Rotating Workforce Scheduling in Airport Security

Johanna Wiesflecker[a,*], Maurizio Tomasella[a], Daniel Guimarans[b,1], Thomas W. Archibald[a]

[a]*Business School, The University of Edinburgh, 29 Buccleuch Place, EH8 9JS, Edinburgh, United Kingdom*
[b]*Amazon, 22 Rue Edward Steichen, L-2540 Luxembourg, Luxembourg*

Abstract

Recently we paired with an airport operator to create a semi-automated rostering tool for its security staff. The tool creates cyclic rosters involving staff working under several contracts, split into teams the size of which are contract dependent. In this paper we extend the general purpose state-of-the-art solution approach to suit this setting, a very typical one in airports. Our rosters encode the days-off patterns as well as the shift types (morning, afternoon, night) on work days for the different teams of workers. The innovations include the introduction of contracts, team sizes and more realistic constraints on days-off patterns in the schedule. We present several solver independent models with extensions and compare them using both constraint programming and mixed integer programming solvers. Our results verify the effectiveness of grouping a workforce into multiple teams (under various contracts) and we demonstrate the advantages of using a solver independent solution approach. Through our experiments, we aim to show airport operators how they can feasibly tackle security rostering, often their single most expensive cost centre.

Keywords: Rotating Workforce Scheduling, Cyclic Rostering, Constraint Programming, Solver Independent Approaches, Airport Security

1. Introduction

In staff scheduling, rostering problems (Van Den Bergh et al., 2013) aim to allocate employees to shifts, while considering both hard and soft constraints, with the former to be met at all times, and violations of some combination of the latter to be minimised (Ernst et al., 2004; Komarudin et al., 2013). The great variety of practical applications results in numerous problem-specific arrangements of constraints and objectives.

Rotating workforce schedules (RWS), also known as cyclic rosters (Rocha et al., 2013a), are adopted in staff scheduling to cover demand whilst aiming for fair allocation of work throughout the planning horizon. The workforce is divided into groups that follow canonically the exact same pattern of days on/off work and shifts. This problem frequently surfaces in the service industry. Most suitable to cyclic rostering are situations with repeating demand patterns, such as train services or flight schedules running according to regular weekly timetables. In this paper we will take a closer look at RWS in airport security, both in general and in the case of a UK-based airport operator we have recently worked together with.

Figure 1 provides a bird's eye, end-to-end view of the considered scheduling process. Airports are often heavily unionised environments where schedules have to be approved months in advance. This is partly due to the time taken to create rosters, and partly due to the approval process itself, which only concludes with an agreement signed between the unions and the airport operator. Ideally, the schedule for an entire operating season (i.e. the airport's summer schedule) is frozen a few (usually 5-7) months in advance. It then stays untouched until a week before each day of operation, when demand forecasts are more reliable and workforce

*Corresponding author
Email addresses: Johanna.Wiesflecker@ed.ac.uk (Johanna Wiesflecker), Maurizio.Tomasella@ed.ac.uk (Maurizio Tomasella), daniel.guimarans@gmail.com (Daniel Guimarans), T.Archibald@ed.ac.uk (Thomas W. Archibald)
[1]The contributions of Daniel Guimarans to this paper are not related to his role at Amazon.

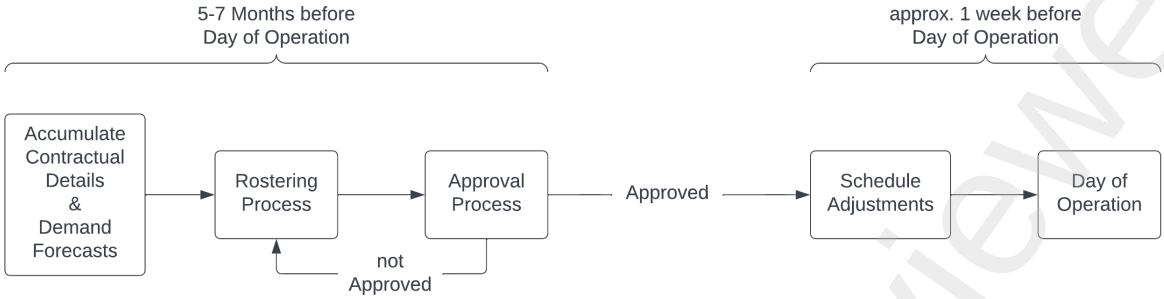

Figure 1: Image of the rostering process for a scheduler.

composition up to date, which allows for fine-tuning. Automating roster creation (currently not the case at the partner airport, nor in many other cases) may enable both parties to devote a higher portion of the overall process to more interactive, meaningful negotiations, for the benefit of everyone, workers included.

A solver-independent approach to modelling and solving general RWS instances is provided in (Musliu et al. (2018)), the first complete method capable of solving all the standard benchmarks. Many new benchmark instances are proposed in the same article, which discusses two solver-independent models, and shows that the best of the two outperforms the then state-of-the-art. The former model states each constraint directly; the latter embeds as much as possible into a single `regular` constraint (Pesant, 2004), by means of a deterministic finite-state automaton.

Table 1: Example Schedule and Demand

Schedule									
contract	team size	line	Mon	Tue	Wed	Thu	Fri	Sat	Sun
c_1	2	1	D	D	D	A	A	/	/
		2	/	/	/	N	N	N	N
		3	N	/	/	D	D	A	A
		4	A	N	N	N	/	/	/
c_2	3	1	A	A	N	/	/	/	D
		2	D	D	A	A	N	N	/
		3	/	/	D	D	A	A	N
		4	/	/	/	/	D	D	A

Demand							
shift type	Mon	Tue	Wed	Thu	Fri	Sat	Sun
Day (D)	5	5	5	5	5	3	3
Afternoon (A)	5	3	3	5	5	5	5
Night (N)	2	2	5	4	5	5	5

While the approach in (Musliu et al. (2018)) appears largely applicable in many contexts, RWS in airport security presents a number of peculiarities, the most salient of which we address in this paper, by substantially adapting the approach by Musliu and colleagues to our novel setting. Our version of RWS features a workforce subdivided into a number of different contracts (e.g., c_1 and c_2 in Table 1). Each contract comes with its own set of specific constraints, the nature and form of which remain largely the same as in (Musliu et al. (2018)). Demand for different shift types— say day, afternoon or night (D, A and N in Table 1, respectively)— requires given staffing levels over time, which in turn depend on the operations that are forecast to take place throughout the week (bottom of Table 1). To meet the demand exactly (say for the Day shift on Mondays), it is possible to pick and mix from different contracts. This requires a

fully flexible, equally skilled (homogeneous) workforce, which is normally the case in airport security. Our contributions in this paper are:

1. An extension of RWS to include multiple teams and contracts.
2. Detailed customisation of the state-of-the-art approach to cater for the extended problem.
3. A conceptual reference model to guide implementation.
4. A case study of the extended problem in a real-life airport security setting.

After discussing the relevant literature guiding our work (Section 2), we provide a formal problem formulation (Section 3), discuss the many modelling and solving techniques involved (Section 4), and evaluate their effectiveness through a first set of computational experiments (Section 5). We then present our airport case study (Section 6), which we study in details through further experimentation (Section 7), prior to concluding (Section 8).

2. Related Works

The study of cyclic rosters dates back to Bennett & Potts (1968), studied weekly schedules for bus crews, first by creating days-off schedules and then allocating specific shifts to each non-off day. The first survey appeared a decade later (Baker, 1976), covering integer linear programming (ILP) formulations and discussing the role of service levels in determining the optimal sizing of workforce. Other early contributions involved ILP and network flow formulations (Bartholdi et al., 1980) and various heuristics (Bechtold, 1981).

Laporte (1999) gave a detailed account of the benefits from creating cyclic rosters by hand— one of very few papers considering separate rosters for different contract types (full-time *vs* part-time). Clearly, scalability of such an approach is an issue. Van Den Bergh et al. (2013) provide a popular, comprehensive and relatively recent review, featuring several works in RWS (for example Laporte & Pesant, 2004; Mörz & Musliu, 2004; Purnomo & Bard, 2007).

Musliu et al. (2018) proposed the current state-of-the-art approach to cyclic rostering. This work builds upon Gärtner et al. (2001) — on the creation of 'shift (building) blocks', the 4-step framework from Musliu et al. (2002) that combines the same building blocks to create cyclic rosters, and the more recent Satisfiability Modulo Theory (SMT) approach from Erkinger & Musliu (2017). Musliu et al. (2018) outline the usefulness of solver-independent modelling for experimental design, by assessing both Constraint Programming (CP) and Mixed Integer Programming (MIP) solvers for RWS problems.

Kletzander et al. (2019) follow in the same footsteps, discussing methods for faster detection of infeasible instances (such as extra checks for infeasible demand patterns), additional constraints for weekly free time (aside from forbidden shift sequences), and a better distribution of free weekends. Becker (2020) instead propose a decomposition approach, with computational experiments showing improvements on previous heuristic approaches to RWS, whilst failing to outperform Musliu et al. (2018).

Many dichotomies pervade the rostering literature, including (a) homogeneous *vs* heterogeneous workforce (in terms of contract types, skills, etc.) and (b) individual workers *vs* teams/groups/crews thereof. Van Den Bergh et al. (2013) testify that the two do not seem to be equally popular, with about 80 papers covering multiple contract types *vs* only 8 papers focusing on crews as the rostering entity, with no other form of worker grouping detected. Teams and groups feature though in more recent literature, particularly healthcare (Olya et al., 2022) and manufacturing applications (Kiermaier et al., 2016; Rocha et al., 2014).

Kiermaier et al. (2016) use identically sized groups of homogeneous workers to reduce the size of their cyclic rostering problem. They mention that the group size and thus the number of lines in a roster could be set as decision variables in the problem but that would make the problem intractable. They opted instead for setting the groups size and line number as parameters in their stochastic MIP formulation and running several different scenarios.

Rocha et al. (2014) consider a cyclic rostering problem in the glass industry which entails the scheduling of five teams of homogeneous workers who work under the same contract. Teams can be seen as the rostering unit in this problem making the example equivalent to a cyclic rostering problem for five individuals. The authors propose a construction heuristic to solve the given rostering problem.

Workforce scheduling under multiple contracts is a prevalent problem in nurse rostering, where contracts are often divided into full-time and part-time (Martin et al., 2013; Maenhout & Vanhoucke, 2009). Pairing cyclic rosters with multiple contracts however can cause problems due to different working requirements as imposed by different contracts. For this reason, workers under different contracts normally do not follow the exact same pattern. Schedules have to be differentiated instead, as the pragmatic approach by hand from Laporte (1999) demonstrated a long time ago.

Maenhout & Vanhoucke (2009) outline a process for creating cyclic schedules for nurses with different preferences and work requirements. Their process starts with generating two cyclic schedules: a basic schedule with early, late and night shifts and a supplementary schedule with only early and late shifts. Nurses first specify whether or not they consider night shifts as an option for them. In a second step, each nurse receive their own individual schedule, which is based around the two cyclical templates. Differentiating elements include the starting point in the working pattern, additional days off for part-time workers, and preferred days off to acknowledge seniority.

More recently, Rocha et al. (2013b) study a cyclic rostering problem for a workforce consisting of both full-time and part-time workers. The aim of the model is, however, to schedule as many full-time workers as possible to begin with. Part-time workers are only used to cover the demand that cannot be covered by scheduled full-time workers. This way part-time staff do not truly follow cyclic rosters.

As the latter articles show, it is possible to combine cyclic rosters with multiple contracts. However, unless a different schedule for each contract is created, individual adjustments to the schedule of each worker might be required, which largely defeats the overall idea of a cyclic working pattern. The following sections will outline our approach to tackling this problem by creating separate cyclic rosters for each contract.

3. Problem Definition

We now provide a formulation for the general cyclic rostering problem with multiple contracts and team sizes, for a fully homogeneous workforce where everyone can perform all tasks.

Solving the problem produces a cyclic roster featuring a number of lines (Table 1) that is exactly equal to the number of teams in the workforce. Each contract has its own number of lines (teams), as indicated by higher level workforce planning decisions. Each line dictates the sequence of shift types / days off to be followed by the team which starts its cycle from that very line (shift types are different non-overlapping portions of the day — as in Table 1).

The resulting weekly patterns that correspond to any one line ensure the demand for workforce is met, for every day and shift type in the week, whilst adhering to the contractual requirements of the workforce. Staff working under different contracts follow slightly different versions of the same underlying working regulation. Therefore, not all teams can follow the exact same schedule throughout the planning horizon. To rectify this issue we introduce a separate cyclic roster for each contract. So, each team rotates through all the lines of teams with the same contract. Once a team reaches the last line of their contract they move to the first line of the same contract for the following week.

All teams of the same contract need to have the same team size but team sizes can differ by contract. There are two reasons behind this. First, the regulation specific cyclical dynamics just outlined. Second, different areas of the process where staff are employed may be more suitable to accommodate teams of different sizes at any one time (shift type).

Varying team sizes between contracts come with an additional advantage. Namely, more flexibility for satisfying demand patterns with a homogeneous workforce, whilst keeping the problem size small to allow for shorter runtimes.

Theoretically, the same, small team size, equal for all contracts, could be adopted, thus maximising said flexibility. Clearly, this will increase the problem complexity, on one hand. More pressing issues that disable this approach exist though. First, the nature of the process may require teams of staff to be rostered together, hence a natural lower bound to the minimum size that can be considered normally exist. Second, the already mentioned process specific design may require explicitly or at least expect ideally different sizes for the team that work in different areas of the process at the same time. Our case study from airport security processes will show more explicitly an example of such practicalities, thus justifying our approach.

To formulate our problem we need the following parameters and sets:

- nc: number of contracts
- $\mathcal{C} = \{1, \ldots, nc\}$: set of all contracts
- nt: number of teams (lines)
- $\mathcal{T} = \{1, \ldots, nt\}$: set of all teams — teams of the same contract have consecutive indices
- g: line length — most often 7 days (one working week)
- ft_c and lt_c: (indices of the) first and last team on contract $c \in \mathcal{C}$
- ts_c: team size for contract $c \in \mathcal{C}$
- $\mathcal{A}_c$: set of permissible shift types for contract $c \in \mathcal{C}$ — often: morning (M), afternoon (A) and night (N). The additional option of a 'day off'(annotated by O) also exists. $\mathcal{A}_c^+$ denotes the set of all permissible shift types for contract c including days off. We also consider $\bigcup_{c=1}^{cn} \mathcal{A}_c = \mathcal{A}$ and $\bigcup_{c=1}^{cn} \mathcal{A}_c^+ = \mathcal{A}^+$.
- $b\_up_{sh,c}$ and $b\_lo_{sh,c}$: maximum and minimum 'block' lengths for shift type $sh \in \mathcal{A}_c^+$ under contract $c \in \mathcal{C}$ — a block being a set of consecutive days in the schedule
- w_up_c and w_lo_c: maximum and minimum lengths of 'work blocks' for contract $c \in \mathcal{C}$, where a work block refers to a set of consecutive working days, i.e. any run of consecutive shift types without a day off.
- R: requirement matrix — each element $R_{sh,j}$ encodes the workforce demand for shift type $sh \in \mathcal{A}$ on day j where $1 \leq j \leq g$. $R0$ also includes the workforce requirement for days off, which is calculated by subtracting the demands for all shift types of the day from the total size of the workforce.
- q, r: integers used for defining the weekends off requirement — often, one must ensure that at least q out of any r weekends are set as off work, where $q \leq r$.
- $\mathcal{F}_c$: set of pairs (sh_1, sh_2) of forbidden shift type combinations for contract $c \in \mathcal{C}$, meaning that sh_1 in any one day of the planning horizon cannot be followed by sh_2 the next day (or, equivalently, sh_2 cannot be preceded by sh_1 in the previous day) — a common example of this is "forward rotation", where the shift type for the next day has to be either the same or a later shift type than the one allocated to the day immediately before, or a day off.

The decision variables for the problem are:

- S: $nt \times g$ matrix, where each element $S_{i,j} \in \mathbf{A}^+$, $1 \leq i \leq nt$, $1 \leq j \leq g$ encodes the shift type assigned to line i on day j.
- w_off_i: binary indicator of whether or not team $i \in \mathcal{T}$ has been allocated a weekend off.

To model the rotation through the lines of teams of the same contract we introduce two functions:

$$u(i) = \begin{cases} i, & ft_c \leq i \leq lt_c \\ i - (lt_c + 1) + ft_c, & lt_c < i \leq 2lt_c - ft_c \end{cases}$$

$$v(i,j) = \begin{cases} (i,j), & 1 \leq j \leq g \\ (u(i+1), j \mod g), & g < j \leq 2g \end{cases}$$

Armed with the above concepts/notation, the following constraint satisfaction problem can be formulated:

$$\sum_{k=0}^{w\_up_c} (S_{v(i,j+k)} = O) > 0 \qquad c \in \mathcal{C}, i \in ft_c \dots lt_c, j \in 1 \dots g \tag{1}$$

$$\sum_{k=1}^{w\_lo_c} (S_{v(i,j+k)} = O) = 0 \qquad c \in \mathcal{C}, i \in ft_c \dots lt_c, j \in 1 \dots g,$$
$$S_{1,j} = O \wedge S_{v(i,j+1)} \neq O \tag{2}$$

$$\sum_{k=0}^{b\_up_{sh,c}} (S_{v(i,j+k)} \neq sh) > 0 \qquad c \in \mathcal{C}, i \in ft_c \dots lt_c, j \in 1 \dots g, sh \in \mathcal{A}_c^+ \tag{3}$$

$$\sum_{k=1}^{b\_lo_{sh,c}} (S_{v(i,j+k)} \neq sh) = 0 \qquad c \in \mathcal{C}, i \in ft_c \dots lt_c, j \in 1 \dots g, sh \in \mathcal{A}_c^+,$$
$$S_{1,j} \neq sh \wedge S_{v(i,j+1)} = sh \tag{4}$$

$$\sum_{sh \in \mathcal{A}_c} \sum_{i=ft_c}^{lt_c} \sum_{j=1}^{g} (S_{i,j} = sh) = (lt_c - ft_c + 1)g \qquad c \in \mathcal{C} \tag{5}$$

$$S_{i,g} = O \wedge S_{i,g-1} = O \leftrightarrow w\_off_i = 1 \qquad i \in \mathcal{T} \tag{6}$$

$$\sum_{k=1}^{r} (w\_off_{u(i+k)} = O) \geq q \qquad c \in \mathcal{C}, i \in ft_c \dots lt_c \tag{7}$$

$$S_{i,j} = sh_1 \rightarrow S_{v(i,j+1)} \neq sh_2 \qquad j \in \mathcal{T}, (sh_1, sh_2) \in \mathcal{F}_c, c \in \mathcal{C} \tag{8}$$

Constraints (1) and (2) ensure that each block of consecutive work days over a contract's cycle is within the contracted limits for consecutive days of work. Similarly, constraints (3) and (4) ensure that the number of consecutive days of the same shift type (including days off) are within the contracted bounds for each contract and shift type. Constraints (5) restrict the assignment of shift types to be from the pool of allowed shift types for each contract. Constraints (6) and (7) model the required pattern of having q weekends off work out of every r consecutive weeks, and constraint (8) rules out forbidden shift combinations for each contract.

To ensure the demand of workers is satisfied for every day and every shift type, the following constraint is also needed:

$$\sum_{c=1}^{nc} ts_c \sum_{i=ft_c}^{lt_c} (S_{i,j} = sh) = R_{sh,j} \qquad j \in 1 \dots g, sh \in \mathcal{A} \tag{9}$$

This ensures that the number of assigned workers satisfies the demand of workers (not just teams). Given that teams of different team sizes might be combined to satisfy demand at certain locations it is important that schedulers keep the workforce demands in mind when setting team sizes for contracts before the scheduling process. Otherwise, the problem becomes infeasible.

4. Modelling and Solving Approach

CP technology has long provided a viable approach to tame managerial problems at various levels, whether strategic, tactical (such as RWS) or operational. Wallace (2020) makes a good case for adopting CP to develop 'intelligent' decision support systems. The book presents many successful examples, modelling and solving various combinatorial problems through a solver independent approach grounded on a high-level modelling language, interfaces to multiple solvers, and control mechanisms over the search for solutions. The already cited state-of-the-art approach to cyclic rostering in Musliu et al. (2018) belongs to the same school

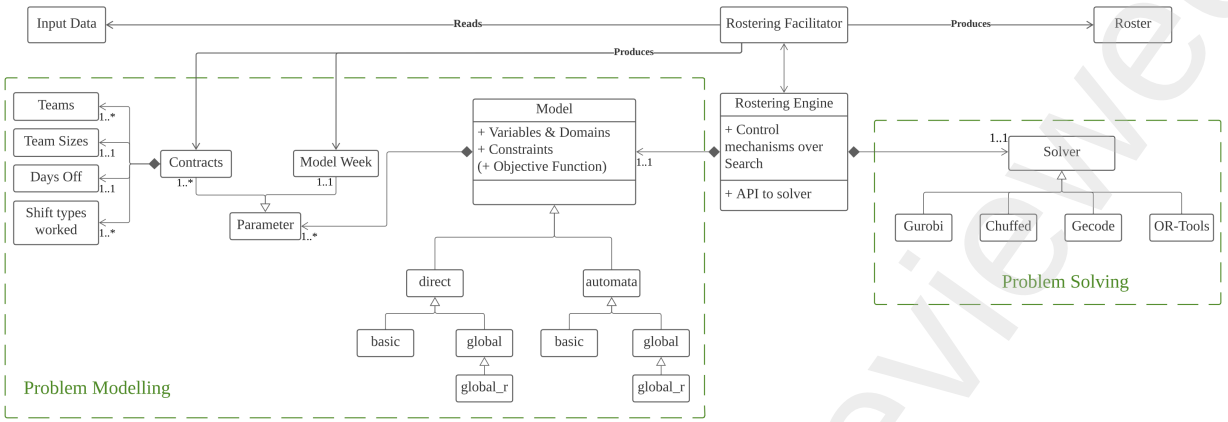

Figure 2: Class diagram of the problem implementation.

of thought. Therefore, it seemed sensible to attempt an adaptation of the latter that can tackle the extended general RWS defined in Section 3.

The present section describes all the main features that this adaptation entails, including considerations at both the conceptual and implementation levels. The intention is to provide guidance to whoever may decide to venture onto a similar pathway, especially when addressing variants of scheduling problems. The next section will then show some empirical evidence from our own implementation, and an early discussion of the pros and cons of such a solver independent, CP based approach to solve our extended RWS problem. If anything, this will give us the opportunity to then move on to discussing our experience of applying this same approach to our real world case of RWS in airport security.

4.1. Modelling and Solving Architecture

Adopting UML class diagram style notation, Figure 2 provides a conceptual overview of our modelling and solving approach to the problem just formulated, showing how the components of any implementation of this approach might be suitably organised.

The core element of the architecture in Figure 2 is what we call the Rostering Engine. The way this class is organised reflects the solver independent nature of the approach, showing the clear separation between Model and Solver, its two main components. The Rostering Engine provides the means for the rostering professional using our system to pair exactly one model with exactly one solver, each chosen from separate collections, virtually at any run of the system. This is achieved via appropriate Application Programming Interfaces (APIs) to the available solvers. Figure 2 names the solvers we used in our own implementation — more details on each to appear in the next sections, but the choice is wider and different collections should be considered for different problems. The Rostering Engine also provides for more specific options to fine-tune the search parameters and thus apply additional controls to the search strategy. For instance, in a CP-based approach to modelling and solving, such as ours, search 'annotations' can be used, within the Rostering Engine, to control the variable order and the value order to be followed by the chosen solver, effectively changing the search tree for the problem, with no other modifications required. In our implementation, the Rostering Engine is implemented in MiniZinc, one of the many features of which is that it allows separation of data (Input Data class) from the model of the problem (Model class).

Input data consist of problem and instance specific details such as: what contracts are available, size of the workforce employed under each contract (number of teams, team size, shift types allowed, etc.), any crucial elements of the layout and logic of the systems/processes in which teams work (e.g. an airport's security hall, or a factory's warehouse), and demand forecasts. It is then the Rostering Facilitator element that reads all the available data and prepares it in a format that is suitable for consumption by the solver. This new version of the input data becomes part of the Model element, which contains, just like any model to

be used in problem solving (Ackoff, 1978): decision variables, parameters, constraints and (perhaps, though not in our problem investigated in this paper) an objective function to be 'optimized' (in whatever sense of the word). Using a language closer to our CP-based view of problem modelling and solving (Wallace, 2020) our Model class is organised around: variables, constraints and parameters. Variables represent the decisions that need to be made in the model and are usually accompanied by a set of possible values - their domains. Constraints impose rules on the decision variables, ruling out incompatible choices, and parameters define each problem instance to be investigated.

The Rostering Facilitator class acts as a 'wrapper' layer (implemented in Python, in our case) to connect the input data with both the engine/model and the roster that is produced (Roster element) as output of the search process. On transforming the raw input data into its usable format as specified within the Model class, the Rostering Facilitator effectively creates the problem instance. Its other main tasks are to set the Solver and Model choices in the Rostering Engine and to create readable outputs of the problem solution (an object of the Roster class).

One last component of the Model class is worth discussing, and that is the Model Week element. In the case study later discussed in details (as well as in many other rostering applications) a 'template week' is artificially created to represent the whole planning horizon. This is normally an effective way to smooth out demand variability across the horizon, thus reducing the cyclic rostering problem to one that assumes the same weekly demand to present itself repeatedly over the entire horizon. While this facilitates problem solving, it relies on appropriate choices to be made by the rostering professional in defining what a model week looks like. For instance, the demand on Monday morning between 6am and 7am may be defined as the maximum value of demand taken, within the same time interval, across all Mondays of the planning horizon. Other alternatives clearly exist, and the final choice is partially an arbitrary one.

Finally, Figure 2 shows that different versions (subclasses) of models may exist, just as in our implementation, effectively leading to a host of model options that can be tested. As we will later discuss, the model variant may be a significant factor to be considered in solving a problem such as our extended version of RWS. All of the options we tested as part of the work on this paper are outlined in more detail in the following Sections 4.2 - 4.5.

4.2. Direct Model

One of the simplest modelling choices is to create a direct implementation of the mathematical formulation, where all constraints are outlined in the modelling language —MiniZinc included —(almost) exactly as they are expressed in the mathematical model. However, while these models can be understood faster by someone who is not familiar with the intricacies of the programming language used, they may not lead to the fastest results, either in a specific instance or more generally for a type of problem. Ruling out such a possibility would be perhaps just as unwise as assuming it represents the only possibility to solve a problem.

4.3. Automata based Models

An automaton is a device that is capable of representing a 'language', according to well-defined rules (Cassandras & Lafortune, 2008). As a modelling formalism, it is generally convenient whenever the concepts of a system's 'state', the 'transitions' between any states, and the 'events' driving such transitions may be applicable. In rostering problems, states may refer to all possible shift combinations that can occur given the workforce rules. Hence, the rules surrounding shift types and days off may be encoded in an automaton.

Figure 3 gives an example automaton for a rostering problem with two shift types (day = D, afternoon = A), days off (O), and forward rotation enforced. We also use w to represent a day where any type of work shift can be assigned. For the purpose of this example, different contracts are not considered. The minimum and maximum lengths of work and shift blocks are as follows: $b\_lo_D = 2$, $b\_up_D = 3$, $b\_lo_A = 2$, $b\_up_A = 3$, $b\_lo_O = 2$, $b\_up_O = 2$, $w\_lo = 3$, $w\_up = 4$. Solid lines represent the decision to have a day shift next, dashed lines represent the decision to have a day off next and dotted lines represent the decision to have an afternoon shift next. Dummy state '*start*' has the purpose of showing that any of O, D or A can be used as a starting point in the diagram. It is not used when the automaton is encoded in MiniZinc (Nethercote et al. (2007)), as we will demonstrate later in this section. Note that in this example the states wwA and

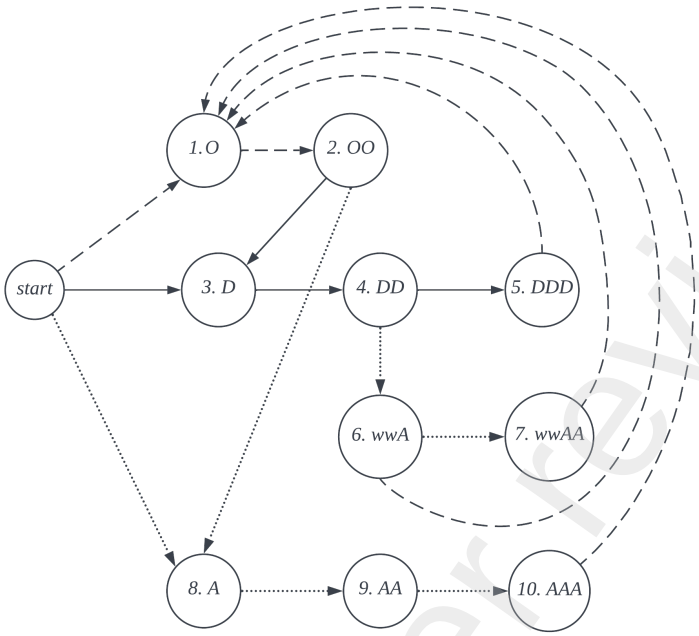

Figure 3: Example of an automaton encoding work requirements.

$wwAA$ could also be called DDA and $DDAA$. However, in other instances there could be an additional shift type that can also be followed by an afternoon shift. In that case wwA and $wwAA$ can be used to encode the transfer from all shift types to afternoon shifts. When dealing with multiple contracts, where shift type rules tend to be contract specific, a separate automaton can be created for each contract. Having a single 'catch-all' automaton for all contracts tends to be problematic due to the complexity of having to set additional rules about which contract is allowed to go through which paths in the automaton. This can be done, and indeed we did try it in our case study, but tends to lead to unreasonably large automata. Most importantly, having separate automata offers a more straightforward modelling approach. It also scales better over time, within the same application, as the pool of contract options being considered evolve.

To implement an automaton in MiniZinc the `regular`$(x, Q, S, d, q0, F)$ 'global constraint' is used (Stuckey et al., 2022). Global constraints are "[...] high-level modelling abstractions, for which many solvers implement special, efficient inference algorithms." (Stuckey et al., 2022, 4.2.1 Global constraints). Using these types of constraints can speed up the solver time compared to directly implemented constraints. The `regular` constraint takes the following six inputs to describe the automaton:

- x is an array representing the schedule of the workforce, where the first entry is the first day of the first line followed by the second day of the first line and so on. The last g entries of the array are the days of the first line repeated. This is necessary to ensure that all the work requirements are also satisfied when teams move from the last line of their schedule back to the first line.
- Q represents the number of states in the automaton. For the example in Figure 3 this would be 10, since the *start* state is just a dummy entry to make our pictorial representation of the automaton easier to understand.
- S represents the values that are being assigned to the schedule. At the moment the `regular` constraint in MiniZinc only allows for integer values to be assigned. So, S is equivalent to the number of shift types that need to be assigned, where each of the integers $1 \dots S$ represents one shift type.

- d is the transition function that maps the transition from each state $1 \dots Q$ to the next state $0 \dots Q$ for each shift type choice $1 \dots S$. The option 0 represents illegal shift choices for each state. For instance, the first few entries of d for the example in Figure 3 would be as shown in Table 4.3.

	O	D	A
1.	2.	0	0
2.	0	3.	8.
⋮	⋮	⋮	⋮

Table 2: First entries of matrix d for the automaton in Figure 3.

  In this table, while starting from the first state, option O will lead to state 2, whilst options D and A are not allowed. From state 2 instead, option O itself is not allowed, whilst options D and A will lead to state 3 and 8, respectively. This function is then encoded as a $Q \times S$ matrix for the `regular` constraint.

- $q0$ represents the initial state of the automaton. Since, we are working with a cyclic roster, this does not matter much as the schedule rotates through the planning horizon. We simply chose state 1.

- F represents the set of accepting states, which represents the states that the automaton is allowed to finish or stop in. In our case this represents all states as we do not care where the final state of the automaton is. Since the first line is repeated in x we have already ensured that all work regulations are satisfied throughout the schedule. So, F is simply all states $1 \dots Q$.

4.4. Global Constraints

In CP, a vast collection of global constraints exist. Given their effectiveness, when starting from a direct model a viable option to be put for testing is to replace one or more constraints in the model with one or more global constraints, and take note of the effects such a change may induce in model runtime and quality of the obtained solutions. In our work we considered two global constraints: `global_cardinality`() and `sliding_sum`(), the explanation of which comes next in this section.

4.4.1. Global Cardinality

The `global_cardinality`(x, a, c) constraint ensures that in the input array x the number of occurrences of a are exactly equal to c (Stuckey et al., 2022, 4.2.1.1. Counting constraints). This constraint has an extension called `global_cardinality_low_up`(x, a, l, u) which ensures that in the input array x the number of occurrences of a are between l and u (Stuckey et al., 2022, 4.2.1.1. Counting constraints). Both constraints can also be used on sets of occurrences a. In that case the limits c or l and u need to have entries for each element of a.

Global cardinality constraints are ideal for modelling demand satisfaction constraints. For example constraint (9) can be rewritten as

$$\texttt{global\_cardinality}([S_{i,j} + r - r | i \in \mathcal{T}, r \in 1 \dots ts_{cr_i}], \mathcal{A}^+, [R_{sh,j} | sh \in \mathcal{A}^+]) \qquad j \in 1 \dots g \tag{10}$$

The expression "$+r - r$" is used to repeat each term of S for every team member that is rostered to work the respective shift. This is necessary since the demand is given in single workers rather than teams. Note, cr_i in this constraint refers to the contract of team i.

This constraint can also be used to model the maximum consecutive days worked constraint (1). The implementation is as follows

$$\texttt{global\_cardinality\_low\_up}(([S_{v(i,j+k)} | k \in 0 \dots w\_up_c], O, 1, w\_up_c + 1) \qquad j \in 1 \dots g, c \in \mathcal{C}, \; i \in ft_c \dots lt_c \tag{11}$$

This means that for every $w\_up_c + 1$ consecutive days in the Schedule for contract c there need to be between 1 and $w\_up_c + 1$ days off.

4.4.2. Sliding Sum

The `sliding_sum`(l, u, s, a) constraint ensures that the sum of every s consecutive terms in the array a lies between l and u (Stuckey et al., 2022, 4.2.1.12. Other declarations). This global constraint is ideal for modelling rules for days off such as weekends off or maximum consecutive days worked.

To model the constraint that at least q out of r weekends need to be assigned off (constraint (7)) we can write

$$\texttt{sliding\_sum}(q, r, r, [w\_off_i | i \in ft_c \dots lt_c] + +[w\_off_i | i \in ft_c \dots ft_c + r - 1]) \qquad c \in \mathcal{C} \tag{12}$$

Note that "[] + +[]" is the convention for concatenating arrays in MiniZinc. This is necessary here, to ensure that the weekends off constraint is satisfied for every r consecutive lines in the schedule.

To model the maximum consecutive days worked constraint we first need to define a new binary matrix O which has entry $O_{i,j} = 0$ for every day off in the schedule (S), and $O_{i,j} = 1$ for every other day in S. We can then rewrite the maximum consecutive days worked constraint (1) as follows:

$$\texttt{sliding\_sum}(0, w\_up_c, w\_up_c + 1, [O_{i,j} | i \in ft_c \dots lt_c, j \in 1 \dots g] + +[O_{ft_c,j} | j \in 1 \dots w\_up_c]) \quad c \in \mathcal{C} \tag{13}$$

Similar to the weekends off constraint, we repeat some terms at the end to ensure that the constraint is satisfied throughout the entire cycle.

4.5. Additional Modelling Choices

More options do exist, in modelling approaches like ours, to try and improve runtimes and/or quality of solutions. In this last subsection, we outline the options we adopted in the work that led to this paper.

4.5.1. Redundant Constraints

Redundant constraints are employed to tell the computer program something that is obvious from the given constraints. One such example is adding a constraint with demand for days off. This is implicitly defined by the demand for all working shifts, since the remaining workforce is automatically assigned a day off. However, writing this constraint explicitly can improve runtimes in the model. This would be done by replacing constraint (9) with

$$\sum_{c=1}^{nc} ts_c \sum_{i=ft_c}^{lt_c} (S_{i,j} = sh) = R0_{sh,j} \qquad j \in 1 \dots g, sh \in \mathcal{A}^+ \tag{14}$$

The demand for days off is evaluated by removing the demand for all other shift types from the total workforce.

4.5.2. Symmetry Breaking

In addition to the above-stated modelling choices there is also the option of adding a symmetry breaking constraint to the model. Due to the nature of cyclic rosters there will always be multiple equivalent solutions to the problem. An example would be using a different line as the starting week of the schedule. The workforce would still work the exact same schedule but have a different starting point. Setting the weekend of the first line of the schedule for each contract to be a weekend off, removes all equivalent solutions that would start with a different line in the roster. This method therefore reduces the solution space of the problem without impacting the quality of the solution produced.

4.5.3. Reduced Complexity

Symmetry Breaking can be taken a step further by hard-coding some constraints and thus reducing problem complexity. An example of this would be to set weekends off in the model so that constraint (7) is satisfied before the solver is applied to the model. This method might not always work as it could remove all feasible solutions from a problem. But for larger instances it can reduce the runtimes significantly, which might be more desirable than finding a particular solution in some cases.

5. Early Testing: Benchmark Instances

To evaluate the performance of our models paired with different solvers we adapted the 50 benchmark instances used by Musliu et al. (2018) to fit our variation of the RWS problem. This way we produced two sets of instances for our analyses. Section 5.1 discusses our instance creation process. We then present the experimental analysis of the different solver—model combinations (Section 5.2).

All Experiments were carried out on an iMac running macOS Monterey Version 12.3.1 with a 3.1 GHz 6-Core Intel i5 processor and 8GB memory. The models were implemented using MiniZinc Version 2.4.3 and all experiments were run using the MiniZinc Python interface and Python 3.8. We imposed a runtime limit of 30 minutes on all instances and used MiniZinc's built-in solvers 'Chuffed' (Version 0.10.4) (Chuffed, 2022) and Gecode (Version 6.3.0)(Gecode, 2022) as CP solvers, and 'Gurobi' (Version 9.0.3) (Gurobi, 2022) as a MIP solver. We also tested our models with the CP solver from the OR-tools suite (Google Developers, 2022), which has won four gold medals in the 2021 MiniZinc Challenge [(Stuckey et al., 2014; MiniZinc Challenge 2021 results , 2021)], a global CP competition.

5.1. Experimental Set-up

For the purpose of comparability, we model our first set of experiments in this paper on the examples and methods used by Musliu et al. (2018). We consider here a simplified version of the problem outlined in Section 3, including constraints (1)–(4) (which limit work and shift blocks), constraints (8) (forbidden shift combinations), and the demand (9). Musliu et al. (2018) also includes a second type of forbidden shift combination, where a day off is forced between two shift types that cannot follow one another. The constraint is modelled as follows:

$$S_{i,j} = sh_1 \wedge S_{v(i,j+1)} = O \rightarrow S_{v(i,j+2)} \neq sh_2 \qquad j \in \mathcal{T}, (sh_1, sh_2) \in \mathcal{F}3_c, c \in \mathcal{C}, \tag{15}$$

where $\mathcal{F}3_c$ is the set of pairs of forbidden shift combinations with a day off in-between, for contract c. Teams and contracts are therefore the only difference between the problem studied in Musliu et al. (2018) and the one we have discussed so far. The aim of our experiments was to check what model—solver combination performs best when multiple contracts and team sizes are explicitly examined, and to look for any overlap with the findings from the state of the art approach.

An analysis of the 50 benchmark cases (Musliu et al., 2019) also studied in Musliu et al. (2018) found that the majority rely on the same set of forbidden shift combinations of length 2 and 3. Some instances only consider combinations of length 2, others do both, no instance only includes combinations of length 3. These two sets of forbidden combinations are then paired with different demand patterns and limits on work and shift blocks. In our work, we recreated the same pattern, based on the same set of forbidden shift combinations, in the same proportions. We then randomly sampled work requirements and demand patterns from the 50 benchmark cases to create the 24 instances considered in this section. Each instance has 2 contracts with different work requirements and the team sizes vary between 1 and 10, with the workforce size varying between 8 and 65 (average of 37). Musliu et al. (2019) also published a set of 2000 benchmark cases, of which we analysed the workforce size distribution, finding a similar spread of workforce size, with the majority lying between 9 and 51 (average of 42).

For the purpose of the experiments in this section we define six different models based on the modelling choices outlined in Section 4: (a) **direct** model—as in subsection 4.2; (b) **automata** model—as in subsection 4.3, with the work requirements (max/min work and shift blocks and forbidden shift combinations) encoded in an automaton; (c) **direct_T** and **automata_T** models—a variant of (a) and (b) in which demand satisfaction is expressed using the `global_cardinality`() constraint; (d) **direct_TO** and **automata_TO** models—a variant of the two models in (c) with the addition of the redundant constraint where the demand for days off is modelled through the use of demand matrix $R0$.

5.2. Results Analysis

Table 3 gives an overview of the results we obtained by testing all 24 instances paired with all four solvers. Column '#tot' gives the total number of instances run and '#tot avg. rt.' gives the total average runtime

solver	model	#tot	#tot avg. rt.	#sat	#sat avg. rt.
Gurobi	automata	24	41.00s	**24**	**41.00s**
	automata_T	24	106.71s	**24**	106.71s
	automata_TO	24	68.82s	**24**	68.82s
	direct	24	136.98s	23	64.65s
	direct_T	24	287.59s	21	71.47s
	direct_TO	24	153.11s	23	81.49s
Chuffed	automata	24	161.70s	22	12.73s
	automata_T	24	247.98s	21	26.21s
	automata_TO	24	96.98s	23	22.92s
	direct	24	49.52s	**24**	49.52s
	direct_T	24	523.83s	18	98.28s
	direct_TO	24	296.43s	21	81.56s
OR-tools	automata	24	803.47s	15	205.36s
	automata_T	24	946.68s	14	336.94s
	automata_TO	24	803.48s	15	205.38s
	direct	24	777.54s	14	46.97s
	direct_T	24	797.80s	14	81.68s
	direct_TO	24	777.52s	14	46.92s
Gecode	automata	24	714.15s	15	62.43s
	automata_T	24	703.01s	15	44.64s
	automata_TO	24	875.35s	13	92.55s
	direct	24	899.91s	13	137.96s
	direct_T	24	985.85s	12	171.30s
	direct_TO	24	969.32s	12	138.26s

Table 3: Average runtime for all instances and satisfied instances, for all model—solver combinations.

across all instances. Similarly, '#sat' and '#sat avg. rt.' give the number and average runtime across all solved instances. Bold entries highlight the shortest average runtime for all solved instances and the highest number of solved instances within the runtime limit. For this set of experiments we found that all instances were solvable, but for some instances certain solver—model combinations failed to find a solution within the given time limit of 30 minutes.

Gurobi and Chuffed were the only solvers that manage to solve all instances. Similar to Musliu et al.'s conclusions (Musliu et al., 2018), Gurobi performs best when paired with the automata models. Chuffed outperforms Gurobi with regards to runtimes (significantly when paired with the automata models). Chuffed produces the fastest runtimes overall, but solves fewer instances when compared to Gurobi. Musliu et al.'s conclusions (Musliu et al., 2018) that Chuffed performs best when paired with direct constraints is also confirmed here. Surprisingly, the CP solver from the OR-tools suite struggled to solve all instances (with over a third of them stopped by the runtime limit) and showed some of the slowest runtimes overall for solved instances. Gecode also failed to solve, on average, just under half of all instances within the set time limit, across all models. However, when just considering the satisfied instances, the runtimes are second only to the results produced by Chuffed.

When looking at the results from the different modelling options, we can see that for most solvers automata-based models produce faster runtimes and a higher number of solved instances. Furthermore, adding temporal requirements also shows a positive impact on overall runtimes. Furthermore, in all solver combinations, apart from Gecode paired with automata-based models, adding the redundant constraint of having demand for days off increases the number of solved instances within the time limit. This clearly shows the positive impact such modelling choices can have on the results.

Overall, the best solver—model combination is Chuffed paired with the direct model, and automata models perform better when paired with different solvers. One issue with automata-based models is the lack

of their user-friendliness for non-expert users. The instances investigated in this section have very simple shift block and order rules that are encoded in the automata. These rules are not necessarily realistic for real-life applications, as we came to see in some of our airport security applications.

6. Case Study —Outline

6.1. Problem Description

As part of this case study, we worked closely with an airport operator on their security staff rostering problem. To begin with, airport passenger security does not offer the best setting for adoption of cyclic rosters, for a reason we now explain.

Two types of demand co-exist in airport security. *Shift-based demand* (Ernst et al., 2004, Section 2.1), the same type addressed in RWS, manifests itself at a minority of physical locations within airports, such as security gateways dedicated to airport employees and contractors, or airline crews. At such locations, demand volumes are relatively low and in general easier to predict more accurately. *Flexible demand* (Ernst et al., 2004, Section 2.1) instead affects passenger security halls, the more critical (and visible!) components of airport security. In airport security halls, flexible demand is affected by both short- and long-term fluctuations in demand patterns, involving relatively high volumes of passengers, whose arrival processes are essentially uncontrollable— passengers reach security halls either from check-in halls or directly from the access route (airport train/tram/bus station or car park, etc.) they chose. Across the five to seven months of a typical airport security roster, long-term fluctuations are driven by changes in the flight timetable. These can be dealt with by creating suitable 'model weeks' that represent the variety seen in the timetable and can be used as reference from which to develop suitable rosters. However, short-term fluctuations on any day of operation can vary significantly, over time intervals of 15 minutes or less, being driven by the largely uncontrollable passenger flows and variable behaviour while passing through security (the latter can also have a tremendous impact on process efficiency, and is difficult to control). At this level, airport operators do not just schedule shift types for their staff, but 'proper' shifts with specific start and end times. Demand can still be predicted to a certain extent, but the model ought to be more detailed than the one outlined in Section 2.

On a separate note, airports are often heavily unionised workplaces, meaning that all rosters need to be jointly approved by the airport operator and union representatives. This usually happens through several meetings where the current roster is discussed and changes are suggested, ultimately to develop a roster that caters for the upcoming season(s). Every new version of the roster put forward for discussions requires, in general, considerable development time, as modifications tend to be manual and supported by spreadsheets, in the best cases we have seen. At our partner airport, the scheduler takes approximately two weeks for any single version of the roster produced, before this can be discussed with the unions. The goal of our case study was to facilitate these discussions with the unions, by semi-automating the scheduling process to speed up the discussion rounds.

On yet another different note, the scale of airport security rostering problems is generally larger than the one we have seen dealt with in articles such as the multiple-cited state of the art approach. At our partner airport, before Covid-19 times, the security workforce consisted of 222 workers, spread over six contracts, with different work and days-off regulations. The workforce had to cover up to ten different 'posts' (locations) across the airport, with different mixes of shift-based and flexible demand (either one of the two, or both). The majority of posts were located in the main security hall of the airport where all passengers had to pass through the security checks (hand luggage scans and metal detectors/body scans) before reaching their allocated gates. The security hall consisted of multiple lanes, each equipped with luggage scanners. Each lane pair shared a body scanner portal in between, thus creating a 'cell' (i.e. two lanes plus a body scanner). Demand forecasts around passenger arrival processes were then translated into an equivalent number of lanes to be kept open at different times of the day, in order to cope with the passenger influx. In addition, the number of workers required to open a new lane at any time depended on whether the other lane in the cell was already open or not.

One of the main rules laid out by the airport was that workers should be allocated to teams that always work 'together', meaning the members of a team have to work at the same location, throughout any given

shift, although locations assigned can vary from day to day throughout the planning horizon. Once created, a team could not be split into sub-teams, ever. In addition, a requirement was that each team would cycle through their schedule at least three to five times during the planning horizon (season) of five to seven months. While covering all locations would suggest using lots of small teams to foster higher levels of flexibility in covering the predicted demand at the requisite 15-minute granularity, this would quickly lead to too many lines in the roster, which in turn would make it impossible to meet the constraint on the requisite number of cycles to be carried out per season. This problem was solved by having varying team sizes by contract, thus keeping the flexibility of covering demand at all locations whilst maintaining the cycle for larger contracts to a reasonable length. An interesting, though brief, discussion on this aspect of synchronising cyclic rosters for a multi-contract workforce can be found in Laporte (1999).

A related, crucial aspect to security rostering at our partner airport is finding the best workforce split, given the available contracts, to facilitate the rostering process. Having a rostering tool that solves within minutes as opposed to weeks would be useful for testing the effect of any such decision on the resulting roster. This related decision making is also currently supported only by spreadsheets, in the best of the cases we have seen. The reality is that airports would prefer to have as many staff in their workforce as possible on many low hour contracts, and small teams, to gain more flexibility in their scheduling, something clearly unacceptable to the unions.

The problem setting described so far in this section is a large-scale rostering problem that goes beyond the allocation of days-off and shift types, but which can be solved by decomposing the rostering process into smaller sub-problems, one of which focuses on days-off and shift types and is modelled and solved according to the scheme discussed in the previous sections. The outcome of such a process is what our partner airport calls a *rough* roster. This schedule covers the days-off and shift-type (morning, afternoon, night) allocation, and is then used as input for the following step (out of scope for this paper), where shift start and end times are allocated.

Our partner airport's security rostering problem is indeed very similar to the RWS problem of Section 3. The most notable differences are the addition of the following constraints:

- each team must have two consecutive days off at least once every two weeks;
- there is an upper limit of teams of a certain size to be on shift each day due to location constraints at a later stage of the problem;
- there is an upper limit on how many days each team can work per line.

In addition, not all of the constraints from the general formulation in Section 3 apply here. Namely, there is no lower limit on the length of work blocks, nor an upper (a lower) limit on the length of shift blocks (constraints (2), (3), and (4) in the general problem formulation). Subsection 6.2 provides a detailed problem formulation for the rostering problem of our partner airport.

6.2. Problem Formulation and Implementation

To formulate the problem that our partner airport faces, some additional notation is required:

- cr_t: contract of team $t \in \mathcal{T}$
- d_up_c: maximum number of working days per line for teams under contract $c \in \mathcal{C}$
- v_off_t: binary indicator encoding whether team $t \in \mathcal{T}$ has been allocated a weekend off or not.
- $R6$: Requirement matrix for teams sized six. This matrix encodes the maximum number of teams of size six that can be rostered on any day for any shift type.
- x, y: integers used for defining the consecutive days off requirement. A typical form is to ensure that at least every x out of y weekends are set as off-work, where $x \leq y$.

Using this notation paired with the one outlined in Section 3 the airport's *rough* roster problem can be formulated as follows:

$$\sum_{j=1}^{g}(S_{i,j} \neq O) \leq d\_up_{cr_i} \qquad \forall i \in \mathcal{T} \tag{16}$$

$$\sum_{k=0}^{w\_up_c}(S_{v(i,j+k)} = O) > 0 \qquad j \in 1 \dots g, i \in ft_c \dots lt_c, c \in \mathcal{C} \tag{17}$$

$$\sum_{sh \in \mathcal{A}_c}\sum_{i=ft_c}^{lt_c}\sum_{j=1}^{g}(S_{i,j} = sh) = (lt_c - ft_c + 1)g \qquad c \in \mathcal{C} \tag{18}$$

$$S_{i,g} = O \wedge S_{i,g-1} = O \leftrightarrow w\_off_i = 1 \qquad i \in \mathcal{T} \tag{19}$$

$$\sum_{k=1}^{r}(w\_off_{u(i+k)} = O) \geq q \qquad i \in ft_c \dots lt_c, c \in \mathcal{C} \tag{20}$$

$$\exists(j \in \{1 \dots g-1\})\big((S_{i,j} = O) \wedge (S_{i,j+1} = O)\big) \leftrightarrow w\_off_i = 1 \qquad i \in \mathcal{T} \tag{21}$$

$$\sum_{k=1}^{y}(v\_off_{u(i+k)} = O) \geq x \qquad i \in ft_c \dots lt_c, c \in \mathcal{C} \tag{22}$$

$$S_{1,j} = sh_1 \rightarrow S_{v(i,j+1)} \neq sh_2 \qquad j \in \mathcal{T}, (sh_1, sh_2) \in \mathcal{F}, c \in \mathcal{C} \tag{23}$$

$$\sum_{c=1}^{nc}\big(ts_c \sum_{i=ft_c}^{lt_c}(S_{i,j} = sh)\big) = R_{sh,j} \qquad j \in 1 \dots g, sh \in \mathcal{A} \tag{24}$$

$$\sum_{c=1}^{nc}\big((ts_c = 6)\sum_{i=ft_c}^{lt_c}(S_{i,j} = sh)\big) \leq R6_{sh,j} \qquad j \in 1 \dots g, sh \in \mathcal{A} \tag{25}$$

Constraints (16) ensure that no team works more than the contracted number of days per line. Similarly, constraints (17) ensure that each block of consecutive work days over a contract's cycle is within the contracted limits for consecutive days of work. Constraints (18) restrict the assignment of shift types to be just out of the pool of shift types allowed for each contract. Constraints (19) and (20) model the required pattern of having q weekends off out of every r consecutive weeks. Similarly, constraints (21) and (22) model the requirement of having x out of every y weeks with at least two consecutive days off. Constraints (23) rule out forbidden shift combinations for each contract. Finally, constraints (24) ensure the workforce demand is satisfied by the schedule, while constraints (25) limit the number of teams of size 6 to ensure that location-specific demands at a later stage can be satisfied.

For the purpose of our experiments in the following section, we consider five different model variants: (i) **basic** model—a direct implementation of the constraints as outlined in this section; (ii) **global** model—where constraints (20) and (22) are modelled using `sliding_sum()`, and constraints (24) and (25) are modelled using `global_cardinality()` and `global_cardinality_low_up()`, respectively; (iii) **global_r** model—same as (ii), with the addition hard-coded weekends off constraints; (iv) **automata** model—where constraints (17), (18), (20), (22), and (23) are modelled through an automaton for each contract, and where constraints (24) and (25) are modelled using `global_cardinality()` and `global_cardinality_low_up()`, respectively; (v) **automata_r** model—same as (iv), with the additional hard-coded weekends off constraints (20) (incorporating them in the automata would increase model complexity unreasonably).

7. Case Study - Model Analysis and Impact

7.1. Instance Creation

To further investigate the efficiency of the proposed model and solver combinations we created a set of synthetic instances based on the case study data we were provided with. The synthetic instances come in 3 sizes (more on this choice in the concluding section):

solver	model	#tot	#tot avg. rt.	#sat	#sat avg. rt.
Gurobi	basic	30	1.22s	**30**	1.22s
	global	30	1.14s	**30**	1.14s
	global_r	30	1.09s	**30**	**1.09s**
	automata	30	138.11s	**30**	138.11s
	automata_r	30	3.76s	**30**	3.76s
Chuffed	basic	30	1313.26s	9	176.15s
	global	30	1347.96s	8	102.67s
	global_r	30	954.36s	16	213.70s
	automata	30	405.08s	24	55.99s
	automata_r	30	1081.88s	12	3.90s
OR-tools	basic	30	1742.69s	1	68.99s
	global	30	1800s	0	NA
	global_r	30	1337.51s	8	64.37s
	automata	30	566.15s	21	37.21s
	automata_r	30	1279.94s	9	65.46s
Gecode	basic	30	1800s	0	NA
	global	30	60.81s	29	0.83s
	global_r	30	120.75s	28	0.78s
	automata	30	547.45s	21	10.51s
	automata_r	30	600.54s	20	0.66s

Table 4: Average runtime for all synthetic instances and satisfied synthetic instances, for all model—solver combinations.

1. size 1: identical problem size to the case study example (48 teams with a total workforce of 222 staff members). The demand patterns are randomised and vary between flexible demand (for up to six lanes) and shift-based demand, for each day and shift type.
2. size 2: the original problem size is doubled (96 teams, 444 staff, up to 12 lanes).
3. size 3: the original problem size is tripled (144 teams, 666 staff, up to 18 lanes).

In total we created 30 synthetic instances (10 of each size) and ran them for all model—solver combinations outlined in the previous section.

7.2. Experimental Analysis

For these experiments we used the same experimental setup as outlined in Section 5, again with a runtime limit of 30 minutes imposed on all instances. Table 7.2 shows the results of our experiments.

Gurobi is the only solver that solves all instances for all models, with the fastest results being produced by pairing it with the **global_r** model. Gecode is the fastest solver for all models apart from **basic** —although not all instances are solved. This result is surprising compared to those in Section5 (Table 3) where it performed only better than OR-tools. This could be due to the increase in global constraint usage in the models in this Section, which facilitates the performance of a CP solver. The CP solver from the OR-tools suite performs the worst, with an average of 8 instances solved over all models, and the slowest runtimes overall. Chuffed performs slightly better, by solving almost double the amount of instances.

Overall, these results do not confirm Musliu et al's conclusions (Musliu et al., 2018) about the best model—solver combination (automata & Gurobi). For this set of realistic instances the best results are achieved by pairing Gurobi with non-automata models, or by pairing Gecode with non-basic models. Depending on what is seen as more important, Gecode produces the fastest results while Gurobi (although slightly slower) solves all instances.

From a practical point of view, automata based models are probably not the most suitable for real life applications as they are difficult to adapt to new work rules — something that is likely to be required. Instead, having a CP model with constraints that can be quickly added by someone familiar with the programming language will be much more user-friendly, which was an important aspect to our partner

airport. For automata-based models, this process would involve defining a new set of states describing all the possible cases of allowed work combinations. In our case this amounted to automata with the number of states ranging from 257 to 559 states per contract.

7.3. Early Impact

The full rostering problem that our partner airport faces is significantly more extensive than outlined in Section 6, with strict rules imposed on work hours, shift locations and related start and end times of shifts. The rough roster investigated here is only the first stepping stone to a fully fledged decision support system for workforce planning in airport security. However, the results of this study are a promising indication about the usefulness and potential impact of such systems. Inspired by these results, the airport has started to move away from solely spreadsheet based workforce planning. They are now testing the use of our approach, with the support of spreadsheets as both a front-end for data input as well as to collect and display the generated roster options to be put forward for discussion with the union representatives. Clearly, the testing phase currently involves rough rosters only, with the aim of striking some acceptable balance between the technical features of our approach and its ease of use by the airport scheduler. At the moment, we are we are collaborating with the airport on the development of the extended version of the tool, so that the full rostering problem faced by the airport can be tackled.

8. Summary and Conclusions

In this paper, we have investigated a new extension to RWS problems, introducing contracts and teams with multiple sizes. This was inspired by a real case study originating in airport security processes.

Our approach to modelling and solving retained all the main features of the state of the art for RWS problems, namely a solver independent, CP-based approach, grounded on a high-level modelling language, interfaces to multiple solvers, and control mechanisms over the search for solutions.

The earlier sections of our paper showed how to approach, both conceptually and implementation wise, RWS problems with multiple teams and contracts. We also investigated some benchmark instances of comparable size to those used by the state of the art approach, through which we empirically argued that the state of the art approach works even for the extended problem.

The later sections instead put our approach to the test of a realistic setting from airport security. The requisite changes to both problem formulation and implementation, with respect to the more general setting, were discussed in details. Our empirical results featured: (a) an overall confirmation that the state of the art approach scales well to problems of the size of our airport security case study; and (b) partially different findings around what the best model—solver combinations might be in this more complex setting, and why.

Overall, our work supports the argument that a solver-independent, CP-based approach for rostering with multiple teams and contracts can be very convenient, as it allows for fast changes amongst the solvers available, and quick comparison of the solutions provided by different model—solver combinations. Runtime performance from across all our experiments seem very reasonable, and orders of magnitude better than the current, highly manual, spreadsheet based approach. There is scope to further exploit the advantages of this approach in the future by automating the comparison between different solutions, produced by different model and solver combinations.

The synthetic instances for the experiments in Section 7.1 were originally based (instances of size '1') on the workforce required to staff the security hall of an airport accommodating about 15 mil. passengers per year (before Covid-19). Our results show the scalability of our approach to airport security halls three times as big. This is roughly equivalent to the number of passengers seen by the top 10 busiest airports in Europe before the Covid-19 Pandemic (The Port Authority of New York and New Jersey, 2019, p. 30). Larger airports are usually split into multiple terminals, with security often scheduled separately for each terminal. Heathrow airport in the UK, 7th busiest airport in the world, saw approximately 80mil. passengers in 2019 (The Port Authority of New York and New Jersey, 2019, p. 30), but spread over 5 terminals (on average, 16mil. per terminal). Similarly, the busiest airport in the world, Hartsfield–Jackson Atlanta International Airport, had 107 mil. passengers (The Port Authority of New York and New Jersey, 2019, p. 30) over

2 terminals (on average just under 55mil. passengers per terminal). These numbers suggest that the approaches outlined in this paper are applicable even to the largest airports in the world.

CRediT authorship contribution statement

Johanna Wiesflecker: Conceptualization, Methodology, Software, Validation, Formal Analysis, Investigation, Data Curation, Writing – Original Draft, Visualization, Project Administration **Maurizio Tomasella**: Conceptualization, Methodology, Resources, Writing - Review & Editing, Supervision, Project Administration **Daniel Guimarans**: Methodology, Resources, Writing - Review & Editing, Supervision **Thomas W. Archibald**: Methodology, Resources, Writing - Review & Editing, Supervision

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper

Funding Sources

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

References

Ackoff, R. L. (1978). *The Art of Problem Solving Accompanied by Ackoff's Fables*. John Wiley & Sons.

Baker, K. R. (1976). Workforce Allocation in Cyclical Scheduling Problems: A Survey. *Operational Research Quarterly (1970-1977)*, *27*, 155. URL: https://www.jstor.org/stable/3009134. doi:10.2307/3009134.

Bartholdi, J. J., Orlin, J. B., & Ratliff, H. (1980). Cyclic Scheduling via Integer Programs with Circular Ones. *Operations Research*, *28*, 1074–1085. doi:10.1287/opre.28.5.1074.

Bechtold, S. E. (1981). Work force scheduling for arbitrary cyclic demands. *Journal of Operations Management*, *1*, 205–214. doi:10.1016/0272-6963(81)90026-7.

Becker, T. (2020). A decomposition heuristic for rotational workforce scheduling. *Journal of Scheduling*, *23*, 539–554. URL: https://doi.org/10.1007/s10951-020-00659-2. doi:10.1007/s10951-020-00659-2.

Bennett, B. T., & Potts, R. B. (1968). Rotating Roster for a Transit System. *Transportation Science*, *2*, 14–34. URL: https://www.jstor.org/stable/25767466. doi:10.1287/trsc.2.1.14.

Cassandras, C. G., & Lafortune, S. (2008). *Introduction to discrete event systems*. Springer.

Chuffed (2022). The chuffed cp solver. URL: https://github.com/chuffed/chuffed [accessed: 02.08.2022].

Erkinger, C., & Musliu, N. (2017). Personnel scheduling as Satisfiability Modulo Theories. In *IJCAI International Joint Conference on Artificial Intelligence* (pp. 614–621). doi:10.24963/ijcai.2017/86.

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

Gärtner, J., Musliu, N., & Slany, W. (2001). Rota: A research project on algorithms for workforce scheduling and shift design optimization. *AI Communications*, *14*, 83–92. URL: http://www.ximes.com/.

Gecode (2022). Gecode: generic constraint development environment. URL: www.gecode.org [accessed: 02.08.2022].

Google Developers (2022). OR-Tools. URL: developers.google.com/optimization [accessed: 02.08.2022].

Gurobi (2022). Gurobi software. URL: www.gurobi.com/ [accessed: 02.08.2022].

Kiermaier, F., Frey, M., & Bard, J. F. (2016). Flexible cyclic rostering in the service industry. *IIE Transactions*, *48*, 1139–1155. doi:10.1080/0740817X.2016.1200202.

Kletzander, L., Musliu, N., Gärtner, J., Krennwallner, T., & Schafhauser, W. (2019). Exact methods for extended rotating workforce scheduling problems. *Proceedings of the 29th International Conference on Automated Planning and Scheduling, ICAPS*, (pp. 519–527).

Komarudin, Guerry, M. A., De Feyter, T., & Vanden Berghe, G. (2013). The roster quality staffing problem - A methodology for improving the roster quality by modifying the personnel structure. *European Journal of Operational Research*, *230*, 551–562. URL: http://dx.doi.org/10.1016/j.ejor.2013.05.009. doi:10.1016/j.ejor.2013.05.009.

Laporte, G. (1999). The art and science of designing rotating schedules. *Journal of the Operational Research Society*, *50*, 1011–1017. doi:10.1057/palgrave.jors.2600803.

Laporte, G., & Pesant, G. (2004). A general multi-shift scheduling system. *Journal of the Operational Research Society*, *55*, 1208–1217. URL: www.palgrave-journals.com/jors. doi:10.1057/palgrave.jors.2601789.

Maenhout, B., & Vanhoucke, M. (2009). The impact of incorporating nurse-specific characteristics in a cyclical scheduling approach. *Journal of the Operational Research Society*, *60*, 1683–1698. doi:10.1057/jors.2008.131.

Martin, S., Ouelhadj, D., Smet, P., Berghe, V., & Özcan, E. (2013). Cooperative search for fair nurse rosters. *Expert Systems with Applications*, *40*, 6674–6683. URL: http://dx.doi.org/10.1016/j.eswa.2013.06.019. doi:10.1016/j.eswa.2013.06.019.

MiniZinc Challenge 2021 results (2021). MiniZinc challenge 2021 results. URL: https://www.minizinc.org/challenge2021/results2021.html.

Mörz, M., & Musliu, N. (2004). Genetic algorithm for rotating workforce scheduling problem. In *ICCC 2004 - Second IEEE International Conference on Computational Cybernetics, Proceedings* (pp. 121–126). doi:10.1109/icccyb.2004.1437685.

Musliu, N., Gärtner, J., & Slany, W. (2002). Efficient generation of rotating workforce schedules. *Discrete Applied Mathematics*, *118*, 85–98. doi:10.1016/S0166-218X(01)00258-X.

Musliu, N., Schutt, A., & Stucker, P. J. (2019). Index of /staff/musliu/benchmarks. URL: https://www.dbai.tuwien.ac.at/staff/musliu/benchmarks/.

Musliu, N., Schutt, A., & Stuckey, P. J. (2018). Solver Independent Rotating Workforce Scheduling. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *10848 LNCS*, 429–445. URL: https://link-springer-com.ezproxy.is.ed.ac.uk/chapter/10.1007/978-3-319-93031-2_31. doi:10.1007/978-3-319-93031-2{_}31.

Nethercote, N., Stuckey, P., Becket, R., Brand, S., Duck, G., & Tack, G. (2007). Minizinc: Towards a standard CP modelling language. In C. Bessiere (Ed.), *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming* (pp. 529–543). Springer-Verlag volume 4741 of *LNCS*.

Olya, M. H., Badri, H., Teimoori, S., & Yang, K. (2022). An integrated deep learning and stochastic optimization approach for resource management in team-based healthcare systems. *Expert Systems With Applications*, *187*. URL: https://doi.org/10.1016/j.eswa.2021.115924. doi:10.1016/j.eswa.2021.115924.

Pesant, G. (2004). A regular language membership constraint for finite sequences of variables. In *International conference on principles and practice of constraint programming* (pp. 482–495). Springer.

Purnomo, H. W., & Bard, J. F. (2007). Cyclic preference scheduling for nurses using branch and price. *Naval Research Logistics (NRL)*, *54*, 200–220. URL: https://onlinelibrary-wiley-com.ezproxy.is.ed.ac.uk/doi/full/10.1002/nav.20201https://onlinelibrary-wiley-com.ezproxy.is.ed.ac.uk/doi/abs/10.1002/nav.20201https://onlinelibrary-wiley-com.ezproxy.is.ed.ac.uk/doi/10.1002/nav.20201. doi:10.1002/NAV.20201.

Rocha, M., Oliveira, J. F., & Carravilla, M. A. (2013a). Cyclic staff scheduling: optimization models for some real-life problems. *Journal of Scheduling*, *16*, 231–242.

Rocha, M., Oliveira, J. F., & Carravilla, M. A. (2013b). Cyclic staff scheduling: optimization models for some real-life problems. *Journal of Scheduling*, *16*, 231–242. doi:10.1007/s10951-012-0299-4.

Rocha, M., Oliveira, J. F., & Carravilla, M. A. (2014). A constructive heuristic for staff scheduling in the glass industry. *Annals of Operations Research*, *217*, 463–478. doi:10.1007/s10479-013-1525-y.

Stuckey, P. J., Feydy, T., Schutt, A., Tack, G., & Fischer, J. (2014). The minizinc challenge 2008-2013. *AI Magazine*, *5*, 55–60.

Stuckey, P. J., Marriott, K., & Tack, G. (2022). The minizinc handbook. URL: https://www.minizinc.org/doc-2.5.3/en/index.html [accessed: 20.03.2022].

The Port Authority of New York and New Jersey (2019). 2019 Airport Traffic Report. URL: www.panynj.gov/airports/en/statistics-general-info.html [accessed: 08.08.2022.

Van Den Bergh, J., Beliën, J., De Bruecker, P., Demeulemeester, E., & De Boeck, L. (2013). Personnel scheduling: A literature review. *European Journal of Operational Research*, *226*, 367–385. doi:10.1016/j.ejor.2012.11.029.

Wallace, M. (2020). *Building decision support systems: using MiniZinc*. Springer.