The Scheduling Algorithm

Eddie Guo

July 7, 2024

I created this scheduling algorithm to optimize call schedule for physicians and healthcare trainees. This system aims to create fair and efficient schedules while respecting various constraints and preferences.

Contents

1 Scheduling Constraints

The scheduling algorithm adheres to several important constraints to ensure fairness, compliance with labor regulations, and resident well-being. These constraints are implemented throughout the scheduling process and are crucial for generating a valid and effective schedule.

1.1 No Back-to-Back Call Shifts

One of the primary constraints is the prohibition of back-to-back call shifts for any resident. This rule is implemented as follows:

- When considering a resident for a call shift on a given date, the algorithm checks if the resident was assigned to a call shift on the previous day.
- If a resident was on call the previous day, they are automatically disqualified from being assigned to the current day's shift.
- This constraint is enforced in the `canTakeCall` function, which returns `false` if the resident was on call the previous day.

This constraint helps prevent physician fatigue and ensures compliance with work-hour regulations.

1.2 Minimum Interval Between Calls

Each resident has a `max1inX` property that defines the minimum number of days that must pass between their call shifts:

- The `canTakeCall` function checks the time elapsed since a resident's last call.
- If the time elapsed is less than the resident's `max1inX` value, the resident cannot be assigned to the current shift.
- This constraint allows for personalized scheduling based on resident seniority, specialization, or other factors that might influence their call frequency.

1.3 Respecting Days Off

The scheduler respects pre-defined days off for each resident:

- Each resident object includes a `daysOff` array containing dates when the resident is unavailable.
- The `canTakeCall` function checks if the current date is in the resident's `daysOff` array.
- If the date is found in `daysOff`, the resident is not considered for that shift.
- This constraint allows for accommodation of vacations, conferences, personal days, and other scheduled absences.

1.4 Fair Distribution of Shifts

While not a hard constraint, the scheduler aims to distribute shifts fairly among residents:

- The `getResidentScore` function considers the total number of shifts already assigned to a resident.
- Residents with fewer assigned shifts are given a higher score, increasing their chances of being selected for future shifts.
- This soft constraint helps balance the workload among residents over the entire scheduling period.

1.5 Weekend Shift Optimization

Depending on the `schedulingPreference`, the algorithm may treat weekend shifts differently:

- If the preference is set to 'minimize_weekend', the optimization phase focuses on reducing the number of weekend shifts for each resident.
- This is achieved by attempting to swap weekend assignments if it improves the overall schedule score.
- While not a hard constraint, this optimization aims to create a more equitable distribution of the often less desirable weekend shifts.

1.6 Handling Constraint Conflicts

In situations where it's impossible to satisfy all constraints (e.g., no available residents for a particular day), the scheduler employs a fallback strategy:

- It selects the least constrained resident, potentially violating the `max1inX` constraint if necessary.
- This ensures that all shifts are filled, even in challenging scheduling scenarios.
- The algorithm prioritizes filling all shifts over strict adherence to the `max1inX` constraint in these rare cases.

These constraints work together to create a schedule that is fair, respects resident preferences and limitations, complies with work regulations, and maintains high-quality patient care by ensuring well-rested physicians.

2 Main Function: generateCallSchedule

The primary function exported by this module is `generateCallSchedule`, which takes the following parameters:

- `startDate`: A JavaScript Date object representing the inclusive start date of the scheduling period. This date marks the beginning of the time range for which the schedule will be generated.
- `endDate`: A JavaScript Date object representing the inclusive end date of the scheduling period. This date marks the end of the time range for which the schedule will be generated.
- `residents`: An array of resident objects. Each object in this array contains detailed information about a single resident, including their name, availability constraints, and scheduling preferences.
- `schedulingPreference`: A string indicating the preference for schedule optimization. For example, 'minimize_weekend' would indicate a preference for reducing weekend shifts. This parameter allows for customization of the scheduling strategy based on specific needs or policies.

3 Key Components and Data Structures

3.1 DateTime Handling

The module leverages the `luxon` library for precise and consistent date and time operations. Specifically:

- `DateTime` object: Used to represent and manipulate dates throughout the scheduling process. This object provides methods for date arithmetic, formatting, and comparison, ensuring accurate handling of time-related operations.
- Date Conversion: JavaScript Date objects (used as input parameters) are converted to luxon DateTime objects for internal processing, then converted back to strings for the final schedule output.

3.2 Schedule Representation

The generated schedule is represented as a JavaScript object with the following structure:

- Keys: Strings representing dates in the 'YYYY-MM-DD' format. Each key corresponds to a single day in the scheduling period.
- Values: Strings representing resident names. Each value indicates the resident assigned to the corresponding date.

For example:

```
{
'2024-07-01': 'Dr. Smith',
'2024-07-02': 'Dr. Johnson',
// ... more date-resident pairs
}
```

3.3 Resident Data Structure

Each resident in the `residents` array is represented by an object with the following properties:

- `name`: A string representing the resident's name. This is used as the identifier for the resident throughout the scheduling process.
- `max1inX`: An integer representing the minimum number of days that must pass between calls for this resident. For example, if `max1inX` is 4, the resident cannot be scheduled for a call more frequently than once every 4 days.
- `daysOff`: An array of date strings in the 'YYYY-MM-DD' format, representing days when the resident is unavailable for calls. This could include vacation days, conference attendance, or other pre-scheduled commitments.

Example of a resident object:

```
{
name: 'Dr. Smith',
max1inX: 4,
daysOff: ['2024-07-04', '2024-07-05']
}
```

3.4 Tracking Variables

The algorithm maintains several important tracking variables throughout the scheduling process:

- `minIntervals`: An object that maps resident names to their respective `max1inX` values. This allows for quick lookup of a resident's minimum interval between calls.
- `lastCall`: An object that maps resident names to their most recently assigned call date. This is used to ensure the `max1inX` constraint is respected and to calculate scores for resident selection.
- `totalShifts`: An object that maps resident names to the total number of shifts they have been assigned so far. This is used to maintain fairness in shift distribution and for score calculation.

Examples of these tracking variables:

```
minIntervals = { 'Dr. Smith': 4, 'Dr. Johnson': 3 }
lastCall = { 'Dr. Smith': '2024-07-01', 'Dr. Johnson': '2024-06-30' }
totalShifts = { 'Dr. Smith': 5, 'Dr. Johnson': 6 }
```

4 Scheduling Algorithm Overview

The scheduling algorithm consists of two main phases: Initial Schedule Fill and Schedule Optimization.

4.1 Initial Schedule Fill

This phase creates an initial schedule by assigning residents to each day in the scheduling period. The process is as follows:

1. Shuffle the `residents` array: This randomization ensures fairness in the initial assignment process, preventing bias towards residents based on their order in the array.
2. Iterate through each date in the scheduling period:
   - Use the `canTakeCall` function to filter available residents for the current date.
   - If there are available residents, use the `getResidentScore` function to select the best candidate.
   - If no residents are available (due to constraints), select the least constrained resident.
   - Assign the selected resident to the current date in the `schedule`.
   - Update the `lastCall` and `totalShifts` tracking variables for the selected resident.

4.2 Schedule Optimization

After the initial fill, this phase attempts to improve the schedule, focusing on weekend assignments:

1. Define weekend days based on the `schedulingPreference` parameter.
2. Iteratively attempt to improve weekend assignments:
   - For each weekend date, evaluate if swapping the assigned resident with another available resident would improve the overall score.
   - If an improvement is found, update the `schedule`, `lastCall`, and `totalShifts` accordingly.
   - Repeat this process until no further improvements can be made.

This optimization phase aims to create a more balanced and fair schedule, particularly for weekend shifts which are often considered less desirable.

5 Algorithm Output

The `generateCallSchedule` function returns a JavaScript object containing:

- `schedule`: The final generated schedule object, mapping dates to assigned residents.
- `totalShifts`: An object summarizing the total number of shifts assigned to each resident, providing a quick overview of shift distribution.

6 Key Functions

6.1 canTakeCall

Determines if a resident can take a call on a given date:

$$\text{canTakeCall}(r, d) = \begin{cases} \text{false} & \text{if } d \in r.\text{daysOff or resident on call previous day} \\ \text{true} & \text{if no previous call or interval } \geq \text{minInterval} \end{cases}$$

6.2 getResidentScore

Calculates a score for assigning a resident to a date:

$$\text{score} = \text{daysSinceLastCall} + 10 \left(\frac{\text{totalDays}}{\text{minInterval}} - \text{totalShifts} \right)$$

7 Algorithm Details

Algorithm 1 Generate Call Schedule

```
procedure GenerateCallSchedule(startDate, endDate, residents, schedulingPreference)
    Initialize schedule, lastCall, totalShifts
    dates ← GenerateDateRange(startDate, endDate)
    ShuffleResidents(residents)
    for each date in dates do
        availableResidents ← FilterAvailableResidents(residents, date)
        if availableResidents is not empty then
            selectedResident ← SelectBestResident(availableResidents, date)
        else
            selectedResident ← SelectLeastConstrainedResident(residents, date)
        end if
        AssignShift(schedule, selectedResident, date)
        UpdateTrackingVariables(lastCall, totalShifts, selectedResident, date)
    end for
    OptimizeWeekendAssignments(schedule, residents, schedulingPreference)
    return schedule, totalShifts
end procedure
```

Algorithm 2 Can Take Call

```
procedure CanTakeCall(resident, date)
    if date in resident.daysOff then
        return false
    end if
    if resident was on call previous day then
        return false
    end if
    lastCallDate ← GetLastCallDate(resident)
    if lastCallDate is null then
        return true
    end if
    interval ← (date - lastCallDate) in days
    return interval ≥ resident.minInterval
end procedure
```

Algorithm 3 Get Resident Score

```
procedure GetResidentScore(resident, date)
    lastCallDate ← GetLastCallDate(resident)
    if lastCallDate is null then
        daysSinceLastCall ← ∞
    else
        daysSinceLastCall ← (date - lastCallDate) in days
    end if
    expectedShifts ← totalDays / resident.minInterval
    shiftDeficit ← expectedShifts - resident.totalShifts
    score ← daysSinceLastCall + 10 * shiftDeficit
    return score
end procedure
```

Algorithm 4 Optimize Weekend Assignments

```
procedure OptimizeWeekendAssignments(schedule, residents, schedulingPreference)
    weekendDays ← GetWeekendDays(schedulingPreference)
    improved ← true
    while improved do
        improved ← false
        for each date in schedule do
            if date is in weekendDays then
                currentResident ← schedule[date]
                for each resident in residents do
                    if resident ≠ currentResident and CanTakeCall(resident, date) then
                        currentScore ← GetResidentScore(currentResident, date)
                        newScore ← GetResidentScore(resident, date)
                        if newScore ¿ currentScore then
                            SwapAssignment(schedule, date, resident)
                            UpdateTrackingVariables(lastCall, totalShifts, resident, date)
                            improved ← true
                        end if
                    end if
                end for
            end if
        end for
    end while
end procedure
```