Technical Manual

Medtrics Automated Scheduling

by J.A.S.T Senior Design team

Team members:

Jasper Ding

AC Li

Tung Phan

Son Pham

04/30/2016

Table of Contents

1. Summary of the problem

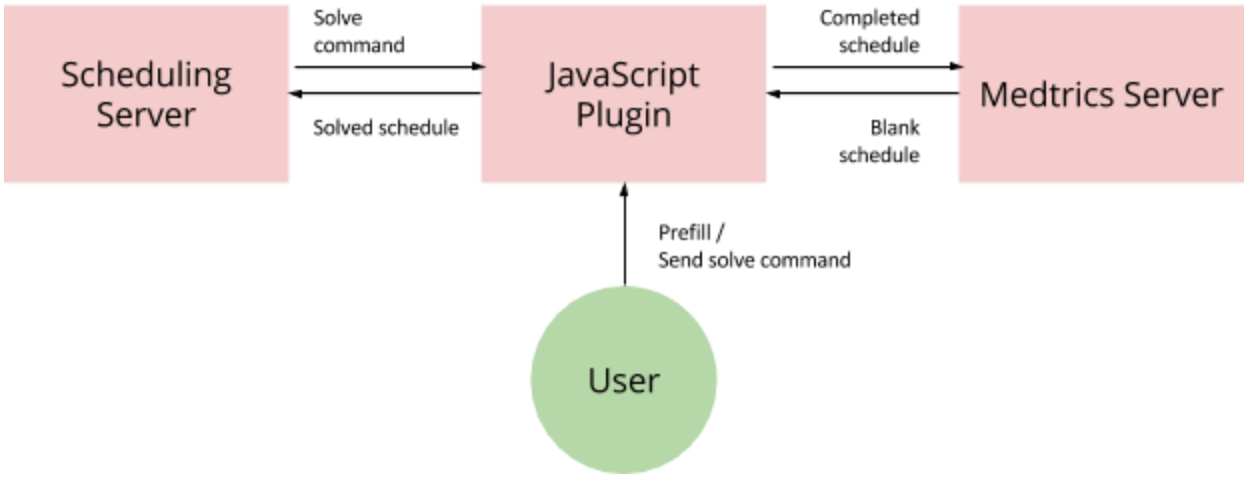
Rotation scheduling remains one of the most time-consuming problem in residency management. Every year, many coordinators work full-time in the span of six weeks just to create a schedule that can satisfy the teaching demands of each department, while still allowing the trainee to graduate with sufficient requirements. There have been many attempts to develop a digital solution in the past, but these attempts have largely been unsuccessful.

The reason for this is that most effort to solve these problems have been done by either individual researchers or large corporations, but not small businesses / start-ups. Researchers often build scheduling algorithm for one specific hospital to work with. Their softwares work really well for that particular problem but fail to generalize to a variety of specificities when scaling up. Start-ups, on the other hand, don't quite have the manpower to focus on such a specific problem like scheduling. Even when they do, they can't survive solely on the scheduling algorithm because program coordinators want a full residency management platform with multiple different features. On the other hand, large corporations on the health care and residency management industry have been slow to adapt to new technology. It is hard to incorporate such a specific feature as rotation scheduling into their wide array of products and this trend should continue for at least 5 years.

This gives the our Senior Design team a unique opportunity. Medtrics is a company that produces residency management software with many different features. The company is currently responsible for helping more than 15 institutions across the nations; it is big enough to have a wide array of different programs but small enough to actually incorporate the algorithm into the system. The company, however, simply doesn't have the manpower to invest on such a specific task and that's where the Senior Design team can contribute its effort to the company. Successful implementation of this feature into Medtrics system would give Medtrics a significant advantage against competitors, as this feature is in high demand across hospitals in the nation.

2. Interface between GUI, Server and Medtrics

We currently assume that we will receive some form of text from Medtrics. Upon receiving the text, the JavaScript plugin parses the text and then shows a blank schedule. Users can interact with the JavaScript plugin to prefill schedule and then send a JSON object to the Server, which the Server then solves and sends back a text string to client.



Three components of the project communicate through usual POST/GET HTTP commands.

Originally, we planned to implement our system to directly interact with the REST API provided by Medtrics. However, many features have not been implemented on Medtrics side due to time constraint of the project.

Some of the next step Medtrics should take:

- Ensure encryption when JavaScript Plugin interact with Medtrics. Current string transfer is not safe especially given the amount of trainee information present.
- Implement all of the constraint and extra attributes required for scheduling on Medtrics server. These include but not limited to:
    - Extend scheduling capability to more than 5 roles.
    - Automated relaxation of constraints for Solver algorithm.
    - A warning page that notifies the coordinator some deficiencies about the constraint.
- Implement REST API within JavaScript Plugin to ensure “RESTful” data transfer.
- Integrate the scheduling platform.

3. Graphical User Interface

Graphical Interface starts as a side project to visualize the performance of the scheduling algorithms, but ends up being a core part of the final deliverable due to its incredibly helpful features. It is now a valuable asset for the co-ordinator to use as a scheduling and tweaking tool.

Main components of the GUI should be fairly easy to identify and use:

1. **Top bar:** Include basic functionality such as upload schedule, clear schedule, solve schedule and save the schedule to CSV
2. **PGY1, PGY2 and PGY3 role tabs:** Allow the user to view different roles. Currently only supports 3 roles.
3. **Warning tab:** Guide the user to a page that describes some deficiencies in the problem constraints. *Still under construction.*
4. **Student schedule:** Show the schedule of each student, each color represents a rotation.
5. **Underdone:** The number of remaining requirements students still have to take
6. **Overdone:** The number of rotations the students have to take more than required.
7. **Min/Max Inspection:** Shows whether current schedule satisfies min max requirement of a specific requirements. Potential improvement includes:
    - Every time a rotation is scheduled, immediately show the inspect of that rotation.

- If the min max requirement is a combination of different roles, shows the requirements of other roles in lighter color and indicate specifically that this is a combined requirement.

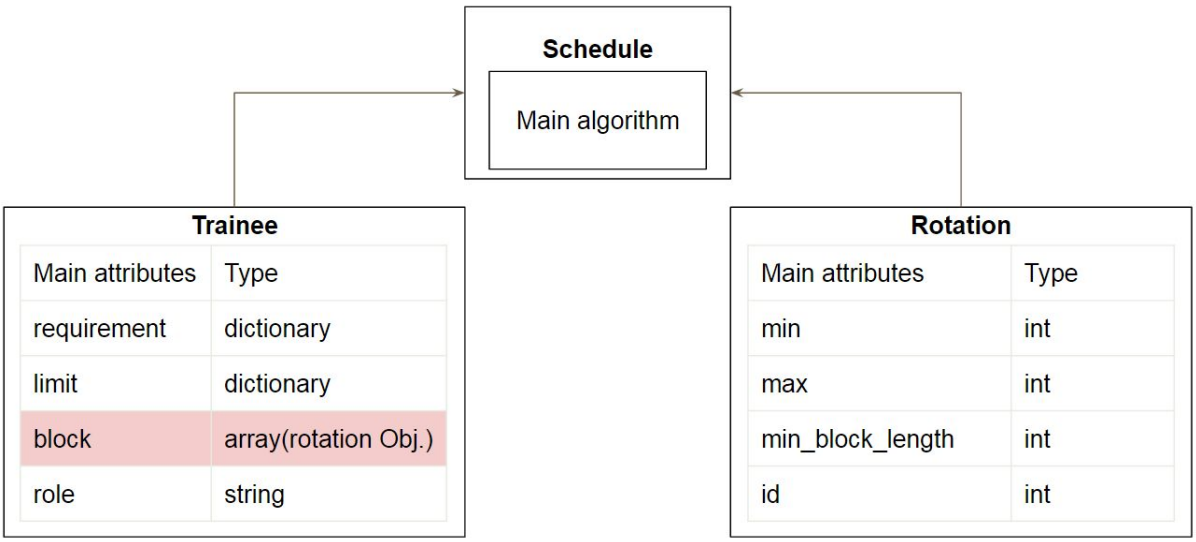
8. **Explore / Edit:** Allow users to switch between explore and edit mode.

For most of the GUI, we use `PIXI.js`, which is a very efficient webGL drawing libraries. The libraries take advantage of the GPU in order to runs graphical shapes efficiently on the webs. Interactions with HTML elements are handled by jQuery. Since all JavaScript files are included in the repository, Medtrics should be able to run the program by simply following the instruction in the `README.md`.

4. Scheduling Algorithms

4.1 Main classes

4.1.1 Broad picture



As the broad picture shown above, we mainly have 3 classes, which are: schedule class, trainee class and rotation class. The schedule class contains the main algorithm, which includes 4 big steps (actually 8 steps in total.)

4.1.2 Trainee class

4.1.2a Main attributes

As shown in the broad picture above, there are 4 main attributes in the Trainee class.

1. `role`: The attribute *role* is the role of the trainee. It will be PGY1, PGY2 and PGY3, which correspond to the class year of the trainee. Mainly, different roles have different sets of requirements.
2. `reqs`: The attribute `reqs` is the requirements need to be satisfied by the trainee throughout the year. The attribute `reqs` is a dictionary of rotation objects along with a request number. It

We save the original requirements, which are given by the user, in the *base_reqs* to protect the data from being changed through the algorithm. And the update will show in *processed_reqs*.

3. `limits`:The attribute limits is the maximum number of each rotation a trainee can take throughout the year. The attribute limits is a dictionary of rotation objects along with a limit number. Mainly, the attribute limits is given by the user and consistent with the role the trainee. We save the original limits in the *base_limits* to protect the data from being changed through the algorithm. And the update will show in *processed_limits*.
4. `block`: The attribute *block* holds the schedule for the trainee. It is initialized as an array with rotation objects which are empty. The final schedule of the trainee will saved in this attribute after running the algorithm.

4.1.2b Functions

Except for the getter and setter functions, functions are printer functions or helper functions which are used to test, debug and display. There are two meaningful functions, calculating the number of overdone and underdone blocks.

1. overdone_blocks: This function will return the number of overdone blocks for the trainee. The number of overdone blocks is the number of rotations that the trainee has been assigned to take minus the number of the rotations that the trainee is required to take. Briefly, the trainee takes the number of rotations more than the requirements.
2. underdone_blocks: This function will return the number of underdone blocks for the trainee. The number of underdone blocks is the number of the rotations that the trainee is required to take minus the number of rotations that the trainee has been assigned to take. Briefly, the trainee takes the number of rotations less than the requirements.

4.1.3 Rotation class

4.1.3a Main attributes

1. `name`: The attribute *name* is the name of the rotation. It is a way of identification of the rotation.
2. `id`: The attribute *id* is the id of the rotation. It is another way of identification of the rotation.
3. `min_block_length`: The attribute *min_block_length* is the minimum block length of the rotation. There will be rotations having different minimum block lengths, which can be 0.25 (a quarter block: a week), 0.5 (a half block: two weeks) and a whole block (a full block: four weeks).

4. `min`: Each rotation should have at least minimum number of trainees along with different roles (and combined roles) for each quarter block. Throughout the whole year(through all 13 blocks or 52 quarter blocks), the minimum number of rotations should be satisfied.
5. `max`: Each rotation should have at most maximum number of trainees along with different roles (and combined roles) for each quarter block. Throughout the whole year(through all 13 blocks or 52 quarter blocks), the maximum number of rotations should be satisfied.

4.1.3b Functions

Except for the getter and setter functions, functions are printer functions or helper functions which are used to test, debug and display.

4.1.4 Schedule class

4.1.4a Main attributes

1. `trainees`: The attribute trainees contains an array of trainee objects. When you initialize a schedule object, you need to give all data about trainees.
2. `rotations`: The attribute rotations contains an array of rotation objects . When you initialize a schedule object, you need to give all data about rotations.
3. `num_block`: The attribute num_block is the number of blocks. The default value of num_block is 52, which is 13 whole blocks times 4 (1 whole block can be divided into 4 quarter blocks).

4.1.4b Functions

1. `fill_in`: This function fills the given rotation into the given block for the given trainee and updates all corresponding min and max. The trainee given to this function can be trainee with any roles.
2. `fill_in_PGY1`: This function fills the given rotation into the given block for the given list of PGY1 trainees and updates all corresponding min and max. The list of trainees given to this function can only contain PGY1 trainees.
3. `fill_in_PGY2`: This function fills the given rotation into the given block for the given list of PGY2 trainees and updates all corresponding min and max. The list of trainees given to this function can only contain PGY2 trainees.

4. `fill_in_PGY3`: This function fills the given rotation into the given block for the given list of PGY3 trainees and updates all corresponding min and max. The list of trainees given to this function can only contain PGY3 trainees.
5. `greedy_step1_4`: This function goes through each whole block for each rotation. For each rotation, a helper function finds at most (rotation min) number of trainees with unsatisfied educational requirements for that rotation. Finally, this function calls appropriate fill-in function to fill the schedule.
6. `greedy_step2_4`: This function goes through each whole block for each rotation. For each rotation, if it has min requirement unsatisfied, find trainees who do not take that rotation over the limit of that rotation to assign to this rotation. If there still exist rotations with unsatisfied min requirement during any period, find trainees who have any pre-fill vacation block and replace vacation to that rotation.
7. `greedy_step3_4`: This function follows the rule: if there exist trainees with unsatisfied educational requirements for some rotations, randomly pick a whole block to assign it to that rotation, meanwhile, the rotation max should not be touched. This step only fills the whole block.
8. `greedy_step4_4`: This function follows the rule: if there exist trainees with unsatisfied educational requirements for some rotations, randomly pick a half block to assign it to that rotation, meanwhile, the rotation max should not be touched. This step only fills the half block.
9. `greedy_step5_4`: This function follows the rule: if there exist trainees with unsatisfied educational requirements for some rotations, randomly pick a quarter block to assign it to that rotation, meanwhile, the rotation max should not be touched. This step only fills the quarter block.
10. `greedy_step6_4`: This function follows the rule: if there exist trainees with unsatisfied educational requirements for some rotations, after doing step 5, find a whole block occupied by vacation to replace it to the rotations, meanwhile, the rotation max should not be touched. This step only fills the whole block.
11. `greedy_step7_4`: This function follows the rule: if there exist trainees with unsatisfied educational requirements for some rotations, after doing step 5, find a half block occupied by vacation to replace it to the rotations, meanwhile, the rotation max should not be touched. This step only fills the half block.
12. `greedy_step8_4`: This function follows the rule: if there exist trainees with unsatisfied educational requirements for some rotations, after doing step 5, find a quarter block occupied

by vacation to replace it to the rotations, meanwhile, the rotation max should not be touched. This step only fills the quarter block.

4.2 Greedy Algorithm

4.2.1 Motivation

Our first approach "Greedy Algorithm" is to use some human-defined heuristics in order to solve the schedule. This approach is relatively simple and straight-forward to implement (because it is essentially a set of rules of thumbs to schedule effectively). This algorithm runs really fast and usually solve roughly 95% of the schedule. The algorithm is also very adaptive and can deal effectively with conflicting constraints. It is recommended that if a problem turns out to have no optimal solution and is unsolvable by the Solver Algorithm, then the co-ordinator should use the Greedy Algorithm to solve 95% of the schedule and use other user-friendly tools to solve the rest 5%.

4.2.2 Overview

In the beginning of the project, we did a lot of research on how to solve a scheduling problem. Unlike solver algorithm, which only needs to define constraints clearly and carefully, greedy algorithm is a bit different. For greedy algorithm, we need to decide which constraint to satisfy first. Because the hospital needs trainees or residents to be responsible for different rotations or departments throughout the year, we chose to satisfy the rotation minimum requirement first, which is also consistent with the paper we read in the beginning of the project.

Basically, we have our greedy algorithm mainly based on four big steps:[1]

1. If there exists a rotation in some period whose teaching service demands have not been satisfied and for which some resident has unsatisfied educational requirements then choose an unassigned resident with non-zero educational requirement to cover that rotation. Repeat until no more residents can be assigned.
2. If no resident exists with unsatisfied educational requirements for a particular rotation that has unsatisfied teaching service demands in some period, choose an unassigned resident to assign to this rotation.

[1] Complexity results for the basic residency scheduling problem https://paperpile.com/c/qeisMn/nPyk

3. If residents exist with unsatisfied educational requirements for some rotations, choose a period to assign them to that rotation.
4. If residents exist with unsatisfied educational requirements for some rotations, choose a period occupied by vacation to replace them to that rotation.

4.3 Scheduling Algorithm - Integer Programming Solver

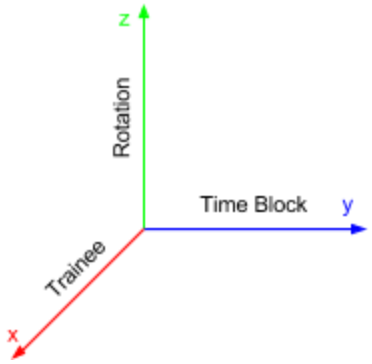
4.3.1 Overview

We translated the rotation scheduling problem into a Integer Programming (IP) Problem, and used Google or-tools [2]linear solver wrapper to interact with the Coin-or Branch and Cut (CBC) Solver[3]. Each rotation demand, graduation requirement, etc. is translated into a mathematical constraint and then inputted into the solver. Since Integer Programming is NP - hard, it is natural that finding a globally optimal solution (if there exists one) requires more time than the locally optimal Greedy solution, even with the help of state-of-the-art third party solvers.

In one of our test cases, a scheduling problem with 120 trainees, 7 rotations in 52 week period was translated into a IP problem with 49920 variables, 10112 constraints and takes up to 5.6 seconds to solve.

4.3.2 Schedule representation in IP Problem

A schedule is represented by a set of points on 3D space with axes: Trainee, Time, Rotation.



[2] Google or-tools https://developers.google.com/optimization/
[3] Coin-or Branch and Cut Solver (CBC) https://projects.coin-or.org/Cbc

For example, Trainee 12, during Time Block 34, doing Rotation 5 can be thought of as point (12, 34, 5) in this 3D space. The way we implemented this representation in code is a 3D array (list of lists of lists in Python) containing 0 or 1. The previous example will be represented by

```
trainee = 12
block = 34
rotation = 5
schedule[trainee][block][rotation] = 1
```

4.3.3 Constraints in IP Problem

4.3.3a A trainee can only do one thing at any given time

(SolverUtil.py solveSchedule() Line 323)

For Trainee **i**, during Time Block **j**, across Rotation **k**'s

$$0 \leq \sum_{k} schedule[i][j][k] \leq 1$$

4.3.3b A rotation has a min and a max number of trainees at any given time

(SolverUtil.py solveSchedule() Line 330)

During Time Block j, in rotation k, there is a min and max number of trainees

$$min\_PGY1 \leq \sum_{i \in PGY1} schedule[i][j][k] \leq max\_PGY1$$

For "or" constraints (PGY2 or 3), the inequality has the same structure.

4.3.3c A trainee has a min and a max number of blocks dedicated to each rotation

(SolverUtil.py solveSchedule() Line 362)

For Trainee **i,** across Rotation **k'**s, the total number of occupied blocks lies in a range. Min = Rotation Graduation Requirement, Max = Rotation Limits

$$min\_Rotation_k \leq \sum_{j \in Blocks} schedule[i][j][k] \leq max\_Rotation_k$$

4.3.3d Solve using prefilled schedule

(SolverUtil.py solveSchedule() Line 316)

If a coordinator wants to make sure that Trainee i does rotation k in time block j, then in the pre-filled schedule, we would have

```
prefilled_schedule[i][j] = k
```

Translating this to the solver involves setting schedule[i][j][k] to 1 and all other schedule[i][j][k'] to 0 (with rotation number k' not equal to k).

4.3.3e Rotations that allows vacations

(SolverUtil.py solveSchedule() Line 310)

If a coordinator wants to put a student in a Rotation that allows vacation, then we would have

```
prefilled_schedule[i][j] = -2
```

Translating this to the solver involves setting all schedule[i][j][k'] to 0 (with rotation number k' denoting rotations that do **not** allow vacations). The other rotations can still be 0 or 1.

4.3.3f Objective: Minimize total number of blocks trainees have to take

(SolverUtil.py solveSchedule() Line 388)

We want to have a schedule as sparse as possible, while still maintaining the rotation workforce demands and student graduation requirements.

$$min \sum_{i,j,k} schedule[i][j][k]$$

4.3.4 Handling half and quarter blocks for Solver

4.3.4a FullHalfQuarter functions

There are 3 functions belonging in this class. Their general purpose is to allow for scheduling Half (2 week) and Quarter (1 week) blocks. The Solver will try to solve from lower resolution (Full 4 week) up to higher resolution (Quarter 1 week). These functions, together with the pruning/doubling step below, got inspirations from Simulated Annealing. Reader should read more about this topic if interested.

The specific details about each function is below:

- `generateFullHalfQuarter(req):` Convert a numeric double requirement value into a set of 3 integers representing requirement counts from low resolution to high resolution. We want to relax the requirement to the nearest lower integer value to allow more room in future, more detailed solving steps. E.g: 1.75 -> (1, 3, 7) because floor(1.75) = 1, floor(1.75 x 2) = 3, floor(1.75 x 4) = 7

- **generateFullHalfQuarterDict**(trainee_req): Apply **generateFullHalfQuarter**(req) to all key-value of trainee_req to get 3 dictionaries containing the 3 types of requirement sets
- **generateFullHalfQuarterPrefilled**(prefilledSchedule): Convert the given prefilledSchedule (which is in 52 week format) into 3 prefilled schedules with different formats (Full 13, Half 26, Quarter 52). Whenever we can't combine 2 blocks due to them having different rotations at higher resolution, we use int -3 to signify ambiguity in the lower resolution prefilled schedule.

4.3.4b Pruning and Doubling Schedules

- **doubleSchedule**(currentSchedule, prefilledSchedule)

  We start by doubling the solved schedule of lower resolution into higher resolution. This is done by either disambiguating the higher resolution blocks using prefilled schedule (when we see a ambiguous -3), or simply including the rotation number twice in the higher resolution schedule.
- **pruneSchedule**(currentSchedule, prefilledSchedule, seed, num_trainee_list, rotations)

  Even after doubling, if we simply continuously solve at increasing resolution, there might not be any space left (Since we are only 'zooming' in without creating space). Therefore, we need to prune the solved schedule into a schedule sparse enough for more block filling. The pruning process starts by randomly choosing a trainee-time block as a candidate for pruning (using provided random seed). If any of the following happens, we cannot prune that block
  - ➔ If the Block is empty
  - ➔ If by pruning that block, we go below the min manpower requirement of a rotation
  - ➔ If the current rotation length is at min block length
  - ➔ If the block was pre-filled, as opposed to filled by scheduling algorithm

  Otherwise, we prune that block (set it to empty block -1) and continue with other trainee/time block combo until none is left.

5. Scrum

User Stories

	Jan	Feb	Mar	Apr
As a coordinator, I want helpful information to tell me what each button does.	3	3	3	3
As a coordinator, I want buttons so that I can pick different algorithms to schedule.	10	5	3	0
As a coordinator, I want an algorithm that automatically schedules for a schedule with at least 90% satisfaction rate.				
As a developer, I want to research several different algorithms to learn which way is the best and most effective	24	20	5	0
As a developer, I want a greedy algorithm to have a fast and easy solution to resort to.	30	25	10	5
As a researcher, I research on Integer Programming to gain knowledge about using solver tools to find the optimal solution.	24	18	10	4
As a coordinator, I want to edit the schedule after it's been generated.	24	20	15	8
As a coordinator, I want to use less clicks to schedule rotation for each student	16	10	5	3
As a coordinator, I want to verify that all requirements are fulfilled.	8	5	0	0
As a coordinator, I want to save the resulting schedule into a file for further usage.	8	4	2	0
As a coordinator, I want to upload all the trainees' data and requirements by uploading a file.	8	2	0	0
As a coordinator, I want to keep track of how different rotations are represented.	10	8	5	2
As Medtrics, I want the algorithm to be integrated so that it works with existing infrastructure and doesn't break the system.				
JSON - REST API	20	15	10	0
As a coordinator, I want to be notified that if an optimal schedule can be found.	10	6	4	2
As an Institution, I want the algorithm to be flexible with different constraints.	30	22	17	10
As a trainee, I want the algorithm to be aware of my status of the progress of fulfilling all my requirements so that I can graduate.	10	10	8	0

As a trainee, I want my rotation to be scheduled based on my preferences so that I can have other plans sometimes during an academic year.	20	15	8	0
As a coordinator, I want to have a way to change the scheduled calendar for the emergency.	20	12	7	0
As a developer and a coordinator, I want to visualize the resulting schedule to see it easier.	20	20	18	16
	295	**220**	**130**	**53**

Table 1. User stories and SCRUM sprints