Лагоша А.М.
студент
Карелова Р.А.
к.п.н., доцент кафедры ИТ
Нижнетагильский технологический институт (филиал) УрФУ
г. Нижний Тагил, Россия

ВАРИАНТ РЕАЛИЗАЦИИ ГЕНЕТИЧЕСКОГО АЛГОРИТМА В РЕШЕНИИ ЗАДАЧИ СОСТАВЛЕНИЯ РАСПИСАНИЯ

Аннотация

В статье представлен вариант реализации генетического алгоритма для автоматизации составления расписания, требования к которому условно разделены на обязательные и желательные. Раскрыта идея учета обязательных условий на этапе создания особи, что экономит время на их оценку и отбор. Необязательные условия учитываются с помощью функции штрафов. Приведены примеры реализации алгоритма в виде кода на C++.

Ключевые слова: *составление расписания, генетический алгоритм, C++.*

Lagosha A.M.
student
Karelova R. A.
Ph. D., associate Professor of the Department of IT
Nizhniy Tagil Technological Institute (branch)UrFU
Nizhny Tagil, Russia

OPTIONAL VERSION OF THE GENETIC ALGORITHM IMPLEMENTATION IN SOLVING THE SCHEDULING PROBLEM

Abstract

The article presents a variant of the implementation of a genetic algorithm for automating scheduling, the requirements for which are conditionally divided into mandatory and desirable. The idea of taking into account mandatory conditions at the stage of creating an individual is disclosed, which saves time on their assessment and selection. Optional conditions are taken into account using the penalty function. Examples of the implementation of the algorithm in the form of C ++ code are given.

Keywords: scheduling, genetic algorithm, C ++.

Введение

Решение задачи составления расписания влияет на эффективность учебного процесса. Для нормального функционирования учебного заведения расписание занятий должно отвечать целому набору критериев и быть составлено своевременно. Несмотря на то, что у каждого учебного заведения набор критериев может содержать какие-то специфические пункты, одними из главных требований, как правило, являются: эффективное использование аудиторного фонда, комфортность обучения для студентов и работы преподавательского состава.

Составление расписания – это задача целочисленного программирования, относящаяся к классу NP-полных задач. Это значит, что данная задача не имеет алгоритма решения за полиномиальное время, что обусловлено спецификой задачи, большим объёмом различной по своему составу исходной информации и числом трудноформализуемых требований [1].

Существует множество различных методов решения задач целочисленного программирования: методы полного перебора, метод ветвей и границ, метод раскраски графов, эвристические методы, решатели SAT (расшифровывается как Satisfiability – задача о выполнимости булевых формул) и MIP (Mixed Integer Programming) [4]. Недостатком данных методов является громоздкость и сложность получаемой математической модели задачи составления расписания, резкий рост временных затрат с ростом объемов исходной информации на поиск решения в силу NP-сложного характера задачи составления расписания в ее классической постановке [3].

В данной статье речь пойдёт о методе решения этой задачи при помощи генетического алгоритма. Алгоритм для решения NP-трудной задачи, как правило, требует компромисса в отношении универсальности, правильности или скорости [4]. Генетический алгоритм является компромиссом в отношении правильности. Он позволяет найти приближённо правильное решение за разумное время, причём, предоставляет выбор: получить неточное решение быстрее или подождать и получить более точное.

Общий подход, реализованный в генетическом алгоритме, предлагает пользователю определить *fitness*-функцию, которая будет оценивать полученные решения, а затем менее удачные из них будут отсеиваться по мере работы алгоритма. Такой подход требует больших временных затрат, так как создается много особей, которые с самого начала своего существования нарушают какие-либо ограничения, но значение *fitness*-функции для них все равно рассчитывается, и только потом они отсеиваются в процессе естественного отбора (или не отсеиваются). При этом нет никакой гарантии, что ограничения в итоге будут соблюдены.

В данной статье предлагается реализовать генетический алгоритм, который будет учитывать специфические требования к расписанию. Такие требования могут быть различными и зависят от особенностей учебного заведения, для которого необходимо составить расписание. Основная идея в том, чтобы использовать специфику данных и их ограничения для того, чтобы сузить область поиска, получить более частное, но более быстрое решение с гарантией выполнения некоторых обязательных условий. В рамках идеи условия, которые необходимо учитывать при составлении расписания, делятся на обязательные и желательные. Начиная с инициализации особей и на протяжении всех этапов работы алгоритма будут выполняться обязательные условия. К примеру, в процессе мутации расписания, аудитория для проведения занятия будет случайным образом выбираться не из всего множества аудиторий, а из списка тех аудиторий, которые указаны для данного занятия как желательные. Иными словами, мутация будет проходить подконтрольно, как и все остальные этапы алгоритма.

Входные данные

В качестве входных данных для алгоритма выступает список заявок на проведение занятий, где каждое занятие соответствует определённой паре в сетке расписания. У каждого учебного заведения могут быть свои форматы заявок на расписание, но у всех таких заявок можно выделить некие общие черты. Далее поясняется, что понимается под заявкой в данном решении.

Каждый объект занятия представляет собой заявку на место и время проведения занятия, а также включает пожелания преподавателя.

Такая заявка включает в себя:

– ФИО преподавателя;

– сложность занятия;

– предпочитаемые дни недели;

– список групп;

– список предпочитаемых аудиторий.

Требования к расписанию

Требования, предъявляемые к расписанию, было решено условно разделить на обязательные и желательные. Обязательные требования должны соблюдаться на каждом шаге работы генетического алгоритма. Это значит, что поиск решения производится только среди допустимых вариантов расписания, что выгодно отличает его от любых алгоритмов полного перебора. Необязательные требования выполняются путём минимизации функции штрафов за нарушение этих требований.

Обязательные требования:

– у группы в одно время может быть только одно занятие;

– у преподавателя в одно время может быть только одно занятие;

– занятие должно проводиться в один из дней, указанных преподавателем;

– в аудитории в одно время может проходить только одно занятие.

Желательные требования:

– минимальное количество «окон» у преподавателей;

– минимальное количество «окон» у групп;

– минимальная суммарная нагрузка на студентов в день [2];

– минимальное количество переходов между различными корпусами (когда аудитории находятся в разных зданиях).

Алгоритм

В данной реализации генетический алгоритм состоит из следующих шагов:

1. Инициализация особей
2. Пока не достигнуто заданное количество итераций
    1. Мутация особей
    2. Выбор N лучших особей
    3. Кроссовер (скрещивание)
    4. Естественный отбор
3. Выбор наилучшей особи (наилучшего решения)

Каждая стадия алгоритма будет подробнее рассмотрена ниже.

Алгоритм имеет пять различных настроек, изменяя которые можно легко его масштабировать:

– размер популяции;

– количество итераций;

– количество отсеиваемых особей;

– количество скрещиваний;

– вероятность мутации в процентах (целое число от 0 до 100).

Особь

Основными сущностями в данном алгоритме выступают особи. Особь в нашем случае это один из возможных вариантов готового расписания. Для особи выполняется очень важный инвариант: в каждый момент времени для данного варианта расписания, который представляет собой особь, выполняются все обязательные требования. Это значит, что не должно быть пересечений ни между группами, ни между преподавателями, ни между аудиториями.

Особь состоит из двух хромосом. Каждая хромосома состоит из генов, обозначаемых целыми числами, причём, номер гена соответствует номеру заявки на занятие. Информационным наполнением первой хромосомы являются аудитории, второй хромосомы — время проведения занятий (пар). Таким образом, в первой хромосоме значением i-го гена является номер (код)

аудитории из подмножества допустимых аудиторий, в которой предполагается провести данное занятие. Аналогично во второй хромосоме значением i-го гена является номер пары из допустимого подмножества временных интервалов (пар) обучения. Это означает, что первая и вторая хромосомы связаны с блоком циклов занятий особой связью, которую можно назвать связью «однозначного соответствия». Временные интервалы пар находятся в диапазоне от 0 до 84 (не включительно). Расписание составляется на две недели (числитель и знаменатель, не включая выходные), в день по 7 занятий. 12 * 7 = 84. (Рис. 1)

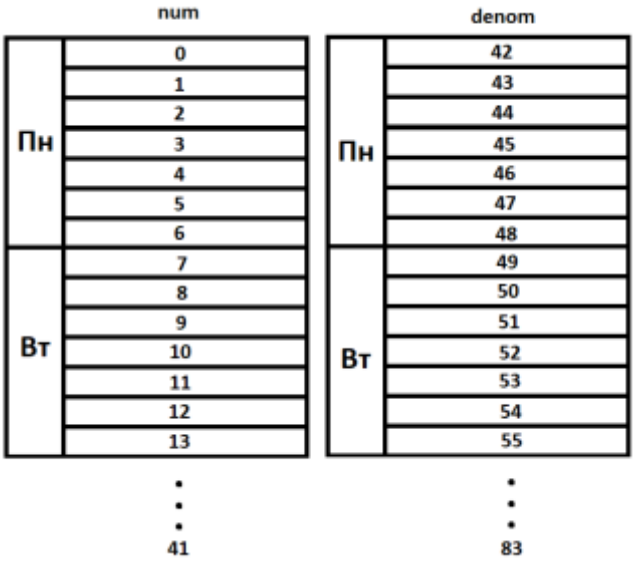

Рисунок 1 – Сетка расписания

Таким образом каждая особь (вариант расписания) будет содержать в себе все необходимые циклы (блоки) занятий с присвоенными им номерами аудиторий и установленными номерами пар (учебных единиц времени) и тем самым однозначно определять место и время проведения занятий, запланированных на семестр [3] (Рис.2). К примеру, из рисунка 2 можно определить, что предмет r_2 пройдёт 4 парой в понедельник в 3 кабинете корпуса 1.

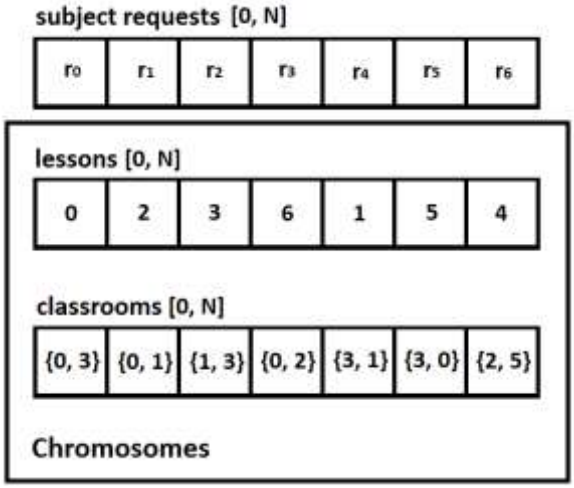

Рисунок 2 – Параллельные массивы: набор заявок и две хромосомы

Проверка выполнения обязательных условий

Выявление пересечений осуществляется простым обходом массива lessons и проверкой: не стоит ли в это же время пара, в которой задействуются одни и те же группы, преподаватели или кабинеты. Ниже представлен упрощённый пример на языке C++, в котором намеренно упущено множество подробностей:

```
struct subject_request
{
// пересекаются ли два пожеления
bool intersects(const subject_request& other) const
{
  return professor == other.professor ||
        set_intersects(groups, other.groups);
}

int professor = 0;
std::vector<int> groups;
};

struct chromosomes
{
// обнаружены ли пересечения при установке
// заявки занятия request на время lesson
bool intersects(const std::vector<subject_request>& requests,
                int request,
                int lesson) const
{
  auto it = std::find(std::begin(lessons), std::end(lessons), lesson);
  while(it != std::end(lessons))
  {
        const int other = std::distance(std::begin(lessons), it);
        if(requests[request].intersects(requests[other]) ||
          classrooms.at(request) == classrooms.at(other))
        {
              return true;
        }

        it = std::find(std::next(it), std::end(lessons), lesson);
  }
```

```
  return false;
}

std::vector<int> lessons;
std::vector<int> classrooms;
};


int main()
{
const std::vector requests{
  subject_request{.professor = 0, .groups = {1, 2, 3}},
  subject_request{.professor = 2, .groups = {3, 5, 8}},
  subject_request{.professor = 1, .groups = {0, 9, 15}}
};
const chromosomes chrom{
  .lessons = {0, -1, 1},
  .classrooms = {1, 2, 3}
};

for(int lesson = 0; lesson < 3; ++lesson)
{
  std::cout << (chrom.intersects(requests, 1, lesson) ?
                "Intersects!" :
                "No intersection") << std::endl;
}


return 0;
}
```

В примере рассмотрена ситуация, когда необходимо присвоить предмету, заявка для которого представлена элементом массива requests с индексом 1, свой временной промежуток. Как видно из консольного вывода, этот предмет нельзя поставить на промежуток 0, потому что это создаёт накладку по группам (у группы 3 получается два занятия в одно время).

Для проверки пересечений двух массивов групп используется функция set_intersects, определение которой представлено ниже:

```
template<class InputIt1, class InputIt2>
bool set_intersects(InputIt1 first1, InputIt1 last1,
```

```
                InputIt2 first2, InputIt2 last2)
{
    while (first1 != last1 && first2 != last2)
    {
        if (*first1 < *first2)
        {
            ++first1;
        }
        else
        {
            if (!(*first2 < *first1))
                return true;

            ++first2;
        }
    }
    return false;
}
template<class SortedRange1, class SortedRange2>
bool set_intersects(const SortedRange1& r1, const SortedRange2& r2)
{
    return set_intersects(std::begin(r1), std::end(r1), std::begin(r2),
std::end(r2));
}
```

Данная функция является немного изменённым аналогом алгоритма std::set_intersection. В отличие от *std::set_intersection* здесь не нужно хранить пересекающиеся элементы, а необходимо лишь установить факт такого пересечения.

В отличие от обычного алгоритма перебора элементов двух диапазонов и проверки их на равенство, имеющего асимптотику $O(M \cdot N)$, этот алгоритм имеет асимптотику O($2 \cdot (N + M) - 1$). Достигается такая асимптотика за счет того, что данный алгоритм рассчитывает на то, что оба диапазона были предварительно отсортированы.

Инициализация особи

Инициализация особи напоминает действия человека, пытающегося составить расписание. Алгоритм инициализации относится к категории «жадных алгоритмов» и позволяет довольно быстро получить решение. Это решение будет плохо удовлетворять желательным требованиям, таким как отсутствие «окон», минимизация количества сложных предметов в день и т.п.

Однако, это решение будет отвечать всем обязательными требованиям, то есть расписание уже не будет содержать пересечений по группам, преподавателям или аудиториям.

Перебирая все временные промежутки (от 0 до 83 включительно) происходит попытка поставить занятие в данное время, не создавая коллизий с другими занятиями. Перебор осуществляется особым образом: сначала происходит попытка поставить первую пару понедельника, если обнаруживается коллизия, то происходит попытка поставить первую пару во вторник и т.д. На рисунке 3 это показано наглядно на примере двух дней и при максимальном количестве 4 пары в день. Зелёным цветом выделены те временные промежутки, на которые алгоритм попытается расставить занятия в первую очередь, а красным цветом отмечены те, к которым алгоритм перейдёт только в том случае, если все остальные места уже заняты. В случае, если занятий слишком много и некоторые из них просто не могут быть расставлены, они игнорируются на этапе инициализации и могут быть расставлены на более поздних этапах.

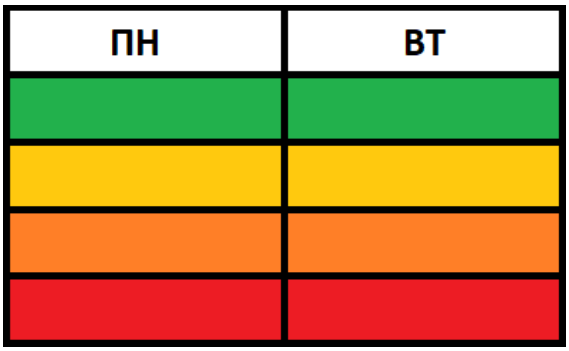

Рисунок 3 – Схема расстановки пар «жадным» алгоритмом (зелёный цвет – расстановка занятий на это время крайне желательна, жёлтый – данное время занятий вполне допустимо, оранжевый – достаточно позднее время проведения занятий, красный – ставить занятия так поздно крайне нежелательно)

Мутация

Для каждого индивида генерируется случайное целое число от 0 до 100 – вероятность мутации данного индивида, которая сравнивается с той вероятностью, что была принята как параметр алгоритма. Если вероятность мутации индивида меньше или равна заданной, то происходит мутация этого индивида.

Мутация может происходить по двум сценариям: изменение временного промежутка или изменение аудитории проведения занятия. Важно отметить, что на этапе мутации также необходимо учитывать все обязательные требования к расписанию, такие как отсутствие коллизий.

Селекция

После процесса мутации необходимо выбрать наиболее удачных особей для последующего скрещивания. Выбор происходит на основании результатов функции оценки качества расписания: чем значение функции меньше, тем лучше особь приспособлена (тем лучше расписание удовлетворяет требованиям). Таким образом, процесс селекции представляет собой выбор N индивидов с наименьшими значениями оптимизируемой функции.

Очевидное решение – отсортировать массив и взять N первых элементов. Стандартная библиотека шаблонов *STL* имеет функцию *std::nth_element*. Данная функция принимает три итератора: итератор на первый элемент, итератор на N-ный элемент и итератор на элемент, следующий за последним. Эта функция выставляет на N-ную позицию тот элемент, который занимал бы её, если бы массив был полностью отсортирован. Алгоритм *std::nth_element* выполняется за линейное время $O(N)$, в то время как любой алгоритм сортировки выполняется за время $O(N*logN)$.

В верхней части рисунка 4 показано состояние массива после работы алгоритма *std::nth_element*, а чуть ниже показано, как выглядел бы полностью отсортированный массив.

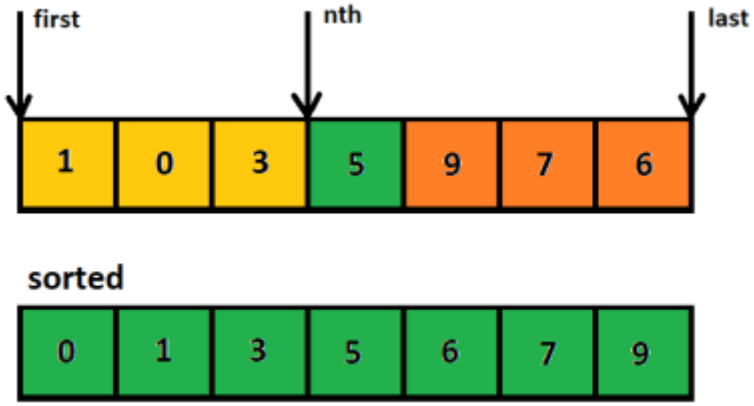

Рисунок 4 – Результат работы алгоритма *std::nth_element*

Естественный отбор

Естественный отбор происходит аналогичным с селекцией образом, только выбираются N особей с наибольшими значениями оптимизируемой функции. Эти N особей заменяются копиями более приспособленных особей (особей с меньшими значениями функции штрафов).

Скрещивание

Скрещивание двух особей происходит по следующей схеме: случайным образом выбирается номер i гена. Затем i-е элементы массивов временных промежутков одной особи и второй особи обмениваются местами. Аналогичным образом происходит и с массивами аудиторий двух особей.

Во время скрещивания необходимо проверять выполнение обязательных требований к расписанию. Достаточно проверять требования, связанные с пересечением групп, преподавателей или аудиторий. Такие требования, как, например, требование об определённых днях или определённых аудиториях проведения занятия будут выполняться автоматически, поскольку обе хромосомы уже будут содержать только допустимые значения.

Функция штрафов

Требования к расписанию условно делятся на обязательные и желательные. Обязательные требования выполняются для каждой особи на всём протяжении работы алгоритма. Это одна из важных составляющих данного алгоритма, выгодно отличающих его от алгоритмов простого перебора и хорошо сказывающаяся на его производительности. Необязательные требования трудно формализовать, к тому же они могут противоречить друг другу. Для того чтобы их учесть, предлагается реализовать функцию штрафов, назначающую особи штрафные очки за каждое нарушение этих требований.

Функция штрафов – это функция, значение которой необходимо минимизировать по ходу работы алгоритма. Она выступает в качестве *fitness*-функции в данном алгоритме (чем меньше её значение, тем лучше особь приспособлена). Функция штрафов представляет собой сумму штрафов за каждое нарушение желательных требований к расписанию. Каждому требованию соответствует свой коэффициент штрафа.

Чтобы вычислить функцию штрафов необходимо совершить обход всех преподавателей и всех групп по всем дням учебного периода (в нашем случае 12 дней). Для этого заранее подготавливается 2 отображения: множества преподавателей на индексы заявок этих преподавателей и групп на индексы заявок, в которых есть эти группы. Затем необходимо пройтись по парам значений [группа; набор индексов заявок] и вычислить максимальное значение каждого вида штрафа.

Для последующих вычислений (например, для вычисления количества «окон» в день) необходимо подготовить данные особым образом: отобразить множество индексов заявок в массиве заявок на множество временных промежутков, в которые предполагается проводить пары, подготовив массив пар значений, где первое значение – номер временного промежутка (от 0 до 83 включительно), а второе значение – индекс заявки на проведение занятия в это время.

Реализация данных манипуляций на языке C++ может выглядеть следующим образом:

```
const std::vector lessonsChromosome{0, 1, 3, 4, 5, 7, 8, 9, 10, 12};
const std::unordered_map<int, std::unordered_set<int>> groupRequests{
```

```
  {0, {1, 0, 3, 5, 8}},
  {1, {2, 6, 7, 9}}
 };

 for(auto&& [group, requests] : groupRequests)
 {
  std::vector<std::pair<int, int>> lessonsRequests;
  lessonsRequests.reserve(std::size(requests));
  std::ranges::transform(requests, std::back_inserter(lessonsRequests),
                         [&](int r) { return std::pair{lessonsChromosome[r], r}; });

      std::ranges::sort(lessonsRequests);
  // manipulations with lessonsRequests ...
 }
```

Затем, для того, чтобы отделить занятия, проходящие в один день, от занятий, проходящих в другой день, необходимо совершить обход списка временных промежутков. Осуществляется это следующим образом: первый элемент списка принимается за начало дня, затем при помощи бинарного поиска находится первый временной промежуток такой, который был бы не меньше, чем номер первого временного промежутка следующего дня – это и будет тот элемент, где заканчивается первый день и начинается второй.

Данный небольшой пример на языке C++ решает проблему отделения одного учебного дня от другого:

```
 // в данном массиве для примера содержатся номера временных промежутков трёх дней
 // первый день [0, 6]: 0, 1, 4, 5
 // второй день [7, 13]: 7, 12
 // третий день [14, 20]: 15
 const std::vector<int> lessons = {0, 1, 4, 5, 7, 12, 15};

 auto firstDayLesson = std::begin(lessons);
 const auto lastLesson = std::end(lessons);
 while(firstDayLesson != lastLesson)
 {
  const int lesson = *firstDayLesson;
  const int currentDay = lesson / MAX_LESSONS_PER_DAY;
  const int nextDayFirstLesson = (currentDay + 1) * MAX_LESSONS_PER_DAY;
```

const auto lastDayLesson = std::lower_bound(std::next(firstDayLesson), lastLesson,

nextDayFirstLesson);

std::copy(firstDayLesson, lastDayLesson, std::ostream_iterator<int>(std::cout, " "));
std::cout << std::endl;

firstDayLesson = lastDayLesson;
}

Данная программа выводит все временные промежутки в порядке возрастания, отделяя каждый новый день новой строкой. На рисунке 5 показано расположение пары итераторов: первый указывает на временной промежуток начала второго дня, а второй - следующий за последним временным промежутком второго дня.

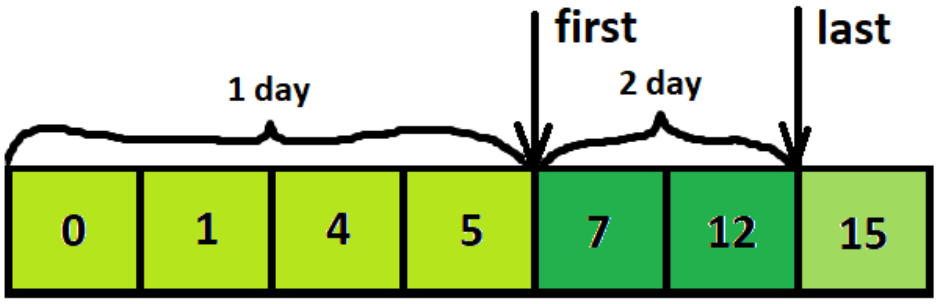

Рисунок 5 – Отделение диапазонов временных промежутков по дням

Вычисление количества окон в день

Суммарное количество «окон» в день для группы вычисляется как сумма разностей каждых двух смежных элементов *1*. Количество «окон» для преподавателей (W_p) вычисляется аналогичным образом.

$$W_g = \sum_{i=1}^{N} (l_i - l_{i-1} - 1)$$

где

l_{i-1} – номер предыдущего временного промежутка;

l_i – номер следующего временного промежутка;

N – количество проводимых занятий для данной группы.

Для вычисления суммарного числа «окон» за день используется *STL* алгоритм *std::inner_product*. На рисунке 6 показано, как можно использовать

функцию *std::inner_product* для расчёта количества «окон». Стрелками показаны положения итераторов.

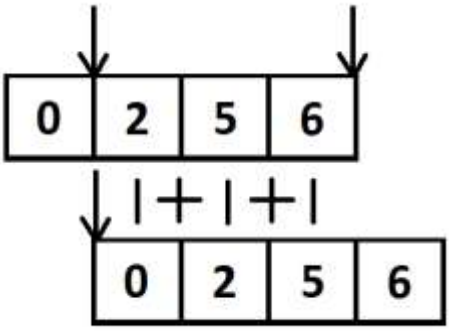

Рисунок 6 – Количество «окон» в день – сумма разностей каждых двух смежных элементов

Предусловия алгоритма: список временных промежутков должен быть отсортирован по возрастанию номеров временных промежутков.

Реализация примера на языке C++:

```
auto calculateGap = [](int first, int second) {
return first - second - 1;
};

const std::vector lessons{0, 2, 5, 6};

std::cout << "Count gaps: " <<
std::inner_product(std::next(std::begin(lessons)), std::end(lessons),
                   std::begin(lessons), 0, std::plus<>{}, calculateGap) <<
std::endl;
```

В качестве *map*-операции используется лямбда «calculateGap», а в качестве *reduce*-операции используется *std::plus*<>.

Вычисление суммарной сложности дня для студента

Начисление штрафов за «слишком сложный день» для студентов происходит следующим образом: порядковый номер пары за день (от 0 до 6) умножается на коэффициент сложности предмета [2]. Чем сложнее предмет, тем раньше он должен быть в сетке расписания и тем меньше таких предметов должно быть в один день.

$$C_g = \sum_{i=0}^{N} (l_i \bmod L_{max} * c_i)$$

где

l_i – номер текущего временного промежутка;

L_{max} – максимальное количество проводимых занятий в день;

c_i – сложность данного предмета от 1 до 4.

Предусловия алгоритма:

– список пар временных промежутков и сложности предметов должен быть отсортирован по возрастанию номеров временных промежутков;

– список не может содержать двух пар с одинаковым номером временного промежутка.

Реализация примера вычисления суммарной сложности предметов на языке C++:

```
const std::vector<std::pair<int, int>> lessons{{21, 1}, {23, 4}, {24, 3}, {25, 2}};

auto reduceOp = [](int accum, std::pair<int, int> p) -> int {
  auto[lesson, complexity] = p;
  return accum + (lesson % MAX_LESSONS_PER_DAY) * complexity;
};

std::cout << "Sum day complexity: " <<
std::accumulate(std::begin(lessons), std::end(lessons),
0, reduceOp) << std::endl;
```

В решении использован *STL* алгоритм *std::accumulate*, а в качестве *reduce*-операции передана лямбда «reduceOp». В примере используется массив пар значений <временной промежуток, сложность предмета> (данные могут быть организованы немного иначе, но принцип останется прежним).

Штраф за переходы между зданиями

Штраф за переходы между зданиями для групп (между различными корпусами учебного заведения) рассчитывается как сумма несовпадений зданий для проведения идущих подряд занятий. Если между парами в различных зданиях есть хотя бы одно «окно», считается, что штрафа удалось избежать. Штраф за переходы между зданиями для преподавателей (W_p) рассчитывается аналогично.

$$B_g = \sum_{i=1}^{N} (b_{i-1} \neq b_i \wedge (l_{i-1} - l_i > 1))$$

где

b_{i-1} – номер предыдущего здания;

b_i – номер следующего здания;

l_{i-1} – номер предыдущего временного промежутка;

l_i – номер следующего временного промежутка;

N – количество проводимых занятий для данной группы.

Предусловия алгоритма:

– список пар временных промежутков и зданий должен быть отсортирован по возрастанию номеров временных промежутков;

– список не может содержать двух пар с одинаковым номером временного промежутка.

Пример вычисления количества переходов между зданиями на языке C++:

```
const std::vector<std::pair<int, int>> lessons{{14, 0}, {15, 1}, {16, 1}, {17, 1}, {18, 0}, {20, 1}};

auto mapOp = [](auto lhs, auto rhs) -> bool {
  const auto[lhsLesson, lhsBuilding] = lhs;
  const auto[rhsLesson, rhsBuilding] = rhs;

  return !(lhsLesson - rhsLesson > 1 || lhsBuilding == rhsBuilding);
};

std::cout << "Count transitions: " <<
std::inner_product(std::next(std::begin(lessons)), std::end(lessons), std::begin(lessons), 0, std::plus<>{}, mapOp) << std::endl;
```

Штраф за невыставленное занятие

Алгоритм гарантирует выполнение такого важного требования как отсутствие пересечений аудиторий, групп и преподавателей, но при этом не гарантирует того, что все занятия будут выставлены. Такое может произойти, если, например, алгоритму было передано слишком много заявок на расписание или предъявленные ограничения были слишком жёсткие. В таком случае нельзя записать в хромосому временных промежутков или кабинетов необходимые значения. Для знакового целого таким значением-маркером может сложить число -1. Для беззнакового целого можно использовать максимальное для данного типа данных значение.

Для того чтобы получить количество невыставленных занятий и кабинетов необходимо посчитать количество таких значений-маркеров.

$$\chi_L = \sum_{i=0}^{N} (l_i \equiv -1), \qquad \chi_C = \sum_{i=0}^{N} (\{b_i, c_i\} \equiv \{-1, -1\}).$$

Итоговая функция штрафов выглядит следующим образом:

$$F(L, C) = k_{w_g} \cdot W_g + k_{B_g} \cdot B_g + k_{C_g} \cdot C_g + k_{w_p} \cdot W_p + k_{B_p} \cdot B_p + k_{\chi_L} \cdot \chi_L + k_{\chi_C} \cdot \chi_C$$

где

k_{w_g} –коэффициент штрафа за «окно» для групп;

W_g – максимальное количество «окон» в день среди всех групп;

k_{B_g} – коэффициент штрафа за переход между зданиями для групп;

B_g – максимальное количество переходов между зданиями в день среди всех групп;

k_{C_g} – коэффициент штрафа за сложность дня для групп;

C_g – максимальная суммарная сложность предметов среди всех групп в день;

k_{w_p} –коэффициент штрафа за «окно» для преподавателей;

W_p – максимальное количество «окон» в день среди всех преподавателей;

k_{B_p} – коэффициент штрафа за переход между зданиями для преподавателей;

B_p – максимальное количество переходов между зданиями в день среди всех преподавателей;

k_{χ_L} – коэффициент штрафа за неопределённое время занятия;

χ_L – количество занятий, которым не удалось присвоить время;

k_{χ_C} – коэффициент штрафа за неопределённую аудиторию;

χ_C – количество занятий, которым не удалось присвоить аудиторию проведения.

Функция принимает на вход 2 множества (2 хромосомы): множество установленных временных промежутков и множество установленных аудиторий для проведения занятий, а возвращает целое значение, которое будет являться оценкой полученного расписания. Каждый вид штрафа рассчитывается для каждой группы/преподавателя за каждый день, и среди полученных значений выбирается максимальное. Это позволяет минимизировать вероятность возникновения крайне «нагруженных» дней для преподавателей и групп. С помощью коэффициентов можно настраивать величину влияния каждого требования на итоговое значение функции оценки.

Итоговый алгоритм

Алгоритм оперирует набором особей, представляющих собой популяцию. На вход алгоритму подаётся набор заявок на занятия, исходя из которых производится инициализация особей и дальнейшая работа алгоритма. Каждый «индивид» реализует механизм кеширования результата вычисления функции приспособленности (метод Evaluate). Критерием остановки алгоритма является количество итераций, так как это самый простой критерий остановки, не требующий дополнительных вычислений.

На каждой итерации выполняется следующий порядок действий:

1. мутация индивидов (с последующим вычислением функции приспособленности);
2. селекция;
3. кроссинговер;
4. вычисление и кеширование значений функции приспособленности (функции штрафов);
5. естественный отбор.

Мутация индивидов и вычисление функции приспособленности (1 и 4 этапы) могут производиться параллельно. Так как особи не имеют общих изменяемых данных или временных зависимостей друг от друга, то нет необходимости в использовании блокировок или защищённых секций кода. Отсутствие таких блокировок позволит извлечь максимальную пользу от параллельного выполнения. На языке C++ это может выглядеть следующим образом:

```
const int selectionCount = 5;
const int crossoverCount = 3;
const int mutationChance = 50;

auto individualLess = [](auto&& lhs, auto&& rhs) {
  return lhs.Evaluate() < rhs.Evaluate();
};

std::random_device randomDevice;
std::mt19937 randGen(randomDevice());

// mutate
std::for_each(std::execution::par_unseq, individuals.begin(), individuals.end(),
[mutationChance](auto&& individual){
  if(individual.MutationProbability() <= mutationChance)
     {
        individual.Mutate();
        individual.Evaluate();
     }
});

// select best
std::nth_element(individuals.begin(), individuals.begin() + selectionCount,
          individuals.end(), individualLess);
```

```
// crossover
for(int i = 0; i < crossoverCount; ++i)
{

std::uniform_int_distribution<std::size_t> selectionBestDist(0, selectionCount - 1);
std::uniform_int_distribution<std::size_t> individualsDist(0, individuals.size() - 1);

   ScheduleIndividual& firstInd = individuals[selectionBestDist(randGen)];
   ScheduleIndividual& secondInd = individuals[individualsDist(randGen)];
   firstInd.Crossover(secondInd);
}

std::for_each(std::execution::par_unseq, individuals.begin(), individuals.end(),
        [](auto&& individual) { individual.Evaluate(); });

// natural selection
std::nth_element(individuals.begin(), individuals.end() - selectionCount,
                individuals.end(), individualLess);

std::copy_n(individuals.begin(), selectionCount, individuals.end() - selectionCount);
```

Данный пример приведен в упрощенном виде, в нем намеренно упущены многие детали, которые будут во многом зависеть от конкретного способа реализации. Здесь используется специальная параллельная реализация стандартного алгоритма *std::for_each*, вызов которой происходит с политикой выполнения *std::execution::par_unseq*, переданного в качестве первого аргумента функции. Политика *std::execution::par_unseq* означает, что элементы списка могут обрабатываться параллельно в любом порядке, то есть, могут использоваться многопоточность и векторизация (например, наборы инструкций «sse», «avx» и т. п.).

Заключение

В результате реализации приведённого решения на языке C++ удалось достичь довольно высокой производительности при низких затратах аппаратных ресурсов. К примеру, генетический алгоритм с размером популяции 1000 особей и числом итераций 1000 находит решение менее чем за

2 секунды (Рисунки 7 и 8). Время выполнения может отличаться в зависимости от объёма входных данных и производительности процессора.

```
TEST_CASE("ScheduleGA benchmark")
{
    std::random_device randomDevice;
    ScheduleDataGenerator generator{randomDevice, 1, 5, 0, 7};
    const ScheduleData data{generator.GenerateSubjectRequests(200)};

    ScheduleGA algo{ScheduleGAParams{
        .IndividualsCount = 1000,
        .IterationsCount = 1000,
        .SelectionCount = 360,
        .CrossoverCount = 220,
        .MutationChance = 49
    }};

    BENCHMARK("ScheduleGA")
    {
        algo.Start(data);
        return algo.Individuals().front().Evaluate();
    };
}
```

Рисунок 7 – Параметры алгоритма

```
Catch_benchmark_ScheduleGA.exe is a Catch v2.13.3 host application.
Run with -? for options

-------------------------------------------------------------------------------
ScheduleGA benchmark
-------------------------------------------------------------------------------
..\..\..\benchmark_ScheduleGA.cpp(6)
...............................................................................

benchmark name                       samples       iterations    estimated
                                     mean          low mean      high mean
                                     std dev       low std dev   high std dev
-------------------------------------------------------------------------------
ScheduleGA                                 100             1     4.17745 m
                                     1.74759 s      1.7281 s     1.77234 s
                                     111.466 ms    93.7169 ms    142.432 ms


===============================================================================
test cases: 1 | 1 passed
assertions: - none -
```

Рисунок 8 – Скорость работы алгоритма в Catch2 Benchmark

Особенность данной реализации генетического алгоритма заключается в том, что поиск решения производится в области допустимых решений, не нарушающих основные требования к расписанию. У такого решения есть два основных преимущества. Во-первых, чем больше таких обязательных ограничений, тем значительнее получается сократить время поиска решений,

поскольку такие требования уменьшают количество степеней свободы на каждом шаге алгоритма. Во-вторых, такая реализация позволяет давать гарантию того, что обязательные требования будут соблюдены в независимости от того, найдено ли оптимальное решение или нет. Наличие таких гарантий очень важно для решения задачи автоматизации составления расписания, поскольку нарушение таких требований приведёт к нарушению учебного процесса.

Желательные требования соблюдаются путём минимизации значения функции штрафов за нарушение этих требований. Это позволяет учитывать даже противоречащие друг другу требования, такие как: требование об отсутствии «окон» и требование об обязательном наличии «окна» между двумя занятиями, проводимыми в разных зданиях учебного заведения.

Литература

1. Аль-Габри, В.М. Обзор литературный источников по теме «Автоматизация составления расписания занятий и экзаменов в высших учебных заведениях» / В.М.Аль-Габри // Вестник Донского государственного технического университета, 2017. - №1 (88). – С.132-143.

2. Веревкин, В.И. Автоматизированное составление расписания учебных занятий вуза с учетом трудности дисциплин и утомляемости студентов / В.И.Веревкин, О.М.Исмагилова, Т.А.Атавин // Доклады ТУСУРа, 2009. - № 1(10), часть 1. – С.221-225.

3. Кабальнов, Ю.С. Композиционный алгоритм составления расписания учебных занятий / Ю.С.Кабальнов. Л.И.Шехтман, Г.Ф.Низамова, Н.А.Земченкова // Вестник УГАТУ, 2006. - Т.7, № 2(15). - С. 99-107.

4. Тим Рафгарден «Совершенный алгоритм. Алгоритмы для NP-трудных задач». – СПб.: Питер, 2021. – 304 с.: ил. (Серия Библиотека программиста). ISBN 978-5-4461-1799-4.