TU WIEN Informatics

Casual Employee Scheduling with Constraint Programming and Metaheuristics

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Ing. Stephan Teuschl, BSc
Registration Number 0680934

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Günther Raidl
Assistance: Univ.-Ass. Dipl.-Ing. Nikolaus Frohner

Vienna, 11th October, 2020

_____   _____
        Stephan Teuschl   Günther Raidl

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

Erklärung zur Verfassung der Arbeit

Ing. Stephan Teuschl, BSc

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, 11. Oktober 2020

_____

Stephan Teuschl

Danksagung

Ich möchte mich an dieser Stelle als Erstes bei meinen Betreuern für ihre große Geduld, das detaillierte Feedback und generelle Hilfestellung bedanken.

Weiters möchte ich auch meinen Eltern und Freunden für deren Verständnis und Rücksichtnahme sowie Geduld und Hilfsbereitschaft danken.

Kurzfassung

Das Feld der Personaleinsatzplanung im Allgemeinen und des Nurse Rostering Problems (NRP) im Speziellen wird seit geraumer Zeit wissenschaftlich untersucht. Oft liegt bei diesen Studien der Fokus auf einzelnen Bereichen wie Krankenhäusern und stationären Einrichtungen. In dieser Diplomarbeit wird ein Problem beschrieben, welches dem NRP ähnelt, den Fokus aber auf geringfügig beschäftige Arbeitskräfte legt. Diese gelegentlichen Mitarbeiter unterscheiden sich von traditionellen Vollzeitmitarbeitern in vielen Bereichen, am Gravierendsten allerdings in deren Zeitvorgaben. Von ihnen wird erwartet, dass sie im Laufe eines Monats und im Rahmen ihrer zeitlichen Möglichkeiten eine gewisse Anzahl von Stunden bzw. Tagen einsatzbereit sind, und das potenziell an unterschiedlichen Arbeitsplätzen und örtlichen Gegebenheiten. Aus den von ihnen bereitgestellten Angaben ihrer Verfügbarkeit muss danach eine Arbeitseinteilung erstellt werden.

Diese Diplomarbeit beschreibt einen Algorithmus, der in der Lage ist, solch eine Einteilung mittels der Angebote der Mitarbeiter und den Anforderungen der Arbeitsplätze zu erstellen. Neben der Optimierung der Anforderungen für die diversen Arbeitsplätze, wird auch die Fairness der Einteilung sowie die Präferenzen der Mitarbeiter an welchen Arbeitsplätzen sie arbeiten wollen, berücksichtigt.

Das Problem wird formell definiert und von zwei unterschiedlichen methodischen Perspektiven beleuchtet: ein Constraint Programming (CP) und ein metaheuristischer Ansatz werden implementiert und hybridisiert. Die gewählte Metaheuristic ist eine Variable Neighbourhood Search, welche als General VNS (GVNS) mit einem integrierten Variable Neighbourhood Descent (VND) implementiert wird. Die zu verbessernde initiale Lösung wird durch CP oder durch eine zusätzlich implementierte greedy Heuristik geliefert.

Von zehn verschiedenen realen Instanzen werden verschiedene Konfigurationen des GVNS und VND mit unterschiedlich generierten initialen Lösungen getestet und verglichen.

Durch die Komplexität des Problems ist CP nicht in der Lage, kompetitive Resultate zu erzielen. Resultate, die mittels einer initialen Lösung und einem darauf folgenden VNS erzielt werden, sind generell erfolgsversprechender. Für einen realen Einsatz liefern Lösungen, die durch eine greedy Heuristik generiert und mittels GVNS verbessert werden, die besten Ergebnisse. Unser Lösungsansatz ist ein mächtiges Werkzeug für menschliche Planer, um innerhalb weniger Stunden hochqualitative Lösungen zu generieren, die mit wenig Adaptionsaufwand in der Praxis benutzt werden können.

Abstract

The field of personnel scheduling in general and of the nurse rostering problem in particular has been studied for a long time, most often focused on unique real-world applications. In this thesis, a problem similar to the nurse rostering problem but focused on casual employees is studied. Casual employees as described in the thesis differ from full-time employees in many aspects, most important of which are time constraints. Casual employees are expected to work a few days a month on days they choose, at possibly different locations, and a roster has to be created from their offerings.

This thesis provides an algorithm that is able to generate such a roster when given the offerings from casual employees and the requirements of their workplaces. In addition to maximizing the amount of employees than can be staffed at any open shift, the fairness of the assignments as well as the preference of workplace of the casual employees is considered. The problem is formalised and approached from two different methodic perspectives. A constraint programming as well as a metaheuristic approach are developed and later hybridised. The chosen metaheuristic is a variable neighbourhood search (VNS), implemented as a general VNS (GVNS) with an embedded variable neighbourhood descent (VND). The initial solution is given by constraint programming (CP) or an additionally implemented greedy construction heuristic.

Different configurations of either GVNS or VND with different types of initial solutions are tested on ten real-world instances and subsequently compared.

Due to the complexity of the problem, pure constraint programming is not able to deliver competitive solutions, whereas approaches combining an initial solution obtained from either CP or a greedy heuristic with the VNS are more promising. Our experimental evaluation indicates that in practice, initial solutions from the custom greedy heuristic that are subsequently improved by the GVNS are most promising. While our approach was not designed to automatically handle all real-world peculiarities arising, it serves as a powerful tool to generate high quality solutions within a few hours to be further adapted by the human planner.

Contents

CHAPTER 1

Introduction

The setting of this thesis is a prominent art institution in Austria, consisting of multiple museums. Each of those museums has their own opening hours, event schedule and requirements for customer service employees. While the baseline of this need is provided by conventionally employed full-time employees, there is much more potential work than there are full-time employees. For this reason, casual employees are employed. These are generally, but not exclusively, university students wanting to earn a bit extra while pursuing their education. Their contract is much more fluid—they can choose when they want to work during a month and only work between one and six shifts a month. The total number of shifts covered by casual employees every month generally hovers around a thousand. The assignments of employees to shifts are currently done manually. As can be expected, this is difficult, exhaustive work that has to be repeated every month during a very short time frame. Due to the nature of the problem, the name used in this work is Casual Employee Scheduling Problem (CESP). The motivation for this project is to provide an alternative for the manual scheduling. Besides offering an immediate real-life application for the result of the research project, the problem as such is unique enough in its definition that it warrants a thorough investigation.

In the course of the project, the real life problem is formally defined and two solution approaches, on the basis of constraint programming and a variable neighbourhood search are discussed, implemented and measured.

The problem ramifications will be described next. For the formal definition, see Chapter 4. Fundamentally, as with most personnel scheduling problems, the goal is to fill a various range of demands with offers. In this case, shift demands are to be satisfied by employees. The planning horizon is a month. A shift consists of a location and a date. Locations can be either one of the houses or events, which can be either during the regular opening hours of a house or run extra. The construction work for a new exhibition is an example for a regular event, while an exclusive late-night guided tour would be an example for a shift running extra. All of the houses that are adding demands to the scheduler for the

CESP are located in Vienna, although locality does not factor into the planning problem. Every location also defines the length of a shift.

The employees can choose the days they may be scheduled for a shift. They separately apply for any shifts outside the general time range by applying to special events. This is because the general expectation when applying to a specific day is that the work will be full-time and generally during the business day, while special events often run late at night or only for a few hours. An employee cannot choose the locations they want to work at directly, simply because in the worst case, if everybody wants to work at location A which has no demands but nobody wants to work at location B which has high demand, the system could not work at all. Special shifts are obviously exempt from this rule, since they are applied to separately. Employees can, however, mark non-event locations as favourite houses, which tells the scheduling personnel—or the algorithm—their location preferences. A person can only be scheduled for one shift a day. An employee can also specify the maximum number of shifts they want to work. This can range from one to six shifts and excludes separately chosen special shifts, since the employee specifically requests them. If an employee selects at least a certain number of shifts for a month, they are also eligible for standby and reserve shifts. These shifts are distributed in addition to the chosen shifts and serve as a buffer for the administrative side of the scheduling. For example, if somebody who is supposed to cover a shift at a certain day falls sick, there are employees on standby to cover the now vacant shift. Ultimately, there is another hard cut-off for employee working hours—they cannot earn more than a certain amount, which is defined in Austria as "Geringfügigkeitsgrenze". Earning less than this amount per month is favourable from a taxation and insurance standpoint, so even if somebody chooses to work six shifts a month and volunteers for seven additional events, they will not be scheduled for more than they can earn without crossing this threshold.

When scheduling, the important goals to keep in mind is that every shift demand should be filled. Shift demands should be covered fairly, i.e., five shifts missing one of eight required employees each is better than one shift missing five. Additional resources can be shifted easier that way, and in the worst case, it is easier for seven employees to have the workload of eight than it is for three employees. Furthermore, shifts assigned should also be covered fairly. This means that it is important that if only half of the work offers given by employees can be filled because there is not enough demand, everybody should be scheduled for half the shifts they wanted. This is, of course, impossible to do in reality, so the best that can be done is as close an approximation as possible. Furthermore, the favourite house selections done by the employees should be adhered to, if at all possible. If not, they should also be distributed as fairly as possible. Naturally, there are also many obvious problems to look out for, e.g., everybody has to cover at least one shift, nobody can be over the hour threshold and so on. This is, in brief, the extent of the problem. Some intricacies have been omitted, as the problem description is complicated enough as-is. As stated before, it will be revisited and formally defined in Chapter 4.

The CESP is a personnel scheduling problem, and shares many characteristics with a lot of them, most notably the Nurse Rostering Problem (NRP), which will be used as

a reference point. The solution approaches considered and implemented are constraint programming (CP) and a variable neighbourhood search (VNS). An introduction to these methods can be found in Chapter 3. As a first step, the problem was formalised and implemented as a CP problem. If an exact method could solve the problem optimally in a reasonable amount of time, that would be the best case. Unsurprisingly, that did not turn out to be the case. Next, a general VNS (GNVS) was implemented and used to improve the solutions provided by the CP. Afterwards, a greedy construction heuristic was implemented to help with diversification and with real-world applicability of the problem, since it uses far less resources than the CP solver. Implementation details will be shown in Chapter 5. The results of the methods and of their hybridization are further discussed in Chapter 6.

Following this chapter, Chapter 2 offers a general view of the NRP and the most striking differences to the CESP, followed by an overview of the current field. Chapter 3 addresses the methodology used to create the solution approaches. Chapter 4 defines the problem formally and also includes a proof for NP-hardness. Chapter 5 describes the solution approaches in detail. Finally, Chapter 6 gives an overview of the computational results and Chapter 7 deals with future plans and a reflection of the work.

State of the Art

2.1 Overview of the Nurse Rostering Problem

The nurse rostering problem (NRP), also called the nurse scheduling problem (NSP) is a combinatorial timetabling problem about scheduling a set of nurses over a certain time frame. It has been studied for decades [6] and is still a very relevant problem in modern times. In general, the main aspects of the problem definition are that there is a set of employees, often nurses in a hospital or another healthcare environment, which should be scheduled for a certain period of time, usually about a month. In most cases, the needed variables and constraints are roughly as follows, as stated by Burke et al. [6]:

- Planning period: the time frame for which the planning is undertaken—often between a week and a month.

- Skill category: different nurses can have different skills or responsibilities and certain shifts can require certain skills.

- Shift type: hospitals often operate on several shifts per day, a common system is having three 8-hour shifts; there can also be different stations, some of which may have certain shift requirements.

- Coverage constraints: these constraints represent the requirements for nurses and skill sets for every shift.

- Time constraints: these constraints represent every time related constraint on the schedule, like personal preferences, amount of shifts a week per person or the common rule of not being able to do two consecutive shifts, even across days (i.e., having a morning shift on the day after a night shift).

- Work regulations: constraints from the workplace, often affecting time constraints, i.e., laws that prevent two shifts without a minimum resting time between.

The constraints just presented are often divided into hard and soft constraints. Hard constraints must be satisfied to produce a feasible solution, while soft constraints should be adhered to, but can be violated in order to generate a viable solution, i.e. it is encouraged to give a nurse a day off if she requests it, but if it is not possible, it is still a valid solution. There can be quite a few soft constraints in any schedule, and the relation between the different constraints is very important for the quality of the solution in practise, and very subjective. A good example might be that a shift needs to be staffed with at least 15 nurses, but 20 would be ideal. A nurse could work the shift, but has requested a day off. If the amount of scheduled nurses is below 15, the answer is obvious, but if they are 15 or 19, the correct answer, and therefore, the better solution, might be more ambiguous. Furthermore, some regulations could be both hard and soft constraints; if a nurse is employed for a full-time job of 40 hours a week, scheduling 45 hours might be allowed, but discouraged, while anything over 50 hours might be forbidden.

Since the problem is firmly rooted in the real world and many studies have been made with real-world examples first and foremost, there are lots of special cases for every variable or variation which was just presented. The way that time-related constraints are handled is probably the most flexible of all, since a lot of the details are just between the nurses and their supervisors — there are many variations of different regulations on how shifts may follow each other, how many free days must be given after a certain number of shifts and how much importance is given to scheduling around personal preferences. A good indication of the different approaches can be seen in Burke et al. [6].

After this brief overview, here is a simple example for a possible NRP variant:

Facet	Text	Definition
Nurses	3	$N = \{A, B, C\}$
Planning period	2 days	$P = \{D_1, D_2\}$
Skill category	2 categories	$S = \{S_1, S_2\}$
Shift types	2 shift types, early and late	$T = \{E, L\}$

- Hard coverage constraints: at least one nurse per shift, regardless of skills

- Soft coverage constraints: all shifts covered

- Time constraints: no two shifts without a shift rest in between

Following this is the exact roster of available employees and requirements in Table 2.1. Two possible solutions for this problem are shown in Tables 2.2 and 2.3.

Table 2.1: NRP overview: Nurses and shift requirements

Nurse	Skills
A	S_1
B	S_1, S_2
C	S_1

Requirements	D_1^E	D_1^L	D_2^E	D_2^L
S_1	1	1	2	2
S_2	0	1	1	0
Sum	1	2	3	2

Table 2.2: NRP overview: Solution A

Requirements	D_1^E	D_1^L	D_2^E	D_2^L
S_1	A	C	A	B,C
S_2		B		
Sum	1(1)	2(2)	1(3)	2(2)

Table 2.3: NRP overview: Solution B

Requirements	D_1^E	D_1^L	D_2^E	D_2^L
S_1	A	C	A	C
S_2			B	
Sum	1(1)	1(2)	2(3)	1(2)

As can be seen in the example, even such a simple problem can have multiple solutions. While solution A has a higher amount of scheduled nurses and therefore less unfilled demands, it schedules only one person for a shift which requires three nurses at best, while solution B schedules two nurses for this shift. Without knowing more about the penalties of the soft constraint violations and the objective function of the problem, even these simple solutions cannot be easily compared.

While it is obvious that most variants of the NRP are complex to solve, it is difficult to find the definitive NRP version due to the sheer possibilities of variants and the origin as a real-world problem. The NRP can generally be seen as a timetabling problem, which has been proven to be NP-complete[8] in the general form. A general mathematical model and NP-hard cases are presented in Brucker et al. [4]. In Chapter 4.6, NP-hardness will be shown for the specific variant presented in this thesis.

In reality, the NRP is still highly relevant as a general scheduling problem, since it can be encountered everywhere in the real world. The smallest private clinic or even a supermarket can be seen as solving a variant of the NRP on a regular basis, often using highly unique variants of the constraint types presented earlier, or entirely different ones.

2.2   Differences Between the NRP and the CESP

While there are many similarities between the NRP and the problem discussed in this thesis, there are also enough differences to warrant a dedicated overview, which will be

given in this chapter. Differences will first be explained and summarised at the end of the chapter in Table 2.6.

The NRP generally schedules employees with a fixed employment contract, mostly full-time. They can generally be expected to be available at all times (or at least working in a predictable, steady pattern), while holidays or sick leave are special cases that have to be managed. Employee recruiting also follows this trend, it is comparably slow and an important decision, since employees cannot easily be hired for just one tough month and then let go again. This is more of an organizational problem with one possible solution being a regional pool of nurses, described by Gutjahr and Rauner [13].

The challenges in scheduling are often based on real workplace constraints which involve relationships between several days. For example, a common constraint is that after a night shift, the nurse has to have a free day (or at least not a morning / day shift). Thus, days are interconnected, and if day d_1 is changed, day d_2 might be violating constraints because of this change.

In contrast to this, the CESP schedules part-time employees who can decide on which days they want to work. They can be scheduled at any day they want to work without any constraints depending on the day before or afterwards. Additionally, they are only available for a certain amount of time each month—far less than expected from a full-time employee. It is possible that an employee is only available for two days in one month and ten in the next. These core rules produce interesting effects. For example, if a shift on day d_1 cannot be staffed with enough personnel, it is possible to staff it adequately by unassigning person A from a shift on day d_2 and assigning them to day d_1, while assigning person B to day d_2. This could be possible if person A chose both day d_1 and d_2, but person B chose only day d_2. For better clarity, a slightly expanded example of this is shown in Table 2.4 and 2.5.

Table 2.4: CESP example: Employee and shift requirements

Employee	Days
A	d_1
B	d_1, d_2
C	d_2, d_3
D	d_3

Requirements	d_1	d_2	d_3
Employees	2	1	1

Table 2.5: CESP example: Solutions

Solution A	d_1	d_2	d_3
Employees	A	B	C

Solution B	d_1	d_2	d_3
Employees	A,B	C	D

Solution A could be a generated by a simple greedy approach, where B is assigned to y, because y is still empty and it is normally regarded as a better move than fully staffing x. At the end, D is not scheduled to any shift, because z was filled by C, and x is left one employee short.

In the CESP, it is not uncommon to have the number of requirements fluctuate wildly between different days, depending on factors like the day of the week, events and general on-goings in the facilities. The employees requesting shifts do not know about this and request based on their personal preferences, which means that there can be substantial discrepancies during meaningful dates, i.e., Christmas. In general, it is best if the amount of work offered is enough to fill all requirements but does not exceed the requirements too much, since it is ideal if everybody can get all the shifts they request. In any case, a very relevant part of the assignment is fairness. Either to the shifts in how far they are understaffed, or to the employees in how many of their shift requests cannot be granted.

Following is Table 2.6, granting a rough overview of the core differences between the NRP and CESP. Coverage constraints and working regulations have been omitted since they are the most domain-specific and do not contribute a lot to the list of differences.

Table 2.6: Differences between NRP and CESP

Category	NRP	CESP
Planning period	Usually 2 – 8 weeks, often 4	4 weeks
Skills	Often used and critical to constraints	No different skill sets
Shift types	Generally 2-3 shifts per day	One shift per day
Employee contract	Conventional long-term contracts	Fluid, month-by-month basis
Employee working hours	Significant amount of time, generally full-time, around 160 hours / month	Small-medium amount of time, between 5 – 60 hours / month
Employee days	Most days can be used for scheduling, often working in set rotation	Employee chooses which and how many days can be scheduled each month
Number of employees	Often dozens	Hundreds
Time constraints	Days are dependant on each other—e.g., no day shift after night shift	Days only indirectly depend on each other because of complicated requested day situation of each individual employee

2.3 Literature

The fields of personnel scheduling in general and the NRP in particular have been well-studied for decades, see the extensive research by Burke et al. [6] and Van den Bergh et al. [32]. There have been many variants of the NRP which have been solved with different approaches. Since many of the NRP variants are sourced from real world applications with unique needs and methodology, there are many approaches tailor-made for one particular variant. To increase the comparability of approaches, standardised

testing instances have been introduced over the decades. Two major examples being the First and Second Nurse Rostering Competition[7], which both introduced a problem and different instances to apply it to. Those instances are often used as benchmarks for new proposed algorithms in the field.

A sizeable amount of the research surrounding the NRP is centred around metaheuristics, with metaheuristic hybrids[28] being the norm. There have been solution approaches using tabu search, simulated annealing, variable neighbourhood search as well as population-based approaches like genetic algorithms and ant colony optimisation.

In Burke et al. [5] the authors develop an algorithm incorporating a tabu search. They initialise their schedule by either beginning from a previous solution or by starting from an empty schedule and randomly adding or removing shifts until they have a feasible schedule. Afterwards, they apply a tabu search that they hybridised with one of two custom heuristics to further improve their results. The two custom heuristics differ in their effectiveness and time-consumption. The authors also stress that they took care to develop a solution that would generate schedules which are not easy to improve by humans. This is how their second custom heuristic works—it considers and applies every beneficial shift exchange between two employees, which is similar in concept to neighbourhood operations from other research.

In Bai et al. [2], a memetic algorithm comprised of a simulated annealing hyper-heuristic (SAHH) and a genetic algorithm is implemented. The genetic algorithm uses a stochastic ranking method to rank feasible and infeasible individuals. It does so by using both an objective function and a penalty function. Each individual is then further improved by the SAHH. Since the main purpose of the SA is finding local optima, the temperature is updated throughout the algorithm and not reset at the start of every SA run. Furthermore, the performance of the low-level heuristics are tracked by the hyper-heuristic and used to guide which heuristics are used in the future.

In Tassopoulos et al. [31], a two-phase adaptive variable neighbourhood search is implemented. Phase 1 deals with the initialization and developing of an initial, randomised solution. In Phase 2, the authors developed two algorithms, one deterministic and stochastic respectively, which are selected by a dynamic probability reflecting their prior successes. These algorithms swap parts of the roster. Afterwards, six different neighbourhoods are applied sequentially. When the algorithm has not produced any improvements for a certain amount of cycles, it is considered to be stuck in a local optimum and a perturbation move is applied.

In Jaradat et al. [16], an elitist ant system metaheuristic is hybridised with an iterated local search (ILS). After constructing an initial solution per ant, the ILS is employed. If the ILS can improve the best solution, an intensification phase is triggered, in which the neighbours of the best solution are randomly explored. If not, a diversification phase is triggered, which perturbs the solution by evaporating the current pheromone trail and generating a new population of ants based on the current elite solution. Afterwards, the algorithm starts the next cycle by using the ILS.

In recent times, exact methods have also been used successfully to solve the NRP. While it is not common to be able to solve any sufficiently large instance of a NRP entirely via exact methods, hybrids of an exact method and a metaheuristic or hybrids between two exact methods have shown good results.

In Qu and He [26], the authors hybridise a constraint programming (CP) approach with a variable neighbourhood search (VNS). The authors focus on the CP aspect and note that due to the size and complexity of most NRP definitions, traditional applications of CP approaches have not been very successful for large instances. Therefore, they implemented a decomposition approach. First, partial high-quality weekly rosters are created by using a satisfiability approach within CP. Then, these weekly rosters are extended and connected by an iterative forward search using an extended model and an optimisation approach. Afterwards, the whole solution is improved by using a VNS.

In Stølevik et al. [30], CP and a variable neighbourhood descent (VND) are used within an ILS. At first, CP builds an initial solution using only hard constraints, which is subsequently improved using the VND. Afterwards, diversification is executed by deleting all the schedules for a random (2% to 30%) number of nurses. Half of the chosen schedules are random, half are picked because of their high penalty scores. After deleting those schedules, CP is employed to create new assignments. This is one cycle of the ILS.

In Rahimian et al. [27], an integer programming (IP) approach is coupled with a VND. In the initialization phase, a greedy algorithm is employed to build the initial solution. Afterwards a VND is used to improve this algorithm until the VND can find no more improvements. The IP aspect is then applied as part of a ruin-and-recreate operation, which means that parts of the solution are destroyed before being recreated by IP. After the final solution is found this way, this solution is once-again improved by IP.

As noted before, the NRP encompasses a lot of variants, among which some are closer to the CESP than the traditional models. In Gutjahr and Rauner [13], the authors consider a dynamic problem variant in which a regional group of hospitals is supported by a pool of nurses to cover shift shortages. The nurses can define their own working hours and can be assigned to any corresponding hospital needing their services during the time they specified. This problem definition resembles the way casual employees request shifts. The authors also go into detail about the common troubles with dynamic problem definitions like imperfect information and delaying assignments. The authors consider a SA and an ACO, but found that the output of the SA was below their expectations. In contrast, the ACO algorithm performed very well in their experiments, which is attributed to ACOs general good viability regarding highly-constrained problems.

2.4 Analysis, Comparison, and Summary

The literature around the personnel scheduling and the NRP in particular is vast and loaded with history. There have been successful applications of tested methods for lots of different variants, and new advances are still being made today. It can be easily seen that

nearly every moderately-known metaheuristic has been successfully applied to the NRP in one way or another. In recent years, research using population-based metaheuristics, especially ant colony optimisation, seems to yield good results, especially when coupled with strategies which can help improve candidate solutions further. The capability of ACO as being both a construction heuristic as well as a diversification strategy are a good fit for a highly constrained problem like the NRP usually is. This combination is pursued in our preliminary work [10], where we applied ACO with an embedded VND for intensification to improve initial CP solutions.

The usage of exact methods to generate an initial solution — particularly using a relaxed set of constraints — has also found success. Afterwards, improvement strategies like a VND or SA can be utilised, depending on the further need for intensification. Furthermore, solution approaches using exact methods to only improve subsets of the instance, thereby reducing the search space have also been successful. These are generally hybridized with another method to deal with the problem as a whole or to combine the improved subsets.

For the solution approaches using improvement methods, many can be reduced to similar basic neighbourhoods which are being used in many other problems, like the exchange of a single shift between two employees. On the other hand, there are also interesting and complex applications that seem tailor-made for personnel scheduling problems, such as moving a connected chunk of assigned shifts for a single employee. This move can leverage the existence of small but connected high-quality assignments without disturbing them. Moves like this are especially powerful in highly-constrained environments like the NRP, where the existing roster already follows several complex rules with regards to their structure. Simple shift-exchanges are far more likely to result in infeasible moves in such problems. Although methodically interesting, due to the differences between the CESP and the NRP as stated in Chapter 2.2, these pattern-shifts are not applicable in the CESP. This is because the strength of the pattern-shift—keeping the relationship between consecutive days—is not relevant to the CESP.

The different use of diversification strategies is also interesting. As discussed in the previous subsection, two relevant strategies found in the literature were deleting the least-quality parts of the roster and deleting the complete roster of an employee. The first strategy seems obvious, and is generally used in a probabilistic function with random assignments. If only the least-quality parts of the roster would be deterministically removed, this strategy might get stuck in another local optimum if the shaking it performs is too narrow. The second strategy, deleting the whole roster of a single employee—in comparison to just deleting single assignments—also feels logical for the NRP. If one assignment of an employee is deleted, due to the highly constrained nature of the problem, it is very possible that there are no additional moves that can be performed, since the other assignments given to the employee more or less force this assignment anyway. This is not the case in the CESP, though, since the schedules of casual employees are generally much more open. If a single assignment is deleted, it is quite possible that the employee can be assigned to a dozen or more other assignments, depending on the offerings of the employee and the state of the schedule.

12

CHAPTER 3

Methodology

In this chapter, we follow Papadimitriou and Steiglitz [25]. The CESP is fundamentally a combinatorial optimisation problem. Optimisation in this context refers to finding the best possible solution for a problem from a set of possible solutions. This can be contrasted to a satisfaction problem, where the relevant output is only if something is possible or not and there is no notion of solution quality. A question that would be answered by a satisfaction problem is "Is it possible to get from A to B?" while a similar question for an optimisation problem is "What is the shortest way from A to B?". Optimisation problems divide into two groups—problems with continuous and problems with discrete variables. The later group is also called combinatorial problems, and this is the focus of this chapter and the problem presented here.

We define an optimisation problem as follows:

Definition 1 *An optimisation problem is a set of instances. An instance is a pair (S, f), where S is the set of all feasible solutions (also known as a solution space or search space) and f is a cost function $f \colon S \to \mathbb{R}$, often called objective function. The problem is to find an $x \in S$ so that $f(x) \leq f(y) \quad \forall y \in S$, which is called a globally optimal solution.*

The methods in this chapter are geared towards problems that are in the complexity class of NP-hardness. NP-hard problems are at least as hard as any problem in NP. Problems in the complexity class of NP are defined as such that they can be solved in polynomial time by a non-deterministic Turing machine. It is, however, possible, to verify the validity of a solution in polynomial time.

To define the non-deterministic Turing machine, we will use the definition used by Karp [17, p. 91]:

Definition 2 *A nondeterministic algorithm can be regarded as a process which, when confronted with a choice between (say) two alternatives, can create two copies of itself,*

13

and follow up the consequences of both courses of action. Repeated splitting may lead to an exponentially growing number of copies; the input is accepted if any sequence of choices leads to acceptance.

In practical terms, this means that after a certain instance size, it is widely believed that there exists no algorithm that can reliably find a valid solution for an NP-hard problem without requiring an unreasonable amount of time. Instance size matters for practical purposes since all possible combinations for a tour through four cities is still solvable on paper, and computers are capable of many thousands of calculations per second, so small instances can still be computed in—in human terms—reasonable time. Although computation might be fast enough, the process is still not efficient.
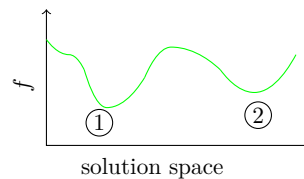

Figure 3.1: Objective function graph with global (1) and local (2) minimum

Figure 3.1 shows an approximation of the solution space and the objective function for an optimisation problem. For ease of reference, all problems in this thesis will be assumed to be minimisation problems. The y-axis denotes the objective function while the x-axis describes a 1D embedding of the solution space of a problem instance. There are two interesting points in the figure. Point(1) has already been encountered in the definition before, and is called the global minimum or optimum. It is the lowest point in the graph and accordingly, the best solution in the solution space. Finding this point is the main objective. Point (2) is called a local minimum. This is the lowest point in the immediate vicinity — in each direction along the solution space, the objective function increases. In NP-hard problems, it is generally not easy to verify if any point is the global minimum.

The problems in this chapter are often complex and generally very time-consuming to solve once they approach a certain instance size. There exist two general approaches to solving those problems, exact and heuristic methods. Exact methods have to reason about the whole solution space and are guaranteed to find the global minimum eventually. This is not to be confused with simply examining every single possible solution, however. There are many different techniques and algorithm in existence which enables them to reduce the amount of time needed and solutions to check. For example, in some problems it is possible to identify certain variable values and then exclude any part of the solution space including this value. Some of these algorithms will be shown in the next section. Often, exact methods are not feasible because the time it would take them to terminate is too long. Heuristic methods, in contrast, make no claims about finding the global minimum. Instead, they take shortcuts through the solution space and generally try to follow the most promising lead in an effort to find good enough solutions in a reasonable time frame. Since heuristic methods do not work through the solution space

in an exhaustive manner, they are generally not able to identify a solution as the optimal solution, either. Because of the complexity of many optimisation problems, heuristic methods are often the only way to find reasonable solutions in time.

In the following sections, these ways of solving problems will be described in more detail. First, constraint programming, an example of an exact method will be shown. Afterwards, the the general idea of heuristic methods and especially the Variable Neighbourhood Search will be explained.

3.1 Constraint Programming

In this chapter, we follow Rossi et al. [29]. Constraint Programming (CP) is a programming paradigm and is generally used to solve constraint satisfaction problems (CSP).

Definition 3 *A CSP P is a triple $P = (V, D, C)$, where V is a set of n variables $V = (v_1, ..., v_n)$, D is a set of n domains that corresponds to these variables $D = (D_1, ..., D_n)$ and C is a set of t constraints $C = (C_1, ..., C_t)$. Each constraint C_j is a pair $C_j = (R_j, S_j)$ whereby $S_j \in V$ is a subset of variables and R_j is a relation on those variables. The objective is to find a set of values $A = (a_1, ..., a_n)$ where $a_i \in D_i$ and each C_j is satisfied for the chosen domains of the variables in S_j. Possible solutions of a CSP can be a valid instantiation of values for each variable or the information that the problem is infeasible. Rossi et al. [29, Chapter 2.2.1]*

A simple example of a well-researched problem in CSP is the following map colouring problem (MCP). In MCP, there is a set of nodes, which are adjacent to any number of other nodes. The task is to assign each node a colour that is different from any colour that an adjacent node has assigned. It is inspired by colouring a given map of countries and trying to have no country be the same colour as any of their neighbours. The decision variant is to find out if a given problem instance can be solved with x colours, while the optimisation variant minimises x.

In this problem instance, there are three variables. All of them have the same domain, consisting of $(red, green, blue)$, the available colours that can be chosen. The constraints ensure that adjacent variables cannot have the same value, so that bordering regions of the map cannot have the same colour.

The definition of this problem is as seen on Table 3.1.

As can be seen in the depiction of the problem in Figure 3.2, B and D are not adjacent. Since they both have the same two adjacent neighbours, a solution can only be valid if B and D have the same colour. A valid solution for this problem is therefore

(A=red ,B=green ,C=blue ,D=green)

15

Table 3.1: Example of a simple map-colouring problem instance

V	D_v
A	(red,green,blue)
B	(red,green,blue)
C	(red,green,blue)
D	(red,green,blue)

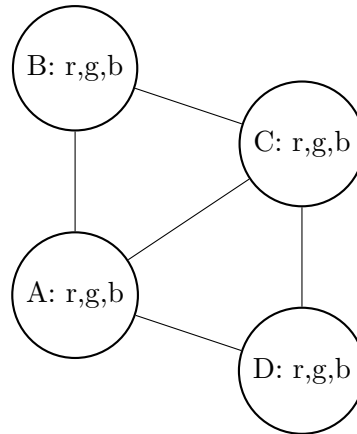| n	C_n
1	(A $\neq$ B)
2	(A $\neq$ C)
3	(A $\neq$ D)
4	(B $\neq$ C)
5	(C $\neq$ D)

Figure 3.2: Simple map-colouring problem instance

Generally, a CSP will be solved by a CP-solver, a specialised type of software that takes CSP-instances as input and outputs solutions. CP-solvers have been in development for decades and employ a vast array of optimised methods and algorithms to solve applicable problems very fast and in an efficient manner, but in their most basic form, they use two central techniques which are called backtracking and constraint propagation.

Backtracking Backtracking is essentially a depth-first search of the solution space represented by a search tree [20]. In backtracking, the variables are sequentially assigned values, which is referred to as partial instantiation. After every assignment, all relevant constraints are checked. If the solution candidate is still viable, the next variable will be assigned. If not, the solver will revert the assignment and assign another value from the domain to the most recently changed variable that still has alternative values in their domain. Since the solver is working with partial instantiation, infeasible parts of the solution space can potentially be eliminated without having to exhaust them fully. For this, consider a slightly different map-colouring problem as shown in Figure 3.3.

In contrast to our original problem from Figure 3.2, the variable D has been omitted and the domain of B has been reduced to include only $D_b = (red)$. The constraints containing D have also been omitted. If the CP-solver starts by instantiating A to red and then tries to instantiate B, it can discover instantly that there are no possible solutions starting
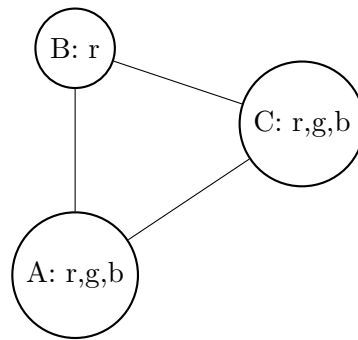
Figure 3.3: Slightly different MCP as Fig. 3.2, reduced domain of B, removal of D

with this partial instantiation. Therefore, C does not have to be instantiated at all and the solver can backtrack to a different value for A.

In the same vein, backtracking suffers from a fundamental problem: consider three variables V_i, V_j, V_k which are instantiated in this order and a constraint C_1 that leaves no valid instantiation for V_k if $V_i = a$. If V_i is instantiated to a, the algorithm will continue by instantiating V_j successfully to any value and ultimately fail with V_k. It will backtrack to V_j and subsequently fail to instantiate any value for V_j before finally backtracking to V_i and will encounter the same sort of problem for any instantiations stating $V_i = a$. Referring back to our previous example in Figure 3.3, this would be the case if the solver started with A and then moved to C before moving to B. This problem can be solved by a technique called constraint propagation.

Constraint Propagation Constraint propagation is used to remove invalid parts from the search space before the search begins, thereby decreasing the search space. This does not exclude any valid solutions, since everything that is removed cannot be part of a valid solution anyway. There are two major types, node and arc consistency.

Node consistency is enforced by considering every unary constraint and deleting any values that violate the constraint. Consider yet again Figure 3.3 with the added constraint $C_4 = \{C = blue\}$. Clearly, any instantiation of $C <> blue$ is invalid and can be safely removed from the domain D_c before starting the search. The problem looks like Figure 3.4 afterwards.

Arc consistency is the binary, directional equivalent to node consistency. An arc (V_i, V_j) is considered consistent, if for every instantiation of V_i there is a valid instantiation for V_j. Note that it is directional, so a consistent arc (V_i, V_j) does not imply a consistent arc (V_j, V_i). If there are instantiations for V_i that do not allow for any valid instantiations for V_j, the instantiation for V_i is discarded from the domain of V_i. This is then repeated for every arc in the problem. The deletion of values from the domain of a variable can cause any arcs that the variable is part of to lose arc consistency. This means that an arc can be revisited a number of times during the enforcement process. If every arc between variables is arc consistent, the whole problem is arc consistent.
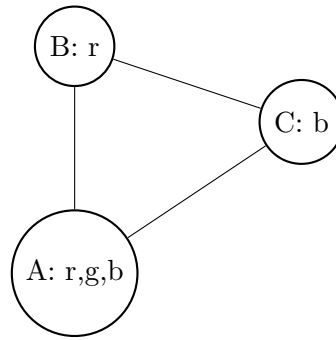
Figure 3.4: Constraint on C restricting it to blue, node consistent

To further illustrate the functionality of arc consistency, the AC-3 algorithm [22] will now be presented in detail. The AC-3 is perhaps the most well-known arc consistency algorithm and is both simple to understand and efficient. It is described in its entirety in Algorithms 2 and 1.

Algorithm 1 Revise

Input: variable i
Input: constraint c
Output: boolean *change*, which states if the arc has been revised
Output: potentially reduced domain D_i
1: **procedure** Revise(i,c)
2:     *change* $\leftarrow$ *false*
3:     **for** $a_i \leftarrow$ each element of D_i **do**
4:         **if** $\nexists a_j \in D_j$ in c that supports assignment to a_i **then**
5:             remove a_i from D_i
6:             *change* $\leftarrow$ *true*
7:         **end if**
8:     **end for**
9:     **return** *change*
10: **end procedure**

Algorithm 2 shows the main loop. In it, every directed arc is added to a list, which is then sequentially revised by calling the function of the same name. *Revise*, as described in Algorithm 1 basically reduces the domain of variable v_i by checking if, for any value $a_i \in D(v_i)$ a value $a_j \in D(v_j), j \in V(c)$ so that the constraint c is satisfied. If not, a_i is removed from D_i. In simpler terms, if i cannot be instantiated to a_i without violating c, a_i is removed from the domain of v_i. If the call to *Revise* reduces D_i, each constraint including v_i is then added to Q. The algorithm ends if Q is empty or a domain has been completely exhausted. As domains are only ever reduced and if no domain is reduced, the queue is reduced, this algorithm does not cycle. Furthermore, since only constraints which include v_i are added to Q after a successful *Revise*, it is also quite

---

Algorithm 2 AC3

Input: set of variables V
Input: set of constraints C
Output: true if arc consistent, false if any domain is empty

1: **procedure** AC3(V,C)
2:      $Q \leftarrow \{(v_i, c) | c \in C, v_i \in V(c)\}$ ▷ binary arcs are therefore two occurrences
3:      **while** $Q \neq \{\}$ **do**
4:          remove (v_i, c) from Q
5:          **if** $Revise(v_i, c)$ **then**
6:              **if** $D(v_i) = \{\}$ **then**
7:                  **return** $false$
8:              **else**
9:                  $Q \leftarrow Q \cap \{(v_j, c') | c' \in C \wedge c' \neq c \wedge v_i, v_j \in V(c') \wedge j \neq i\}$
10:              **end if**
11:          **end if**
12:      **end while**
13:      **return** $true$
14: **end procedure**

---

effective in reducing unneeded calls to *Revise*. There are more efficient algorithms, but the AC3 shows the fundamental way that arc consistency can be easily implemented in a polynomial algorithm.

The problem of backtracking mentioned before can be efficiently solved by enforcing arc consistency. Figure 3.5 shows the former problems of Figure 3.3 and Figure 3.4 adjusted for arc consistency. As can be seen, the problem in Figure 3.4 is already solvable without resorting to backtracking at all.
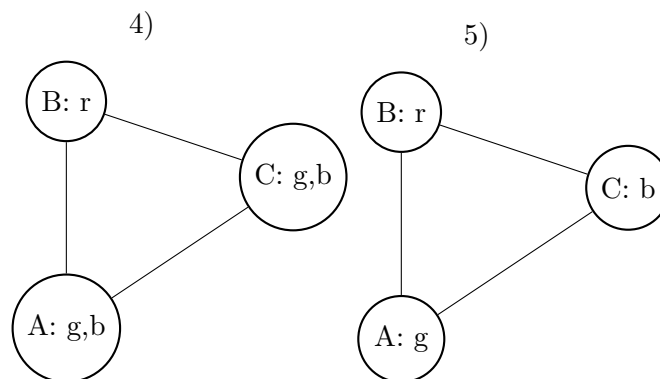

Figure 3.5: Arc consistent problems from Figure 3.3 and Figure 3.4

Variable Ordering Heuristics Another interesting and central technique is the selection on where to begin the instantiation. This is known as a variable ordering heuristic. There are basically two categories of heuristics for variable ordering. The first category is concerned with the domain size of variables, while the second is concerned about the structure of the CSP. For the first category, it is often most promising to begin with the most restricted variables. In small problems, it is easy to work out which variable is the most constrained, but in large instances, this might not be the case. In those cases, domain knowledge can be used to tell the solver which variables to focus on first. The second category uses the fact that a CSP can be represented as a graph. Heuristics in this group can work in different ways, e.g., finding cycles in the graph and instantiating variables to cut those cycles, thereby making it a tree. The tree can then easily be solved via arc consistency.

For the next example, constraint propagation will not be considered, for the sake of keeping the problem simple. In reality, of course, it works especially well with variable ordering by domain size. We are going back to the state of our problem during the backtracking example in Figure 3.3, in which the domain of variable B was reduced to $B = (red)$. A variable ordering heuristic that assigns the variable with the smallest domain first would encourage the instantiation of B first, which is promising for this problem. If the solver would start instantiation with $A = red$, it might need to exhaust a good number of guesses before coming to the conclusion that $A = red$ is not part of any feasible solution, especially if it continues by instantiating C afterwards. When starting with B, there is no other way than starting with $B = red$.

More details about constraint propagation and ordering heuristics can be found in Chapters 3 and 4 of [29].

3.2 Heuristic Optimisation

In this chapter we follow Blum and Roli [3]. In contrast to exact methods, heuristic methods do not aim to prove optimality. Instead, they try to provide good—ideally near-optimal—solutions in a reasonable time-frame. Heuristic methods applied to NP-hard problems can never know if they found an optimal solution on their own. One of the most relevant ways to categorise different types of heuristic methods is by how they are applied to solutions. This differentiation yields two categories—construction and improvement heuristics. Construction heuristics usually start with an empty solution and add parts until the solution is at least feasible. This then yields a minimally feasible solution. Construction heuristics can also just build a complete solution, which is then simply called an initial solution. Improvement heuristics, on the other hand, start with an initial solution (often provided by a construction heuristic) and then iteratively improve this solution. The biggest part of this chapter will be devoted to improvement solutions.

The simplest way to employ an improvement heuristic is in form of a neighbourhood search, also called local search or local neighbourhood search.

Definition 4 *A neighbourhood structure is a function $N : S \rightarrow 2^S$ that assigns to every $s \in S$ a set of neighbours $N(s) \subseteq S$. $N(s)$ is called the neighbourhood of S. [3, p. 269]*

The selection of the neighbours is done by a step function or improvement strategy, which is generally a simple change in the solution. For this thesis, neighbours do not have to be feasible, which means that the change applied to the solution could move them from a feasible to an infeasible state. A neighbourhood in a scheduling problem could be swapping the shifts of two employees or adding another shift to an employee. A neighbour $y \in N(x)$ of a solution x therefore is a solution that had a corresponding neighbourhood operation applied upon it. The neighbour is then evaluated and if it meets certain conditions, is selected as the next solution. Otherwise, the neighbour is discarded. There are different ways to select neighbours, which are either deterministic or non-deterministic and the most common ones are referred to as first improvement, best improvement, and random ordering. First improvement, also known as next improvement, is a strategy that searches the neighbourhood in a specified order and returns the first neighbour that has a better objective value than the original solution. Best improvement is a deterministic strategy searches through the whole neighbourhood and returns the best neighbour. Random ordering is, as the name implies, non-deterministic and returns a random neighbour. For first improvement and random ordering the order of the neighbours is of relevance, therefore different implementations of first improvement can lead to different results with the same inputs. Best improvement searches the whole neighbourhood and is therefore not affected by ordering, but is slower than first improvement. The general local search is presented in Algorithm 3.

Algorithm 3 Local search

Input: solution x
Input: neighbourhood structure N
Output: improved solution w.r.t. neighbourhood N x
1: **procedure** Local search(x, N)
2:     **repeat**
3:         choose an $y \in N(x)$ ▷ generally best or first neighbour
4:         **if** $f(y) \leq f(x)$ **then**
5:             $x \leftarrow y$
6:         **end if**
7:     **until** stopping criteria met
8:     **return** x
9: **end procedure**

Heuristics have a problem that exact methods do not: they are not guaranteed that they find the best solution, or even any particularly good solution at all. As discussed before, the solution space in any moderately difficult problem is complex and heuristics can easily get stuck in local optima, as seen in Figure 3.6, where f denotes the objective value reachable by the neighbourhood. Since the goal is to minimise the objective value,

when starting at this particular part in the solution space, local search (as presented in Algorithm 3) has no possibility of escaping from the local optima.
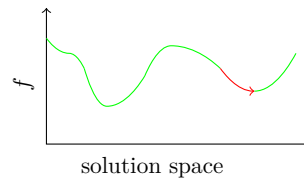

Figure 3.6: local search converging toward local minimum

The solution to this problem lies in metaheuristics. Whereas heuristics are generally simple constructs that solve a specific problem, metaheuristics are algebraic frameworks that are problem-independent. They use subordinate, more problem-specific heuristics and guide them to efficiently exploit and explore the solution space.

A central aspect to how metaheuristics work are the terms intensification and diversification. Intensification is a measure of the capability of a method to improve the current solution. Local search is a simple example for an intensification method. Diversification is the ability to search different parts of the solution space than the current one. This is what the local search method sorely lacks. A simple example would be to randomly generate solutions. Both capabilities need to be present for a metaheuristic to be successful. There are many different metaheuristics, each of them providing different advantages and disadvantages depending on the problem, so care must be taken which metaheuristic is applied to solve a specific problem. For example, there exist specialised metaheuristics for very large problem instances and for very constrained problems. Furthermore, different metaheuristics can be customised and tuned in very different ways.

3.2.1 Variable Neighbourhood Search

For this problem, a General Variable Neighbourhood Search (GVNS) was chosen, which is a variant of the Variable Neighbourhood Search (VNS) . The VNS was first introduced by Mladenović and Hansen [23] together with a few variants. Generally, metaheuristics tend to have a theme, which in the case of VNS is the structured change of neighbourhoods. Compared to many other metaheuristics, VNS is conceptually relatively simple and can therefore easily be tuned or adapted. This problem in particular lends itself to a VNS, since there are many naturally arising neighbourhoods.

VNS is a single-solution metaheuristic. These kinds of metaheuristics start with an initial solution and try to improve it incrementally by utilizing neighbourhood moves. In the following, the basic concepts of the VNS as described in Hansen and Mladenović [14] will be outlined.

VNS is based upon the three following facts:
Fact 1 A local minimum w.r.t. one neighbourhood structure is not necessary so with another.

22

Fact 2 A global minimum is a local minimum w.r.t. all possible neighbourhood structures.

Fact 3 For many problems, local minima w.r.t. one or several neighbourhoods are relatively close to each other.

From Fact 3 follows that knowledge of one local minimum is helpful in the search of the global minimum, although it may not be the global minimum itself. The authors further explain that those three facts can be used in three different ways:

- Deterministic

- Stochastic

- Both

Variable Neighbourhood Descent The Variable Neighbourhood Descent (VND) is obtained when implementing the deterministic method. It is obtained by changing neighbourhoods in a predictable manner and by exhausting the neighbourhoods. It is very efficient and thorough in finding the local minimum w.r.t. the present neighbourhoods. The VND does not have any means of randomness, which would help travel to different parts of the solution space. Instead, it relies on many neighbourhoods that aim to cover a large part of the solution space. Often, there are neighbourhoods of different sizes present in a VND, with the smaller neighbourhoods placed at the beginning of the neighbourhood set and the large and computationally expensive neighbourhoods at the end of a neighbourhood set. The VND is shown in Algorithm 4 with a best improvement step function. The algorithms in the later chapters also use best improvement, although first improvement is also possible.

Reduced VNS The stochastic method is called Reduced VNS (RVNS), and is achieved by including a shaking operation instead of a local search, i.e. choosing a random neighbour from the neighbourhood. Through the shaking operation, a jump in the solution space is performed, and local minima can be evaded this way. It is useful on very large problem instances, when local search is costly. The downside is that the improvements are not applied in an efficient, structured form and thus results can be erratic and non-repeatable. The algorithm is presented in Algorithm 5. For the RVNS (and any non-deterministic algorithms), additional stopping criteria must be created, since the random nature of shaking will never be exhausted. Such stopping criteria are generally time- or iteration-related, e.g. n cycles without improvement.

Basic VNS Finally, both approaches to the problem are combined in the basic VNS (VNS). It features the shaking mechanism from the RVNS together with a local search procedure. It is shown in Algorithm 6. The shaking gives the VNS the needed properties of diversification, while the local search procedure subsequently provides intensification.

Algorithm 4 Variable Neighbourhood Descent

Input: solution x
Input: set of neighbourhood structures N, for $N_1, ..., N_{kmax}$
Output: improved solution w.r.t. set of neighbourhood structures N

1: **procedure** VND(x, N)
2:     $k \leftarrow 1$
3:     **repeat**
4:         find y with $f(y) \leq f(y'), \forall y' \in N_k(x)$
5:         **if** $f(y) < f(x)$ **then**
6:             $x \leftarrow y$
7:             $k \leftarrow 1$
8:         **else**
9:             $k \leftarrow k + 1$
10:         **end if**
11:     **until** $k > k_{max}$
12:     **return** x
13: **end procedure**

Algorithm 5 Reduced Variable Neighbourhood Search

Input: solution x
Input: set of neighbourhood structures N, for $N_1, ..., N_{kmax}$
Output: improved solution w.r.t. set of neighbourhood structures N

1: **procedure** RVNS(x, N)
2:     **repeat**
3:         $k \leftarrow 1$
4:         **repeat**
5:             $y \leftarrow$ random $N_k(x)$
6:             **if** $f(y) < f(x)$ **then**
7:                 $x \leftarrow y$
8:                 $k \leftarrow 1$
9:             **else**
10:                 $k \leftarrow k + 1$
11:             **end if**
12:         **until** $k > k_{max}$
13:     **until** stopping criterion met
14:     **return** x
15: **end procedure**

General VNS As an additional variant, the General VNS was introduced in Hansen et al. [15]. It is simply a VNS where the embedded local search has been replaced with a VND. It has both the advanced intensification capabilities that the VND possesses (with regards to local search) and the diversification properties of the VNS. It has to be

---

Algorithm 6 Basic Variable Neighbourhood Search

Input: solution x
Input: set of neighbourhood structures N, for $N_1, ..., N_{kmax}$
Input: local search LS
Output: improved solution w.r.t. set of neighbourhood structures N
 1: **procedure** VNS(x, N)
 2:     **repeat**
 3:         $k \leftarrow 1$
 4:         **repeat**
 5:             $y \leftarrow$ random $N_k(x)$
 6:             $y' \leftarrow LS(y)$
 7:             **if** $f(y') < f(x)$ **then**
 8:                 $x \leftarrow y'$
 9:                 $k \leftarrow 1$
10:             **else**
11:                 $k \leftarrow k + 1$
12:             **end if**
13:         **until** $k > k_{max}$
14:     **until** stopping criterion met
15:     **return** x
16: **end procedure**

---

noted that the neighbourhoods of the VND and the VNS are generally disjunct since their purpose is completely different. VND neighbourhood structures tend to focus on intensification by (deterministic) improvement while VNS neighbourhoods focus on diversification by shaking and disrupting the solution. A common neighbourhood for the shaking is a neighbourhood that randomly destroys part of the solution. The resulting solution is generally worse than the original solution, but will be improved by the VND in the next step. If the VND-improved solution is better than the original, it will become the new best solution. This is further illustrated in Algorithm 7.

As a simple example of different neighbourhoods working together, a VND will be introduced. The structural change of neighbourhoods is the central aspect of any of the introduced VNS-variants, and the VND is most concise for a presentation. Consider the Travelling Salesman Problem (TSP). In the TSP, a travelling salesman needs to travel through all the cities on his route in the most efficient way possible. A simple TSP-instance with a feasible, but obviously inefficient initial solution is illustrated in Figure 3.7. The example was originally taken from [19] and subsequently modified to better fit this explanation. The solution is to be improved by a VND as described in Algorithm 4 and has a set of neighbourhoods N with $|N| = k = 2$ defined as follows:

- N_1: 2-opt: in the TSP, 2-opt is generally defined as exchanging two links in a tour—that is, removing two links and then reattaching the open segments in a

---

Algorithm 7 General Variable Neighbourhood Search

Input: solution x
Input: set of VND neighbourhood structures N^D, for $N_1^D, ..., N_{kmax}^D$
Input: set of VNS neighbourhood structures N^S, for $N_1^S, ..., N_{lmax}^S$
Output: improved solution w.r.t. set of neighbourhood structures N^D, N^S

  1: **procedure** GVNS(x, N^D, N^S)
  2:      **repeat**
  3:          $l \leftarrow 1$
  4:          **repeat**
  5:             $x' \leftarrow$ random $N_l^S(x)$
  6:             $k \leftarrow 1$
  7:             **repeat**
  8:                 find x'' with $f(x'') \leq f(x'''), \forall x''' \in N_k^D(x')$
  9:                 **if** $f(x'') < f(x')$ **then**
10:                    $x' \leftarrow x''$
11:                    $k \leftarrow 1$
12:                 **else**
13:                    $k \leftarrow k + 1$
14:                 **end if**
15:             **until** $k > k_{max}$
16:             **if** $f(x') < f(x)$ **then**
17:                $x \leftarrow x'$
18:                $l \leftarrow 1$
19:             **else**
20:                $l \leftarrow l + 1$
21:             **end if**
22:          **until** $l > l_{max}$
23:      **until** stopping criterion met
24:      **return** x
25: **end procedure**

---

different manner than before

- N_2: 3-opt: similar to 2-opt, only that three links are exchanged

Note the order of the neighbourhoods. The 2-opt, which is narrower and computationally less intensive is used first. Afterwards, if the 2-opt neighbourhood is exhausted, the 3-opt neighbourhood is used to apply larger changes to the solution. If the 3-opt is able to improve the solution, 2-opt might find different changes again, so the VND goes back to the first neighbourhood, 2-opt. VND does not require neighbourhoods to be of different sizes, they can also just tackle different parts of the solution space, like a neighbourhood that adds nodes and one that moves nodes.
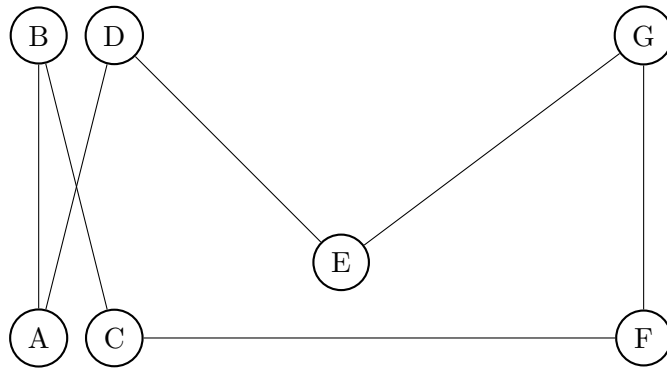
Figure 3.7: Initial TSP solution

When executing the VND, the first iteration of the 2-opt neighbourhood will return the best neighbour of the current solution, which might be a straightening of the left side of the links in Figure 3.7. A solution for that is $A \rightarrow C$ and $B \rightarrow D$ and looks like illustrated in Figure 3.8.
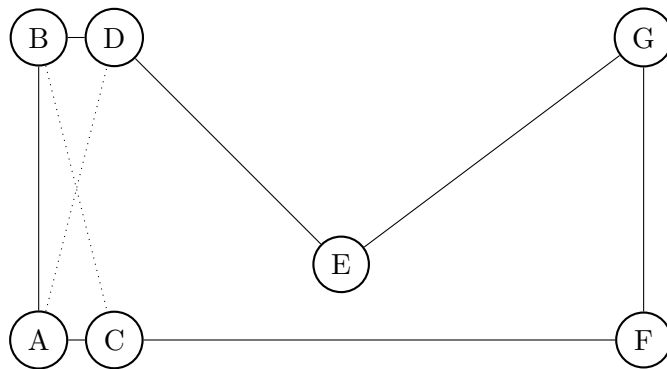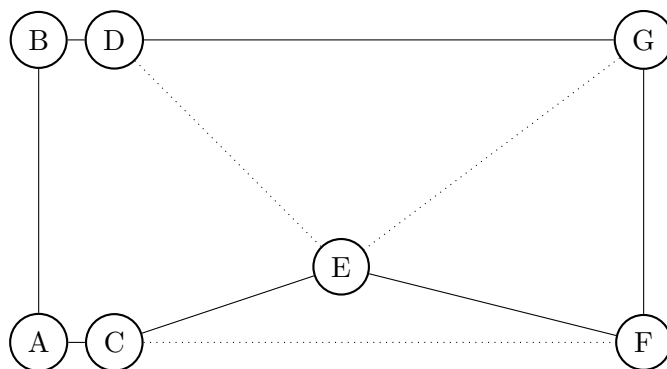

Figure 3.8: TSP after 2-opt

Figure 3.9: TSP after subsequent 3-opt

After the 2-opt improvement, the tour looks far better, but it is still not very efficient. The remainder of the tour is tangled in a more complex way though, and 2-opt will be unable to improve this problem. At some point, it will fail to find even minor improvements and k will be incremented by one, therefore triggering the usage of the 3-opt neighbourhood. The 3-opt neighbourhood will in turn be able to fix the problem by changing the links to $C \to E$, $E \to F$ and $G \to D$. The result looks like Figure 3.9. Afterwards, the neighbourhood value k will be reset to 1 and 2-opt will be active again. In this instance, there are no more improvements to be made and both 2-opt and afterwards 3-opt will not find better solutions and terminate with the proposed solution.

There are many different metaheuristics that can be used to solve complicated problems, and they differ in key aspects. Depending on the problem and the preferred implementation as well as the existence of certain capabilities, different metaheuristics can be chosen. A common factor to divide metaheuristics by is if they work on a single solution or if they are population-based. Examples for single-solution metaheuristics are tabu search [12], in which a set of previous configuration is stored in a tabu list which cannot be returned to, or simulated annealing [18], which will allow a neighbourhood move to a neighbour with a higher objective value than the original with a certain probability in order to escape local optima. Examples for population-based metaheuristics are genetic algorithms, which work with a set of individuals which change via mutations and recombination or ant colony optimisation [9], in which individuals lay pheromone trails proportional to their fitness and to which other individuals feel drawn to. For more information on metaheuristics, see [3].

Problem Formalization

In this chapter, we explain and define our problem formally. First, inputs will be introduced, followed by hard and soft constraints. The objective function will then be described, and the chapter will be closed by showing that the problem is in the complexity class of NP-hard problems.

4.1 Input

Employees   A group of people working in the customer service, which is our main resource to be scheduled.

$W = \{w_1, ..., w_{n_w}\}$

Days   Employees have to be scheduled for every day in the considered time period, which is a month.

$D = \{d_1, ..., d_{n_d}\}$

Holidays are also of special interest because the payment is different. Working on a holiday is paid twice the usual amount.

$D^H \subseteq D$

Houses   There are several museums that need regular staffing. They require a varying amount of employees every day.

$H = \{H_1, ..., H_{n_h}\}$

Events   Events are special occurrences which are normally one-time only (although they can repeat) and can happen at any day, at any time and at any place. They can

also take a varying amount of time. Both special exhibits, which can last for months, and the closing of a hall for repairs for a few hours can be such events.

$$E = \{E_1, ..., E_{n_\mathrm{p}}\}$$

Additionally, there is a special type of event, for which employees register themselves separately. These events are generally out of expected time-boundaries, often late at night. When applying for general shifts, it is generally understood that they are in a certain range of time, mostly between 8 o'clock and 20 o'clock, and normally last a full workday. Therefore, when there are events that defy this understanding, they are published separately and employees specifically sign up for each special event they would like to work at.

$$E_s \subseteq E$$

Favourite Houses   Every employee can have zero or more favourite houses. Those are the places they would like to be stationed at the most. Choosing no favourite house is internally the same as choosing every house, since both choices essentially mean that there is no preference. These nominations have the character of a wish, not a need, and will be adhered to if possible. It is not possible to favourite events. This would not be useful anyway, since every application towards an event is only valid for this event.

$$H_w \subseteq H$$

Locations   A location is the logical unit an employee can be assigned to. It consists of houses, events and the special virtual units standby and reserve. Reserve and standby will be converted to a real house or event prior to taking place, and are only for organizational purposes. Internally, they are used for redundancy reasons. Employees assigned to the reserve will definitely have a shift to work at, but will not know where until the same day. Employees on standby might be called during the morning if there is an unexpected shortage for this day.

$$L = \{Standby, Reserve\} \cup E \cup H$$

There are additional constants that define how many standby and reserve shifts employees may have each month.

$C_\mathrm{sta} = 1$ — formerly, this was defined as $C_\mathrm{sta} = \lfloor \frac{1}{4} \cdot N_w + 0.5 \rfloor$ - where N_w is the number of shifts an employee requests for themselves—but the current distribution of employees to shifts allows an easier formulation, simply limiting every employee to one of these shifts.

$C_\mathrm{res} = 1$

$C_\mathrm{minShift} = 2$ — the minimum number of shifts an employee has to choose before being eligible for reserve or standby shifts.

Shifts The central unit of work is a shift. Shifts are defined by day and location. Therefore, only one shift exists per day and location. Shift duration is measured in hours.

$$S \subseteq D \times L$$

Requirements For every shift, there is a requirement for employees.

$$R_s \in \mathbb{N}_0 \qquad \forall s \in S$$

Time of shifts Every shift has a start and an end time. The duration of the shift can be computed from those times and is expressed in hours.

$$Ts_s \in [0, 24]$$

$$Te_s \in [0, 24]$$

Shifts are generally required to be possible, starting before they end.

$$Ts_s < Te_s$$

Furthermore, the duration of a shift is defined as the time between start of a shift and end of a shift.

$$\Delta_s = Te_s - Ts_s$$

Number of Shifts Every employee may request anything from one shift upwards. The actual number of shifts assigned to a person is capped by the maximum amount of money they may earn each month. This is based on a regulation in Austria. Special event shifts E_s for which an employee signed up separately are not affected by this number.

$$N_w \in \mathbb{N}$$

$C_{\text{maxWork}} \in \mathbb{N}$ — this constant defines how many hours a month a person may work while adhering to the governmental regulation

Chosen Shifts Every employee may choose any shift where they have time to work. From the chosen shifts, up to N_w will be assigned to the employee, while making sure that C_{maxWork} is not exceeded. Usually, an employee will select more shifts than the number of desired shifts.

$$S_w \subseteq S \qquad \forall w \in W$$

4.2 Solution

A solution describes when and where which employee will work at a shift, and is modelled as a set of assignments A, one for each employee.

Assignment An assignment consists of a set of shifts which have been assigned to a certain employee.

$$A_w \subseteq S_w \qquad \forall w \in W$$

4.3 Constraints

Generally, there are two types of constraints—hard constraints and soft constraints. If any hard constraint is violated, the solution is infeasible. This means that any solution must satisfy every hard constraint for every part of the solution. Soft constraints are used to give an indication of quality to the solution. The amount of soft constraint violations will be used to compute the overall fitness of a solution.

Hard Constraints:

1. Everybody has to work at least one shift.

   $$C_1^h = |A_w| \geq 1 \qquad \forall w \in W$$

2. Nobody may do more shifts than requested—except special shifts, for which people register separately.

   $$C_2^h = |\{(d,l) \in A_w | l \notin E_s\}| \leq N_w$$

3. Nobody may work for longer than the regulation in Austria permits. Working on holidays yields twice the payment, so every hour working on an holiday is worth double the time.

   $$C_3^h = \sum_{(d,l) \in A_w} ((|\{d\} \cap D^H| + 1) \cdot \Delta_{(d,l)}) \leq C_{\text{maxWork}} \qquad \forall w \in W$$

4. Requirements cannot be exceeded.

   $$C_4^h = |\{w \in W | s \in A_w\}| \leq R_s \qquad \forall s \in S$$

5. Employees with fewer than C_{minShift} chosen shifts cannot be assigned a reserve or standby shift.

   $$C_5^h = (d,l) \in A_w : l \notin Res, Sta \qquad \forall w \in W : |S_w| < C_{\text{minShift}}$$

6. No employee may have more than C_{res} Reserve shifts.

   $$C_6^h = |\{(d,l) \in A_w | l \in Res\}| \leq C_{\text{res}} \qquad \forall w \in W$$

7. No employee may have more than C_{sta} standby shifts.

   $$C_7^h = |\{(d,l) \in A_w | l \in Sta\}| \leq C_{\text{sta}} \qquad \forall w \in W$$

8. No employee may have two assigned shifts that intersect each other.

   $$C_8^h = \forall d \in D, \forall l_1, l_2 \in L, l_1 \neq l_2 : (d, l_1) \in A_w \rightarrow \neg \exists (d, l_2) \in A_w \wedge (Ts_{(d,l_1)} \leq Ts_{(d,l_2)} < Te_{(d,l_1)}) \qquad \forall w \in W$$

9. There may be no unassigned Reserve or Standby shift.

$$C_9^h = |\{w \in W | (d,l) \in A_w\}| = R_{(d,l)} \qquad \forall (d,l) \in S | l \in \{Sta, Res\}$$

Soft Constraints The soft constraints of this problem are split between two fairly conventional constraints, C_1^s and C_4^s and two rather unique constraints concerning fairness, C_2^s and C_3^s. C_1^s is a typical constraint which drives the main objective, assigning shifts to employees, while C_4^s minimises employee assignments to houses that are no favourite of theirs. C_2^s and C_3^s are fairness constraints, which means that they try to balance the number and type of shifts each employee is assigned to in relation to the global utilization. C_1^s and C_4^s are modelled using the mean squared error (MSE), while C_2^s and C_3^s are modelled using variance.

1. There should not be any unassigned (open) shifts. If that is not possible, balance the shifts between all open possibilities so that many shifts will have small deficiencies and no shift will have an unreasonably large number of missing assigned employees.

$$C_1^s(A) = \frac{1}{|S|} \sum_{s \in S} (1 - \frac{|\{w \in W | s \in A_w\}|}{R_s})^2$$

2. The rate of employment should be fair—the ratio of assigned shifts to ideally wanted shifts should be as constant as possible for all employees. The constraint is expressed in hours instead of shifts since some shifts only take a few hours, while others are a full workday. Employees generally assume to be given a full day when applying for shifts, so if someone gets assigned five shifts of three hours each, that is a big difference from someone getting assigned five shifts eight hours each.

$$D_{max} = \max_{s \in S} \Delta_s$$

$$\bar{x} = \frac{1}{|W|} \sum_{w \in W} \frac{\sum\limits_{s \in A_w} \Delta_s}{N_w \cdot D_{max}}$$

$$C_2^s(A) = \frac{1}{|W|} \sum_{w \in W} (\frac{\sum\limits_{s \in A_w} \Delta_s}{N_w \cdot D_{max}} - \bar{x})^2$$

3. The reserve and standby shifts should be split fairly between the eligible employees. The relevant variables for this constraint are the hours spent doing reserve and standby shifts versus all the hours assigned to a person.

$$\bar{x} = \frac{1}{|W|} \sum_{w \in W} \frac{\sum\limits_{d \in D} \sum\limits_{l \in \{Sta, Res\}} \sum\limits_{s \in A_w \cap \{(d,l)\}} \Delta_s}{\sum\limits_{s \in A_w} \Delta_s}$$

$$C_3^s(A) = \frac{1}{|W|} \sum_{w \in W} (\frac{1}{\sum\limits_{s \in A_w} \Delta_s} (\sum\limits_{d \in D} \sum\limits_{l \in \{Sta, Res\}} \sum\limits_{s \in A_w \cap \{(d,l)\}} \Delta_s) - \bar{x})^2$$

4. If at all possible, employees should not work in any house that is not a favourite of theirs, and if they have to work this way, the misses should be balanced between employees as well as possible.

$$C_4^s(A) = \frac{1}{|W|} \sum_{w \in W} (\frac{|\{(d,l) \in A_w | l \notin H \backslash H_w\}|}{|A_w|})^2$$

4.4 Example Assignment

Following the definitions, Figure 4.1 shows a small example of a CESP instance as a graph. Chosen shifts can be seen as black lines between employees w and shifts d, l, favoured shifts as green lines. Assignments are bold lines, while the red lines denote constraint C_7^h, the inability to be assigned two shifts on the same day. Other constraints have been kept out of the representation for greater clarity.

The solution in the Figure is $A = \{\{(d_1, l_1)\}, \{(d_1, l_1), (d_2, l_2)\}, \{(d_2, l_1)\}\}$. All requirements are satisfied, and favoured shifts are respected as much as possible, although the only real choice was which shift w_2 and w_3 would be assigned on d_2.

Employee	Days	FavLoc
w_1	d_1	l_1
w_2	d_1, d_2	l_2
w_3	d_2	-

Requirements	d_1	d_2
l_1	2	1
l_2	0	1
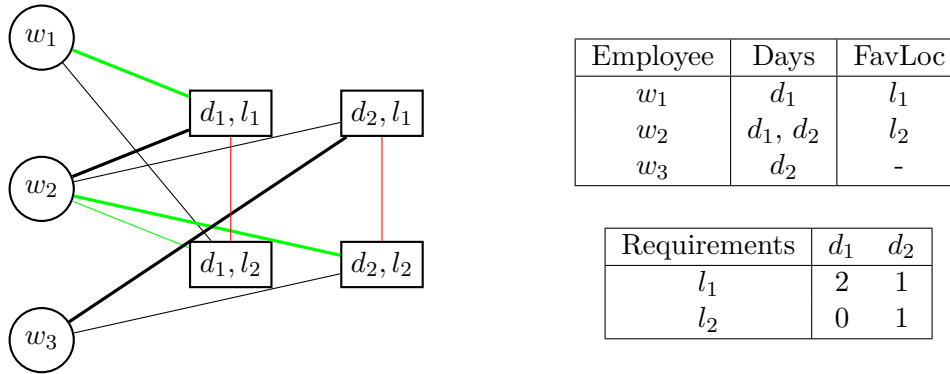
Figure 4.1: Sample CESP instance in graph representation. Green lines point to a favourite location, red lines are shifts that cannot be assigned to the same person, bold lines are the actual assignments.

4.5 Objective Function

The fitness of our solution will be determined by the objective function. It is implemented as a weighted sum approach, whereby each of the soft constraint computation functions are multiplied by a custom weight and then summed up to generate a single value. As stated before, the goal for this problem is minimization, so the smaller the value, the better. The objective function f for an assignment A thus looks like the following:

$$f(A) = \lambda_1^s C_1^s(A) + \lambda_2^s C_2^s(A) + \lambda_3^s C_3^s(A) + \lambda_4^s C_4^s(A)$$

Finding the correct weights λ^s for any of the constraints is difficult work at best, since it is directly shaping the solution. If the weights are incorrect, the solution will feel bad or skewed. This work can only be done right by domain experts, and even then it is an approximation. For this problem, we focused on C_1^s — trying to fulfil all shift requests. This is the primary goal of the algorithm after all, and the chances for anything to disrupt this goal should be slim. The fairness constraints have been left at a default

— they generally balance themselves out quite well and can be used with any weight $\lambda^s > 0$ to ensure that if a fairer way to schedule a shift without losing the assignment or violating a favourite house constraint exists, it will be taken, which is exactly how they are intended. The favourite houses constraint is granted a little extra weight. This means that, in accordance with its quadratic nature based on each employee, solutions which are ignoring the preferences of an employee too much will not be considered unless they offer tremendous upsides.

Therefore, the final weights to be used to guide the algorithm have been determined to be as follows:

$\lambda_1^s = 10$, $\lambda_2^s = 1$, $\lambda_3^s = 1$, $\lambda_4^s = 2$

4.6 NP-Hardness

There are different complexity classes for problems, and defining which class a problem belongs to is important for choosing approaches to solve the problem. Defining a problem as NP-hard is possible by performing a reduction from any NP-complete [17] problem to this problem. To prove that this is the case, following is a reduction from the NP-complete [11] bin-packing problem.

Proposition 1 The Casual Employee Scheduling Problem as presented in this chapter is NP-hard.

Proof Recall that the Bin-Packing problem asks if a set of items of different sizes can be packed in a defined number of bins with a uniform capacity.

Let j items $a_1, ..., a_j$ with size $s_1 > 0, ..., s_j > 0$ as well as a bin capacity B and a number k of bins $b_1, ..., b_k$ be given. Let $c(a, b) = 1$ if an item is placed in a certain bin and 0 otherwise. The decision version of the bin-packing problem can then be defined as such:

$$\sum_{l=1}^{k} c(a_i, b_l) = 1 \qquad \forall i \in \{1, ..., j\}$$
$$\sum_{i=1}^{j} c(a_i, b_l)s_i \leq B \qquad \forall l \in \{1, ..., k\}$$

Informally, each item must be placed in exactly one bin and the capacity of no bin may be exceeded.

This can be transformed into a CESP-instance in the following way: The number of days D is equal to the number of items j. Each bin b is an employee with $C_{\text{maxWork}} = B$, who has chosen every shift. The amount of shifts for each employee is $N_w = j$. Each item a_i is a house H_i which requires exactly one open shift at day i with a duration of s_i. There are no standby or reserve shifts.

Since it is not possible to set constraint weights, all constraints need to be accounted for. C_4^s can be ignored if every employee favours every house, since then there will never be

a penalty. C_3^s can similarly be ignored since there are no standby or reserve shifts. C_2^s can potentially introduce problems, since the growth rate is different from the growth rate of C_1^s. This can also be addressed in a simple matter. Simply set N_w to a high number approaching infinity. Any value of $N_w > j$ is irrelevant for the conclusion of the CESP, and setting it high enough basically ensures that $\bar{x}$, being dependent on N_w in the denominator, approaches zero. This will then be subtracted from essentially the same number, also having N_w in the denominator, before being squared, summed up $|W|$ times and being divided by $|W|$. Meanwhile, each unassigned item is a whole house worth of non-assignments, since $R_s = 1$ and each house consisting of only one shift. Therefore, each unassigned item adds $\frac{1}{|S|}\lambda_s^1$ to the objective function, which is surely higher than the result of of C_2^s.
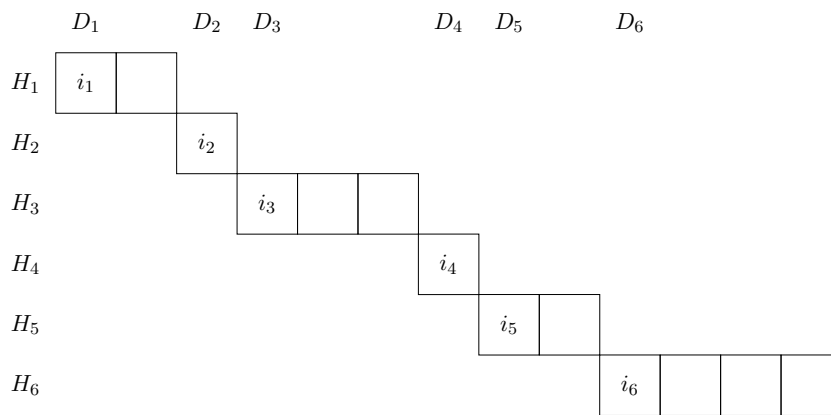
Figure 4.2: Sample set of items from a bin-packing problem transformed to CESP houses with open shifts of certain length
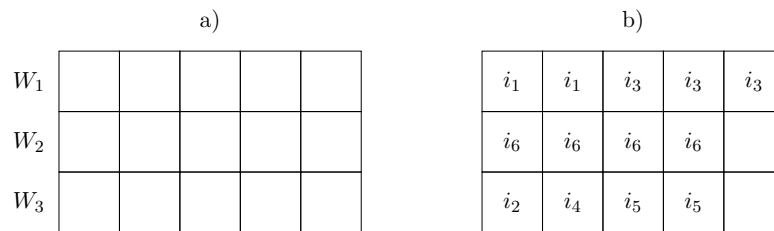
Figure 4.3: Sample set of employees, each block corresponding to a shift length of one, before (a) and after (b) being assigned shifts from the list of Figure 4.2.

As a simple example of this, imagine $k = 3$ bins of size $B = 5$. Further, let there be $j = 6$ items with sizes (2,1,3,1,2,4) respectively. The resulting CESP instance is portrayed in Figures 4.2 and 4.3. Figure 4.2 shows the resulting houses and days that are created from the items, with their respective shift lengths. Figure 4.3 first shows the number of employees with their potential capacity $C_{\mathrm{maxWork}} = 5$, whereby each block denotes one possible shift size. Then, a possible solution with the items / shifts from Figure 4.3 is

shown. In terms of the CESP, this reads as such that employee W_1 has shifts at H_1 at D_1 and H_3 at D_3. Therefore, the problem is solved.

This transformation clearly does not exceed polynomial time constraints. When solving the problem, employees will be assigned shifts. The amount of shifts assigned will be maximised by C_1^s, with the goal of not leaving any shift open. The shifts assigned to an employee will never exceed C_{maxWork}, so the capacity constraint of bin-packing is fulfilled. If every shift requirement can be assigned to an employee, the value of the constraint C_1^s will be 0, and the bins can be packed that way. If the result is not 0, the items do not fit in the bins. The resulting set of assignments will include the way the bins are packed.

To show that the hard constraints are not violated by this transformation, they will be explained in order.

C_1^h can probably introduce problems, if there are less items than bins. This can be circumvented by simply removing any excess bins until there is an item for every bin before reducing to CESP.

C_2^h is easily fulfilled by the transformation setting it N_w to the maximum number of items / houses.

C_3^h is the pivot on which the whole transformation rests and it being fulfilled is paramount to the reduction.

C_4^h is also easy to ensure and important to fulfil.

C_5^h to C_7^h and C_9^h are generally irrelevant since there are no standby or reserve shifts, or already explained during C_1^h. C_8^h is impossible to violate in this instance, since there is only one shift every day. Therefore, all constraints are fulfilled by the transformation.

Solution Approaches

As mentioned before, the problem was considered from both an exact and a metaheuristic perspective, and algorithms for both approaches have been implemented. These were then combined into a hybrid approach, which is the version that is used for the computational study and real-life application. Note that the metaheuristic approach alone cannot be utilised to solve the problem because the chosen metaheuristic, the VNS, is an improvement heuristic and therefore needs an initial solution to start with.

5.1 Constraint Programming Approach

The exact approach to the problem was implemented with MiniZinc [24], a constraint programming wrapper for a language called FlatZinc. FlatZinc in turn utilises different CP-solvers like Gecode to actually solve the problems. The goal of the implementation was a faithful reproduction of the mathematical model presented in Chapter 4.

In general, the implementation of the model in MiniZinc was relatively straightforward. The most relevant distinction is probably the logical difference between sets and MiniZinc arrays, which resulted in an additional constraint in the MiniZinc model, since the subset function was not implemented at the time of implementation.

Recall that $A_w \subseteq S_w \quad \forall w \in W$ defines the set of assignments for employee w as a subset of the shifts w has chosen. Therefore, an assignment can never be a shift that was not chosen by w. In the MiniZinc model, this relation is modelled by the constraint C_0^{rh}:

```
constraint forall (w in Workers, l in Locations, d in Days)
        (( w in assign[l,d] −> w in worker_shifts[l,d]) );
```

Furthermore, while the mathematical model defines assignments as a set of shifts assigned to employees, the MiniZinc model is defined the other way around. There is an array of

shifts with a set of employees for each possible shift. This was done because it is easier to model and compute the MiniZinc model this way.

```
array[Locations,Days] of var set of Workers: assign;
```

Another important part of MiniZinc models is the specification of the actual objective the solver is supposed to accomplish. There are three parts that can be changed to influence the solver for this. First, all variables that are supposed to be computed by the solver are declared like this:

```
var int: solutionValue;
```

The *var* keyword tells the solver that this variable should be computed. There are, as discussed previously, two ways that a problem can be solved: optimisation or satisfaction. In satisfaction problems, the solver just searches for one variable combination that does not violate any constraints, whereas in optimisation problems, the solver tries to minimise a variable. In this problem, the corresponding line of code is this:

```
minimize (solutionValue);
```

Optionally, MiniZinc offers the functionality of telling the solver on how to achieve the stated goal in the most promising way. Ideally, all decision variables in a problem would be linked in a way that only a set few variables could be directly influenced by the solver. In our case, the *solutionValue* variable is defined as the sum of our soft constraints. The soft constraints meanwhile are functions of one variable, *assignments*. The only way to change *solutionValue*, therefore, is to change *assignments*. In some problems, that might be not as clear. Parts of the variable might also be more interesting than others. In our model, the following statement could be used to solve the problem:

```
solve ::seq_search([
set_search([assign[index_standby,d] | d in Days],first_fail,
    indomain,complete),
set_search([assign[l,d] | l in Locations, d in Days],first_fail
    ,indomain,complete)
])
```

While this is not the actual configuration of our problem, it serves as a good example. This statement tells the solver to solve sequentially for two sets—first for assignments for the standby shifts, and afterwards for all the remaining shifts. This might be a good idea because the standby shifts are part of a hard constraint that fails if a standby shift is not accounted for. The other interesting part here is the second parameter, $first\_fail$, which states that the variables assigned should be ordered by their domain size, ascending. This way, incompatible variable configurations will be more likely to fail early, which results in less overall runtime.

Another problem of the implementation were the backend solvers, since the problem utilises floating-point variables for soft constraints C_2^s and C_3^s. Most solvers do not

support floating-point variables, while others do not support other required functions. Others are not freely available for use in non-university environments, in which the solution will be used in reality. For those reason, the backend solver that was ultimately chosen was Gecode [1].

The best case would be to be able to solve all real-world instances to proven optimality. It was expected from the beginning that this was not very realistic because of the complexity of the problem and the instance sizes. This prediction came through, and while the creation of the first solution that MiniZinc provides is generally fast enough (see Chapter 6 for details), the solution itself is woefully unoptimised and further improvements are rare and not really relevant, taking hours or days to find a better solution that simply switches two assignments. In short, while satisfaction is generally fast, optimisation is very slow.

5.2   Metaheuristic approach

The main part of the implementation in this thesis is the metaheuristic approach. It is a GVNS as presented in Chapter 3 with two shaking neighbourhoods and five VND neighbourhoods. The VND-algorithm is described in Algorithm 8, while the complete GVNS does not differ in execution from the version in the prior chapter. Furthermore, an extension to the VND has been developed and implemented.

The main idea behind the extension is the observation that the neighbourhoods for this problem divide the solution space between them. Therefore, it would be reasonable that a neighbourhood would generate a lot of good improvements for a while, and then diminish very rapidly as other parts of the solution space become more interesting, thereafter generally only providing very little improvement while costing a lot of time. The extension of the VND tries to smooth over this behaviour by systematically shifting the VND neighbourhoods when they are not able to find new solutions. It does so by introducing an offset that gets incremented when a neighbourhood is not able to produce a better neighbour than the original in a certain amount of steps or time. The offset itself influences with which neighbourhood the VND will start the next time it finds an improvement via this computation: $i \leftarrow (k + \text{offset}) mod |N|$. In this version, i is used to refer to the current neighbourhood while k is the original neighbourhood. i is in in the domain of $|N|$, the number of neighbourhoods. For example, the VND generally starts with neighbourhood 1 after finding an improvement, but with an offset of 1, it would start with neighbourhood 2. At any given time, it will still travel through all neighbourhoods before deciding that there is no improvement to be made any more. The extension can be toggled by inputs, and if it is turned off, the algorithm works like a conventional VND. This extension is similar in spirit to hyper-heuristics guiding which heuristic should be used to improve a solution based on prior performance, but far more simplistic.

For increased efficiency, delta evaluation was implemented. After finding a neighbour, it is generally necessary to ensure that the new neighbour is feasible, and if it is, the objective value has to be calculated. Feasibility can be ensured by the neighbourhood

structures themselves, in that they do not allow infeasible moves in the first place or by checking every new neighbour after generation. This can be better or worse for performance depending on the application. In this algorithm, new neighbours are checked for feasibility after generation. Because of all this, the checking for general fitness of the new neighbour is a task that is done very often, and the runtime of the algorithm will depend greatly on how efficient it is implemented. The simple way is to check the whole solution after each move, while delta evaluation evaluates just the parts that have been changed during the neighbourhood move. This is equivalent to a runtime of $O(n_w)$ or $O(|S|)$, depending on the constraint, for checking the whole solution and a runtime of $O(1)$ for delta evaluation. It was implemented by each neighbourhood providing a list of suggested changes instead of modifying the solution directly. The list of changes and the solution are then used by the methods used for fitness checking to infer which parts of the solution should be evaluated again.

For most hard constraints, this is an easy task. If the changes propose only one employee to gain or lose assignments, it is only relevant to check if this employee now has potentially too many hours versus verifying the hours of every employee. For soft constraints, this is not as easy. While C_1^s and C_4^s are relatively straightforward, the variance of C_2^s and C_3^s as presented in Chapter 4 cannot be partially evaluated and was calculated using the Steiner translation theorem. After every successful move, e.g., after a neighbour has been accepted as the new current solution, an additional full feasibility check and objective value calculation is done for increased reliability and to reduce potential drift by floating point inaccuracies.

Neighbourhoods, as explained before, are the backbone of any kind of VNS. Since a GVNS was implemented as a solution approach for this problem, there are both shaking and VND neighbourhoods. These neighbourhoods are not the same, nor should they be. The idea behind the VNS is to provide diversification, while the idea of the VND is to provide intensification, so the neighbourhoods need to be different.

5.2.1 VND Neighbourhoods

The goal in creating the VND neighbourhoods for this problem is to partition the solution space. In many other problems, especially ones with less complex constraints and a clearer way to an obviously better solution (e.g. unmodified TSP), it is advisable to create neighbourhoods of different sizes for a VND, so that the small neighbourhoods can be exhausted before bigger jumps in the solution space are made. In this problem, the best objective value is a compromise.

As an example for the compromise, C_1^s and C_4^s can be fundamentally working against each other. If a shift is unassigned and could be assigned to an employee who has not favoured this shift, C_1^s would urge the solution towards making this move, while C_4^s would be violated. The weights of the constraints and the seriousness of the infraction would decide if this step were to be taken. To illustrate this point, there is the following example, as seen in Table 5.1.

Algorithm 8 Variable Neighbourhood Descent with neighbourhood shifting

---

Input: solution x
Input: set of neighbourhood structures N_k, for $N_1, ..., N_{k_{\max}}$
Output: improved solution w.r.t. set of neighbourhood structures N x

 1: **procedure** VND(x, N,shift)
 2:     offset $\leftarrow 0$
 3:     offsetWait $\leftarrow$ false
 4:     $k \leftarrow 1$
 5:     **repeat**
 6:         $i \leftarrow (k+\text{offset})\% k_{\max}$
 7:         $y \leftarrow$ best neighbour of $N_i(x)$
 8:         **if** $f(y) \leq f(x)$ **then**
 9:             $x \leftarrow y$
10:             $k \leftarrow 1$
11:             **if** offsetWait == true **then**
12:                 offset $\leftarrow$ offset $+1$
13:                 offsetWait $\leftarrow$ false
14:             **end if**
15:         **else**
16:             **if** shift == true $\wedge$ shift criteria met **then**
17:                 offsetWait $\leftarrow$ true
18:             **end if**
19:             $k \leftarrow k + 1$
20:         **end if**
21:     **until** $k > k_{\max}$
22:     **return** x
23: **end procedure**

Table 5.1: Example of soft constraint effects on sample solution, original state

w	N_w	H_w	S_w
1	1	{1}	{1}
2	1	{1}	{1}
3	1	{1}	{1}
4	1	{}	{1}
5	1	{}	{1}

(l, d)		1	
		R	A
1	5	{1, 2, 3}	

C_s	Weight	Value
1	1	4
2	0	0
3	0	0
4	1	0
Total		4

In this example, there is one shift, S_1, with a requirement of 5 employees. Three employees are already assigned to the shift, and all of them have the shift in their favourite houses. There are two other employees, who are not yet assigned to the shift, but could potentially be. Those employees do not have the shift favoured. In this example, the constraint weights are purposely simple, with C_2^s and C_3^s being completely left out. In the initial case, our objective function is initially 4. As a next step, the algorithm adds another

employee to the shift, depicted in Table 5.2.

Table 5.2: Soft constraint example, additional employee assigned

(l,d)		1
	R	A
1	5	$\{1,2,3,4\}$

C^s	Weight	Value
1	1	1
2	0	0
3	0	0
4	1	1
Total		2

W_4 is assigned to the shift, and therefore the constraints change their value. C_1^s decreases from 4 to 1, while C_4^s increases to 1. All in all, the assignment decreases our objective value by 2. Finally, the algorithm assigns the last employee W_5 to the shift. This is shown in Table 5.3.

Table 5.3: Soft constraint example, all employees assigned

(l,d)		1
	R	A
1	5	$\{1,2,3,4,5\}$

C^s	Weight	Value
1	1	0
2	0	0
3	0	0
4	1	4
Total		4

It is easily observable that the objective value is now back to 4, and that this is a worse solution for these constraint weights. If $\lambda_1^s = 5$, that is, the weight of C_1^s was much higher, this would be the preferable case. This example should showcase how the different constraints value different outcomes for the solution and how the different neighbourhoods that will be explained in the following sections try to address those demands.

In the following, each neighbourhood will be introduced in turn. Every neighbourhood influences soft constraints C_2^s and C_3^s, although it is not especially obvious at first glance. Both C_2^s and C_3^s scale with the hours worked by an employee and each neighbourhood either influences the assignments of an employee, of a shift, or both. In the case of changing an employee, it can change the hours an employee works. In the case of changing shift, it can change which shift is worked, which in turn can change the hours of the employees working the shift. Therefore, no meaningful change in assignments can exist that could not influence C_2^s or C_3^s.

MoveShift

This neighbourhood move changes the location of an assignment, while keeping the same day as the original. This is intended to move employees from a saturated shift to a shift that needs more people. Furthermore, it can also resettle an employee to a shift that is

at one of their favourite houses. All soft constraints are impacted by this neighbourhood. It is not possible to move away from a standby or reserve shift with this neighbourhood, since then it would not be fully staffed and this would violate hard constraint C_9^h. Figure 5.1 shows a working example of the neighbourhood.

(a) original

(b) after move shift
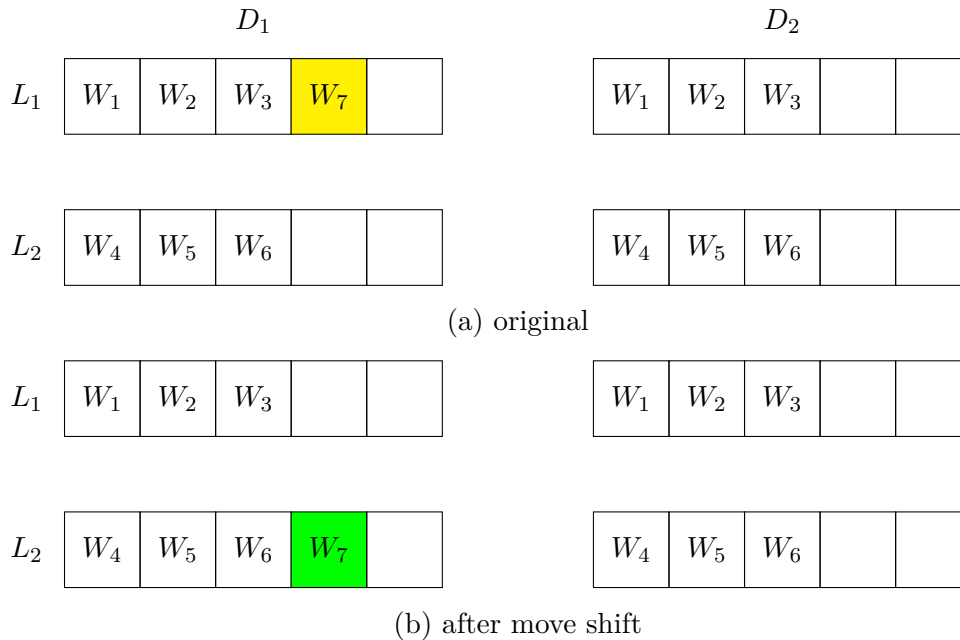
Figure 5.1: Move Shift

AddShift

This is a very simple neighbourhood. It simply tries to add another assignment to the solution. It is the only way to change the amount of assignments present in a solution. There is no way to delete assignments within the VND, since the central problem to be solved by this algorithm is the assignment of employees to shifts and the VND is responsible for intensification. The neighbourhood influences all soft constraints. As before, C_3^s cannot be changed by additional shifts, since it has to be completely filled by hard constraints. Figure 5.2 illustrate the workings of the neighbourhood.

ReassignShift

This neighbourhood move reassigns a shift to a different employee. It does so by deassigning the original employee and then assigning a different employee to the same shift. Figure 5.3 illustrates the concept. It is used to assign a shift to someone who would fit it better than the original employee. This can be because the other person has fewer assignments than should be—which is the case for C_2^s and C_3^s or because the original

(a) original

(b) after add shift
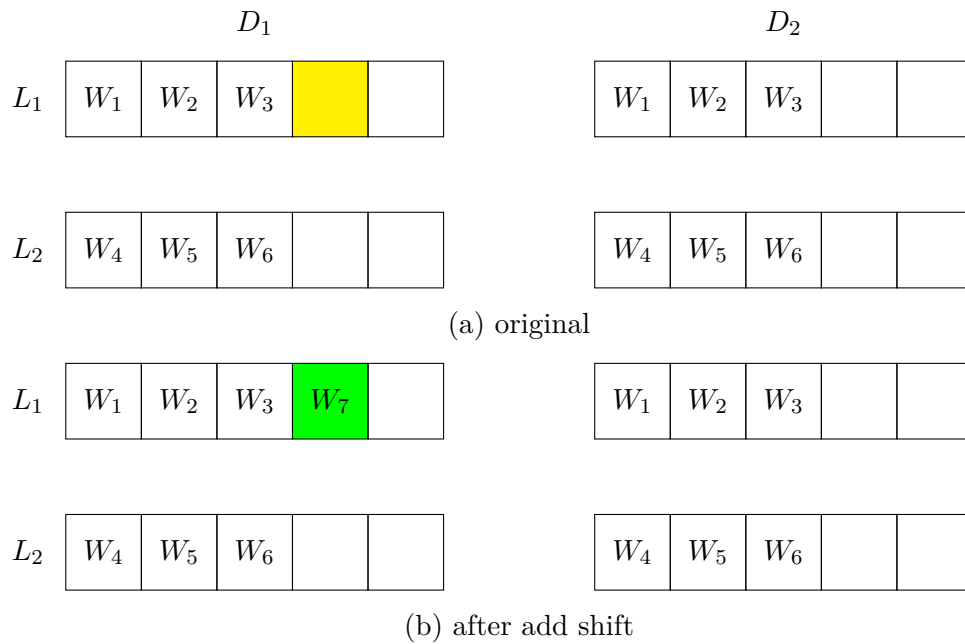
Figure 5.2: AddShift

assignee has not favoured a shift whilst the replacement employee has, which is the case for C_4^s. C_1^s is not affected, since no change in shifts is done.

ReassignToDifferentDayShift

This neighbourhood move changes both the location and the day of an assignment while keeping the employee. It thereby changes the whole shift. This is one of two neighbourhoods that can change the day on which an employee is scheduled to work after an initial assignment is set. The neighbourhood is relevant to all soft constraints. The algorithm is described in Figure 5.4. Having a neighbourhood that only changes the day while keeping the location has been considered, but the idea has been ultimately discarded. The reasoning for this is twofold. First, locations are different from each other (some are events, some are in favourite houses, some are not) while days are not. Any day that an employee is willing to work is fundamentally the same for the employee. Second, it would severely restrict the neighbourhood for some cases. For example, there are at times event shifts that only need staffing for a single day, in which the originally proposed neighbourhood would not be able to do anything.

SwapShift

This kind of neighbourhood is very common in many problems, it simply swaps two assignments. This is the biggest neighbourhood in the VND and while it generally does not change the objective value too much by itself, it keeps the balance and helps with the

D_1 D_2

L_1 | W_1 | W_2 | W_3 | W_7 | | | W_1 | W_2 | W_3

L_2 | W_4 | W_5 | W_6 | | | W_4 | W_5 | W_6

(a) original

L_1 | W_1 | W_2 | W_3 | W_8 | | | W_1 | W_2 | W_3

L_2 | W_4 | W_5 | W_6 | | | W_4 | W_5 | W_6

(b) after reassign shift
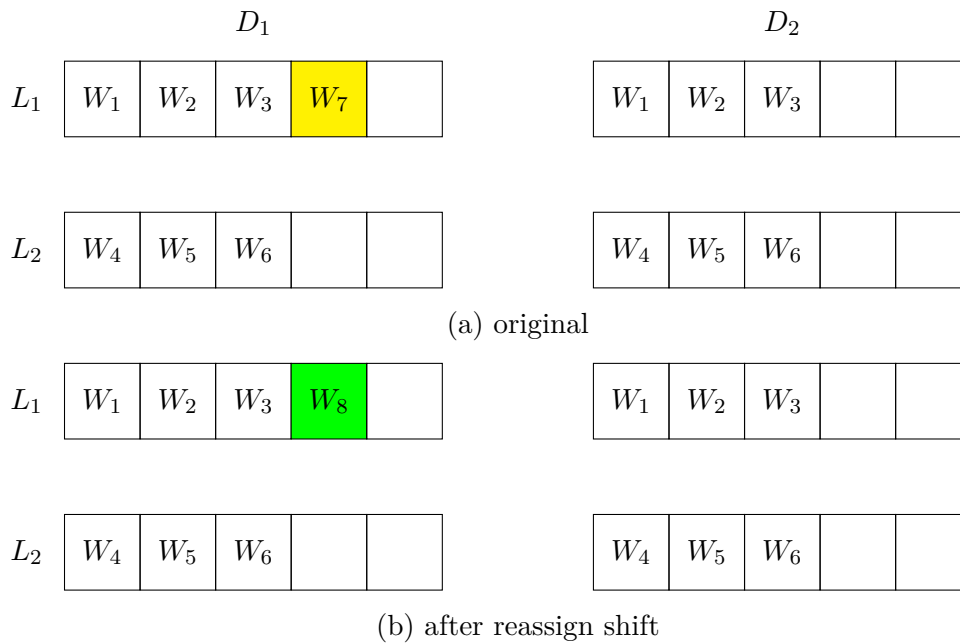
Figure 5.3: ReassignShift

D_1 D_2

L_1 | W_1 | W_2 | W_3 | W_7 | | | W_1 | W_2 | W_3

L_2 | W_4 | W_5 | W_6 | | | W_4 | W_5 | W_6

(a) original

L_1 | W_1 | W_2 | W_3 | | | W_1 | W_2 | W_3

L_2 | W_4 | W_5 | W_6 | | | W_4 | W_5 | W_6 | W_7

(b) after ReassignToDifferentDayShift shift
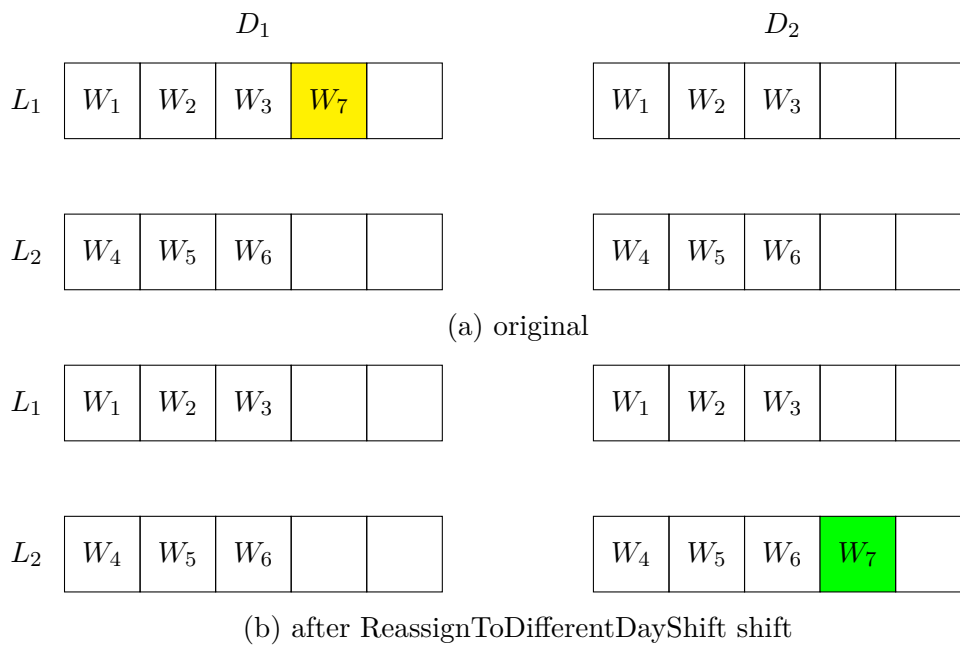
Figure 5.4: ReassignToDifferentDayShift

fairness constraints. Furthermore, it often allows to reach different parts of the solution

space due to its size. Informally, the swap shift is also an easy human behaviour to emulate. The algorithm is detailed in Figure 5.5. The affected constraints are C_2^s, C_3^s, C_4^s, since everybody keeps the same amount of shifts and the shifts do not get changed either.
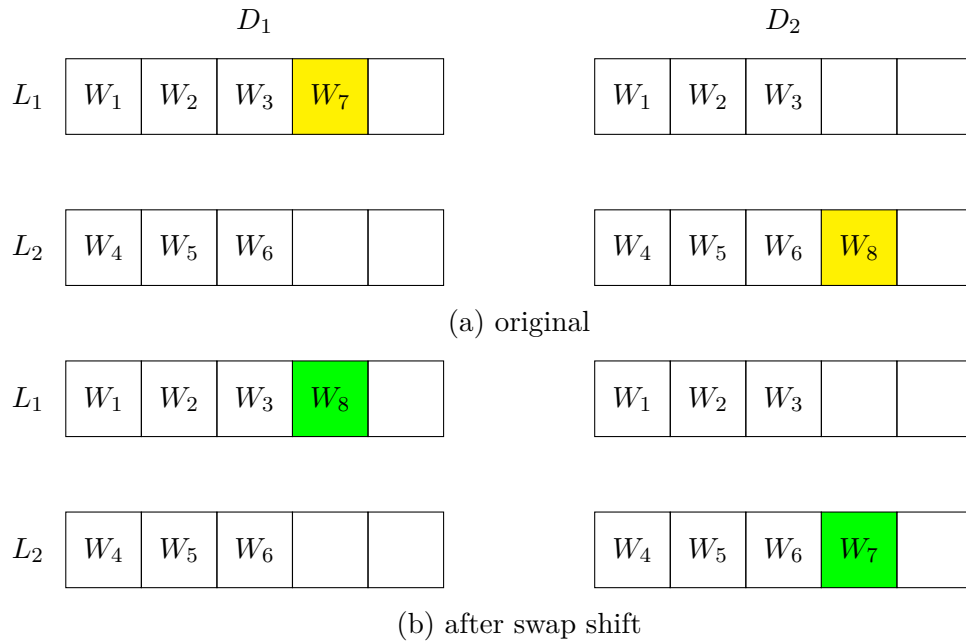
Figure 5.5: SwapShift

5.2.2 Shaking Neighbourhoods

Neighbourhoods of a VNS are generally designed to add diversification capabilities to a solution, i.e. to enable shaking. In this VNS, two neighbourhood types have been included, one being a very simple destruction neighbourhood and one being a very domain-specific shaking mechanism. It is not enough to simply disturb the original solution if it is done deterministically (e.g. by destroying the five worst assignments), chances are that the subsequent recreation might just reassign them in the same way and the solution would still be in a local minimum. Therefore, these neighbourhoods generally operate in a random fashion and will perform their move even if the result is worse than before, which it normally is.

In contrast to the VND neighbourhoods, the shaking neighbourhoods in this problem can be parametrised. That is, the amount of shaking they perform can be defined, and they can also be employed multiple times.

RemoveShifts

This is a basic neighbourhood that is very common amongst any kind of perturbation scheme. It simply removes a number of randomly chosen assignments from the solution. The only restriction implemented in the neighbourhood is in form of the hard constraints— the last shift of an employee cannot be removed and neither can any standby or reserve shifts. Improvements from the literature have also been investigated. One improvement [30] works by destroying a number of the worst rosters together with random rosters. This keeps the random aspect while also providing some disturbance in areas that obviously need the improvement. Destroying the whole roster of a person in a general NRP is more valuable than in this problem, though, since the days are interconnected in the NRP and only destroying part of the schedule of a single employee might not be enough to reassign different shifts. In the CESP, this does not apply, so the destruction of a whole roster does not seem to be that valuable when compared to simply deleting random assignments. A bad roster with regards to the CESP could be defined as a roster with particularly high penalty scores for the soft constraints, meaning either over/underdeployment or deployment to the wrong shifts. Destroying the worst rosters as such is probably difficult, but an easy way to implement something like that would be view every assignment that violates C_4^s as a bad assignment, and to dedicate a number of deletions to those assignments.

SwapReserveAndStandbyShifts

This is an unusual neighbourhood for a VNS and very domain-specific. Furthermore, it is dedicated to an area that the previous neighbourhood cannot touch. This neighbourhood swaps a number of standby and reverse shifts with regular shifts, like the swap neighbourhood. Since standby and reserve shifts are so highly constrained that they cannot be unassigned by the RemoveShifts neighbourhood explained previously, another method of shaking those shifts needed to be found. This neighbourhood achieves part of that idea. It is basically the same procedure as the swap neighbourhood done in sequence, only choosing the affected assignments randomly.

Summary

The following Table 5.4 sums up the presented neighbourhoods as well as some relevant attributes of them. The values used in the table are defined beforehand and are based upon the definitions in Chapter 4.

$$n_l = |L| \tag{5.1}$$

$$n_w = |W| \tag{5.2}$$

$$n_s = |S| \tag{5.3}$$

$$n_a = \sum_{w \in W} |A_w| \tag{5.4}$$

49

As we know that $S = (D \times L)$, it is clear that $n_l \leq n_s$. Just from the variable definitions, n_a cannot be set in relation to n_s or n_w, but when including the constraints, particularly C_1^h, which states that each employee has to work at least one shift, the lower bound of n_a can be set as n_w, so $n_w \leq n_a$. In real applications, n_a is the variable to be maximised and should ideally be:

$$|A_w| = \sum_{s \in S} R_s \qquad \forall (w \in W) \tag{5.5}$$

which will result in $n_s < n_a$ for most real-world applications, but depending on R_s, it does not have to be. If there are many shifts with zero requirements and the rest of the shifts also require a low number of employees, it is possible that $n_s > n_a$.

Table 5.4: Overview over neighbourhood move properties

Move Name	$O(\cdot)$	Affected C^s	Summary
MoveShift	$n_a \cdot n_l$	$C_1^s, C_2^s, C_3^s, C_4^s$	Changes location of assignment
AddShift	$n_s \cdot n_w$	$C_1^s, C_2^s, C_3^s, C_4^s$	Adds another assignment
ReassignShift	$n_a \cdot n_w$	C_2^s, C_3^s, C_4^s	Changes employee of assignment
ReassignDayShift	$n_a \cdot n_s$	$C_1^s, C_2^s, C_3^s, C_4^s$	Changes shift of assignment
SwapShift	$n_a \cdot n_a$	C_2^s, C_3^s, C_4^s	Swaps assignments between employees

5.2.3   Hybrid Approach

Due to the GVNS being an improvement metaheuristic, it always needs an initial solution which it cannot generate itself. This initial solution can be provided by MiniZinc, but a custom construction heuristic has also been implemented for this purpose.

Generation

This problem, like many other problems, needs an initial solution, which is different from just starting with an empty schedule and adding assignments. This is because there are a few hard constraints that would be violated by an empty schedule. Therefore, the task of creating an initial feasible solution and the task of improving this solution is not the same. While it is theoretically possible to just create a perfect initial solution, this is, as discussed earlier, not realistic. The most realistic way of arriving at a good solution in a reasonable time-frame for this problem is therefore to construct an initial, feasible solution by one means and then to try to improve it further.

In addition to generating initial solutions with MiniZinc, a simple greedy construction heuristic was implemented and integrated in the algorithm. It features random initial solutions to help with diversification, since the VNS is a single-solution metaheuristic. In general, it is a simple greedy routine with some domain-specific improvements. The results obtained by using an initial solution created by this heuristic are comparable to starting with a MiniZinc solution in terms of quality. There is the additional benefit of

using far less computation time and space than MiniZinc, generally requiring only a few seconds. Furthermore, the generated solutions have a vastly reduced runtime because the greedy heuristic takes care of a greater amount of trivial assignment choices than the MiniZinc solver and includes some domain-specific improvements. For more detailed results, refer to Chapter 6.

An interesting observation of the workings of MiniZinc and the greedy construction heuristic are that the MiniZinc initial solution is adhering to all constraints introduced in Chapter 4, while the greedy heuristic only tries to assign the maximum amount of shifts, preferably while adhering to the favourite house preferences set by the employees. While the greedy heuristic does not explicitly check if all constraints are considered, the resulting initial solution is checked after completion, and the heuristic automatically restarts if invalid. Due to its randomness, the greedy heuristic typically does not exceed more than five restarts while often needing none before being able to generate an initial solution.

The algorithm of the greedy construction heuristic is relatively simple. Algorithm 9 illustrates the concept. First, two sorted lists are generated: employees by number of days they applied for shifts and shifts by number of employees that are applicable for the shift. Both lists are sorted ascending, so that the most constrained variables will be used first. The heuristic works by iterating through those lists, starting by the most constrained shifts and employees. This is supposed to mimic the behaviour of constraint programming (and general logic) in choosing the least open variable first. Then, in this order, event shifts, standby and reverse shifts and general shifts are assigned. The event shifts are assigned first because they are generally the most constrained and can easily be overwritten by a relatively unimportant normal shift for which many alternatives exist. This is done deterministically and illustrated in Algorithm 10. Furthermore, they do not consume any of the number of shifts an employee wants to work and are seldom so numerous as to outright deny other constraints. Afterwards, standby and reserve shifts are assigned randomly. Finally, the remaining shifts are filled semi-randomly, choosing a random candidate when the sorting algorithm is tied for any spot, described in Algorithm 11.

Algorithm 9 Greedy construction heuristic main body

Input: solution with no assignments x
Output: initial solution x
 1: **procedure** GENERATEINITIALSOLUTION(x)
 2:     $e \leftarrow$ sortEmployees(x)
 3:     $s \leftarrow$ sortShifts(x)
 4:     $x \leftarrow$ assignEvents(x,getEventShifts(s),e)
 5:     $x \leftarrow$ randomAssignStandbyAndReserve(x)
 6:     $x \leftarrow$ assignShifts(x,getRemainingShifts(s),e)
 7:     **return** x
 8: **end procedure**

Algorithm 10 assignEvents

Input: solution x
Input: set of sorted shifts to be assigned s
Input: sorted list of employees e
Output: solution x
 1: **procedure** ASSIGNEVENTS(x,s,e)
 2:     update $\leftarrow$ true
 3:     **while** update == true **do**
 4:         update $\leftarrow$ false
 5:         **while** update == false **do**
 6:             $e' \leftarrow$ nextEmployee(e)
 7:             **while** update == false $\wedge s' \neq NULL$ **do**
 8:                 $s' \leftarrow$ nextShift(s)
 9:                 **if** shift applicable for employee **then**
10:                     $x \leftarrow$ new Assignment(e', s')
11:                     update $\leftarrow$ true
12:                     $e \leftarrow$ sort(e')
13:                     $s \leftarrow$ sort(s')
14:                 **end if**
15:             **end while**
16:         **end while**
17:     **end while**
18:     **return** x
19: **end procedure**

Improvement

The two methods of generating solutions in this thesis are generation by MiniZinc and generation by the greedy construction heuristic presented just now.

After generation, the valid solutions are then further improved by a metaheuristic, in this case, VND or the whole GVNS . For these two approaches, different termination criteria exist. For a pure VND, which is deterministic, the solution is improved until it is optimal with respects to the neighbourhoods in the VND (or optionally, until some sort of timeout is reached before the locally optimal result is reached). When using the GVNS, the shaking makes it impossible to run out of solutions to try, and since it also is not possible to decide if any given solution is optimal, there can be no definitive stopping point. Therefore, the only viable way to stop the GVNS or any metaheuristic with a randomised component is to use a timeout or an iteration limit.

Algorithm 11 assignShifts

Input: solution x
Input: set of sorted shifts to be assigned s
Input: sorted list of employees e
Output: solution x

1: **procedure** ASSIGNSHIFTS(x,s,e)
2:     update $\leftarrow$ true
3:     **while** update == true **do**
4:         update $\leftarrow$ false
5:        **for** $e' \leftarrow$ next applicable employee that needs shifts **do**
6:           **for** $s' \leftarrow$ next shift still needing employees **do**
7:             **if** shift s' applicable for employee e' **then**
8:               **if** shift in favourite house of e' **then**
9:                 $x \leftarrow$ new Assignment(e', s')
10:                 update $\leftarrow$ true
11:                 $e \leftarrow$ sort(e')
12:               **else**
13:                 backlog $\leftarrow s'$
14:               **end if**
15:             **end if**
16:           **end for**
17:           **if** backlog not empty and e' still needs shifts **then**
18:             $x \leftarrow$ new Assignments(e',backlog)
19:             update $\leftarrow$ true
20:             $e \leftarrow$ sort(e')
21:           **end if**
22:        **end for**
23:     **end while**
24:     **return** x
25: **end procedure**

CHAPTER 6

Computational Study

In this chapter, the computational results of the presented algorithms will be discussed. The study has been conducted by using ten real-world instances over the course of two years. Different data points have been chosen based on the different circumstances of the particular planning horizon. A brief overview can be seen in Table 6.1. Instances of the CESP can generally be in one of two different states: $\sum_w N_w > \sum_s R_s$ or vice versa. That is to say, either there is more potential work than requirements (e.g., February 2019), or more requirements than people who could fill them (e.g., December 2019). To give those statements a little bit of context, February is often a slow month — it is right after the holidays and university students often have less university arrangements than during the semester. In stark contrast, during the winter months, there are often the largest exhibitions of the year. If there is an abundance of offered work, then the main problem will be to satisfy the fairness constraints as well as possible while minimizing the amount of shifts that ignore the favourite houses declarations of the employees. Else, the main challenge is to assign employees to open shifts as fairly as possible.

Technically, the study has been run on a virtualised system with a 12-core Intel Xeon Gold 6134 at 3.20 GHz and 20 GB of RAM. Java 13.0.1 and MiniZinc 2.3.2 with GeCode 6.1.1 have been used. The order of the VND-neighbourhoods has been optimised by using irace [21], using the same instances as in the study. All VND neighbourhoods use the BestImprovement-strategy.

The order determined by irace was:

- moveShift

- addShift

- reassignShift

- reassignToDifferentDayShift

- SwapShift

The shaking neighbourhoods have also been optimised by irace, although confidence in the optimality of this configuration is not very high. For one, it is quite a complex task in itself to even choose a right order, and for the shaking neighbourhoods, order and size had to be optimised. Furthermore, VND neighbourhoods were also set to be optimised again, because there might be different optimal configurations when using the GVNS versus running just the VND. Another big part is that the GVNS has random elements within the shaking neighbourhoods which are being optimised and single results indicating one configuration being better than another are never absolute. Since optimising using irace is very resource-intensive, some options needed to be constrained a bit. For the process, irace was only allowed to choose in step sizes of 10, so nothing < 10 could have been chosen. Furthermore, the number of shaking neighbourhoods was fixed. The VND results determined were the same as above, and the shaking neighbourhood order and sizes were as such:

- RemoveShifts 20

- SwapReservesAndStandbyShifts 20

- RemoveShifts 40

- SwapReservesAndStandbyShifts 30

All neighbourhoods are described in detail in the corresponding Chapter 5.2.1.

Due to the nature of the problem, there are a few pitfalls when reading the data, which will be explained in the following. Even if $\sum_w N_w > \sum_s R_s$ holds, this does not mean that every requirement can be satisfied. It is possible that nobody is available for a certain shift, or more realistically, shifts on special days like Christmas might have high requirements but a small amount of people who want to work on that day.

Furthermore, there are two types of shifts. Normal shifts are assigned based on employees opting in for a specific day, and are counted against the shift limit N_w. For event shifts, employees will choose a specific shift, which is not counted against the limit N_w. Instead, it is assumed that every special shift that an employee volunteers for is intended to be worked. Therefore, the numbers are shifted again—$\sum N_w < \sum_s R_s$ still does not necessarily mean that shifts will be left unassigned.

The table definitions are the same as seen in Chapter 4. For added clarity, the instance overview in Table 6.1 will have both $\sum_s R_s$ and $\sum_s R'_s$, which is defined as $\sum_{\{(d,l)\in S|l\notin E_s\}} R_s$ — requirements without special event shifts. While $|H|$ and $|E|$ will be separate in the following table, in all the other tables, where applicable, they will be combined into $|L|$ as per the definition.

---

[1]$\sum_{\{(d,l)\in S|l\notin E_s\}} R_s$ — requirements without special event shifts

Table 6.1: Overview of the instances

Instance	$|W|$	$|D|$	$|H|$	$|E|$	$\sum_w N_w$	$\sum_s R_s$	$\sum_s R_s'$ [1]
June 2018	198	30	14	5	851	1064	910
July 2018	196	31	14	3	823	1150	1057
September 2018	183	30	12	4	777	917	834
October 2018	261	31	12	16	980	1059	804
February 2019	172	28	11	4	746	678	567
April 2019	170	30	11	7	718	914	786
August 2019	200	31	13	3	767	944	879
December 2019	281	31	12	12	1160	1524	1275
February 2020	177	29	11	7	628	760	698
March 2020	234	31	12	10	987	970	798

The rest of the chapter consists of evaluations of the algorithms. In this order, the initial solution construction algorithms, the VND performance and the GVNS performance will be showcased. Afterwards, the non-deterministic algorithms will be run multiple times and their statistical values will be presented in Chapter 6.4. An overview and summary of all the results is presented in Chapter 6.5 followed by some in-depth objective values for some of the results. At the end of the chapter, a careful comparison to the results of the human planners with many caveats will be presented.

In the vast majority of the sections, three configurations will be evaluated. Each of the configurations uses the same neighbourhood order and settings as presented previously, their difference is in the generation algorithm for their initial solution and whether neighbourhood shifting is enabled or not.

The three configurations are:

- GG: The first configuration uses initial solutions by greedy construction heuristic and no neighbourhood shifting

- MZ: The second configuration uses initial solutions by MiniZinc generation and no neighbourhood shifting.

- GGNBS: The third configuration uses initial solutions by greedy construction heuristic and utilises neighbourhood shifting.

As can be seen, there is no configuration that uses an initial solution by MiniZinc generation and has enabled neighbourhood shifting. This is because it is reasonable to assume that the differences between GG and MZ and GG and GGNBS can be used to extrapolate possible benefits of another configuration. Therefore, another set of tables with probably very similar results does not seem relevant.

For efficiency, delta evaluation was implemented as a second step after the algorithm as such was already finished. It was therefore possible to measure performance and runtime

of a sample instance both before and after introducing delta evaluation. The results can be seen in Table 6.2.

Table 6.2: Time used for calcuation of different tasks of algorithm on Instance December 2019 configuration GGNBS

Type	Neighbourhoods	Feasibility	Objective value	Rest	$\Delta_t[m]$
Delta	92%	4%	3%	1%	166
Full	28%	20%	51%	1%	514

6.1 Generation

In the following, the initial solution construction algorithms, generation by MiniZinc and by greedy construction heuristic, will be shown and contrasted. For the greedy construction heuristic, each instance was used to generate an initial solution 100 times, as can be seen in Table 6.3.

Table 6.3: Statistics for the runs for initial solutions by greedy construction heuristic

Instance	# runs	$\overline{f}$	min f	max f	$\overline{\Delta t}[s]$
June 2018	100	0.508737	0.292888	0.704425	1.102
July 2018	100	2.06908	1.69798	2.36706	0.989
September 2018	100	0.67476	0.509202	0.829376	0.821
October 2018	100	0.386239	0.364701	0.408151	1.666
February 2019	100	0.355947	0.329626	0.376256	0.654
April 2019	100	0.611008	0.444807	0.754155	1.184
August 2019	100	0.914605	0.795904	1.01204	0.942
December 2019	100	0.626811	0.547968	0.723275	2.495
February 2020	100	0.340465	0.283001	0.46238	0.582
March 2020	100	0.343198	0.325177	0.363557	1.348

The median over those instances was then used as a single greedy heuristic sample for later runs of the VND or GVNS which require a single initial solution. The median result was chosen because it is the fairest depiction of the greedy heuristic before the subsequent tests which would only generate one result and improve this. If the minimum result was chosen instead, it could skew the study unfairly in favour of the greedy heuristic. The median initial solutions can be seen in Table 6.4.

For the MiniZinc generation, MiniZinc was invoked once and the result can be seen in Table 6.5. While using the same exact instance as has been used for the greedy heuristic

---

[2]In December 2019, a main favourable house was not used as such because of organizational details, therefore the amount of C_4^s misses is very high because many employees did not update their preferences accordingly.

Table 6.4: Chosen median initial solutions by greedy construction heuristic

Instance	|W|	|D|	|L|	$\sum_s R_s$	$\sum_w \|A_w\|$	C_4^s miss	f
June 2018	198	30	19	1064	1046	67	0.511753
July 2018	196	31	17	1150	994	11	2.06385
September 2018	183	30	16	917	877	77	0.675832
October 2018	261	31	28	1059	1059	71	0.386344
February 2019	172	28	15	678	672	88	0.357097
April 2019	170	30	18	914	864	89	0.609888
August 2019	200	31	16	944	877	109	0.922087
December 2019 [2]	281	31	24	1524	1419	236	0.626041
February 2020	177	29	18	760	742	112	0.336798
March 2020	234	31	22	970	969	125	0.342923

instances, two instances could not be resolved by the chosen solver within the allotted time-frame of one hour. These will be highlighted in this table, but omitted in all the latter tables. Therefore, there will only be eight instances used for the MiniZinc-based solutions.

Table 6.5: Initial solutions by MiniZinc solver

Instance	|W|	|D|	|L|	$\sum_s R_s$	$\sum_w \|A_w\|$	C_4^s miss	f	$\Delta_t[m]$
June 2018	198	30	19	1064	915	199	2.03043	4.76
July 2018	196	31	17	1150	929	215	3.21263	4.13
September 2018	183	30	16	917	834	227	1.5328	3.35
October 2018	-	-	-	-	-	-	-	60
February 2019	-	-	-	-	-	-	-	60
April 2019	170	30	18	914	773	197	2.12269	3.48
August 2019	200	31	16	944	819	276	2.01435	4.03
December 2019	281	31	24	1524	1259	393	2.64121	11.13
February 2020	177	29	18	760	710	240	1.25008	3.42
March 2020	234	31	22	970	956	323	0.696922	7.22

While not being on the tables, relevant points to consider when comparing the generation approaches are:

- Computation time and cost: while the greedy heuristic takes a few seconds and a few hundred megabyte of RAM, the MiniZinc solver takes between 3 and 12 minutes for the first solution and uses between 2 and 15 gigabyte of RAM.

- Reliability: as seen in Table 6.5, MiniZinc is not always able to find a solution within the time limit, which was one hour in this case. It is generally possible to manipulate an instance slightly for MiniZinc to be able to solve it in a reasonable

time frame, but that takes time, effort and knowledge. At some point, instance size is also a real problem.

- Solution quality: Without wanting to pre-empt interesting conclusions from the latter tables, solution quality is a factor. Initial solutions by MiniZinc generally start in a less desirable state than greedy generated solutions and will increase runtime until the VND reaches the local minimum—if the program is only used for a short amount of time, this is paramount. Interestingly, solutions using greedy generated initial solutions and MiniZinc initial solutions do show very similar results when being run until the VND reaches its local minimum, with no clearly superior algorithm. It is relevant to note at this point that MiniZinc solutions work with more information from the start—it includes the soft constraints and all hard constraints, while the greedy heuristic just tries to build a good initial roster based on a minimum of rules and restarts if the generated roster is not feasible.

6.2 VND

The following section shows the results of solving the instances with different configurations while only running a VND. The types of initial solutions are explained in the previous section, being generated by the greedy heuristic (GG and GGNBS) or by MiniZinc (MZ). The greedy heuristic configurations are with (GGNBS) and without (GG) shifting underperforming neighbourhoods, as explained in Chapter 5.2. In general, the recently presented median initial greedy generated solution is used for both GG and GGNBS.

6.2.1 15 Minutes

The first results have been measured after running the VND for 15 minutes. As can be seen from Tables 6.6, 6.7 and 6.8, the initial solution by GGNBS shows better runtimes and can get a little ahead in bigger instances especially. The solutions using MZ meanwhile are not competitive at all after the first 15 minutes.

Table 6.6: VND configuration GG after 15 minutes

Instance	$|W|$	$|D|$	$|L|$	$\sum_s R_s$	$\sum_w |A_w|$	C_4^s miss	f	$\Delta_t[m]$
June 2018	198	30	19	1064	1050	29	0.0651353	15
July 2018	196	31	17	1150	1007	9	0.234874	15
September 2018	183	30	16	917	878	11	0.161932	15
October 2018	261	31	28	1059	1059	33	0.0819516	15
February 2019	172	28	15	678	672	19	0.0471305	11.17
April 2019	170	30	18	914	870	24	0.11497	15
August 2019	200	31	16	944	882	30	0.212428	15
December 2019	281	31	24	1524	1428	217	0.303567	15
February 2020	177	29	18	760	749	21	0.0936465	12.98
March 2020	234	31	22	970	969	17	0.0486883	15

Table 6.7: VND configuration MZ after 15 minutes

Instance	$|W|$	$|D|$	$|L|$	$\sum_s R_s$	$\sum_w |A_w|$	C_4^s miss	f	$\Delta_t[m]$
June 2018	198	30	19	1064	1044	89	0.0973182	15
July 2018	196	31	17	1150	1006	49	0.257491	15
September 2018	183	30	16	917	878	20	0.165259	15
April 2019	170	30	18	914	862	41	0.129369	15
August 2019	200	31	16	944	879	78	0.23685	15
December 2019	281	31	24	1524	1425	221	0.286973	15
February 2020	177	29	18	760	751	25	0.103586	15
March 2020	234	31	22	970	969	104	0.188905	15

Table 6.8: VND configuration GGNBS after 15 minutes

| Instance	$|W|$	$|D|$	$|L|$	$\sum_s R_s$	$\sum_w |A_w|$	C_4^s miss	f	$\Delta_t[m]$
June 2018	198	30	19	1064	1048	17	0.0605393	15
July 2018	196	31	17	1150	1001	7	0.236282	15
September 2018	183	30	16	917	877	13	0.165798	15
October 2018	261	31	28	1059	1059	22	0.0606749	15
February 2019	172	28	15	678	672	19	0.0467215	10.7
April 2019	170	30	18	914	867	22	0.116186	15
August 2019	200	31	16	944	877	21	0.215636	15
December 2019	281	31	24	1524	1424	214	0.303317	15
February 2020	177	29	18	760	747	17	0.0947747	11.77
March 2020	234	31	22	970	969	13	0.0465501	15

6.2.2 Completion

The following is a run to completion, which means that the VND has no time limit and can run until it is unable to find a better result in any of its neighbourhoods, also referred to as reaching a local minimum. Since the VND is a deterministic algorithm without diversification, this is bound to happen. Here, both the final result as well as the duration of the run are of importance, and the results are interesting. The results of configuration GG are shown in Table 6.9, and will be used a baseline for the other two configurations.

When utilizing neighbourhood shifting as seen in Table 6.11 , the time needed to converge is lowered while the results are pretty similar. It would seem reasonable that GGNBS would give worse results if the neighbourhood ordering we used is optimal, since the shifting cycles the order of the neighbourhoods, thus using non-optimal configurations more often than not. The differences between the generated solutions and the MiniZinc solutions as seen in Table 6.10 is also interesting—while none of the solutions are very far apart in either objective value or actual results, the MiniZinc solutions seem to be a bit better in assigning shifts and a bit worse with the C_4^s constraint, while obviously taking far longer than either of the other results. Furthermore, plots of the changes

in neighbourhoods for configurations GG and GGNBS have been created. For each configuration, there is a plot over the whole improvement process and a plot over only the last hour. For GG, these are Figures 6.1 and 6.2 respectively. For GGNBS, they are Figures 6.3 and 6.4. The most interesting part of the visualisation is how the neighbourhood changes are different in the two configurations. GGNBS usually exhausts one neighbourhood and then moves on, which leads to very uniform looking curves. GG instead adheres to the neighbourhood order, which is especially nice to compare in the more detailed Figures 6.2 and 6.4.

Table 6.9: VND configuration GG to completion

Instance	$|W|$	$|D|$	$|L|$	$\sum_s R_s$	$\sum_w |A_w|$	C_4^s miss	f	$\Delta_t[m]$
June 2018	198	30	19	1064	1056	10	0.0535215	44.72
July 2018	196	31	17	1150	1008	6	0.229451	35.38
September 2018	183	30	16	917	878	11	0.161147	24.57
October 2018	261	31	28	1059	1059	0	0.0319417	38.72
February 2019	172	28	15	678	672	19	0.0471305	11.57
April 2019	170	30	18	914	870	22	0.112915	24.67
August 2019	200	31	16	944	886	19	0.196101	26.72
December 2019	281	31	24	1524	1448	49	0.159002	224.42
February 2020	177	29	18	760	749	21	0.0936465	12.27
March 2020	234	31	22	970	969	0	0.0387968	32.38

Table 6.10: VND configuration MZ to completion

Instance	$|W|$	$|D|$	$|L|$	$\sum_s R_s$	$\sum_w |A_w|$	C_4^s miss	f	$\Delta_t[m]$
June 2018	198	30	19	1064	1047	7	0.0549755	60.88
July 2018	196	31	17	1150	1010	10	0.236646	38.63
September 2018	183	30	16	917	880	9	0.158936	31.2
April 2019	170	30	18	914	865	24	0.116743	30.15
August 2019	200	31	16	944	885	21	0.198219	43.87
December 2019	281	31	24	1524	1446	44	0.157538	225.1
February 2020	177	29	18	760	751	24	0.102661	16.8
March 2020	234	31	22	970	969	1	0.0387589	56.93

A succinct comparison for only the core values of these tests can be found in Chapter 6.5.

6.3   GVNS

In this section, the same tests as before have been run, this time using a GVNS. Since the GVNS is a combination of a VND with a diversification mechanism in form of a VNS, no runs to completion can possibly be achieved. Therefore, a run with a timeout of 12 hours has been done. This leaves the GVNS enough time to repeatedly apply shaking algorithms for even the largest instances. These results are based on one execution for

Table 6.11: VND configuration GGNBS to completion

Instance	$|W|$	$|D|$	$|L|$	$\sum_s R_s$	$\sum_w |A_w|$	C_4^s miss	f	$\Delta_t[m]$
June 2018	198	30	19	1064	1050	8	0.0530264	38.42
July 2018	196	31	17	1150	1009	4	0.225622	29.53
September 2018	183	30	16	917	878	13	0.161547	18.52
October 2018	261	31	28	1059	1059	0	0.0315137	40.48
February 2019	172	28	15	678	672	19	0.0467215	9.23
April 2019	170	30	18	914	873	25	0.111193	18.88
August 2019	200	31	16	944	882	17	0.201962	28.47
December 2019	281	31	24	1524	1435	38	0.162306	166.62
February 2020	177	29	18	760	747	17	0.0947747	10.38
March 2020	234	31	22	970	969	1	0.0386226	31.35
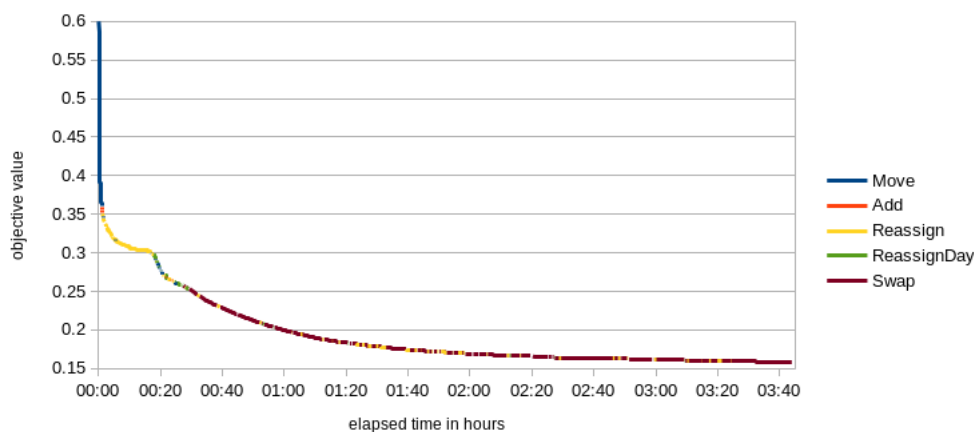
Figure 6.1: Visualisation of the improvements done by VND GG for Instance December 2019. Each neighbourhood is their own colour, described in the legend.

every instance, and since GVNS has random elements, are not completely representative. They should only be used to show the approximate results. In the next chapter, Chapter 6.4 all algorithms have been run multiple times on a single instance to get a closer look on their performance. As before, the first table, Table 6.12 shows the solutions generated by configuration GG, Table 6.13 shows the results for MZ and 6.14 shows the results for GGNBS.

Additionally to the results of the algorithm, some additional properties of the GVNS have also been recorded. Figures 6.5 and 6.6 show the change of objective function over time for the instances September 2018 and August 2019, respectively. Each node in the visualisation corresponds to a shaking and subsequent improvement by the VND which ultimately found a new best solution. Both plots start relatively skewed at around the one hour mark, which is the time for the first completion of the VND. As can be
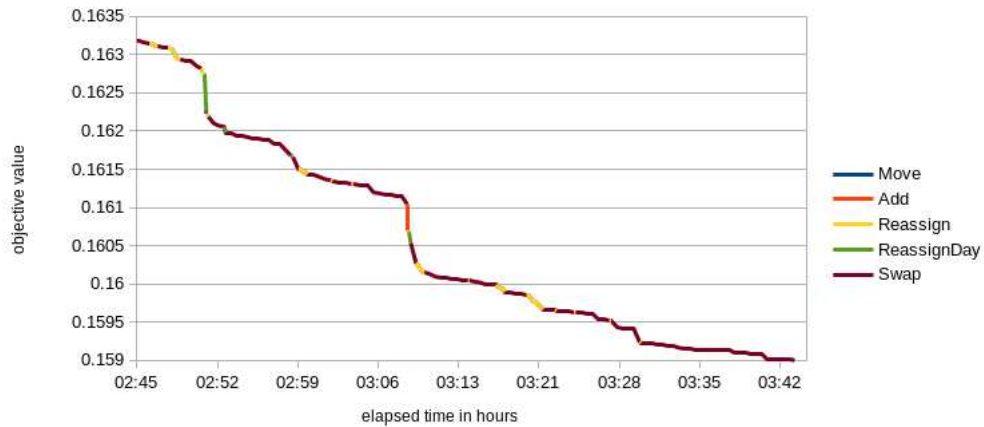
Figure 6.2: Visualisation of the improvements done by VND GG for Instance December 2019. Closeup of the last few hours to see the neighbourhood changes in detail.
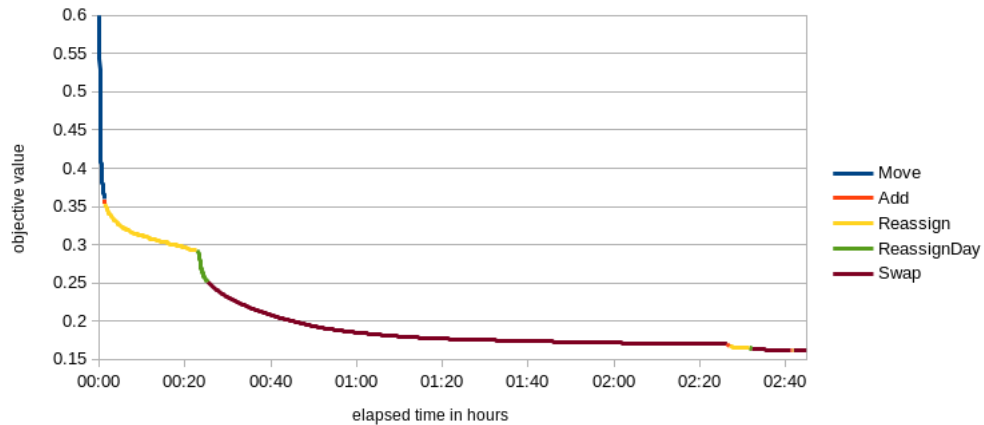
Figure 6.3: Visualisation of the improvements done by VND GGNBS for Instance December 2019. Each neighbourhood is their own colour, described in the legend.

seen, improvements come fast during the first hours, and while they do slow during the latter half, none show real signs of stagnating. Also, the visualisations have been done for comparably smaller instances, with larger instances presumably showing the same pattern, only stretched out over a large time-frame, since the completion of each shaking and subsequent improvement just takes a lot longer.

Since the 12 hour GVNS run is the longest test that has been done with the instances, it is the perfect place to provide additional statistics on how each of the neighbourhoods is performing. For this behalf, there are two tables, detailing the VND and shaking neighbourhood performance of configurations GG and GGNBS.
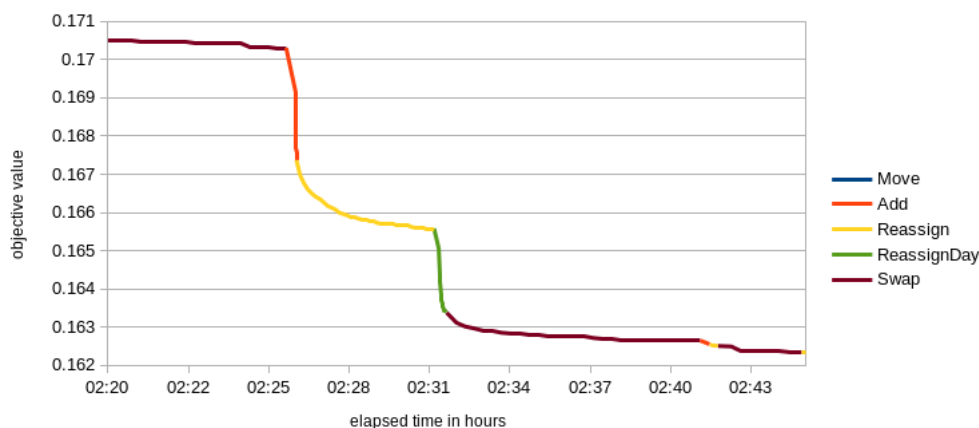
Figure 6.4: Visualisation of the improvements done by VND GGNBS for Instance December 2019. Closeup of the last few hours to see the neighbourhood changes in detail.

Table 6.12: GVNS solution configuration GG after 12h

Instance	$|W|$	$|D|$	$|L|$	$\sum_s R_s$	$\sum_w |A_w|$	C_4^s miss	f
June 2018	198	30	19	1064	1058	2	0.0501021
July 2018	196	31	17	1150	1018	3	0.214228
September 2018	183	30	16	917	880	6	0.155464
October 2018	261	31	28	1059	1059	1	0.0303422
February 2019	172	28	15	678	672	12	0.0415056
April 2019	170	30	18	914	876	18	0.0988159
August 2019	200	31	16	944	895	14	0.175012
December 2019	281	31	24	1524	1452	38	0.15679
February 2020	177	29	18	760	751	13	0.0867861
March 2020	234	31	22	970	968	0	0.0382205

Table 6.13: GVNS solution configuration MZ after 12h

Instance	$|W|$	$|D|$	$|L|$	$\sum_s R_s$	$\sum_w |A_w|$	C_4^s miss	f
June 2018	198	30	19	1064	1058	2	0.0503643
July 2018	196	31	17	1150	1022	4	0.208552
September 2018	183	30	16	917	880	4	0.153827
April 2019	170	30	18	914	878	17	0.100181
August 2019	200	31	16	944	893	14	0.180825
December 2019	281	31	24	1524	1454	36	0.154181
February 2020	177	29	18	760	753	13	0.0848931
March 2020	234	31	22	970	969	1	0.038306

Table 6.14: GVNS solution configuration GGNBS after 12h

Instance	$|W|$	$|D|$	$|L|$	$\sum_s R_s$	$\sum_w |A_w|$	C_4^s miss	f
June 2018	198	30	19	1064	1057	3	0.0505064
July 2018	196	31	17	1150	1018	3	0.213172
September 2018	183	30	16	917	880	5	0.154967
October 2018	261	31	28	1059	1058	0	0.0299924
February 2019	172	28	15	678	672	12	0.0414375
April 2019	170	30	18	914	878	19	0.0987089
August 2019	200	31	16	944	897	13	0.173479
December 2019	281	31	24	1524	1445	35	0.15652
February 2020	177	29	18	760	751	7	0.0851367
March 2020	234	31	22	970	968	0	0.0385363
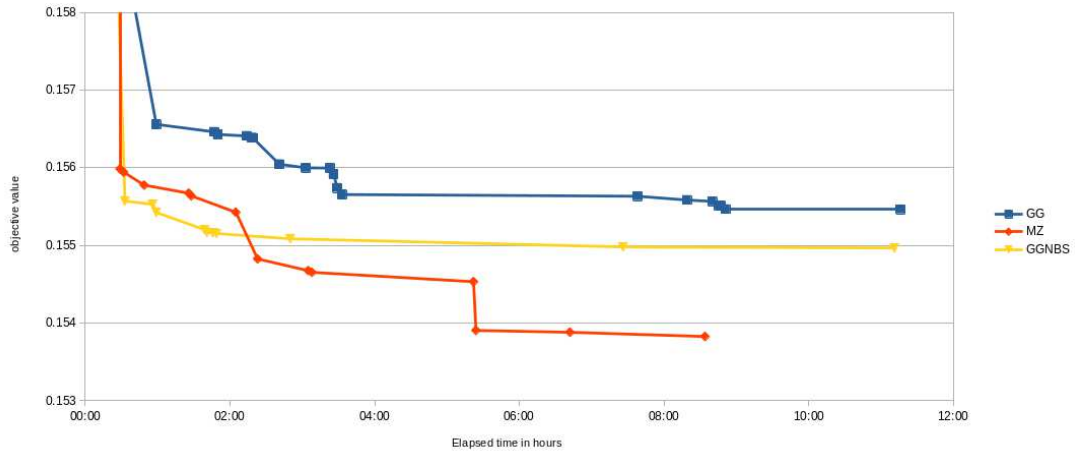
Figure 6.5: Visualisation of the improvements done by GVNS for Instance September 2018. Starting points are the construction heuristics, which are not shown to preserve focus. GG, GGNBS: 0.6758, MZ: 1.5328

The VND neighbourhood statistics tables will be discussed first. The results can be seen in Tables 6.15 and 6.16. For each neighbourhood, the number of successful moves is written first, followed by the number of unsuccessful moves. One of the most interesting parts in this comparison is the number of additional VND neighbourhood moves the GGNBS has completed in relation to the GG. The GGNBS configuration consistently has a higher successful to unsuccessful move percentage than the GG configuration. It can also be seen how the GGNBS configuration is skewed to use the SwapShift neighbourhood more often. This is the largest neighbourhood which provides lots of mostly small improvements. This is not to say that each move is of the same value—by shifting the neighbourhoods from their normal position, a situation where a suboptimal neighbourhood move is performed by the currently first neighbourhood, while the original first neighbourhood would have
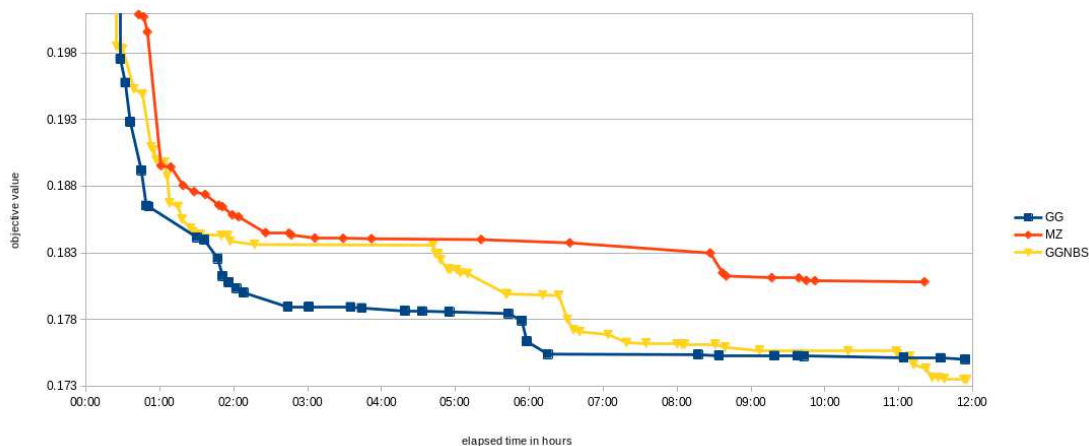
Figure 6.6: Visualisation of the improvements done by GVNS for Instance August 2019. Starting points are the construction heuristics, which are not shown to preserve focus. GG, GGNBS: 0.9220, MZ: 2.014

had a better move, is a very realistic possibility.

Table 6.15: GVNS solution configuration GG after 12h, VND neighourhood statistics

Instance	Move	Add	Reass	ReassDay	Swap
June 2018	4.7k/7.1M	3.0k/194.4k	4.1k/22.1M	1.6k/926.4k	13.4k/26.3M
July 2018	12.4k/21.5M	16.4k/774.1k	1.3k/9.2M	3.2k/22.8M	10.5k/28.4M
September 2018	8.5k/12.6M	5.8k/176.9k	21.5k/37.8M	2.0k/1.1M	18.8k/32.1M
October 2018	5.2k/8.3M	3.1k/248.8k	35.0k/189.6M	76/24.3k	6.3k/23.4M
February 2019	15.2k/16.7M	8.3k/367.9k	103.7k/218.4M	1.5k/60.0k	11.1k/19.9M
April 2019	10.6k/14.4M	7.9k/307.4k	9.6k/10.0M	3.2k/3.6M	26.7k/30.7M
August 2019	13.5k/23.2M	12.3k/346.2k	2.3k/6.8M	3.3k/15.4M	24.8k/42.2M
December 2019	2.0k/5.2M	662/91.5k	2.4k/16.9M	566/2.4M	4.4k/12.0M
February 2020	13.4k/16.3M	8.3k/243.3k	34.9k/38.7M	2.2k/745.0k	17.0k/25.1M
March 2020	6.4k/10.0M	3.9k/208.0k	50.2k/155.7M	743/20.4k	7.2k/18.3M

As can also easily be seen, some neighbourhoods, in particular ReassignToDifferentDay, really do not perform well on certain instances. It is entirely possible that it suffers from being so far in the back of the queue, but it also does not fare much better using the GGNBS configuration. Another interpretation would be that it is unreasonable to find open shifts in some instances, and that it can be emulated by a combination of other neighbourhoods, maybe MoveShift and SwapShift. On the other hand, this neighbourhood provides relatively high impact moves when it is able to find neighbours, as can be seen on Figures 6.1 and 6.3.

The shaking neighbourhood statistics can be seen in Tables 6.17 and 6.18. They are similar to the VND statistics, having first the shaking operations that resulted in a

Table 6.16: GVNS solution configuration GGNBS after 12h, VND neighbourhood statistics

Instance	Move	Add	Reass	ReassDay	Swap
June 2018	2.4k/1.4M	3.5k/181.6k	5.6k/9.7M	851/515.2k	18.2k/35.9M
July 2018	12.3k/9.2M	22.2k/1.1M	1.2k/3.8M	4.1k/14.4M	16.1k/47.1M
September 2018	3.7k/1.7M	5.6k/192.8k	21.1k/22.9M	559/241.5k	24.8k/42.6M
October 2018	1.8k/778.0k	4.4k/311.0k	32.3k/145.7M	311/118.6k	11.7k/44.9M
February 2019	6.3k/2.5M	11.4k/485.3k	124.9k/220.8M	2.9k/127.6k	21.4k/41.9M
April 2019	5.0k/2.5M	7.7k/181.9k	6.7k/3.6M	1.1k/805.8k	30.8k/36.7M
August 2019	10.4k/6.9M	13.9k/344.6k	2.4k/2.1M	1.4k/3.3M	31.3k/56.4M
December 2019	1.6k/1.2M	1.5k/94.0k	3.2k/10.9M	620/981.6k	8.2k/25.3M
February 2020	8.2k/3.8M	12.5k/389.8k	49.5k/38.7M	3.0k/944.1k	33.9k/51.0M
March 2020	2.0k/808.5k	4.2k/229.3k	52.3k/128.5M	909/28.0k	8.9k/24.3M

new local minimum after subsequent VND operations followed by all the valid shakings. Furthermore, the sum of all successful shakings and the time that the last solution has been found are also depicted.

Table 6.17: GVNS solution configuration GG after 12h, shaking neighbourhood statistics. The first four columns are the shaking neighbourhoods, afterwards is the sum of successful shaking attempts and the time the last solution has been found.

Instance	Rem20	Swap20	Rem40	Swap30	$\#Shaking^{succ}$	Last Sol $\Delta_t[h]$
June 2018	10/25	3/15	0/11	2/11	15	11.60
July 2018	6/65	8/59	4/51	4/46	22	8.93
September 2018	17/37	3/20	4/16	0/12	24	11.28
October 2018	1/12	2/11	2/9	0/7	5	6.93
February 2019	10/40	2/30	2/28	0/26	14	10.48
April 2019	13/51	5/38	5/32	1/27	24	11.52
August 2019	10/64	13/54	4/40	5/36	32	11.92
December 2019	6/9	2/3	0/0	0/0	8	10.73
February 2020	12/45	2/33	3/30	2/27	19	9.77
March 2020	3/14	1/11	3/10	0/7	7	8.88

An interesting part here is the last time that the GVNS made any improvement to the solution. There are some values that definitely stand out, like the 3.55 hours for the March 2020 instance in Table 6.18. However, none of the very low numbers in any of the two tables are replicated in the other, and most of the instances found improvements in the last hours, so it can be argued that the GVNS could be let run even longer to achieve better results. For a closer view of the improvements over time, refer to Figures 6.5 and 6.6 earlier in the section.

Table 6.18: GVNS solution configuration GGNBS after 12h, shaking neighbourhood statistics. The first four columns are the shaking neighbourhoods, afterwards is the sum of successful shaking attempts and the time the last solution has been found.

Instance	Rem20	Swap20	Rem40	Swap30	$\#Shaking^{succ}$	Last Sol $\Delta_t[h]$
June 2018	6/24	3/18	2/15	4/13	15	11.45
July 2018	14/93	5/79	1/74	4/72	24	9.80
September 2018	10/29	0/19	2/18	0/16	12	11.18
October 2018	7/20	2/13	5/10	0/5	14	10.93
February 2019	21/59	1/38	1/36	0/35	23	11.27
April 2019	14/50	4/36	0/32	3/32	21	7.63
August 2019	26/89	22/63	2/40	6/38	56	11.92
December 2019	8/13	2/5	0/3	0/2	10	8.88
February 2020	12/61	2/49	6/47	2/41	22	9.63
March 2020	8/19	0/11	2/11	1/9	11	3.55

6.4 Statistics for Variable Results

In this chapter are some of the previous results with variable outcome (that is, diversification strategies of any kind) and their performance over several runs based on one instance. This can be seen in Table 6.19 and should give a feeling on how random and diverse the solutions generated can be and what kind of performance should be expected from each of them.

The first lines of the table are titled "Reference"—these are two results that have been pulled from the respective tables already presented for convenience, to see how the repeated results measure up against the others. The other tests are VND runs on freshly generated instances for each run, which is the only way to randomise the VND, and some variations of VNS runs.

Table 6.19: Statistics for multiple runs of different algorithms for instance August 19

Algorithm	# runs	$\overline{f}$	min f	max f
VND GG Completion Ref	1	0.196101	0.196101	0.196101
VNS GGNBS 12h Ref	1	0.173479	0.173479	0.173479
New gen. VND Completion GG	10	0.200017	0.191796	0.203353
New gen. VND Completion GGNBS	10	0.204342	0.199288	0.212167
VNS GG 8h	10	0.178225	0.175471	0.181294
VNS MZ 8h	10	0.180952	0.177056	0.183412
VNS GGNBS 8h	10	0.17862	0.17512	0.183775

As can be easily seen, running VND multiple times to completion with different instances is not a viable alternative to running a VNS. If we compare the minimum from the 10 runs of VND with the maximum of the 10 VNS runs, the difference is still huge and shows no sign of being close.

As a second observation, the VNS results suggest that greedy generated initial solutions with or without neighbourhood shifting (GGNBS and GG) achieves comparable results, while solutions based on MiniZinc are slightly worse. GGNBS and GG being very close can be relatively easy to argue—the problem has a lot of possible solutions, and the configurations just take different ways to arrive at similarly good ones. Coupled with shaking by the VNS, it is possible that the differences between the configurations are negligible. MZ delivering worse results is not that obvious, though. It could be argued that the additional time they take to complete the first VND gives them a disadvantage at the VNS stage, or that the initial solutions generally lead them in slightly worse areas of the solution space. It could also be just an issue of this specific instance. Additionally, Figure 6.6 in the GVNS chapter can give a visualisation of how the GVNS behaves.

6.5   Comparison Between Modes

This chapter offers a quick reference and overview of the presented configurations and their performances in the case of a VND to completion and a GVNS after 12 hours, in Tables 6.20 and 6.21 respectively.

Table 6.20: Comparison VND to completion

Instance	GG	MZ	GGNBS
June 2018	0.0535	0.0550	**0.0530**
July 2018	0.2295	0.2366	**0.2256**
September 2018	0.1611	**0.1589**	0.1615
October 2018	0.0319	-	**0.0315**
February 2019	0.0471	-	**0.0467**
April 2019	0.1129	0.1167	**0.1112**
August 2019	**0.1961**	0.1982	0.2020
December 2019	0.1590	**0.1575**	0.1623
February 2020	**0.0936**	0.1027	0.0948
March 2020	0.0388	0.0388	**0.0386**

6.6   Human Planning Results

In an effort to compare the results of the work to the currently used methods, the finalised used schedules developed by hand have also been evaluated based on the objective function and weights used in the rest of the study. Sadly, this is not as easy as it might seem at first, and all of the results presented are comparatively inaccurate. There are a couple of reasons for this, which can be reduced to three differences:

- The time the schedule was submitted: most of the schedules were submitted after the month was over, and there is no former version. That means that any changes between the initial schedule and the end of the month are in the schedule,

Table 6.21: Comparison VNS over 12h

Instance	GG	MZ	GGNBS
June 2018	**0.0501**	0.0503	0.0505
July 2018	0.2142	**0.2086**	0.2131
September 2018	0.1555	**0.1538**	0.1550
October 2018	0.0303	-	**0.0300**
February 2019	0.0415	-	**0.0414**
April 2019	0.0988	0.1002	**0.0987**
August 2019	0.1750	0.1808	**0.1734**
December 2019	0.1568	**0.1542**	0.1565
February 2020	0.0867	**0.0849**	0.0851
March 2020	**0.0382**	0.0383	0.0386

thus changing it. For example, people get sick or additional shifts are needed. Furthermore, as can be seen, there are generally always more employees—and sometimes even locations—in the finished schedules than at the beginning, owing to people responding later or short-term requirements.

- The state of the schedule: The schedule generated by the algorithms in this thesis is used as an initial solution for the human planners which can then further be refined depending on the circumstances, as discussed in the first point. The objective function does reflect that.

- The freedom of the planners: The algorithms work with constraints which are defined in a manner that enables the human planners to then break some of them. It could, for example, be beneficial to assign more people than strictly required to a shift, but the algorithm does not have the knowledge (or need) to know or understand that.

The results of these differences are that some hard constraints are broken, especially C_2^h, C_4^h and C_6^h / C_7^h. C_2^h is violated if a person does more shifts than they chose. This is easily done in reality because of two possibilities. Shifts can be switched or taken over from somebody else or there might be additional need for a specific shift later in the month, which will be communicated and where employees can volunteer. Both of these occurrences can push somebody over the amount of chosen shifts they initially stated, but it is not possible to decide which kind of shift it is.

C_4^h is violated if more people than required are assigned to a shift, and it is a simple matter of a planner seeing that there are more offers than demand and assigning additional shifts where they feel it would be useful. This is not a planning error, just a call that the algorithm is not able to make. The objective function is not able to handle these assignments, though, which is why the fairness constraints will be skewed starkly.

C_6^h and C_7^h are violated if somebody has more than the maximum amount of reserve or standby shifts. This does not happen often, but is possible, since it is actually a nice to have for the planners. While the algorithms are instructed to adhere strictly to the maximum, human planners can overrule this. Furthermore, the current maximum used for the study was one shift, while it was two at the time some of the instances have been used.

C_9^h is violated if there are not enough standby- or reserve shifts assigned. Since these shifts are virtual and are transformed to real shifts anyway, this is also okay. It is generally a sign that there was a shortage of work offers, though.

To prepare for the first table, a few unique columns have to be explained. Since there is overbooking of the required employees for individual shifts ($\sum_w A_w > \sum_s R_s$), it is no longer enough to just have those two values, because the amount of open shifts can no longer be gleaned from those numbers. Therefore, a new column opSh (open shifts) has been introduced, which denotes that number. The rest of the columns show the amount of violations for the respective constraints. This can be seen in Table 6.22.

Table 6.22: Human planning results

Instance	|W|	|D|	|L|	$\sum_s R_s$	$\sum_w A_w$	opSh[3]	C_4^s	C_2^h	C_4^h	$C_{6/7}^h$	C_9^h
June 2018[4]	212	30	19	1064	902	162	76	43	0	0	56
July 2018	201	31	17	1150	974	184	42	92	8	2	59
September 2018	201	30	16	944	878	66	69	46	0	3	2
October 2018	250	31	28	1059	1222	6	12	17	65	0	0
February 2019	173	28	15	678	720	7	14	8	38	0	1
April 2019	174	30	18	914	832	82	52	6	0	1	2
August 2019	201	31	16	944	878	66	61	46	0	3	2
December 2019	311	31	23	1533	1567	22	109	123	34	3	2
February 2020	176	29	18	760	769	18	50	32	20	1	1
March 2020	239	31	25	985	1021	17	23	8	42	0	0

As can clearly be seen, the numbers do not line up well with the algorithm-solved instances. The closest way to to compare them seems to be to compare the open shifts with the respective number of the algorithms ($\sum_s R_s - \sum_w |A_w|$) and the C_4^s misses. This gives a rough insight to two of the objective function components, with no easy way to do the same for the other two. A rough comparison can be seen in Table 6.23. The only real deduction to be made from this is that the algorithms seem more adapt at minimizing C_4^s misses, which is easily explained, since the strong point of any computer is to do monotonous tasks willingly and without error, such as every shift with every other shift to reduce misses, which is practically impossible for a human.

---

[4]open shifts

[4]final schedule for this month misses a lot of information required for correct comparison, therefore values here are especially unreliable

[5]Algorithmic input—since the number of locations and workers differs from human planning results and algorithmic inputs, the input values for the algorithms are denoted extra

Table 6.23: Human planning results compared to algorithm instances, configuration GGNBS

Instance	Human planning					Alg input[5]		GVNS12			VND Full	
	$|W|$	$|L|$	opSh	Sh%	C_4^s	$|W|$	$|L|$	opSh	Sh%	C_4^s	opSh	C_4^s
June 2018	212	19	162	15.2%	76	198	19	**7**	**0.6%**	**3**	14	8
July 2018	201	17	184	16%	42	196	17	**132**	**11.5%**	**3**	141	4
September 2018	201	16	66	6.9%	69	183	16	**37**	**4.0%**	**5**	39	13
October 2018	250	28	6	0.5%	12	261	28	1	0.1	**0**	0	0
February 2019	173	15	7	1.0%	14	172	15	**6**	**0.8%**	12	6	19
April 2019	174	18	82	8.9%	52	170	18	**36**	**3.9%**	19	41	25
August 2019	201	16	66	6.9%	61	200	16	**47**	**5.0%**	13	62	17
December 2019	311	23	**22**	**1.4%**	109	281	24	79	5.2%	**35**	89	38
February 2020	176	18	18	2.3%	50	177	18	**9**	**1.2%**	**7**	13	17
March 2020	239	25	17	1.7%	23	234	22	2	0.2%	**0**	1	1

Lastly, there is a comparison of soft constraint violations for algorithmic and human values which can be seen in Table 6.24. As can be seen, the algorithmic results are generally more optimised and consistent, as the σ values are lower. The different C_3^s $\bar{x}$ values exist because the number of assigned shifts is different for human and algorithmic results.

Table 6.24: Algorithmic vs. human soft constraint comparison. First column are algorithmic values, second are human values. C_1^s and C_4^s are computed by using the RMSE of the soft constraint value, C_2^s and C_3^s by using the mean $\bar{x}$ and the standard deviation of the soft constraint in parenthesis. All values interpreted as percentages. Algorithmic values are from VNS 12h GGNBS. The algorithmic approach is almost dominant regarding the defined objectives.

Instance	RMSE C_1^S	Algorithmic / Human $\bar{x}(\sigma)$ C_2^S	$\bar{x}(\sigma)$ C_3^S	RMSE C_4^S
June 2018	**1.26%**/44.14%	76.09%(**21.07%**)/73.10%(52.46%)	6.51%(6.37%)/0.83%(**2.38%**)	**1.50%**/15.83%
July 2018	**10.71%**/45.50%	82.70%(**30.66%**)/91.85%(55.71%)	6.34%(6.36%)/0.57%(**1.64%**)	**1.55%**/13.63%
September 2018	**10.26%**/18.72%	80.10%(**21.08%**)/84.09%(39.81%)	7.14%(**6.65%**)/9.21%(12.03%)	**1.95%**/19.58%
October 2018	**0.25%**/4.64%	75.10%(**15.50%**)/86.21%(43.12%)	7.75%(**7.68%**)/9.25%(13.21%)	**0.00%**/6.49%
February 2019	**1.65%**/3.82%	56.63%(**17.22%**)/78.40%(60.57%)	11.48%(**8.30%**)/12.22%(16.47%)	**3.28%**/10.15%
April 2019	**5.34%**/16.43%	80.36%(**22.23%**)/88.49%(36.40%)	8.53%(**7.87%**)/10.08%(12.26%)	**8.53%**/13.13%
August 2019	**8.10%**/18.72%	77.76%(**31.59%**)/84.09%(39.81%)	7.44%(**7.33%**)/9.21%(12.03%)	**3.66%**/17.55%
December 2019	9.80%/**7.95%**	87.27%(**22.59%**)/96.38%(37.72%)	5.33%(**7.06%**)/7.04%(13.78%)	**4.73%**/16.70%
February 2020	**3.54%**/7.50%	74.78%(**25.24%**)/81.36%(31.59%)	9.03%(**8.53%**)/11.20%(13.84%)	**2.86%**/16.05%
March 2020	**1.02%**/10.07%	64.09%(**17.83%**)/42.78%(23.44%)	8.74%(**7.56%**)/8.02%(11.13%)	**0.00%**/7.15%

Summary and Future Work

This thesis presents a new problem in the field of personnel scheduling that is closely related to the nurse rostering problem, the casual employee scheduling problem. It is formally defined in the thesis on the basis of a real-world occurrence and shown to be NP-hard. Both an exact and a heuristic solution are developed, based on the formal definition. The exact solution is based on constraint programming, and can be used to generate initial solutions most of the time, but not consistently. It is not performant enough to find competitive solutions. As an alternative, a greedy construction heuristic has been implemented, which consistently generates reasonable initial solutions tailored to this exact problem. The heuristic solution developed is a GVNS, consisting of a VND and a shaking mechanism, each with their own set of neighbourhoods. Different variants of the VND and GVNS are tested against real-world instances and found to give high-quality initial solutions for actual real-life rostering efforts.

In retrospective, starting from a naturally grown real-world problem and defining it with the input of multiple people was an organizational challenge that could have been handled a lot smoother. Aspects like fairness and phrases like 'should not happen' can mean something very different to different people, and some of the more complex rules have taken quite a few iterations to be developed in the way they are now. Some changes have only been made after the first schedules have been delivered and required quite a bit of redesigning.

The algorithms as presented in this thesis are efficient enough to solve the real-world problem they were designed to do. Of course, there is still a lot that could work better. There are always efficiency tweaks that could also be implemented, since most code is never truly optimal. Inquiries could be made into the GVNS algorithm to check how much a solution changes from the first completed VND run to a GVNS solution a set time later. How much of the initial solution is still intact—and is it because it is near-optimal or because the GVNS is not able to shake enough? Additional time could be used to change the balance between VNS and VND, either by removing VND neighbourhoods or

by limiting the time of each VND run within the VNS. Furthermore, as discussed when introducing the shaking neighbourhoods, research on their performance could still be done. From a purely practical standpoint, it would be interesting to adapt the system to allow for slight changes. Given a current best solution, change the input a little bit—like adding or subtracting an additional employee—and let it recalculate a new solution. The problem here is less in the technical aspects, since this can be done already, but from organizational aspects. Rosters are given out before the start of the month and not changed in any meaningful way afterwards, if they do not absolutely have to be. Furthermore, it is basically a question of good user interface design to make this process easier for the end user than just scheduling the additional shift themselves. This is, while being part of the real-world problem, not in the scope of the thesis.

MiniZinc model

```
%%%
%  input variables
%%%

% number of employees
int: workers;
% number of days
int: days;
% number of houses
int: houses;
% number of events
int: n_events;
% locations = houses + n_events
int: locations;
% the array index of the reserve category
int: index_reserve;
% the array index of the standby category
int: index_standby;
% minShift for standby/reserve shifts (C_minShift)
int: sr_minshift;
% (C_reserve) - maximum number of reserve shifts
int: r_maxshift;
% (C_standby) - maximum number of standby shifts
int: s_maxshift;
% h4: maximum hours per employee per month
int: max_hours;
set of int: special_events;

set of int: Houses = 1..houses;
set of int: Locations = 1..locations;
set of int: LocsWOStandby = Locations diff {index_standby};
set of int: Days = 1..days;
set of int: Workers = 1..workers;

% requirements for employees
array[Locations,Days] of int: requirements;

% number of shifts that employees want to do this month
array[Workers] of int: num_shifts;
% the durations of the shifts
```

```minizinc
array[Locations] of int: durations;
array[Workers,Locations] of var int: dur_arr;
% maximum time that durations can take
int: max_duration = max(durations);

% set of holidays
set of int: holidays;

% the times where an employee can work
array[Locations,Days] of set of Workers: worker_shifts;
% the solution variable - the assigned shifts for an employee
array[Locations,Days] of var set of Workers: assign;

% which houses are favouritable?
set of int: favable_houses;
% favourite houses per employee
array[Workers] of set of Houses: fav_houses;

% helper statistics
var int: shift_num;
var int: asg_num;


%%%
% hard constraints
%%%

% h0: only work when you have time (not needed in model)
constraint forall (w in Workers, l in Locations, d in Days) (( w in assign[l,d] ->
w in worker_shifts[l,d]) );
% h1: everybody has to work at least one shift
constraint forall (w in Workers) (sum([ w in assign[l,d] | l in Locations, d in
Days ]) > 0 \/ num_shifts[w] == 0);
% h2: not more than possible for each employees
constraint forall (w in Workers) (sum([ w in assign[l,d] | l in LocsWOStandby, d in
 Days ]) <= num_shifts[w]);
% h3: not more than maximum hours
constraint forall (w in Workers, l in Locations) (dur_arr[w,l] = (sum([ w in assign
[l,d] | d in Days]) + sum([ w in assign[l,d] | d in holidays])) * durations[l]);
constraint forall (w in Workers) (sum([ dur_arr[w,l] | l in Locations]) <=
max_hours*60);
% h4: not more employees than stated in requirements
constraint forall (l in Locations, d in Days) (sum([ w in assign[l,d] | w in
Workers ]) <= requirements[l,d]);
% h5: employees with few shifts cannot be assigned to reserve or standby
constraint forall (w in Workers) ( num_shifts[w] < sr_minshift -> (sum([ w in
assign[sr,d] | d in Days, sr in {index_standby,index_reserve}]) == 0));
% h6: cannot surpass max reserve shifts
constraint forall (w in Workers)  (sum([w in assign[index_reserve,d] | d in Days])
<= r_maxshift);
% h7: cannot surpass max standby shifts
constraint forall (w in Workers)  (sum([w in assign[index_standby,d]  | d in Days])
 <= s_maxshift);
% h8: time - no two shifts on the same day
constraint forall (w in Workers, d in Days, l1 in Locations, l2 in Locations) (l1
== l2 \/ ((w in assign[l1,d]) -> ({w} intersect assign[l2,d] == {})));
% h9: no unassigned reserve or standby shifts
constraint forall (l in {index_standby,index_reserve}, d in Days) (sum([w in assign
[l,d] | w in Workers]) == requirements[l,d]);

%%%
% soft constraints
%%%
```

```
% helpers
array[Workers] of var int: numbers_of_assigned_shifts;
constraint forall (w in Workers) (numbers_of_assigned_shifts[w] = sum([w in assign[
l,d] | l in Locations, d in Days]));
array[Workers] of var float: assigned_shift_hours;
constraint forall (w in Workers) (assigned_shift_hours[w] = sum([(w in assign[l,d])
*durations[l]/60.0 | l in Locations, d in Days]));
array[Workers] of var float: assigned_floating_shift_hours;
constraint forall (w in Workers) (assigned_floating_shift_hours[w] = sum([(w in
assign[index_reserve,d])*durations[index_reserve]/60.0 | d in Days]) + sum([(w in
assign[index_standby,d])*durations[index_standby]/60.0 | d in Days]) );

% s1: minimize mse of shift shortage
array[Locations,Days] of var 0.0..1.01: shift_shortage_arr;
constraint forall (l in Locations, d in Days) (if requirements[l,d] > 0 then
shift_shortage_arr[l,d] = 1.0 - card(assign[l,d])/requirements[l,d] else
shift_shortage_arr[l,d] = 0.0 endif);
var 0.0..1.01: shift_shortage_mse;
var int: number_of_non_empty_shifts = sum([requirements[l,d] > 0 | l in Locations,
d in Days]);
constraint shift_shortage_mse = 1/number_of_non_empty_shifts*sum([
shift_shortage_arr[l,d]*shift_shortage_arr[l,d] | l in Locations, d in Days]);

% s2 and s3: shift fairness among employees
array[Workers] of var 0.0..1.01: workers_desired_shifts_satisfaction;
constraint forall (w in Workers) (if num_shifts[w] > 0 then
workers_desired_shifts_satisfaction[w] = 0.00001+assigned_shift_hours[w]/(
num_shifts[w]*max_duration/60.0) else workers_desired_shifts_satisfaction[w] = 0
endif);
var 0.0..1.01: workers_desired_shifts_satisfaction_mean;
constraint workers_desired_shifts_satisfaction_mean = 1.0/workers*sum([
workers_desired_shifts_satisfaction[w] | w in Workers]);
var 0.0..1.01: workers_desired_shifts_satisfaction_variance;
constraint workers_desired_shifts_satisfaction_variance = 1.0/workers * sum([(
workers_desired_shifts_satisfaction[w] - workers_desired_shifts_satisfaction_mean)
*(workers_desired_shifts_satisfaction[w] -
workers_desired_shifts_satisfaction_mean) | w in Workers]);

array[Workers] of var 0.0..1.01: workers_floating_shift_hours_fractions;
constraint forall (w in Workers) (if assigned_shift_hours[w] > 0 then
workers_floating_shift_hours_fractions[w] = assigned_floating_shift_hours[w]/
assigned_shift_hours[w] else workers_floating_shift_hours_fractions[w] = 0.0 endif
);
var 0.0..1.01: workers_floating_shift_hours_fractions_mean;
constraint workers_floating_shift_hours_fractions_mean = 1.0/workers*sum([
workers_floating_shift_hours_fractions[w] | w in Workers]);
var 0.0..1.01: workers_floating_shift_hours_fractions_variance = 0.00001+1.0/
workers * sum([(workers_floating_shift_hours_fractions[w] -
workers_floating_shift_hours_fractions_mean)*(
workers_floating_shift_hours_fractions[w] -
workers_floating_shift_hours_fractions_mean) | w in Workers]);

% s4: house preferences
array[Workers] of var 0.0..1.01: workers_unpreferred_shifts_ratio;
constraint forall (w in Workers) (if numbers_of_assigned_shifts[w] > 0 then
workers_unpreferred_shifts_ratio[w] = sum([w in assign[l,d] | l in Locations, d in
 Days where (l in (favable_houses diff fav_houses[w]))])/
numbers_of_assigned_shifts[w] else workers_unpreferred_shifts_ratio[w] = 0.0 endif
);
var 0.0..1.01: workers_unpreferred_shifts_ratio_mse;
constraint workers_unpreferred_shifts_ratio_mse = 1.0/workers * sum([
workers_unpreferred_shifts_ratio[w]*workers_unpreferred_shifts_ratio[w] | w in
```

```minizinc
Workers]);


%%%
% computation of values and statistics
%%%
% statistic: number of assigned assignments
constraint asg_num = sum([card(assign[l,d]) | l in Locations, d in Days]);
% statistic: number of total shifts that should have been assigned
constraint shift_num = sum([num_shifts[w] | w in Workers]);

array[1..4] of var float: g_arr_unweighted;
constraint g_arr_unweighted = [shift_shortage_mse,
workers_desired_shifts_satisfaction_variance,
workers_floating_shift_hours_fractions_variance,
workers_unpreferred_shifts_ratio_mse];
array[1..4] of var float: g_arr = [10*shift_shortage_mse,
workers_desired_shifts_satisfaction_variance,
workers_floating_shift_hours_fractions_variance,
workers_unpreferred_shifts_ratio_mse];
var float: g_ges = sum(g_arr);

solve ::seq_search([
set_search([assign[index_standby,d] | d in Days],first_fail,indomain,complete),
set_search([assign[l,d] | l in Locations, d in Days],first_fail,indomain,complete)
])

%satisfy;
minimize (g_ges);

%%%
% output
%%%

output [
"asg_num = " ++ show(asg_num) ++ "\n" ++
"shift_num = " ++ show(shift_num) ++ "\n" ++
" unweighted = "++ show(g_arr_unweighted) ++ "\n" ++
" ges = "++ show(g_ges) ++": " ++ show(g_arr) ++ "\n" ++
""];
```

List of Figures

List of Tables

Bibliography

[1] Gecode, 2020. URL https://www.gecode.org/.

[2] Ruibin Bai, Edmund K. Burke, Graham Kendall, Jingpeng Li, and Barry McCollum. A hybrid evolutionary approach to the nurse rostering problem. *IEEE Transactions on Evolutionary Computation*, 14(4):580–590, 2010.

[3] Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys*, 35(3):268–308, 2003.

[4] Peter Brucker, Rong Qu, and Edmund Burke. Personnel scheduling: Models and complexity. *European Journal of Operational Research*, 210(3):467–473, 2011.

[5] Edmund K. Burke, Patrick De Causmaecker, and Greet Vanden Berghe. A hybrid tabu search algorithm for the nurse rostering problem. In *Asia-Pacific Conference on Simulated Evolution and Learning*, volume 1585 of *LNCS*, pages 187–194. Springer, 1999.

[6] Edmund K. Burke, Patrick De Causmaecker, Greet Vanden Berghe, and Hendrik Van Landeghem. The state of the art of nurse rostering. *Journal of Scheduling*, 7(6): 441–499, 2004.

[7] Sara Ceschia, Nguyen Dang, Patrick De Causmaecker, Stefaan Haspeslagh, and Andrea Schaerf. The second international nurse rostering competition. *Annals of Operations Research*, 274(1-2):171–186, 2019.

[8] Tim B Cooper and Jeffrey H Kingston. The complexity of timetable construction problems. In *International Conference on the Practice and Theory of Automated Timetabling*, volume 1153 of *LNCS*, pages 281–295. Springer, 1995.

[9] Marco Dorigo and Gianni Di Caro. Ant colony optimization: a new meta-heuristic. In *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99*, volume 2, pages 1470–1477. IEEE, 1999.

[10] Nikolaus Frohner, Stephan Teuschl, and Günther R Raidl. Casual employee scheduling with constraint programming and metaheuristics. In *International Conference on*

Computer Aided Systems Theory, volume 12013 of *LNCS*, pages 279–287. Springer, 2019.

[11] Michael R Garey and David S. Johnson. Complexity results for multiprocessor scheduling under resource constraints. *SIAM Journal on Computing*, 4(4):397–411, 1975.

[12] Fred Glover. Tabu search—part i. *ORSA Journal on Computing*, 1(3):190–206, 1989.

[13] Walter J Gutjahr and Marion S Rauner. An ACO algorithm for a dynamic regional nurse-scheduling problem in austria. *Computers & Operations Research*, 34(3): 642–666, 2007.

[14] Pierre Hansen and Nenad Mladenović. Variable neighborhood search. In *Handbook of Metaheuristics*, pages 145–184. Springer, 2003.

[15] Pierre Hansen, Nenad Mladenović, and Dragan Urošević. Variable neighborhood search and local branching. *Computers & Operations Research*, 33(10):3034–3045, 2006.

[16] Ghaith M Jaradat, Anas Al-Badareen, Masri Ayob, Mutasem Al-Smadi, Ibrahim Al-Marashdeh, Mahmoud Ash-Shuqran, and Eyas Al-Odat. Hybrid elitist-ant system for nurse-rostering problem. *Journal of King Saud University-Computer and Information Sciences*, 31(3):378–384, 2019.

[17] Richard M Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Springer, 1972.

[18] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[19] Bernhard Korte, Jens Vygen, B Korte, and J Vygen. *Combinatorial Optimization*, volume 2. Springer, 2012.

[20] Vipin Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI Magazine*, 13(1):32–32, 1992.

[21] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Thomas Stützle, and Mauro Birattari. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58, 2016.

[22] Alan K Mackworth. Consistency in networks of relations. *Artificial intelligence*, 8 (1):99–118, 1977.

[23] Nenad Mladenović and Pierre Hansen. Variable neighborhood search. *Computers & Operations Research*, 24(11):1097–1100, 1997.

[24] Nicholas Nethercote, Peter J Stuckey, Ralph Becket, Sebastian Brand, Gregory J Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In *International Conference on Principles and Practice of Constraint Programming*, volume 4741 of *LNCS*, pages 529–543. Springer, 2007.

[25] Christos H Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity.* Courier Corporation, 1998.

[26] Rong Qu and Fang He. A hybrid constraint programming approach for nurse rostering problems. In *International Conference on Innovative Techniques and Applications of Artificial Intelligence*, pages 211–224. Springer, 2008.

[27] Erfan Rahimian, Kerem Akartunalı, and John Levine. A hybrid integer programming and variable neighbourhood search algorithm to solve nurse rostering problems. *European Journal of Operational Research*, 258(2):411–423, 2017.

[28] Günther R Raidl, Jakob Puchinger, and Christian Blum. Metaheuristic hybrids. In *Handbook of Metaheuristics*, pages 385–417. Springer, 2019.

[29] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of Constraint Programming.* Elsevier, 2006.

[30] Martin Stølevik, Tomas Eric Nordlander, Atle Riise, and Helle Frøyseth. A hybrid approach for solving real-world nurse rostering problems. In *International Conference on Principles and Practice of Constraint Programming*, volume 6876 of *LNCS*, pages 85–99. Springer, 2011.

[31] Ioannis X Tassopoulos, Ioannis P Solos, and Grigorios N Beligiannis. A two-phase adaptive variable neighborhood approach for nurse rostering. *Computers & Operations Research*, 60:150–169, 2015.

[32] Jorne Van den Bergh, Jeroen Beliën, Philippe De Bruecker, Erik Demeulemeester, and Liesje De Boeck. Personnel scheduling: A literature review. *European Journal of Operational Research*, 226(3):367–385, 2013.