 the maximum duration of the last working period. The minimum duration of the first working period is implicitly guaranteed by restricting the possible values for $Breaks$ so a break can start earliest after the minimum first working period. The minimum duration of the last working period is ensured by expression 3.2 so that the last break ends latest before the minimum last working period. Let $Breaks_{S_i}[first]$ be the first break in $Breaks_{S_i}$ which is not 0 and $Breaks_{S_i}[last]$ be the last break in $Breaks_{S_i}$.

$$\forall x \in \{1, .., length(Breaks_{S_i})\} \Big(Lengths_{S_i}[x] \geq 2 \implies Breaks_{S_i}[x+1] - (Breaks_{S_i}[x] + Lengths_{S_i}[x]) \leq w_2 \Big) \quad (3.5)$$

$$Breaks_{S_i}[first] - shiftStart_{S_i} \leq w_2 \quad (3.6)$$

$$shiftEnd - (Breaks_{S_i}[last] + Lengths_{S_i}[last] - 1) \leq w_2 \quad (3.7)$$

Minimum Break Durations. If a working period is equal to or greater a certain length w, the following break must have a duration of at least b timeslots. Expression 3.8 deals with working periods between breaks. The minimum break duration for the first working period is addressed separately. Expression 3.9 ensures that the first break has the required minimum break duration according to the duration of the preceding working period.

$$\forall x \in \{1, .., length(Breaks_{S_i})\} \Big(Lengths_{S_i}[x] \geq 2 \wedge Breaks_{S_i}[x+1] - (Breaks_{S_i}[x] + Lengths_{S_i}[x]) \geq w \implies Lengths_{S_i}[x+1] \geq b \Big) \quad (3.8)$$

$$Breaks_{S_i}[first] - shiftStart_{S_i} \geq w \implies Lengths_{first} \geq b \quad (3.9)$$

Duration of Breaks. The duration of each break must be within the range from a minimum b_1 to a maximum b_2. Additionally, a break duration of length 0 is allowed in our formulation. In case there are breaks with longer duration, there are less breaks than maximum possible number of breaks. Thus, some positions in the array $Breaks$ and $Lengths$ are not assigned any breaks. This is indicated by a break duration of length 0.

$$\forall x \in \{1, .., length(Breaks_{S_i})\} \Big(Lengths_{S_i}[x] = 0 \vee (Lengths_{S_i}[x] \geq b_1 \wedge Lengths_{S_i}[x] \leq b_2) \Big) \quad (3.10)$$

Non-Existing Breaks In case there are less than the maximum possible number of breaks, some positions in the array *Breaks* and *Lengths* are not assigned any breaks. Besides the break duration being set to 0, the break start is enforced to be 0 as well. This is necessary so that all non-existing breaks are positioned and grouped together in the beginning of the array.

$$\forall x \in \{1, .., length(Breaks_{S_i})\} \left(Lengths_{S_i}[x] = 0 \implies Breaks_{S_i}[x] = 0 \right) \quad (3.11)$$

Breaktime. A number of required breaktime is assigned to each shift S_i. It is based on the duration of the shift. The sum of break lengths of breaks associated with a shift (sum of b-slots) must be equal to the required amount of breaktime.

$$\sum_{x=1}^{length(Breaks_{S_i})} Lengths_{S_i}[x] = breaktime_{S_i} \quad (3.12)$$

B-Slots. For each shift S_i a reference in $BreaksPerTimeslot$ is made if there is a break (b-slot) at this timeslot. The timeslot is marked as b-slot with the value 1 in case the timeslot lies within a break. Let ts be a position in an array representing a set of timeslots (e.g. $BreaksPerTimeslot$) ranging from 1 to the length of the array, denoted as length(). And let $BreaksPerTimeslot_{S_i}[ts]$ be the value stored at the position ts.

$$\forall x \in \{1, .., length(Breaks_{S_i})\}\ \forall ts \in \{1, .., length(BreaksPerTimeslot_{S_i})\}$$
$$\left(Lengths_{S_i}[x] \geq 2 \implies \right.$$
$$\left. BreaksPerTimeslot_{S_i}[ts] = \begin{cases} 1 & \text{if } Breaks_{S_i}[x] \leq (ts + shiftStart_{S_i} - 1) \\ & < Breaks_{S_i}[x] + Lengths_{S_i}[x] \\ 0 & \text{otherwise (by default)} \end{cases} \right) \quad (3.13)$$

R-Slots. For each shift S_i a reference in $rSlotsPerTimeslot$ is made if there is a reacquaintance pause (r-slot) at this timeslot. There are four different cases. The timeslot is an r-slot, if the current timeslot is not marked a b-slot and the preceding timeslot is. The timeslot is not an r-slot in case the timeslot is marked a b-slot, the

preceding timeslot is not marked a b-slot or if it is the first timeslot of the shift.

$$\forall ts \in \{1, .., length(BreaksPerTimeslot_{S_i})\}$$
$$\left(rSlotsPerTimeslot_{S_i}[ts] = \begin{cases} 1 & \text{if } BreaksPerTimeslot_{S_i}[ts-1] = 1 \\ & \quad \wedge BreaksPerTimeslot_{S_i}[ts] = 0 \\ 0 & \text{if } BreaksPerTimeslot_{S_i}[ts] = 1 \\ 0 & \text{if } BreaksPerTimeslot_{S_i}[ts-1] = 0 \\ 0 & \text{if } i = 0 \end{cases} \right) \tag{3.14}$$

3.4 Fitness Function and Objective

Definition 3.9. (*Overcover*). An integer array storing the overcover for each timeslot of the planning period. Overcover is reached when there are more employees working than required.

Definition 3.10. (*Undercover*). An integer array storing the undercover for each timeslot of the planning period. Undercover is reached when there are less employees working than required.

Definition 3.11. ($nrB/Rslots$). An integer array storing the sum of b- and r-slots for each timeslot of the planning period.

Definition 3.12. ($requiredStaff$). An integer array storing the number of required staff at each timeslot of the planning period. Values are given as input of the problem instance.

Definition 3.13. ($nrShifts$). An integer array storing the number of shifts at each timeslot of the planning period. Values are given as input of the problem instance.

Coverage Calculation. First for each timeslot the sum of all b- and r-slots of all shifts needs to be established ($nrB/Rslots$). $count(shifts)$ denotes the number of shifts. Then it is determined if there is an under- or overcover. The *Coverage* is calculated by subtracting the required number of employees $requiredStaff$ and the number of b- and r-slots $nrB/Rslots$ from the number of shifts $nrShifts$ (equal to the available number of employees) (expression 3.16). The absolute value of the resulting *Coverage* is written to *Overcover* and *Undercover*, respectively (expression 3.17, 3.18).

$$\forall ts \in \{1, .., length(nrB/Rslots)\} \left(nrB/Rslots[ts] = \right.$$
$$\left. \sum_{i=1}^{count(shifts)} (BreaksPerTimeslot_{S_i}[ts] + rSlotsPerTimeslot_{S_i}[ts]) \right) \tag{3.15}$$

$$\forall ts \in \{1, .., length(Coverage)\} \Big(Coverage[ts] = nrShifts[ts] - nrB/Rslots[ts] - requiredStaff[ts]\Big) \quad (3.16)$$

$$\forall ts \in \{1, .., length(Overcover)\} \left(Overcover[ts] = \begin{cases} \mid Coverage[ts] \mid & \text{if } Coverage[ts] > 0 \\ 0 & \text{otherwise (by default)} \end{cases}\right) \quad (3.17)$$

$$\forall ts \in \{1, .., length(Undercover)\} \left(Undercover[ts] = \begin{cases} \mid Coverage[ts] \mid & \text{if } Coverage[ts] < 0 \\ 0 & \text{otherwise (by default)} \end{cases}\right) \quad (3.18)$$

Fitness. The $Fitness$ is computed as weighted sum of $Undercover$ and $Overcover$ over all timeslots. The weights w_o and w_u are given as input of a problem instance.

$$Fitness = \sum_{ts=1}^{length(Coverage)} w_o * Overcover[ts] + w_u * Undercover[ts] \quad (3.19)$$

Objective. The objective is to minimize the $Fitness$.

Figure 3.2 shows an example with five shifts and the calculation of $Coverage$, $Overcover$, $Undercover$ and $Fitness$ with given total number of shifts and required staff per timeslot of the planning period.

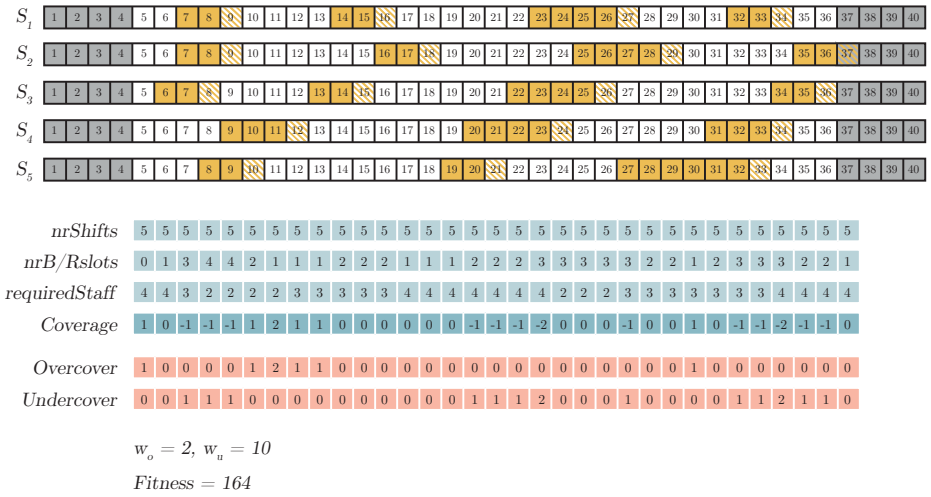

Figure 3.2: A graphical example of the calculation of $Coverage$, $Overcover$, $Undercover$ and $Fitness$ of five shifts. First $nrB/Rslots$ needs to be determined by accumulating all b- and r-slots for each timeslot. Then the coverage is calculated by subtracting $nrB/Rslots$ and $requiredStaff$ from $nrShifts$. Finally the over- and undercover is derived. Setting the weights w_o to 2 and w_u to 10, the resulting fitness amounts 216.

CHAPTER 4

Solving the Break Scheduling Problem

To solve the break scheduling problem, we propose a large neighborhood search algorithm. It is made up of an initialisation phase and two sub-algorithms: the local search and a mixed integer programming (MIP) algorithm. To solve the MIP formulation the solver CPLEX is used.

In the following, we first explain the large neighborhood search algorithm. Then the description of the initialisation and its sub-algorithms local search and mixed integer programming is given.

4.1 Large Neighborhood Search

Large neighborhood search (LNS) was first introduced by Shaw in 1998 [Sha98]. Neighborhoods are generated implicitly by a *destroy* and *repair* method. First the destroy method destructs a part of the current solution, leaving the solution to be infeasible. Then the repair method rebuilds the solution to be feasible again. The destroy method is usually designed in a way such that different parts of the solution are destroyed in each iteration. The degree of destruction determines the size of the resulting neighborhood. Choosing the repair method one can decide upon using an optimal repair operation or a heuristic one. A good overview on LNS is given by Pisinger and Ropke [PR10].

For solving the Bsp we suggest a combination of local search and mixed integer programming called in separate sub-algorithms. The first sub-algorithm local search (see chapter 4.3) is based on Beer et al. [BGM+10] and uses two moves to optimize the solution: *swap* and *assignment*. Additionally, a random-walk procedure is used to escape local optima. Both moves focus only on a single break, which results in rather smaller neighborhoods, but speeds up the time needed in each iteration. The second sub-algorithm generates a

larger neighborhood by removing the break assignment of an entire shift (see chapter 4.4). The solution is reconstructed with an optimal repair method using the solver CPLEX.

To optimize the solution the sub-algorithms are applied alternately. In each iteration a sub-algorithm is chosen by a selection procedure. We propose a selection procedure which is based on a probability function. It is dependent on several parameters and gradually increases with the time elapsed e. e is normalized by the time limit for the runtime of LNS μ. λ regulates the rate of increase of the probability. The maximum probability is set by the probability factor η. A graphical illustration of the sub-algorithm probability is shown in figure 4.1. In the following, a description of the parameters as well as the probability function is given.

Probability Curve Gradient λ. A parameter which determines the gradient of the probability curve.

LNS Time Limit μ. A parameter which determines a time limit for the overall runtime of LNS.

Probability Factor η. A parameter which regulates the probability for a sub-algorithm to be selected.

Sub-Algorithm Probability A function determining the probability for a sub-algorithm to be chosen. The probability is dependent on the time elapsed e, which is normalized to be within 0 and 1.

$$\frac{\lambda^{\frac{e}{\mu}} - 1}{\lambda - 1}\,\eta \tag{4.1}$$

Algorithm 4.1 shows the framework of our LNS algorithm. After the initialization of all relevant variables an initial solution is produced using an initialization procedure (see chapter 4.2). Then the optimization procedure (algorithm 4.1: line 9-19) is executed until the runtime exceeds the predefined LNS time limit μ, optimizing the current solution. Sub-algorithms are chosen by the sub-algorithm probability function (function 4.1). The further the time elapsed the higher the probability for the mixed integer programming (MIP) sub-algorithm to be chosen. The maximum probability is given by the probability factor η. The returned solution is kept as current solution $\mathcal{K}_{current}$ and accepted as best solution $\mathcal{K}_{best}$ in case it is better or equal to the best known solution so far.

For the implementation the following additional variables are used:

Definition 4.1. (Solution $\mathcal{K}$). A mapping of a set breaks $\mathcal{B}$ to each shift S in $\mathcal{S}$: $(\mathcal{S},\{\mathcal{B}_{S1},\mathcal{B}_{S2},...,\mathcal{B}_{Sn}\})$.

Definition 4.2. (Coverage $\mathcal{C}$). Function $\alpha : T \to \mathbb{N}$ assigns an integer number indicating the coverage at each timeslot in the planning period. The coverage at a timeslot is determined by the number of employees on duty (equivalent to number of shifts) minus the number of employees on break minus the required number of employees. A negative coverage results in an undercover whereas a positive results in an overcover.

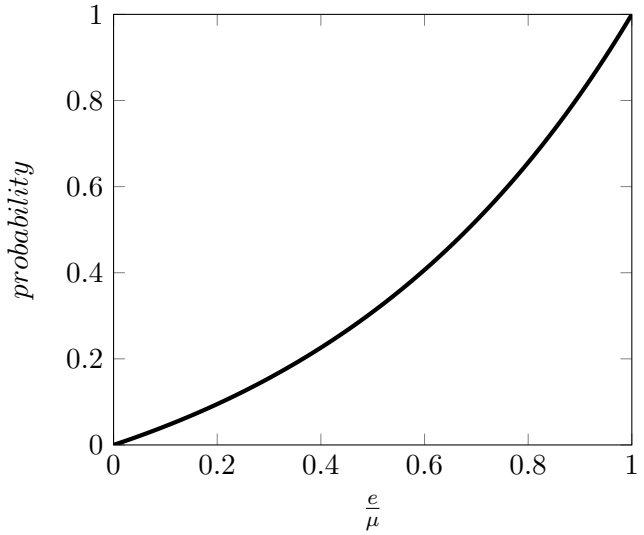

Figure 4.1: Probability function for a sub-algorithm to be chosen. It is dependent on the time elapsed e, which is normalized by the LNS time limit for the overall runtime μ. The further advanced the time the higher the probability.

In total three procedures for choosing a sub-algorithm were tested (see chapter 5): random-selector (RS), timebound-selector (TS) and probability-selector (PS). RS decides randomly at each iteration which sub-algorithm to choose. It leads to rather poor results as the MIP sub-algorithm, which is very time consuming, gets chosen too often. As a consequence the number of overall iterations is much lower as well as the number of improvements. TS switches between the sub-algorithms each time no improvements were made for a specified amount of time. Those time limits are set differently for local search and MIP in accordance with their general time consumption. TS reached better solutions but still not performing as desired. Experience showed that using local search the greatest improvement steps happen in the beginning of the runtime and become less and less over time. TS does not adapt to those changes. In contrast to RS and TS choosing sub-algorithms by using a probability function (PS) regulates the usage of the sub-algorithms throughout the runtime. Due to the characteristics of the sub-algorithms a function was selected, which uses the MIP sub-algorithm very rarely in the beginning and increases the usage over time, limiting it to a maximum probability to prevent it from being to dominant. The sub-algorithm probability function is meant to countermeasure the stagnation of improvements experienced in local search by allowing the MIP sub-algorithm gradually to be chosen more often but giving local search still the opportunity to reach improvements. In general this procedure is meant to cause a vast exploration of the search space throughout the runtime. As such it is the superior procedure and was used for experiments in this work.

Algorithm 4.1: BreakScheduler

load problem instance;
initialize $\mathcal{S}$, $\mathcal{C}$;
for *each* S *in* $\mathcal{S}$ **do**
    determine the need for a lunch break;
    calculate breaktime;
end
$\mathcal{K}_{current} \leftarrow$ Initialisation($\mathcal{C}$, $\mathcal{S}$);
$\mathcal{K}_{best} \leftarrow \mathcal{K}_{current}$;
repeat
    probability $\leftarrow$ random number between 0 and 100;
    if $\frac{\lambda^{\frac{e}{\mu}}-1}{\lambda-1} * \eta <=$ *probability* **then**
        // e: time elapsed; μ: time limit for overall runtime
        $\mathcal{K}_{current} \leftarrow$ LocalSearch($\mathcal{K}_{current}$, $\mathcal{S}$);
    else
        $\mathcal{K}_{current} \leftarrow$ MixedIntegerProgramming($\mathcal{K}_{current}$, $\mathcal{S}$);
    end
    if $\mathcal{K}_{current}$.*fitness* $<= \mathcal{K}_{best}$.*fitness* **then**
        $\mathcal{K}_{best} \leftarrow \mathcal{K}_{current}$;
    end
until *runtime exceeds* μ;
output $\mathcal{K}_{best}$;

4.2 Initialisation

For the initialisation a simple heuristics using a list of allowed break positions is applied. The break assignment for each shift is performed sequentially. As a first step the position of the lunch break is set. Then the list of allowed break positions is generated by applying the constraints C_1 (Break Positions) and the minimum duration of working periods (part of C_3), i.e. removing the positions before the minimum break start and after the maximum break end, etc. The position of the lunch break as well as the minimum duration of adjacent working periods are removed from the list. Positions for monitor breaks are chosen randomly out of the list of allowed break positions. Each time a break is added the list is reduced by the break position and the minimum duration of adjacent working periods. In case the attained solution is not legal in regard to the constraints C_3 (Duration of Working Periods), C_4 (Minimum Break Durations) and $F3$ (Breaktime) the list is reverted to the point it only contained the lunch break. Monitor break positions are chosen anew. Compared to a solely randomly selected break pattern this method is significantly faster due to ruling out solutions which are illegal in terms of the constraints C_1 (Break Positions), C_2 (Lunch Break), C_5 (Duration of Breaks) and the minimum duration of working periods (part of C_3).

The pseudo-code for the heuristics is shown in algorithm 4.2. It takes as input the initial coverage $\mathcal{C}$ (containing only the number of employees on duty minus the required number of employees) and a list of all shifts $\mathcal{S}$. The generation of a break pattern for each shift is separated into two phases: 1) insertion of the lunch break and 2) insertion of monitor breaks. The list of allowed break positions ($freeSlots$) is initially set to all timeslots of the shift within the positions p_1 (earliest timeslot a break may start) and p_2 (latest timeslot a break may end). In the first phase it is determined if the shift needs a lunch break according to its duration. In case the shift duration exceeds a certain length a lunch break is needed, which is set to the middle of the shift (see algorithm 4.2: line 4). Should this position violate the constraint C_2 the lunch break is either set to the earliest or latest possible point (algorithm 4.2: lines 5-9). Then the $freeSlots$ list is reduced by the timeslots of the lunch break and the minimum working period before and after (to ensure minimum working period durations). In the second phase a monitor break is added repeatedly until no breaktime remains to be assigned or there are no longer any allowed break positions. The position of the break is chosen randomly out of the $freeSlots$ list. In case the monitor break - which is 2 slots long - fits into the position, the $freeSlots$ list is reduced as before. Otherwise, the slot is removed from the $freeSlots$ list, as no legal break assignment at this slot is possible. This procedure (algorithm 4.2: lines 14-22) is repeated until a break pattern satisfying all constraints is found. Afterwards the coverage $\mathcal{C}$ is updated accordingly and the newly created initial solution is returned.

4.3 Local Search

As part of the large neighborhood search algorithm a local search procedure is used, which is based on Beer et al. [BGM+10]. It comprises two local neighborhoods produced by two different moves: *assignment* and *swap*. The *assignment* assigns a break a new start within its related shift. This may result in appending the break to a preceding or succeeding break forming one longer break. Figure 4.2 shows two examples of the assignment of a break to a new position. The move *swap* exchanges a break with a predecessor or successor break of the same shift but with a different duration. An illustration of two break swaps is given in figure 4.3. Each neighborhood is a set of all solutions reached by applying the move *assignment* and *swap* to a single break, respectively. A minimum-conflicts-based heuristics is used to determine the break on which the move is applied on. The idea is to focus on parts of the current solution which are deviating from the required optimum. Thus the heuristic only considers timeslots which have a shortage or excess of employees. To ensure only an increase in quality of the solution, in general only solutions with a better fitness are accepted. Equal or worse solutions are accepted only with a varying probability, which decreases rapidly each time a worse solution is found and is reset when a better solution is reached. To avoid local optima a random-walk strategy is applied. Again a break is selected through the minimum-conflicts heuristic. The break is then moved to the next free random position found, which is reached by computing a random position within the shift and move one

Algorithm 4.2: Initialisation

Input: $\mathcal{C}$, $\mathcal{S}$
Output: $\mathcal{K}$

```
for each S in 𝒮 do
    freeSlots ← list of all possible slots where a break can be;
    if S_i has lunch break then
        set B_lunch to the middle of the shift;
        if lunch start does violate lunchTimeConstraints then
            set B_lunch at earliest possible point;
        else if lunch end does violate lunchTimeConstraints then
            set B_lunch at latest possible point;
        end
        store B_lunch in ℬ;
        reduce freeSlots;
    end
    repeat
        revert ℬ and freeSlots to only contain the B_lunch;
        while breaktime remains AND freeSlots is not empty do
            randomSlot ← select timeslot randomly out of freeSlots;
            if previous or following timeslot of randomSlot is in freeSlots then
                store B_monitor in ℬ;
                reduce freeSlots;
            else
                remove the randomSlot from freeSlots;
            end
        end
    until all TemporalConstraints are satisfies;
    for each B in ℬ do
        for each timeslot in B_i do
            update 𝒞 at timeslot;
        end
        if B_i has no contiguous successor then
            consider reacquaintance timeslot in 𝒞;
        end
    end
    𝒦_initial ← create initial solution;
    return 𝒦_initial;
end
```

timeslot forward until a legal position is found. At the shift end it proceeds from the shift start until all possible slots have been tried. In case no legal position can be found no rearrangement is made. Otherwise the break is moved to the new position. The solution is accepted regardless its quality, and thus the resulting fitness could be worse than the previous one. Such a move is necessary to lead the algorithm into new areas in the solution space and as a consequence escape local optima. The higher the probability of the random-walk procedure, the greater the diversification of the search. Hence, a small probability which is enough to escape local optima but not broaden the search too much is needed. The random-walk procedure is used with the small probability of 2.5%.

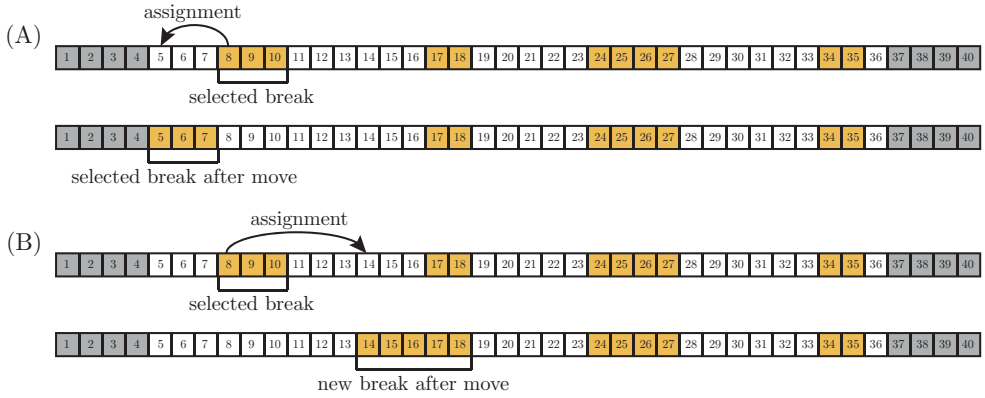

Figure 4.2: Two examples (A) and (B) for the move assignment. In both examples the selected break is assigned a new start position. In the case of example (B) the selected break is moved next to another break and thus forming one longer break.

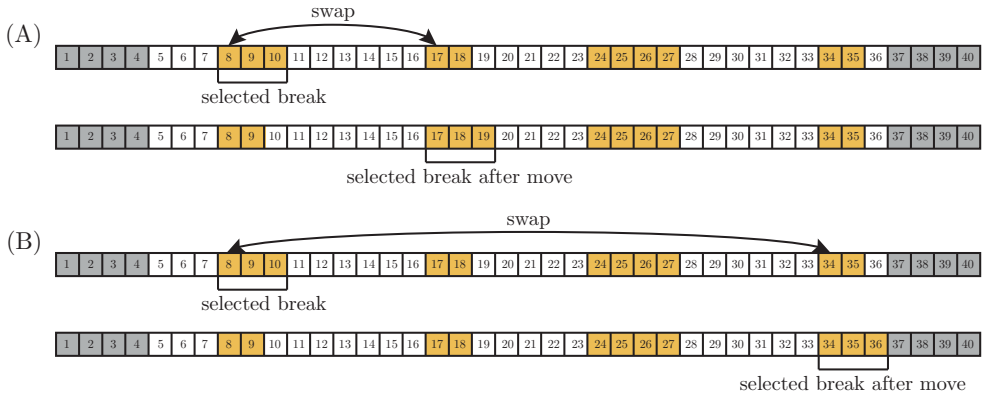

Figure 4.3: Two examples (A) and (B) for the move swap. The selected break can be swapped with any break of different length provided that all constraints are fulfilled.

The pseudo-code for the local search is given in algorithm 4.3, 4.4, 4.5, 4.6 and 4.7. In

algorithm 4.3 a break is selected out of a conflict list. For the selection a timeslot which has over- or undercover is determined. Then a shift containing the timeslot is picked at random and one break within is shift is picked randomly. After the selection of a break a move is chosen with a particular likelihood. The move *assignment* (algorithm 4.4) first removes the selected break from $\mathcal{B}$ and then creates the neighborhood of solutions reached by reinserting the break at any other possible timeslot. Each solution is evaluated by an acceptance procedure (algorithm 4.7) and the best solution is kept in memory. Finally, if the best solution was set, it is returned. In case there was none, changes are reverted and the input solution is returned. *Swap* (algorithm 4.5) works in a similar way except the neighborhood is composed of all solutions achieved by swapping the selected break with other breaks within the shift. *randomWalk* (algorithm 4.6) simply chooses a position to place the selected break by random. The break is moved one slot further until a valid solution is found. The solution is accepted and returned.

Algorithm 4.3: LocalSearch

Input: $\mathcal{K}_{current}$, $\mathcal{S}$
Output: $\mathcal{K}$

```
1  B_selected ← choose B randomly out of conflict list;
2  S_selected ← S associated with B_selected;
3  probability ← random number between 0 and 100;
4  if probability <= 2.5 then
5  |  K_new ← randomWalk(K_current, S_selected, B_selected);
6  else if probability <= 50 then
7  |  K_new ← moveAssignment(K_current, S, B_selected);
8  else
9  |  K_new ← moveSwap(K_current, S, B_selected);
10 end
11 return K_new;
```

4.4 Mixed Integer Programming

In the second part of the large neighborhood search a larger neighborhood is created and the solution is rebuilt using an exact method - mixed integer programming. In contrast to the local search procedure, which focuses on one break, this neighborhood is created by removing the break assignment of an entire shift. As shown in figure 4.4 the resulting neighborhood is much larger then those of the moves *assignment* and *swap* and it compasses both of them. To solve the subproblem of scheduling breaks for the designated shift, the MIP sub-algorithm hands the subproblem over to the solver CPLEX. CPLEX is instructed to favour optimality over feasibility meaning the primary aim is to find optimal solutions and less effort may be applied to finding feasible solutions early. Further a time limit ν for each CPLEX instance is set. When ν is exceeded CPLEX halts and returns the best solution found so far. The status of the returned solution needs to

Algorithm 4.4: moveAssignment

Input: $\mathcal{K}_{current}$, $S_{selected}$, $B_{selected}$
Output: $\mathcal{K}$

```
1  remove B_selected from B;
2  update C;
3  for all possible breakslots in S_selected do
4      if breakslot is free for break AND breakslot != B_selected then
5          move B_selected to new position;
6          if all TemporalConstraints are satisfied then
7              K_temp ← create temporary solution;
8              if acceptSolution(K_temp) then
9                  K_best ← K_temp;
10             end
11         end
12     end
13 end
14 if K_best exists then
15     update C;
16     return K_best;
17 else
18     revert changes;
19     return K_current;
20 end
```

be checked for being either feasible or even optimal. Below the definition for the CPLEX time limit is stated:

CPLEX Time Limit ν. A parameter which determines a time limit for each instance of CPLEX.

In algorithm 4.8 the pseudo-code for the MIP sub-algorithm is given. At first a shift is selected out of a conflicts list. It is a similar approach as applied for the local search sub-algorithm. A timeslot which has over- or undercover is determined. Then a shift containing the timeslot is picked at random. After the selection of a shift the model for CPLEX is declared, including all necessary variables, constants and expressions. The objective is stated and the model is handed over to the CPLEX solver. When finished (either because an optimal solution is found or the time limit is reached) the solution status is examined. The solution is accepted in case it is optimal or feasible and of better quality. Otherwise it is discarded and the input solution is returned.

Algorithm 4.5: moveSwap

Input: $\mathcal{K}_{current}$, $S_{selected}$, $B_{selected}$
Output: $\mathcal{K}$

```
1  for all B in S_selected do
2      if B_selected.length != B_i.length AND positions are free for break swap then
3          swap B_selected and B_i;
4          if all TemporalConstraints are satisfied then
5              K_temp ← create temporary solution;
6              if acceptSolution(K_temp) then
7                  K_best ← K_temp;
8              end
9          end
10     end
11 end
12 if K_best exists then
13     update C;
14     return K_best;
15 else
16     return K_current;
17 end
```

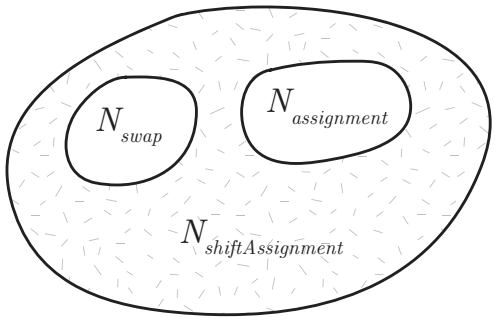

Figure 4.4: Neighborhoods of one iteration step derived by the different sub-algorithms. $N_{shiftAssignment}$ is reached by reassigning all breaks of a shift, $N_{assignment}$ by the move *assignment* and N_{swap} by the move *swap*. The neighborhood $N_{shiftAssignment}$ is much larger and it comprises both neighborhoods $N_{assignment}$ and N_{swap}.

Algorithm 4.6: randomWalk

Input: $\mathcal{K}_{current}$, $S_{selected}$, $B_{selected}$
Output: $\mathcal{K}$

```
1  remove B_selected from B;
2  update C;
3  randomPosition ← choose random timeslot out of possible breakslots in S_selected;
4  repeat
5  |  repeat
6  |  |  move randomPosition further by one timeslot;
7  |  until randomPosition is free for break;
8  until all TemporalConstraints are satisfied AND not all possible breakslots have
   been tried;
9  if all possible breakslots have been unsuccessfully tried then
10 |  revert changes;
11 |  return K_current;
12 end
13 set B_new to randomPosition;
14 store B_new in B;
15 K_new ← create new solution;
16 update C;
17 return K_new;
```

Algorithm 4.7: acceptSolution

Input: $\mathcal{K}_{temp}$
Output: boolean

```
1  recalculate K_temp.fitness;
2  if K_temp.fitness is better then K_best.fitness then
3  |  probabilityCount ← 0;
4  |  return true;
5  else if fitness is equal or worse then
6  |  increment probabilityCount by one;
7  |  if 100/probabilityCount >= random integer between 0 and 100 then
8  |  |  return true;
9  |  end
10 else
11 |  return false;
12 end
```

Algorithm 4.8: MixedIntegerProgramming

Input: $\mathcal{K}_{current}$, $\mathcal{S}$
Output: $\mathcal{K}$

```
1  S_selected ← choose S randomly out of conflict list;
2  cplexModel ← initialize cplex instance;
3  Define all variables, expressions and the objective for cplex;
4  Solve cplexModel with the cplex solver;
5  if cplexModel is solved then
6      if cplexModel.status equals optimal OR (cplexModel.status equals feasible AND
       fitness is better or equal) then
7          K_new ← create new solution;
8          merge result for S_selected in K_new;
9          update C;
10         return K_new;
11     end
12     else
           /* Cplex could not produce a better or optimal solution.        */
13         return K_current;
14     end
15 end
16 else
       /* Cplex could not solve the problem.                               */
17     return K_current;
18 end
```

CHAPTER 5

Computational Experiments

Several experiments have been conducted in order to further develop the final algorithm designed in this thesis as well as for evaluation of the final outcome. Throughout this work various parameter emerge, which influence the results of the algorithm. For instance, CPLEX Time Limit ν determines, if CPLEX has enough time to produce an optimal solution, a feasible one or maybe not one at all. Furthermore, different procedures for the choice of sub-algorithm were tested. The choice made at each iteration has a great influence on leading the search towards diversification or intensification.

In the following, we first state the experimental settings. Then a detailed description and discussion on the evaluation of parameters and the sub-algorithm selectors is given. Finally we present the final results and make a comparison to previous publications.

5.1 Experimental Settings

The algorithm is programmed in Java integrating the solver CPLEX. Tests were performed on a Dell Latitude E6420 with four Intel Core i7-2640M @2.80 GHz and 8 GB RAM using up to four cores. For each experiment ten runs were executed. For reasons of comparison with previous works the overall runtime is set to one hour (if not stated otherwise). As the MIP sub-algorithm is very time-consuming, higher time limits for the overall runtime have been tested as well.

For the evaluation 30 instances have been used, the same as presented by Beer et al. [BGM+10]. 10 are randomly generated and 20 were obtained from a real-life scenario. The former are described and publicly available in [ben08]. [ben08] further provides a validator to verify the feasibility of generated solutions. The real-life instances and the settings for all instances were obtained from a real-life use case for supervisory personnel. The settings, which are the same for all instances, are given below:

- T: $length = 2016$

- C_1: $p_1 = 7$, $p_2 = 7$
- C_2: $h = 73$, $g = 6$, $l_1 = 43$, $l_2 = 72$
- C_3: $w_1 = 6$, $w_2 = 20$
- C_4: $w = 11$, $b = 4$
- C_5: $b_1 = 2$, $b_2 = 12$

All values are given in timeslots and one timeslot lasts five minutes. Thus a planning period of 2016 timeslots corresponds to exactly one week. It is of cyclic nature, which means the schedule is repeating itself on a weekly basis. Real-life instances differ from randomly generated in terms of shift length. On average randomly generated instances show more shifts (146 compared to 128 shifts), but real-life instances have a higher average shift length. Shifts lengths for random instances range from 7 to 9 hours and score an average length of 7.9 hours. Real-life instances have shift lengths ranging from 5 to 12.6 hours and the average shift length is 9.8 hours, about 2 hours compared to random instances. Table 5.1 gives an overview on all instances, their number of shifts and the average shift length. In general it can be said, the greater the shift length, the larger the search space. Thus, for solving a subproblem with an exact method more time is spent on finding an optimal solution.

For the parameter testing two instances were selected, one out of each set of instances, random and real-life. If not stated otherwise, one single shift was considered for optimization in the MIP sub-algorithm.

5.2 Evaluation of Parameters and Sub-Algorithm Selectors

For the testing two instances were chosen: Random1-5 and 2fc04a. Experiments were conducted in the same way as the final results were computed (see section 5.1). Before parameters were tested the selector for the sub-algorithms was evaluated. In total three procedures for choosing a sub-algorithm were tested: random-selector (RS), timebound-selector (TS) and probabilty-selector (PS). The first selector RS is a simple procedure choosing the sub-algorithm randomly at each iteration. TS is a more elaborated procedure making use of two parameters: time since the last improvement by local search and time since the last improvement by MIP. Using TS the frequency of switches between sub-algorithms correlates with the time elapsed since the last improvement. For each sub-algorithm a separate time limit is defined. The time counter is reset each time an improvement is made. In case the time limit is exceeded there is a switch in sub-algorithm. The test results for both, RS and TS, can be found in Table 5.2. Neither RS nor TS performed satisfactory. Results for RS are almost as high as those using MIP only (initialisation + repeatedly MIP sub-algorithm). This is due to MIP being chosen too frequently. As it is very time-consuming the numbers of iterations are decreased

Random			**Real-Life**		
name	*no. shifts*	*avg. shift length*	*name*	*no. shifts*	*avg. shift length*
Random1-1	137	8.0 h	2fc04a	135	9.7 h
Random1-2	164	7.7 h	2fc04a03	134	9.8 h
Random1-5	141	7.7 h	2fc04a04	137	9.7 h
Random1-7	157	8.0 h	2fc04b	126	9.8 h
Random1-9	151	7.8 h	3fc04a	124	9.9 h
Random1-13	124	8.3 h	3fc04a03	123	10.0 h
Random1-24	137	7.8 h	3fc04a04	128	9.8 h
Random1-28	124	7.7 h	3si2ji2	151	9.0 h
Random2-1	179	8.0 h	4fc04a	124	9.9 h
Random2-4	162	7.9 h	4fc04a03	123	10.0 h
			4fc04a04	127	9.8 h
			4fc04b	125	9.7 h
			50fc04a	130	9.8 h
			50fc04a03	130	9.9 h
			50fc04a04	131	9.8 h
			50fc04b	126	9.9 h
			51fc04a	129	9.8 h
			51fc04a03	129	9.8 h
			51fc04a04	130	9.8 h
			51fc04b	126	9.8 h

Table 5.1: List of all instances used for evaluation: 10 randomly generated and 20 obtained from a real-life scenario. For each instance the number of shifts and the average shift length in hours is given. Real-life instances show a higher average shift length by about 2 hours.

significantly as well as the number of improvements. TS achieved far better results yet not as good as desired. Experience showed that local search achieves the most improvements in the beginning of the runtime and the number of improvements decreases over time, meaning that the time interval between improvements is getting longer. Setting the time limits to fixed numbers does not deal with these kind of circumstances. From a very early stage onwards the local search sub-algorithm is hardly able to achieve improvements as the time limit is reached beforehand in most of the cases. With regard to the characteristics of the sub-algorithms a third selector PS was developed. PS chooses a sub-algorithm using a probability function (4.1). The function is designed to use the MIP sub-algorithm very rarely in the beginning and increases the usage over time, yet give the local search sub-algorithm still the opportunity to find improvements. To prevent the MIP sub-algorithm from being used too frequently the probability is limited by a maximum value. Figure 5.1 illustrates the improvements made throughout the runtime by the different selectors for two example runs using TS or PS respectively. Using TS (Figure 5.1, left) improvements made through local search stop abruptly already in a

very early stage (coloured in blue). This is due to a small time limit of 1 minute for local search. The time limit was chosen so small in order to enable switches from local search to MIP (coloured in red) already in the early stage. Experiments with slightly longer time limits (5 minutes; see Table 5.2) have also shown no significant increases in solution quality. As the parameter is set to a fixed value it does not correspond to the increasing time interval between improvements. Local search usually makes rapid improvements in the very beginning of the runtime, which are essential for the further progress of the algorithm. Thus stopping it after a certain time limit does affect the quality of the solution. In contrast using PS (Figure 5.1, right) local search continues to improve the solution for further several minutes resulting in a much lower fitness from the start. Additionally, local search still makes improvements within the first 40 minutes and MIP is able to make improvements to the end of the runtime. PS was able to reach a fitness of 721 in contrast to 850 reached with TS. PS was identified as the superior selector and parameter tests and final results were computed using PS.

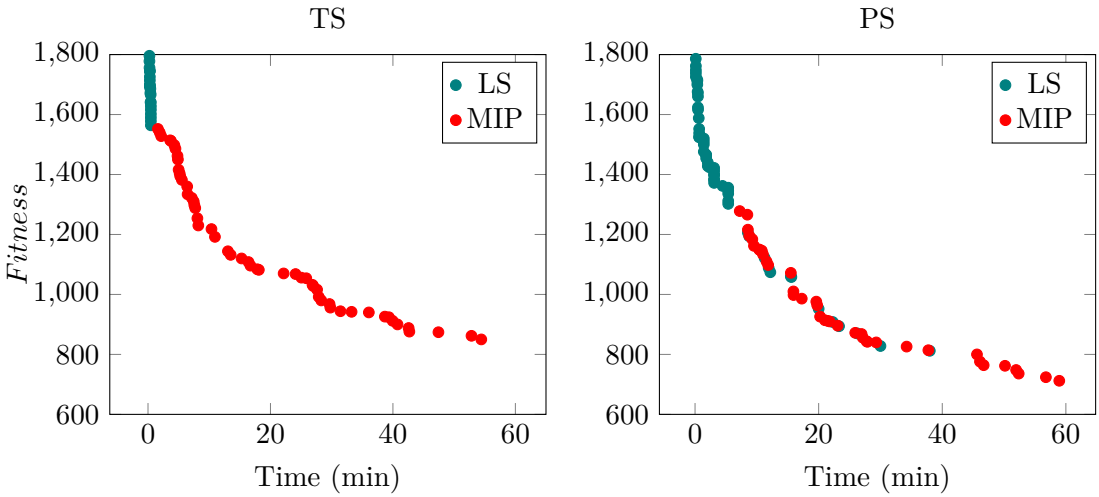

Figure 5.1: Improvements made during runtime using TS (left) and PS (right) on the instance Random1-5. All improvement steps from a fitness of about 13,500 (initial fitness) to 1,800 are made within the first 30 seconds by local search. Thus, the data range on the y-axis was set to a maximum of 1,800, to give a better focus on the fitness at a later time. In these two examples TS reached a final fitness of 850 and PS of 712. It can be seen that PS is able to achieve better results already from an early stage on onwards. Settings TS: time limit local search = 1 min, time limit MIP = 2 min, $\nu = 5$; settings PS: $\lambda = 5$, $\eta = 20$, $\nu = 5$

For the evaluation of parameters different settings of each parameter were tested one by one. A total number of 56 experiments were performed taking a total runtime of 2,736 hours. The following parameter were considered in the evaluation:

- LNS Time Limit μ: A parameter which determines a time limit for the overall

$\mu = 1$ hours		**Random1-5**		**2fc04a**	
selector	ν	*best*	*mean*	*best*	*mean*
RS	*2 min*	3,722	4,986	10,674	11,498
TS1	*2 min*	752	841	3,110	3,169
TS2	*5 min*	774	892	3,126	3,179
TS3	*2 min*	716	825	2,976	3,099

Table 5.2: Results for the testing of the selectors RS and TS. In TS1 and TS2 the time limit until a sub-algorithm is switched was 1 minute for local search and 3 minutes for MIP, in TS3 both were set to 5 minutes.

runtime of LNS in hours.

- Probability Curve Gradient λ: A parameter which determines the gradient of the probability curve.
- Probability Factor η: A parameter which regulates the probability for a sub-algorithm to be selected given in percent.
- CPLEX Time Limit ν: A parameter which determines a time limit in minutes for each instance of CPLEX.

λ is responsible for the increase in probability for the MIP sub-algorithm to be chosen. The smaller λ the more linear is the function. The higher λ the flatter is the function in the first 80% of the runtime followed by a steep rise. The idea is to set λ to a value so it counters the stagnation of improvements made by local search. Values of 5 and 15 have been tested for λ.

η gives the maximum possible probability that can be reached. Values range from 0 and 100. If it is set to 0 only local search would be used. 100 means that at the end of the runtime the MIP sub-algorithm has a 100% probability to be chosen. η is meant to balance the usage of both sub-algorithms. When choosing a value one should consider the time needed for an iteration. The MIP sub-algorithm needs several minutes for one iteration whereas local search is able to run up to 12 million iterations per minute. Yet, local search needs more iterations in order to achieve an improvement. Thus, η should be big enough to allow MIP to be used already in the first half of runtime, yet be small enough to give local search sufficient time for improvements. Values of 10, 20 and 30 have been tested.

ν is the time limit set for CPLEX. As a full shift is considered for optimization this step is more time consuming, so ν should be set to several minutes in order to allow CPLEX to return a solution. But the overall runtime is in general limited to one hour. At one iterations step of MIP only one out of about 140 shifts is being optimized. Thus, the step should not take more then approximately 5 minutes. Experiments with 2, 5 and 10 minutes were conducted.

For the instance Random1-5 PS achieved the best results with the settings $\lambda = 5$, $\eta = 20$ and $\nu = 5$ in both best and mean (see Table 5.3). As can be seen from Table 5.4 results for PS on the instance 2fc04a do not differ greatly from results with only local search (LS; initialisation + repeatedly local search sub-algorithm). This is because in many cases CPLEX can not produce a better solution or any at all within the time limit. Most of the shifts for real-life instances are longer (beyond 9 hours) making the subproblem too complex to analyse for CPLEX. One reason could be that time limits for CPLEX were too tight. Therefore higher values for ν and μ were tested as well. Results for a runtime of two and three hours are given in 5.5 and 5.6. Still only little improvements were made on the 2fc04a instance. A larger time limit for CPLEX (e.g. 10 minutes) does not lead to a better performance. Figure 5.2 compares the mean results of different time limits and parameter settings to each other. In general a tendency towards the longer the overall runtime the greater the positive effect of MIP on the performance can be observed. Whereas results for LS stay almost the same with different μ, LNS is still able to improve greatly with higher μ. For the computation of final results the following parameter settings were chosen:

- LNS1: $selector = PS$, $\mu = 1$, $\lambda = 5$, $\eta = 5$, $\nu = 5$
- LNS2: $selector = PS$, $\mu = 1$, $\lambda = 5$, $\eta = 5$, $\nu = 2$

Random1-5 - μ = 1 hour

λ/η	*5/10*		*5/20*		*5/30*		*15/20*		*LS*		*MIP*	
ν	*best*	*mean*	*best*	*mean*	*best*	*mean*	*best*	*mean*	*best*	*mean*	*best*	*mean*
2 min	712	774	648	795	698	**770**	656	759			4,540	5,346
5 min	670	778	**612**	**770**	680	783	652	812				
10 min	686	781	734	806	784	875	684	849				
-									1,066	1,171		

Table 5.3: Results for the parameter testing on the instance Random1-5 with an overall runtime of 1 hour. Results were computed using PS with different parameter settings and LS only as well as MIP only for comparison. Best results are in bold.

2fc04a - μ = 1 hour

λ/η	*5/10*		*5/20*		*5/30*		*15/20*		*LS*		*MIP*	
ν	*best*	*mean*	*best*	*mean*	*best*	*mean*	*best*	*mean*	*best*	*mean*	*best*	*mean*
2 min	3,016	3,082	**2,948**	**3,069**	3,082	3,117	3,010	3,088			10,910	11,574
5 min	3,036	3,099	2,996	3,122	3,092	3,165	3,050	3,117				
10 min	3,058	3,106	3,088	3,147	3,030	3,127	3,030	3,124				
-									3,046	3,095		

Table 5.4: Results for the parameter testing on the instance 2fc04a with an overall runtime of 1 hour. Results were computed using PS with different parameter settings and LS only as well as MIP only for comparison. Best results - in bold - do not differ greatly from local serach only as CPLEX was not able to return a solution in many cases.

	Random1-5 - μ = 2 hours						**2fc04a - μ = 2 hours**					
λ/η	*5/20*		*LS*		*5/20 - 2 shifts*		*5/20*		*LS*		*5/20 - 2 shifts*	
ν	*best*	*mean*	*best*	*mean*	*best*	*mean*	*best*	*mean*	*best*	*mean*	*best*	*mean*
2 min	506	617					2,982	3,045				
5 min	512	673					3,000	3,067				
10 min	618	680			1,058	1,207	3,010	3,075			3,030	3,117
-			952	1,094					3,060	3,084		

Table 5.5: Results for the parameter testing on the instances Random1-5 and 2fc04a with an overall runtime of 2 hours. Results were computed using PS with different parameter settings and LS for comparison. Increasing ν does not lead to an improved performance. Tests for 2fc04a show only little improvement compared to results using LS. Additionally, test were made with 2 shifts simultaneous being handed over to CPLEX, showing a poor outcome.

	Random1-5 - μ = 3 hours				**2fc04a - μ = 3 hours**			
λ/η	*5/20*		*LS*		*5/20*		*LS*	
ν	*best*	*mean*	*best*	*mean*	*best*	*mean*	*best*	*mean*
2 min	476	564			2,936	3,003		
5 min	472	596			2,932	3,025		
10 min	496	599			2,984	3,062		
-			1,010	1,115			2,992	3,049

Table 5.6: Results for the parameter testing on the instances Random1-5 and 2fc04a with an overall runtime of 3 hours. Results were computed using PS with different parameter settings and LS for comparison. Increasing ν leads to worse performance. Tests for 2fc04a show only little improvement compared to using LS.

	Random1-5 μ **= 1 week**	**2fc04a** μ **= 1 week**
λ/η / ν	*5/20 - all shifts* *single run*	*5/20 - all shifts* *single run*
1 week	1,012	2,996

Table 5.7: Results for the parameter testing on the instances Random1-5 and 2fc04a with an overall runtime of 1 week. All shifts were given as input to CPLEX. Results are the same as for using LS, due to CPLEX not being able to process all shifts at once.

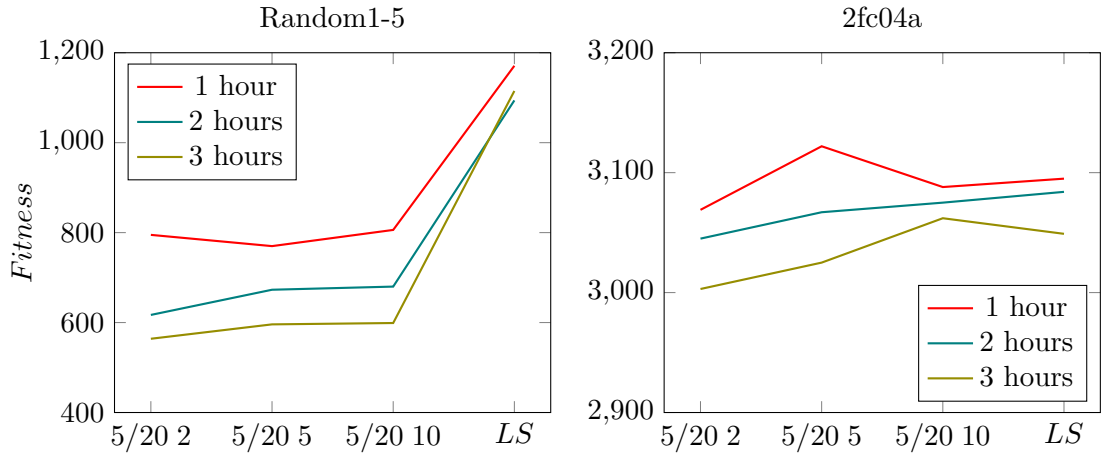

Figure 5.2: Comparison of the mean values of the test results using PS with different parameter settings and LS on the instance Random1-5 (left) and 2fc04a (right). A longer runtime leads to a slighly better fitness compared relatively to using LS. Yet a higher CPLEX time limit does not improve the solution quality.

In [Koc15] Kocabas presented a mixed integer model, which optimizes all shifts at once, but for a special case of Bsp. We conducted a similar experiment for the general Bsp giving all shifts at once to CPLEX. The overall runtime as well as CPLEX time limit was set to one week. CPLEX was not able to cope with the complexity and did not return a solution. Results (Table 5.7) are the same as results using LS for several hours. One reason could be that the number of expressions is too high. Due to the implementation of this model the constraints are expressed for each shift separately. The fitness is then computed for all shifts together. Processing each shifts separately leads to a great number of expressions. For example the instance Random1-5 has 141 shifts with an average length of 90 timeslots. This results in 6 breaks per shift and about 900 expressions per shift. In total CPLEX has to handle about 127,000 expression for the instance Random1-5. Future work could be to rewrite the model so it is better suited for the optimization of a large number of shifts.

5.3 Results and Comparison

Final results for all instances were computed with the following parameter settings:

- LNS1: $selector = PS$, $\mu = 1$, $\lambda = 5$, $\eta = 5$, $\nu = 5$
- LNS2: $selector = PS$, $\mu = 1$, $\lambda = 5$, $\eta = 5$, $\nu = 2$

For comparison also results of using only local search (LS) were computed. LS is our implementation of the local search algorithm based on Beer et al. [BGM+10] made up of the initialisation and the repeated use of the local search sub-algorithm. The results can be seen in Table 5.8, giving the best, mean and standard deviation σ of each instance. For most of the instances LNS2 achieved better results then LNS1. Especially for the real-life instances it performed better in best and mean for almost all the instances. For the random instances both LNS versions clearly outperform LS. Although the difference in results of LS, LNS1 and LNS2 is not significant for the real-life instances, still LNS1 and LNS2 outperform LS in most instance.

Results from previous publications are shown in Table 5.9. Beer et al. [BGM+10] introduced the first benchmark results for the break scheduling problem. The results shown in Table 5.9 were computed by using a local search algorithm with a min-conflicts-random-walk procedure and a randomly created initial solution (Random init) or an initial solution created by solving the corresponding simple temporal problem (STP init). MAPS was developed by Widl [Wid10] and is holding the current upper bounds for 18 out of 20 real-life instances. MAPS is a memetic algorithm using genetic operators combined with local search and a penalty system. Kocabas [Koc15] gave an integer linear programming formulation (ILP) to solve the Bsp as a whole. The overall time limit was set to two hours, but in only two cases the time limit was reached. ILP achieved new upper bounds for all except one of the randomly generated instances. However, this approach could not be applied to the real-life instances. The large neighborhood search algorithm presented in this thesis was able to outperform Beer et al. [BGM+10] in all of the random instances, but not the real-life instances. Comparing the results of Beer et al. and the implementation of local search of this thesis, one can see that the latter performed slightly worse. Yet, due to being programmed in different programming languages a direct comparison might be inaccurate. In contrast to Java, Comet [MVH05] (used by Beer et al. and Widl) was specifically designed for constrained-bases local search algorithms.

In contrast to our expectations, LNS was not able to surpass the best known results [BGM+10] [Wid10] [Koc15]. One reason could be the complexity of the mixed integer model as it comprises a hight number of expressions. This led to the problem that CPLEX could not improve on the solution in a significant number of iterations. Still compared to Kocabas [Koc15] we were able to apply our algorithm to the real-life instances.

$\mu = 1$ hours	**LNS1**			**LNS2**			**LS**		
Instance	*best*	*mean*	σ	*best*	*mean*	σ	*best*	*mean*	σ
Random1-1	794	964	88	**726**	**889**	71	1,240	1,331	57
Random1-2	818	930	95	**700**	**884**	110	1,162	1,263	105
Random1-5	**612**	**780**	87	714	797	73	1,066	1,171	73
Random1-7	1,034	1,149	92	**930**	**1,075**	102	1,336	1,379	26
Random1-9	978	**1,082**	68	**940**	1,085	78	1,214	1,414	97
Random1-13	**740**	929	81	784	**885**	73	998	1,111	80
Random1-24	**724**	**811**	97	730	861	109	940	1,059	75
Random1-28	646	779	163	**606**	**745**	138	982	1,091	76
Random2-1	**1,212**	1,376	124	1,218	**1,363**	73	1,458	1,572	76
Random2-4	922	1,162	159	**844**	**1,088**	119	1,332	1,475	83
2fc04a	**2,996**	3,126	64	3,004	3,096	54	3,050	**3,091**	24
2fc04a03	**2,984**	3,103	47	3,022	3,077	31	3,018	**3,065**	30
2fc04a04	3,160	3,212	30	3,116	3,178	36	**3,094**	**3,152**	23
2fc04b	2,228	2,302	35	2,216	2,264	36	**2,184**	**2,252**	36
3fc04a	1,850	1,929	43	**1,814**	**1,869**	35	1,846	1,884	27
3fc04a03	**1,790**	1,888	46	**1,790**	**1,873**	47	1,834	1,893	26
3fc04a04	2,062	2,118	33	2,084	2,112	23	**2,044**	**2,087**	23
3si2ji2	3,518	3,537	13	**3,484**	**3,525**	24	3,512	3,538	15
4fc04a	**1,774**	1,923	55	1,786	1,888	50	1,846	**1,887**	25
4fc04a03	1,902	1,953	32	**1,856**	**1,915**	41	1,876	1,919	18
4fc04a04	2,042	2,145	49	2,032	2,100	60	**2,030**	**2,095**	28
4fc04b	1,668	1,758	42	**1,644**	**1,739**	39	1,728	1,759	15
50fc04a	1,920	2,019	50	**1,848**	**1,970**	79	1,942	1,998	36
50fc04a03	1,986	2,056	39	**1,928**	**2,001**	36	1,926	2,014	41
50fc04a04	2,054	2,133	47	**2,036**	2,125	40	2,082	**2,123**	27
50fc04b	2,122	2,213	53	**2,106**	**2,198**	58	2,144	2,202	37
51fc04a	2,270	2,334	44	2,226	2,338	53	**2,218**	**2,314**	46
51fc04a03	2,204	2,283	49	**2,140**	**2,222**	43	2,156	2,251	45
51fc04a04	2,340	2,433	45	**2,292**	**2,371**	52	2,296	2,377	43
51fc04b	2,668	2,728	34	**2,576**	**2,621**	27	2,600	2,653	39

Table 5.8: Final results computed with the parameter settings LNS1 and LNS2. σ is the standard deviation. Results of using only local search (LS) are added for comparison. Settings for LNS1 and LNS2 only differ in the CPLEX time limit, which is set to 5 minutes or 2 minutes respectively. Other parameters are set as follows: $\mu = 1$, $\lambda = 5$, $\eta = 5$. LNS2 achieves better results as LNS1 in most of the instances. Both outperform LS in all of the instances.

	Random init [BGM+10]			STP init [BGM+10]			MAPS [WM14]			ILP [Koc15]
Instance	*best*	*mean*	σ	*best*	*mean*	σ	*best*	*mean*	σ	
Random1-1	1,728	1,972	177	-	-	-	346	440	48	**84**
Random1-2	1,654	1,994	172	-	-	-	370	476	65	**228**
Random1-5	1,284	1,477	99	-	-	-	378	418	29	**360**
Random1-7	1,860	1,077	154	-	-	-	496	583	42	**408**
Random1-9	1,358	1,658	213	-	-	-	318	423	51	**108**
Random1-13	1,264	1,535	245	-	-	-	370	445	55	**348**
Random1-24	1,586	1,712	74	-	-	-	542	611	43	**408**
Random1-28	1,710	2,020	233	-	-	-	**222**	318	71	228
Random2-1	1,686	1,855	142	-	-	-	724	889	75	**636**
Random2-4	1,712	2,053	242	-	-	-	476	535	45	**114**
2fc04a	3,094	3,248	84	3,112	3,224	86	**2,816**	**2,961**	71	-
2fc04a03	3,100	3,229	61	3,138	3,200	39	**2,834**	**2,934**	54	-
2fc04a04	3,232	3,371	68	3,234	3,342	60	**2,884**	**2,954**	60	-
2fc04b	2,017	2,104	92	**1,822**	2,043	99	1,884	**1,948**	49	-
3fc04a	1,746	1,809	49	1,644	1,767	102	**1,430**	**1,533**	67	-
3fc04a03	1,632	1,804	87	1,670	1,759	53	**1,440**	**1,514**	40	-
3fc04a04	1,942	2,2032	51	1,932	1,980	40	**1,614**	**1,718**	48	-
3si2ji2	3,626	3,692	35	3,646	3,667	14	**3,177**	**3,206**	17	-
4fc04a	1,694	1,851	126	1,730	1,817	48	**1,478**	**1,540**	29	-
4fc04a03	1,666	1,795	87	1,748	1,834	55	**1,430**	**1,502**	42	-
4fc04a04	1,918	2,017	95	1,982	2,064	62	**1,606**	**1,674**	48	-
4fc04b	1,440	1,527	56	1,410	1,489	49	**1,162**	**1,233**	48	-
50fc04a	1,750	1,861	95	1,672	1,827	81	**1,548**	**1,603**	36	-
50fc04a03	1,718	1,847	96	1,686	1,813	84	**1,402**	**1,514**	67	-
50fc04a04	1,790	1,985	83	1,792	1,917	64	**1,480**	**1,623**	89	-
50fc04b	1,854	2,012	91	1,822	1,954	77	**1,818**	**1,900**	56	-
51fc04a	2,048	2,204	89	2,054	2,166	62	**1,886**	**2,074**	87	-
51fc04a03	2,004	2,096	60	1,950	2,050	86	**1,886**	**1,949**	46	-
51fc04a04	2,058	2,195	64	2,116	2,191	53	**1,958**	**2,039**	52	-
51fc04b	2,380	2,514	106	**2,244**	2,389	94	2,306	**2,367**	43	-

Table 5.9: Results from previous publications. σ is the standard deviation. STP init and Random init are a local search algorithm introduced by Beer et al. [BGM+10], MAPS is a memetic algorithm developed by [Wid10] and Kocabas [Koc15] presented an integer linear programming formulation. MAPS and ILP hold the upper bounds for all instances, except in the two 2fc04b and 51fc04b.

CHAPTER 6

Conclusions

In this thesis a large neighborhood search was proposed to solve the break scheduling problem. It comprises a local search algorithm and a mixed integer programming procedure (MIP). For the MIP procedure we gave for the first time a formulation for the BSP without restrictions on the number of breaks per shift. The implementation of the LNS algorithm is made up of two sub-algorithms: local search and MIP. The local search sub-algorithm, which is based on [BGM+10], uses two moves - *swap* and *assignment* - and a random-walk procedure to escape local optima. The sub-algorithm focuses on the optimization of only a single break. The MIP sub-algorithm removes the break assignment of an entire shift (destroy method) and reconstructs the solution optimally (repair method) - $shiftAssignment$. To solve the MIP formulation the solver IBM ILOG CPLEX Optimization Studio was used.

A set of experiments was conducted in order to test different sub-algorithm selectors and parameter settings. In total 56 experiments were performed taking a total runtime of 2,736 hours. Three different selectors were designed and tested: random-selector (RS), timebound-selector (TS) and probability-selector (PS). RS chooses the sub-algorithm randomly at each iteration. TS makes the choice based on the time elapsed since the last improvement. PS uses a function regulating the probability of a sub-algorithm to be chosen. Further, four different parameters were tested: LNS time limit μ, probability cure gradient λ, probability factor η and CPLEX time limit ν.

The main conclusions of the evaluation of parameters and sub-algorithm selectors are as following:

- An elaborated procedure to select a sub-algorithm is superior to choosing sub-programs randomly.
- Improvements made by local search in the very beginning of the runtime are essential for the further progress of the algorithm.

- PS was found to be the best selector and was used for the computation of the final results.
- The algorithm performs best when the usage of the MIP sub-algorithm is limited. The maximum probability of it to be choosen was set to 20% and the CPLEX time limit was set to 2 to 5 minutes.
- There is a tendency towards the longer the overall runtime of the LNS algorithm the greater the positive effect of MIP on the performance.

Final results showed that the LNS algorithm outperforms our implementation of local search in most cases. However, it did not yet reach the upper bounds of the best known results so far [BGM+10] [Wid10] [Koc15]. Still compared to Kocabas [Koc15] we were able to apply our algorithm to the real-life instances.

Future work could be to redesign the mixed integer programming model in a more efficient way. It would be interesting to consider other formulations for the Bsp without restrictions to be able to solve the whole Bsp. A way to further optimize performance could be the implementation of other destroy-and-repair methods for the MIP sub-algorithm, e.g. single breaks of different shifts within the same time interval. Additionally, it would be interesting to investigate the use of a tabu search.

List of Figures

List of Tables

List of Algorithms

Bibliography

[Ayk96] Turgut Aykin. Optimal shift scheduling with multiple break windows. *Management Science*, 42(4):591–602, 1996.

[Ayk00] Turgut Aykin. A comparative evaluation of modeling approaches to the labor shift scheduling problem. *European Journal of Operational Research*, 125(2):381–397, 2000.

[ben08] Shift Design and Break Scheduling Benchmarks. Benchmarks, 2008. `http://www.dbai.tuwien.ac.at/proj/SoftNet/Supervision/Benchmarks/`.

[BGM+08] Andreas Beer, Johannes Gärtner, Nysret Musliu, Werner Schafhauser, and Wolfgang Slany. Scheduling breaks in shift plans for call centers. In *The 7th International Conference on the Practice and Theory of Automated Timetabling, Montréal, Canada*, 2008.

[BGM+10] Andreas Beer, Johannes Gärtner, Nysret Musliu, Werner Schafhauser, and Wolfgang Slany. An ai-based break-scheduling system for supervisory personnel. *IEEE Intelligent Systems*, 25(2):60–73, 2010.

[BJ90] Stephen E Bechtold and Larry W Jacobs. Implicit modeling of flexible break assignments in optimal shift scheduling. *Management Science*, 36(11):1339–1351, 1990.

[CGQR11] Marie-Claude Côté, Bernard Gendron, Claude-Guy Quimper, and Louis-Martin Rousseau. Formal languages for integer programming modeling of shift scheduling problems. *Constraints*, 16(1):54–76, 2011.

[CGR11] Marie-Claude Côté, Bernard Gendron, and Louis-Martin Rousseau. Grammar-based integer programming models for multiactivity shift scheduling. *Management Science*, 57(1):151–163, 2011.

[Dan54] G. B. Dantzig. A comment on eddie's traffic delays at toll booths. *Operations R*, 2:339–341, 1954.

[DGGM+13] Luca Di Gaspero, Johannes Gärtner, Nysret Musliu, Andrea Schaerf, Werner Schafhauser, and Wolfgang Slany. Automated shift design and break scheduling. In *Automated Scheduling and Planning*, pages 109–127. Springer, 2013.

[GGM+10] Luca Di Gaspero, Johannes Gärtner, Nysret Musliu, Andrea Schaerf, Werner Schafhauser, and Wolfgang Slany. A hybrid LS-CP solver for the shifts and breaks design problem. In *Hybrid Metaheuristics - 7th International Workshop, HM 2010, Vienna, Austria, October 1-2, 2010. Proceedings*, pages 46–61, 2010.

[GMS05] Johannes Gärtner, Nysret Musliu, and Wolfgang Slany. A heuristic based system for generation of shifts with breaks. In *Applications and Innovations in Intelligent Systems XII*, pages 95–106. Springer, 2005.

[IBM16] IBM. IBM ILOG CPLEX Optimization Studio. Software, 2016. `http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer`.

[Koc15] Deniz Kocabas. Exact methods for shift design and break scheduling. Master's thesis, Vienna University of Technology, 2015.

[MSW09] Nysret Musliu, Werner Schafhauser, and Magdalena Widl. A memetic algorithm for a break scheduling problem. In *8th Metaheuristic International Conference, Hamburg, Germany*, 2009.

[MVH05] Laurent Michel and Pascal Van Hentenryck. The comet programming language and system. In *International Conference on Principles and Practice of Constraint Programming*, pages 881–881. Springer, 2005.

[PR10] David Pisinger and Stefan Ropke. Large neighborhood search. In *Handbook of metaheuristics*, pages 399–419. Springer, 2010.

[PS82] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Dover Books on Computer Science. Dover Publications, 1982.

[QR10] Claude-Guy Quimper and Louis-Martin Rousseau. A large neighbourhood search approach to the multi-activity shift scheduling problem. *J. Heuristics*, 16(3):373–392, 2010.

[RCS10] Monia Rekik, Jean-François Cordeau, and François Soumis. Implicit shift scheduling with multiple breaks and work stretch duration restrictions. *J. Scheduling*, 13(1):49–75, 2010.

[Sch10] Werner Schafhauser. *TEMPLE - A Domain Specific Language for Modeling and Solving Real-Life Staff Scheduling Problems*. PhD thesis, Vienna University of Technology, 2010.

[Sha98] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *Principles and Practice of Constraint Programming—CP98*, pages 417–431. Springer, 1998.

[Tho95] Gary M Thompson. Improved implicit optimal modeling of the labor shift scheduling problem. *Management Science*, 41(4):595–607, 1995.

[Tho96] Gary M Thompson. Optimal scheduling of shifts and breaks using employees having limited time-availability. *International Journal of Service Industry Management*, 7(1):56–73, 1996.

[TP07] Gary M Thompson and Madeleine E Pullman. Scheduling workforce relief breaks in advance versus in real-time. *European Journal of Operational Research*, 181(1):139–155, 2007.

[TW06] Pascal Tellier and George White. Generating personnel schedules in an industrial setting using a tabu search algorithm. In *PATAT*, volume 2006, pages 293–302, 2006.

[Wid10] Magdalena Widl. Memetic algorithms for break scheduling. Master's thesis, Vienna University of Technology, 2010.

[WM14] Magdalena Widl and Nysret Musliu. The break scheduling problem: complexity results and practical algorithms. *Memetic Computing*, 6(2):97–112, 2014.