Exact Methods for Extended Rotating Workforce Scheduling Problems

Abstract

In many professions daily demand for different shifts varies during the week. The rotating workforce scheduling problem deals with the creation of repeating schedules for such demand and is therefore of high practical relevance. This paper investigates solving this real-life problem with several new practically relevant features. This includes early recognition of certain infeasibility criteria, complex rest time constraints regarding weekly rest time, and optimization goals to deal with optimal assignments of free weekends. We introduce a state-of-the-art constraint model and evaluate it with different extensions. The evaluation shows that many real-life instances can be solved to optimality using a constraint solver. Our approach is under deployment in a state-of-the-art commercial solver for rotating workforce scheduling.

Introduction

In many professions different shifts are required to cover varying requirements including areas like health care, protection services, transportation, manufacturing or call centers. This problem may surface in many shapes, using different demands and constraints.

In several applications it can be beneficial to obtain a rotating schedule where each employee rotates through the same sequence of shifts and days off across several weeks, however, at different offsets within the rotation. As the design of shift schedules highly influences the work-life balance of the employees, such problems are subject to a wide range of constraints, dealing not only with the demand for employees in different shifts, but also legal and organizational constraints that determine allowed shift assignments.

Due to its importance there has been ongoing research on the rotating workforce scheduling (RWS) problem, and results have found their way into commercial software. The contributions of this work are twofold. We introduce and solve a new extended problem that includes several new additions based on the experience from working with these problems in practice. Extensions include new constraints for fast detection of infeasible instances, complex constraints to respect weekly rest times, as well as soft constraints optimizing free weekends in the schedule, turning the satisfaction problem into an optimization problem. To solve the problem we provide a new constraint model and implement it in the constraint modelling language MiniZinc.

Both the core model and the extended models are then evaluated on a standard set of benchmark instances based on real-life examples using the constraint solver Chuffed and compared to recent literature. The results show that the extended models greatly improve the handling of infeasible instances and allow to incorporate complex rest time constraints and optimization goals providing optimal solutions for the majority of the benchmark instances in short computational time. With our work we further improve the state-of-the-art solver for rotating workforce scheduling and enable this solver to be used in more complex real-life situations.

Related Work

Due to high practical relevance various versions of employee scheduling problems have been investigated for several decades. For an overview of existing literature refer to surveys like (Burke et al. 2004; Ernst et al. 2004; Van den Bergh et al. 2013; De Bruecker et al. 2015).

The rotating workforce scheduling problem can be classified as a single-activity tour scheduling problem with non-overlapping shifts and rotation constraints (Baker 1976; Restrepo, Gendron, and Rousseau 2016) and is known to be NP-complete (Chuin Lau 1996).

So far the problem has been addressed with a range of different methods. Complete approaches include a network flow formulation (Balakrishnan and Wong 1990), integer linear programming (Laporte, Nobert, and Biron 1980), several constraint programming formulations (Laporte 1999; Musliu, Gärtner, and Slany 2002; Laporte and Pesant 2004; Triska and Musliu 2011) and an approach with satisfiability modulo theories (Erkinger and Musliu 2017). There is also work on heuristic approaches (Musliu 2005; 2006), the creation of rotating schedules by hand (Laporte 1999), and using algebraic methods (Falcón et al. 2016).

The current state-of-the-art complete method for standard RWS was introduced by (Musliu, Schutt, and Stuckey 2018). It uses a solver independent formulation in the MiniZinc constraint language, either with a direct representation or using a regular automaton, and applies both the lazy clause generation solver Chuffed and the MIP solver Gurobi. It is the first complete method able to solve the standard benchmark set of 20 instances and introduces new benchmark instances that we also use for comparison.

Existing work on RWS mostly deals with the standard version of the problem, requiring any feasible solution, or delegates the selection of preferred solutions to the user in an interactive process (Musliu, Gärtner, and Slany 2002). While standard RWS already has practical relevance, the extensions allow to deal with more complex issues and provide solutions that are of higher value in real-life applications.

Problem Definition for Standard RWS

A rotating workforce schedule consists of the assignment of shifts or days off to each day across several weeks for a certain number of employees. Table 1 shows an example for four employees (or four equal-sized groups of employees), assigning the three shift types day shift (D), afternoon shift (A), and night shift (N). Each employee starts their schedule in a different row, moving from row i to row $i \bmod n + 1$ (where n is the number of employees) in the following week.

Empl.	Mon	Tue	Wed	Thu	Fri	Sat	Sun
1	D	D	D	D	N	N	-
2	-	-	A	A	A	A	N
3	N	N	-	-	D	D	D
4	A	A	N	N	-	-	-

Table 1: Example schedule for 4 employees

Problem Specification

We start by defining the basic version of the rotating workforce scheduling problem and recall definitions and notation by (Musliu, Gärtner, and Slany 2002) and (Musliu, Schutt, and Stuckey 2018). Extensions to the formulation are introduced in the next section. We define:

- n: Number of employees.
- w: Length of the schedule, typically $w = 7$ as the demands repeat in a weekly cycle. The total length of the planning period is $n \cdot w$, as each employee rotates through all n rows.
- $\mathbf{A}$: Set of work shifts (activities), enumerated from 1 to m, where m is the number of shifts. A day off is denoted by a special activity O with numerical value 0 and we define $\mathbf{A}^+ = \mathbf{A} \cup \{O\}$.
- R: Temporal requirements matrix, an $m \times w$-matrix where each element $R_{i,j}$ corresponds to the number of employees that need to be assigned shift $i \in \mathbf{A}$ at day j. The number of employees o_j that need to be assigned a day off on day j can be calculated by $o_j = n - \sum_{i=1}^{m} R_{i,j}$.
- ℓ_w and u_w: Minimal and maximal length of blocks of consecutive work shifts.
- ℓ_s and u_s: Minimal and maximal lengths of blocks of consecutive assignments of shift s given for each $s \in \mathbf{A}^+$.
- Forbidden sequences of shifts: Any sequences of shifts (like $N\ D$, a night shift followed by a day shift) that are not allowed in the schedule. This is typically required due to legal or safety concerns. In practice in is usually sufficient to forbid sequences of length 2 or sequences of length 3 where the middle shift is a day off. These are also the kind of restrictions used in the benchmark instances for rotating workforce scheduling.

  In our model, we use a set $\mathbf{F}_s^2 \subseteq \mathbf{A}$ for each $s \in \mathbf{A}$ to denote forbidden sequences of length 2, such that $x \in \mathbf{F}_s^2$ declares that shift x must not follow shift s.

  Forbidden sequences of length 3 are given as a set $\mathbf{F}_3$ of arrays of length 3 containing elements of $\mathbf{A}^+$. This definition could be extended to arbitrary lengths ℓ using corresponding sets $\mathbf{F}_\ell$.

The task is to construct a cyclic schedule S, represented as an $n \times w$-matrix, where each $S_{i,j} \in \mathbf{A}^+$ denotes the shift or day off that employee i is assigned during day j in the first period of the cycle. The schedule for employee i through the whole planning period consists of the cyclic sequence of all rows of S starting with row i.

Instead of the matrix representation, the same schedule can also be represented as an array T which is equal to the schedule of the first employee, where T_i denotes the shift assignment on day i with $1 \leq i \leq n \cdot w$. As the schedule is cyclic, we could choose any day in the schedule to correspond to the first element in T. We define the offset o with $0 \leq o < w$ to denote the position of the first element in T within the planning period.

Constraint Model

Our main model uses a direct representation of the constraints as described in this section. Some aspects are modelled in a different way compared to (Musliu, Schutt, and Stuckey 2018), most notably the different way to deal with cyclicity using the offset o.

For any array or matrix the indices are modulo its dimension. Within this description, such modulo operations are omitted for better readability. We define $\mathbf{N} = \{1, \ldots, n\}$, $\mathbf{W} = \{1, \ldots, w\}$ and $\mathbf{NW} = \{1, \ldots, n \cdot w\}$.

The following equations model the demand.

$$\sum_{i=0}^{n-1} (T_{d+w \cdot i} = s) = R_{s,d+o} \qquad \forall d \in \mathbf{W}, s \in \mathbf{A} \quad (1)$$

$$\sum_{i=0}^{n-1} (T_{d+w \cdot i} = O) = n - \sum_{i=1}^{m} R_{i,d+o} \qquad \forall d \in \mathbf{W} \quad (2)$$

Equation (1) models the demand for each day d and each shift type s. The left side counts occurrences of s on day d, the right side uses the offset to access the correct column of the demand matrix. Equation (2) is a redundant constraint that counts the number of day-off assignments for each day d. This is not necessary to obtain a complete model of the problem, however, constraint satisfaction solvers can benefit from such redundant definitions.

The next equations introduce symmetry breaking constraints which are also used to make dealing with the cyclic nature of the problem easier in following constraints.

$$T_1 \neq O \tag{3}$$
$$T_{n \cdot w} = O \tag{4}$$

Equations (3) and (4) declare that the first element of T has to hold a working shift, while the last element of T has to hold a day off. In principle any day of the planning period could be used as the first day as it is cyclic. Taking into account that every reasonable rotating workforce scheduling problem contains at least one working day and at least one day off, we can set the first element of T to align with the beginning of a working block.

This has two advantages. First, it eliminates several symmetric versions of the same solution, reducing cyclic occurrences of the same solution from $n \cdot w$ possible notations to the number of working blocks within the solution. Second, this guarantees that blocks of the same shift type or working blocks (consecutive days without day-off assignments) can never cycle across the end of T, eliminating the need to deal with cyclicity in their definition.

Next, constraints for the lengths of shift blocks and working blocks are defined.

$$\forall j \in \{1, \ldots, \ell_s - 1\} : T_{i+j} = s \qquad \forall s \in \mathbf{A}^+, i \in \mathbf{NW}, T_i = s, T_{i-1} \neq s \tag{5}$$

$$i + u_s > n \cdot w \vee \exists j \in \{\ell_s, \ldots, u_s\} : T_{i+j} \neq s \qquad \forall s \in \mathbf{A}^+, i \in \mathbf{NW}, T_i = s, T_{i-1} \neq s \tag{6}$$

$$\forall j \in \{1, \ldots, \ell_w - 1\} : T_{i+j} \neq O \qquad \forall i \in \mathbf{NW}, T_i \neq O, T_{i-1} = O \tag{7}$$

$$i + u_w > n \cdot w \vee \exists j \in \{\ell_w, \ldots, u_w\} : T_{i+j} = O \qquad \forall i \in \mathbf{NW}, T_i \neq O, T_{i-1} = O \tag{8}$$

Equation (5) defines the minimum block length for all shift types including day-off assignments. For all elements T_i containing shift s, where the block starts at i (corresponding to $T_{i-1} \neq s$), the next elements of T until the minimum length must also contain shift s. Equation (6) defines the maximum block length, stating that no later than u_s elements after the block start a different shift type has to occur. Additionally, if i is too close to the end of T, the block will end anyway, giving rise to the inequality part.

Equations (7) and (8) define the same constraints for working blocks, using ℓ_w and u_w as bounds and checking for any working shift ($\neq O$) instead of a specific shift type s.

Finally the forbidden sequences need to be modelled.

$$T_i \neq s \vee T_{i+1} \notin \mathbf{F}_s^2 \qquad \forall s \in \mathbf{A}, i \in \mathbf{NW} \tag{9}$$

$$\exists j \in \{1, \ldots, \ell\} : X_j \neq T_{i+j-1} \qquad \forall X \in \mathbf{F}_\ell, i \in \mathbf{NW} \tag{10}$$

Equation (9) models sequences of length 2, denoted by the set $\mathbf{F}_s^2$ of shift types not allowed to follow shift type s. Forbidden sequences of arbitrary length ℓ are modelled in (10), where for each possible match of each forbidden sequence at least one element must differ from the forbidden sequence.

Further symmetry breaking constraints might be applied to determine the offset o if certain conditions hold.

$$o = \min \left\{ d \in \mathbf{W} \;\middle|\; \sum_{s \in \mathbf{A}} D_{s,d} > \sum_{s \in \mathbf{A}} D_{s,d-1} \right\} - 1 \tag{11}$$

$$(\forall s \in \mathbf{A}, d \in \mathbf{W} : D_{s,d} = D_{s,d+1}) \rightarrow o = 0 \tag{12}$$

Equation (11) models the case that there is any day d where the previous day $d-1$ has a lower total demand for shifts, then at least one new shift block has to start on day d, making it possible to fix the offset. Note that (11) cannot be applied if the condition does not hold for any d, however, in that case (12) might be applied if the demand is constant for each shift. In this case week days are completely symmetrical, allowing to fix the offset to 0.

Problem Extensions

This section introduces several new extensions to the problem and provides the formal constraint model for them. The extensions cover different aspects of the problem and its solving process, dealing with the detection of infeasible instances, the introduction of complex rest time constraints and the optimization towards better scheduling of free weekends.

Detecting Infeasible Instances

The standard benchmark data set consists of 20 instances derived from real life scenarios, all of them admitting feasible solutions. However, the larger instance data set by (Musliu, Schutt, and Stuckey 2018) also includes infeasible instances. The results show that the solver Chuffed also used by us has difficulties identifying those instances. However, in practice it is important to provide fast feedback to the user when they give infeasible settings so that they can correct their input.

Two particular infeasibility tests are described in this section. As these are defined on input parameters only, several infeasible instances can be detected already while compiling the instance for the solver, while there is still one consistent formulation of the problem.

Infeasible Weekly Fluctuation. Consider the following demand for any shift s together with $\ell_s = 3$ and $u_s = 4$.

Mon	Tue	Wed	Thu	Fri	Sat	Sun
5	5	5	5	5	2	2

Table 2: Infeasible demand for shift s

Table 2 defines a common pattern for weekly demand, weekdays with more demand and a weekend with less demand. The increase in demand from Sunday to Monday requires at least three new work blocks of shift s to start on Monday, similarly the decrease from Friday to Saturday requires at least three work blocks of shift s to end on Friday. Next, as the minimum block length is 3, all three blocks starting on Monday and all three blocks ending on Friday span across Wednesday. On the other hand, all six blocks are distinct as the maximum block length is 4 and therefore no block can cover Monday to Friday at once. Therefore, at least six shifts of type s are required on Wednesday, which is higher than the demand and results in infeasibility of the instance.

More formally, this observation can be generalized as follows.

$$\forall j \in \{u_s+1,\dots,2\cdot\ell_s-1\}, k \in \{j-\ell_s,\dots,\ell_s-1\}: \\ R_{s,i+k} \geq R_{s,i} - R_{s,i-1} + R_{s,i+j-1} - R_{s,i+j} \\ \forall s \in \mathbf{A}, i \in \mathbf{W} \quad (13)$$

Equation (13) extends the example above for arbitrary block lengths. j iterates through possible distances between block starts and block ends (5 in the example), k iterates through days of guaranteed overlap (only Wednesday in the example). The check is performed for every shift type s and every possible start day i.

Bounding the Number of Blocks. Another observation is the fact that in a cyclic schedule the number of work blocks and the number of free blocks is equal. On the other hand, both for work blocks and free blocks a minimum and maximum number of blocks can be calculated from the required number of shifts and the allowed block lengths.

$$low_w = \left\lceil \frac{r}{u_w} \right\rceil \quad (14)$$

$$up_w = \left\lfloor \frac{r}{\ell_w} \right\rfloor \quad (15)$$

$$low_O = \left\lceil \frac{n\cdot w - r}{u_O} \right\rceil \quad (16)$$

$$up_O = \left\lfloor \frac{n\cdot w - r}{\ell_O} \right\rfloor \quad (17)$$

$$low = \max\{low_w, low_O\} \quad (18)$$

$$up = \min\{up_w, up_O\} \quad (19)$$

Equations (14) and (15) define lower and upper bounds for the number of work blocks, using the total demand for work shifts $r = \sum_{i=1}^{m}\sum_{j=1}^{w} R_{i,j}$. Equations (16) and (17) define lower and upper bounds for the number of free blocks. As the numbers of blocks need to be equal, (18) and (19) define the common bounds. Instances with $low > up$ can immediately be classified as infeasible.

Using Block Bounds for Redundant Constraints. We introduce two possibilities to use the block bounds for redundant constraints. The first uses a global cardinality constraint that takes four arguments: an array to operate on, an array of values to count in the first array, and lower and upper bounds for the corresponding values.

$$gcc_{lu}([(T_i \neq O \wedge T_{i-1} = O) - (T_i = O \wedge T_{i-1} \neq O) \\ \mid i \in \mathbf{NW}], [-1,1], [low, low], [up, up]) \quad (20)$$

Equation (20) defines an array holding -1 for the start of off-blocks and 1 for the start of work blocks and sets lower and upper limits for the counts of both of them. Note that this constraint does not enforce both counts to be equal, however, it propagates lower and upper bounds well to restrict the search space.

The second possibility uses counting arrays of length $n\cdot w$ for the number of days off C^O, the number of work days C^w, and the number of blocks C^b.

$$C_i^O = \begin{cases} 0 & \text{if } i = 1 \\ C_{i-1}^O + (T_i = O) & \text{otherwise} \end{cases} \quad (21)$$

$$C_i^w = \begin{cases} 1 & \text{if } i = 1 \\ C_{i-1}^w + (T_i \neq O) & \text{otherwise} \end{cases} \quad (22)$$

$$C_i^b = \begin{cases} 1 & \text{if } i = 1 \\ C_{i-1}^b + (T_{i-1} = O \wedge T_i \neq O) & \text{otherwise} \end{cases} \quad (23)$$

$$(low - C_i^b)\cdot \ell_O \leq n\cdot w - r - C_i^O \leq (up - C_i^b)\cdot u_O \\ \forall i \in \mathbf{NW}, C_i^b < C_{i+1}^b \quad (24)$$

$$(low - C_i^b)\cdot \ell_w \leq n\cdot w - C_i^w \leq (up - C_i^b)\cdot u_w \\ \forall i \in \mathbf{NW}, C_i^b < C_{i+1}^b \quad (25)$$

Equations (21), (22) and (23) count the number of days off, work days and blocks (counting every time a work block starts) up to each day $i \in \mathbf{NW}$. These counts are then used in (24) and (25) to bound the remaining number of off-blocks and work blocks after every day i where a new work block starts on the next day, using the bounds for the number of blocks.

Weekly Rest Time

While forbidden sequences of shifts can be used to handle minimum free time between consecutive shifts, work regulations often contain different, more complex regulations for free time. Real-world scenarios often need to consider a weekly rest time. Typically once a week a certain minimum amount of time has to be free without interruption. Further, it might be possible to have exceptions once every few weeks where the weekly rest time might be shorter according to certain rules.

Definition. The following requirements are defined to consider weekly rest times.

- g: Time granularity, given as the number of time slots per day, in this paper we use minute level granularity with $g = 1440$.
- $start_s$ and end_s: As weekly rest times consider the time between shifts, start and end times need to be defined for each shift. The times are given in minutes relative to the day the shift is assigned to, e.g., a night shift s ending at 6:00 on the next day has $end_s = 1800$.
- wr: Minimum weekly rest time (full weekly rest) in minutes, denoting the minimum time from the end of the last shift before the weekly rest time to the beginning of the next shift after the weekly rest time.
- A weekly rest needs to contain a full free day (0:00 to 24:00, i.e, no night shift from the previous day might overlap).

- A weekly rest is required in each calendar week (Monday to Sunday). Each rest period counts for the week where more than half of the rest is located, if the rest time is exactly split between two weeks, it counts for the later week.
- wr_{red}: Reduced minimum weekly rest time in minutes.
- e: Number of exceptions per sp weeks.
- sp: Span in weeks for the number of exceptions and the calculation of the average.
- Every week in the planning period needs to have a weekly rest of length at least wr with the exception of e times in a rolling horizon of sp weeks where the weekly rest can be reduced to wr_{red}, however, the average across sp weeks always needs to be at least wr.

Constraint Model. To capture the rest time between shifts an array $Rest$ of length $n \cdot w$ is defined. For a day off we define $start_O = g$.

$$Rest_i = \begin{cases} Rest_{i-1} + start_{T_i} & \text{if } T_{i-1} = O \\ g - end_{T_{i-1}} + start_{T_i} & \text{otherwise} \end{cases} \quad (26)$$

Equation (26) first considers the case $T_{i-1} = O$ where $Rest_{i-1}$ holds an intermediate rest time starting at the last shift and ending at the end of day $i-1$. In any case, $start_{T_i}$ holds the rest time from midnight until the start of the next shift, while $g - end_{T_{i-1}}$ holds the rest time from the previous shift until midnight.

Next, the individual requirements are modelled via a boolean matrix D of dimension $6 \times n \cdot w$.

$$D_{1,i} = (Rest_i \geq wr) \quad (27)$$

$$D_{2,i} = (T_i \neq O) \quad (28)$$

$$D_{3,i} = (T_{i-1} = O) \quad (29)$$

$$D_{4,i} = (end_{T_{i-2}} \leq g) \quad (30)$$

$$D_{5,i} = \left(\frac{Rest_i}{2} < g \cdot ((i + o - 1) \bmod 7) + start_{T_i} \right) \quad (31)$$

$$D_{6,i} = (Rest_i \geq wr_{red}) \quad (32)$$

Each of (26) to (32) is evaluated for $i \in \mathbf{NW}$. Equation (27) models the minimum weekly rest time, (32) the reduced minimum weekly rest time. (28) ensures that only the end of a rest period is considered, not intermediate results that are used in (26). Equations (29) and (30) make sure that the full free day is respected. If less than half of the rest is located in the previous week, $D_{5,i}$ is set to *true* in (31), the right side of the inequality specifies the time from the start of the current week to the start of the shift on day i.

The content of matrix D is also used to allow users to understand why a certain rest period does or does not qualify as a weekly rest period.

Now the position P_i of the weekly rest period for each week i can be determined, using $x = (i-1) \cdot w + j - o$.

$$P_i = \max\{0; j \mid D_{1,x} \wedge D_{2,x} \wedge D_{3,x} \wedge D_{4,x} \wedge (j \leq w) = D_{5,x}\} \qquad \forall i \in \mathbf{N} \quad (33)$$

Equation (33) defines the position of the weekly rest period for each week i by taking the latest rest period that qualifies as a proper weekly rest using the elements from D and is still assigned to the correct week. Here, $(j \leq w) = D_{5,x}$ makes sure that either the weekly rest still ends in the current week and is also assigned to the current week or it ends in the next week, but is not yet assigned to the week where it ends. This also guarantees that no j with $j > w + \frac{u_o}{2} + 1$ needs to be considered as the corresponding rest would be counted towards the following week for sure. In case no proper weekly rest is found, P_i is set to 0.

Exceptions. Now regular weekly rest for each week could be enforced simply requiring $P_i \neq 0$ for all i, however, we want to consider exceptions as follows.

$$\sum_{j=0}^{sp-1} (P_{i+j} = 0) \leq e \qquad \forall i \in \mathbf{N} \quad (34)$$

Equation (34) counts the number of weeks for each span of length sp where no proper weekly rest can be found.

To check the reduced weekly rest the position P_i^{red} for each week i is defined, using $x = (i-1) \cdot w + j - o$.

$$P_i^{red} = \begin{cases} P_i & \text{if } P_i \neq 0 \\ \max\{0; j \mid D_{6,x} \wedge D_{2,x} \\ \qquad \wedge (j \leq w) = D_{5,x}\} & \text{otherwise} \end{cases} \qquad \forall i \in \mathbf{N} \quad (35)$$

$$P_i^{red} \neq 0 \qquad \forall i \in \mathbf{N} \quad (36)$$

Equation (35) is built similar to (33), but uses the reduced rest time and does not enforce the full free day. If P_i already holds a valid weekly rest, it is simply transferred to P_i^{red}. (36) ensures that the reduced minimum weekly rest time is never violated.

Finally the average rest time needs to be checked for each span of sp weeks.

$$\sum_{j=0}^{sp-1} (Rest_{(i+j-1) \cdot w + P_{i+j}^{red} - o} \geq wr \cdot sp) \qquad \forall i \in \mathbf{N} \quad (37)$$

Equation (37) calculates the sum of the weekly rest times for the given span of weeks. Note that the index of P^{red} is modulo n and the index of $Rest$ modulo $n \cdot w$. Further the index of $Rest$ is not correct for $P_{i+j}^{red} = 0$, but in this case (36) is already violated anyway.

Optimizing Free Weekends

In the previous software implementation the user was presented a choice in several stages of the algorithm, potentially selecting from a large number of feasible solutions. However, defining properties of beneficial solutions beforehand and including these definitions in the model allows to transform the satisfaction problem into an optimization problem and to shift the selection process to the solver.

Shift work can be very disruptive to the social life of employees, e.g., social interactions with friends and family might be hard to schedule as free time is arranged in various different patterns compared to employees with regular free weekends.

Therefore, we chose to optimize the free time on weekends and provide different measurements of desirable weekend schedules. Note that for each optimization goal we also include bounds that allow the solver to immediately stop if the bound can be reached. While there is no room to discuss them here, they are used for the evaluation results.

Maximizing the Number of Free Weekends. Note that our week length w could be any positive number, however, in this context we use regular 7-day weeks where Saturday and Sunday are considered weekend. Usually it is beneficial to have a whole free weekend compared to a weekend with only one free day. We first define the number of free weekends f as follows.

$$f = \sum_{i=1}^{n} (T_{w \cdot i - o - 1} = O \wedge T_{w \cdot i - o} = O) \tag{38}$$

Equation (38) counts the number of weeks where both Saturday and Sunday do not have a shift assigned.

However, having Saturday and Sunday free might not be enough. Consider the case of having a night shift assigned to the Friday before a free weekend, where most of Saturday will probably be spent sleeping. Therefore, while not every option can be described here in detail, in practice it is important to have a flexible definition and allow settings like no Friday night shift before free weekends or even lexicographic combinations of different priorities.

Optimizing the Distribution of Weekends. While the number of free weekends is a good candidate for optimization, it is not the only possible measure regarding weekend schedules. Consider a schedule where two weekends are free in a row, followed by six weeks without a free weekend. Usually, a more regular distribution, e.g., a free weekend every four weeks, would be considered better.

In the following we use a boolean array *Free*, denoting for every week i whether the corresponding weekend is free. This array could be built using any of the definitions of free weekends above or a different one. We use free Saturday and Sunday as defined in (38). Next an array *Dist* of length n holding distances to the next free weekend is defined and used to minimize the maximum distance.

$$\mathit{Dist}_i = \begin{cases} \min\{j \in \mathbf{N} \mid \mathit{Free}_{i+j}\} & \text{if } \mathit{Free}_i \\ 0 & \text{otherwise} \end{cases} \qquad \forall i \in \mathbf{N} \tag{39}$$

$$d_m = \begin{cases} n+1 & \text{if } \max(\mathit{Dist}) = 0 \\ \max(\mathit{Dist}) & \text{otherwise} \end{cases} \tag{40}$$

Equation (39) gathers the distance to the next free weekend. The index of *Free* is modulo n, therefore in the worst case the distance is n. In (40), the maximum distance d_m is defined, taking into account the case of no free weekends in the whole schedule, which gets assigned the worst possible value for d_m.

A different possibility to deal with weekend distances, not only optimizing the maximum distance, is presented in the following, using a slightly modified distance definition $\widehat{\mathit{Dist}}$.

$$\widehat{\mathit{Dist}}_i = \begin{cases} \min\{j \in \mathbf{N} \mid \mathit{Free}_{i+j}\} - 1 & \text{if } \mathit{Free}_i \\ n & \text{otherwise} \end{cases} \qquad \forall i \in \mathbf{N} \tag{41}$$

$$d = \sum_{i=1}^{n} \widehat{\mathit{Dist}}_i^{\,2} \tag{42}$$

Equation (41) defines the distance to the next free weekend such that maximum values are assigned to weekends that are not free. Therefore, in (42) non-free weekends have the worst penalty, leading to the optimization of the number of free weekends, but due to the penalization of higher distances, at the same time the distribution of weekends is optimized.

Evaluation

This section describes the implementation of the models and the choice for the solver and evaluates the different models on a set of benchmark instances that are based on real-life problems. All experiments were carried out on an Intel Core i7-7500 CPU with 2.7 GHz and 16 GB RAM.

Implementation and Solver

All models described above were implemented using the solver-independent modelling language MiniZinc 2.2.2 (Nethercote et al. 2007). It allows to directly specify the constraint models and compiles them into a format called FlatZinc, which is understood by a wide range of solvers. This allows to evaluate the performance of different solvers or to switch to a new solver whenever new technologies advance without having to recreate the model.

A preliminary version of our model was tested with a range of different solvers. However, the lazy clause generation solver Chuffed (Chu et al. 2018) was clearly the best choice among the tested solvers for the presented model. Therefore, in the following evaluation Chuffed 0.10.3 is used for all experiments.

To increase the efficiency of the search, both MiniZinc and Chuffed offer settings to guide the search in the right direction. MiniZinc uses search annotations that allow to specify a custom search strategy. A wide range of different strategies was tested with different preliminary models. While details are omitted here, overall the best results where archived using the variable selection strategy `smallest` and the value selection strategy `indomain_min` on T, assigning the smallest possible values first.

Regarding Chuffed, a range of flags is available to influence the search. Again, a range of combinations was tried to find the best settings. Free search turned out to be the most important flag. This allows to switch on each restart between the user-specified search from the MiniZinc annotation and the activity-based search which is default for Chuffed. Further the restart scale was set to 1000 and variable assignment decisions are restricted to the main variables instead of additional ones introduced during compilation.

For the experiments we focus on the standard 20 benchmark instances[1] to look at results in more detail, while for the first comparison regarding the core model we use the extended 50 instances by (Musliu, Schutt, and Stuckey 2018) which also include infeasible instances.

Core Model

First we evaluate the core model `CORE` using equations (1) to (12) in comparison with the model `EXT1` including extensions for infeasible instances and the global cardinality constraint using (1) to (20) as well as the model `EXT2` additionally containing the block count constraints using (1) to (25). The timeout is set to 3600 seconds, average values in seconds are calculated across all instances including timeouts.

Table 3 shows the results of the comparison. Clearly both `EXT1` and `EXT2` can provide a major reduction in runtime (avg rt) compared to `CORE` with drops in the average runtime of 49% and 54%. This is especially visible for the unsatisfiable instances, where runtime drops by 67% for `EXT1` and `EXT2` can determine infeasibility for all 8 infeasible instances in an average of 1.3 seconds.

Detailed comparison shows that 4 infeasible instances are caught by (13). While no instances have $low > up$, several have those two values equal or very close. Especially for those instances `EXT2` is often beneficial or even necessary to prevent a timeout, while for some other instances the additional constraints make `EXT2` slower than `EXT1`, visible in the higher average regarding satisfiable instances. As `EXT1` and `EXT2` are strong on different instances, a combination `BEST`, executing both models until the first one has a result, can further provide a major improvement, allowing to solve all 50 instances with an average of 25 seconds. This is also a viable strategy in practice, as it can easily be done using two threads.

In comparison to (Musliu, Schutt, and Stuckey 2018), we can improve the number of instances solved by Chuffed from 48 to 50 and both `EXT1` and `EXT2` provide significantly better results for infeasible instances both regarding the number of finished instances and the runtime.

Weekly Rest Time

In this section we compare the runtimes for the models with weekly rest time constraints (26) to (37) added to the models `EXT1` and `EXT2`, resulting in `EXT1_WR` and `EXT2_WR`. We use the following settings for weekly rest times.

- $start = [6 \cdot 60,\ 14 \cdot 60,\ 22 \cdot 60]$
- $end = [14 \cdot 60,\ 22 \cdot 60,\ 30 \cdot 60]$
- $wr = 36 \cdot 60$; $wr_{red} = 24 \cdot 60$
- $e = 1$; $sp = 4$

The shift types represent typical early, late and night shifts, the weekly rest time is set to 36 hours with the possible exception of 24 hours once within 4 weeks. Instances with only 2 shift types do not have a night shift. All time spans are in minutes.

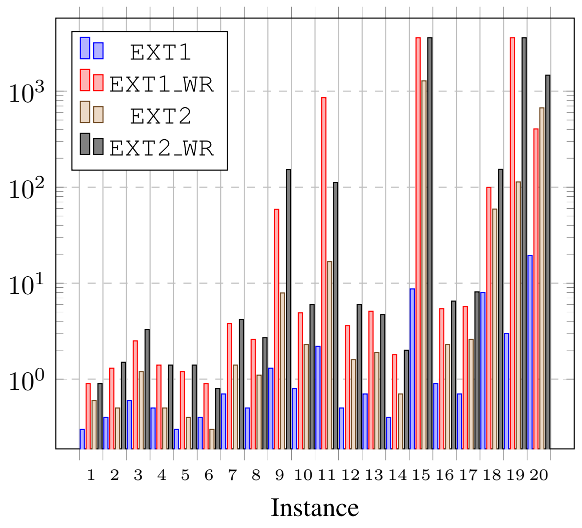

Figure 1: Runtime with weekly rest time constraints

Figure 1 shows the results of the comparison. The y-axis depicts the runtime in seconds. With weekly rest constraints, instance 2 was proven unsatisfiable and instances 15 and 19 ran into timeout while for the other 17 instances a valid result was found.

The results provide several insights. First, for almost all instances it is still possible to find a valid solution with the additional constraints, which is welcome. On the other hand, infeasibility of instance 2 shows that we cannot assume to always find such a solution, therefore, integrating these constraints is useful and important. Further, most schedules actually need the reduced weekly rest exceptions to get feasible, which supports the decision to also model these additional constraints. While 2 large instances that admit a solution without weekly rest can now not be solved within 1 hour, for the majority of instances a solution can still be found within a few seconds, justifying the usage in many practical scenarios.

Optimizing Free Weekends

For the optimization, from a practical point of view, it is not only important to reach a proven best result, but often also to get results fast. This does not have to be a contradiction, as many solvers, including Chuffed, allow the delivery of intermediate solutions using the flag `a`. This allows users to already consider intermediate plans, either accepting them at some point or deciding to spend more runtime on potential further improvements, while, given enough runtime, the best solution is still guaranteed in the end. Therefore, this evaluation shows not only the best reached solutions, but also the first reached solution and how the solver approaches the best solution, using `EXT1` as the base that is extended by the optimization goals.

Figure 2 shows the optimization of free weekends maximizing f. Each instance generates one trace where each

[1] `http://www.dbai.tuwien.ac.at/staff/musliu/benchmarks/`

Model	#total	avg nodes	avg mem	avg rt	#sat	avg rt	#unsat	avg rt
CORE	45	3.6m	64MB	445.9	40	271.1	5	1364.1
EXT1	47	1.8m	50MB	226.9	40	184.3	7	450.8
EXT2	49	251k	128MB	206.9	41	246.1	8	1.3
BEST	50	82k	52MB	25.5	42	30.2	8	0.8

Table 3: Evaluation results using infeasibility and block count constraints

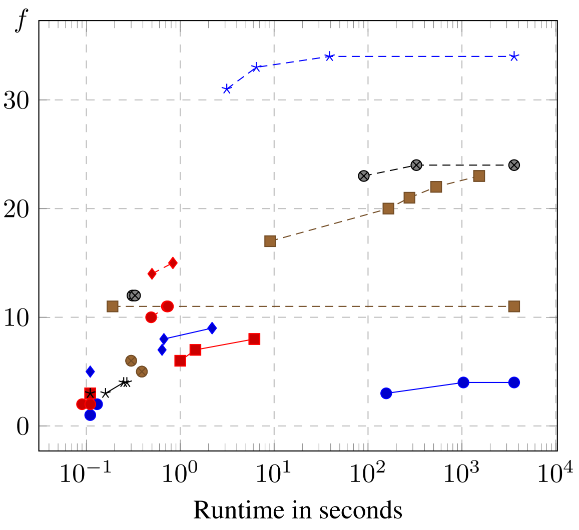

Figure 2: Maximizing the number of free weekends f

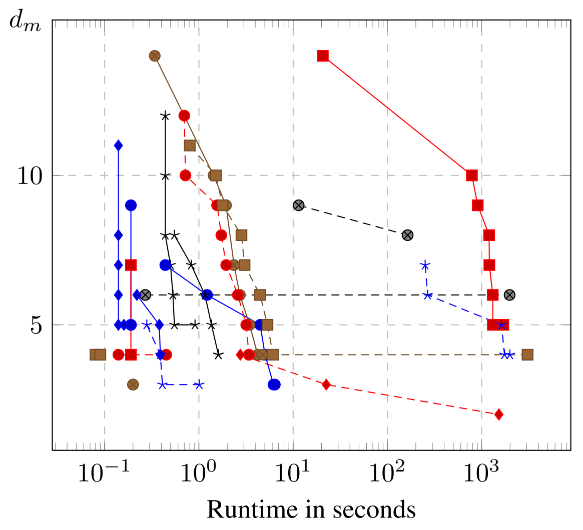

Figure 3: Minimizing the maximum distance d_m

mark represents a new best solution, allowing to investigate how solutions improve over time. The last mark indicates proof of optimality or timeout. For 18 out of 20 instances a solution is found within the timeout of 3600 seconds, for 14 instances the best solution is found and proven optimal. For a few instances long horizontal lines indicate that no better solution is found, but optimality is not proven. However, for 15 instances a first solution is found within 10 seconds, for 13 of those even optimality is proven within 10 seconds, indicated by the cluster of traces in the lower left corner. This shows viability and practical applicability of the approach for the majority of problems.

Figure 3 shows the same graph for the objective to minimize d_m. Steep declines for many instances show that there is more potential for improvement starting from the first results and that results can be improved very fast. Here, 16 instances get a first solution within 10 seconds, 14 are solved to proven optimality within 10 seconds.

For the distance measure d the results are similar, giving 15 first solutions within 10 seconds, for 9 instances the best solution is proven in 10 seconds. However, this time 7 instances run into the timeout before proving the optimum. As the definition of the objective is more complex, there is more room for small optimizations, resulting in many new best solutions during the search. With the majority of instances having early first solutions, the user can choose a good trade-off between solution quality and runtime.

Conclusion

We have presented a new exact model for rotating workforce scheduling that is implemented using the modelling language MiniZinc and solved with the lazy clause generation solver Chuffed. First we added new model extensions that showed to be very efficient in detecting infeasible instances in very little computational time and used bounds for the number of blocks to define redundant constraints. Then we added new real-life requirements in the form of complex weekly rest time constraints and are still able to solve the majority of the benchmark instances in short computational time. We introduced new objectives to optimize the scheduling of free weekends and showed that for the majority of instances the optimum can be found and proven in short computational time. Moreover, the output of intermediate solutions allows the user to decide about the trade-off between runtime and quality while running the solver.

In total this provides several improvements to our previous software as well as to the state-of-the-art modelling of the problem that are currently being integrated as a core component of the next iteration of our commercial software.

References

Baker, K. R. 1976. Workforce allocation in cyclical scheduling problems: A survey. *Journal of the Operational Research Society* 27(1):155–167.

Balakrishnan, N., and Wong, R. T. 1990. A network model for the rotating workforce scheduling problem. *Networks* 20(1):25–42.

Burke, E. K.; De Causmaecker, P.; Berghe, G. V.; and Van Landeghem, H. 2004. The State of the Art of Nurse Rostering. *Journal of Scheduling* 7(6):441–499.

Chu, G.; Stuckey, P. J.; Schutt, A.; Ehlers, T.; Gange, G.; and Francis, K. 2018. Chuffed, a lazy clause generation solver. `https://github.com/chuffed/chuffed`.

Chuin Lau, H. 1996. On the complexity of manpower shift scheduling. *Computers & operations research* 23(1):93–102.

De Bruecker, P.; Van den Bergh, J.; Belin, J.; and Demeulemeester, E. 2015. Workforce planning incorporating skills: State of the art. *European Journal of Operational Research* 243(1):1–16.

Erkinger, C., and Musliu, N. 2017. Personnel scheduling as satisfiability modulo theories. In *International Joint Conference on Artificial Intelligence – IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, 614–621.

Ernst, A.; Jiang, H.; Krishnamoorthy, M.; and Sier, D. 2004. Staff scheduling and rostering: A review of applications, methods and models. *European Journal of Operational Research* 153(1):3–27.

Falcón, R.; Barrena, E.; Canca, D.; and Laporte, G. 2016. Counting and enumerating feasible rotating schedules by means of Gröbner bases. *Mathematics and Computers in Simulation* 125:139–151.

Laporte, G., and Pesant, G. 2004. A general multi-shift scheduling system. *Journal of the Operational Research Society* 55(11):1208–1217.

Laporte, G.; Nobert, Y.; and Biron, J. 1980. Rotating schedules. *European Journal of Operational Research* 4(1):24–30.

Laporte, G. 1999. The art and science of designing rotating schedules. *Journal of the Operational Research Society* 50:1011–1017.

Musliu, N.; Gärtner, J.; and Slany, W. 2002. Efficient generation of rotating workforce schedules. *Discrete Applied Mathematics* 118(1-2):85–98.

Musliu, N.; Schutt, A.; and Stuckey, P. J. 2018. Solver independent rotating workforce scheduling. In *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, 429–445. Springer.

Musliu, N. 2005. Combination of local search strategies for rotating workforce scheduling problem. In *International Joint Conference on Artificial Intelligence – IJCAI 2005, Edinburgh, Scotland, UK, July 30 - August 5, 2005*, 1529–1530.

Musliu, N. 2006. Heuristic methods for automatic rotating workforce scheduling. *International Journal of Computational Intelligence Research* 2(4):309–326.

Nethercote, N.; Stuckey, P. J.; Becket, R.; Brand, S.; Duck, G. J.; and Tack, G. 2007. MiniZinc: Towards a standard CP modelling language. In Bessière, C., ed., *Principles and Practice of Constraint Programming CP 2007*, volume 4741 of *Lecture Notes in Computer Science*, 529–543. Springer Berlin Heidelberg.

Restrepo, M. I.; Gendron, B.; and Rousseau, L.-M. 2016. Branch-and-price for personalized multiactivity tour scheduling. *INFORMS Journal on Computing* 28(2):334–350.

Triska, M., and Musliu, N. 2011. A constraint programming application for rotating workforce scheduling. In *Developing Concepts in Applied Intelligence*, volume 363 of *Studies in Computational Intelligence*. Springer Berlin / Heidelberg. 83–88.

Van den Bergh, J.; Belin, J.; De Bruecker, P.; Demeulemeester, E.; and De Boeck, L. 2013. Personnel scheduling: A literature review. *European Journal of Operational Research* 226(3):367–385.