МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ
Кафедра дискретной математики и алгоритмики

МЕТОДЫ РЕШЕНИЯ ЗАДАЧИ О НАЗНАЧЕНИЯХ В РЕЖИМЕ РЕАЛЬНОГО ВРЕМЕНИ

Курсовая работа

Азявчикова Алексея Павловича
студента 4-го курса
специальности 1-31 03 04
«Информатика»

Научный руководитель:
старший преподаватель
Г. П. Волчкова
Консультант:
старший преподаватель
И. Д. Лукьянов

Минск, 2024

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В современном мире, характеризующемся динамично меняющимися условиями и высокой конкуренцией, эффективное управление ресурсами является ключевым фактором успеха любой организации. Особенно актуальной становится задача оптимального распределения задач между сотрудниками, позволяющая максимизировать производительность и минимизировать временные затраты. Традиционные методы планирования и распределения задач, основанные на статическом анализе и ручном управлении, часто оказываются неэффективными в условиях постоянно поступающих новых задач и изменяющейся доступности работников.

Целью данного курсового проекта является разработка автоматической системы распределения задач между работниками в режиме онлайн, покрытие её тестами и оценка временных характеристик основных операций над нею.

ГЛАВА 1
ОСНОВНЫЕ ОПРЕДЕЛЕНИЯ И ПОНЯТИЯ

1.1 Основные понятия

Определение 1. *Задача*: единица работы, требующая выполнения. Характеризуется типом и вероятностью появления.

Определение 2. *Работник*: исполнитель задач. Характеризуется набором навыков (списком типов задач, которые он способен выполнять).

Определение 3. *Онлайн-алгоритм*: Алгоритм, обрабатывающий входные данные (поступающие задачи и приходящие работники) последовательно, без знания будущих данных. [2]

Определение 4. *Оффлайн-алгоритм*: Алгоритм, имеющий доступ ко всем входным данным (всем задачам и работникам) заранее, до начала процесса распределения. [6]

1.2 Постановка задачи

Пусть $Tasks = \{t_1, t_2, \ldots, t_n\}$ - множество задач, поступающих последовательно. Каждая задача t_i характеризуется своим типом $type(t_i) \in Types$, где $Types$ - множество всех возможных типов задач.

Пусть $Workers = \{w_1, w_2, \ldots, w_m\}$ - множество работников. Для каждого работника w_j определено множество $WorkerAbilities(w_j) \subseteq Types$ - типы задач, которые он способен выполнять.

На каждом шаге i алгоритм получает задачу t_i и должен немедленно назначить её одному из доступных работников w_j таких, что $type(t_i) \in WorkerAbilities(w_j)$. Если таких работников нет, задача t_i остаётся невыполненной. Допускается назначать каждому работнику не более одной задачи.

Задачей курсовой работы является реализация онлайн-планировщика на языке C++, который распределяет работы по доступным работникам, максимизируя количество выполненных задач.

Другими словами, планировщик должен поддерживать следующие виды операций:

1. $AddWorkerTypeInfo(WorkerType, WorkerAbilities)$: получить и обработать информацию о новом типе работника представленную в виде списка работ, которые он может выполнять ($WorkerAbilities$);

2. $AddWorker(Worker)$: обработать событие появления работника (гарантируется, что информация о типе работника уже была предоставлена структуре);

3. $AddTask(Task, OnScheduledCallback)$: обработать событие появления новой задачи и вызвать переданную функцию $OnScheduledCallback$ в момент, когда задаче будет назначен работник.

1.3 Области применения онлайн-алгоритмов о назначениях

Примеры областей применения онлайн-алгоритмов о назначениях:

• Райдшеринговые платформы: Сервисы, подобные Uber и Lyft, используют онлайн алгоритмы для определения цены поездки в зависимости от спроса и предложения в режиме реального времени. Когда поступает запрос на поездку, система должна назначить водителя и определить цену, учитывая текущее местоположение водителей, ожидаемое время ожидания и прогнозируемый спрос.

• Сопоставление водителей с заказами в службах доставки еды: Платформы доставки еды, такие как Delivery Club и Яндекс.Еда, используют онлайн алгоритмы для назначения курьеров заказам. Система должна учитывать расстояние до ресторана и клиента, время приготовления заказа, наличие свободных курьеров и желаемое время доставки, чтобы минимизировать время ожидания клиентов и максимизировать эффективность работы курьеров.

• Распределение задач в облачных вычислениях: Облачные провайдеры используют онлайн алгоритмы для распределения вычислительных ресурсов между различными задачами. Когда поступает новый запрос на вычисления, система должна решить, на каком сервере его выполнить, учитывая доступные ресурсы, требования задачи и целевые показатели производительности.

• Оптимизация логистики в цепях поставок: Компании, занимающиеся логистикой, используют онлайн алгоритмы для управления транспортными потоками и распределения грузов. Когда поступает новый заказ на перевозку, система должна назначить транспортное средство, определить маршрут и учесть ограничения по вместимости, срокам доставки и стоимости перевозки.

• Подбор персонала на краткосрочные проекты: Платформы для поиска фрилансеров и временных сотрудников могут использовать онлайн алгоритмы для сопоставления специалистов с краткосрочными проектами. Когда публикуется новый проект, система должна быстро найти подходящих кандидатов, учитывая их навыки, опыт, доступность и желаемую оплату.

ГЛАВА 2
РЕАЛИЗАЦИИ ОНЛАЙН-АЛГОРИТМА

2.1 Первый вариант реализации алгоритма

2.1.1 Описание

Пусть дана последовательность событий, состоящая из появления задач и работников. И пусть также в момент появления каждого работника уже известна информация о его типе, т.е. известно, какие задачи способен выполнять работник и какова вероятность появления этих задач.

Тогда будем поддерживать отсортированный набор типов работников по возрастанию суммарной вероятности появления задач, которые данный тип способен выполнять. (это можно сделать, сложив типы в std::map или в std::vector, который в последующем будет отсортирован (но такой способ работает, если информация о всех типах известна заранее, т.е. на старте алгоритма)) В момент появления нового работника будем класть его в контейнер, позволяющий сопоставлять типу работника всех незанятых работников этого типа. (В данной работе использовался std::vector из предположения, что количество различных типов невелико ($\leqslant 10^4$)). А в момент появления новой задачи будем совершать проход по отсортированным типам до момента, пока не встретим тип, для которого имеются незанятые работники. В случае, если такого типа не нашлось, задача остается неназначенной, а алгоритм при этом продолжает работу.

2.1.2 Временная сложность

Здесь и далее введем обозначения: $n = |Workers|$ - общее число пришедших работников, $m = |Tasks|$ - общее число пришедших задач, W - количество различных типов работников, $T = |Types|$ - количество различных типов задач.

• AddWorkerTypeInfo работает за $O(log(W))$ так как требуется вставить информацию и новом типе в отсортированный контейнер, что работает по свойствам std::map за $O(log(W))$ [4].

• AddWorker работает за $O(1)$ так как требуется положить нового работника в список работников такого же типа, хранящийся в векторе векторов, что происходит по свойствам std::vector за $O(1)$.

• AddTask работает за $O(W * log(W))$ так как в худшем случае требуется пройтись по всему набору отсортированных типов работников, что по свойствам std::map работает за $O(W * log(W))$, так как каждый переход осуществляется за $O(log(W))$.

2.1.3 Затраты памяти

Затраты памяти составляют $O(n+W+W*T)$, так как требуется сохранить информацию обо всех незанятых работниках ($O(n)$) и поддерживать std::map для всех типов работников, что занимает ещё $O(W)$ памяти, а также потребуется хранить матрицу размера $W * T$ для быстрого определения возможности типом работника выполнять заданный тип задач.

2.2 Второй вариант реализации алгоритма

2.2.1 Описание

В данной реализации мы будем полагать, что информация обо всех типах работников известна к моменту поступления первой задачи. Главным отличием данной реализации от предыдущей будет попытка учесть количество работников для соответствующего типа, чтобы избегать ситуации, когда мы использовали всех работников, соответствующих типу с наименьшей суммарной вероятностью появления задач, способных быть выполненными этим типом, из-за чего некоторые маловероятные задачи лишаются исполнителей. Для этого мы будем сортировать типы работников (для которых существуют работники) по суммарной вероятности появления задач этого типа, делённой на количество работников данного типа. (т.е. каждому типу работников будет поставлена в соответствие величина $\frac{\sum_{j=1}^{T} p_j}{\sum_{i=1}^{W} I_{ij}}$, где p_j - вероятность появления j-й задачи, а I_{ij} - индикатор события "работник с типом i может выполнить задачу типа j")

При появлении работника мы будем пересчитывать метрику соответствующего ему типа. А при появлении задачи будем пытаться выбрать тип работника с наименьшей метрикой, способный выполнять данный тип задач и для которого остались незанятые работники. Если такой тип найти не удастся, алгоритм оставит задачу невыполненной и перейдет к выполнению следующих задач. Для хранения информации о метриках типов в отсортированном виде и её динамическом обновлении при добавлении работников и назначении работников на задачи будем использовать std::set.

2.2.2 Временная сложность

• AddWorkerTypeInfo работает за $O(1)$ так как гарантируется, что информация о работнике заданного типа поступает позже информации о типе, а значит нам не потребуется на данном этапе вставлять информацию о типе в отсортированный по введенной выше метрике контейнер, а нужно будет лишь сложить тип в список известных типов, представленный в виде std::vector, что работает за O(1).

• AddWorker работает за $O(log(W))$ так как требуется изменить значение метрики для типа данного работника, что работает по свойствам std::set [4] (в котором эти типы хранятся) за $O(log(W))$, так как число типов в std::set = $O(W)$.

• AddTask работает за $O(W * log(W))$ так как в худшем случае требуется пройтись по всему набору отсортированных типов работников, что по свойствам std::set работает за $O(W * log(W))$, так как каждый переход осуществляется за $O(log(W))$. А также потребуется затратить $O(log(W))$ времени на обновление метрики для соответствующего типа работников, если будет найден подходящий тип.

2.2.3 Затраты памяти

Затраты памяти составляют $O(n + W + W * T)$, так как требуется сохранить информацию обо всех незанятых работниках ($O(n)$) и поддерживать std::set для всех типов работников, что занимает ещё $O(W)$ памяти, а также потребуется хранить матрицу размера $W * T$ для быстрого определения возможности типом работника выполнять заданный тип задач.

ГЛАВА 3
ТЕСТИРОВАНИЕ РЕАЛИЗОВАННЫХ АЛГОРИТМОВ

3.1 Тестирование корректности работы

Тестирование корректности работы реализованных структур проводилось с помощью рандомизированных тестов и адресного санитайзера для C++ [3]. Для этого генерировались заведомо выполнимые тестовые сценарии, и, запуская на них тесты, проверялась корректность назначения работников на работы (действительно ли алгоритм назначает существующих работников и действительно ли эти работники способны выполнять данный тип задач). Также было написано некоторое количество тестов, проверяющих логику решения на тривиальной задаче, а именно: в ситуации, когда каждый тип работников способен выполнять только один тип задач. В такой ситуации ожидалось, что алгоритм, вне зависимости от его реализации, назначит исполнителей всем задачам. Для тестирования использовалась библиотека gtest.

3.2 Генерация тестовых сценариев

Для генерации заведомо выполнимых тестов использовалось точное решение оффлайн-задачи, основанное на венгерском алгоритме.

3.2.1 Венгерский алгоритм

Алгоритм Куна-Манкреса, также известный как венгерский алгоритм [1] [5], предназначен для решения задачи о назначениях. В данном алгоритме предполагается, что каждый работник i может выполнять любую работу j с определенной стоимостью C_{ij}. Для моделирования невозможности выполнить работу этим стоимостям будут присваиваться достаточно большие значения, чтобы при существовании корректного решения получить меньшую суммарную стоимость работ.

Алгоритм основан на построении совершенного паросочетания минимального веса в двудольном графе $G = (U, V, E)$, где U --- множество работников, V --- множество работ, а E --- множество ребер с весами C_{ij}.

Ключевую роль играют потенциалы вершин: u_i для работников $i \in U$ и v_j для работ $j \in V$. Ребро (i, j) называется допустимым, если $C_{ij} = u_i + v_j$. Алгоритм итеративно увеличивает размер паросочетания, используя только допустимые ребра.

Описание алгоритма:

1. **Инициализация:**

• Установить потенциалы работников: $u_i = 0$ для всех $i \in U$.

• Установить потенциалы работ: $v_j = \min_{i \in U} C_{ij}$ для всех $j \in V$.

• Инициализировать паросочетание $M = \emptyset$.

2. **Поиск увеличивающего пути:** Пока $|M| < n$:

• Выбрать свободного работника $i \in U$ (не входящего в M).

• Запустить поиск в ширину по допустимым ребрам, начиная с работника i, помечая посещенные вершины. Для этого использовать очередь Q и массив `visited`.

• Если найден свободный узел работы $j \in V$ (не входящий в M), то найден увеличивающий путь. В этом случае реконструировать увеличивающий путь от j до i (обычно с помощью массива предков), расширить паросочетание M путем инвертирования ребер на найденном пути (добавить ребра, которых не было в M, и удалить ребра, которые были) и перейти к следующей итерации основного цикла (шаг 2).

• Если увеличивающий путь не найден, то вычислить $\Delta = \min\{C_{ij} - u_i - v_j\}$ для всех посещенных работников i и непосещенных работ j. Затем обновить потенциалы: $u_i = u_i + \Delta$ для всех посещенных работников i; $v_j = v_j - \Delta$ для всех работ j, достижимых по чередующейся цепи из допустимых ребер, начиная со свободного работника i (это можно эффективно реализовать, используя массив предков, построенный во время поиска в ширину).

3. **Результат:** Паросочетание M содержит оптимальное назначение. Стоимость назначения равна $\sum_{(i,j) \in M} C_{ij}$.

Классическая реализация алгоритма Куна-Манкреса имеет сложность $O(n^3)$.

3.2.2 Генерация тестов на основе оффлайн-алгоритма

Интерфейс тестового генератора должен быть таким, чтобы для заданного числа задач с заданными вероятностями появления каждой из них генерировалась последовательность задач заданного размера, удовлетворяющая заданным вероятностям с небольшой погрешностью. Для этого сначала генерируется набор задач по заданным вероятностям, а затем предпринимается попытка сгенерировать для этого набора работников, способных выполнить имеющуюся последовательность. Для этого используются идеи бинарного поиска по ответу, и последовательность работников генерируется до тех пор, пока не станет возможным выполнение поставленных задач. Определить, возможно ли выполнение поставленных задач, помогает вышеописанный венгерский алгоритм. Имея последовательность работников, из которой можно назначить биективно работников задачам, остается лишь убрать из нее незадействованных работников.

Далее требуется решить более верхнеуровневую проблему: генерацию типов работников. Для этого вводится гиперпараметр теста *p*, обозначающий вероятность появления у некоторого типа работника способности к выполнению выбранной задачи. Этот параметр имеет смысл процента заполненности матрицы компетенций, получаемой если поставить в соответствие каждому типу

работника i столбец, а каждому типу задачи j строку, где на пересечении будет стоять 0 или 1 в зависимости от того, способен ли i тип выполнять j тип задач (0 - нет, 1 - да). Т.е. для заданного p мы будем генерировать компетенции работников (несколько вариантов), а для каждого варианта таких компетенций будем генерировать тестовые сценарии, по которым будем считать метрику среднего числа выполненных задач до первого провала и медиану от того же значения.

3.3 Оценки качества алгоритма

Напомним, что под p мы понимаем вероятность появления у i типа работников способности выполнять j тип задач.

Ниже приведены замеры качества для разных матриц компетенции при заданном p, где Avg означает среднее значение числа успешно назначенных задач до первой неудачи, а Mdn - медиану числа успешно назначенных задач до первой неудачи.

Для каждого типа матрицы компетенции (ниже обозначаемой как setup) генерировалось не менее 50 тестовых последовательностей с числом используемых работников и задач не менее 100.

Таблица 3.3.1 – Сравнительная точность алгоритмов при p=50%.

	Algo-1 Avg	Algo-1 Mdn	Algo-2 Avg	Algo-2 Mdn
Setup-1	0.973	0.98	0.99	1
Setup-2	0.976	0.985	0.994	0.995
Setup-3	0.983	0.995	0.994	1
Setup-4	0.987	1	0.995	1
Setup-5	0.98	0.995	0.996	1
Setup-6	0.99	1	0.996	1
Setup-7	0.985	1	0.998	1
Setup-8	0.982	1	0.999	1
Setup-9	0.993	1	0.999	1
Setup-10	0.985	1	1	1

Как видно, при большом значении параметра p оба алгоритма ведут себя очень хорошо, что объясняется тем, что из-за относительно большого набора компетенций у каждого типа работников повышается взаимозаменяемость работников. При ещё большем увеличении значения p результаты работы алгоритмов только улучшаются.

Таблица 3.3.2 – Сравнительная точность алгоритмов при p=40%.

	Algo-1 Avg	Algo-1 Mdn	Algo-2 Avg	Algo-2 Mdn
Setup-1	0.948	0.95	0.98	0.99
Setup-2	0.931	0.96	0.982	0.995
Setup-3	0.927	0.95	0.986	0.99
Setup-4	0.979	0.99	0.987	1
Setup-5	0.939	0.97	0.988	0.99
Setup-6	0.948	0.965	0.993	1
Setup-7	0.956	0.98	0.993	1
Setup-8	0.961	0.995	0.993	1
Setup-9	0.969	1	0.993	1
Setup-10	0.971	0.975	0.995	1

Таблица 3.3.3 – Сравнительная точность алгоритмов при p=30%.

	Algo-1 Avg	Algo-1 Mdn	Algo-2 Avg	Algo-2 Mdn
Setup-1	0.8328	0.835	0.8582	0.87
Setup-2	0.681	0.69	0.872	0.9
Setup-3	0.8108	0.84	0.8802	0.93
Setup-4	0.726	0.75	0.8914	0.905
Setup-5	0.7864	0.81	0.915	0.935
Setup-6	0.8204	0.84	0.9194	0.95
Setup-7	0.892	0.905	0.9348	0.945
Setup-8	0.8666	0.9	0.9382	0.95
Setup-9	0.8258	0.84	0.9412	0.97
Setup-10	0.8922	0.91	0.9462	0.96

Как видно, при уменьшении p, т.е. при сужении компетенций работников, метрики ухудшаются.

Таблица 3.3.4 – Сравнительная точность алгоритмов при p=20%.

	Algo-1 Avg	Algo-1 Mdn	Algo-2 Avg	Algo-2 Mdn
Setup-1	0.579	0.62	0.733	0.76
Setup-2	0.671	0.71	0.782	0.81
Setup-3	0.802	0.8	0.815	0.84
Setup-4	0.837	0.84	0.82	0.84
Setup-5	0.891	0.9	0.838	0.86
Setup-6	0.903	0.92	0.885	0.91
Setup-7	0.801	0.8	0.892	0.92
Setup-8	0.889	0.91	0.897	0.93
Setup-9	0.836	0.86	0.904	0.92
Setup-10	0.8308	0.84	0.8802	0.93

Самых плохих показателей удалось добиться при значении p=20-10%. Для первого алгоритма самое маленькое значение Avg=0.579, а для второго - 0.72

Таблица 3.3.5 – Сравнительная точность алгоритмов при p=10%.

	Algo-1 Avg	Algo-1 Mdn	Algo-2 Avg	Algo-2 Mdn
Setup-1	0.8775	0.905	0.8135	0.83
Setup-2	0.805263	0.84	0.861053	0.88
Setup-3	0.609	0.625	0.812	0.855
Setup-4	0.8775	0.905	0.8135	0.83
Setup-5	0.958571	1	0.836429	0.88
Setup-6	0.726	0.745	0.8815	0.9
Setup-7	0.642222	0.64	0.72	0.73
Setup-8	0.8305	0.86	0.805	0.82
Setup-9	0.609	0.625	0.812	0.855
Setup-10	0.8775	0.905	0.8135	0.83

ЗАКЛЮЧЕНИЕ

В процессе выполнения курсовой работы были получены следующие результаты:

- Разработаны 2 вариантов требуемого алгоритма;
- Протестирована корректность работы разработанных алгоритмов;
- Проведены оценки качественных характеристик разработанных алгоритмов;
- Реализованные алгоритмы упевают в худшем случае рапределить не менее 57% (первый) и 70%(второй) задач до первой неудачи.

Программный код реализованных структур и тесты к ним выложены на github: https://github.com/AzyavchikovAlex/Coursework_7sem.

В дальнейшем развитие курсовой работы будет связано с усложнением математической модели задачи и совершенствованием тестирующей системы для неё.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

[1] Magnanti T. L. Orlin J. B. Ahuja, R. K. Network flows: theory, algorithms, and applications, 1993.

[2] Rivest R. L. Stein C. Cormen T. H., Leiserson C. E. Introduction to algorithms, 2022.

[3] MemorySanitizer documentation [Electronic resource]. Mode of access: https://clang.llvm.org/docs/MemorySanitizer.html. Date of access: 20.12.2024.

[4] C++ documentation [Electronic resource]. Mode of access: https://en.cppreference.com/w/. Date of access: 20.12.2024.

[5] Christos H. Papadimitriou and Kenneth Steiglitz. Combinatorial optimization: Algorithms and complexity, 1998.

[6] Vazirani V.V. Approximation algorithms, 2001.

ПРИЛОЖЕНИЕ А

Интерфейс планировщика

```
typedef std::function<void(Worker)> WorkerCallback;

class SchedulerInterface {
 public:
  SchedulerInterface() = delete;
  virtual ~SchedulerInterface() = default;
  SchedulerInterface(WorkerType max_worker_type, TaskTypeId max_task_type)
      : max_task_type_(max_task_type), max_worker_type_(max_worker_type) {
  }
  virtual void AddWorkerTypeInfo(WorkerType type,
                                 WorkerAbilities abilities) = 0;
  virtual void AddWorker(Worker worker) = 0;
  virtual void AddTask(Task task,
                       std::function<void(Worker)> onScheduledCallback) = 0;

  virtual void UpdateSelfTime(Time time) = 0;

 protected:
  const WorkerType max_worker_type_;
  const TaskTypeId max_task_type_;
};

typedef std::shared_ptr<SchedulerInterface> SchedulerPtr;
```

ПРИЛОЖЕНИЕ Б

Интерфейс тестового сценария

```
struct TestCase {
  std::vector<Query> queries;
  WorkerType max_worker_type;
  TaskTypeId max_task_type;
};
```

```
struct Query {
  ActionPtr action;
  Time time;
};
```

```
class Action {
 public:
  Action() = default;
  virtual ~Action() = default;
  virtual void operator()(const SchedulerPtr& scheduler) = 0;
};

typedef std::shared_ptr<Action> ActionPtr;
```

ПРИЛОЖЕНИЕ В

Событие появления информации о типе работника

```
#pragma once

#include <utility>

#include "Model/action.h"

class AddWorkerTypeInfoAction : public Action {
 public:
  AddWorkerTypeInfoAction(WorkerType type, WorkerAbilities  abilities)
  : abilities_(std::move(abilities)), type_(type) {}

  void operator()(const SchedulerPtr& scheduler) override {
    scheduler->AddWorkerTypeInfo(type_, abilities_);
  }

 private:
  WorkerAbilities abilities_;
  WorkerType type_;
};
```

ПРИЛОЖЕНИЕ Г

Событие появления работника

```
#pragma once

#include "Model/scheduler.h"
#include "Model/action.h"

class AddWorkerAction : public Action {
 public:
  explicit AddWorkerAction(Worker worker) : worker_(worker) {}

  void operator()(const SchedulerPtr& scheduler) override {
    scheduler->AddWorker(worker_);
  }

 private:
  Worker worker_;
};
```

Событие появления задачи

```
#pragma once

#include "Model/action.h"
#include "Model/scheduler.h"

class AddTaskAction : public Action {
 public:
  explicit AddTaskAction(Task task,
                         WorkerCallback callback = [](Worker /*worker*/) {})
      : task_(task), callback_(std::move(callback)) {}
  ~AddTaskAction() override {
    callback_ = std::move([](Worker) {});
  }

  void operator()(const SchedulerPtr& scheduler) override {
    scheduler->AddTask(task_, callback_);
  }

  void SetCallback(WorkerCallback callback) {
    callback_ = std::move(callback);
  }

 private:
  Task task_;
  WorkerCallback callback_;
};
```

Метрика качества алгоритма

```cpp
#include "first_fail.h"

#include <cassert>

#include "Schedulers/Actions/add_task.h"

namespace FirstFail {

Percent Measure(const SchedulerPtr& scheduler, const TestCase& test) {
  size_t tasks_count = 0;
  for (auto& q : test.queries) {
    if (auto ptr = dynamic_cast<AddTaskAction*>(q.action.get()); ptr
        != nullptr) {
      ++tasks_count;
    }
  }
  size_t finished_tasks_count = 0;
  for (auto& q : test.queries) {
    bool is_task_query = false;
    bool is_finished = false;
    if (auto ptr = dynamic_cast<AddTaskAction*>(q.action.get()); ptr
        != nullptr) {
      is_task_query = true;
      ptr->SetCallback([&is_finished](Worker /*worker*/) {
        is_finished = true;
      });
    }
    q.action->operator()(scheduler);

    if (is_task_query) {
      dynamic_cast<AddTaskAction*>(q.action.get())->SetCallback([](Worker /*
    worker*/) {});
      if (!is_finished) {
        break;
      } else {
        ++finished_tasks_count;
      }
    }
  }
  assert(tasks_count > 0);
  return static_cast<Percent>(finished_tasks_count)
      / static_cast<Percent>(tasks_count);
}

} // namespace FirstFail
```

Код генерации тестов

```
TestCase TasksAfterWorkersGenerator::Generate() {
  // add worker types info
  std::vector<Query> preparation_queries(GeneratePreparationQueries());

  // add workers
  std::vector<Query> worker_queries;
  for (size_t i = 0; i < workers_count_; ++i) {
    worker_queries.push_back(Query{
        std::make_shared<AddWorkerAction>(GenerateWorker(static_cast<WorkerId>(i
    ))),
        Time{}
    });
  }

  // add tasks
  std::vector<Query> task_queries;
  size_t finished_tasks_count = 0;
  std::vector<WorkerId> executors(workers_count_);
  for (size_t i = 0; i < workers_count_; ++i) {
    auto id = static_cast<TaskId>(i);
    task_queries.push_back(Query{
        std::make_shared<AddTaskAction>(GenerateTask(id),
                                        [&finished_tasks_count, &executors, id](
                                            Worker worker) {
                                          ++finished_tasks_count;
                                          executors[id] = worker.id;
                                        }),
        Time{}
    });
  }

  auto is_feasible_sequence = [&](size_t using_workers_count) {
    finished_tasks_count = 0;
    SchedulerPtr offline_scheduler = std::make_shared<OfflineScheduler>(
        max_worker_type_,
        max_task_type_,
        task_queries.size());
    for (auto& q : preparation_queries) {
      q.action->operator()(offline_scheduler);
    }
    assert(using_workers_count <= worker_queries.size());
    for (size_t i = 0; i < using_workers_count; ++i) {
      worker_queries[i].action->operator()(offline_scheduler);
    }
    for (auto& q : task_queries) {
      q.action->operator()(offline_scheduler);
    }
    return finished_tasks_count == static_cast<int>(task_queries.size());
  };

  size_t l = 0; // impossible
  size_t r = workers_count_; // possible
  while (!is_feasible_sequence(r)) {
    r <<= 1;
    assert(r <= workers_count_ * 8);
```

```
    while (worker_queries.size() < r) {
      worker_queries.push_back(Query{
          std::make_shared<AddWorkerAction>(GenerateWorker(static_cast<WorkerId
    >(worker_queries.size()))),
          Time{}});
    }
  }
  while (l + 1 < r) {
    size_t mid = (r + l) >> 1;
    if (is_feasible_sequence(mid)) {
      r = mid;
    } else {
      l = mid;
    }
  }

  // clear redundant workers
  std::iota(executors.begin(), executors.end(), -1);
  finished_tasks_count = 0;
  is_feasible_sequence(r);
  std::unordered_set<WorkerId> used_workers;
  for (WorkerId id : executors) {
    used_workers.insert(id);
  }
  size_t insert_pos = 0;
  for (size_t i = 0; i < worker_queries.size(); ++i) {
    if (used_workers.contains(static_cast<WorkerId>(i))) {
      worker_queries[insert_pos++] = worker_queries[i];
    }
  }

  for (const auto& q : worker_queries) {
    if (auto action = dynamic_cast<AddTaskAction*>(q.action.get());
        action != nullptr) {
      action->SetCallback([](Worker) {});
    }
  }
  worker_queries.resize(insert_pos);
  assert(task_queries.size() == worker_queries.size());

  TestCase test;
  std::for_each(preparation_queries.begin(),
                preparation_queries.end(),
                [&](const Query& q) {
                  test.queries.push_back(q);
                });
  std::for_each(worker_queries.begin(),
                worker_queries.end(),
                [&](const Query& q) {
                  test.queries.push_back(q);
                });
  std::for_each(task_queries.begin(), task_queries.end(), [&](const Query& q) {
    test.queries.push_back(q);
  });

  return test;
}

Worker TasksAfterWorkersGenerator::GenerateWorker(WorkerId id) const {
  return Worker{
```

```
      static_cast<WorkerType>(workers_generator_.GetRandomEventIndex()), id};
}

Task TasksAfterWorkersGenerator::GenerateTask(TaskId id) const {
  auto type = static_cast<TaskTypeId>(tasks_generator_.GetRandomEventIndex());
  auto p = task_types_probabilities_[type];
  return Task(TaskType{type, p}, id);
}

std::vector<Query> TasksAfterWorkersGenerator::GeneratePreparationQueries()
    const {
  std::vector<Query> preparation_queries;
  preparation_queries.reserve(max_worker_type_);
  for (size_t i = 0; i < max_worker_type_; ++i) {
    auto type = static_cast<WorkerType>(i);
    auto abilities = worker_types_abilities_[i];
    preparation_queries.push_back(Query{
        std::make_shared<AddWorkerTypeInfoAction>(type, abilities),
        Time{}
    });
  }
  return preparation_queries;
}

WorkerType TasksAfterWorkersGenerator::GetMaxWorkerType() const {
  return max_worker_type_;
}

TaskTypeId TasksAfterWorkersGenerator::GetMaxTaskTypeId() const {
  return max_task_type_;
}
```