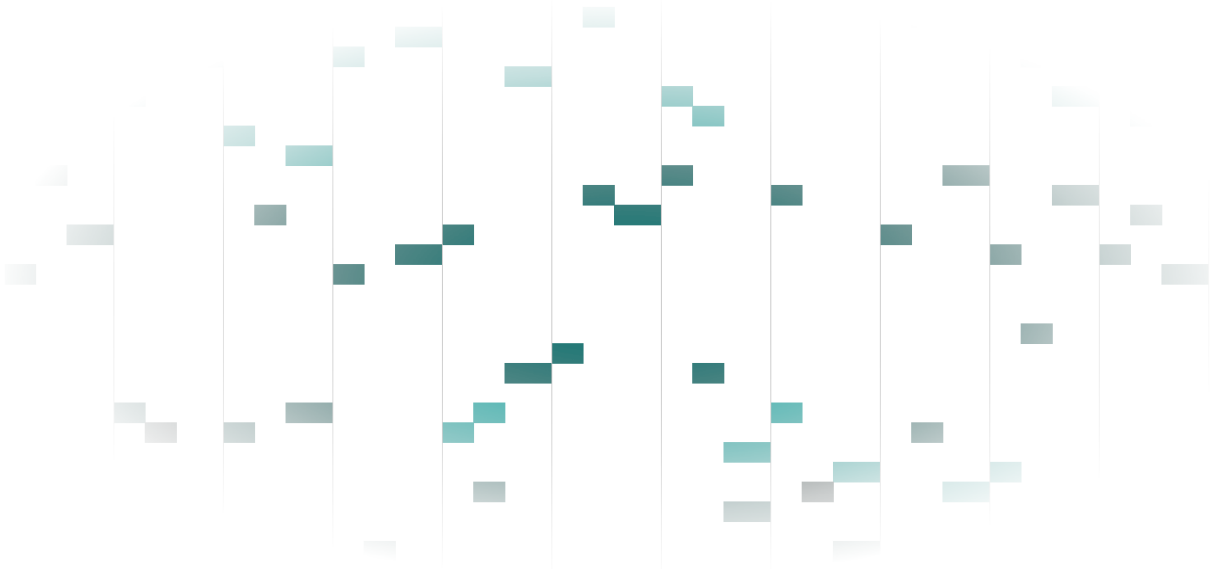


Automatic Shift Scheduling for Healthcare Personnel using Satisfiability Modulo Theory

Master's thesis in Systems, Control and Mechatronics

ALVIN COMBRINK
STEPHIE DO

DEPARTMENT OF ELECTRICAL ENGINEERING

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2021
www.chalmers.se

Master's thesis 2021

Automatic Shift Scheduling for Healthcare Personnel using Satisfiability Modulo Theory

ALVIN COMBRINK
STEPHIE DO

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering
Division of Automation
Chalmers University of Technology
Gothenburg, Sweden 2021

Automatic Shift Scheduling for Healthcare Personnel using Satisfiability Modulo Theory
ALVIN COMBRINK
STEPHIE DO

Supervisor: Kristofer Bengtsson, Associate Professor, Department of Electrical Engineering
Supervisor: Sabino Francesco Roselli, PhD Student, Department of Electrical Engineering
Examiner: Knut Åkesson, Professor, Department of Electrical Engineering

Master's Thesis 2021
Department of Electrical Engineering
Division of Automation
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: An artistic rendition of a shift schedule generated using the developed system.

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2021

Automatic Shift Scheduling for Healthcare Personnel using Satisfiability Modulo Theory

ALVIN COMBRINK
STEPHIE DO
Department of Electrical Engineering
Chalmers University of Technology

Abstract

Shift scheduling involves allocating shifts to healthcare personnel in a way that complies with regulations and preferences. The task is laborious and time-consuming since it is often done manually. This is particularly true in the healthcare sector where many administrative tasks are allocated to qualified health care providers instead of dedicated administrators. With this in mind, a system for automating shift scheduling has been developed in this thesis to be used in healthcare services. To achieve this, a mathematical model is created that consists of 10 logically formulated constraints that are able to model a wide range of requirements. Using two independent optimization methods based on Satisfiability Modulo Theory (SMT) and Genetic Algorithms, it is demonstrated that SMT is better suited to handle a problem at this level of complexity under such restricting constraints. With the developed automatic scheduling system, a schedule that satisfies all hard constraints for a representative test case is generated in a considerably short time.

Keywords: Shift Scheduling, Nurse Scheduling Problem, Satisfiability Modulo Theory, Genetic Algorithms, Optimization

Acknowledgements

We would first like to express our gratitude for all the invaluable support and guidance provided by the supervisors of this thesis, Associate Professor Kristofer Bengtsson and PhD student Sabino Francesco Roselli of the automation division at Chalmers University of Technology. Without you, we would not have seen the wonderful world of optimization in the same way. Additionally, a special thanks goes out to Doctor Pia Laurin, Tore Vingare and Ingrid Fritzell at Sahlgrenska University Hospital for their insights into the needs in healthcare. Finally, to our friends and family who have supported us throughout our journey at Chalmers, we will optimize your schedules for you.

Alvin Combrink and Stephie Do, Gothenburg, June 2021

Contents

List of Figures

List of Tables

1
Introduction

For most organizations with shift-based personnel schedules, the question of who to assign to what shifts is of great importance. Shift-scheduling, or rostering, plays an essential role in maintaining an organizations ability to meet the demands of its services as it addresses the very fundamental requirement of having the right staff at the right place at the right time. At first glance this may seem trivial, but it has been shown for decades by the scientific community [9][12] that determining how to assign staff to shifts is very seldom a straightforward task.

Rostering is difficult for two main reasons. The first is that there are many constraints and requirements that must be taken into consideration. An organization is typically concerned with financial aspects, such as reducing personnel costs and ensuring that all shifts have qualified staff members assigned. The employees on the other hand have requirements regarding workload, when they are on leave, how fair the distribution of shifts is among the staff, etc. Additionally, external bodies such as governmental agencies and worker unions introduce a whole myriad of other requirements. The second reason is that the complexity of a schedule scales exponentially when increasing the time-frame, number of shifts or number of staff. As such, shift-scheduling problems are often highly complex with an astronomically large number of possible solutions [8].

Despite the difficulty with rostering, organizations are strongly incentivized to invest in efficient shift-scheduling processes as it can bring enormous benefits to the organization as well as to its employees. Aside from the direct financial gains of reducing the time spent on scheduling, an efficient shift-schedule has been shown to increases employee satisfaction, performance and well-being, see for example [10], [13] and [14]. These sometimes unquantifiable effects lead to very tangible long-term rewards for an organization. A well-designed shift schedule is particularly important in industries with work hours outside of traditional business hours, as is common in healthcare, since a disrupted circadian rhythm has shown to bring a wide variety of negative health and social effects [15], which can be mitigated through better scheduling practices [2].

The nature of the shift-scheduling problem can be traced back to many forms, such as “The Nurse Scheduling Problem”(NSP), “The Nurse Rostering Problem”(NRP) or “The Physician Rostering Problem” (PRP). These formulations are similar in nature and are in themselves more complex than simply solving a shift assignment problem due to the fact that nurses and medical staff have different expertise and

must be assigned to different shifts, around the clock, where it is unacceptable to not meet the demands of hospital services through unassigned shifts [9]. They usually include requirements of widely different varieties. It is also common to have non-negotiable requirements that must be satisfied for a schedule to be feasible, known as hard constraints, as well as non-essential objectives that should be fulfilled if possible, known as soft constraints. Different disciplines have addressed the NSP over many decades. There are numerous extensive surveys, such as [9], [8] and [12], that discuss the most prominent methods used to solve the NSP. These include, but are not limited to, mathematical programming, artificial intelligence and meta-heuristic methods. A common conclusion is that many developed solutions are based upon too many simplifications to be useful in a real hospital environment or have simply not been integrated into the day-to-day functions of a hospital, and thus creating a gap between theory and practice and by doing so negating the usefulness of them. It is stated that to make progress within this field, the full range of needs and requirements must be taken into account. Furthermore, it is concluded that no method is by itself sufficient to fully tackle the NSP and therefore, a hybrid solution using both exact and heuristic methods could offer significant improvements.

As stated above, organisation's have much to gain from better scheduling processes and more efficient shift-schedules. This is particularly true for Swedish health care services. A clinical study in 2019 conducted by McKinsey & Company [6] regarding time management and productivity in the Swedish healthcare found that roughly 19% of a doctor's time, or one day per week, is allocated for administrative work. Sweden is amongst the top in the EU when it comes to the number of doctors per capita but falls in the lower percentile when it comes to patient visits. Furthermore, it was found that only 39% of the time is devoted to direct care for patients compared to 66% in the UK. McKinsey concluded that the healthcare system is in need of, among other things, new technology that streamlines the system. Only recently have hospitals started to move away from the time-consuming manual scheduling, referred to as self-scheduling [9], and begun applying automatic systems.

Sahlgrenska University Hospital is currently Sweden's largest hospital and one of the largest in northern Europe. At Queen Silvia Children's Hospital, a part of Sahlgrenska, over 100 doctors at three pediatric wards and an emergency room as well as over a dozen on-call doctors are scheduled by one doctor with over 20 years of experience. The scheduling is done manually approximately every 2 months and takes a significant portion of time that could otherwise be spent on treating patients. Therefore, interest has been raised to develop an automatic shift-scheduling system that is compatible with current work processes and can reduce the time taken from direct patient care.

1.1 Previous Research

The NSP was the subject of attention in the 2010 International Nurse Rostering Competition [16], where a method developed in [28] took first prize in all three categories. The method consists of two phases, one which solves the problem of

assigning workdays to nurses and a second phase for assigning shifts on the days that they are assigned to. The optimization problems in both phases are solved using integer programming and a mix of soft and hard constraints. The results are promising as satisfactory solutions were found in relatively short times. Since the paper's publication, many new methods have sprung up which likely take advantage of the increase in available computing power. Nonetheless, the method shows significant computational efficiency which would be advantageous for the current project.

A study in 2013 [27] introduced constraint programming to a real case of NSP for a few mid-size hospitals in Chile that were working with the uncommon "fourth shift" system. The name can be deceptive as a day is only divided into two shifts, and not four. The two shifts consist of a day shift that starts 08.00 in the morning and ends 20.00 in the evening and a night shift that starts at 20.00 and ends at 08.00. A problem that the paper found with earlier methods was that they were not general enough to be applied in different hospitals which required distinct constraints. With regards to their findings, they propose two models that solve the NSP and were concluded to be easily adaptable for any fourth shift system situation. One of the models were designed using normal logical expressions while the other one was modelled with regular constraints. Regular constraints are used to enforce sequences, shift sequences in this case, to take a specific value according to a finite automaton. The final result showed that both models were efficient and managed to solve the NSP for over 600 workers in roughly 10 seconds.

A more recent method, developed in [30], attempts to solve the NSP using a two-stage heuristic approach. The first stage finds a solution where all hard constraints are met by assigning shifts in chronological order to random staff members, and backtracking is done to ensure that no constraints are violated. The second stage refines the schedule by swapping shift assignments that violate soft constraints while ensuring the hard constraints are still fulfilled. Having two stages is said to greatly reduce the methods computational complexity. Additionally, the method is developed under the pretext that it should be compatible with Excel such that it can be used in practice. It is concluded that the developed method cannot guarantee optimal solutions, as often is the case with heuristic-based methods, but is able to generate adequate solutions in a much shorter time when compared to exact methods.

Another heuristic-based method by [22] implements a classical genetic algorithm to generate schedules for an actual emergency department at a Spanish hospital, which has a mix of hard and soft constraints imposed on their schedules. Genetic algorithms are primarily implemented using objective functions which makes the handling of hard and soft constraints less straightforward. The presented method initially generates a population of schedules that satisfy all hard constraints, which are then evolved by a crossover that only crosses whole weeks, after which a repair function mends any hard constraint violations. Therefore, every schedule in every iteration does not violate any hard constraints, making them so-called feasible. This type of evolutionary algorithm that always stays within the bounds of feasibility is

discussed in [23] and [26] where they argue that many optimal solutions lie along the borders of feasibility and as such entails that maintaining a fraction of infeasible solutions within a population leads to more optimal solutions for the population as a whole.

1.2 Purpose

The aim of this project is to develop a working scheduling system that should be able to solve the NSP, tailored for and approved by Sahlgrenska University Hospital. The automatic scheduling systems task is to generate a schedule that fulfils all hard constraints within a reasonable time and is initially for the personnel who are working on-call shifts in the pediatric ward. The on-call consists of two categories, transplantation and backup, and requires different qualifications from the personnel. One positive aspect of self-scheduling is the flexibility that comes with it. As this is seen as a desirable trait, the systems functions should not only be able to find a feasible schedule but also guide during the scheduling process by suggesting changes or checking feasibility after changes have been made. Hence, the program should optimize the scheduling process such that it facilitates the schedulers work.

One requirement is that the developed program must be compatible with the hospital's existing software and routines, otherwise the practical usefulness of the system will likely suffer. The program that will be used to construct the interface is Excel as it is what the hospital staff are most comfortable with. Excel is widely used and provides a solid user-interface platform for individuals with different professional knowledge about computers. As a result, Excel provides the high level of intricacy that is needed for the development as well as a reasonably low level of complexity to increase user-friendliness. The developing stage should therefore consider the interaction between the user and the system by constructing an interface that is intuitive.

1.2.1 Main Research Questions

The project is done in several stages. The first and most essential stage involves developing a mathematical model to represent the optimization problem. This entails creating a system that can treat any number of constraints, of varying forms and importance. In conjunction with this, one or several suitable methods to find satisfactory solutions are developed. The next stage is to develop a user interface between the algorithm and the Excel environment. The interface should be customized for the schedulers comfort. The described steps are applied to Sahlgrenska's on-call schedule and an evaluation will be performed for comparing the developed scheduling methods to find the benefits and disadvantages.

With regards to the objective of the thesis, a few research questions have been constructed

RQ1. *What types of constraints must be included in the mathematical model to*

describe the scheduling optimization problem?
It is essential for a schedule to comply with a wide range of requirements for it to be used in a practical setting. Therefore, a mathematical model must be developed where requirements are described and implemented in the form of constraints on the optimization problem. A large part of doing so involves extracting the underlying constraints from common requirements placed on shift schedules. The goal is to minimize the number of constraints that must be modelled without decreasing the breadth of requirements that they are able to implement. Once the underlying constraints have been determined, they must then be expressed within the mathematical model using modelling languages such as propositional logic or bit-vectors, depending on the optimization method. The optimization method along with the mathematical model determine what types of requirements can be included, as well as the restrictions on computational complexity.

RQ2. *Which optimization method or methods can generate the most satisfactory schedules and are most suitable for the current application?*
The optimization method that is most satisfactory is, in this case, able to find a solution that satisfies the most constraints in a reasonably fast time. Computation time is therefore crucial as it is the biggest limitation and sets the boundaries for what methods that can be used and for what requirements that are possible to implement. Since hard constraints must be fulfilled, all generated schedules that are feasible can be considered equal. Using the soft constraints however, two schedules can be compared by evaluating how well they optimize an objective function. Therefore, if two methods are able to achieve a feasible solution that fulfills all hard constraints, then they can be evaluated by comparing how well they fulfill soft constraints for particular scenarios.

1.3 Scope

The thesis is conducted during a time frame that equals 30 university credits, corresponding to one semester of full-time studies. Due to the limited amount of time, only a few methods are evaluated and implemented. Conclusions can therefore only be drawn based on the chosen methods. Tools employed fall under the classes of constraint programming and stochastic optimization. The programming language Python is used and has been chosen partly because it is the language the authors are most comfortable with but also because it can be made to interact with Excel, which is needed to easily create the user interface. Interaction with the user is created in a way such that little to no previous knowledge is needed to learn how to utilize the tool. As the project is in collaboration with Sahlgrenska University Hospital, the automatic schedule is tailored for their needs, but an effort is made to make the models as general as possible for it to fit other purposes and thereby increase its usefulness.

1.4 Outline

Chapter 2 provides background theory regarding the two optimization methods used, Satisfiability Modulo Theory and Genetic Algorithms. Chapter 3 describes the mathematical model used in the optimization process, which includes definitions of fundamental components, operators and constraints. The use of the mathematical model with the two optimization methods are described in chapters 4 and 5, respectively, which include benchmarks using a test case. Chapter 6 provides information about the test case used in chapters 4 and 5 and presents the resulting schedules using each of the methods. Finally, discussions are provided in chapter 7 and conclusions in chapter 8.

2
Background Theory

The problem is approached using two forms of optimization. The first uses satisfiability modulo theory which falls under the class of constraint programming. In the literature, methods of this class are typically referred to as exact methods. The second is a genetic algorithm under stochastic optimization algorithms. As opposed to exact methods, stochastic methods use random processes to guide the optimization and rely on probabilities of finding satisfactory solutions rather than guaranteeing it. The theories presented below are meant to provide a better understanding of the underlying mechanisms behind the two methods.

2.1 Constraint Programming

Constraint programming (CP) is a class of optimization methods where the solution space to a problem with a large number of candidate solutions is constrained. Typically, the problem is described using decision variables that must be assigned values. Explicit rules on how the values of decision variables may be set simultaneously are referred to as constraints. A problem, with a search space that has been bounded by constraints, is commonly referred to as a constraint satisfaction problem (CSP) and can be solved with CP using a variety of techniques [20].

An example of a problem that can be solved using CP is described as

$$\begin{aligned} x_1 + x_2 &\leq 10 \\ \text{subject to } x_1 + 2 &= x_2 \\ x_1, x_2 &\in \{1, \ldots, 10\}. \end{aligned} \tag{2.1}$$

Here, the decision variables are x_1 and x_2, the constraint is $x_1 + 2 = x_2$ and the goal is to find assign the variables such that $x_1 + x_2 \leq 10$. Even though the search space can involve both x_1 and x_2 taking any values in the domain $\mathcal{D} = \{1, \ldots, 10\}$, the constraint eliminates all solutions where this is not fulfilled, reducing the number of candidate solutions from 100 to 8. The search space is therefore bounded by the constraint.

A constraint can either be *hard* or *soft*. A hard constraint must be satisfied, which thus eliminates certain mappings of values to decision variables, depending on if they fulfill the constraint or not [17]. Too many hard constraints can sometimes make the model unsatisfied, which makes the model *over-constrained*. This happens

when, for instance, two hard constraints contradict each other. In those cases, it is better to make use of soft constraints which are preferred but not a requirement for the model to be satisfiable. In other words, the hard constraints are relaxed to become soft to make room for feasible solutions. It is also possible to assign priorities to soft constraints by introducing weighted soft constraints. When a CSP involves soft constraints, it is referred to as a constraint optimization problem (COP).

2.2 Boolean Satisfaction Problem

Boolean satisfaction problems (SAT) are problems that are modelled with Boolean variables together with propositional logic such as disjunctions, conjunctions and negations. SAT problems are part of CSP but are less general as they are restricted to only including two values, *true* or *false* [19]. The advantage that this simplification brings is that modern SAT-solvers are particularly effective at finding solutions to problems with large numbers of variables in a reasonably short time [24].

A number of terms that are widely used in SAT are briefly defined and explained below [19].

Literal: A variable or its negation (x, $\neg x$).

Clause: A formula or expression formed by several literals.

First-order logic/predicate logic: Extension of propositional logic that also includes quantifiers such as for all $\forall$, and exists $\exists$.

Assignment: A formula, which contains variables and operators, is assigned from a domain D if its variables are mapped to elements in D.

Satisfiability: If a formula evaluates to be *True* under *a* given assignment to its variables, then it is *satisfied*. Otherwise, it is *unsatisfied*.

2.2.1 Normal Forms

Common practice to solve SAT problems begins with transforming the expressions into normal forms that align with the decision procedure, since the procedure normally expects the constraints to be in a certain syntax form [1][19]. Two common logical operators are

$\bigwedge_{\gamma \in \Gamma} \gamma$	:	The formula evaluates to *True* if all Boolean variables γ, $\forall \gamma \in \Gamma$, are true, otherwise *False*.
$\bigvee_{\gamma \in \Gamma} \gamma$	:	The formula evaluates to *True* if any of the Boolean variables γ, $\forall \gamma \in \Gamma$, are true, otherwise *False*.

The operators lay the foundation for the normal forms. There are different types of normal forms whereof the most relevant ones are highlighted.

Disjunctive Normal Form: Disjunctive normal form (DNF), is a formula which is expressed as

$$\bigvee_i (\bigwedge_j x_{i,j})$$

where the variable x is a literal. The expression can also be read as an *OR* of *ANDs*.

Conjunctive Normal Form: Conjunctive normal form (CNF), is a formula which is expressed as

$$\bigwedge_i (\bigvee_j x_{i,j})$$

where the variable x is a literal. It can be seen as the opposite of a disjunctive normal form where it is instead read as an *AND* of *ORs*.

Any formula that consists of Boolean variables can be translated into either CNF or DNF and thereafter solved using various methods. The DNF can, for example, be solved by looking at each *AND* clause, check the satisfiability and if one is satisfiable then the expression is also satisfiable, but this approach is very inefficient as it causes the number of clauses to explode exponentially [3].

2.2.2 Conflict-Driven Clause Learning

Modern SAT-solvers can be divided into two categories depending on their solver mechanism. The first underlying method makes use of backtracking through a binary tree, also known as *Conflict-Driven Clause Learning* (CDCL) [19]. The other method uses *stochastic search* for an assignment that evaluates as *false*. In most cases, CDCL based solvers are considered more advantageous compared to the stochastic search. The reason is that CDCL-based solvers are complete, while stochastic search is not. CDCL can simply be explained as an algorithm that makes a decision regarding a variable, either by propagating the variables implications or, if a conflict emerges, perform backtracking. The algorithm for the CDCL-SAT can be seen in algorithm 1.

Input: A propositional CNF formula
Output: Unsatisfiable if the formula is not satisfiable else Satisfiable

```
function: CDCL;
while true do
    while BCP() = "conflict" do
        BacktrackLevel := AnalyzeConflict();
        if BacktrackLevel < 0 then
            return "Unsatisfiable";
        end
        Backtrack(BacktrackLevel) ;
    end
    if not Decide() then
        return "Satisfiable";
    end
end
```

Algorithm 1: CDCL-SAT algorithm [19].

In algorithm 1, *Decide()* is a function that assigns *true* to a chosen unassigned variable and outputs false if all variables have been assigned. *BCP()* is short for Boolean Constraint Propagation and is a function that apply the *unit clause rule* on the formula. A unit is a clause that is not satisfied and all but one literal have been assigned. The unit clause rule is used for finding variables that must be assigned a certain Boolean variable [3]. The rule is defined according to [19] as

> "Given a partial assignment under which a clause becomes unit, it must be extended so that it satisfies the unassigned literal of this clause"

If the *BCP()* function finds a conflict, which is when literals in a formula have been assigned and the formula is not satisfiable, then *AnalyzeConflict()* will be called.

AnalyzeConflict() function is mainly responsible for backtracking. Backtracking is commonly used for solving CSP and SAT problems [1]. Generally for Boolean variables, the backtracking process starts by the root of a binary tree and descends further down as the process continues. The nodes of the tree represent, in this case, a SAT problem and the leaves represent that the SAT problem is either satisfiable or not. CDCL uses a special type of backtracking, mainly based on conflict clauses. The CDCL traverses through the tree to find where the conflicts arose, and adds a formula that contains information about the conflict, which is how the solver learns. The solver continues to backtrack until a feasible assignment has been found for all formulas, including the new ones, or until it proves to be unsatisfiable [24]. *Backtrack()* is a function that gets rid of assignments that are of a higher decision level(depth of the backtracking tree) than the current one [19].

2.3 Satisfiability Modulo Theory

SAT formulations restrict the domain of how a problem can be expressed since they only allow for propositional logic. The level of expressiveness is therefore relatively low, but it brings with it the advantage that modern SAT-solvers can solve large propositional logic problems very fast, something that researchers wanted to take advantage of [4]. This is where Satisfiability Modulo Theory (SMT) comes in [19]. SMT evolved to include a wide range of first-order theories beyond propositional logic, such as arrays and bit-vectors which are useful to model certain problems, allowing for more freedom and expressiveness. The tools that are used for solving SMT problems are called SMT-solvers and are commonly based on SAT-solvers.

A *theory* in SMT deals with models that belong to a specific theory T, and not just any arbitrary model [3]. A model that belongs to T defines the interpretation of a set of non-logical symbols (e.g. predicates or constants) and is referred to as *signature* $\sum$. Theories can therefore be regarded as classes ($\mathbf{A}$) of models that share the same signature ($T = (\sum, \mathbf{A})$). Different methods exist depending on what theory is applied. More of the different theories and their solver mechanism can be found in chapter 2.3.1. But before that, two approaches that can be used for solving SMT-problems are presented.

Lazy Approach - The lazy approach is the most prominent approach in SMT. As previously known, converting formulas into DNF and solving them could be tremendously time-consuming [3][19]. Instead, the lazy approach takes advantage of a *theory solver* and *SAT engine* (CDCL SAT-solver) in order to gradually check DNFs only if needed, hence the *lazy*.

Eager Approach - A less common approach where the main feature is to, by encoding, fully reduce the SMT formulation into a SAT formulation and feed it into a SAT-solver [3][19]. Computation-wise, the procedure can in some cases be very expensive. The reduction to SAT can be done in only one step which means that the SAT-solver only needs to be run once to give a result.

2.3.1 Z3 Theorem Solver

One of the modern open-source SMT-solvers is called Z3 Theorem Prover. Z3 uses a newer version of CDCL, namely *Davis-Putnam-Loveland-Logemann* (DPLL) and was externally released by Microsoft in 2007 [11]. Z3 was recently benchmarked against two other SMT-solvers, Gurobi and OptiMathSat, and outperformed both when it came to solving a Job Shop Scheduling problem, but it was noted by the authors that previous studies found equal performance by Z3 and OptiMathSat [25]. The DPLL mechanism can sometimes prove to be limiting as the SAT-solvers and theory solvers are treated like black-boxes [4]. A short overview over solvers used for different modelling strategies is presented here.

2.3.1.1 Arithmetic

Z3 utilizes a variety of solvers depending on the arithmetical constraint that is used. A table summarizing the arithmetic theories as well as their solvers can be found in [7] and is presented below

Table 2.1: Arithmetic logical theories and solvers used in Z3 [7].

Logic	Solver	Example
Linear Real Arithmetic	Dual Simplex	$x + \frac{1}{2}$ y ≤ 3
Linear Integer Arithmetic	Cuts + Branch	$a + 3b \leq 3$
Mixed/Real Integer	Cuts + Branch	$x + a \geq 4$
Integer Difference Logic	Floyd-Warshall	$a - b \leq 4$
Real Difference Logic	Bellman-Ford	$x - y \leq 4$
Unit Two-variable Inequalities	Bellman-Ford	$x + y \leq 4$
Polynomial Real Arithmetic	Model-based CAD	$x^2 + y^2 \leq 1$
Non-linear Integer Arithmetic	CAD + Branch + Linearization	$a^2 = 2$

In table 2.1, a, b are integers and x, y are real values. Z3 is by default set to make use of infinite precision arithmetic and can thus generate a very precise output. The drawback with this is that computation time is in some cases sacrificed [7].

The simplex method is one of the most widely used methods for solving arithmetic linear programming (LP) problems [21]. The LP problem is usually written in its standard form

$$\begin{aligned} \min\, & c^T x \\ \text{subject to } & Ax = b \\ & x \geq 0 \end{aligned} \tag{2.2}$$

where A is an $m \times n$-matrix, c and x are $n \times 1$-vectors and b is a $m \times 1$-vector where $b \geq 0$. The standard form can always be written in its dual form and vice versa [21], which is expressed as

$$\begin{aligned} \max\, & b^T y \\ \text{subject to } & A^T y = c \\ & y \geq 0 \end{aligned} \tag{2.3}$$

The Dual form can sometimes be easier to solve than the standard form. If one of them yields a feasible solution, then that solution is the same for both the forms. Inequalities in the constraints can be rewritten as equalities by introducing *surplus* and *slack* variables. The standard form can thereafter be assembled into a single matrix called a *tableau* to make the simplex computation easier since it allows for matrix manipulation.

$$\begin{bmatrix} A & b \\ c^T & 0 \end{bmatrix} \tag{2.4}$$

Simplex iteratively finds a minimum optimal value using the the tableaux together with algebraic operations.

2.3.1.2 Propositional Logic

Z3 can accept different formulas as input, such as one defined by propositional logic [7]. For example, if the following logic is to be solved

$$\neg(Rain \wedge Sunny) \tag{2.5}$$

then in Z3, it would be modelled in the Python language as

```
from z3 import *

Rain, Sunny = Bools('Rain Sunny')
s = Solver()
s.add(Not(And(Rain, Sunny)))
print(s.check())
print(s.model())
```

which returns

```
sat
[Rain = True, Sunny = False]
```

The first output, `s.check`, returns that the model is `sat`, meaning that there exists a feasible solution to the problem [7]. The second output, generated by `s.model()`, returns the Boolean decision variables and their assignment that enables the verdict sat. The output is just one of several possible solutions, the outputs

```
[Rain = False, Sunny = True]
[Rain = False, Sunny = False]
```

would in this case also fulfill the constraint and therefore would be feasible solutions as well.

Microsoft released a complementary part to Z3 during 2016, called *vZ* which enables the possibility to solve SMT optimization problems [11]. The two mechanisms behind are the *OptSMT* and *MaxSMT* engines. The first engine, OptSMT is added as a primal phase to the Simplex core, dedicated to finding the optimal value for arithmetic expressions. The second engine MaxSMT, takes advantage of several different *MaxSAT* solvers, whose purpose is to try to solve as many soft constraints as possible [24]. A variety of solvers are applied that are able to tackle the optimization problem in different ways. For example, the MaxSAT solver WMax invokes

penalties if the tracked soft constraint is unsatisfiable [11]. Z3 is therefore able to take on soft constraints (`s.add_soft()`), and turn the problem into an optimization modulo theory (OMT) problem that aims to optimize an objective function. The weights are automatically set to 1 for all constraints if nothing else is specified.

Another interesting function is, for instance, the `s.unsat_core()` which returns the clause of conjunctions that are unsatisfiable within a Boolean formula, also known as the unsatisfiable core. Other operations include `s.pop()` and `s.push()` used for reverting and creating formulas under a scope, which are useful for similar problems that share constraints [5].

2.4 Genetic Algorithms

The rise of cheap and readily available computational power has affected countless areas of science, and optimization is no exception. Alongside the classical areas of optimization, there have grown numerous disciplines that are able to harness the power that computers have brought with them. Stochastic optimization is one such discipline and is specialized in solving optimization problems by using stochastic processes. Within stochastic optimization are evolutionary algorithms with its largest subgroup, genetic algorithms.

As the name implies, genetic algorithms take their inspiration from the processes that have given rise to, and in a sense optimized to their environments, all living species on our planet. They attempt to mimic reproduction, mutation and natural selection, but instead of applying these processes on biological creatures, the subjects are candidate solutions to a problem. A population consisting of candidate solutions are bred, mutated, and selected based on how well they optimize the problem compared to the others. Those that do well are, probabilistically, chosen to be included in the next generations population. Over generations, the population is moved toward better performing solutions until the best solution is deemed satisfactory.

Genetic algorithms can optimize black box problems, where the underlying system is unknown or difficult to model. This is because the algorithms do not concern themselves with the system itself but instead only the output and how each candidate solution compares to other candidate solutions. That is not to say that problem-specific information cannot be taken advantage of though. Genetic algorithms are appropriate when the search space is large and when optimal solutions are not necessary. The very use of randomness implies that finding a global optimum is not guaranteed, and since candidate solutions are only taken in relation to each other, genetic algorithms are susceptible to getting stuck in local optima. For a more comprehensive explanation on genetic algorithms, see for example [29].

2.4.1 A Standard Genetic Algorithm

Genetic algorithms typically have the same general structure. Starting with the candidate solutions, each one consists of the encoded information of their input to the optimization problem. These are synonymously referred to as genomes or chromosomes. For example, if the function $f(x, y)$ is to be minimized, then a candidate solution must contain the information regarding its input values x' and y'. This can be straight-forward, such as where the genome is simply $[x', y']$, or more complicated where it must be run through a decoding process to extract x' and y'. There are advantages and disadvantages with different types of encodings, usually related to the mutation and crossover processes described below.

The next important part of a genetic algorithm is the objective function, also referred to as the fitness function. Each chromosome must be compared to the other chromosomes, and for that to happen they must each get a fitness value that represents how well they optimize the problem. In simple cases where the problem function outputs a single value, it can be used as the fitness value. However, this is not always the case. For example, if a genetic algorithm is used to find the settings for an autonomous vehicle that should drive a certain distance as fast as possible without crashing, then the problem function can be said to output two values, the time t that it took to drive and a variable γ which is 0 if it crashed and 1 if it did not. A suitable objective function may then be γ/t such that all settings that lead to a crash get the fitness of 0 and all others get a value that increases as the time reduces. This demonstrates that the problem function is not always possible to use as the objective function, but it is important to have an objective function that can be used to compare candidate solutions.

Once all chromosomes in the population have been evaluated using the objective function, the process of selection begins. Selection entails selecting chromosomes that shall be included in the population used in the next generation. Therefore, selection is performed with a bias toward the chromosome with higher fitness values. Common procedures used to do this are presented in section 2.4.2. Chromosome that have been selected are then subjected to random mutations, which alter the genes in the chromosome in some way. The mutations are typically confined to only a few genes at a time and done with a certain probability. Mutations are described in more detail in section 2.4.3. Next comes the simulation of reproduction between two chromosomes, called crossover. This entails trading genes between two chromosomes and is described in section 2.4.4. A last step is taken to ensure that the best candidate solution is carried over to the next population to not destroy or lose it through mutations and crossover, called elitism and is described in 2.4.5. Finally, at this point the new population should be complete, upon which the process is repeated in the next generation. After a certain number of generations or until some termination criteria has been reached, the process is halted and the candidate solution that produces the highest fitness value is selected as the final solution.

2.4.2 Selection

Selection refers to the process of selecting candidate solutions in the current generations population to create the population that will be used in the next generation. This is done using the fitness values such that the solutions with higher values are more likely to be selected. The two most common selection procedures are tournament selection and roulette wheel selection. Let f_i be the fitness value for candidate solution i in the population $\mathcal{P}$.

Tournament selection involves simulating tournaments between randomly selected chromosomes by selecting n_t random chromosomes from the population and in order of descending fitness values select the best with a probability of p_t. If none are selected, then select the one with the lowest fitness score. The procedure is shown in algorithm 2. Typical values for the parameters are $n_t < 10$ and $0.5 \leq p_t \leq 0.9$, but this is entirely situation specific.

Let $\mathcal{T}$ contain n_t randomly selected chromosomes in $\mathcal{P}$

while $1 < |\mathcal{T}|$ **do**
  $r \sim U(0, 1)$;
  if $r < p_t$ **then**
    Select the chromosome in $\mathcal{T}$ with the highest fitness value;
  else
    Remove the chromosome with the highest fitness value from $\mathcal{T}$;
  end
end
Since no chromosome has been selected, return the only one left in $\mathcal{T}$;

Algorithm 2: Tournament selection.

Roulette wheel selection is performed by selecting chromosomes with a probability proportional to its fitness value. This is akin to a roulette wheel where each slice represents a chromosome, and the arc length of the slice is proportional to the fitness value. A randomly selected point on the wheel perimeter will select the chromosome of the corresponding slice. Therefore, the higher a chromosomes fitness score is in relation to the others, the more likely it is to be selected. Each chromosome i is given a cumulative relative fitness value,

$$\phi_i = \frac{\sum_{j=1}^{i} f_j}{\sum_{j=1}^{|P|} f_j}, \quad i = 1, \ldots, |P|, \tag{2.6}$$

and the chromosome with the smallest i that satisfies

$$r < \phi_i \tag{2.7}$$

where $r \sim U(0, 1)$ is selected. As opposed to tournament selection, roulette wheel selection does not require any parameters.

2.4.3 Mutation

Mutations play an important role in genetic algorithms as they contribute with new genetic material. A mutation seldom benefits the individual chromosome being mutated, but in the long term allows for more exploration of the search space. Mutations are typically done gradually, it is motivated in [29] that the best probability of mutating a gene is n_m/m where $n_m = 1$ is the desired average number of mutations per chromosome and m is the size of the chromosome, meaning there is on average 1 mutation per chromosome. Note that the decision to mutate a gene is taken with a probability independent of other genes, and therefore it is possible for the number of genes that are mutated to fall between 0 and m per chromosome. The manner in which a mutation is carried out depends entirely on the form of the chromosome's genes. If the genes are real numbers, then it is common to use a real-number creep mutation where the new value is sampled from a distribution centered at the old value. If instead Booleans are used, then a mutation simply switches value to the opposite Boolean value.

2.4.4 Crossover

The crossover process is taken from natures ability to combine the genomes of two individuals to produce offspring. Crossover can be performed with one or more crossover points. Figure 2.1 shows an example of two chromosomes with Boolean genes. Crossover with a single point is done by selection a random point along the chromosomes and trading all the genes after it between the two chromosomes, seen in figure 2.2. Similarly, two-point crossover is performed by selecting two random points along the chromosome and trading all the genes in between, seen in figure 2.3. Any number of crossover points can of course be used.

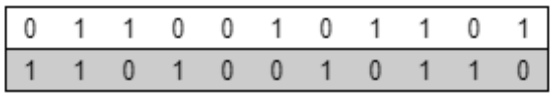


Figure 2.1: Two chromosomes with Boolean genes.

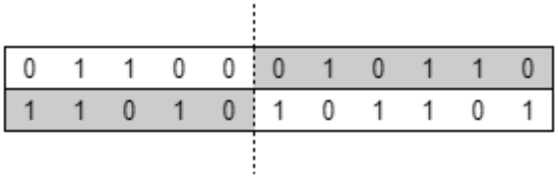


Figure 2.2: Two chromosomes with Boolean genes where a single-point crossover is performed.

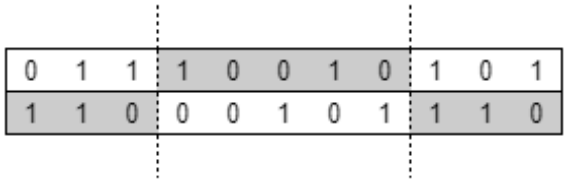


Figure 2.3: Two chromosomes with Boolean genes where two-point crossover is performed.

Crossover is performed between two chromosomes with a certain probability p_c. If the probability is set too high, it can have negative effects on the diversity of the population. A particularly successful chromosome spreads quickly within a population through crossover, and therefore increases the risk of the optimization getting stuck in a local optimum. Due to this, it is in some cases beneficial to not use crossover at all [29].

2.4.5 Elitism

The last common process found in genetic algorithms is elitism. As a population is built using the previous populations chromosomes, mutations and crossover are performed and potentially destroy the best performing chromosomes. Therefore, at every generation, the n best performing chromosomes are copied, unedited, over to the next generations population. This has the advantage of a non-decreasing best chromosome fitness over generations and ensures that the final chromosome that is selected as the solution is in fact the best chromosome over all generations.

3
Mathematical Model

The mathematical model is an important part of the optimization problem as it sets the framework from within the optimization methods operate. This includes a mathematical representation of a schedule, the staff members, the shifts and all other parameters related to them. All constraints that dictate aspects of a feasible schedule are also part of the model. Therefore, the mathematical model captures all information relevant to solving the optimization problem as well as the interactions between them.

3.1 Fundamentals

Staff

A staff member p is a tuple defined as

$$p = \left(p^{dw}, p^{\mathcal{Q}}\right) \tag{3.1}$$

where p^{dw} is the desired workload and $p^{\mathcal{Q}}$ is the set of qualifications. The set $\mathcal{P}$ contains all staff members such that $p \in \mathcal{P}$, and $\mathcal{Q}$ is the set of all staff member qualifications such that $p^{\mathcal{Q}} \subseteq \mathcal{Q}$. Note that the uppercase superscript denotes that $p^{\mathcal{Q}}$ is also set, as a staff member may possess several qualifications.

Each staff member has a desired workload p^{dw}, measured as a fraction of some unit of workload per time. A common form of workload per time unit is the number of hours per week, and each staff member then works a certain fraction of it. It will be shown in section 3.3.9 that the unit does not matter, but instead the relative values between staff members. Each staff member also has a set of qualifications $p^{\mathcal{Q}}$ that defines what types of shifts that they are qualified to be assigned to.

Shifts

A shift s is a tuple defined as

$$s = \left(s^{t}, s^{sd}, s^{\mathcal{QR}}, s^{b}, s^{w}\right) \tag{3.2}$$

where s^{t} is the shift type, s^{sd} is the start day, s^{d} is the duration, $s^{\mathcal{QR}}$ is the set of qualification requirements where $s^{\mathcal{QR}} \subseteq \mathcal{Q}$, s^{b} is the burden and s^{w} is the workload. The set $\mathcal{S}$ contains all shifts such that $s \in \mathcal{S}$.

A shift is defined as a period of time where specific duties must be carried out by the staff members that have been assigned to it. Each shift has a defined shift type s^t, for example a *Cashier* or *Transplantation on-call*, which can be used to define various shift groups. Note that a shift only has one shift type and that they are simply used as a way to group shifts. Therefore, the shift types of two shifts must be identical for them to be considered to have the same type. For example, a shift with the type *Cashier Afternoon* does not have the same shift type as a shift with *Cashier Night*, even though they both entail having the same duties. In addition to the shift type, a shift also has a set of qualification requirements $s^{\mathcal{QR}}$ that is used to assess if a staff member is qualified to perform the duties of the shift.

Each shift has its own workload s^w, which is used to balance the amount of work each staff member is assigned to. It initially makes sense to define a shifts workload as its duration s^d as then all staff members that work an equal workload also work an equal amount of time. In most cases that would suffice, but if the scheduler would want to differentiate the burden between certain shift types, such that an hour of one type is easier than an hour of another, then a more suitable workload unit would be the time spent working a shift multiplied by its burden s^b, where the burden of a shift type is taken in relation to other shift type burdens. Thus, a shifts workload is $s^w = s^d s^b$.

Shift Sharing

It is possible for any number of staff members to be assigned to a single shift, meaning that they share it. Whether shift sharing is allowed or not is up to the scheduler to decide. However, there are two approaches to take when designing a system to be able to handle situations where sharing is or is not allowed. Either it can be assumed that all shifts may be shared by any staff members and thereafter have a framework for the scheduler to convey which shifts and staff members that are not allowed to share, or conversely assume by default that all shifts may only be assigned one person and then allow for the scheduler to defines specific shifts and staff members that may share. The latter approach is taken here as it is assumed that shifts will in most cases be intended for one person and therefore reduce the amount of needed input from the scheduler.

There are two things that must be defined, the set of shifts that may be shared and the set of staff members that may share them. It is important to note that the set of staff members defines the exact combination of staff members that are allowed to share the shifts. In other words, it is not allowed for a subset of the set of staff members to share the shifts, only *all* of the staff members in the set are allowed to share the shifts. Additionally, it can be considered less burdensome for a shift to be shared than working the shift alone, and therefore a counted workload must also be defined. The counted workload is a factor that the shift workload is multiplied by before being added to the staff members personal workload. For example, the counted workload may be 1/2 meaning that the shifts workload is halved for each staff member sharing it. This makes intuitive sense if two people are sharing the shift as the burden is half of what it would be if working alone. Another example

to demonstrate this is that if staff members A, B and C were to share 6 shifts and the counted workload is set to 1/3, then each staff member will been assigned a workload equivalent to working 2 shifts by themselves.

Let a tuple d, referred to as a shift-sharing, be defined as

$$d = \left(d^{\mathcal{S}}, d^{\mathcal{P}}, d^{cw}\right) \tag{3.3}$$

where $d^{\mathcal{S}}$ is the set of shifts, $d^{\mathcal{P}}$ is the set of staff members and d^{cw} is the counted workload for all the shifts in $d^{\mathcal{S}}$ being shared by staff in $d^{\mathcal{P}}$. Let the set $\mathcal{D}$ be the set of all shift-sharings such that $d \in \mathcal{D}$.

Overlapping Shifts

Overlapping shifts are when more than one shift occur at the same time. This means that it is possible for a staff member to be assigned to overlapping shifts and therefore have to perform the duties of those shifts at the same time. Let o, referred to as an overlapping shift combination, be a tuple defined as

$$o = \left(o^{\mathcal{S}}, o^{\mathcal{P}}\right) \tag{3.4}$$

where $o^{\mathcal{S}}$ is a set of shifts that overlap and $o^{\mathcal{P}}$ is a set of staff members that are allowed to be assigned to *all* shifts in $o^{\mathcal{S}}$. Let the set $\mathcal{O}$ contain all overlapping shift combinations such that $o \in \mathcal{O}$. Consequently, there must exist an overlapping shift combination for every possible combination of overlapping shifts. For example, if a schedule has 3 shifts that occur at the same time, then there are 4 possible overlapping shift combinations (3 pairs and one with all three shifts together). It is possible to discard all overlapping shift combinations where $o^{\mathcal{P}}$ is empty in order to reduce the size of $\mathcal{O}$, but that would mean that the calculations to find overlapping shifts will need to be re-performed when evaluating if any staff member is assigned to overlapping shifts that no one is allowed to be. Another way to reduce the size of $\mathcal{O}$ is to discard all overlapping shift combinations where *all* staff members are allowed to be assigned, i.e. all o where $o^{\mathcal{P}} = \mathcal{P}$, as it is irrelevant if and who is assigned to the shifts in the shift combination.

Decision Variables

The decision variables are defined as

$$a_{p,s} = \begin{cases} True & \text{if shift } s \text{ is assigned to staff member } p \\ False & \text{otherwise.} \end{cases} \tag{3.5}$$

There is subsequently one decision variable for each staff member and shift combination, and are all contained in the $|\mathcal{P}| \times |\mathcal{S}|$-matrix A, referred to as the assignment matrix. Neither the rows of A, corresponding to staff members, nor the columns, corresponding to shifts, are necessarily in any order.

3.2 Operators

A number of Boolean operators are used to define the constraints in section 3.3 and are defined as

$\bigwedge_{\omega\in\Omega}(\omega)$: The *and* operator is true if all Boolean variables ω, $\forall\omega \in \Omega$ are *true*, otherwise *false*.

$\bigvee_{\omega\in\Omega}(\omega)$: The *or* operator is true if any Boolean variable ω, $\forall\omega \in \Omega$ is *true*, otherwise *false*.

$\underset{\omega\in\Omega}{\mathrm{ALN}}(\omega, n)$: The *at least* n operator is true if at least n of the Boolean variables ω, $\forall\omega \in \Omega$ are true, otherwise false.

$\underset{\omega\in\Omega}{\mathrm{AMN}}(\omega, n)$: The *at most* n operator is true if at most n of the Boolean variables ω, $\forall\omega \in \Omega$ are true, otherwise false.

Additionally, Iverson brackets [18] are used to map Boolean variables to integers such that

$$[\omega] = \begin{cases} 1 & \text{if } \omega \text{ is true} \\ 0 & \text{otherwise.} \end{cases} \tag{3.6}$$

3.3 Constraints

For a schedule to have any form of practical usefulness, it must satisfy certain requirements specific to the context in which it will be used. These requirements may stem from needs, limitations, regulations and personal preferences, and may take many forms and address many different aspects of a schedule. Examples include how many consecutive days a staff member may work before given leave or whether a shift may be shared by multiple staff members, etc. Requirements vary depending on the context, a hospital has other needs than a retail store, but also over time as the duties of the workplace changes, staff rotate or develop and laws and regulations are updated. Therefore, the list of requirements that the schedule must comply with can be regarded as a variable. Creating a system that is not able to adapt to and account for changing requirements would within short be rendered useless.

Requirements are defined as rules that the schedule must obey, but so far, they are simply abstract ideas in the mind of the scheduler. For the scheduling system to implement the requirements, they must be described in a clear and logical way. This is done through what is defined here as constraints. A constraint takes certain input and compiles the set of logical rules that the schedules decision variables must obey for the requirement to be fulfilled.

The difference between a requirement and a constraint can either be subtle or stark, depending on the level of generalization or abstractness that the constraint has been designed with. A constraint with virtually no abstractness is therefore the same as the requirement that it attempts to implement, while a constraint with a high de-

gree of abstractness can implement the requirement but is not locked to the values of it. For example, let a requirement on the schedule be that *all staff members may work a maximum of three days in a row*. Starting at an almost non-existent level of abstractness, the constraint is defined exactly as the requirement. A step toward generalizing the constraint could be to let the maximum number of consecutive days be variable, or let the number of consecutive days have a minimum and maximum variable. Now the constraint is defined as *all staff members may work a minimum of x and maximum of y days in a row*. A next step could be to vary the staff that the constraint applies to such that all or only certain staff have restrictions on the number of consecutive days. In the same way as varying the staff, the type of shifts can also be variable. This gives the constraint *all defined staff members may work a minimum of x and maximum of y days in a row of the defined shifts*. The constraint without any generalization can only be used in one way, to limit the number of consecutive working days to three for all staff. The generalized constraint on the other hand can do the same and much more, for example ensure that "Doctors Sigurd and Adam" work a minimum of two consecutive days of shifts at "Ward A or B" on weekdays.

As can be seen from the example above, the more general a constraint becomes, the more types of requirements it can describe. This has several advantages. First, the scheduler is given the ability to implement more types of requirements and therefore increase the quality of the generated schedules. Second, fewer types of constraints need to be implemented in the scheduling system which could offer more potential to optimize calculations and reduce the total calculation time. Finally, and perhaps most importantly, the scheduling system can better cope with different configurations of requirements and therefore be used in many more applications and over a longer time. Although, these advantages depend on the degree of generalization to which the constraints are defined. This raises the question of how abstract constraints should be designed, if there are any disadvantages with increasing their abstractness.

The clearest disadvantage of a higher abstractness is the accompanying higher complexity. From the scheduler's perspective, a more complex constraint requires more understanding of the logic behind it as well as the interactions between the defined components and therefore is harder to use. The increased functionality gained by more generalization is only useful if the scheduler is able to take advantage of it. A constraint that is too complex can be simplified for the scheduler by a user interface that presents its various uses separately, but implements them in the scheduling system as two instances of the constraint. For example, the user interface can show one constraint that limits the workload for all staff members and another that limits the workload of a specific staff member, while both are fed as two instances of the same constraint with variably defined staff members to the scheduling system. From the perspective of the scheduling systems performance, constraints with low levels of generalization and complexity offer more chances to take advantage of their structure to optimize calculations. This is of course specific to the optimization method and constraint in question and is not necessarily always true.

In order to design a system with a high degree of generality, so that it can cope with the ever-changing requirements, the constraints are designed with a high level of abstractness. By doing so, more freedom is given to the scheduler to tailor each constraint and thereby obtain a higher quality schedule. Constraints are constructed using boilerplate form so that the values can be inputted by the scheduler. An example of a boilerplate, taken from the constraint described in section 3.3.1, looks like

> The number of shifts in ____ that are not assigned at least one staff member in ____ must fall between ____ and ____.

For example, if the requirement is that *All shifts must be assigned at least one staff member*, then the boilerplate can be used to construct a constraint that ensures this. The scheduler may then fill in *All*, *All*, *0* and *0* in the blank spaces to get

> The number of shifts in *All* that are not assigned at least one staff member in *All* must fall between *0* and *0*.

This would create one instance of the constraint corresponding to this particular boilerplate. The same boilerplate can then be copied and filled in with other inputs to create multiple versions of the same type of constraint. To demonstrate, another copy of the same boilerplate can take the inputs *Week 1*, *Helge*, *2* and *7* to implement the requirement that *Helge must be free from* 2 *to* 7 *shifts in week* 1, to give

> The number of shifts in *Week 1* that are not assigned at least one staff member in *Helge* must fall between *2* and *7*.

In addition to being able to create multiple copies of a constraint, each constraint must also be designated as either hard or soft. If a constraint is hard, then the schedule must fulfil it. If instead the constraint is soft, then the schedule must try to fulfil it, but if that is not possible then the schedule will still be accepted.

Since schedules in this case are considered to have finite time horizons, the current schedule will most likely come into effect after a previous schedule. Some constraints will depend on how staff members are assigned before the current schedule and therefor will need to take the previous schedule into account. For example, if a requirement is that a staff member should not work two weekends in a row, then the last weekend of the previous schedule must be considered when assigning the first weekend of the current schedule. Therefore, several of the constraints described below include a section describing how to include a previous schedule. Since the previous schedule is constant, there is no need to set conditions on its assignment matrix as there is no way to change it. Therefore, conditions can be pre-computed and applied to the current schedules decision variables directly. For instance, let A^{-1} be the decision matrix for the previous schedule and let the function $f_{p,s}(A^{-1})$ be true if assigning the shift s in the current schedule to staff member p will violate a constraint. It is then more efficient to apply the constraint $\neg a_{p,s}$ directly instead

of applying $f_{p,s}(A^{-1}) \rightarrow \neg a_{p,s}$ since it is already determined that $f_{p,s}(A^{-1})$ is true.

3.3.1 Assignment of Shifts

It can be argued that the most fundamental requirement to place on a schedule is with regards to how many shifts that must be assigned a staff member. After all, the whole point of creating a shift-schedule is to determine what shifts to assign to who. For self-scheduling, this is obvious, but in the context of optimization this must be defined explicitly or else a feasible schedule may be found where shifts are not assigned to staff. In most cases the goal is likely for all shifts to be assigned to someone, but for the sake of generality it opens more possibilities for the scheduler to decide the degree to which the shifts are assigned. In some situations, it is possible that the scheduler would like to give the optimizer more freedom by allowing a certain tolerance of unassigned shifts. This could be the case when, for example, scheduling employee training sessions where it is not necessary for all sessions to be assigned someone, especially if it interferes with other important shifts. Therefore, the *Assignment of Shifts* constraint is defined as

> The number of shifts in $\tilde{\mathcal{S}}$ that are not assigned at least one staff member in $\tilde{\mathcal{P}}$ must be greater or equal to x and less than or equal to y.

where the scheduler defines which staff members are in $\tilde{\mathcal{P}} \subseteq \mathcal{P}$, which shifts are in $\tilde{\mathcal{S}} \subseteq \mathcal{S}$ as well as the values of x and y. The reason for constraining the number of unassigned shifts instead of the number of assigned shifts is because in most cases, assumingly, all shifts must be assigned. Therefore it is easier to define $x = y = 0$ instead of calculating the total number of shifts and setting x and y equal to that. Note that it would be equally viable to allow for the scheduler to define a minimum and maximum in terms of a percentage of the total number of shifts in $\tilde{\mathcal{S}}$ and thereafter calculate the equivalent values for x and y. Therefore, the choice of defining x and y as the limits on the number of unassigned shifts is done without loss of generality.

The constraint is divided into two parts and is mathematically formulated as

$$\underset{s \in \tilde{\mathcal{S}}}{\mathrm{ALN}} \left(\bigwedge_{p \in \tilde{\mathcal{P}}} (\neg a_{p,s}), x \right) \tag{3.7}$$

$$\underset{s \in \tilde{\mathcal{S}}}{\mathrm{AMN}} \left(\bigwedge_{p \in \tilde{\mathcal{P}}} (\neg a_{p,s}), y \right) \tag{3.8}$$

The formulations above read as "the number of shifts in $\tilde{\mathcal{S}}$, where no staff members are assigned to it, must be at least x (eq. 3.7) and at most y (eq. 3.8)".

3.3.2 Shared Shifts

Shift sharing is addressed in section 3.1 and relates to what staff members that are allowed to share what shifts, the *Shared Shifts* constraint applies these definitions.

If a shift is assigned to multiple staff members, then there must be a shift-sharing d such that the shift is included in its set of shifts $d^{\mathcal{S}}$ and the set of assigned staff members are equal to its set of staff members $d^{\mathcal{P}}$, otherwise the shift is not allowed to be shared by the combination of staff members. This defines which assignments of shifts to multiple people are valid. Like the *Assignment of Shifts* constraint in section 3.3.1, the scheduler may wish to define at which degree the optimizer is constrained to only assigning valid groups of staff members. It is possible that there is some leniency by allowing for some shifts to be shared by undefined groups of staff members. Therefore, the *Shared Shifts* constraint is defined as

> The number of invalid staff groups assigned to shifts in $\tilde{\mathcal{S}}$ must be greater or equal to x and less than or equal to y.

The set of shifts $\tilde{\mathcal{S}} \subseteq \mathcal{S}$, as well as the limits x and y are defined by the scheduler. The constraint is divided into two parts and is mathematically formulated as

$$\underset{s\in\tilde{\mathcal{S}}}{\text{ALN}}\left(\underset{p\in\mathcal{P}}{\text{ALN}}\left(a_{p,s},2\right)\rightarrow\neg\bigvee_{d\in\mathcal{D}}\left(s\in d^{\mathcal{S}}\wedge\{p\in\mathcal{P}|a_{p,s}\}=d^{\mathcal{P}}\right),x\right) \tag{3.9}$$

$$\underset{s\in\tilde{\mathcal{S}}}{\text{AMN}}\left(\underset{p\in\mathcal{P}}{\text{ALN}}\left(a_{p,s},2\right)\rightarrow\neg\bigvee_{d\in\mathcal{D}}\left(s\in d^{\mathcal{S}}\wedge\{p\in\mathcal{P}|a_{p,s}\}=d^{\mathcal{P}}\right),y\right) \tag{3.10}$$

The formulation above reads as "if at least two staff members have been assigned to a shift, then there is not a shift-sharing where the shift is in the set of shifts and the assigned staff members are equal to the set of staff members. The number of shifts in $\tilde{\mathcal{S}}$ that this is fulfilled for must be at least x (eq. 3.9) and at most y (eq. 3.10)."

3.3.3 Qualified Assignments

Shifts have certain requirements regarding the skills that the assigned staff members must have, to be able to perform the duties of it. Therefore, each shift has a set of qualification requirements $s^{\mathcal{QR}}$ that must be a subset of the assigned staff members qualifications set $p^{\mathcal{Q}}$. If this is not fulfilled for any staff member and shift, the shift is then said to have been assigned unqualified staff members. The degree of how strictly the shifts must be assigned qualified staff members can be decided by the scheduler, since it is possible in some cases that a certain number of shifts do not necessarily need qualified staff. This is perhaps not so common in hospital settings but more in retail and service industries. It is assumed that in most cases all shifts must be assigned qualified staff members, which makes it more practical to define limits on how many shifts that may *not* have qualified staff assignments as opposed to how many that *should* since the limits can then simply be set to 0 for all staff members to be qualified for the shifts that they have been assigned to. Therefore, the *Qualified Assignments* constraint is defined as

> The number of shifts in $\tilde{\mathcal{S}}$ that have been assigned an unqualified staff member in $\tilde{\mathcal{P}}$ must be greater or equal to x and less than or equal to y.

The set of shifts $\tilde{\mathcal{S}} \subseteq \mathcal{S}$ and set of staff members $\tilde{\mathcal{P}} \subseteq \mathcal{P}$ is defined by the scheduler, as well as the values for x and y. The constraint is divided into two parts and is mathematically formulated as

$$\underset{s \in \tilde{\mathcal{S}}}{\text{ALN}} \left(\bigvee_{p \in \tilde{\mathcal{P}}} \left(a_{p,s} \wedge \left(s^{\mathcal{QR}} \not\subseteq p^{\mathcal{Q}} \right) \right), x \right) \tag{3.11}$$

$$\underset{s \in \tilde{\mathcal{S}}}{\text{AMN}} \left(\bigvee_{p \in \tilde{\mathcal{P}}} \left(a_{p,s} \wedge \left(s^{\mathcal{QR}} \not\subseteq p^{\mathcal{Q}} \right) \right), y \right) \tag{3.12}$$

The formulation above reads as "the number of shifts in $\tilde{\mathcal{S}}$, where any staff member in $\tilde{\mathcal{P}}$ is assigned but the shifts qualifications requirements set is not a subset of their qualifications set, must be between x (3.11) and y (3.12)."

3.3.4 Overlapping Shifts

Overlapping shifts refer to shifts that occur simultaneously, and has been addressed in section 3.1. An overlapping shift combination refers to a combination of shifts that overlap in time. For each overlapping shift combination o with the set of shifts $o^{\mathcal{S}}$ and set of allowed staff members $o^{\mathcal{P}}$, if a staff member is assigned to all the shifts in $o^{\mathcal{S}}$ but is not in $o^{\mathcal{P}}$, then the assignment is not allowed. The *Overlapping Shifts* constraint is defined as

> The number of times staff in $\tilde{\mathcal{P}}$ are assigned to overlapping shifts, that they are not allowed to be assigned to, must be greater or equal to x and less than or equal to y.

where the staff set $\tilde{\mathcal{P}} \subseteq \mathcal{P}$ and limits x and y are defined by the scheduler. The mathematical formulation for this constraint is divided into two parts and is defined as

$$\sum_{o \in \mathcal{O}} \sum_{p \in \tilde{\mathcal{P}} \setminus o^{\mathcal{P}}} \left[\bigwedge_{s \in o^{\mathcal{S}}} (a_{p,s}) \right] \geq x \tag{3.13}$$

$$\sum_{o \in \mathcal{O}} \sum_{p \in \tilde{\mathcal{P}} \setminus o^{\mathcal{P}}} \left[\bigwedge_{s \in o^{\mathcal{S}}} (a_{p,s}) \right] \leq y \tag{3.14}$$

where Iverson brackets are used. The formulation above is read as "the number of times a staff member in $\tilde{\mathcal{P}}$, who is not in an overlapping shift combinations o's set of allowed staff members $o^{\mathcal{P}}$, is assigned to all shifts in $o^{\mathcal{S}}$ must be equal to or larger than x (eq. 3.13) and equal to or smaller than y (eq. 3.14)."

3.3.5 Staff Combination Assignments

The *Staff Combination Assignments* constraint enables constraining the number of shifts that a specific combination of staff members are assigned to, and is defined as

> The number of shifts in $\tilde{\mathcal{S}}$ that are assigned to all staff members in $\tilde{\mathcal{P}}$, and only them, must be greater or equal to x and less than or equal to y.

The set of shifts $\tilde{\mathcal{S}} \subseteq \mathcal{S}$, set of staff members $\tilde{\mathcal{P}} \subseteq \mathcal{P}$ and limits x and y are defined by the scheduler. This constraint limits the number of shifts that have been assigned to an exact combination of staff members, as opposed to the *Assignment of Shifts* in 3.3.1 where any staff members in the staff set will do. Therefore, for an assignment to be counted here, all staff members in $\tilde{\mathcal{P}}$ and no one else must be assigned to it. This constraint is useful in many ways. The most obvious use is to limit the number of shifts that a specific group of people are assigned to, which could be useful when allowing for certain shifts to be shared but not wanting sharing to occur too often. Another use could be to restrict a staff member from being assigned to certain shifts without sharing. For example, if there is only one person in $\tilde{\mathcal{P}}$ and $x = y = 0$, then that person will not be assigned to any shifts in $\tilde{\mathcal{S}}$ unless sharing it with someone else.

The mathematical formulation of this constraint is

$$\underset{s \in \tilde{\mathcal{S}}}{\text{ALN}} \left(\{p \in \mathcal{P} | a_{p,s}\} = \tilde{\mathcal{P}}, x \right) \tag{3.15}$$

$$\underset{s \in \tilde{\mathcal{S}}}{\text{AMN}} \left(\{p \in \mathcal{P} | a_{p,s}\} = \tilde{\mathcal{P}}, y \right) \tag{3.16}$$

and is read as "the number of shifts in $\tilde{\mathcal{S}}$, where all staff members in $\tilde{\mathcal{P}}$ are assigned and all that are not in $\tilde{\mathcal{P}}$ are not assigned, must be between or equal to x (eq. 3.15) and y (eq. 3.16)."

3.3.6 Consecutive Days

It is common in workplaces to assign a staff member to shifts on multiple consecutive days as it helps build routines for the staff. However, the number of consecutive days that a staff member may work, and the number of days of rest after, are usually regulated by workplace policies as well as the law. The constraint *Consecutive Days* addresses these requirements and is defined as

> The number of consecutive days of shifts in $\tilde{\mathcal{S}}$ for a staff member in $\tilde{\mathcal{P}}$ must be greater or equal to x and less than or equal to y.

There are several ways that this constraint may be utilized. One way is to simply limit the number of consecutive days of work assigned to the staff by including all shifts and setting x and y accordingly. Another way is to ensure that staff work certain weekday patterns. For example, one pattern could be $2 - 3 - 2$, meaning a staff member is either assigned to Monday-Tuesday, Wednesday-Thursday-Friday or Saturday-Sunday. The constraint would then be used to limit the minimum and maximum number of consecutive days of shifts on, for instance, Mondays and Tuesdays to 2 and 2, meaning that the only allowed assignment is to work on both days.

The mathematical formulation of this constraint is divided into two parts, one part that deals with the minimum number of consecutive days and the other with the maximum consecutive days. The mathematical formulation for the minimum number of consecutive days is

$$\left(a_{p,s} \wedge \bigwedge_{s' \in \tilde{\mathcal{S}}} \left(s'^{sd} = s^{sd} - 1 \rightarrow \neg a_{p,s'}\right)\right) \rightarrow \bigwedge_{k \in \{1,..,x-1\}} \left(\bigvee_{s' \in \tilde{\mathcal{S}}} \left(a_{p,s'} \wedge s'^{sd} = s^{sd} + k\right)\right)$$

$$\forall p \in \tilde{\mathcal{P}} \quad \forall s \in \tilde{\mathcal{S}} \tag{3.17}$$

and is read as "if a staff member p is assigned to a shift s and not assigned to any shift s' that occurs on the day before, then they should be assigned to at least one shift on each of the $x-1$ following days". For the maximum number of consecutive days, the mathematical formulation is

$$\left(a_{p,s} \wedge \bigwedge_{s' \in \tilde{\mathcal{S}}} \left(s'^{sd} = s^{sd} - 1 \rightarrow \neg a_{p,s'}\right)\right) \rightarrow \bigvee_{k \in \{1,...,y\}} \left(\bigwedge_{s' \in \tilde{\mathcal{S}}} \left(s'^{sd} = s^{sd} + k \rightarrow \neg a_{p,s'}\right)\right)$$

$$\forall p \in \tilde{\mathcal{P}} \quad \forall s \in \tilde{\mathcal{S}} \tag{3.18}$$

The formulation above reads as "if a staff member p is assigned to a shift s and not assigned to any shift on the day before, then there must exist a day on the following y days where they are not assigned to any shifts." The number of days a staff member is assigned to in a period of $y+1$ consecutive days is thus at most y, meaning that they do not exceed the maximum number of consecutive days.

Consecutive Days with Previous Schedule

The previous schedule only needs to be taken into account if there is no gap between the previous schedule's end date and the new schedules start date, since if there is then it would not be possible to create a consecutive work series starting in the previous schedule and ending in the current. Assuming that there is no gap between the schedules, let the set $\tilde{\mathcal{S}}^{-1}$ contain all shifts in the previous schedule that fulfil the same criteria as when defining $\tilde{\mathcal{S}}$. For every staff member p, let l_p be the length of the consecutive work series of shifts in $\tilde{\mathcal{S}}^{-1}$ that ends on the last day of the previous schedule. If none such exists, $l_p = 0$. To ensure that the minimum and maximum consecutive length is achieved, the mathematical formulations are

$$\bigwedge_{\tilde{l} \in \{1,...,x-l_p\}} \left(\bigvee_{s \in \tilde{\mathcal{S}}} \left(s^{sd} = F + \tilde{l} - 1 \rightarrow a_{p,s}\right)\right) \qquad \forall p \in \tilde{\mathcal{P}} \text{ if } l_p \neq 0 \text{ and } l_p < x \tag{3.19}$$

$$\bigvee_{\tilde{l} \in \{1,...,y-l_p+1\}} \left(\neg \bigvee_{s \in \tilde{\mathcal{S}}} \left(s^{sd} = F + \tilde{l} - 1 \rightarrow a_{p,s}\right)\right) \qquad \forall p \in \tilde{\mathcal{P}} \text{ if } l_p \neq 0 \text{ and } l_p \leq y \tag{3.20}$$

where F is the start date for the current schedule. Eq. 3.19 reads as "For every day $\tilde{l}$ after the last day in the previous schedule, there must be a shift that starts on that day that staff member p is assigned to." and ensures that the minimum consecutive length is achieved. Eq. 3.20 ensures that the consecutive length is shorter or equal to y and reads as "at least one of the following $y - l_p + 1$ must be without any assignments of shifts in $\tilde{\mathcal{S}}$." If the length of consecutive workdays at the end of the previous schedule already exceeds the maximum length y, then the following constraint is added instead

$$\bigwedge_{s \in \tilde{\mathcal{S}}} \left(s^{sd} = F \rightarrow \neg a_{p,s} \right) \qquad \forall p \in \tilde{\mathcal{P}} \text{ if } y < l_p \tag{3.21}$$

and read as "No shifts in $\tilde{\mathcal{S}}$ that start on the first day of the schedule may be assigned to staff member p".

3.3.7 Before and After Consecutive Days

As addressed in the *Consecutive Days* constraint in section 3.3.6, it is common in workplaces to limit the number of consecutive working days, but another common requirement dictates the number of off-days that must follow. Without such requirements, it would be possible to assign staff members to multiple back-to-back consecutive working days with only a single day off between them. Therefore, the *Before and After Consecutive Days* constraint implements these requirements.

A reasonable assumption to make is that the number of off-days following consecutive workdays depends on both the number of consecutive days as well as the type of shifts. For instance, the common work week consists of five consecutive days of work and is followed by two days off. If instead and employee were to work more than five days in a row, it would be reasonable for them to be eligible to more days off afterward. Similarly, if instead the shifts had a longer duration, common in hospitals, than perhaps more days off are needed to recover.

Similarly, requirements may not only concern the days following consecutive workdays but also the preceding days. If a consecutive work series is anticipated to be demanding on the staff member assigned, it could be desired to assign a number of days off before. Again, the number of days off could depend on both the number and type of shifts worked consecutively.

Consecutive workdays may not need to be followed or preceded by days off, just days without certain types of shifts. Adding this distinction enables a larger number of requirements to be taken into account. For example, it could be a requirement to not assign a staff member to too many consecutive days of one specific type of shift, but still be allowed to work other types of shifts before or after. This defines the constraint as

> For a staff member in $\tilde{\mathcal{P}}$, x to y consecutive days of shifts in $\tilde{\mathcal{S}}$ must be preceded by n days without shifts in $\tilde{\mathcal{S}}_n$ and followed by m days without shifts in $\tilde{\mathcal{S}}_m$.

The set of staff members $\tilde{\mathcal{P}} \subseteq \mathcal{P}$, sets of shifts $\tilde{\mathcal{S}}, \tilde{\mathcal{S}}_n, \tilde{\mathcal{S}}_m \subseteq \mathcal{S}$ and the values for x, y, n and m are defined by the scheduler. Note that if a consecutive work series is longer than y, then the constraint no longer applies.

The mathematical formulation is defined as

$$\left(a_{p,s} \wedge \bigwedge_{s' \in \bar{\mathcal{S}}_1} (\neg a_{p,s'}) \wedge \bigwedge_{\tilde{c} \in \{1,\dots,c-1\}} \left(\bigvee_{s' \in \bar{\mathcal{S}}_2} (a_{p,s'})\right) \rightarrow \bigwedge_{s' \in \bar{\mathcal{S}}_3 \cup \bar{\mathcal{S}}_4} (\neg a_{p,s'})\right) \tag{3.22}$$

$$\forall c \in \{x, \dots, y\} \quad \forall s \in \tilde{\mathcal{S}} \quad \forall p \in \tilde{\mathcal{P}}$$

where

$$\bar{\mathcal{S}}_1 = \left\{s' \in \tilde{\mathcal{S}} | s'^{sd} = s^{sd} - 1 \vee s'^{sd} = s^{sd} + c\right\} \tag{3.23}$$

$$\bar{\mathcal{S}}_2 = \left\{s' \in \tilde{\mathcal{S}} | s'^{sd} = s^{sd} + \tilde{c}\right\} \tag{3.24}$$

$$\bar{\mathcal{S}}_3 = \left\{s' \in \tilde{\mathcal{S}}_n | \bigvee_{\tilde{n} \in \{1,\dots,n\}} \left(s'^{sd} = s^{sd} - \tilde{n}\right)\right\} \tag{3.25}$$

$$\bar{\mathcal{S}}_4 = \left\{s' \in \tilde{\mathcal{S}}_m | \bigvee_{\tilde{m} \in \{1,\dots,m\}} \left(s'^{sd} = s^{sd} + c + \tilde{m} - 1\right)\right\} \tag{3.26}$$

The formulation in equation 3.22 reads as "for every staff member in $\tilde{\mathcal{P}}$, shift in $\tilde{\mathcal{S}}$ and possible consecutive shift length $c = x, \dots, y$, if the staff member is assigned to the shift and is not assigned to any shift on the day before or c days after the shift, and is assigned to a shift on all of the following $c - 1$ days, then the staff member is working a consecutive series of c days and therefore may not be assigned to any of the shifts in $\tilde{\mathcal{S}}_n$ on the preceding n days or the shifts in $\tilde{\mathcal{S}}_m$ on the following m days."

Before and After Consecutive Days with a Previous Schedule

If a previous schedule is taken into consideration when determining the current schedule, then additional constraints are used. Let the sets $\tilde{\mathcal{S}}^{-1}$, $\tilde{\mathcal{S}}_n^{-1}$ and $\tilde{\mathcal{S}}_m^{-1}$ include all shifts in the previous schedule that fulfil the same criteria used to define the corresponding sets for the current schedule. There are three ways in which a previous schedule can affect the current schedule. The first is when a consecutive work series of shifts in $\tilde{\mathcal{S}}^{-1}$ was assigned in the previous schedule. Since no shift assignments in the previous schedule can be changed, only shifts in $\tilde{\mathcal{S}}_m$ that start in the current schedule within m days may not be assigned, see figure 3.1. For each staff member p in $\tilde{\mathcal{P}}$, let the length of the last consecutive series of shifts in $\tilde{\mathcal{S}}^{-1}$ be l_p and the day that it ends on be c_p. If $c_p < F - 1$ and $x \leq l_p \leq y$, where F is the start date of the current schedule, then the first $m + c_p - F + 1$ days in the current schedule must be free from all shifts in $\tilde{\mathcal{S}}_m$. The mathematical formulation for this

is

$$\bigwedge_{\tilde{c}\in\{0,\dots,m+c_p-F\}} \left(\bigwedge_{s\in\tilde{\mathcal{S}}_m} \left(s^{sd} = F + \tilde{c} \rightarrow \neg a_{p,s} \right) \right) \tag{3.27}$$

$$\forall p \in \tilde{\mathcal{P}} \text{ if } c_p < F - 1 \text{ and } x \leq l_p \leq y$$

and is read as "no shifts in $\tilde{\mathcal{S}}_m$ starting on all of the first $m + c_p - F + 1$ days in the current schedule may be assigned to staff member p."

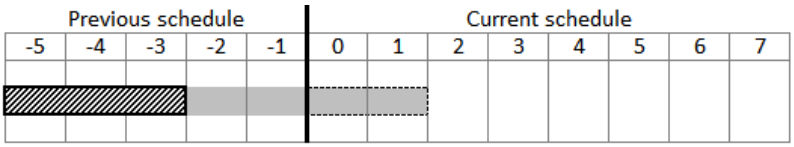


Figure 3.1: 3 consecutive workdays in the previous schedule (diagonally striped) affects the assignments of shifts on the $m = 4$ following days (gray). Only the current schedule can be constrained and therefore only shifts in $\tilde{\mathcal{S}}_m$ on the two first days in the current schedule (dashed border) is ensured to not be assigned.

The second way for a previous schedule to affect the current schedule is when $c_p = F - 1$, as then it is possible to assign shifts in the beginning of the current schedule such that the consecutive shift series in the previous schedule continues into the current schedule. See figure 3.2 for an example. For such an assignment to be allowed, the existing consecutive series in the previous schedule must be shorter than y and there may not be any assignments of shifts in $\tilde{\mathcal{S}}_n^{-1}$ such that $c_p^n + n + l_p \geq F$, where c_p^n is the date of the last shift in $\tilde{\mathcal{S}}_n^{-1}$ that staff member p was assigned to in the previous schedule. Otherwise, the constraint that no shifts in $\tilde{\mathcal{S}}_n^{-1}$ may be assigned n days before the consecutive series will be violated. This give the mathematical formulation

$$\begin{aligned} \left(\bigwedge_{\tilde{c}\in\{0,\dots,c-1\}} \bigvee_{s\in\tilde{\mathcal{S}}} \left(s^{sd} = F + \tilde{c} \rightarrow a_{p,s} \right) \wedge \bigwedge_{s\in\tilde{\mathcal{S}}} \left(s^{sd} = F + c \rightarrow \neg a_{p,s} \right) \right) \\ \rightarrow \bigwedge_{s\in\bar{\mathcal{S}}_m} \left(s \in \bar{\mathcal{S}}_1 \rightarrow \neg a_{p,s} \right) \end{aligned} \tag{3.28}$$

$$\forall c \in \{x - l_p, \dots, y - l_p\} \quad \forall p \in \tilde{\mathcal{P}} \text{ if } c_p = F - 1,\ c_p^n + n + l < F,\ l_p < y$$

where

$$\bar{\mathcal{S}}_1 = \left\{ s \in \tilde{\mathcal{S}} | s^{sd} = c', \quad c' = F + c, \dots, F + c + m - 1 \right\} \tag{3.29}$$

and reads as "if all days from F to $F + c - 1$ have a shift in $\tilde{\mathcal{S}}$ assigned but none on $F + c$, then no shifts in $\tilde{\mathcal{S}}_m$ may be assigned within m days". If instead $c_p = F - 1$, $c_p^n + n + l \geq F$ or $l_p \geq y$, then staff member p may not be assigned to any shifts in

$\tilde{\mathcal{S}}$ on the first day of the current schedule, which has the formulation

$$\bigwedge_{s \in \tilde{\mathcal{S}}} \left(s^{sd} = F \rightarrow \neg a_{p,s} \right) \qquad \forall p \in \tilde{\mathcal{P}} \text{ if } c_p = F - 1 \text{ or } c_p^n + n + l \geq F \text{ or } l_p \geq y. \tag{3.30}$$

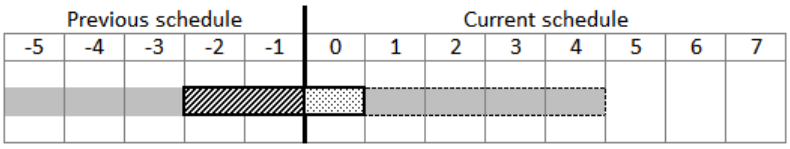


Figure 3.2: 3 consecutive workdays are created if a shift in $\tilde{\mathcal{S}}$ is assigned on the first day of the current schedule (dotted). For it to be valid, no shifts in $\tilde{\mathcal{S}}_m$ may be assigned during the $m = 4$ days after (dashed border) or shifts in $\tilde{\mathcal{S}}_n$ during the $n = 3$ days before (grey without border).

The third and final way that a previous schedule can affect the current schedule is through assignments of shifts in $\tilde{\mathcal{S}}_n^{-1}$ which restrict how early in the current schedule a consecutive shift series may begin. See figure 3.3. For a staff member p, with the last shift in $\tilde{\mathcal{S}}_n^{-1}$ that they are assigned to on c_p^n, no consecutive shift may begin on $c_p + n$ or earlier. This can be ensured by using a window of x days, starting on F and sliding up to $c_p^n + n$, and limiting the number of days with assignments of shifts in $\tilde{\mathcal{S}}$ in the window to $x - 1$. This is provided that $F \leq c_p^n + n$, otherwise the last shift in $\tilde{\mathcal{S}}_n^{-1}$ that was assigned to the staff member occurs too far in the past to affect the current schedule. The mathematical formulation is thus

$$\bigwedge_{\tilde{c} \in \{F, \ldots, c_p^n + n\}} \left(\bigvee_{\tilde{c}' \in \{0, \ldots, x-1\}} \left(\bigwedge_{s \in \tilde{\mathcal{S}}} s^{sd} = \tilde{c} + \tilde{c}' \rightarrow \neg a_{p,s} \right) \right) \qquad \forall p \in \tilde{\mathcal{P}} \text{ if } F \leq c_p^n + n \tag{3.31}$$

which is read as "for every start day $\tilde{c}$ of a sliding window, there must exist a day $\tilde{c}'$ in the sliding window of length x where none of the shifts in $\tilde{\mathcal{S}}$ that start then are assigned to staff member p." This ensures that the maximum length of a consecutive shift in the window is at most $x - 1$.

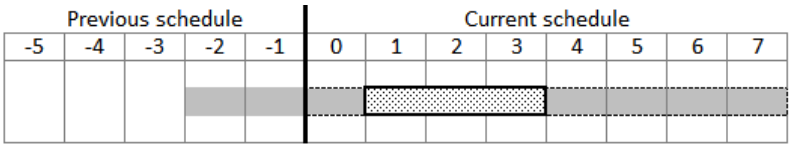


Figure 3.3: For a consecutive work series of shifts in $\tilde{\mathcal{S}}$ to be valid (dotted), no shifts in $\tilde{\mathcal{S}}_n$ or $\tilde{\mathcal{S}}_n^{-1}$ may be assigned within $n = 3$ days before. Only shifts in the current schedule can be constrained (dashed border) while the shifts in the previous schedule (gray without border) cannot.

3.3.8 Assignment Fractions

It is a fundamental requirement to be able to control what types of shifts staff members are assigned to. Aside from ensuring that staff members are not assigned to shifts that they are not qualified for, addressed in the *Qualified Assignments* constraint in section 3.3.3, there are other considerations related to the types of shifts that staff members are assigned. For instance, the scheduler must be able to exclude a staff member from all shifts during a period of time if they are on vacation. On the contrary, the scheduler must also be able to assign specific shifts to staff members. In cases where staff members are qualified for several different types of shifts, there may be a desire to ensure that a certain percentage of their shifts are of either type. Therefore, an abstraction of all of these requirements is that the scheduler must be able to control the fraction of shifts that a staff member is assigned to that is of a specific type. Since shifts may have varying length and burden, the faction used is not in quantity but rather in workload. Therefore, the *Assignment Fractions* constraint is defined as

> The fraction of the workload of shifts that a staff member in $\tilde{\mathcal{P}}$ is assigned to that is in $\tilde{\mathcal{S}}$ must be greater or equal to x and less than or equal to y.

The set of staff members $\tilde{\mathcal{P}}$, the set of shifts $\tilde{\mathcal{S}}$ and limits x and y are defined by the scheduler. The mathematical formulation of this constraint is

$$\sum_{s \in \tilde{\mathcal{S}}} s^w \left[a_{p,s}\right] / \sum_{s \in \mathcal{S}} s^w \left[a_{p,s}\right] \geq x \qquad \forall p \in \tilde{\mathcal{P}} \tag{3.32}$$

$$\sum_{s \in \tilde{\mathcal{S}}} s^w \left[a_{p,s}\right] / \sum_{s \in \mathcal{S}} s^w \left[a_{p,s}\right] \leq y \qquad \forall p \in \tilde{\mathcal{P}} \tag{3.33}$$

3.3.9 Fair Workload

For many workplaces and particularly in hospitals where the demand of services must be satisfied, there is typically a fixed number of staff members and a fixed number of shifts that must be shared between them. As described in 3.1, each shift has a workload equal to the product of its duration and burden. The total workload demand is thus the sum of all shifts workloads and must be distributed to the staff. Each staff member has a desired workload that must be taken into consideration. In most countries the typical workload is around 40 hours per week per employee, however, it is common for employees to deviate from this. Based on this, a workplace has a total desired workload that in most cases will deviate from the demanded workload.

If the total desired workload equals that of the demanded workload, then it would make sense to assign each staff member to the exact workload that they desire, since all staff members will then be satisfied and the demanded workload will be covered. This is unlikely though, and raises the question of how to distribute the demanded workload fairly amongst staff members. Assume that the demanded workload is higher than the total desired workload, one way to distribute the workload could

be to assign every staff member the workload that they desire and then evenly distribute whatever workload that is left. In that way, all staff members work an equal amount above their desired workload. This works well when all staff members have similar desired workloads, but if there is a large variation then those that desire a lower workload will have a higher fractional exceedance. For example, if two staff members have a desired workload of 30 and 60 respectively, and the demanded workload is 150, then the first person will be assigned $30 + 30 = 60$ while the other will be assigned $60 + 30 = 90$. In other words, the first staff member will be assigned 200% of what they desire and the other only 150%. Therefore, a fairer way to distribute the workload would be to do so in relation to each staff members desired workload. In the example above, this would give the first staff member the workload of $30 + 20 = 50$ and the second $60 + 40 = 100$, which leads to both staff members being assigned 167% of their desired workload.

In addition to fairly distributing the workload in the current scheduling period, it may be desired to include the previous scheduling period as well. Each staff member has a desired workload, but since there will almost always be a deviation between the total desired workload for the staff and the demanded workload, each staff member will have a certain fair workload that they should be assigned. However, it will not always be possible to assign exactly the fair workload to every staff member since shifts are discrete and therefore their workloads cannot be split up between staff members. Many other constraints must also be considered other than a fair workload distribution. Therefore, a staff member will often be assigned more or less workload than what is fair. By including the previous scheduled period, someone who was assigned above the workload that is fair may be assigned a lower amount in the current schedule. This defines the *Fair Workload* constraint as

> The normalized deviation between the mean workload ratio and the workload ratio for a staff member in $\tilde{\mathcal{P}}$ should be less than or equal to δ_{max} for assigned shifts in $\tilde{\mathcal{S}}$.

The set of staff members $\tilde{\mathcal{P}}$, the set of shifts $\tilde{\mathcal{S}}$ and limit δ_{max} are defined by the scheduler.

The *Fair Workload* constraint addresses the requirement of fair workloads between staff members in $\tilde{\mathcal{P}} \subseteq \mathcal{P}$ in regards to the shifts in $\tilde{\mathcal{S}} \subseteq \mathcal{S}$. Beginning with a staff member p's total assigned workload, it is defined as

$$w_p = \sum_{s \in \tilde{\mathcal{S}}} s^w \left[a_{p,s} \right] \tag{3.34}$$

and the desired workload is defined as

$$\tilde{w}_p = p^{dw} n \bar{w} \tag{3.35}$$

where $\bar{w}$ is a commonly agreed upon standard value of workload per time unit (e.g. 40 hours per week), n is the length of the scheduled period with the same time unit as $\bar{w}$ and p^{dw} is staff member p's desired fraction of $\bar{w}$. It will be shown below that

$\bar{w}$ cancels out, but is included here for explanatory purposes. An example of this is where the standard measure of workload per time unit is 40 hours of work per week and the total period to be scheduled is 10 weeks, then someone who works at a 50% pace has a desired workload of 200 hours during the period. After a schedule has been established for a period, each staff member will have a ratio of scheduled workload over desired workload, $w_p/\tilde{w}_p$, which should ideally be equal to 1 as the staff member is then given the exact workload that they desire. This is not always possible as the total workload that must be scheduled for the period is not likely to equal the total desired workload of all the staff members. If the workplace is understaffed, then the average workload ratio will be higher than 1 and the staff will be working more than desired. Therefore, the workload ratio can be regarded as a form of satisfaction measure which should ideally be equal for all staff members, as they would then be equally satisfied, or unsatisfied, with their scheduled workload. Another aspect to take into consideration is the staff members scheduled and desired workloads during the previous period. If a staff member has a higher workload ratio during the previous period than the average, then it would be more suitable for them to have a lower ratio during the current period and by doing so even out the total workload ratio. Therefore, the workload ratio is defined as

$$\gamma_p = \frac{w_p^{-1} + w_p^0}{\tilde{w}_p^{-1} + \tilde{w}_p^0} = \frac{w_p^{-1} + w_p^0}{\bar{w}\left(p^{dw,-1}n^{-1} + p^{dw,0}n^0\right)} \tag{3.36}$$

where the superscripts -1 and 0 denote the previous and current scheduled time periods, respectively. It should be noted that w_p^{-1} and $\tilde{w}_p^{-1}$ can include the sum of of all previous periods scheduled and desired workloads, as far back in time as the scheduler desires to take into consideration when determining a fair workload in the current period. There are many ways to describe how even the workload ratios for all staff members must be, here the scheduler defines a maximum normalized deviation from the mean, δ_{max}, that a staff members workload ratio may have. The mean workload ratio is

$$\bar{\gamma} = \sum_{\tilde{p}\in\tilde{\mathcal{P}}} \frac{\gamma_{\tilde{p}}}{\left|\tilde{\mathcal{P}}\right|} = \frac{1}{\left|\tilde{\mathcal{P}}\right|\bar{w}} \sum_{\tilde{p}\in\tilde{\mathcal{P}}} \frac{w_{\tilde{p}}^{-1} + w_{\tilde{p}}^0}{\tilde{p}^{dw,-1}n^{-1} + \tilde{p}^{dw,0}n^0} \tag{3.37}$$

This gives the normalized deviation from the mean for a staff member p as

$$\begin{aligned}\alpha_p = \frac{\gamma_p}{\bar{\gamma}} - 1 &= \frac{\frac{w_p^{-1}+w_p^0}{\bar{w}\left(p^{dw,-1}n^{-1}+p^{dw,0}n^0\right)}}{\frac{1}{\left|\tilde{\mathcal{P}}\right|\bar{w}}\sum_{\tilde{p}\in\tilde{\mathcal{P}}}\frac{w_{\tilde{p}}^{-1}+w_{\tilde{p}}^0}{\tilde{p}^{dw,-1}n^{-1}+\tilde{p}^{dw,0}n^0}} - 1 \\ &= \frac{\left|\tilde{\mathcal{P}}\right|\frac{w_p^{-1}+w_p^0}{p^{dw,-1}n^{-1}+p^{dw,0}n^0}}{\sum_{\tilde{p}\in\tilde{\mathcal{P}}}\frac{w_{\tilde{p}}^{-1}+w_{\tilde{p}}^0}{\tilde{p}^{dw,-1}n^{-1}+\tilde{p}^{dw,0}n^0}} - 1\end{aligned} \tag{3.38}$$

which must satisfy

$$|\alpha_p| \leq \delta_{\max}. \tag{3.39}$$

Note that $\bar{w}$ in equation 3.38 cancels out and therefore can be omitted when defining the desired workloads in equation 3.35. Additionally, if there is no previous schedule to include, all variables related to the previous schedule are set to 0. The mathematical formulation of this constraint is therefore

$$\alpha_p \leq \delta_{\max}, \forall p \in \tilde{\mathcal{P}} \tag{3.40}$$

$$-\delta_{\max} \leq \alpha_p, \forall p \in \tilde{\mathcal{P}} \tag{3.41}$$

3.3.10 Consecutive Shift Types

It is generally desired in hospitals and other settings that consecutive shifts that staff members are assigned to should be of the same type. Assigning consecutive shifts of the same type contributes to a more orderly work balance and enables staff members to adjust to the day-to-day specifics of a workplace. For example, if a staff member has worked one cashier-shift and is working the day after, then the they should ideally be assigned to the same type of shift to maintain the work-related routine that has been established. This requirement might not apply to all staff members or shift types and therefore, a constraint that can fulfil a wider range of requirements related to consecutive shift types is defined as

> All consecutive days of shifts in $\tilde{\mathcal{S}}$ for a staff member in $\tilde{\mathcal{P}}$ must be of the same type.

where $\tilde{\mathcal{S}} \subseteq \mathcal{S}$ and $\tilde{\mathcal{P}} \subseteq \mathcal{P}$ are defined by the scheduler. The constraint is mathematically formulated as

$$a_{p,s} \rightarrow \left(\bigvee_{s' \in \mathcal{S}} \left(a_{p,s'} \wedge s'^{sd} = s^{sd} - 1 \wedge s'^{t} = s^{t} \right) \vee \bigwedge_{s' \in \tilde{\mathcal{S}}} \left(s'^{sd} = s^{sd} - 1 \rightarrow \neg a_{p,s'} \right) \right)$$

$$\forall s \in \tilde{\mathcal{S}} \quad \forall p \in \tilde{\mathcal{P}} \tag{3.42}$$

The formulation above reads as "If a staff member has been assigned to a shift s in $\tilde{\mathcal{S}}$ then they should either be assigned to a shift s' in $\mathcal{S}$ on the day before that has the same type or not be assigned to any shift at all on the day before."

Consecutive Shift Types with a Previous Schedule

The previous schedule only needs to be taken into account if there is no gap between the previous schedules end date and the new schedules start date, since if there is then it would not be possible to assign a shift in the current schedule such that it contradicts this constraint. If the previous schedule ends on the day before the current schedule starts, then then constraint is formulated in the same way except that the shifts in $\tilde{\mathcal{S}}$ also include the shifts on the last day of the previous schedule.

3.4 Model Composition

The assembled mathematical model using all constraint formulations described in section 3.3 is defined as

$$\underset{s\in\tilde{\mathcal{S}}}{\mathrm{ALN}}\left(\bigwedge_{p\in\tilde{\mathcal{P}}}\left(\neg a_{p,s}\right),x\right) \tag{3.43}$$

$$\underset{s\in\tilde{\mathcal{S}}}{\mathrm{AMN}}\left(\bigwedge_{p\in\tilde{\mathcal{P}}}\left(\neg a_{p,s}\right),y\right) \tag{3.44}$$

$$\underset{s\in\tilde{\mathcal{S}}}{\mathrm{ALN}}\left(\underset{p\in\mathcal{P}}{\mathrm{ALN}}\left(a_{p,s},2\right)\rightarrow\neg\bigvee_{d\in\mathcal{D}}\left(s\in d^{\mathcal{S}}\wedge\left\{p\in\mathcal{P}|a_{p,s}\right\}=d^{\mathcal{P}}\right),x\right) \tag{3.45}$$

$$\underset{s\in\tilde{\mathcal{S}}}{\mathrm{AMN}}\left(\underset{p\in\mathcal{P}}{\mathrm{ALN}}\left(a_{p,s},2\right)\rightarrow\neg\bigvee_{d\in\mathcal{D}}\left(s\in d^{\mathcal{S}}\wedge\left\{p\in\mathcal{P}|a_{p,s}\right\}=d^{\mathcal{P}}\right),y\right) \tag{3.46}$$

$$\underset{s\in\tilde{\mathcal{S}}}{\mathrm{ALN}}\left(\bigvee_{p\in\tilde{\mathcal{P}}}\left(a_{p,s}\wedge\left(s^{\mathcal{QR}}\not\subseteq p^{\mathcal{Q}}\right)\right),x\right) \tag{3.47}$$

$$\underset{s\in\tilde{\mathcal{S}}}{\mathrm{AMN}}\left(\bigvee_{p\in\tilde{\mathcal{P}}}\left(a_{p,s}\wedge\left(s^{\mathcal{QR}}\not\subseteq p^{\mathcal{Q}}\right)\right),y\right) \tag{3.48}$$

$$\sum_{o\in\mathcal{O}}\sum_{p\in\tilde{\mathcal{P}}\setminus o^{\mathcal{P}}}\left[\bigwedge_{s\in o^{\mathcal{S}}}\left(a_{p,s}\right)\right]\geq x \tag{3.49}$$

$$\sum_{o\in\mathcal{O}}\sum_{p\in\tilde{\mathcal{P}}\setminus o^{\mathcal{P}}}\left[\bigwedge_{s\in o^{\mathcal{S}}}\left(a_{p,s}\right)\right]\leq y \tag{3.50}$$

$$\underset{s\in\tilde{\mathcal{S}}}{\mathrm{ALN}}\left(\left\{p\in\mathcal{P}|a_{p,s}\right\}=\tilde{\mathcal{P}},x\right) \tag{3.51}$$

$$\underset{s\in\tilde{\mathcal{S}}}{\mathrm{AMN}}\left(\left\{p\in\mathcal{P}|a_{p,s}\right\}=\tilde{\mathcal{P}},y\right) \tag{3.52}$$

$$\begin{cases}\left(a_{p,s}\wedge\bigwedge\limits_{s'\in\tilde{\mathcal{S}}}\left(s'^{sd}=s^{sd}-1\rightarrow\neg a_{p,s'}\right)\right)\rightarrow\bigwedge\limits_{k\in\{1,..,x-1\}}\left(\bigvee\limits_{s'\in\tilde{\mathcal{S}}}\left(a_{p,s'}\wedge s'^{sd}=s^{sd}+k\right)\right)\\ \forall p\in\tilde{\mathcal{P}}\qquad\forall s\in\tilde{\mathcal{S}}\end{cases} \tag{3.53}$$

$$\begin{cases}\left(a_{p,s}\wedge\bigwedge\limits_{s'\in\tilde{\mathcal{S}}}\left(s'^{sd}=s^{sd}-1\rightarrow\neg a_{p,s'}\right)\right)\rightarrow\bigvee\limits_{k\in\{1,\ldots,y\}}\left(\bigwedge\limits_{s'\in\tilde{\mathcal{S}}}\left(s'^{sd}=s^{sd}+k\rightarrow\neg a_{p,s'}\right)\right)\\ \forall p\in\tilde{\mathcal{P}}\qquad\forall s\in\tilde{\mathcal{S}}\end{cases} \tag{3.54}$$

$$\begin{cases}\left(a_{p,s}\wedge\bigwedge\limits_{s'\in\bar{\mathcal{S}}_1}\left(\neg a_{p,s'}\right)\wedge\bigwedge\limits_{\tilde{c}\in\{1,\ldots,c-1\}}\left(\bigvee\limits_{s'\in\bar{\mathcal{S}}_2}\left(a_{p,s'}\right)\right)\right)\rightarrow\bigwedge\limits_{s'\in\bar{\mathcal{S}}_3\cup\bar{\mathcal{S}}_4}\left(\neg a_{p,s'}\right)\\ \forall c\in\{x,\ldots,y\}\qquad\forall s\in\tilde{\mathcal{S}}\qquad\forall p\in\tilde{\mathcal{P}}\end{cases} \tag{3.55}$$

$$\begin{cases} \sum_{s\in\tilde{\mathcal{S}}} s^w\left[a_{p,s}\right] / \sum_{s\in\mathcal{S}} s^w\left[a_{p,s}\right] \geq x \\ \forall p \in \tilde{\mathcal{P}} \end{cases} \tag{3.56}$$

$$\begin{cases} \sum_{s\in\tilde{\mathcal{S}}} s^w\left[a_{p,s}\right] / \sum_{s\in\mathcal{S}} s^w\left[a_{p,s}\right] \leq y \\ \forall p \in \tilde{\mathcal{P}} \end{cases} \tag{3.57}$$

$$\begin{cases} \alpha_p \leq \delta_{\max} \\ \forall p \in \tilde{\mathcal{P}} \end{cases} \tag{3.58}$$

$$\begin{cases} -\delta_{\max} \leq \alpha_p \\ \forall p \in \tilde{\mathcal{P}} \end{cases} \tag{3.59}$$

$$\begin{cases} a_{p,s} \rightarrow \left(\bigvee\limits_{s'\in\mathcal{S}} \left(a_{p,s'} \wedge s'^{sd} = s^{sd} - 1 \wedge s'^{t} = s^{t} \right) \vee \bigwedge\limits_{s'\in\tilde{\mathcal{S}}} \left(s'^{sd} = s^{sd} - 1 \rightarrow \neg a_{p,s'} \right) \right) \\ \forall s \in \tilde{\mathcal{S}} \qquad \forall p \in \tilde{\mathcal{P}} \end{cases} \tag{3.60}$$

4

Z3 Implementation and Performance

Z3 uses first order propositional logic as one of its modelling language similar to what is used when defining the constraints in chapter 3, yet implementing them is not always as straightforward as simply applying the mathematical model into the solver. The reason for this is that certain formulations, although theoretically sound, lead to poor performance in a practical setting and thus require slight adjustments in their definition. Additionally, there can exist several ways to formulate a constraint that lead to different computation times as the Z3-solver is able to take advantage of different theories. This chapter presents modifications that are done to the mathematical model in order to aid computational efficiency. A benchmark test is carried out to compare the computation time between logical formulations, but no significant difference between computation times were found. Finally, how computation times change when assigning constraints as hard or soft are briefly explored.

4.1 Modifications

The implementation of the constraint *Fair workload* described in section 3.3.9 is not implemented into the solver exactly like the definition (3.40 and 3.41). This is because the theoretical mathematical model entangles the different decision variables in a way that is too complex for the solver. This leads to significantly longer times that the solver can run for before finding a solution. Running the actual definition of the constraint on the Z3 solver did not yield any results even after running the solver for several hours. As computation time is a crucial factor for the scheduling program, an alternative version that reduces the computation time significantly, is instead used with the drawback of having a slightly different meaning.

Each person's workload ratio is

$$\gamma_p = \frac{w_p^{-1} + w_p^0}{\tilde{w}_p^{-1} + \tilde{w}_p^0} = \frac{w_p^{-1} + w_p^0}{\bar{w}\left(p^{dw,-1}n^{-1} + p^{dw,0}n^0\right)} \tag{4.1}$$

but instead of using the actual workload w_p^0, which in itself is a large variable that involves many other decision variables, it is pre-calculated for each staff as the sum of the workloads for all shifts in $\tilde{\mathcal{S}}$ divided by the sum of all staff desired workloads. This multiplied with the staff member p's desired workload gives the approximate

workload that they should work this period,

$$w_p^0 = \frac{\sum_{s \in \tilde{\mathcal{S}}} s^w}{\sum_{p' \in \tilde{\mathcal{P}}} p'^{dw}} p^{dw} \tag{4.2}$$

The difference can be seen when comparing with the actual workload

$$w_p^0 = \sum_{s \in \tilde{\mathcal{S}}} s^w \left[a_{p,s}\right] \tag{4.3}$$

As can be noted, the actual workload requires summing over the decision variable which causes the model to explode since there are too many parameters that are interacting with each other for each staff member. The mean workload ratio and normalized deviation is calculated in the same way as before.

$$\bar{\gamma} = \sum_{\tilde{p} \in \tilde{\mathcal{P}}} \frac{\gamma_{\tilde{p}}}{\left|\tilde{\mathcal{P}}\right|}, \quad \alpha_p = \frac{\gamma_p}{\bar{\gamma}} - 1 \tag{4.4}$$

$$|\alpha_p| \leq \delta_{max} \quad \forall p \in \tilde{\mathcal{P}} \tag{4.5}$$

4.2 Benchmark

A benchmark test is conducted on three constraints, *Assignment of Shifts*, *Shared Shifts* and *Qualified Assignments*, in order to see if different formulations affect the computation time. Two scenarios are used, one scenario does not take any previous period's schedule into account while the second scenario does. Both scenarios are further elaborated in chapter 6. The benchmark is performed on a Windows computer with an Intel Core i7-8550U CPU and the script is run without any other programs in the background. The formulations are benchmarked on the compiler time and the solver time. Compiler time is the time it takes to compile all the constraints and add them to the solver while the solver time is the time it takes for Z3 to find a solution. In addition, a benchmark of how soft constraints affect the computation time will be presented.

4.2.1 Assignment of Shifts

$$\underset{s \in \tilde{\mathcal{S}}}{\text{ALN}} \left(\bigwedge_{p \in \tilde{\mathcal{P}}} \left(\neg a_{p,s}\right), x \right) \tag{4.6}$$

$$\underset{s \in \tilde{\mathcal{S}}}{\text{AMN}} \left(\bigwedge_{p \in \tilde{\mathcal{P}}} \left(\neg a_{p,s}\right), y \right) \tag{4.7}$$

The first constraint, *Assignment of Shifts*, can be modelled in different ways. The ALN and AMN operator can either be modelled using Z3s equivalent built-in functions `PbLe` and `PbGe`, or with the logical operator `Sum(If(Boolean condition, 1, 0))`.

The `If(Boolean condition, 1, 0)` outputs 1 if the Boolean condition is *true* and 0 if it is *false*. When comparing the sum over the `If` operator to the limits x and y, either `Sum(If)` $> x - 1$ or `Sum(If)` $\geq x$ can be used.

The $\bigwedge(\neg a_{p,s})$ formulation can be formulated with the Z3 logical `AND(Not(...))` operator, which works in the same way, or with `Not(OR(...))` equivalent to $\neg \bigvee a_{p,s}$.

Table 4.1: Compiler and solver time for the different formulations of the constraint Assignment of Shifts in Z3.

AMN	ALN	$\bigwedge$	x	y	Scenario 1 Compiler	Scenario 1 Solver	Scenario 2 Compiler	Scenario 2 Solver
`If`	`If`	`Not OR`	$\geq x$	$\leq y$	13.12s	0.71s	12.34s	0.67s
`If`	`If`	`Not OR`	$> x - 1$	$< y + 1$	12.26s	0.66s	12.21s	0.67s
`PbGe`	`PbLe`	`AND Not`			12.58s	0.74s	12.16s	0.67s
`If`	`If`	`AND Not`	$> x - 1$	$< y + 1$	12.42s	0.65s	12.46s	0.69s
`PbGe`	`PbLe`	`Not OR`			12.33s	0.65s	12.17s	0.67s
`If`	`If`	`AND Not`	$\geq x$	$\leq y$	12.53s	0.70s	12.46s	0.67s

As can be seen in table 4.1, the solver time does not vary significantly depending on what operator or scenario is used.

4.2.2 Shared Shifts

$$\underset{s\in\tilde{\mathcal{S}}}{\mathrm{ALN}}\left(\underset{p\in\mathcal{P}}{\mathrm{ALN}}\left(a_{p,s},2\right)\rightarrow\neg\bigvee_{d\in\mathcal{D}}\left(s\in d^{\mathcal{S}}\wedge\left\{p\in\mathcal{P}|a_{p,s}\right\}=d^{\mathcal{P}}\right),x\right) \tag{4.8}$$

$$\underset{s\in\tilde{\mathcal{S}}}{\mathrm{AMN}}\left(\underset{p\in\mathcal{P}}{\mathrm{ALN}}\left(a_{p,s},2\right)\rightarrow\neg\bigvee_{d\in\mathcal{D}}\left(s\in d^{\mathcal{S}}\wedge\left\{p\in\mathcal{P}|a_{p,s}\right\}=d^{\mathcal{P}}\right),y\right) \tag{4.9}$$

$\underset{p\in\mathcal{P}}{\mathrm{ALN}}\left(a_{p,s},2\right)$ in the *Shared Shift* constraint can be modelled by using different operators. For example, one way is to take $\neg$`(Sum(If(`$a_{p,s}$`, 1, 0)) <= 1)`, read as "the sum of the number of assigned shifts to person p should not be 0 or 1". Alternatively, the operator `PbLe` can be used instead of the `If` operator. A third way to construct the constraint is to use `PbGe` and set the minimum value to 2. The different combinations and their benchmark time can be seen in table 4.2, where it can be noted that the times do not differ significantly.

Table 4.2: Compiler and solver time for the different formulations of constraint Shared Shifts in Z3.

	Scenario 1		Scenario 2	
ALN	Compiler	Solver	Compiler	Solver
`PbGe`	12.08s	0.64s	12.16s	0.70s
`Sum If`	13.23s	0.74s	12.52s	0.67s
`PbLe`	12.23s	0.64s	12.39s	0.74s

4.2.3 Qualified Assignment

$$\underset{s\in\tilde{\mathcal{S}}}{\text{ALN}}\left(\bigvee_{p\in\tilde{\mathcal{P}}}\left(a_{p,s}\wedge\left(s^{\mathcal{QR}}\not\subseteq p^{\mathcal{Q}}\right)\right),x\right) \tag{4.10}$$

$$\underset{s\in\tilde{\mathcal{S}}}{\text{AMN}}\left(\bigvee_{p\in\tilde{\mathcal{P}}}\left(a_{p,s}\wedge\left(s^{\mathcal{QR}}\not\subseteq p^{\mathcal{Q}}\right)\right),y\right) \tag{4.11}$$

The *Qualified Assignment* was simply modified by either using the `If` in the same way as the *Assignment of Shifts* constraint, or `PbLe/PbGe` Z3 functions for the AMN/ALN operators.

Table 4.3: Compiler and solver time for the different formulations of constraint Qualified Assignment in Z3.

		Scenario 1		Scenario 2	
AMN	ALN	Compiler	Solver	Compiler	Solver
`PbGe`	`PbLe`	12.63s	0.65s	12.09s	0.69s
`If`	`If`	12.14s	0.72s	11.99s	0.83s

Table 4.3 displays that there is a small difference between the two implementations, suggesting that there might be a correlation between computation efficiency and the function used for this constraint.

4.2.4 Hard and Soft

Whether a constraint is assigned as hard or soft has an impact on the computation time. For example, for the case described in chapter 6 but where both the *Fair Workload* constraints are changed to soft instead of hard, the solver time increased from 0.67s to 1.02s. Furthermore, changing all constraints to soft yielded a solver time of 2.04s.

5

Genetic Algorithm Implementation and Performance

In this chapter, the operations used in the genetic algorithm are described. These operations include what is typically used in genetic algorithms but are modified in many ways to better suit the problem of shift scheduling. Many of the operations are dependent on certain variables specific to how each constraint is evaluated. Therefore, the implementation of constraints and how they are used in the context of the algorithm is described. Finally, the performance of the algorithm is tested using the case described in chapter 6. This involves benchmarking various configurations of the algorithm and finally using the best one to perform a longer optimization of the schedule.

5.1 Genetic Algorithm Operations

The genetic algorithm uses a number of operations, such as selection, mutations and crossover, that enable the population of candidate solutions to be optimized toward better fitness scores. The methods behind these processes are described here, as many of them are not the standard versions described in chapter 2.4.

Encoding

The encoding used here is that of the decision variable defined in chapter 3, which has several advantages. Firstly, it is easy to manipulate and perform matrix calculations with as each row corresponds to a staff members shift assignments while each column corresponds to which staff members are assigned to it. Secondly, many of the constraints defined in chapter 3 can then be used directly. A disadvantage with this encoding is that it contains a lot of unnecessary information. For each shift, it defines both who is and who isn't assigned while it would suffice to only define one or the other. This contributes to a longer chromosome and subsequently more calculations to perform. Nonetheless, the encoding is intuitive and useful as it uses the same convention as the defined constraints.

Initialization

The initial population consists of randomly generated individuals. This can be done in several ways, either starting with no assignments, one random staff member

per shifts, all assignment variables randomly set to *true* or *false*, and many more. Two methods are used here. The first is to start without any assignments. The idea behind this is that then the optimization algorithm can build the schedule in a bottom-up fashion, without having to undo faulty assignments. The second method is to randomly assign one staff member per shift. In many cases it could be advantageous to assign *anybody*, and thus begin the optimization with diversity in the population.

Evaluation and the Objective Function

The evaluation of a candidate solution i is done by calculating a fitness value $\delta_{i,j}$ for each individual constraint j placed on the schedule and then defining the candidates total fitness value as

$$F_i = \sum_j \delta_{i,j}. \tag{5.1}$$

Because of the way that the different constraint types in chapter 3 are modelled, they are either fulfilled or not. It makes intuitive sense to use the number of fulfilled constraints as a fitness value, meaning $\delta_{i,j} = 1$ if constraint j is fulfilled and otherwise 0, but in practice this would lead to some problems. In a sense, the number of fulfilled constraints is a lower resolution of the fitness of a candidate solution, as the degree at which the constraints are fulfilled or not is lost. This would lead to a greater likelihood of finding several candidate solutions with the same fitness value, despite one being superior to the other. If the superiority of one solution is not reflected in the fitness value, then there will be no incentive to favor it.

To increase the resolution of the fitness value, a value representing to what degree a constraint has been been violated is calculated, referred to as the violation severity and denoted as $v_{i,j}$ for candidate solution i and constraint j. It can be used to calculate the constraint fitness value, $\delta_{i,j}(v_{i,j})$. How the constraint fitness values are calculated for each constraint type is described in appendix A.

Using an objective function often incorporates the problem of how to distinguish between hard and soft constraints. One way to go about this is to apply weights to the various constraints depending on if they are hard or soft. This would increase the effect hard constraints have on the fitness score and therefore incentivize the optimization toward fulfilling them over soft constraints. Related to this is also the case of different constraints having different importance levels, regardless of if they are hard or soft. This too can be solved by weighting the constraint fitness scores. But, determining the values of the weights is not trivial and therefore no attempt is made here to use weights.

Selection

A typical tournament selection, described in section 2.4.2, is used to select candidate solutions for the next generations population.

Mutation

A simple approach to mutation is described in section 2.4.3. The problem with such an approach is that the size of a chromosome is so large that many blindly performed mutations will have to be done for a feasible schedule to finally be reached. Since the evaluation process to calculate the fitness value is computationally heavy, this quickly makes the method unpractical. To solve this, a method is developed and referred to as *targeted mutation*.

Let a $|\mathcal{P}| \times |\mathcal{S}|$-matrix M^i, with the same size as the genome, be referred to as the mutation matrix for candidate solution i. Each element $M^i_{a,b}$ corresponds to an assignment of shift b to a staff member a. Initially, all elements are set to 0. As the candidate solution is evaluated with respect to each constraint, the elements in M^i will be incremented to indicate that a mutation of the corresponding assignment would increase the chance of the candidate solution fulfilling the constraint. Once the entire evaluation is complete, M^i can be interpreted as a map indicating to what degree each assignment violates the constraints. This is then used to guide the mutations. Let the parameter n_m indicate the desired average number of mutations per chromosome. For each assignment variable a, b in candidate solution i, the probability of it being mutated is

$$p_{a,b} = \frac{n_m}{|M_i|} M^i_{a,b}. \tag{5.2}$$

The higher the value of $M^i_{a,b}$ is, the more likely that the gene corresponding to the assignment of shift b to staff member a will be mutated. How each constraint type increments the elements in the mutation matrix is described in appendix A.

In addition to targeted mutation, another variant of the mutation procedure called *guided mutation* is developed and is closely linked to the *Shared Shifts* constraint in chapter 3. It is assumed that in most cases the *Shared Shifts* constraint will be set such that the number of allowed invalid staff groups sharing a shift will be smaller than the total number of shifts. Using this, mutations can be guided so that only valid shift sharing staff groups, or single staff members, are assigned to shifts. This drastically reduces the size of the search space. Without any restrictions there are 2^n possible assignments of a shift, where n is the number of staff members. If all invalid staff group assignments are removed, the number of possible assignments becomes $n + 1 + |\mathcal{D}|$ where $\mathcal{D}$ is defined in chapter 3 as the set of allowed shift sharings. The addition of 1 comes from the possibility of no staff members being assigned. By reducing the search space, the algorithm explores more of the regions with higher quality solutions and thus the optimization is accelerated. However, the assumption that the *Shared Shifts* constraint is setup in this way is still an assumption and will not always be true. Nonetheless, guided mutations can be used as a primer to the population such that it is used early in the optimization and then turn off at some point to allow for the *Shared Shifts* constraints to be taken into account.

Crossover

The procedure of crossover is described in section 2.4.4. Two-point crossover is used here with probability p_c. Note that special care must be taken since each genome is a matrix and not a one-dimensional array. If crossover is performed along the dimension of staff members, then this would have the effect of trading all the shift assignments for specific staff members between schedules. This is somewhat counter-intuitive and likely detrimental to the optimization, since it would lead in many cases to shifts being unassigned or shared by multiple people. If crossover is instead performed along the dimension of shifts, then schedules will be trading the assignments of specific shifts between each other, which would be less destructive and make it possible for shifts that are assigned to the "right" staff members to be shared within the population. However, the shifts are not ordered in any particular way in the assignment matrix. This means that when performing crossover, all the shifts that are to be traded do not follow each other chronologically. This means that instead of trading portions of time in the schedule, the schedules will simply be trading randomly selected shifts that happen to be ordered after one another in the assignment matrix. Therefore, the crossover used here is based on the chronological order of shifts.

There are several ways to determine the first and last day, the crossover points, of the schedule that is to be traded. Two methods are used here. The first is the standard crossover method where crossover points are selected randomly. Let s^{sd}_{min} and s^{sd}_{max} be the first and last days that shifts start on. The two crossover points are then

$$c_1, c_2 \sim \mathcal{U}\left(s^{sd}_{min}, s^{sd}_{max}\right). \tag{5.3}$$

where $c_1 \leq c_2$ without loss of generality. All shifts $s \in \mathcal{S}$ with a start day that satisfies $c_1 \leq s^{sd}_s \leq c_2$ are traded between the schedules. A possible disadvantage with this method is that the days in the middle of the schedule are more likely to be selected for crossover than the days toward the beginning and end. For a schedule that starts on day 1 and ends on day n, the probability of a day γ being selected as part of the crossover is

$$p(\gamma) = 1 - \underbrace{\left(\frac{\gamma - 1}{n}\right)^2}_{p(c_1<\gamma, c_2<\gamma)} - \underbrace{\left(\frac{n - \gamma}{n}\right)^2}_{p(\gamma<c_1, \gamma<c_2)} = \frac{-2\gamma^2 + 2(n+1)\gamma - 1}{n^2} \tag{5.4}$$

which shows that the probability depends on the placement of the day γ in the schedule and the it is highest at the center of the schedule. An effect of this is that advantageous scheduling at the start or end of a schedule will be harder to spread throughout the population. To overcome this, a second method is developed where each day has an equal probability of being included in the crossover, called *uniform crossover*. This is done by randomly selecting the length and position of a window as

$$l \sim \mathcal{U}(1, n), \quad p \sim \mathcal{U}(2 - l, n) \tag{5.5}$$

which are used to calculate the crossover points

$$\begin{aligned} c_1 &= s^{sd}_{\min} + \max(1, p) - 1 \\ c_2 &= s^{sd}_{\min} + \min(n, p + l - 1) - 1 \end{aligned} \tag{5.6}$$

This is akin to selecting a random position of a sliding window of length l along the schedule, starting where the last day of the window is at $s^{sd}_{\min}$ up until the first day of the window is at $s^{sd}_{\max}$. This means that l is not necessarily the length of the portion of the schedule that is traded. The probability of a day γ being included in the crossover is equal to the length of the window l divided by the total number of positions that it may take along the schedule, for each of the n possible window sizes. This gives

$$p(\gamma) = \frac{1}{n} \sum_{l=1}^{n} \frac{l}{n + l - 1} \tag{5.7}$$

which does not depend on γ and is thus equal for all days.

Elitism

Elitism is used, where one copy each of the best n_e candidate solutions are inserted into the next generations population.

5.2 Performance

The performance of the genetic algorithm is presented here. First, a benchmark of the various procedures is done by using different configurations of the algorithm and letting it run for a set amount of time, after which noting the best fitness score. Finally, the best configuration is used to generate a schedule for a longer time.

5.2.1 Benchmark of the GA Procedures

To benchmark the various procedures in the genetic algorithm, the case presented in chapter 6 is used, with 31 hard and 12 soft constraints and where the previous schedule is included. Schedules are generated using various configurations of the genetic algorithm. Each configuration is given a set amount of time, after which the fitness score and number of satisfied hard and soft constraints are saved. The procedures that are benchmarked include initialization without assignments and with only one assignment per shift, standard and targeted mutations as well as standard and uniform crossover. The parameters used for all configurations are

Optimization time:	5 minutes
Population size:	30
Tournament size, n_t:	3
Tournament parameter, p_t:	0.5
Crossover probability, p_c:	0.3
Mutations per chromosome, n_m:	1

The results are shown in table 5.1. It should be noted that variations occur due to the stochastic nature of the algorithm and therefore the results should be taken with scepticism. The fitness when using targeted mutations (avg. −54.62) is significantly above the fitness for standard mutations (avg. −74.18), implying that targeted mutations outperform standard mutations. The results are mixed when it comes to the two crossover procedures. Uniform crossover outperforms standard crossover when used with standard mutations, while the opposite is true when using targeted mutations. Initialization without assignments (avg. −66.79) show little difference compared to assigning one random staff member per shift (avg. −62.02). The best results are found when initializing with one staff member per shifts, using targeted mutations and standard crossover.

Table 5.1: The results after 5 minutes of generating a schedule for the case presented in chapter 6 using various configurations of the genetic algorithm. There are in total 31 hard and 12 soft constraints.

Initialization	Mutation	Crossover	Fitness	Hard	Soft
No assignments	Standard	Standard	−83.55	5	7
No assignments	Standard	Uniform	−73.69	6	7
No assignments	Targeted	Standard	−52.27	9	6
No assignments	Targeted	Uniform	−57.64	6	6
One per shift	Standard	Standard	−79.49	6	4
One per shift	Standard	Uniform	−60.00	4	7
One per shift	**Targeted**	**Standard**	**−48.77**	**6**	**6**
One per shift	Targeted	Uniform	−59.80	7	4

When using the guided mutation procedure during the first half of the optimization, after applying targeted mutation, together with standard crossover and one staff member per shift initialization, result is a fitness score of −27.80 with 8 hard constraints and 5 soft constraints satisfied. It appears that guided mutation significantly increases performance when used during the first portion of the optimization.

5.2.2 Results Using the Best Configuration

Using one staff member per shifts initialization, standard crossover and guided mutations during the first 75% of the optimization and targeted mutations during the rest, a schedule is generated for an hour. Figure 5.1 shows the best fitness score over time as the algorithm generates the schedule. It can be seen that the rate at which the fitness score increases decreases over time. Very little improvement is seen in the last 10 minutes of running, implying that the algorithm has begun to converge to an optimum. The result is a schedule with a fitness score of 47.49, with only 14/31 hard constraints and 8/12 soft constraints satisfied. The resulting schedule is seen in section 6.4.

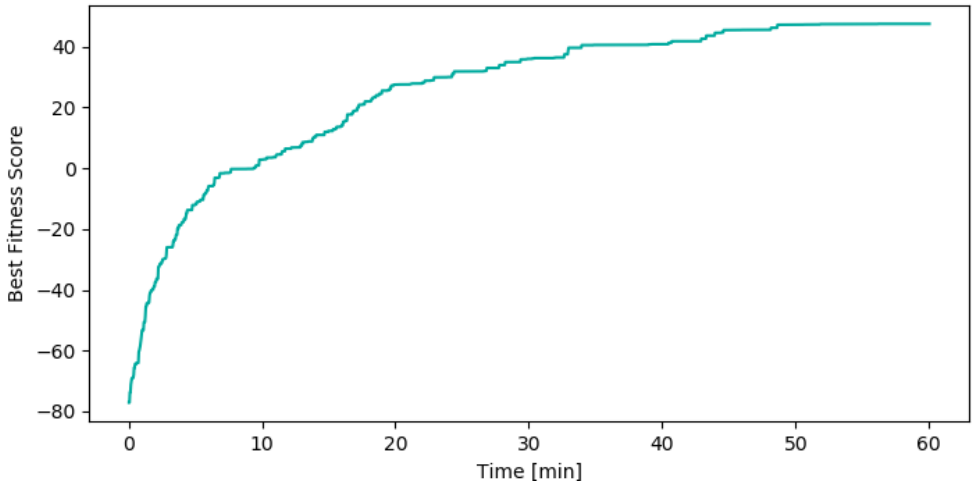


Figure 5.1: A graph showing the best fitness score over time for the genetic algorithm using one staff member per shift initialization, standard crossover, guided mutation during the first 75% and targeted mutation during the rest.

6
Case

The scheduling program is tested on the summer schedule which ranges from 26/6/2021 until 5/9/2021 for the pediatric wards at Queen Silvia Children's Hospital. Summer periods are especially tricky to schedule manually since there are a lot of requirements from personnel to be on leave for holiday during certain periods and days. The model is often prone to be over-constrained during the summer period and it is not uncommon that compromises have to be made regarding staff members leave requests. Therefore, the summer period is a interesting scenario for testing the scheduling program since the other periods are usually less constrained, thus giving room for a larger search space. The complete user interface developed and used to generate schedules for the case is found in section 6.2.

6.1 Case Description

The case parameters for the on-call schedule are

- There are 8 backup doctors, 4 transplantation doctors and 4 who are qualified for both.
- The backup on-call shifts are twice as burdensome as transplantation on-call shifts, they are therefore given the burden values of 10 and 5 respectively.
- The duration of one shift is 24 hours and extends over one day.
- The holidays include Fridays, Saturdays and Sundays and the midsummer holiday on the 25'th and 26'th of June, which coincides with a weekend.
- The normal workdays are thus Mondays, Tuesdays, Wednesdays and Thursdays.
- All staff are allowed to be assigned to overlapping transplantation and backup shifts during workdays.

For more information about each staff members special requirements and workload, see table B.1 in appendix B. Additionally, there are special cases that must be taken into account. For example, a special requirement seen in table B.1 is that staff member 4 should only be assigned to 2 workdays during week 32. Also, there are only two instances where staff members are allowed to share shifts during the summer period, and both occur during the midsummer weekend, the 25'th to the 27'th of June. The first is for staff members 14 and 15 to share the transplantation shifts and the second is for staff members 6 and 9 to share the backup shifts. This is not only allowed but is also a requirement, which will be addressed in section 6.1.2.

6.1.1 Previous Schedule

A previous schedule that was created by the scheduler through self-scheduling can be seen in figure 6.1. The scheduler wishes to take the previous schedule into account as the staff members should work more if they received fewer shifts the previous period and vice versa. The previous period included weeks 10 to 14. Staff members 1 and 3 were on leave and staff members 4 and 11 are extras for the summer period, which is why they were not scheduled any shifts during the previous period.

Staff member 14 worked the last weekend of the previous schedule and requires a special constraint to solve that they should be assigned the next weekend since another constraint is violated that way. A run with *Check* macro, described in section 6.2, showed that 2 out of the 11 hard constraints and 3 out of the 6 soft constraints were violated for the self-scheduling of the previous periods shifts.

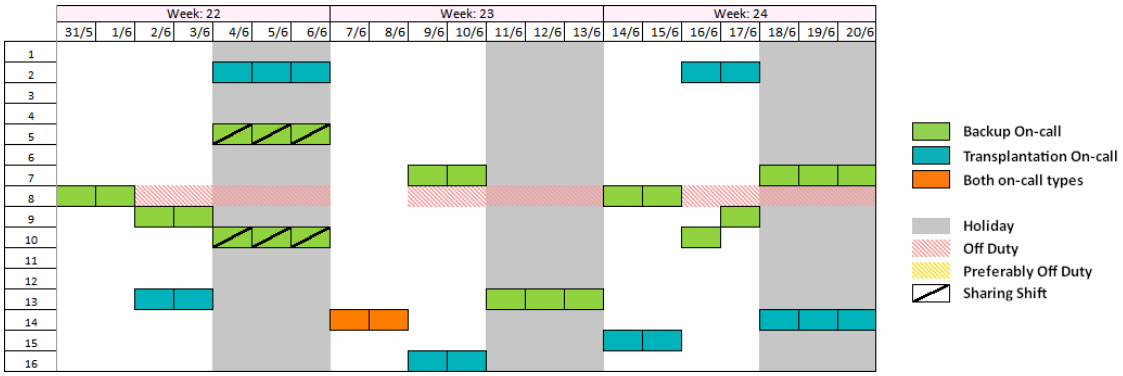


Figure 6.1: The last 3 weeks of the shift allocations for the previous period, which was done through self-scheduling.

6.1.2 Requirements

All requirements that are placed on the schedule are presented in this section, each followed by the boilerplate used to implement them. The requirements are sorted in order of the constraints that are used to implement them, described in section 3.3. Unless stated otherwise, all constraints are considered hard.

Assignment of Shifts

The following requirements are implemented using the *Assignment of Shifts* constraint in section 3.3.1.

R1. *All shifts must be assigned at least one staff member.*

> The number of the shifts in **All** that are not assigned at least one staff member in **All** must be greater or equal to **0** and less than or equal to **0**.

R2. *Staff member 4 should only work 2 shifts, both on workdays during week 32.*

> The number of the shifts in **workdays, week 32** that are not assigned at least one staff member in **4** must be greater or equal to **2** and less than or equal to **2**.

Shared Shifts

The following requirement is implemented using the *Shared Shifts* constraint in section 3.3.2.

R3. *No instances of invalid shift sharing may occur.*

> The number of invalid staff groups assigned to shifts in **All** must be greater or equal to **0** and less than or equal to **0**.

Qualified Assignments

The following requirement is implemented using the *Qualified Assignments* constraint in section 3.3.3.

R4. *A staff member can only be assigned to shifts they are qualified for.*

> The number of shifts in **All** that have been assigned an unqualified staff member in **All** must be greater or equal to **0** and less than or equal to **0**.

Overlapping Shifts

The following requirement is implemented using the *Overlapping Shifts* constraint in section 3.3.4.

R5. *No instances of invalid overlapping shifts may occur.*

> The number of staff in **All** assigned to unsanctioned overlapping shifts must be greater or equal to **0** and less than or equal to **0**.

Staff Combination Assignments

The following requirements are implemented using the *Staff Combination Assignments* constraint in section 3.3.5.

R6. *Staff members 14 and 15 should share the transplantation shifts on the 25-27th of July.*

> The number of shifts in **Transplantation, 25/7-27/7** that are assigned to all staff members in **14, 15**, and only them, must be greater or equal to **0** and less than or equal to **0**.

R7. *Staff members 6 and 9 should share the backup on-call shifts on the 25-27th of July.*

> The number of shifts in **Backup, 25/7-27/7** that are assigned to all staff members in **6, 9**, and only them, must be greater or equal to **0** and less than or equal to **0**.

Consecutive Days

The following requirements are implemented using the *Consecutive Days* constraint in section 3.3.6.

R8. *A staff member can be assigned a maximum of 3 shifts consecutively*

> The number of consecutive days of shifts in **All** for a staff member in **All** must be greater or equal to **1** and less than or equal to **3**.

R9. *The shifts should, if possible, be assigned such that each staff works consecutively according to one of the following patterns: Monday-Tuesday, Wednesday-Thursday, Friday-Saturday-Sunday.*

This is a preferable requirement thus, the constraints are labelled as *soft*. One constraint for each pattern has to be constructed.

> The number of consecutive days of shifts in **Monday and Tuesday** for a staff member in **All** must be greater or equal to **2** and less than or equal to **2**.

> The number of consecutive days of shifts in **Wednesday and Thursday** for a staff member in **All** must be greater or equal to **2** and less than or equal to **2**.

> The number of consecutive days of shifts in **Friday, Saturday and Sunday** for a staff member in **All** must be greater or equal to **3** and less than or equal to **3**.

Before and after Consecutive Days

The following requirements are implemented using the *Before and after Consecutive Days* constraint in section 3.3.7.

R10. *If a staff member has been assigned 3 shifts consecutively of any type, then the staff member should not be assigned any shifts 2 days before or after.*

> For a staff member in **All**, **3** to **3** consecutive days of shifts in **All** must be preceded by **2** days without shifts in **All** and followed by **2** days without shifts in **All**.

R11. *If a staff member has been assigned shifts on* 3 *holidays in a row, then they should not be assigned any shifts on holidays at least* 14*, but preferably* 21*, days before or after.*

This requirement consists of two constraints. The first is a hard constraint regarding 14 days before or after and the second is a soft constraint regarding 21 days before or after.

> For a staff member in **All except 14**, **3** to **3** consecutive days of shifts in **Holidays** must be preceded by **14** days without shifts in **Holidays** and followed by **14** days without shifts in **Holidays**.

Note that an exception is made for staff member 14 since they worked three consecutive holidays at the end of the previous period but is required to work the midsummer holiday the week after. A *soft* constraint is introduced to overcome this problem.

> For a staff member in **14**, **3** to **3** consecutive days of shifts in **Holidays** must be preceded by **14** days without shifts in **Holidays** and followed by **14** days without shifts in **Holidays**.

The *soft* constraint where preferably 21 days should pass without assigning the staff a holiday shift can be formulated as

> For a staff member in **All**, **3** to **3** consecutive days of shifts in **Holidays** must be preceded by **21** days without shifts in **Holidays** and followed by **21** days without shifts in **Holidays**.

Assignment Fractions

The following requirements are implemented using the *Assignment Fractions* constraint in section 3.3.8.

R12. *At least* 30*% of the workload assigned to each staff member that is qualified for both shift types must be from each type of shift.*

The requirement can be divided into two constraints, one that deals with the backup on-call and one that deals with the transplantation on-call. For the backup on-call, the constraint is formulated as

> The fraction of the workload of shifts that a staff member in **13, 14, 15, 16** is assigned to that is in **Backup on-call** must be greater or equal to **0,3** and less than or equal to **1**.

And for the transplantation on-call, the constraint is formulated as

> The fraction of the workload of shifts that a staff member in **13, 14, 15, 16** is assigned to that is in **Transplantation on-call** must be greater or equal to **0,3** and less than or equal to **1**.

R13. *Every staff member's off-duty and preferably off-duty periods must be honoured.*

The difference between being off-duty and being preferably off-duty is that the first is a hard constraint while the other is a soft constraint. Both are implemented using the *Assignment Fractions* constraint for each staff member. As an example, staff member 2 is off-duty during weeks 25-28 and 31-32 and preferably off-duty from 30/7-1/8. Therefore, the hard constraint

> The fraction of the workload of shifts that a staff member in **2** is assigned to that is in **weeks 25-28 and 31-32** must be greater or equal to **0,3** and less than or equal to **1**.

and the soft constraint

> The fraction of the workload of shifts that a staff member in **2** is assigned to that is in **30/7-1/8** must be greater or equal to **0,3** and less than or equal to **1**.

is added. This is done for every staff member using their respective off-duty and preferably off-duty dates.

Fair Workload

The following requirements are implemented using the *Fair Workload* constraint in section 3.3.9.

R14. *All staff members should receive as equal of a workload as possible for all shifts.*

> The normalized deviation between the mean workload ratio and the workload ratio for a staff member in **All** should be less than or equal to **0,3** for assigned shifts in **All**.

R15. *Holidays should be evenly distributed between each staff member.*

> The normalized deviation between the mean workload ratio and the workload ratio for a staff member in **All except 8** should be less than or equal to **1** for assigned shifts in **Holidays**.

Note that this constraint applies to every staff except for staff member 8 since they do not wish to work holidays. It would not be possible to even out the workload, in relation to the other staff, for someone who does not work during holidays.

The summer period is according to the scheduler, the shortest period to schedule during the whole year. The requirement is therefore hard to satisfy since there are only a limited number of holidays available, which is why the number is set relatively high.

Consecutive Shift Types

The following requirements are implemented using the *Consecutive Shift Types* constraint in section 3.3.10.

R16. *Consecutive shifts that have been assigned a staff member must be of the same type.*

> All consecutive days of shifts in **All** for a staff member in **All** must be of the same type.

6.2 User Interface

The user interface was requested by the scheduler to be made in Microsoft Excel. An Excel add-in called XLWings was installed, enabling the interaction between a Python script and Excel. The interface consists of three main sheets which can be utilized to input values and preferences. The functionality of the program will further be explained in details.

”Input Data” is the first sheet the and contains boilerplates that deals with necessary information. The boilerplates are divided into 5 tables and each table contains information about relevant dates, shifts, staff, shift-sharing and overlapping shifts. All table values are editable. The 5 tables with information from the case can be seen in figure 6.2.

The second sheet ”Constraints” is where the constraints are entered as boilerplates. Two examples can be seen in figure 6.3 and as can be observed, the constraints are written in the same way as the formulations of the requirements. The boilerplate can be perceived to be a bit tricky to understand since the constraints can be modelled in different ways using the boilerplate.

Start Date	2021-06-21
End Date	2021-09-05
Holidays	Fri + Sat + Sun

Shifts

Type	Days	Start hour	Duration (hours)	Qualifications required	Shift workload
Backup	All	0	24	bj	10
Transplantation	All	0	24	tx	5

Staff

Name	Qualifications	Workload desired	Preferably off-duty	Off-duty
1	T	1		w27:30 + 2/9 + w32:33
2	T	1	30/7:1/8	w25:28 + w31:32
3	T	0,2		All - w27:28
4	T	0,1		All - workday,w32
5	B	1	21:22/6	23:27/6 + 16:18/7 + w29:32
6	B	0,75	9:11/7 + 13:15/8	w28:31 + workdays+ 19:22/8 + 26:29/8

Shift-sharing

Names	Shift type	Days	Counted workload
14, 15	All	25:27/6	0,5
6, 9	All	25:27/6	0,5

Overlapping shifts

Names	Shift Type	Days
All	Backup + Transplantation	Workdays

Figure 6.2: Boilerplate tables for the Input Data sheet. Note that the staff table has been cut-off to save space.

1. Assignment of Shifts

The number of the shifts in ___ that are not assigned at least one staff member in ___ must be greater or equal to Min and less than or equal to Max.

Priority	Shift Type	Shift Days	Staff	Min	Max
Hard	All	All	All	0	0
Hard	Transp	Workday, w32	Robert Saalman	2	2

2. Shared Shifts

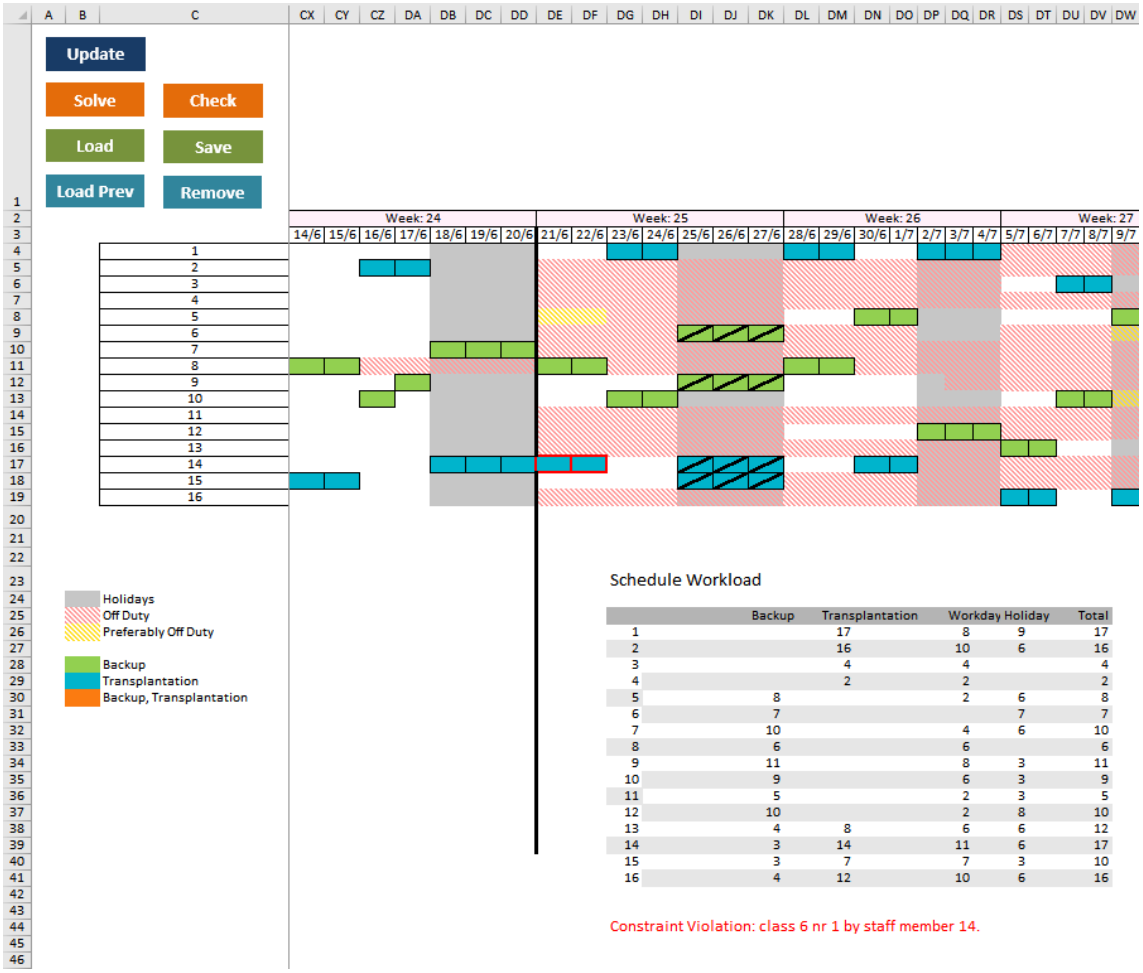
The number of invalid staff groups assigned to shifts in ___ must be greater or equal to Min and less than or equal to Max.

Priority	Shift Type	Shift Days	Min	Max
Hard	All	All	0	0

Figure 6.3: Boilerplate tables for the constraints. Note that only two constraint type are shown to save space.

The generated schedule can be seen in the last sheet, see figure 6.4. The sheet enables some functionalities in the shape of buttons localized on the left hand-side. The program is initiated by either loading a previous schedule (*Load Prev*), if it should be taken into account, or solve it directly with the button *Solve*. The Solve button starts the Python script for the SMT-solver and outputs a schedule. The

black bold line shows where the previous schedule ends and where the new generated schedule starts.



	Backup	Transplantation	Workday	Holiday	Total
1		17	8	9	17
2		16	10	6	16
3		4	4		4
4		2	2		2
5	8		2	6	8
6	7			7	7
7	10		4	6	10
8	6		6		6
9	11		8	3	11
10	9		6	3	9
11	5		2	3	5
12	10		2	8	10
13	4	8	6	6	12
14	3	14	11	6	17
15	3	7	7	3	10
16	4	12	10	6	16

Figure 6.4: User interface for generating, changing and editing the schedule. The main functions are implemented in the macro-buttons placed on the upper left corner.

The *Update* button enables the functionality of updating changes made in schedule. For example, if the scheduler would like to change the color or move around the assigned shifts. Constraints may become violated after changes have been made, in order to check the exact constraint and staff member that are affected, the button *Check* is used. Check marks in the schedule where the violation occurs and outputs the staff and exact constraints that has been violated under the *Schedule Workload* table. In the figures example, shifts has been placed such that staff member 14 is working more than 3 consecutive days in a row and thus violating the R8 requirement under the Consecutive Days constraint. The two green buttons *Load* and *Save* are used for loading and saving schedules that has been created.

6.3 The Z3 Results

The resulting schedule generated by the scheduling program in Excel can be seen in figure 6.5. The program has managed to satisfy all hard constraints while 5 out of 12 soft constraints have been violated of which a evident one can be seen on week 32, where staff member 6 is assigned to the three holidays when they would rather be off-duty. Compilation of constraints took 12.08 s and solving took 0.68 s.

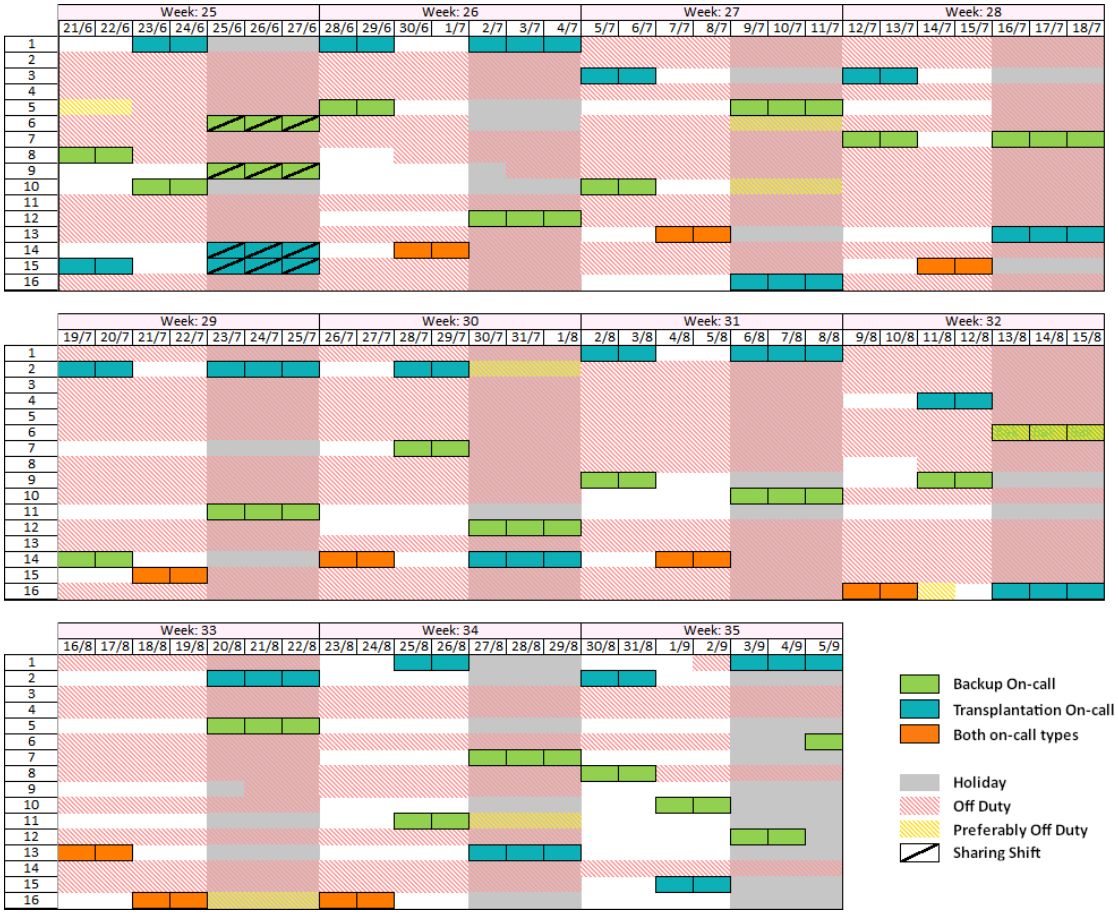


Figure 6.5: The schedule generated using the SMT-solver Z3. Rows represent staff members and columns represent dates.

6.4 The Genetic Algorithm Results

The schedule generated by the genetic algorithm after 1 hour is seen in figure 6.6. It is only able to satisfy 14 of the 31 hard constraints and 8 of the 12 soft constraints, meaning it is not able to generate a feasible schedule.

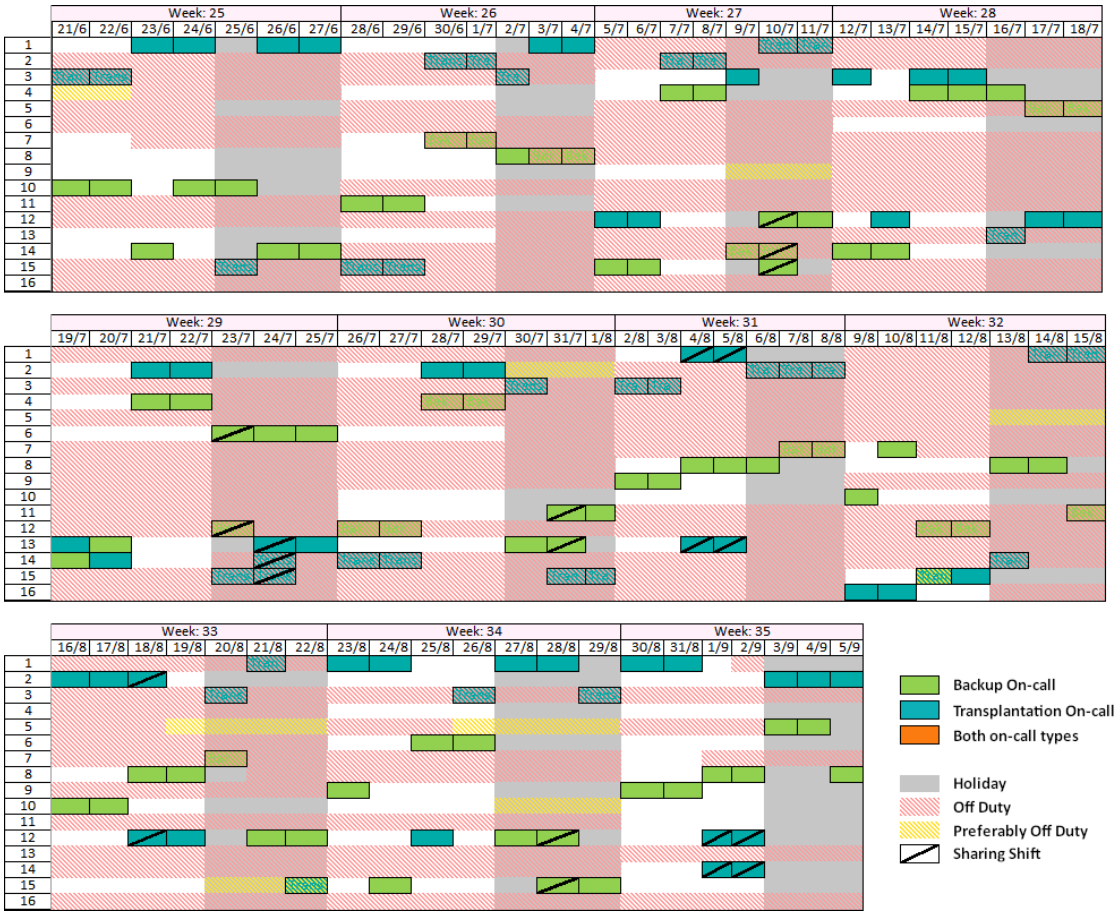


Figure 6.6: The schedule generated using the genetic algorithm. Rows represent staff members and columns represent dates.

7
Discussion

The problem of shift scheduling has been addressed in this project by creating a mathematical model that describes the intricate components and constraints. This has been used in two optimization methods and applied to a case scenario. Discussions regarding these areas are found in this chapter.

7.1 Mathematical Model

The mathematical model consists of 10 constraint types that are designed with not only the health sector in mind, but most other areas where shift scheduling is used. Hence, a large degree of generality has been used to define them in order to cover a wider range of requirements. This has lead to each one of the constraint types being able to model many and sometimes very different requirements. Although, a disadvantage with designing the constraints with such a high level of generality is that it in many ways leads to less efficient implementations. Several of the constraints will typically be used in one way. For instance, it is likely that most schedules will require that no shifts are unassigned. The implementation of the absolute rule that no shifts are allowed to be unassigned would be more efficient than the implementation of the constraint that is intended to model this requirement, the *Assignment of Shifts* constraint.

It is important to note that the primary sources of inspiration for the types of requirements that can be modelled using the constraints are from previous research and the on-call scheduling at the pediatric ward. Therefore, it is highly likely that there are a whole host of requirements that cannot be modelled using the constraints. For example, it is not possible to model shift-patterns such as the Monday-Thursday-Saturday pattern since no constraints exists for patterns that are not consecutive. This does not necessarily mean that the system cannot be of use, as it is possible to use the provided constraints to generate a schedule and then adjust it manually, thus saving the scheduler time. To further expand the range of requirement that the system is able to model, these requirements must first be determined. In order to do that, further implementation on a variety of scenarios is needed.

7.2 Z3

The theorem solver Z3 managed to find a feasible solution that satisfied all hard constraints as well as several soft constraints in a reasonably fast time. Benchmarking

tests for two scenarios showed that it was much faster for Z3 to solve the problem compared to the time it took to compile the constraints. The fast computation time for this application can be traced back to the CDCL algorithm that targets clauses of conflict and incorporates the conflict in the next implementation, hence cutting off infeasible solutions fast. Mostly using Boolean variables, propositional logic and arithmetic as modelling language may have contributed to the acceleration of the process immensely since they are the fundamental syntaxes of Z3.

7.2.1 Computation Time

During the first iterations of the program, binary variables were used as the modelling language for the decision variables instead of Boolean variables. It proved to greatly slow down the program and switching to Boolean variables rapidly increased the computation time, which is not peculiar since SAT-solvers are optimized for solving Boolean satisfaction problems. Other conclusions that were made was for example that integers, instead of floats, proved to speed up the solving time in some cases by a significant amount.

Z3 uses a variety of algorithms depending on the type of constraints that are declared. It is hard to say if Z3 uses Simplex for the linear arithmetic constraint or cuts and branching since what happens inside the solver is not easily deduced. Benchmarking of different operators were mainly presented in chapter 4. The benchmark tests that were presented demonstrated different results depending on the constraint. More tests need to be conducted together with statistical analysis to determine if there are any correlation between implementation and computation time. The scenarios that were used did not include many variables since the summer period is the shortest period of the year. The first three constraints are also straightforward and do not include many parameters, changes made in the implementation of constraints are therefore not particularly influential.

The fast computation time can be traced to having relatively few decision variables together with rather constricting constraints that makes the search space much smaller. Z3 tends to explode exponentially depending on how many variables are included. It is therefore hard to predict how much time the program will need to solve larger problems with more staff members or shifts.

The use of soft instead of hard constraints also slowed down the computation time. Soft constraints turns a problem, which can output any solution as long as it abides the constraints, into an optimization problem where as many soft constraints as possible should be satisfied. The conversion from hard to soft initiates extra steps for Z3, which includes activating the OptSMT for linear arithmetic and MaxSMT for soft constraints, a process that is more time-consuming than just checking whether the formulas are satisfiable or not.

7.2.2 Improvements

The shift durations in the case were all equal to one day and the scheduling program was therefore never tested on shorter shifts, for example shift of 8 hours. Constraints were formulated and constructed such that it would be possible to input shifts that lasts for a few hours but it has never been implemented and tested. The user interface for the generated schedule could be improved to present a schedule which is more easily readable for hourly shifts, since the shifts are currently only presented as daily shifts. The program was also only tested by one scheduler and was only used for generating the summer schedule. More implementations and tests have to be done in order to find improvements and errors.

Currently, the state of the program does not aid in finding out which constraints that make the model unsatisfiable, if that were the case. As it is now, the scheduler must use logical reasoning to evaluate which constraint makes the model unsatisfiable. Creating constraints can hence be tricky and the scheduler must therefore have some knowledge about how the constraints interact with each other. For example, if the scheduler adds a hard constraint that all shifts need to be assigned and adds another hard constraint that no one should work on Mondays, then the model will output that it is unsatisfiable without describing why. Part of the future work would therefore include creating a function that informs which constraint that should be relaxed to soft constraints or removed entirely. A solution could be to make use of the built-in function `s.unsat_core()` which extracts the unsatisfiable cores of some asserted constraint. Such a function would improve many different perspectives of the scheduling program. For example, the scheduler would be able to set all the constraints to hard and then use the unsatifiable core to find out which constraints to relax to find a feasible solution.

Other relevant built-in functions are the commands `s.push()` and `s.pop()` which can be used for adding new constrains to an existing model. The commands are suitable for situations when one would like to save a current model solution but also try to add another constraint that should be considered, e.g. if someone calls in sick. The method was never tested due to the time frame of the project but it is believed to perhaps make the process of finding a feasible solution quicker when new constraints are added.

7.3 Genetic Algorithm

As is clearly seen by the performance of the genetic algorithm, in section 5.2, and the resulting schedule for the case in figure 5.1, it does not perform well enough to have any practical usefulness. After an hour of running, which is already a significant amount of time, the algorithm was not able to generate a feasible schedule that fulfills all hard constraints. There are many possible reasons for this, which will be discussed in this section.

Local optima in the objective function play a large role in the performance of the

genetic algorithm. The objective function, or fitness score, is designed in a way where each individual constraint makes its contribution to a candidate solutions final value. This is believed to cause local maxima where infeasible solutions exist. For example, an assignment of a shift to a staff member that fulfills several constraints but violates another can be at a local optima. This happens if any other assignment leads to a lower fitness score, hence the algorithm is incentivized to keep the assignment even though it violates a constraint. The objective function is therefore essential in guiding the optimization, yet is poorly equipped to handle multiple constraints where some must be fulfilled. Additionally, as the constraint fitness scores are defined, they have a large difference in scale. The *Assignment Fractions* constraint fitness score, for instance, falls in the range from 0 to 1 per staff member which is well below that for other constraints. This causes some constraints to have a larger impact on the fitness score and therefore be prioritised. In the best case, an objective function is always increasing toward feasible solutions, but such a function has proven itself difficult to design.

One of the strengths of genetic algorithms are that their stochasticity aids in avoiding local optima. The point of having a population of candidate solutions is, after all, to increase the genetic diversity and provide the optimizer with multiple varying solutions. In the case of shift scheduling, the number of possible solutions is so astronomical that a large population size is even more advantageous as a larger region is then possible to explore. However, the drawback of a larger population is the additional calculations that must be performed, the effect is roughly proportional to the population size. This means that every candidate solution in every generation has its computational cost. In the case of the generated schedule in figure 5.1, it is clear that there are multiple possibilities for improvement. For example, during week 32, the *backup on-call* shift on the 10th of August can be reassigned to staff member 10 which would lead to a higher fitness score. Likewise for the *backup on-call* shift on the 24th of August to staff member 9. The question is, why have random mutations not corrected these assignments? The answer is believed to be that there simply have not been enough randomly performed mutations for these specific ones to be selected. For every random mutation that leads to an increase in fitness score, there are countless that reduce it instead and subsequently are not passed on to the next generation. If the computational cost for every candidate solution were lower, more mutations could be done using a larger population in a given amount of time. Inversely, it can be said that the time it takes to process a candidate solution is too high for the mutations to be performed through pure randomness.

In the case presented in chapter 6, there are 16 staff members and 154 shifts. This means that the probability for a mutation to randomly occur on one specific assignment when using a standard mutation procedure, with the parameter for the average number of mutations $n_m = 1$, is only $1/(16 \cdot 154) \approx 0.04\%$. If all candidate solutions were identical in a population of 30, the size that was used to generate the schedule, then the probability of at least one being mutated at a specific gene is only approximately 1.2% per generation. This puts the size of the search space into perspective and also provides an understanding for how difficult it becomes

for random mutations to further improve a schedule as it becomes more and more optimized. A conclusion from this can be made that the search space for a shift scheduling problem, even for one with relatively few staff members and shifts as is the case here, borders on the capabilities of genetic algorithms. The use of *targeted mutations* improves the speed of the optimization somewhat, as seen in section 5.2, and even more so the use of *guided mutations*. These help by directing the mutations, removing an element of randomness and incorporating the constraints placed on the schedule at a key stage in the optimization. This can be likened to giving the optimizer a flash light, as opposed to letting it stumble around in the dark and punishing it when it takes a step in the wrong direction. Despite the relative success that the modified mutation processes have shown, it can be concluded that they are not sufficient and that the degree of influence that the constraints have on the mutations must be higher if there is to be any hope of success. This is something that has been included in many developed methods, such as those addressed in section 1.1, as well as the relation between strictly adhering to feasible solutions and tolerating infeasible solutions in order to increase the exploration of the search space.

The recommendation given here regarding future work on genetic algorithms does not only apply to shift scheduling but to all optimization problems that contain hard and soft constraints, which is to develop methods capable of solving such problems in a robust and standardized way. It is our understanding that the solution space for a problem with hard and soft constraints will contain many feasible and infeasible solutions. The goal is essentially to explore all feasible solutions to find those that satisfy most soft constraints. For the common cases within shift scheduling, almost any feasible schedule is only one assignment away from becoming infeasible. This can be visualized as the solution space being speckled with feasible solutions rather than having clearly defined regions of feasibility. For genetic algorithms that use purely randomized mutation, this causes problems since most mutations will result in infeasible solutions. In many cases, multiple specific mutations are required for the schedule to remain feasible, but the probability of that happening is exceedingly low. More deliberate mutations that ensure feasibility without hindering the exploration of the solution space is the key to genetic algorithms success with these types of problems. There is, to our knowledge, no standard operation within the area of evolutionary algorithms the addresses this problem and would therefore require more attention. The benefit of such a method would be a whole range of new problems where genetic algorithms could be applied.

7.4 Comparison

For the case described in chapter 6, the Z3 solver showed superior results compared to the genetic algorithm. It was able to generate a feasible schedule in a relatively short time while the genetic algorithm could not, even with several orders of magnitude more time. It can be concluded that this is due to the solution space being dominated by infeasible solutions. Z3 is much better equipped to handle hard constraints that exclude many solutions, and in many cases this leads to it finding solutions even faster than otherwise. Genetic algorithms, on the other hand, are

not able to distinguish between feasible and infeasible solutions in the same way and therefore have a hard time navigating to feasible solutions though stochastic mutations.

Despite the results, there are several advantages with genetic algorithms over the Z3 solver. When the size of the problem grows, such as if there are more staff members and shifts than there are in the case of chapter 6, then it becomes difficult to predict how much time the Z3 solver needs to find a solution. Genetic algorithms give the possibility of displaying information regarding its progress as well as the ability to halt the optimization prematurely if its best current solution is good enough. This is especially useful when the problem includes many soft constraints, as the Z3 solvers performance deteriorates when the solution space contains many feasible solutions with few that fulfill all soft constraints.

Another advantage genetic algorithms have over Z3 is the relative freedom when it comes to modelling of the constraints. Genetic algorithms are not bound to first-order logic since they typically use objective functions, making it possible to include many other constraints modelled with higher-order logic.

For the use in health care and similar environments, the SMT approach is considered to be the most appropriate method as it displays the best performance in the case of chapter 6, which is representative of the typical problem size and complexity. If the upper limit of the Z3 solvers capabilities are met, a hybrid solution using both constraint programming to reduce the solution space size and stochastic optimization methods to explore the remaining regions could offer more potential than either methods by themselves.

8

Conclusion

The problem of shift scheduling has been around for decades and refers to the procedure of assigning staff members to shifts while satisfying a range of requirements. These requirements can take many forms and address various aspects of a schedule, from the maximum number of consecutive shifts to the level of workload fairness among the staff. A defining characteristic of shift scheduling problems is their immense complexity, even the most modest of schedules create solution spaces of astronomical sizes. This study has attempted to define and model common requirements placed on schedules with a mathematical model, as well as create a suitable scheduling system by comparing two different optimization methods.

A mathematical model was developed with regards to the research question *"What types of constraints must be included in the mathematical model to describe the scheduling optimization problem"*. The model consists of 10 unique constraint types that are capable of implementing many common schedule requirements. These constraints have been designed with a maximum level of abstractness in order to expand the range of requirements that they are capable of modelling. The system must further be applied to different scenarios to find weaknesses in the system's architecture. For example, there may be requirements that are outside the framework of what the mathematical model is able to implement, such as shift patterns that do not occur on consecutive days.

Using the mathematical model, schedules were generated using both the SMT-solver Z3 and a modified genetic algorithm in order to answer the second research question, *"Which optimization method or methods can generate the most satisfactory schedules and are most suitable for the current application?"*. The system was applied to a test case that included creating a summer-schedule for on-call doctors at Sahlgrenska University Hospital. Both methods were implemented on the case and the outcome showed significant difference in performance and generated results. Z3 displayed a faster solving time and resulted in a schedule where all hard constraints and several of the soft constraints were satisfied. Z3 makes efficient use of a learning algorithm based on CDCL, which targets conflicting clauses in the constraint formula and applies it to the solving process, enabling fast correction of infeasible solutions. The genetic algorithm, on the other hand, was significantly slower and was not able to generate a schedule that did not violate several constraints. It is concluded that genetic algorithms are poorly equipped to solve shift scheduling problems where the solution space is large and dominated by infeasible solutions. This is because the algorithm has difficulty in navigating toward feasible solutions

using purely randomized mutations. Two mutation procedures were developed, *targeted mutations* and *guided mutations*, that remove some of the stochasticity and incorporate the constraints into the mutations. This improved the speed of the algorithm significantly, yet was not sufficient to solve the problem within a reasonable time.

With the results of this work, shift schedules can be generated that abide several distinct requirements. The system has a practical use in not only the industry that it was designed for, but also in any area where shift based scheduling is used. It is capable of generating a schedule in a matter of seconds and far outperforms self-scheduling.

Bibliography

[1] Krzysztof Apt. "Constraint programming in a nutshell". In: *Principles of Constraint Programming*. Cambridge University Press, 2003, pp. 54–81. DOI: 10.1017/CBO9780511615320.003.

[2] Clare L Bambra et al. "Shifting schedules: the health effects of reorganizing shift work". In: *American journal of preventive medicine* 34.5 (2008), pp. 427–434.

[3] Clark Barrett and Cesare Tinelli. "Satisfiability modulo theories". In: *Handbook of Model Checking*. Springer, 2018, pp. 305–343.

[4] Clark Barrett et al. *Satisfiability Modulo Theories, volume 185 of Frontiers in Artificial Intelligence and Applications*. 2009.

[5] Nikolaj Bjørner et al. "Programming Z3". In: *International Summer School on Engineering Trustworthy Software Systems*. Springer. 2018, pp. 148–201.

[6] Oscar Boldt-Christmas et al. *Tid till vård ger vård i tid Hur möjliggör vi en bättre användning av läkarens tid och en ökad produktivitet?* Tech. rep. July 2019.

[7] Jonathan P Bowen, Zhiming Liu, and Zili Zhang. *Engineering Trustworthy Software Systems*. Springer, 2017.

[8] Peter Brucker, Rong Qu, and Edmund Burke. "Personnel scheduling: Models and complexity". In: *European Journal of Operational Research* 210.3 (2011), pp. 467–473.

[9] Edmund K Burke et al. "The state of the art of nurse rostering". In: *Journal of scheduling* 7.6 (2004), pp. 441–499.

[10] Chiara Dall'Ora et al. "Characteristics of shift work and their impact on employee performance and wellbeing: A literature review". In: *International Journal of Nursing Studies* 57 (2016), pp. 12–27. ISSN: 0020-7489. DOI: https://doi.org/10.1016/j.ijnurstu.2016.01.007. URL: https://www.sciencedirect.com/science/article/pii/S0020748916000080.

[11] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver". In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.

[12] M. Erhard et al. "State of the art in physician scheduling". In: *Eur. J. Oper. Res.* 265 (2018), pp. 1–18.

[13] Paola Ferri et al. "The impact of shift work on the psychological and physical health of nurses in a general hospital: a comparison between rotating night shifts and day shifts". In: *Risk management and healthcare policy* 9 (2016), p. 203.

[14] Daniel J. Gottlieb et al. “Effect of a Change in House Staff Work Schedule on Resource Utilization and Patient Care”. In: *Archives of Internal Medicine* 151.10 (Oct. 1991), pp. 2065–2070. ISSN: 0003-9926. DOI: 10.1001/archinte.1991.00400100131022. eprint: https://jamanetwork.com/journals/jamainternalmedicine/articlepdf/615645/archinte_151_10_022.pdf. URL: https://doi.org/10.1001/archinte.1991.00400100131022.

[15] J Malcolm Harrington. “Health effects of shift work and extended hours of work”. In: *Occupational and Environmental medicine* 58.1 (2001), pp. 68–72.

[16] Stefaan Haspeslagh et al. “The first international nurse rostering competition 2010”. In: *Annals of Operations Research* 218.1 (2014), pp. 221–236.

[17] Tan Li June et al. “Implementation of Constraint Programming and Simulated Annealing for Examination Timetabling Problem”. In: *Computational Science and Technology*. Springer, 2019, pp. 175–184.

[18] Donald E Knuth. “Two notes on notation”. In: *The American Mathematical Monthly* 99.5 (1992), pp. 403–422.

[19] Daniel Kroening and Ofer Strichman. *Decision procedures*. Springer, 2016.

[20] Robin Moser. *Exact Algorithms for Constraint Satisfaction Problems*. Logos Verlag Berlin GmbH, 2013, pp. 1–10.

[21] J.C. Nash. “The (Dantzig) simplex method for linear programming”. In: *Computing in Science Engineering* 2.1 (2000), pp. 29–31. DOI: 10.1109/5992.814654.

[22] Javier Puente et al. “Medical doctor rostering problem in a hospital emergency department by means of genetic algorithms”. In: *Computers & Industrial Engineering* 56.4 (2009), pp. 1232–1242.

[23] Tapabrata Ray et al. “Infeasibility driven evolutionary algorithm for constrained optimization”. In: *Constraint-handling in evolutionary optimization*. Springer, 2009, pp. 145–165.

[24] Sabino Francesco Roselli. “On Scheduling Using Optimizing SMT-Solvers”. PhD thesis. Department of Electrical Engineering, Chalmers University of Technology, 2020.

[25] Sabino Francesco Roselli, Kristofer Bengtsson, and Knut Åkesson. “SMT solvers for job-shop scheduling problems: Models comparison and performance evaluation”. In: *2018 IEEE 14th International Conference on Automation Science and Engineering (CASE)*. IEEE. 2018, pp. 547–552.

[26] Hemant Kumar Singh, Md Asafuddoula, and Tapabrata Ray. “Solving problems with a mix of hard and soft constraints using modified infeasibility driven evolutionary algorithm (IDEA-M)”. In: *2014 IEEE Congress on Evolutionary Computation (CEC)*. IEEE. 2014, pp. 983–990.

[27] Ricardo Soto et al. “Nurse and paramedic rostering with constraint programming: A case study”. In: *Romanian Journal of Information Science and Technology* 16.1 (2013), pp. 52–64.

[28] Christos Valouxis et al. “A systematic two phase approach for the nurse rostering problem”. In: *European Journal of Operational Research* 219.2 (2012), pp. 425–433.

[29] Mattias Wahde. *Biologically inspired optimization methods: an introduction*. WIT press, 2008.

[30] Tse-Chiu Wong, Mai Xu, and Kwai-Sang Chin. “A two-stage heuristic approach for nurse scheduling problem: A case study in an emergency department”. In: *Computers & Operations Research* 51 (2014), pp. 99–110.

A

Implementation of the Constraints in the Genetic Algorithm

Every constraint contributes to both the fitness value F_i and the mutation matrix M^i for each candidate solution i, but how they do this depends on the type of constraints that they are and is described below. All descriptions below are to be taken in the context of the definitions of the corresponding constraints in chapter 3. All implementations of the constraints are done in largely the same way as they are defined in chapter 3, but some differences occur for the sake of computational efficiency.

Assignment of Shifts

For constraint j of the type *Assignment of Shifts*, the violation severity is defined as

$$v_{i,j} = \max\left(x - |\bar{\mathcal{S}}|, 0, |\bar{\mathcal{S}}| - y\right) \tag{A.1}$$

where $\bar{\mathcal{S}} \subseteq \tilde{\mathcal{S}}$ is the set of shifts that are not assigned someone in $\tilde{\mathcal{P}}$. The constraint fitness score is then

$$\delta_{i,j} = -v_{i,j}. \tag{A.2}$$

If $|\bar{\mathcal{S}}| < x$, then the values of the mutation matrix elements $M^i_{p,s}$ for all shifts $s \in \tilde{\mathcal{S}} \setminus \bar{\mathcal{S}}$ and staff members $p \in \tilde{\mathcal{P}}$ are incremented by 1. Otherwise, if $y < |\bar{\mathcal{S}}|$ then the values of the mutation matrix elements $M^i_{p,s}$ for all shifts $s \in \bar{\mathcal{S}}$ and staff members $p \in \tilde{\mathcal{P}}$ are incremented by 1 instead.

Shared Shifts

For constraint j of the type *Shared Shifts*, the constraint violation severity is defined as

$$v_{i,j} = \max\left(x - |\bar{\mathcal{S}}|, 0, |\bar{\mathcal{S}}| - y\right) \tag{A.3}$$

where $\bar{\mathcal{S}} \subseteq \tilde{\mathcal{S}}$ is the set of shifts that are assigned an invalid staff group. The constraint fitness score is then

$$\delta_{i,j} = -v_{i,j}. \tag{A.4}$$

If $|\bar{\mathcal{S}}| < x$ then the values of the mutation matrix elements $M^i_{p,s}$ for all shifts $s \in \tilde{\mathcal{S}} \setminus \bar{\mathcal{S}}$ and $p \in \mathcal{P}$ are incremented by 1. Otherwise, if $y < |\bar{\mathcal{S}}|$ then the values of the

mutation matrix elements $M^i_{p,s}$ for all shifts $s \in \bar{\mathcal{S}}$ and $p \in \mathcal{P}$ are incremented by 1 instead.

Qualified Assignments

For constraint j of the type *Qualified Assignments*, the constraint violation severity is defined as

$$v_{i,j} = \max\left(x - |\bar{\mathcal{S}}|, 0, |\bar{\mathcal{S}}| - y\right) \tag{A.5}$$

where $\bar{\mathcal{S}} \subseteq \tilde{\mathcal{S}}$ is the set of shifts with unqualified staff members assigned. The constraint fitness score is then

$$\delta_{i,j} = -v_{i,j}. \tag{A.6}$$

If $|\bar{\mathcal{S}}| < x$ then the values of the mutation matrix elements $M^i_{p,s}$ for all shifts $s \in \tilde{\mathcal{S}} \setminus \bar{\mathcal{S}}$ and $p \in \mathcal{P}$ are incremented by 1. Otherwise, if $y < |\bar{\mathcal{S}}|$ then the values of the mutation matrix elements $M^i_{p,s}$ for all shifts $s \in \bar{\mathcal{S}}$ and $p \in \mathcal{P}$ are incremented by 1 instead.

Overlapping Shifts

For constraint j of the type *Overlapping Shifts*, the constraint violation severity is defined as

$$v_{i,j} = \max\left(x - c, 0, c - y\right) \tag{A.7}$$

where c is the number of times a staff member in $\tilde{\mathcal{P}}$ has been assigned unsanctioned overlapping shifts. The constraint fitness score is then

$$\delta_{i,j} = -v_{i,j}. \tag{A.8}$$

For every overlapping shift combination $o \in \mathcal{O}$, staff member $p \in \tilde{\mathcal{P}} \setminus \mathcal{O}^{AS}_o$ and shift in $s \in \mathcal{O}^{OC}_o$, if $c < x$ and $\neg A_{p,s}$, then mutation matrix element $M^i_{p,s}$ is incremented. If instead $y < c$, then only if $\bigwedge_{s' \in \mathcal{O}^{OC}_o} A_{p,s'}$ is $M^i_{p,s}$ incremented.

Staff Combination Assignments

For constraint j of the type *Staff Combination Assignments*, the constraint violation severity is defined as

$$v_{i,j} = \max\left(x - |\bar{\mathcal{S}}|, 0, |\bar{\mathcal{S}}| - y\right) \tag{A.9}$$

where $\bar{\mathcal{S}} \subseteq \tilde{\mathcal{S}}$ is the set of shifts where all staff members in $\tilde{\mathcal{P}}$, and only them, are assigned to. The constraint fitness score is then

$$\delta_{i,j} = -v_{i,j}. \tag{A.10}$$

If $|\bar{\mathcal{S}}| < x$, then all mutation matrix elements $M^i_{p,s}$ where $p \in \tilde{\mathcal{P}}$ and $s \in \tilde{\mathcal{S}} \setminus \bar{\mathcal{S}}$ are incremented. If instead $y < |\bar{\mathcal{S}}|$, then the mutation matrix elements $M^i_{p,s}$ where $p \in \tilde{\mathcal{P}}$ and $s \in \bar{\mathcal{S}}$ are incremented.

Consecutive Days

The *Consecutive Days* constraint is slightly more complicated to define the severity and fitness score for. Starting with the severity, any situation where the number of days in a series of consecutive days of work is not between x and y violates the constraint. Therefore, the severity for constraint j of the type *Consecutive Days* is

$$v_{i,j} = \sum_{c=1}^{C} \max\left(x - l_c, 0, l_c - y\right) \tag{A.11}$$

where C is the number of individual occasions where someone works consecutive working days (also including single days), and l_c is equal to the length of the consecutive work series c. For example, the schedule seen in figure A.1 shows which days four staff members work on. If the constraint values are $x = 3$ and $y = 5$, then all consecutive working days with a length shorter than 3 and longer than 5 will contribute to the severity. In this case, the severity is $v_{i,j} = \underbrace{3-2}_{A} + \underbrace{6-5}_{C} + \underbrace{3-1+3-2}_{D} = 5$.

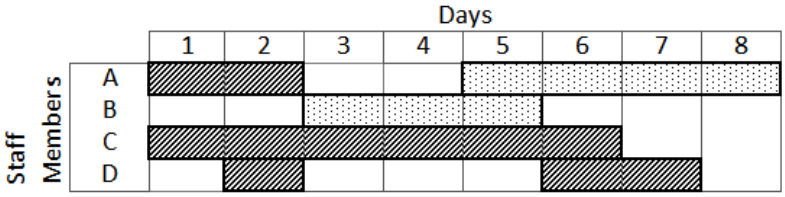


Figure A.1: The days that the staff members work. The consecutive working days with length shorter than $x = 3$ or longer than $y = 5$ (diagonal stripes) contribute to the constraint severity, unlike the consecutive days that are between 3 and 5 (dotted).

The *Consecutive Days* constraints fitness score is not simply the negative of the severity, as it is for the constraints above. The reason behind this is because if a staff member is assigned a single shift through a random mutation, as will often be the case, then that assignment will create a consecutive work series of length 1. If a *Consecutive Days* constraint has the lower bound $1 < x$, then the assignment of the shift will cause the severity to increase and subsequently the fitness to decrease. This will create a local minimum where no shifts are assigned to any staff members and therefore hinder the optimization. Instead, the fitness score is designed to be unaffected when a consecutive work series length is 1, and increase as the length increases toward x. Once the length of the consecutive workday series is between or equal to x and y, the fitness stays constant until exceeding y, at which point the fitness will begin to decrease again. See figure A.2 for an example. The constraint fitness score is therefore

$$\delta_{i,j} = \sum_{c=1}^{C} \max\left(x - 1, 0\right) - \max(x - l_c, 0, l_c - y) \tag{A.12}$$

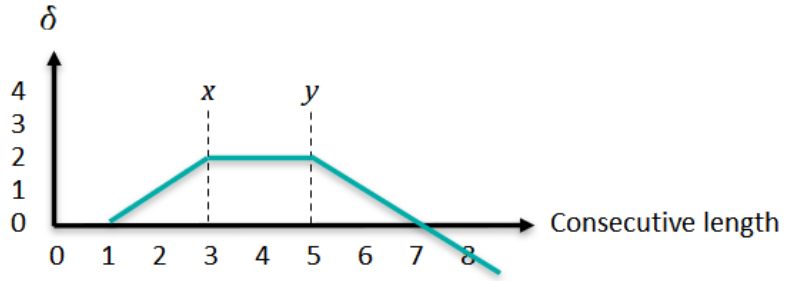


Figure A.2: A graph showing how the Consecutive Days constraint fitness score changes as the length of consecutive workdays increase, where the constraint values are set to $x = 3$ and $y = 5$.

The mutation matrix is altered for every consecutive work series c that staff member $p \in \tilde{\mathcal{P}}$ is working. If $l_c < x$, then for all shifts $s \in \tilde{\mathcal{S}}$ that is either part of or occurs on the day before or after the consecutive work series, the mutation matrix elements $M^i_{p,s}$ are incremented. This is so that the optimization is incentivized to either increase the the length l_c or remove the consecutive work series entirely. If instead $y < l_c$, then only for the shifts $s \in \tilde{\mathcal{S}}$ that are part of the consecutive series are the mutation matrix elements $M^i_{p,s}$ incremented.

Before and After Consecutive Days

For constraint j of type *Before and After Consecutive Days*, the severity is

$$v_{i,j} = \sum_{c=1}^{C} |\bar{\mathcal{S}}^c_n| + |\bar{\mathcal{S}}^c_m| \tag{A.13}$$

where $\bar{\mathcal{S}}^c_n \subseteq \tilde{\mathcal{S}}_n$ are all shifts on the preceding n days and $\bar{\mathcal{S}}^c_m \subseteq \tilde{\mathcal{S}}_m$ are all shifts on the following m days of consecutive work series c that the person who is assigned to c is also assigned to. The fitness score is

$$\delta_{i,j} = -v_{i,j} \tag{A.14}$$

The mutation matrix elements $M^i_{p,s}$ are incremented for every shift $s \in \bar{\mathcal{S}}^c_n \cup \bar{\mathcal{S}}^c_m$ and staff member $p \in \tilde{\mathcal{P}}$ assigned to c, for every consecutive work series c.

Assignment Fractions

For constraint j of the type *Assignment Fractions*, the severity is

$$v_{i,j} = n \tag{A.15}$$

where n is the number of staff members in $\tilde{\mathcal{P}}$ who's workload fractions fall outside of x and y. The constraint fitness score is

$$\delta_{i,j} = -\sum_{p \in \tilde{\mathcal{P}}} \max\left(x - \gamma_p, 0, \gamma_p - y\right) \tag{A.16}$$

where γ_p is the workload fraction for staff member p. If $\gamma_p < x$ for staff member $p \in \tilde{\mathcal{P}}$, then for all shifts $s \in \{s' \in \tilde{\mathcal{S}} | \neg A_{p,s'}\} \cup \{s' \in \mathcal{S} \setminus \tilde{\mathcal{S}} | A_{p,s'}\}$, the mutation matrix element $M^i_{p,s}$ is incremented. If instead $y < \gamma_p$, then all the mutation matrix elements $M^i_{p,s}$ for shifts $s \in \{s' \in \tilde{\mathcal{S}} | A_{p,s'}\} \cup \{s' \in \mathcal{S} \setminus \tilde{\mathcal{S}} | \neg A_{p,s'}\}$ are incremented instead.

Fair Workload

Similar to the *Assignment Fractions* constraint type, constraint j of type *Fair Workload* has the severity

$$v_{i,j} = n \tag{A.17}$$

where n is the number of staff members who's normalized workload deviation falls outside of x and y. The constraint fitness score is

$$\delta_{i,j} = -\sum_{p \in \tilde{\mathcal{P}}} \max\left(-\delta_{\max} - \alpha_p, 0, \alpha_p - \delta_{\max}\right) \tag{A.18}$$

where α_p is the normalized workload deviation for staff member p and $\delta_{\max}$ is the maximum workload deviation, as defined in chapter 3. For each staff member $p \in \tilde{\mathcal{P}}$, if $\alpha_p < -\delta_{\max}$ then the mutation matrix elements $M^i_{p,s}$ for shifts $s \in \tilde{\mathcal{S}}$ where $\neg A_{p,s}$ are incremented. If instead $\delta_{\max} < \alpha_p$ then the mutation matrix elements $M^i_{p,s}$ for shifts $s \in \tilde{\mathcal{S}}$ where $A_{p,s}$ are incremented.

Consecutive Shift Types

Finally, constraint j of type *Consecutive Shift Types* has the severity

$$v_{i,j} = \sum_{p \in \tilde{\mathcal{P}}} |\bar{\mathcal{S}}^p| \tag{A.19}$$

where $\bar{\mathcal{S}}^p \subseteq \tilde{\mathcal{S}}$ is the set of shifts that are assigned to staff member $p \in \tilde{\mathcal{P}}$ and are not preceded by a free day or a shift assignment with the same type. The constraint fitness is then

$$\delta_{i,j} = -v_{i,j} \tag{A.20}$$

For each staff member $p \in \tilde{\mathcal{P}}$, the mutation matrix elements $M^i_{p,s}$ for shifts $s \in \bar{\mathcal{S}}^p$ are incremented.

B

Additional Information Regarding the Case

B.1 Staff Information

Table B.1: Information about each staff and their preferences during the summer period. T represents the qualification for transplantation on-call and B is for the backup on-call.

Staff p	**Qualifications** $(\mathcal{S}_p^Q)$	**Workload Desired** (s_p^{dw})	**Preferably Off-duty**	**Off-duty**
1	T	1		week 27-30 2/9 week 32-33
2	T	1	30/7-1/8	week 25-28 week 31-32
3	T	0,2		Week 25-26 Week 29-35
4	T	0,1		All except 2 workdays week 32
5	B	1	21-22/6	23-27/6 16-18/7 week 29-32
6	B	0,75	9-11/7 13-15/8	19-22/8 26-29/8 week 28-31 All Workdays
7	B	1		21/6-11/7 30/7-22/8

8	B	1		All except (21-22/6, 28-29/6 9-10/8, 30-31/8)
9	B	1		3/7-1/8 21-29/8
10	B	1	9-11/7	week 28-30 week 32-33
11	B	0,5	27-29/8	week 25-28
12	B	0,8		week 25 week 27-29 week 31-34
13	T, B	1		week 25-26 week 29-32
14	T, B	1		2-4/7 week 27-28 6-8/8 week 32-35
15	T, B	1		28/6-11/7 26/7-1/8 9-29/8 23-25/7 week 31
16	T, B	1	11/8 20-22/8	week 25-26 week 28-31

DEPARTMENT OF ELECTRICAL ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY