Режимы алгоритмов составления расписания

Алгоритмы составления расписания можно использовать в двух режимах: итеративном и пакетном (batch-режим).

Итеративный режим.

Итеративным режимом назовём выполнение алгоритма многократно при появлении новых данных. В контексте данной задачи это означает, что каждый раз, когда новый преподаватель будет добавлять информацию о своих предпочтениях, алгоритм будет просчитывать возможные варианты расписания с учётом:
1) уже имеющихся данных, которые имеют приоритет, перед новыми
2) внесённых преподавателем изменений
Так, при каждом новом пересчёте уже составленные расписания преподавателей считаются утверждёнными, что даёт возможность не считать их заново, тем самым ускорив работу алгоритма.

Из преимуществ данного подхода можно выделить возможность преподавателя, не дожидаясь других, увидеть своё расписание. Также, с точки зрения организации, плюсом можно считать стимул преподавателей как можно раньше заполнить форму сбора информации, чтобы иметь приоритет перед другими. Таким образом появляется возможность раньше закончить процесс составление расписания.

Недостатки данного режима более существенны на практике, чем его преимущества, так как основным недостатком является закрепление наиболее удобного расписания за теми, кто заполнил форму сбора раньше остальных, что может повлечь коллизии. Можно рассмотреть случай, когда первый преподаватель, заполнивший форму сбора, выбирает большое окно для проведения экзаменов и алгоритм бронирует за ним несколько определённых случайных дат, а остальные оставляет свободными. Тогда второй преподаватель, уезжающий на конференцию в свободные даты и готовый провести экзамены в даты, занятые первым, уже не сможет это сделать, потому что у первого преподавателя приоритет, и его уже нельзя подвинуть на сводные даты.

Пакетный режим.

Пакетным режимом в данном случае будет единоразовая обработка всех полученных сведений. Для этого необходимо определить дедлайн для сбора пожеланий преподаватель, при наступлении которого составить предварительное расписание для всех за один вызов алгоритма.

Преимуществом такого подхода является возможность установить приоритеты преподавателей и групп, на основе которых можно решать коллизии в случае невозможности нахождения идеального решения, учитывающего пожелания каждого.

Недостатком такого подхода можно назвать невозможность до дедлайна получить расписание, но в реальности это не будет глобальным минусом, потому что составленное таким образом предварительное расписание, всё равно, в последствии может быть скорректировано вручную.

Описание алгоритма полного обхода графа

Среди исходных данных выделим два типа данных:
- Event - элемент, описывающий проведение аттестации определённого типа (зачёт/ экзамен/ консультация и т.п.) по определённому курсу у конкретной учебной группы конкретным преподавателем.
- DateTimeClass - тройка дата проведения, время начала, аудитория. Стоит оговорится, что у разных типов аттестаций разная продолжительность, поэтому примем

продолжительность элемента DateTimeClass за час. Таким образом, необходимо будет занимать несколько DataTimeClass подряд, чтобы провести, например, 3-часовой зачёт.

Задачу составления расписания можно свести к заполнению булевой матрицы вида S = {Event: 0 .. k, DateTimeClass: 0 .. n}. Значение Sij = true в данном случае будет означать, что аудитория в определённое время занята проведением аттестации i.

Также, задача имеет ряд ограничений, которые необходимо учесть при заполнении матрицы:

1. В одной аудитории в одно время может проводиться только одно событие.
2. В один день преподаватель не может работать больше, чем n часов (n = 6)
3. Группа может проходить не более n аттестаций в день, где n зависит от типа аттестации (зачёт/экзамен/консультация/…)
4. Перед и после аттестации группа должна иметь n дней отдыха, где n зависит от типа аттестации
5. Преподаватель может проводить аттестации только в выбранные дни (в идеальном исходе)
6. Преподаватель может проводить аттестации только в выбранное время (в идеальном исходе)
7. Тип аудитории для проведения аттестации должен совпадать (или быть лучше) с типом, необходимым для проведения конкретной аттестации. Рассмотрим пример. Пусть тип=1 соответствует компьютерной аудитории с проектором, тип=2 - компьютерной без проектора, тип=3 - обычной лекционной аудитории. Если для проведения аттестации преподаватель запросил аудиторию с компьютерами (тип=2), то условию удовлетворят тип <= 2.
8. Вместимость аудитории должна быть больше или равна количеству студентов в группе.

Стоит заметить, что выполнение первого условия влечёт за собой наличие в одном столбце матрице не более одного значения true, из-за чего матрицу можно заменить на целочисленный массив, где индексы соответствуют индексу возможного DataTimeClass, а значения - индексу события, которое будет там проведено.

Будем считать, что идеальное решение найдено, если в каждой строке матрицы есть значение true или каждое число от 0 до k встречается в массиве решения, что означает, что каждое событие можно сопоставить какому-то DataTimeClass с учётом перечисленных выше ограничений.

Таким образом, попробуем перебрать в худшем случае все перестановки событий по возможным датам, чтобы определить, существует ли идеальное решение, удовлетворяющее всем ограничениям.

Исходные данные:

```
Event[] events;
DateTimeClass[] dtc; //массив, отсортированный по аудиториям, датам, времени (именно в таком порядке)
int maxTimePerDay = 8;
```

Решение:

```
int[] solution;
```

Вспомогательные переменные:

```
int eventSize;
int dtcSize;
```

Метод определения возможности поиска идеального решения. Если решение найти возможно, то первое такое найденное решение, будет находиться в массиве solution.

```
boolean findSolution() {
    int event = 0;
```

```
    while (event < eventSize) {
        int time = this.getTime(event);
        int nextTime = this.findNextStartTime(event, time);
        if (nextTime > 0) {
            this.cleanEvent(event);
        }
        if (nextTime == -1) { //если не удалось найти другого подходящего DateTimeClass для
этого события
            if (event == 0) {
                return false; //если речь о первом событии, то уже были перебраны все остальные
варианты, и решения нет
            } else {
                event--; // иначе возвращаемся к предыдущему событию и пробуем изменить для
него DateTimeClass
            }
        } else {
            this.submitDateTimeClass(event, nextTime); //бронируем за событием время и
аудиторию
            event++; // переходим к следующему событию
        }
    }
    return true;
}
```

Для поиска возможного места и времени для определённого события с учётом всех ограничений используем метод, приведённый ниже. В случае, если событию невозможно найти свободную аудиторию, метод вернёт -1, а в противном случае - индекс начала DataTimeClass. Если метод вернёт -1, то необходимо вернуться к предыдущему рассмотренному событию и попытаться изменить его DataTimeClass.

```
private int findNextStartTime(int event, int time) {
    if (time < 0) return -1;

    int duration = events[event].type.duration;
    for (int i = time; i < dtcSize; i++) {
        DateTimeClass location = dtc[i];
        Event ev = events[event];
        Group group = ev.group;
        Teacher teacher = ev.teacher;
        boolean success = false;

        if (solution[i] == -1 // аудитория в это время свободна
                && ev.wishedClassroomType <= location.classroom.type //есть ли в аудитории
нужное оборудование
                && group.size <= location.classroom.size //влезает ли группа в аудиторию
                // не слишком ли сейчас поздно для начала проведения длительного события
                && location.classroom.num == dtc[i + duration].classroom.num
                && location.date.equals(dtc[i + duration].date)
                // подходит ли дата и время преподавателю
                && Arrays.stream(Arrays.stream(teacher.date).toArray()).anyMatch(dt ->
dt.equals(location.date))
                && Arrays.stream(Arrays.stream(teacher.time).toArray()).anyMatch(t -> t == location.time)
                && Arrays.stream(Arrays.stream(teacher.time).toArray()).anyMatch(t -> t == location.time
+ duration)
        ) {
            int teacherTime = 0;
            int countPerDay = 0;
            for (int k = 0; k < dtcSize; k++) {
                // если время-место свободны, они не повлияют на ограничения
                if (solution[k] == -1) continue;
```

```
            Event currentEvent = events[solution[k]];
            DateTimeClass currentLocation = dtc[k];

            // если проверяемая аудитория в это время занята
            if (location.date.compareTo(currentLocation.date) == 0) {
                break;
            }
            // если у преподавателя в этот день уже были занятия
            if (currentEvent.teacher.name.equals(teacher.name) && currentLocation.date ==
location.date
            ) {
                teacherTime++;
                // если преподаватель уже достаточно отработал в этот день.
                if (teacherTime > maxTimePerDay - duration) {
                    break;
                }
            }
            if (currentEvent.group.group.equals(group.group)) {
                //если в этот день уже были занятия у этой группы
                if (currentEvent.type.type.equals(ev.type.type) && location.date ==
currentLocation.date) {
                    countPerDay++;
                    if (ev.type.countPerDay < countPerDay) {
                        break;
                    }
                }
                //если до или после события есть другое событие, до которого меньше
"отдыха", чем нужно
                if (location.date.minusDays(ev.type.pauseBefore).compareTo(currentLocation.date) <=
0
                        && location.date.plusDays(ev.type.pauseAfter).compareTo(currentLocation.date)
>= 0) {
                    break;
                }

            }
            if (k == dtcSize - 1) success = true;
        }
    }
    if (success) return i;
  }
  return -1;
}
```

Бронирование происходит заполнением нужных элементов массива-решения индексом текущего события:

```
private void submitDateTimeClass(int event, int nextTime) {
  int duration = events[event].type.duration;
  for (int i = nextTime; i < nextTime + duration; i++) {
    solution[i] = event;
  }
}
```

При изменении брони для события, необходимо удалить из массива все значения, соответсвующие индексу этого события и заменить их на -1:

```
private void cleanEvent(int event) {
  for (int i = 0; i < dtcSize; i++) {
```

```
        if (solution[i] == event) solution[i] = -1;
    }
}
```

Оценка сложности алгоритма

Сложность описанного выше алгоритма составляет O(n*k*logN), так как в худшем случае для k событий придётся проверить n аудиторий в определённое время, для проверки валидности которых каждый раз нужно обходить остальные варианты аудиторий по времени. Логарифм N, потому что при обходе остальных аудиторий на ранних этапах заканчивается на первых итерациях, а с накоплением забронированных аудиторий, требует обходить всё больше вариантов.

Учёт приоритетов преподавателей

На практике возникают ситуации, когда недостаток аудиторий и накладки в личных расписаниях преподавателей не позволяют найти такое расписание, которое удовлетворит всем пожеланиям. По этой причине приходится учитывать приоритеты ограничений и минимизировать потери.

Существует ряд ограничений, поступиться с которыми нельзя. В их число входят правила проведения аттестаций и физические условия, связанные с проведением экзаменов в аудиториях. Но есть и менее жёсткие факторы - это пожелания преподавателей к датам и времени экзаменов.

Каждый преподаватель указывает дни и время, когда ему было бы удобно присутствовать на аттестации. В некоторых случаях удобство эквивалентно тому, что в остальные дни преподаватель в принципе не может принимать экзамены. Приоритет таких ограничений должен быть выше, чем у случаев, когда преподавателю просто комфортнее было бы присутствовать на экзаменах в определённые дни.

Таким образом, введём множество приоритетов
P:{0, 1, 2}, где
0 - пожелания обусловлены только комфортом,
1 - учёт пожеланий крайне желателен, но только в крайнем случае, подлежит обсуждениям
2 - пожелания, учёт которых обязателен без обсуждений (актуально для совместителей, утвердивших поездку в командировку и т.п.)

Система приоритетов в случае появления коллизий при составлении расписания позволит в первую очередь пытаться нарушить только пожелания с наименьшим приоритетом и только, если это необходимо, с более высокими.

Модернизируем алгоритм поиска «идеального» решения:

Для этого нужно добавить массив boolean-ов, в котором будут фиксироваться те события, для которых приходится искать решения без учёта предпочтений преподавателей.

```
boolean[] ignoreWishes = = new boolean[eventSize];
```

Также, необходимо отсортировать массив событий по приоритетам преподавателей, чтобы искать решения без учёта высокоприоритетных преподавателей в последнюю очередь.

```
Arrays.sort(events);
```

Метод определения возможности поиска решения теперь содержит условный оператор, который в случае определения того, что идеального решения найти не удаётся, переходит в режим игнорирования пожеланий преподавателя для конкретного события.

```
boolean findSolution() {
    int event = 0;
    while (event < eventSize) {
        int time = this.getTime(event);
        int nextTime;
        this.cleanEvent(event);
        nextTime = this.findNextStartTime(event, time);

        //переходим в режим игнорирования пожеланий преподавателя, если не смогли найти
идеального решения
        if(!ignoreWishes[event] && nextTime == -1){
            ignoreWishes[event] = true;
            nextTime = this.findNextStartTime(event, 0);
        }

        if (nextTime == -1) { //если не удалось найти другого подходящего DateTimeClass для
этого события
            if (event == 0) {
                return false; //если речь о первом событии, то уже были перебраны все остальные
варинты, и решения нет
            } else {
                ignoreWishes[event] = false;
                event--; // иначе возвращаемся к предыдущему событию и пробуем изменить для
него DateTimeClass
            }
        } else {
            this.submitDateTimeClass(event, nextTime); //бронируем за событием время и
аудиторию
            event++; // переходим к следующему событию
        }
    }
    return true;
}
```

Метод поиска следующего подходящего времени и аудитории теперь может работать в двух режимах. В режими игнорирования ищутся только те DateTimeClass, которые хотя бы в какой-то мере не удовлетворяют пожеланиям преподавателя.

```
private int findNextStartTime(int event, int time) {
    if (time < 0) return -1;

    int duration = events[event].type.duration;
    for (int i = time; i < dtcSize; i++) {
        DateTimeClass location = dtc[i];
        Event ev = events[event];
        Group group = ev.group;
        Teacher teacher = ev.teacher;
        boolean success = false;

        boolean teacherWishes = Arrays.stream(Arrays.stream(teacher.date).toArray()).anyMatch(dt
-> dt.equals(location.date))
                && Arrays.stream(Arrays.stream(teacher.time).toArray()).anyMatch(t -> t == location.time)
                && Arrays.stream(Arrays.stream(teacher.time).toArray()).anyMatch(t -> t == location.time
+ duration);
        if (ignoreWishes[event]) {
            teacherWishes = !teacherWishes;
        }
        if (solution[i] == -1 // аудитория в это время свободна
                && ev.wishedClassroomType <= location.classroom.type //есть ли в аудитории
нужное оборудование
                && group.size <= location.classroom.size //влезает ли группа в аудиторию
```

```
            // не слишком ли сейчас поздно для начала проведения длительного события
            && location.classroom.num == dtc[i + duration].classroom.num
            && location.date.equals(dtc[i + duration].date)
            // подходит ли дата и время преподавателю
            && teacherWishes
        ) {
            int teacherTime = 0;
            int countPerDay = 0;
            for (int k = 0; k < dtcSize; k++) {
                // если время-место свободны, они не повлияют на ограничения
                if (solution[k] == -1) continue;

                Event currentEvent = events[solution[k]];
                DateTimeClass currentLocation = dtc[k];

                // если проверяемая аудитория в это время занята
                if (location.date.compareTo(currentLocation.date) == 0) {
                    break;
                }
                // если у преподавателя в этот день уже были занятия
                if (currentEvent.teacher.name.equals(teacher.name) && currentLocation.date ==
location.date
                ) {
                    teacherTime++;
                    // если преподаватель уже достаточно отработал в этот день.
                    if (teacherTime > maxTimePerDay - duration) {
                        break;
                    }
                }
                if (currentEvent.group.group.equals(group.group)) {
                    //если в этот день уже были занятия у этой группы
                    if (currentEvent.type.type.equals(ev.type.type) && location.date ==
currentLocation.date) {
                        countPerDay++;
                        if (ev.type.countPerDay < countPerDay) {
                            break;
                        }
                    }
                    //если до или после события есть другое событие, до которого меньше
"отдыха", чем нужно
                    if (location.date.minusDays(ev.type.pauseBefore).compareTo(currentLocation.date) <=
0
                            && location.date.plusDays(ev.type.pauseAfter).compareTo(currentLocation.date)
>= 0) {
                        break;
                    }

                }
                if (k == dtcSize - 1) success = true;
            }
        }
        if (success) return i;
    }
    return -1;
}
```

В результате имеется алгоритм, который итеративно обходит все возможные варианты размещения аттестаций по аудиториям, сначала пытаясь найти идеальный вариант, подходящий всем. В случае неудачи, начиная с наименее приоритетных преподавателей он

начинает игнорировать приодеты, чтобы с меньшими затратами найти наиболее подходящее расписание.

Использование генетического алгоритма для оптимизации составления расписания с ограничениями.

Определим основные понятия генетического алгоритма для простого случая составления расписания без ограничений.

Ген будет представлять собой объект с полями: Группа, Преподаватель, Предмет, Тип аттестации, Дата, Время, Аудитория.

Популяция - все возможные комбинации расположения аттестаций по аудиториям во времени.

Особь - некоторый набор генов (пример расписания)

Такая задача сводится к нахождению наилучшей особи - расписания.

Исходные данные при таком условии - 4 датасета:
Events (Аттестации) - с полями Группа, Преподаватель, Предмет, Тип аттестации
Dates (даты сессии) - с полем Дата
Time (время сессии) - с полем Время
Classrooms (Аудитории) - с полем Аудитория

Для получения популяции необходимо взять декартово произведение всех четырёх датасетов.

Фитнес-функция в данном случае должна всего-лишь давать очки за каждую уникальную аттестацию в особи и снимать их за накладки во времени и аудиториях. Тогда самой сильной особью будет расписание со всеми необходимыми событиями, проходящими в разных аудиториях в разное время.

Но в данном случае необходимо учитывать описанные ранее ограничения, из-за чего исходные данные пополнятся следующими датасетами:

Events (Аттестации) - с полями Группа, Преподаватель, Предмет, Тип аттестации, Тип аудитории, Размер группы, Приоритет преподавателя
Classrooms (Аудитории) - с полеми Аудитория, Тип аудитории, Вместимость
AttestationType (Тип аттестации) - с полями Тип аудитории, Отдых перед, Отдых после, Максимальное число в день, Продолжительность
TeacherDates(Даты, выбранные преподавателем) - Преподаватель, Дата
TeacherTime(Время, выбранное преподавателем) - Преподаватель, Время

Теперь для получения популяции нужно выполнить соединить эти датасеты следующим образом:

```
Events CROSS JOIN Dates CROSS JOIN Time
 -- Только аудитории нужного типа и размера
INNER JOIN Classrooms
        ON (Classrooms.«Тип аудитории» <= Events.«Тип аудитории»  AND
Classrooms.Вместимость >= Events.«Размер группы»)
 -- Присоединяем информацию о типе аттестации
INNER JOIN AttestationType USING («Тип аудитории»)
```

Фитнесс-функции при таких входных условиях добавляет очки в следующих случаях:
1) Наличие уникальных аттестаций.
2) Присутствие пары хромосом Преподаватель - Дата в датасете TeacherDates. Начисление очков растёт с приоритетом преподавателя.

3) Присутствие пары хромосом Преподаватель - Время в датасете TeacherTime. Начисление очков растёт с приоритетом преподавателя.
Снимать очки особям нужно за:
1) Наличие пересечения хромосом Аудитория, Дата, Время +- Продолжительность
2) Наличие в особи генов с одинаковой группой, имеющие пересекающиеся хромосомы (Дата - Отдых перед) и Дата или (Дата + Отдых после) и Дата
3) Наличие в особи генов с количеством одинаковых хромосом Группа, Дата большим, чем значение хромосомы Максимальное число в день.

Применимость генетического алгоритма для оптимизации составления расписания с ограничениями.

Из преимуществ генетического алгоритма можно выделить:
1) Адаптивность времени выполнения.
Можно ограничить количество поколений алгоритма, по истечении которых алгоритм оставит наиболее подходящее расписание.
2) Адаптивность качества.
Если задать условием остановки алгоритма - достижение некоторого счёта в фитнес-функции, можно завершить работу с приемлемыми потерями, полученными от пренебрежения пожеланиями преподавателями, например.

Недостатками генетического алгоритма для задачи составления расписания будут:
1) Массивные входные данные.
Для формирования популяции необходимо брать декартово произведение нескольких датасетов и манипулировать большой начальной популяцией.
2) Сложность подбора параметров фитнес-функции.
На практике необходимо уметь выделять более строгие и менее строгие ограничения. Это необходимо корректно отразить в счёте обучающей функции.
3) Длительная отладка.
Побор параметров генетического алгоритма подразумевает многократный прогон его на данных, близких к реальным. С учётом специфики задачи работа одного запуска может занять продолжительное время.