Постановка задачи

Для составления расписания учебной сессии в университете необходимо каждой запланированной аттестации подобрать день, время и аудиторию проведения. Известны доступные аудитории и множество аттестаций, которые необходимо провести в рамках сессии. Для каждой аудитории известно время ее доступности. Для каждого типа аттестации определена длительность, максимальное количество в день и количество дней отдыха до и после её проведения. Нужно составить расписание сессии так, чтобы общая эффективность расписания была наибольшей.

При составлении расписании также необходимо учитывать ряд ограничений:

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.

Данная задача относится к классу NP-полных задач, и для её решения рассмотрим алгоритмы динамического и линейного программирования, алгоритм на графах, эволюционный алгоритм и метод численной оптимизации.

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.

Применение динамического программирования для составления расписания.

Алгоритмы в парадигме динамического программирования основаны на разбиении сложной задачи на более простые.

В данном случае для составления расписания сложной задачей является для всех аттестаций назвать возможные расстановки их во времени по аудиториям. Задачей проще будет поиск возможных распределений i-ей аттестации во времени по кабинетам, если известно, какими способами можно расставить i-1 других аттестаций.

Составим матрицу решений вида:

	T1	T2	T3	...	Tn
E1	{T1}	{T2}	-	...	{Tn}
E2	{Tn,T1}	{T1,T2};{Tn,T2}	{T1,T3}	...	{T1,Tn}
...	...	...	...	...	...
Ek	{Tn,T2,...,T1};	-	{T1,T2,...,T3};{Tn,T2,...,T3}	...	-

В столбцах - список запланированных аттестаций E1..Ek, в строках - возможные окна для их проведения T1..Tn. В ячейке матрицы xij для аттестаций Ei записываем расписания для аттестаций E1..Ei-1, которые допускают проведение Ei в окно Tj.

Чтобы получить полные возможные расписания, необходимо из последней строки выбрать все непустые массивы. Каждый из них будет содержать k значений, где на месте i будет записано окно для проведения аттестации Ei.

Рассмотрим алгоритм заполнения такой матрицы.
В методе findSolutions() построчно обходятся строки матрицы и заполняются массивами возможных расписаний.

```
public List<List<Integer>> findSolutions() {
    //заполнение матрицы решений
    for (int i = 0; i < dtcSize; i++) {
        if (simpleCheck(0, i)) {
            solutionsMatrix[0][i] = new ArrayList<>();
            solutionsMatrix[0][i].add(Collections.singletonList(i));
        }
    }
    int nullCount = 0;
    for (int i = 1; i < eventSize; i++) {
        for (int j = 0; j < dtcSize; j++) {
            // если сама аудитория не подходит не подходит для проведения аттестации, нет
смысла проверять её загруженность
            if (simpleCheck(i, j)) {
                for (int k = 0; k < dtcSize; k++) {
                    if (solutionsMatrix[i][j - 1] != null) {
                        for (List<Integer> path : solutionsMatrix[i][j - 1]) {
                            if (pathsCheck(i, j, path)) {
                                if (solutionsMatrix[i][j] == null) {
                                    solutionsMatrix[i][j] = new ArrayList<>();
                                }
                                List<Integer> newPath = new ArrayList<>(path);
                                newPath.add(j);
                                solutionsMatrix[i][j].add(newPath);
                            }
                        }
                    }
                }
            }
            /* Если во время попытки разместить какую-то аттестацию выясняется, что её не
возможно поставить,
            сразу возвращаем null, чтобы не обходить зря следующие аттестации
             */
            if (solutionsMatrix[i][j] == null) {
                nullCount++;
            }
            if (nullCount == dtcSize) {
                return null;
            }
        }
    }
    // собираем все возможные решения расстановки всех событий в один массив
    solutions = new ArrayList<>();
    for (int i = 0; i < dtcSize; i++) {
        if (solutionsMatrix[eventSize - 1][i] != null) {
            solutions.addAll(solutionsMatrix[eventSize - 1][i]);
        }
    }
    return solutions;
}
```

Для первого события E1 возможность проведения в окно Tj обуславливается только пожеланиями преподавателя к дате и времени и пригодностью аудитории. Такого типа проверки проводятся в методе simpleCheck(int i, int j). В нём происходит проверка условий, которые не завязаны на уже существующем расписании.

```
private boolean simpleCheck(int i, int j) {
    Event ev = events[i];
    DateTimeClass location = dtc[j];
```

```
    Teacher teacher = ev.teacher;
    Group group = ev.group;
    int duration = ev.type.duration;

    return 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);
}
```

Для последующих аттестаций необходимо ориентироваться на расписания, состоящие из предыдущих событий.

Для аттестации Ei при записи в столбец j в случае, когда окно Tj не удовлетворяет пожеланиям преподавателя или не подходит типом аудитории, сразу оставляем null (прочерк), а в противном случае обходим только непустые массивы решений из предыдущей строки, выбираем те, для которых выполняются условия:

- массив не содержит Tj
- преподаватель в этот день не отработал максимальное для себя количество часов
- у группы не будет превышено максимальное число аттестаций этого типа в день j
- дата ни одного экзамена из массива не попадает в интервал отдыха до и после проведения аттестации этого типа

На i-ую позицию подходящих массивов дописываем Tj.

Проверки такого рода проводятся в методе pathsCheck(int i, int j, List<Integer> path) и в него необходимо передавать расписание, которое path нужно проверить на возможность добавления в него новой аттестации.

```
private boolean pathsCheck(int i, int j, List<Integer> path) {
    Event ev = events[i];
    DateTimeClass dateTimeClass = dtc[j];
    Teacher teacher = ev.teacher;
    Group group = ev.group;
    LocalDate pauseStart = dateTimeClass.date.minusDays(ev.type.pauseBefore);
    LocalDate pauseEnd = dateTimeClass.date.plusDays(ev.type.pauseAfter);
    int duration = ev.type.duration;

    int teacherTimeToday = ev.type.duration;
    int groupsEventsToday = 1;
    boolean success = true;
    for (int k = 0; k < path.size(); k++) {
        DateTimeClass d = dtc[path.get(k)];
        Event e = events[k];
        if (d.date == dateTimeClass.date) {
            if (teacher.name.equals(e.teacher.name)) {
                teacherTimeToday += e.type.duration;
            }
            if (group.group.equals(e.group.group)) {
                groupsEventsToday++;
            }
        }
        if (
```

```
            // если одна из предыдущих аттестаций попадает в окно отдыха до и после текущей
аттестации
                (d.date == dateTimeClass.date) && ((d.time + e.type.duration > dateTimeClass.time) ||
(dateTimeClass.time + duration > d.time))
                        //если в этот день есть аттестации, которые по времени накладываются с
текущей
                        || (d.date.compareTo(pauseStart) >= 0) && (d.date.compareTo(pauseEnd) <= 0)
                        // если преподаватель переработал максимум рабочих часов в день
                        || (teacherTimeToday > maxTimePerDay)
                        // если у группы больше аттестаций этого типа, чем положено
                        || (groupsEventsToday > ev.type.countPerDay)
        ) {
            success = false;
            break;
        }
    }
    return success;
}
```

Таким образом, на каждом шаге динамического алгоритма проверяются только заведомо возможные расстановки аттестаций, что существенно снижает его сложность.

Алгоритмы на графах. Обход графа в глубину.

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 построению графа, у которого в качестве вершин графа примем аттестации (Event), а в качестве рёбер - временные окна (DateTimeClass). Решением будет являться связный граф состоящий из всех вершин множества Event.

Стратегия поиска решения в глубину, состоит в том, чтобы идти максимально далеко уходить вглубь графа, насколько это возможно. Возврат происходит только в том случае, если в текущей вершине не осталось рёбер, которые ведут в следующую нерассмотренную вершину.

Заполним булеву матрицы вида S = {Event: 0 .. k, DateTimeClass: 0 .. n}. Значение Sij = true в данном случае будет означать, что к i-му узлу ведёт ребро j, то есть аудитория в определённое время занята проведением аттестации i.

Важно помнить, что значение true в ячейке может быть проставлено, только при соблюдении ограничений, описанных в пункте Постановка задачи.

Стоит заметить, что выполнение ограничения, что в одной аудитории в одно время может проводиться только одно событие, влечёт за собой наличие в одном столбце матрицы 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. Такое решение добавляется в список всех идеальных решений.

Исходные данные:

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

Решения:

```
List<int[]> solutions;
```

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.

```
void findSolutions() {
    int event = 0;
    while (event < eventSize) {
        int time = this.getTime(event);
        this.cleanEvent(event);
        int nextTime = this.findNextStartTime(event, time);
        if (nextTime == -1) { //если не удалось найти другого подходящего DateTimeClass для этого события
            if (event == 0) {
                return; //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
        }
        // если решение найдено, добавляем его в список решений и продолжаем искать решения
        if (event == eventSize){
            solutions.add(solution.clone());
            event--;
        }
    }
}
```

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;
    }
}
```

Обход в глубину с учётом 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(this.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 приоритеты, чтобы с меньшими затратами найти наиболее подходящее расписание.

Использование генетического алгоритма для поиска решений.

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 поиска решений 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) Нелинейная зависимость качества от длительности выполнения.
Часто в эволюционных алгоритмах последующее поколение бывает менее приспособлено, чем предыдущее, а значит, заранее предугадать количество итераций при заданном минимальном пороге качества нельзя.
3) Сложность подбора параметров фитнес-функции.
На практике необходимо уметь выделять более строгие и менее строгие ограничения. Это необходимо корректно отразить в счёте обучающей функции.
4) 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.

Метрики выбора лучшего решения.

Для оценки предложенных расписаний подсчитаем несколько метрик:
1) Длительность сессии для каждого преподавателя с учётом его приоритета
D = sum[1..teachersCount](prior*(lastEvent - firstEvent))

Считаем, что расписание преподавателя должно быть максимально компактным по датам, а значит разницу между первым и последним днём проведения экзаменов нужно минимизировать.

2) Суммарная длительность минимального отдыха между экзаменами по группам
P = sum[1..groupsCount](min(event - prevEvent))
Считаем, что студентам желательно не иметь маленьких перерывов между экзаменами, поэтому стремимся максимизировать P для расписания.

3) Суммарная длительность сессии по группам
S = sum[1..groupsCount](lastDate)

Чем раньше закончится сессия, тем раньше у иногородних студентов появится возможность уехать домой, а значит порядковый номер последнего дня сессии для группы нужно пытаться минимизировать.

4) Суммарное кол-во рабочих дней для преподавателей с учётом их приоритетов
W = sum[1..teachersCount](prior*(workingDaysCount))

Считаем, что преподавателю удобно иметь максимальное количество дней, свободных от проведения экзаменов, а значит метрику W стараемся минимизировать. На практике это означает, что лучше провести несколько зачётов в один день, чем заставлять человека несколько раз в неделю приезжать в университет.

Метод getRates(int[] sol) позволит рассчитать все описанные выше метрики для конкретного расписания и вернёт их в качестве одномерного массива длины 4.

```
/*
возвращает массив метрик
индекс 0 - Длительность сессии для каждого преподавателя с учётом его приоритета
индекс 1 - Суммарное кол-во рабочих дней для преподавателей с учётом их приоритетов
индекс 2 - Суммарная длительность сессии по группам
индекс 3 - Суммарная длительность минимального отдыха между экзаменами по группам
 */
public int[] getRates(int[] sol) {
    Map<Teacher, StartEnd> mapTeacherDuration = new HashMap<>();
    Map<Teacher, Set<LocalDate>> mapTeacherWorkingDays = new HashMap<>();
    Map<Group, LocalDate> mapGroupDuration = new HashMap<>();
    Map<Group, PauseDate> mapGroupsPause = new HashMap<>();
    for (int i = 0; i < eventSize; i++) {
        if (sol[i] != -1) {
            Event e = events[sol[i]];
            if (!mapTeacherDuration.containsKey(e.teacher)) {
                mapTeacherDuration.put(e.teacher, new StartEnd(dtc[i].date));
            } else {
                LocalDate start = mapTeacherDuration.get(e.teacher).start;
                mapTeacherDuration.put(e.teacher, new StartEnd(start, dtc[i].date));
            }

            mapGroupDuration.put(e.group, dtc[i].date);

            if (!mapGroupsPause.containsKey(e.group)) {
                mapGroupsPause.put(e.group, new PauseDate(dtc[i].date));
            } else {
                LocalDate date = mapGroupsPause.get(e.group).date;
                long pause = ChronoUnit.DAYS.between(date, dtc[i].date);
                if (pause > 0) {
                    long minPause = mapGroupsPause.get(e.group).pause;
                    if (pause < minPause) {
                        mapGroupsPause.put(e.group, new PauseDate(dtc[i].date, pause));
                    } else {
                        mapGroupsPause.put(e.group, new PauseDate(dtc[i].date, minPause));
                    }
```

```
                }
            }
            Set<LocalDate> set;
            if (mapTeacherWorkingDays.containsKey(e.teacher)) {
                set = mapTeacherWorkingDays.get(e.teacher);
            } else {
                set = new HashSet<>();
            }
            set.add(dtc[i].date);
            mapTeacherWorkingDays.put(e.teacher, set);
        }
    }
    AtomicInteger teacherDurationRate = new AtomicInteger();
    mapTeacherDuration.forEach((t, se) -> teacherDurationRate.addAndGet((int) (t.prior *
(ChronoUnit.DAYS.between(se.start, se.end)))));

    AtomicInteger groupDurationRate = new AtomicInteger();
    mapGroupDuration.forEach((gr, se) -> groupDurationRate.addAndGet(se.getYear() * 365 +
se.getDayOfMonth()));

    AtomicInteger groupsPauseRate = new AtomicInteger();
    mapGroupsPause.forEach((gr, se) -> groupsPauseRate.addAndGet((int) se.pause));

    AtomicInteger teacherWorkingDaysRate = new AtomicInteger();
    mapTeacherWorkingDays.forEach((t, d) -> teacherWorkingDaysRate.addAndGet(t.prior *
d.size()));
    return new int[]{
            teacherDurationRate.get(),
            teacherWorkingDaysRate.get(),
            groupDurationRate.get(),
            groupsPauseRate.get()
    };
}
```

Все эти метрики имеют разные шкалы, какие-то мы пытаемся максимизировать, какие-то минимизировать, одни метрики должны вносить более весомый, а другие - менее весомый вклад, поэтому каждую метрику для всех рассматриваемых решений необходимо нормализовать. Это приведёт данные метрики к единой шкале, улучшит их интерпретируемость.

Применим минимаксную нормализацию: $X' = (X - X_{min})/(X_{max} - X_{min})$

Далее сделаем так, чтобы большему значению каждой метрики соответствовало более предпочтительное расписание. Для этого те метрики, для которых лучшее решение имеет меньшее значение - вычтем из 1. Тем самым соотношение всех значений метрики среди расписаний останется таким же, но качество метрики будет расти с качеством решения.

Приведение метрик к единому виду происходит в методе findSolutionsRates() по описанному выше принципу.

```
public float[][] findSolutionsRates() {
    float[] min = new float[4];
    float[] max = new float[4];

    float[][] normalRates = new float[4][solutions.size()];

    // записываем в массив метрик расписаний и для каждой метрики ищем максимум и
минимум
    for (int i = 0; i < solutions.size(); i++) {
        int[] sol = solutions.get(i);
```

```
        int[] rates = getRates(sol);
        for (int j = 0; j < 4; j++) {
            if (min[j] < 0 || min[j] > rates[j]) min[j] = rates[j];
            if (max[j] < 0 || max[j] < rates[j]) max[j] = rates[j];
            normalRates[j][i] = rates[j];
        }
    }
    //минимаксная нормализация
    float[] rates = new float[solutions.size()];
    for (int i = 0; i < solutions.size(); i++) {
        for (int j = 0; j < 3; j++) {
            normalRates[j][i] = 1 - (normalRates[j][i] - min[j]) / (max[j] - min[j]);
        }
        normalRates[3][i] = (normalRates[3][i] - min[3]) / (max[3] - min[3]);
    }
    return normalRates;
}
```

Целочисленное линейное программирование. Метод ветвей и границ.

Метод ветвей и границ применяется для оптимизации задач полного перебора и как нельзя кстати придётся для модернизации алгоритма полного обхода графа в глубину.
Суть этого метода состоит в том, чтобы в процессе обхода графа, отбрасывать заведомо менее оптимальные решения, чем у же найденные.

Такой подход потребует выделить память под хранение метрик качества - массивы с метриками текущего расписания и лучшего расписания на данный момент:

```
long[] metrics;
long[] bestMetrics;
```

В качестве метрики качества возьмём разброс между первым и последним днём проведения сессии для каждого преподавателя. Чем он меньше, тем лучше считается расписание. Естественно, можно выбрать и другие критерии качества расписания, но алгоритм при этом поменяется незначительно.

Модернизируем функцию поиска решений методом обхода графа в глубину findSolutions(), добавив в качестве условия перехода к следующей аттестации проверку на то, является ли расписание на данном этапе, худшим, чем предыдущее полностью составленное расписание.

```
void findSolutions() {
    int event = 0;
    while (event < eventSize) {
        int time = this.getTime(event);
        this.cleanEvent(event);
        int nextTime = this.findNextStartTime(event, time);
        if (nextTime == -1) { //если не удалось найти другого подходящего DateTimeClass для
этого события
            if (event == 0) {
                return; //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
```

```
            // если решение уже хуже, чем лучшее, можно не продолжать его строить
            if (event == 0 || !isItWorse(event)) {
                event++; // переходим к следующему событию
            }
        }
        // если решение найдено, добавляем его в список решений и продолжаем искать
другие
        if (event == eventSize) {
            solutions.add(solution.clone());
            // теперь метрики этого решения считаются лучшими
            if (bestMetrics[0] == -1) {
                bestMetrics = metrics.clone();
            }
            event--;
        }
    }
}
```

Функция isItWorse(int event) пересчитывает для рассматриваемого преподавателя длительность его сессии и проверяет, не является ли оно худшим, чем у того же преподавателя в предыдущем полном расписании.

```
private boolean isItWorse(int event) {
    Event e = events[event];
    int i = event - 1;
    List<Integer> list = Arrays.asList(solution);
    int ind = list.indexOf(event);
    LocalDate min = dtc[ind].date;
    LocalDate max = min;
    //для рассматриваемого преподавателя пересчитываем длительность его сессии
    while (i >= 0 && events[i].teacher.name.equals(e.teacher.name)) {
        LocalDate dt = dtc[list.indexOf(i)].date;
        if (dt.compareTo(max) > 0) {
            max = dt;
        } else if (dt.compareTo(min) < 0) {
            min = dt;
        }
        i--;
    }
    long diff = ChronoUnit.DAYS.between(min, max);
    metrics[event] = diff;
    int t = teachers.indexOf(e.teacher);
    // если это не первое решение и длительнольность сессии преподавателя уже больше,
чем в прошлом решении, считаем, что это хуже
    return bestMetrics[0] != -1 && bestMetrics[t] < metrics[t];
}
```

Преимуществом данного алгоритма относительно полного перебора решений является его большая скорость выполнения за счёт отсечения менее пригодных вариантов расписания ещё до полного их составления. Но минусом такой оптимизации может стать неудачный выбор метрики оценки качества, из-за чего хорошие расписания могут не попасть в ответ.

Численные методы оптимизации. Алгоритм имитации отжига.

Для выбора лучшего из расписаний и расчёта всех метрик необходимо будет дополнительно обойти x расписаний, состоящих из n возможных временных окон. А потом просуммировать их по p преподавателям и z учебным группам.

Если входные данные были такими, что было найдено 5-10 возможных расписаний, то не составит труда просчитать их все, так как относительно n (временные окна по аудиториям), оно вносит минимальный вклад в сложность вычислений.

Но рассмотрим случай, когда методом полного перебора было найдено большое количество возможных расписаний, и число x достаточно велико и даже сравнимо с n. В такой ситуации было бы полезно находить более оптимальные решения без расчёта метрик для каждого расписания, жертвуя некоторой погрешностью.

Для этого можно применить оптимизационный алгоритм имитации отжига. Он основан на имитации физического процесса отжига металлов - при постепенно понижающейся температуре переход атома из одной ячейки кристаллической решётки в другую происходит с некоторой вероятностью, которая понижается с понижением температуры.

При помощи моделирования такого процесса ищется такая точка или множество точек, на котором достигается минимум некоторой числовой функции F(x), где x принадлежит множеству возможных решений.

Для задачи выбора лучшего расписания функция F(x) - функция, вычисляющая качество расписания, например, по метрике S (суммарная продолжительность сессии для каждой группы) из раздела Метрики выбора лучшего решения, которая рассчитывается в методе getRate(int i), где i - индекс расписания в массиве решений.

```
public int getRate(int i) {
    if (metrics[i]!=0) return metrics[i];
    int[] sol = solutions.get(i);
    Map<Group, LocalDate> mapGroupDuration = new HashMap<>();

    for (int j = 0; j < eventSize; j++) {
        if (sol[j] != -1) {
            mapGroupDuration.put(events[sol[j]].group, dtc[j].date);
        }
    }
    AtomicInteger groupDurationRate = new AtomicInteger();
    mapGroupDuration.forEach((gr, se) -> groupDurationRate.addAndGet(se.getYear() * 365 +
se.getDayOfMonth()));
    metrics[i] = groupDurationRate.get();
    return groupDurationRate.get();
}
```

Решение ищется последовательным вычислением точек x0..xn пространства X, где n - количество итераций понижения температуры. Каждая последующая точка претендует на то, чтобы лучше предыдущих приближать решение. В качестве исходных данных принимается точка x0. На каждом шаге алгоритм вычисляет новую точку и понижает значение счётчика - температуры Qi, и когда он достигает нуля, алгоритм останавливается в этой точке.

Температурой для данной задачи возьмём некоторую константу, достаточно большую, чтобы хватило итераций для нахождения примерного оптимума, но при этом меньшую, чем x. Пусть это будет константа Q0 = n/10.

X0..Xn для задачи выбора оптимального расписания - возможные варианты расписаний.

К точке Xi на каждом шаге применяется оператор A, который случайным образом модифицирует её, в результате чего получается новая точка X*. Но перейдёт ли X* в Xi+1

$$P(\overline{x^*} \to \overline{x_{i+1}} \mid \overline{x_i}) = \begin{cases} 1, & F(\overline{x^*}) - F(\overline{x_i}) < 0 \\ \exp\left(-\dfrac{F(\overline{x^*}) - F(\overline{x_i})}{Q_i}\right), & F(\overline{x^*}) - F(\overline{x_i}) \geqslant 0 \end{cases}$$

произойдёт с вероятностью, которая вычисляется в соответствии с распределением Гиббса:
Так, в качестве оператора A используем функцию randomIndex(int i), которая будет возвращать индекс следующего рассматриваемого решения .

```
public int randomIndex(int i){
  int r = random.nextInt(solutions.size()-i);
  if (r==i) return i+1;
  return r;
}
```

A вероятность по формуле Гиббса рассчитывается в методе probability(int from, int to, int q), где from - индекс i, to - индекс, полученный оператором A из i, q - текущее значение температуры.

```
public double probability(int from, int to, int q){
   float diff = metrics[to]-metrics[from];
   if (diff<0) return 1;
   return Math.exp(-diff/q);
}
```

Теперь осталось создать цикл понижения температуры и вернуть индекс расписания, которое будет выбрано, когда температура станет нулевой.

```
public int findBestSolution(){
   int i = 0;
   metrics[0]=getRate(0);
   for (int q = dtc.length/10; q > 0; q--) {
      int x = randomIndex(0);
      getRate(x);
      double p = probability(i,x,q);
      if(random.nextFloat() < p){
         i = x;
      }
   }
   return i;
}
```

Из преимуществ этого метода оптимизации можно выделить его скорость, потому что расчёт метрик качества расписания осуществляется только для установленного числа решений. Но при этом это снижает точность выбора наиболее удачного расписания, так как присутствует фактор случайности при выборе следующего решения на каждом шаге алгоритма.