The Journal of Open Source Software

Computer-Aided Generation of N-shift RWS

Benjamin Edward Bolling[1]

1 European Spallation Source ERIC

DOI: 10.21105/joss.03431

Software

- Review
- Repository
- Archive

Editor: Mark A. Jensen

Reviewers:

- @magedhelmy1
- @ShantanuDash

Submitted: 03 May 2021
Published: 06 July 2021

License
Authors of papers retain copyright and release the work under a Creative Commons Attribution 4.0 International License (CC BY 4.0).

Statement of Need

All around the world, research institutes and industrial complexes make use of workforces working multiple shifts per day in order to utilise maximum efficiency and profitability of the facility. Creating shift work schedules has, however, always been a challenging task, especially such that are equal for all workers and at the same time distributes the shifts evenly and properly to prevent staff burnout.

The purpose and aim of this package is hence to support research institutes and industrial complexes at which non-standard working hours are applicable with a computational tool to create rotational workforce schedules by providing the user (schedule-maker) with all possible schedules for a set of input constraints/conditions (such as shift lengths, weekly working hours and -resting time) by constructing and utilising a Combinatoric Generator and a Cartesian Product calculator.

Conclusively, this package provides a graphical user interface (based on PyQt5 (Riverbank Computing Limited, 2016)) tool for generating and constructing acceptable shift arrays if there are any possible arrays following all user-defined constraints. These can be exported to the file formats ODS, CSV, and txt, with the arrays ready to be used as they are or as templates for further modifications (e.g. swapping shifts between workers and hence taking into account individual workers' needs).

Introduction

In order to achieve schedules for the workers that treats everyone equally, the focus of this package is on so-called rotational workforce schedules (RWSs). Rotational workforce schedules means that the schedule rotates after time, and hence, the other option would be static shift schedules. In this project, the term 'shift arrays' is defined to represent all possible schedules following a list of constraints, originating from e.g. country laws, organisational needs, and/or workforce requests.

Computational Approach and Results

In this approach, each worker has the same schedule shifted by one week, resulting in that all workers follow the same schedule. The project has been divided into two phases, *Boolean Shift Arrays* (in which boolean shift arrays are generated) and *From Boolean Shift Arrays to a RWS* (in which a selected boolean shift array is shaped into its final RWS layout).

Boolean Shift Arrays (phase 1)

A boolean shift array is defined such that 1 means that the worker is working and 0 that the worker is not. The input species (also known as constraints) and their respective values used are shown in Table 1 below.

DRAFT

Table 1: Constraints, i.e. the variables and their meanings, and some example values.

Variable	Meaning	Value
N	number of shifts per days	2
n_{cf}	number of days off clustered	2
n_S	number of shifts per shift cycle	18
n_W	number of weeks to cycle over	4
n_{wd}	number of working days per week	7
n_{wS}	Number of workers per shift (minimum)	1
t_d	daily minimum continuous resting time	11
t_r	weekly minimum single continuous resting time	36
t_s	shift lengths	8.33
t_W	weekly working hours per worker	36.00

Since each week also resembles a worker, the shift array can be set up as a matrix with 7 columns (each representing the days of a week) and $n_W/7$ rows (each representing a worker). The columns can then be summed to achieve the shift occupancy (or how many people are working each shift). Thus, the phase1 algorithm only allows shift arrays to pass for which all shifts are occupied by at least one worker, with a shift represented by the first n_{wd} days for each week. In order to extend to not only use single shifts but also 2- or 3-shifts, a logical condition was added into the algorithm: For N shifts per day, each day has to be filled with at least N workers.

In order to avoid all working days from being clustered together, the constraint for weekly minimum single continuous resting time is added (t_r). The algorithm ensures that all passed shift arrays have at least t_r hours of free-time over any given 7-day period.

The number of shifts per shift array is calculated by

$$n_S = \text{ceil}(t_W/t_s) \tag{1}$$

with the reason for using ceiling function (and not the floor function) being the argument that it is better with a couple of more hours than fewer. In order to cluster days off (n_{cf}), the algorithm's GUI has an optional additional constraint that serves this purpose and simply does not allow shift arrays with 0:s in clusters less than this through.

By using the input $n_W \times n_{wd}$ as the iterable and n_S as the length of subsequences of elements from the iterable, the same methodology as the *combinations* function of the *itertools* module in Python (Riverbank Computing Limited, 2020) (a combinatoric generator) is used for creating each shift array. By imposing the other inputs as constraints on whether a shift array should be appended to accepted shift arrays, the reason for not using the built-in Python module becomes clear: Python's built-in module returns all array combinations that are possible without any imposed constraints which quickly escalates to becoming too large for a personal computer's internal memory to handle.

With this, the final result is an array of shift arrays in which each shift array is filled with $7n_S$ 1:s and $n_W(7 - n_S)$ 0:s whilst obeying the above mentioned constraints. The number of possible combinations (C) can be then be expressed as:

$$C = n_W \times \frac{n_{wd}!}{n_S!(n_W \times n_{wd} - n_S!}. \tag{2}$$

DRAFT

From Boolean Shift Arrays to RWS (phase 2)

In this phase, a new list of combinations with free days clustered in pairs has been generated and a combination selected to proceed with (combination 212 as it has two out of four weekends off (note the zeroes in the bottom table in Figure 2 to the right).

Pressing the *Find solutions* results in what is shown in Figure 3 (right figure). A schedule can also be constructed completely by hand, but note that the algorithm will find all possible combinations that obey the given constraints. The algorithm is a Cartesian Product calculator, in which each set is a list of shifts (1 = Day, 2 = Evening, etc.) with one set per working day:

$$\text{combinations} = \begin{pmatrix}1\\2\\\vdots\end{pmatrix} \times \begin{pmatrix}1\\2\\\vdots\end{pmatrix} \times \ldots \times \begin{pmatrix}1\\2\\\vdots\end{pmatrix} = \prod_{i=1}^{n_{wd}} \begin{pmatrix}1\\2\\\vdots\end{pmatrix}_i = \begin{cases}[1\ 1\ \ldots\ 1]\\[1\ 1\ \ldots\ 2]\\\vdots\\[2\ 2\ \ldots\ 1]\\[2\ 2\ \ldots\ 2]\end{cases} \tag{3}$$

where each array in the resulting product is considered as a possible shift schedule matrix. Imposing constraints (resting time between shifts and ensuring all shifts are filled) on each combinations results in solutions from which the user can choose between.

Since all combinations are stored in a matrix form before different combinations are removed from the final solutions matrix, large datasets require severe amount of internal memory for the Cartesian Product method to work. For this, a controlling script has been implemented which calculates a pre-estimate of required internal memory. The required internal memory for different operations can be roughly calculated by

$$IM \approx N_C \times n_S = N^{n_S} \times n_S \tag{4}$$

returning the memory demand IM in bytes and where $N_C = N^{n_S}$ is the total number of combinations (without any constraints imposed).

If the estimated expected internal memory requirement for an operation exceeds 1Gb, the user is prompted whether to continue with the default Cartesian Product method or to use a less internal memory demanding recursive method.

Benchmarking results

Benchmarking Computer Specifications

The algorithm benchmarking was done on an Apple MacBook Pro with the specifications as defined in Table 2.

Table 2: Benchmarking computer specifications.

Definition	Value
Computer type:	Apple MacBook Pro (13-inch, 2019)
OS:	macOS Mojave v. 10.14.6
Processor:	2.8 GHz Intel Core i7 processor
Internal Memory:	16 GB 2133 MHz LPDDR3
Graphics Card:	Intel Iris Plus Graphics 655 1536 MB

DRAFT

Benchmarking Phase 1

In the GUI, there is a "fast generation" checkbox which stops the algorithm from further calculations once the first 100 approved combinations have been found. This way, computation time can be lowered (in comparison to "full generation" which will go through all possible combinations from the boolean array). For the parameters defined in Table 1, the time it took to complete decreased from 508.7 s (for a full generation) to 24.55 s (for the full generation) (see Table 3), which is a decrease in time by 95%.

The parameters used are defined in Table 1, with the exception of N and Shift types' labels. Note that for Table 3, the number (#) of weeks given is the minimum amount of weeks required for a full shift cycle in order to find acceptable combinations for the N-shift problems (with $N = 1, 2, 3$ for single-, two- and three-shifts, respectively). The free days clustering option is not selected for the benchmarking.

Table 3: Benchmarking for fast and full generation of the Boolean Arrays (as defined in Section 3.1 for Phase 1), and the number of combinations and approved combinations found for full generations of the Boolean Arrays (as defined in Section 3.1 for Phase 1). The types are single-, two- or three-shifts during 5 or 7 days per week.

Type:	# of weeks:	Total # of Combinations:	Approved Combinations:	Time (fast) [s]:	Time (full) [s]:
1-shift, 5d/w	1	1	1	7.224e-05	7.224e-05
1-shift, 7d/w	2	2 002	462	1.497e-02	5.211e-02
2-shift, 7d/w	4	1.312e+07	1.668e+06	24.55	508.7
3-shift, 7d/w	5	1.476e+09	1.138e+07	3 087	6.627e+04

Plotting the benchmarking results yields the logarithmic graph in Figure 4. As can be seen, the computation time T_C increases exponentially with the number of weeks in a shift cycle on average in accordance with

$$T_C(\text{full}) = \exp\{5.046 \times n_W\} \times 9 \times 10^{-7} \tag{5}$$

and

$$T_C(\text{fast}) = \exp\{4.254 \times n_W\} \times 2 \times 10^{-6} \tag{6}$$

for the full and fast generations, respectively.

DRAFT

Benchmarking Phase 2

If the given combination has only a single shift specie, there is one solution for the given combination. If there are more than one shift specie, multiple solutions may be found. The main impact on time consumption is the number of combinations N_C. Limiting factors are not limited to time only but also on the internal memory due to that a Cartesian Product method is used, meaning all combinations are stored as list objects. Some values have been timed and calculated in Table 4 using the Cartesian Product method.

Table 4: Benchmarking for Phase 2: Time and estimated internal memory (IM) required for obtaining all combinations and solutions for different n_S and N using the Cartesian Product method.

Type (N):	n_S:	n_W:	Combinations:	Solutions:	IM:	Time [s]:
2-shift	14	3	16 384	7	229.38 kB	0.2963
2-shift	18	4	262 144	64	4.7186 MB	5.843
3-shift	14	3	4 782 969	0	66.96 MB	92.54
3-shift	18	4	387 420 489	-	6.9736 GB	-

Conclusions

In this project, an algorithm has been constructed which generate schedules for different number of weeks to cycle over. The current issue is that the computational complexity (and hence the required computation time) increases with the number of weeks per cycle, as can be seen in Table 3 and Figure 4. This means that for a higher amount of weeks in a shift cycle, this application will need further development in order to have more efficient ways of finding the solutions and/or deployment of the application onto super-computers for generating the Boolean Arrays.

For up to 5 weeks in a shift cycle it is possible to use a general-purpose computer such as the benchmarking Apple MacBook Pro with specifications defined in Table 2.

It has thus been demonstrated that the application can be used to generate 1, 2 and 3-shift schedules. Future development plans include adding an automated assignment function of shift types in phase 2, which would further strengthen the usability of this application.

Figures

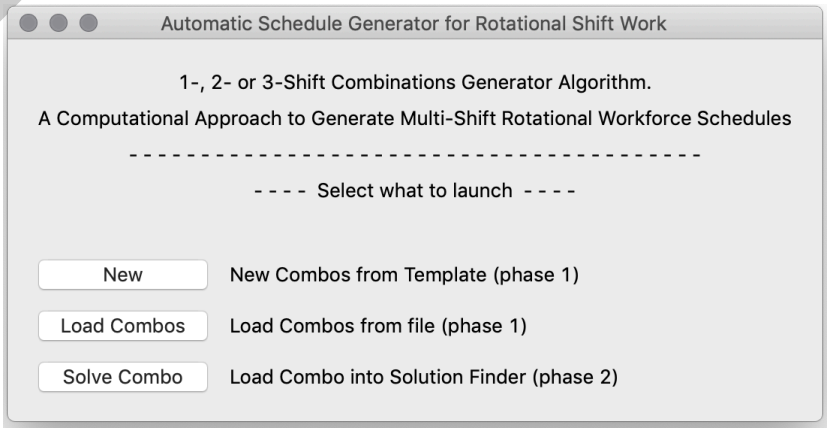

Figure 1: The RWSing Application's launcher.

DRAFT

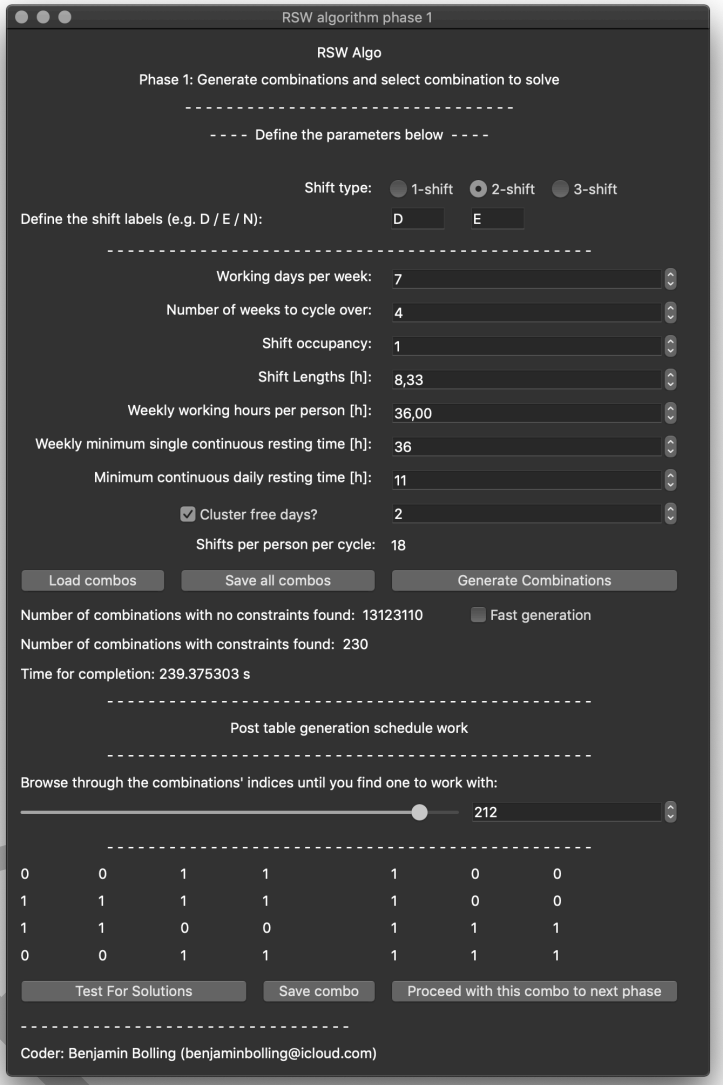

Figure 2: The RWSing Application's algorithm's "phase 1 GUI," in which the combinations have been generated.

The Journal of Open Source Software

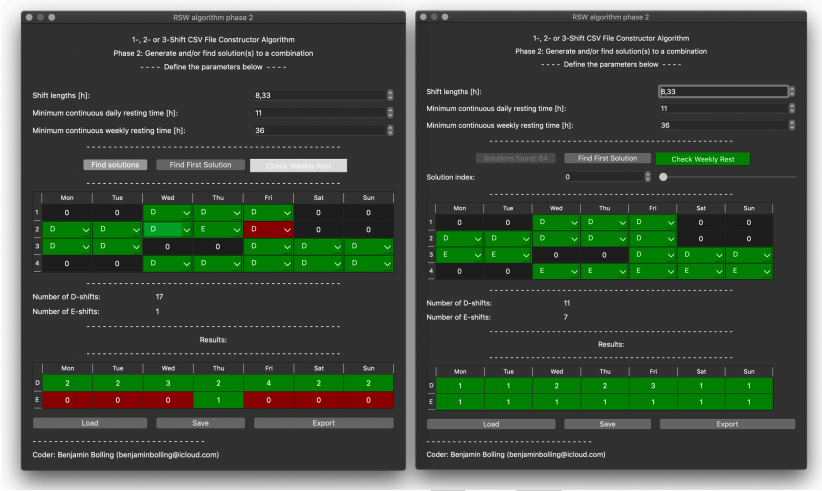

Figure 3: The RWSing Application's algorithm's "phase 2 GUI" as launched from the "phase 1 GUI" and with the second Thursday's shift changed to an evening shift (left) and after finding solutions, showing the first solution (right).

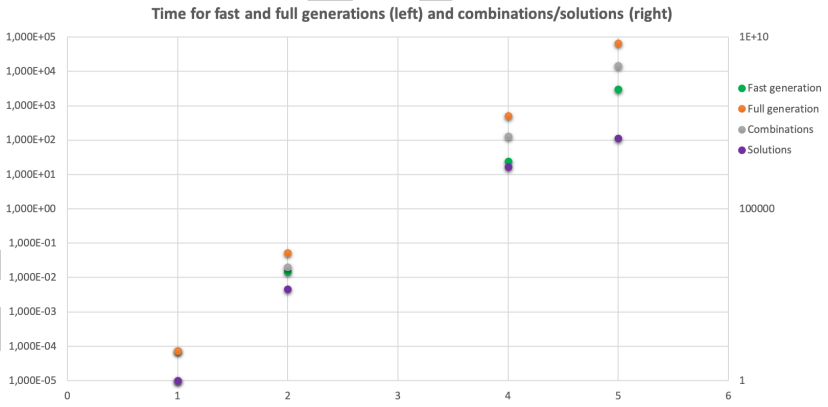

Figure 4: The benchmarking results in respect of time for fast- and full generation of the boolean arrays (on the left vertical axis), and the number of combinations gone through and the solutions found (on the right vertical axis).

Acknowledgements

The author wants to thank his direct line-manager at European Spallation Source for asking the question if it would be possible to create a software for generating shift schedules, which lead to the idea of creating this project and after a while lead to this final state. The author also wants to thank the reviewers for taking their time reviewing this project.

References

Riverbank Computing Limited. (2016). *PyQt5: Python bindings for the Qt cross platform UI and application toolkit*. https://www.riverbankcomputing.com/software/pyqt/

Riverbank Computing Limited. (2020). *Python Language Reference* (Version 3.8.2). http://www.python.org