**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**МЕХАНИКО-МАТЕМАТИЧЕСКИЙ ФАКУЛЬТЕТ**

**Кафедра веб-технологий и компьютерного моделирования**

**ПОИСК РЕШЕНИЯ ЗАДАЧИ КОММИВОЯЖЕРА**

Курсовая работа

Сидоровой Алины Павловны

студентки 2 курса,   
специальность «веб-программирование и интернет-технологии»

Научный руководитель:

ассистент кафедры математической кибернетики В.Р. Харитонова

Минск, 2020

**ОГЛАВЛЕНИЕ**

[Введение 3](#_Toc40906783)

[Глава 1. Постановка задачи коммивояжера и алгоритмы для ее решения 5](#_Toc40906784)

[1.1 Постановка задачи коммивояжера 5](#_Toc40906785)

[1.2 Муравьиный алгоритм 6](#_Toc40906786)

[1.2.1 Понятие муравьиного алгоритма 6](#_Toc40906787)

[1.2.2 Принцип реализации 8](#_Toc40906788)

[1.2.3 Вариации муравьиного алгоритма 10](#_Toc40906789)

[1.3 Метод ветвей и границ 12](#_Toc40906790)

[1.3.1 Понятие метода ветвей и границ 12](#_Toc40906791)

[1.3.2 Принцип реализации метода ветвей и границ (алгоритма Литтла) 14](#_Toc40906792)

[Глава 2. Реализация выбранных алгоритмов и анализ полученных   
результатов 16](#_Toc40906793)

[2.1 Реализация муравьиного алгоритма 16](#_Toc40906794)

[2.2 Реализация метода ветвей и границ 20](#_Toc40906795)

[2.3 Анализ работы алгоритмов и сравнение полученных результатов 22](#_Toc40906796)

[Заключение 26](#_Toc40906797)

[Список использованных источников 27](#_Toc40906798)

[Приложение А](#_Toc40906799) [Код реализации муравьиного алгоритма и метода ветвей и границ на языке программирования Java 28](#_Toc40906800)

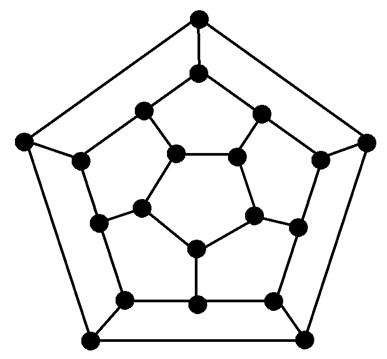
# ВВЕДЕНИЕ

**Комбинаторика** – раздел математики, посвященный решению задач выбора и расположения элементов некоторого, обычно конечного множества в соответствии с заданными правилами.

Каждое такое правило определяет способ построения некоторой конструкции из элементов исходного множества, называемой **комбинаторной конфигурацией**. Поэтому можно сказать, что целью комбинаторного анализа является изучение комбинаторных конфигураций. Это изучение включает в себя вопросы существования комбинаторных конфигураций, алгоритмы их построения, оптимизацию таких алгоритмов, а также решение задач перечисления, в частности определение числа конфигураций данного класса. Простейшим примером комбинаторных конфигураций являются перестановки, сочетания и размещения.

**Классические комбинаторные** **задачи** – это задачи выбора и расположения элементов конечного множества, имеющие в качестве исходной некоторую формулировку развлекательного содержания типа головоломок.

В 1859 г. знаменитый ирландский математик У. Гамильтон предложил детскую головоломку, в которой предлагалось совершить «кругосветное путешествие» по 20 городам, расположенных в различных частях земного шара. Игра сводилась к обходу по рёбрам всех вершин правильного додекаэдра (Рисунок1) и возврату в исходную точку при условии, что ни в одну из вершин нельзя заходить более одного раза. Пути, обладающие таким свойством, называются **гамильтоновыми циклами**.



(Рисунок 1)

Задача о гамильтоновых циклах в графе получила различные обобщения. Одно из этих обобщений – **задача коммивояжера**, имеющая ряд применений в исследовании операций, в частности при решении некоторых транспортно-логистических проблем.

**Коммивояжер** – это не свободно путешествующий турист, а деловой человек, ограниченный временными, денежными или какими-либо другими ресурсами. Гамильтонова задача может стать задачей о коммивояжере, если каждое из ребер снабдить числовой характеристикой. Это может быть километраж, время на дорогу, стоимость билета и т.д. Таким образом, условные характеристики дадут числовой ряд, элементы которого могут быть распределены между ребрами как угодно.

Целью данной курсовой работы является изучение понятия и особенностей задачи коммивояжера, а также поиск оптимального решения поставленной задачи на примере муравьиного алгоритма и метода ветвей и границ.

В ходе написания курсовой работы для достижения поставленной цели решены следующие задачи:

1. Поставлена задача коммивояжера;
2. Рассмотрены алгоритмы решения задачи: метод ветвей и границ, муравьиный алгоритм;
3. Проанализированы преимущества и недостатки использования предложенных алгоритмов;
4. Реализованы алгоритмы на языке программирования Java;
5. Проанализированы результаты алгоритмов для различных входных данных.

# ГЛАВА 1. ПОСТАНОВКА ЗАДАЧИ КОММИВОЯЖЕРА И АЛГОРИТМЫ ДЛЯ ЕЕ РЕШЕНИЯ

## 1.1 Постановка задачи коммивояжера

**Задача коммивояжера** является одной из самых известных задач комбинаторной оптимизации. Она заключается в отыскании самого выгодного маршрута, проходящего через указанные города хотя бы по одному разу. В условиях задачи указываются критерий выгодности маршрута (кратчайший, самый дешёвый критерий и т.д.) и соответствующие матрицы расстояний, стоимости и т.д. Как правило указывается, что маршрут должен проходить через каждый город только один раз, в таком случае выбор осуществляется среди гамильтоновых циклов [1].

Задача коммивояжера относится к числу трансвычислительных: уже при относительно небольшом числе городов (66 и более) она не может быть решена методом полного перебора вариантов даже самыми мощными компьютерами за время, меньшее нескольких миллиардов лет.

Дело в том, что количество возможных маршрутов равно n!, где n – количество городов. Известно, что n! – предпоследняя по скорости возрастания среди простейших функций (перед nn). К примеру, для 100 городов количество вариантов будет представляться 158-значным числом. Мощная ЭВМ, способная перебирать миллион вариантов в секунду, будет решать задачу на протяжении примерно 3⋅10144 лет. Не спасает ситуацию даже то, что для каждого варианта маршрута имеется 2n равноценных, отличающихся только выбором начального пункта и направлением обхода. Перебор с учётом этого наблюдения сокращается незначительно — до количества вариантов, указанных в формуле (1).

(1)

Задача коммивояжера широко применяется при разработке программного обеспечения. Она является упрощенной моделью для многих других задач дискретной оптимизации, а также часто является подзадачей. В своей области (оптимизации дискретных задач) она служит своеобразным катализатором, стимулирующим разработку наиболее эффективных методов, алгоритмов и способов их машинной реализации.

Задача коммивояжера формулируется следующим образом: на плоскости (в пространстве) расположены N городов, заданы расстояния между каждой парой городов. Требуется найти маршрут минимальной длины с посещением каждого города ровно один раз и возвращением в исходную точку.

В терминах теории графов задачу можно сформулировать так: требуется найти гамильтонов цикл наименьшей стоимости во взвешенном полном графе мощности N, т.е., выйдя из стартовой вершины, посетить каждую вершину графа ровно один раз и вернуться в начальную по кратчайшему пути.

Общая постановка задачи коммивояжера и большинство её частных случаев, относится к классу NP-сложных задач, т.к. для нее до сих пор не существует эффективного алгоритма для нахождения точного решения.

Существует несколько **частных случаев** общей постановки задачи:

* **Геометрическая** задача коммивояжера (также называемая планарной или евклидовой, когда матрица расстояний отражает расстояния между точками на плоскости);
* **Треугольная** задача коммивояжёра (когда на матрице стоимостей выполняется неравенство треугольника);
* **Симметричная** задача коммивояжера, в которой расстояния заданы между любыми двумя городами и матрица расстояний симметрична: Dji =  Dij;
* **Асимметричная** задача коммивояжера допускает несимметричность матрицы Dji ≠ Dij. В ещё более общем случае, пути между некоторыми городами могут отсутствовать (т.е. иметь бесконечную длину).

Существует множество методов решения задачи Коммивояжера, например, «жадный алгоритм», «деревянный алгоритм», «алгоритм Дейкстры», «генетический алгоритм» и другие. В данной курсовой работе были реализованы «метод ветвей и границ» и «муравьиный алгоритм».

## 1.2 Муравьиный алгоритм

### 1.2.1 Понятие муравьиного алгоритма

**Муравьиные алгоритмы** – это семейство приближенных полиномиальных алгоритмов для решения различных сложных оптимизационных задач, в том числе задачи коммивояжера и аналогичных задач поиска маршрутов на графах. Суть подхода заключается в анализе и использовании модели поведения муравьёв, которые ищут пути от колонии к источнику питания, и представляет собой метаэвристическую оптимизацию. Первая версия алгоритма, предложенная итальянским ученым, доктором наук Марко Дориго в 1992 году, была направлена на поиск оптимального пути в графе.

Принципы поведения муравьев выдержали испытания на протяжении 100 миллионов лет — именно столько времени назад муравьи «колонизировали» Землю. Муравьи относятся к социальным насекомым, живущим внутри некоторого коллектива – колонии.

Основу «социального» поведения муравьев составляет самоорганизация — множество динамических механизмов, обеспечивающих достижение системой глобальной цели в результате низкоуровневого взаимодействия ее элементов. Принципиальной особенностью такого взаимодействия является использование элементами системы только локальной информации. При этом исключается любое централизованное управление и обращение к глобальному образу, представляющему систему во внешнем мире. Самоорганизация является результатом взаимодействия следующих четырех компонентов:

* Случайность**;**
* Многократность;
* Положительная обратная связь;
* Отрицательная обратная связь.

В природе встречается такое явление, как **мельница (карусель) смерти**. Явление состоит в том, что один или небольшая группа муравьев начинает без видимой причины бегать по замкнутому кругу, постепенно вовлекая в свой бесконечный цикл все больше и больше других муравьев. Муравьи продолжают свой бег до тех пор, пока не падают замертво, и муравьиный круг продолжает своё вращение до полного истощения, оставляя за собой полчища погибших.

В 1921 году американский путешественник Уильям Биб в своей книге «Край джунглей» описал виденный им в Гайане круг муравьев-эцитонов диаметром около 365 метров, в котором каждый из муравьёв совершал полный цикл за 2,5 часа. Эта карусель просуществовала 2 дня, усеивая почву под собой мертвыми телами, пока небольшая группа рабочих муравьев не отделилась от общего движения и не увела за собой оставшихся в живых в лес.

Муравьиные алгоритмы основаны на следующей особенности поведения муравьев в природе.

При поиске путей к источникам пищи муравьи (первоначально) ходят в случайном порядке, помечая пройденный путь специальным веществом – феромоном. Количество откладываемого муравьем феромона на ребре графа обратно пропорционально длине маршрута. Чем короче маршрут, тем больше феромона будет отложено на соответствующих ребрах графа и тем больше муравьев будет использовать их при построении своих маршрутов. Отложенный на ребрах феромон выступает как усилитель, он позволяет хорошим маршрутам сохраняться в глобальной памяти муравейника. Эти маршруты могут быть улучшены на последующих итерациях алгоритма.

Если другие муравьи находят такие тропы (пути, маршруты), они, вероятнее всего, пойдут по ним, причем чем больше концентрация феромона на пути, тем больше вероятность. Вместо того, чтобы отслеживать цепочку, муравьи укрепляют её при возвращении, если в конечном счете находят источник питания.

Со временем феромонная тропа начинает испаряться, тем самым уменьшая свою привлекательность. Чем больше времени требуется для прохождения пути до цели и обратно, тем сильнее испарится феромонная тропа. На коротком пути прохождение будет более быстрым и как следствие, плотность феромонов остается высокой.

Испарение феромонов также имеет свойство избежания стремления к локально-оптимальному решению. Если бы феромоны не испарялись, то путь, выбранный первым муравьем, был бы самым привлекательным. В этом случае, исследования пространственных решений были бы ограниченными.

Таким образом, когда один муравей находит (например, короткий) путь от колонии до источника пищи, другие муравьи, с большей вероятностью пойдут по этому пути, и положительные отзывы в конечном счете приведут всех муравьев к одному, кратчайшему, пути.

Можно заметить, что феромон играет роль положительной обратной связи в системе – чем больше муравьев двигается по помеченному пути, тем больше он становится привлекательным для других муравьев. В результате, через некоторое время большая часть муравьев будет двигаться от муравейника до найденного источника пищи по одному и тому же пути.

В то же время моделирование испарения феромона является отрицательной обратной связью и гарантирует нам, что найденное локально оптимальное решение не будет единственным, т.е. муравьи будут искать и другие пути. Если мы моделируем процесс такого поведения на некотором графе, ребра которого представляют собой возможные пути перемещения муравьев, в течение определённого времени, то наиболее обогащенный феромоном путь по ребрам этого графа и будет являться решением задачи, полученным с помощью муравьиного алгоритма.

### 1.2.2 Принцип реализации

Работа алгоритма начинается с создания колонии из N муравьев и помещения их в одну из вершин графа, которую будем называть стартовой. Все ребра графа помечаются одинаковым значением феромона, например, 𝜏ij = 1, задается матрица расстояний с элементами dij > 0 – длина ребра ij (расстояние между вершинами i и j), причем элементы dij (где i = j) полагаются равными 0 или ∞ (бесконечность). После этого муравьи начинают перемещаться по графу. Каждый муравей в процессе движения хранит список запретов, так называемый «список табу», в котором содержатся вершины, в которых данный муравей уже побывал. На каждом шаге список запретов пополняется новым узлом (вершиной), а перед новой итерацией алгоритма – то есть перед тем, как новый муравей проходит путь – он опустошается.

Кроме списка запретов, при выборе узла для перехода муравей опирается на «привлекательность» ребер, которые он может пройти. Она зависит,   
во-первых, от расстояния между вершинами (то есть от веса ребра), а во-вторых, от следов феромонов, оставленных на ребре прошедшими по нему ранее муравьями. В отличие от весов ребер, которые являются постоянными, следы феромонов обновляются на каждой итерации алгоритма: как и в природе, со временем следы испаряются, а проходящие муравьи, напротив, усиливают их.

Пусть k-й муравей находится в i-й вершине, а вершина j – одна из вершин, доступных для перехода, т.е. не содержится в списке запретов Xk. Тогда вероятность перехода k-го муравья в вершину j определяется по формуле (2)

(2)

где 𝜂ij = 1/dij, 𝜏ij – текущее количество феромона на ребре ij, а α и β — два регулируемых параметра, задающие веса следа феромона и видимости при выборе маршрута. При α = 0 будет выбран ближайший город, что соответствует жадному алгоритму в классической теории оптимизации. Если β = 0, тогда работает лишь феромонное усиление, что влечет за собой быстрое вырождение маршрутов к одному субоптимальному решению.

Собственно выбор города (вершины) осуществляется по принципу «колеса рулетки»: каждый город на ней имеет свой сектор с площадью, пропорциональной вероятности (2). Для выбора города нужно сгенерировать случайное число, и определить сектор, в который оно попадет.

Если у k-го муравья не осталось непосещенных вершин, он возвращается в стартовую. После возвращения в стартовую вершину каждый муравей обновляет только те ребра, по которым ходил. Концентрация феромона на ребре ij пересчитывается по формуле (3), где 𝛥𝜏ijk считается по формуле (4),   
𝜌 – коэффициент испарения (от 0 до 1), Lk – длина пути k-го муравья.

(3)  
 (4)

Таким образом, чем короче путь муравья, тем больше феромона он оставит на ребрах, по которым он ходил. Описанные действия (перемещение по графу и обновление феромонов) представляют собой одну итерацию муравьиного алгоритма. Итерации повторяются до тех пор, пока не будет выполнено одно из условий: исчерпано число итераций, достигнута необходимая точность, получен единственный путь (алгоритм сошелся к некоторому решению).

### 1.2.3 Вариации муравьиного алгоритма

Результаты первых экспериментов с применением муравьиного алгоритма для решения задачи коммивояжера были многообещающими, однако далеко не лучшими по сравнению с уже существовавшими методами. Однако простота классического муравьиного алгоритма (называемого «муравьиной системой») оставляла возможности для доработо. В основном, эти усовершенствования связаны с большим использованием истории поиска и более тщательным исследованием областей вокруг уже найденных удачных решений. Ниже рассмотрены наиболее примечательные из модификаций.

**Элитарная муравьиная система** (Elitist Ant System)

Из общего числа муравьёв выделяются так называемые «элитные муравьи». Опыт показывает, что, проходя ребра, входящие в короткие пути, муравьи с большей вероятностью будут находить еще более короткие пути. Таким образом, эффективной стратегией является искусственное увеличение уровня феромонов на самых удачных маршрутах. Для этого на каждой итерации алгоритма каждый из элитных муравьев проходит путь, являющийся самым коротким из найденных на данный момент. В такой системе количество элитных муравьёв является дополнительным параметром, требующим определения. Так, для слишком большого числа элитных муравьёв алгоритм может «застрять» на локальных экстремумах.

Эксперименты показывают, что, до определенного уровня, увеличение числа элитных муравьев является достаточно эффективным, позволяя значительно сократить число итераций алгоритма. Однако, если число элитных муравьев слишком велико, то алгоритм достаточно быстро находит субоптимальное решение и застревает в нем. Как и другие изменяемые параметры, оптимальное число элитных муравьев следует определять опытным путем.

**Ant-Q**

Лука Гамбарделла и Марко Дориго опубликовали в 1995 году работу, в которой они представили муравьиный алгоритм, получивший свое название по аналогии с методом машинного обучения Q-learning. В основе алгоритма лежит идея о том, что муравьиную систему можно рассматривать как систему обучения с подкреплением. Ant-Q усиливает эту аналогию, заимствуя многие идеи из Q - обучения.

Алгоритм хранит Q-таблицу, которая сопоставляет каждому из ребер величину, определяющую «полезность» перехода по этому ребру. Эта таблица изменяется в процессе работы алгоритма – то есть обучения системы. Значение полезности перехода по ребру вычисляется исходя из значений полезностей перехода по следующим ребрам в результате предварительного определения возможных следующих состояний. После каждой итерации полезности обновляются исходя из длин путей, в состав которых были включены соответствующие ребра.

**Ant Colony System**

В 1997 году те же исследователи опубликовали работу, посвященную еще одному разработанному ими муравьиному алгоритму. Для повышения эффективности по сравнению с классическим алгоритмом, ими были введены три основных изменения:

1. Уровень феромонов на ребрах обновляется не только в конце очередной итерации, но и при каждом переходе муравьев из узла в узел.
2. В конце итерации уровень феромонов повышается только на кратчайшем из найденных путей.
3. Алгоритм использует измененное правило перехода: либо, с определенной долей вероятности, муравей выбирает лучшее – в соответствии с длиной и уровнем феромонов – ребро, либо производит выбор так же, как и в классическом алгоритме.

**Max-min муравьиная система** (MMAS)

В том же году Томас Штютцле и Хольгер Хус [9] предложили муравьиный алгоритм, в котором добавляются граничные условия на количество феромонов (τmin, τmax). Феромоны откладываются только на глобально лучших или лучших в итерации путях. Такое большое внимание к локальным оптимумам компенсируется вводом ограничений на максимальную и минимальную концентрацию феромонов на ребрах, которые крайне эффективно защищают алгоритм от преждевременной сходимости к субоптимальным решениям.

На этапе инициализации, концентрация феромонов на всех ребрах устанавливается равной максимальной. После каждой итерации алгоритма только один муравей оставляет за собой след – либо наиболее успешный на данной итерации, либо элитный. Этим достигается, с одной стороны, более тщательное исследование области поиска, с другой – его ускорение.

**Ранговая муравьиная система** (ASrank)

Бернд Бульнхаймер, Ричард Хартл и Кристина Штраусс разработали модификацию классического муравьиного алгоритма, в котором в конце каждой итерации муравьи ранжируются в соответствие с длинами пройденных ими путей. Количество феромонов, оставляемого муравьем на ребрах, таким образом, назначается пропорционально его позиции. Кроме того, для более тщательного исследования окрестностей уже найденных удачных решений, алгоритм использует элитных муравьев.

**Длительная ортогональная колония муравьёв** (COAC)

Механизм отложения феромонов COAC позволяет муравьям искать решения совместно и эффективно. Используя ортогональный метод, муравьи в выполнимой области могут исследовать их выбранные области быстро и эффективно, с расширенной способностью глобального поиска и точностью.

Ортогональный метод планирования и адаптивный метод регулирования радиуса могут также быть расширены на другие алгоритмы оптимизации для получения более широких преимуществ в решении практических проблем.

## 1.3 Метод ветвей и границ

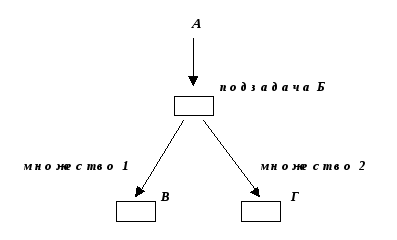
### 1.3.1 Понятие метода ветвей и границ

**Метод ветвей и границ** — общий комбинаторный алгоритмический метод для нахождения оптимальных решений различных задач оптимизации, особенно дискретной и комбинаторной оптимизации. По существу, метод является развитием метода полного перебора с отсевом подмножеств множества допустимых решений, заведомо не содержащих оптимальных решений. Метод был впервые предложен Алисой Ленд и Элисон Дойг в 1960 г. для решения задач линейного программирования [10].

В этом методе определяется некая функция, как весовая числовая функция для различных вариантов решения задачи, и вычисляются численные значения этой функции для различных вариантов решения. Цель алгоритма — найти конфигурацию, на которой эта функция достигает максимального или минималь­ного значения. Сам же метод отыскания и построения такой функции получил название метод ветвей и границ. Такое название возникло из-за того, что в нем используется ветвление и вычисляются границы.

Рассмотрим более подробно что понимается под ветвлением и что понимается под границей.

**Ветвление** - это процесс разбиения всех вариантов решения задачи на два множества, таким образом, что какая-либо конкретная подзадача принадлежит только одному из этих множеств. В этом случае говорят, что ветвление происходит на этой подзадаче. Если пользоваться терминологией древовидной структуры, то говорят, что ветвление происходит на определенной ветви древовидной структуры.



(Рисунок 2)

На рисунке 2 показана древовидная структура. Ветвление производится в точке Б при рассмотрении подзадачи Б, дальнейшее решение задачи пойдет либо по пути Б-В, либо по пути Б-Г. Ветвление происходит в точке Б и разбивается на два подмножества. Сущность ветвления заключается в том, что какая-то конкретная подзадача Б входит в множество 1 и не входит в множество 2 вариантов решения задачи.

Следующим шагом является вычисление нижних границ, т.е. вычисление весовой функции для всей вариантов множества 1 или множества 2. Для каждого множества 1 или 2 существует конечное число вариантов решения, в которых весовая функция принимает определенные значения. Очевидно, что для каждого из множеств весовой функции имеется определенная граница, нижняя или верхняя, по которой происходит оценка дальнейшего пути решения задачи.

Допустимое решение, дающее наименьшую верхнюю оценку, называют **рекордом**. Если оценка снизу целевой функции на данном подмножестве (например, множество 1) не меньше оценки сверху, то рассматриваемое подмножество не содержит решения лучше рекорда и может быть отброшено, т.е. путь Б-В может быть исключен из дальнейшего рассмотрения (правило отсева). Если значение целевой функции на очередном решении меньше рекордного, то происходит смена рекорда.

Будем говорить, что подмножество решений просмотрено, если установлено, что оно не содержит решения лучше рекорда. Если просмотрены все элементы разбиения, алгоритм завершает работу, а текущий рекорд является оптимальным решением. В противном случае среди непросмотренных элементов разбиения выбирается множество, которое в определенном смысле является перспективным. Оно подвергается разбиению (ветвлению). Новые подмножества анализируются по описанной выше схеме. Процесс продолжается до тех пор, пока не будут просмотрены все элементы разбиения.

Частным случаем применения метода «ветвей и границ**»** для конкретной задачи – задачи коммивояжера, является **алгоритм Литтла**, предложенный в 1963 году группой авторов Дж. Литлом, К. Мурти, Д. Суини, К. Кэролом [11]. Общая идея состоит в том, что нужно разделить огромное число перебираемых вариантов на классы и получить оценки (снизу – в задаче минимизации, сверху – в задаче максимизации) для этих классов, чтобы иметь возможность отбрасывать варианты не по одному, а целыми классами. Трудность состоит в том, чтобы найти такое разделение на классы (ветви) и такие оценки (границы), чтобы процедура была эффективной.

### 1.3.2 Принцип реализации метода ветвей и границ (алгоритма Литтла)

Для начала, как и в предыдущем алгоритме, задается матрица C весов дуг (расстояний) для полного взвешенного ориентированного (или неориентированного) графа с элементами cij > 0 – вес дуги, полагая cij = ∞ при i = j.

При построении оценки снизу на каждом этапе работы алгоритма матрица расстояний подвергается такому преобразованию с трудоемкостью O(n2), чтобы в каждой её строке и каждом столбце появился хотя бы один нуль. Более точную оценку снизу можно получать, решая задачу о назначениях на матрице расстояний за время O(n3), при этом улучшается эффективность отсечений в дереве решений.

Для вычисления оценки длины частично построенного маршрута снизу в алгоритме Литтла применяется следующее преобразование матрицы. Сначала из каждой строки вычитается её наименьший элемент, в результате в каждой строке образуется один (или больше) нулей. Затем из каждого столбца вычитается его наименьший элемент. Это преобразование основано на том факте, что если из любого столбца или строки матрицы вычесть константу, то стоимость оптимального маршрута уменьшается на величину этой константы, а сам маршрут остается прежним. Сумма всех вычтенных при этом величин и будет оценкой снизу для всех вариантов маршрута, строящегося по данной матрице.

Следующее действие — выбор некоторого ребра графа (элемента матрицы). Выбор ребра на очередном шаге производится таким образом, чтобы оптимальный вариант маршрута содержал выбранное ребро с наибольшей вероятностью. Для этого для каждого нулевого элемента cij высчитывается «штраф за неиспользование». Он равен сумме минимальных значений матрицы в i-й строке и j-м столбце, не включая сам элемент cij. Окончательно выбирается ребро (элемент) с максимальным «штрафом» (если таких элементов несколько, берется любой из них).

Этот выбор определяет разбиение множества всех маршрутов на два подмножества: подмножество {i, j}, содержащее ребро cij, и подмножество {i, j}, не содержащее ребро cij. Выбранное ребро включается в маршрут для вариантов маршрута 1-го подмножества и исключается из всех вариантов маршрута 2-го подмножества. В результате оценка снизу для всех вариантов маршрута 2-го подмножества увеличивается на вес минимального ребра в строке или столбце, без учета выбранного ребра.

Для каждого полученного множества создается своя матрица расстояний. Матрица множества, содержащего выбранное ребро, получается путем приравнивания всех элементов i-й строки и j-го столбца к ∞ (т.е. удалением i-й строки и j-го столбца), а также заменой элемента cji на ∞, чтобы избежать преждевременного зацикливания программы.

В матрице множества, не содержащего выбранное ребро, элемент cij заменяется на ∞, т.к. не входит в результирующий маршрут для данного ответвления.

Полученные матрицы подвергаются аналогичному преобразованию с выбором ребра и т.д. Таким образом, строящееся при этом дерево решений получается двоичным (бинарным), каждому его узлу соответствует своя матрица расстояний.

Алгоритм продолжается до тех пор, пока в подмножестве маршрутов с наименьшей оценкой не останется всего один маршрут, его оценка и есть вес минимального гамильтонового цикла, т.е. решения; путь к корню дерева ветвления восстанавливает сам цикл.

Модификация алгоритма Литтла состоит в том, что при обработке алгоритмом очередного узла в дереве решений матрица расстояний подвергается дополнительному преобразованию. При вычислении оценки снизу стоимости вариантов маршрутов, строящихся из текущего узла, в матрице расстояний в каждой строке и каждом столбце появляется не менее чем по одному нулю.

Дополнительное преобразование состоит в том, что в матрице отмечается такая группа строк, в которых имеется ровно по одному нулевому элементу и все эти нули находятся в одном и том же столбце. Если такая группа, состоящая хотя бы из двух строк, найдена, то минимальный ненулевой элемент (пусть его величина равна a) в этих строках вычитается из всех строк группы и добавляется к тому столбцу, в котором находятся нули. В результате ранее полученные нули в этой группе строк сохраняются и добавляется, по крайней мере, один новый нуль.

Оптимальный маршрут при этом не изменяется, а его стоимость уменьшается на величину a(p−1), где p — количество строк в группе. На такую же величину увеличивается оценка снизу стоимости вариантов маршрутов, строящихся из текущего узла. Матрица просматривается до тех пор, пока не будут обработаны все такие группы строк. Далее аналогичным образом в матрице просматриваются столбцы, которые, если будут найдены подобные группы, подвергаются аналогичному преобразованию.

# ГЛАВА 2. РЕАЛИЗАЦИЯ ВЫБРАННЫХ АЛГОРИТМОВ И АНАЛИЗ ПОЛУЧЕННЫХ РЕЗУЛЬТАТОВ

## 2.1 Реализация муравьиного алгоритма

В ходе курсовой работы я реализовала два алгоритма: муравьиный алгоритм и метод ветвей и границ на языке программирования Java.

Общими входными данными для обоих алгоритмов являются случайно сгенерированные симметричные матрицы расстояний *distance* с бесконечностью по диагонали, т.к. переход из вершины в саму себя неосуществим.

Для наглядности рассмотрим матрицы размерности 5 и 8 в таблицах 1 и 2.

(Таблица 1 – матрица расстояний для графа порядка 5)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Номера вершин | 1 | 2 | 3 | 4 | 5 |
| 1 | ∞ | 20 | 9 | 5 | 14 |
| 2 | 20 | ∞ | 15 | 4 | 27 |
| 3 | 9 | 15 | ∞ | 7 | 30 |
| 4 | 5 | 4 | 7 | ∞ | 24 |
| 5 | 14 | 27 | 30 | 24 | ∞ |

(Таблица 2 – матрица расстояний для графа порядка 8)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Номера вершин | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 1 | ∞ | 18 | 3 | 23 | 27 | 5 | 8 | 8 |
| 2 | 18 | ∞ | 19 | 8 | 9 | 5 | 23 | 14 |
| 3 | 3 | 19 | ∞ | 28 | 19 | 8 | 20 | 19 |
| 4 | 23 | 8 | 28 | ∞ | 28 | 15 | 25 | 26 |
| 5 | 27 | 9 | 19 | 28 | ∞ | 7 | 24 | 20 |
| 6 | 5 | 5 | 8 | 15 | 7 | ∞ | 22 | 5 |
| 7 | 8 | 23 | 20 | 25 | 24 | 22 | ∞ | 29 |
| 8 | 8 | 14 | 19 | 26 | 20 | 5 | 29 | ∞ |

Также, как указывалось выше, для данного алгоритма задается матрица феромонов *pheromone* с элементами, равными 1, и нулями по главной диагонали.

*Pheromone* и *distance* инициализируются в конструкторе класса *Creator* от переданного количества вершин n:

public Creator(int n) {  
 this.distance = createDistanceMatrix(n);  
 this.pheromone = createPheromoneMatrix(n);  
}

public ArrayList<ArrayList<Double>> createDistanceMatrix(int n) {  
 ArrayList<ArrayList<Double>> distance = new ArrayList<>();  
 for (int i = 0; i < n; i++) {  
 distance.add(new ArrayList<>());  
 }  
 for (int i = 0; i < n; i++) {  
 for (int j = i; j < n; j++) {  
 if (i == j) {  
 distance.get(i).add(Double.*POSITIVE\_INFINITY*);  
 } else {  
 distance.get(i).add((double) Math.*round*(Math.*random*() \* 30 + 1));  
 distance.get(j).add(distance.get(i).get(j));  
 }  
 }  
 }  
 return distance;  
}  
  
public ArrayList<ArrayList<Double>> createPheromoneMatrix(int n) {  
 ArrayList<ArrayList<Double>> pheromone = new ArrayList<>(n);  
 for (int i = 0; i < n; i++) {  
 pheromone.add(new ArrayList<>());  
 }  
 for (int i = 0; i < n; i++) {  
 for (int j = 0; j < n; j++) {  
 if (i == j) {  
 pheromone.get(i).add(0.0);  
 } else {  
 pheromone.get(i).add(1.0);  
 }  
 }  
 }  
 return pheromone;  
}

Для реализации алгоритма задаются несколько констант:

* *RHO* = 0.2 – коэффициент испарения**;**
* *ALPHA* = 1, *BETTA* = 1 – постоянные, регулирующие степень влияния количества феромона на ребре и его длины соответственно;
* *ANT\_AMOUNT* = 20 – количество муравьев колонии, т.е. количество итераций алгоритма;

В ходе алгоритма каждый муравей хранит список непосещенных вершин *unvisited*, свой путь *path* и длину пути *length*.

Так как стартовой точкой в моей реализации является вершина 1, список *unvisited* по умолчанию заполняется вершинами от 2 до n, где n – количество вершин графа, а в путь добавляется единственная вершина 1.

Основным методом (функцией) реализованного алгоритма является метод *findShortestWay* класса *WayType*.

На каждой итерации цикла создается новый муравей, затем в методе *createPath* генерится его путь, и в методе *updatePheromone* обновляется количество феромона на ребрах.

Если длина найденного пути на конкретной итерации меньше длины текущего лучшего маршрута, лучший путь обновляется, в *length* записывается его длина, и запоминается «лучший» муравей, который нашел данный путь. Метод возвращает «лучшего» муравья.

public Ant findShortestWay(Creator creator, int n) {  
 Update update = new Update();  
 double length = 0;  
 Ant bestAnt = new Ant();  
 for (int k = 0; k < *ANT\_AMOUNT*; k++) {  
 Ant ant = new Ant();  
 WayType wayType = new WayType(n);  
 creator.createPath(ant, wayType, creator);  
 update.updatePheromone(ant, creator, n);  
 if (length == 0) {  
 bestAnt = ant;  
 length = ant.getLength();  
 } else if (ant.getLength() < length) {  
 length = ant.getLength();  
 bestAnt = ant;  
 }  
 }  
 return bestAnt;  
}

Метод *createPath* создает путь, основываясь на выборе следующей вершины в методе *chooseVertex* и считает длину полученного маршрута. Метод *chooseVertex*, в свою очередь, разбивает отрезок [0; 100] на секторы с площадью, пропорциональной вероятности, вычисленной в методе *probability* по формуле (2) из предыдущей главы.

public void createPath(Ant ant, WayType wayType, Creator creator) {  
 while (wayType.getUnvisited().size() != 0) {  
 ant.chooseVertex(creator, ant, wayType);  
 }  
 ant.getPath().add(1);  
  
 for (int i = 0; i < ant.getPath().size() - 1; i++) {  
 ant.setLength(ant.getLength() + distance.get(ant.getPath().get(i) - 1).get(ant.getPath().get(i + 1) - 1));  
 }  
}

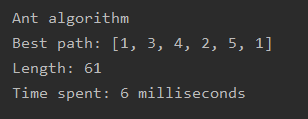
public void chooseVertex(Creator creator, Ant ant, WayType wayType) {  
 int prob = (int) ( 1 + Math.*random*() \* 99 );  
 ArrayList<Integer> P = creator.createProbabilityArray(ant, wayType);  
 ArrayList<Integer> unvisited = wayType.getUnvisited();  
  
 for(int i = 0; i < P.size() - 1; i++) {  
 if(prob >= P.get(i) && prob <= P.get(i + 1)) {  
 this.path.add(unvisited.get(i));  
 unvisited.remove(i);  
 break;  
 }  
 }  
}

public int probability(Ant ant, int to, ArrayList<ArrayList<Double>> pheromone, ArrayList<ArrayList<Double>> distance) {  
 int from = ant.getPath().get(ant.getPath().size() - 1);  
 double sum = 0;  
 for (int item : unvisited) {  
 sum += Math.*pow*(pheromone.get(from - 1).get(item - 1), *ALPHA*) \* Math.*pow*(1.0 / distance.get(from - 1).get(item - 1), *BETTA*);  
 }  
 double numerator = Math.*pow*(pheromone.get(from - 1).get(to - 1), *ALPHA*) \* Math.*pow*(1.0 / distance.get(from - 1).get(to - 1), *BETTA*);  
 double Pij = numerator / sum;  
 return (int) Math.*round*(Pij \* 100);  
}

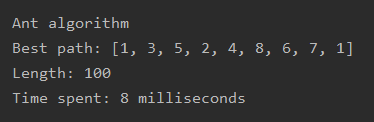
Метод *updatePheromone* реализует формулы (3) и (4) из предыдущей главы:

public void updatePheromone(Ant ant, Creator creator, int n) {  
 double delta = 1.0 / ant.getLength();  
 for (int i = 0; i < n; i++) {  
 for (int j = i; j < n; j++) {  
 if (i != j){  
 creator.getPheromone().get(i).set(j, (1 - *RHO*) \* creator.getPheromone().get(i).get(j));  
 creator.getPheromone().get(j).set(i, creator.getPheromone().get(i).get(j));  
 }  
 }  
 }  
 ArrayList<Integer> path = ant.getPath();  
 for (int i = 0; i < path.size() - 1; i++) {  
 creator.getPheromone().get(path.get(i) - 1).set(path.get(i + 1) - 1, creator.getPheromone().get(path.get(i) - 1).get(path.get(i + 1) - 1) + delta);  
 creator.getPheromone().get(path.get(i + 1) - 1).set(path.get(i) - 1, creator.getPheromone().get(path.get(i) - 1).get(path.get(i + 1) - 1));  
 }  
}

Для выше указанных входных данных получился результат, представленный на рисунках 1 и 2.



(Рисунок 1)



(Рисунок 2)

## 2.2 Реализация метода ветвей и границ

В качестве входных данных используются матрицы расстояний, представленные в таблицах 1 и 2 предыдущего пункта.

Главным классом является класс *BranchAndBound* с полями:

* *matrix* – матрица расстояний**;**
* *record* – лучший маршрут. По умолчанию задается упорядоченным множеством вершин 1, 2, …, n;
* *recordWeight* – длина лучшего маршрута.

Как и в муравьином алгоритме, поиск оптимального пути ведется из вершины 1, т.е. существует n - 1 вариантов для выбора следующей вершины, которые перебираются в цикле класса *Main*, и для каждого варианта вызывается метод *matrixUpdate* для поиска оптимального маршрута.

Таким образом начальная матрица представляет собой матрицу с бесконечными элементами 1-й строки и i-го столбца (i = 2, 3, …, n) и элементом на позиции (i, 1), что означает, что ребро (1, i) обязательно входит в проверяемый маршрут.

В главном методе *matrixUpdate* класса *BranchAndBound* входными параметрами являются: текущая матрица расстояний *matrix*, которая постоянно изменяется в ходе алгоритма, текущий маршрут *path* и текущая нижняя граница *lowerBound*, которая в итоге будет соответствовать длине найденного пути.

Для начала задаем (или обновляем) нижнюю границу поиска с помощью функции *findMin* по принципу, описанному в пункте 1.3.2. Далее, если нижняя граница больше, чем длина лучшего маршрута, «обрезаем» эту ветвь поиска, т.к. граница, а, следовательно, и длина строящегося маршрута, будет только увеличиваться, либо оставаться неизменной.

Если текущий маршрут содержит все вершины, а значит алгоритм дошел до конца одной из ветвей поиска, обновляем текущий лучший путь *record* и его длину *recordWeight*.

В противном случае с помощью функции *findZeros* выбираем ребро, от которого пойдет следующее ветвление. В конструкторе класса *MatrixHandler* создаются новые матрицы расстояний *Y* и *N* для двух множеств: множества, содержащего выбранное ребро, и множества не содержащего его, соответственно. К маршруту, строящемуся для множества с матрицей *Y* добавляется выбранное ребро.

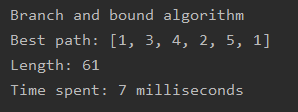
Далее метод рекурсивно вызывается для каждого созданного множества, и для каждого из них данная процедура продолжается до тех пор, пока не будет найден один из возможных оптимальных путей, или ветвь поиска не «отсечется».

public void matrixUpdate(ArrayList<ArrayList<Double>> matrix, ArrayList<Integer> path, double lowerBound) {  
 ArrayList<Double> minRow = new ArrayList<>();  
 ArrayList<Double> minColumn = new ArrayList<>();  
 lowerBound = findMin(matrix, minRow, lowerBound);  
 transpose(matrix);  
 lowerBound = findMin(matrix, minColumn, lowerBound);  
 transpose(matrix);  
  
 // cut branch if it is longer, than record  
 if (lowerBound >= recordWeight) {  
 return;  
 }  
  
 if (path.size() == record.size()) {  
 record.clear();  
 record.addAll(path);  
 this.recordWeight = lowerBound;  
 return;  
 }  
  
 if (findZeros(matrix, path).size() != 0) {  
 ArrayList<Integer> currentZero = findZeros(matrix, path).get(0);  
 MatrixHandler handler = new MatrixHandler(matrix, currentZero.get(0), currentZero.get(1));  
  
 ArrayList<Integer> newPath = new ArrayList<>(path);  
 newPath.add(currentZero.get(1) + 1);  
  
 BranchAndBound Y = new BranchAndBound(handler.getY(), newPath, lowerBound);  
 matrixUpdate(Y.matrix, Y.record, Y.recordWeight);  
  
 BranchAndBound N = new BranchAndBound(handler.getN(), path, lowerBound);  
 matrixUpdate(N.matrix, N.record, N.recordWeight);  
 }  
}

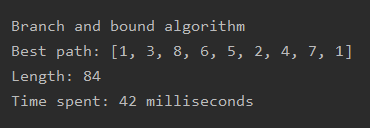
Создание матриц для полученных множеств описывается функциями *setY* и *setN*. Матрица *Y* получается путем замены всех элементов строки *row* и столбца*col* и элемента (*col, row*) на бесконечность. Матрица *N* получается из исходной путем замены элемента (*row, col*) на бесконечность.

public void setY(ArrayList<ArrayList<Double>> matrix, int row, int col) {  
 this.Y = new ArrayList<>();  
 for (int i = 0; i < matrix.size(); i++) {  
 this.Y.add(new ArrayList<>());  
 for (int j = 0; j < matrix.size(); j++) {  
 if (i == row || j == col) {  
 this.Y.get(i).add(Double.*POSITIVE\_INFINITY*);  
 } else {  
 this.Y.get(i).add(matrix.get(i).get(j));  
 }  
 }  
 }  
 this.Y.get(col).set(row, Double.*POSITIVE\_INFINITY*);  
}  
  
public void setN(ArrayList<ArrayList<Double>> matrix, int row, int col) {  
 this.N = new ArrayList<>();  
 for (int i = 0; i < matrix.size(); i++) {  
 this.N.add(new ArrayList<>());  
 for (int j = 0; j < matrix.size(); j++) {  
 if (i == row && j == col) {  
 this.N.get(i).add(Double.*POSITIVE\_INFINITY*);  
 } else {  
 this.N.get(i).add(matrix.get(i).get(j));  
 }  
 }  
 }  
}

На основе входных данных получен результат, представленный на рисунках 3 и 4.



(Рисунок 3)



(Рисунок 4)

## 2.3 Анализ работы алгоритмов и сравнение полученных результатов

Протестировав муравьиный алгоритм и метод ветвей и границ на разном количестве вершин с различными, рандомно сгенерированными, матрицами расстояний я пришла к следующим заключениям.

Начнем с **муравьиного алгоритма**. В данном алгоритме может варьироваться не только матрица расстояний, но и объявленные в начале реализации константы.

Как отмечалось ранее, *ALPHA* и *BETTA* — константы, отвечающие за степень влияния количества феромона на ребрах и длины ребра при выборе маршрута. Чем меньше *ALPHA*, тем ближе алгоритм к так называемому «жадному» алгоритму, т.е. процент вероятности выбора вершины зависит только от длины потенциального ребра маршрута. В полноценном «жадном алгоритме» при выборе вершины муравей опирается только на длину потенциального ребра маршрута и выбирает ту, для которой процент вероятности перехода наибольший, т.е. не осуществляется рандомный выбор среди разбиения отрезка [0; 100].

При уменьшении параметра *BETTA* большее влияние оказывает количество феромона на ребре. Такой подход влечет за собой быстрое вырождение маршрутов к одному субоптимальному решению, что также не является желательным.

Так как цель моей курсовой работы состоит в том, чтобы изучить классический муравьиный алгоритм, я подобрала выше указанные константы так, чтобы влияние феромона и длины ребра было равным, т.е. *ALPHA = BETTA =* 1.

Следующая постоянная, которая также может повлиять на точность алгоритма, *RHO* – коэффициент испарения. От этого параметра зависит то, насколько быстро «испаряется» феромон с ребра. Чем меньше *PHO*, тем выше «запоминаемость» алгоритма, т.е. если *RHO =* 0, то феромон не будет испаряться, а лишь усиливаться, и привлекательность наиболее длинных ребер будет оставаться неизменной по сравнению с начальным уровнем.

Еще один параметр, отвечающий за точность нахождения наиболее оптимального маршрута, – количество муравьев в колонии. Чем больше муравьев в колонии, тем сильнее выделяются наиболее часто используемые ребра, и тем выше вероятность нахождения действительно самого оптимального решения, хотя увеличение количества муравьев ведет к увеличению времени работы алгоритма.

К сожалеиню, ни один из выше перечисленных параметров не способен сделать муравьиный алгоритм по-настоящему точным. Данный метод далек от полного перебора, что влечет за собой неточность поиска. Тем не менее, при малом количестве вершин (3 – 6) в 90% случаев данный алгоритм действительно находит самый оптимальный (короткий) маршрут, причем за довольно короткий промежуток времени.

Перейдем к **методу ветвей и границ**. Опытным путем была выявлена неточность описания данного алгоритма в различных, изученных мной, источниках.

Дело в том, что если искать маршрут строго по указанному алгоритму, то, начиная с количества вершин, равного 7 (есть частные случаи и для меньшего количества) с большой вероятностью может произойти преждевременное зацикливание. Это значит, что вместо одного гамильтонова цикла, алгоритм выдает решение, состоящее из двух независимых циклов, также в сумме проходящих через все вершины.

Главным недостатком описанной реализации является нахождение ребер в произвольной (неупорядоченной) последовательности, что не гарантирует невозврат в исходную точку до окончания работы алгоритма.

Чтобы обойти данную проблему и добиться точных результатов для любого количества вершин, я задала стартовую вершину, равную 1, и немного модифицировала принцип выбора нулевого элемента:

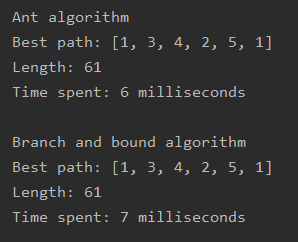
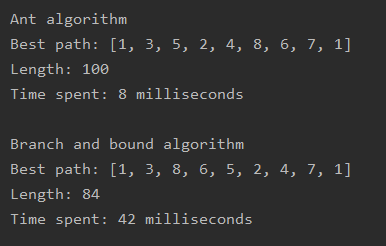
public ArrayList<ArrayList<Integer>> findZeros(ArrayList<ArrayList<Double>> matr, ArrayList<Integer> Path) {  
 ArrayList<ArrayList<Integer>> zeros = new ArrayList<>();  
 ArrayList<Double> coeffList = new ArrayList<>();  
 for (int i = 0; i < matr.size(); i++) {  
 for (int j = 0; j < matr.size(); j++) {  
 if (matr.get(i).get(j) == 0 && i == Path.get(Path.size() - 1) - 1) {  
 zeros.add(new ArrayList<>(Arrays.*asList*(i, j)));  
 coeffList.add(findCoefficients(matr, i, j));  
 }  
 }  
 }  
 if (zeros.size() != 0) {  
 findMaxCoeff(zeros, coeffList);  
 }  
 return zeros;  
}

В этой функции при нахождении нулевого элемента проверяется, является ли начало проверяемого ребра концом последнего ребра текущего маршрута, для реализации связности маршрута. В список нулевых элементов заносятся только те ребра, которые удовлетворяют данному условию, и выбор элемента с наибольшим штрафом производится только среди них.

С помощью приведенной модификации удается избежать нежелательного замыкания пути, вплоть до возвращения в исходную вершину 1 на последней итерации алгоритма.

Большим преимуществом метода ветвей и границ является нахождение действительно кратчайшего маршрута, т.к. алгоритм является модификацией (оптимизацией) полного перебора путем отсечения заведомо плохих решений. Однако по той же причине время выполнения алгоритма при большом количестве вершин во много раз превышает время выполнения эвристических методов, таких как муравьиный алгоритм.

Для сравнения еще раз приведем результаты выполнения обоих алгоритмов для одинаковых матриц расстояний (рисунки 4 и 5).

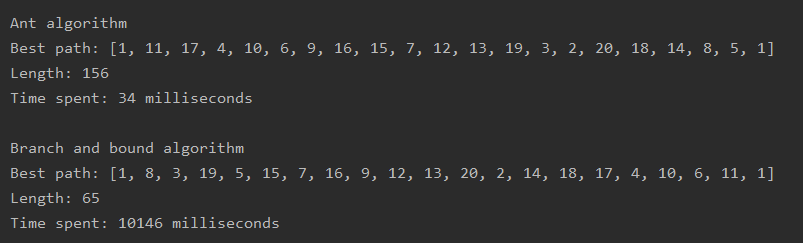
 

(Рисунок 4) (Рисунок 5)

Можно заметить, что для 5 вершин оба алгоритма отработали одинаково хорошо и даже нашли одинаковый маршрут. Для количества вершин, равных от 3 до 6, метод ветвей и границ может работать даже быстрее, если порядок следования вершин в полученном маршруте максимально приближен к прямому (1, 2, …, n). На графах такой размерности реализованный мной муравьиный алгоритм в большинстве случаев находит наиболее оптимальное решение с погрешностью в 1 – 10 единиц, в зависимости от начальных весов ребер.

Для второй матрицы (размерности 8) муравьиный алгоритм отработал уже хуже в плане длины маршрута, однако метод ветвей и границ работает в 5 раз дольше, что тоже нельзя назвать очень хорошим результатом.

Чем больше количество вершин в графе, тем менее точно муравьиный алгоритм находит оптимальное решение, и тем дольше отрабатывает метод ветвей и границ. Для графа порядка n ≥ 20 метод ветвей и границ находит путь почти в два раза короче, чем муравьиный алгоритм, но и работает в несколько сотен раз дольше, что можно увидеть на рисунке 6.

  
(Рисунок 6)

# ЗАКЛЮЧЕНИЕ

Задача коммивояжера является одной из наиболее знаменитых задач теории комбинаторики и пользуется популярностью благодаря тому, что к ней сводится довольно большое количество практических задач.

Среди современных практических приложений задачи можно выделить: доставку продуктов в магазин со склада, работу почтальона, мониторинг объектов (нефтяные вышки, базовые станции сотовых операторов), изготовление отверстий на специализированном станке.

Для решения задачи коммивояжера используют различные группы простейших методов: полный и случайный перебор, метод имитации отжига, жадный и деревянный алгоритмы. Широкое применение получили различные модификации более эффективных методов, таких как метод ветвей и границ, генетических алгоритмов, а также алгоритм муравьиных колоний.

Можно сказать, что на сегодняшний день не существует эффективного алгоритма для решения задачи коммивояжера, т.к. каждый алгоритм имеет свои сильные и слабые стороны. Те, что выигрывают по точности поиска, значительно уступают по скорости работы.

Однако в настоящее время остается актуальным поиск точных и приближенных способов решения этой задачи как с теоретической, так и с практической точек зрения. Более того, темпы современной жизни меняют отношение человека ко времени, сегодня пользователь постоянно ищет возможности сократить время ожидания, найти оптимальное решение в кратчайшие сроки. Все это свидетельствует о росте в будущем потребности в эффективном решении задачи коммивояжера и иных родственных ей оптимизационных задач, которые позволили бы существенно сэкономить ограниченные ресурсы организаций.

В ходе сравнения рассмотренных в данной работе алгоритмов я выделила для себя муравьиный алгоритм. Несмотря на то, что он не является точным, мне кажется, что он является более эффективным, нежели метод ветвей и границ, так как при большом количестве городов (вершин) время выполнения второго алгоритма может затянуться на несколько минут, часов и т.д.

Время работы алгоритма является весьма немаловажным фактором для сравнения эффективности, и в этом метод ветвей и границ существенно проигрывает муравьиному алгоритму, в то время, как второй все же находит одно из оптимальных решений, пусть и не лучшее.

Полная реализация данных алгоритмов представлена в приложении А.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. http://ru.wikipedia.org/wiki/Задача\_коммивояжёра [Электронный ресурс] – Дата доступа: 16.05.2020
2. Новиков Ф.А. Дискретная математика для программистов / Ф.А. Новиков. – Санкт-Петербург, Питер, 2001. – 304 с.;
3. Корбут А.А., Дискретное программирование / А.А. Корбут, Ю.Ю. Финкельштейн – М. Наука. Гл. ред. физ.-мат. лит., 1969;
4. Кристофидес Н. Теория графов. Алгоритмический подход / Н. Кристофидес – М: Мир, 1978;
5. Оре О. Графы и их применение / О. Оре; пер. с англ. под ред. И.М. Яглома – М., «Мир», 1965 – 174 с.;
6. Липатов Е.П. Теория графов и её применения / Е.П. Липатов – М., Знание, 1986 – 32 с.;
7. Dorigo M. Swarm Intelligence: from Natural to Artificial Systems / М. Dorigo, Е. Bonavear – Oxford University Press, 1999 – 307 p.;
8. Dorigo М. Ant Colony Optimization, Technical Report No / M. Dorigo, M. Birattari, T. Stutzle TR/IRIDIA/2006-023, 2006.;
9. MAX-MIN Ant System and local search for the traveling salesman problem: IEEE International Conference on Evolutionary Computation / T. Stützle, H. Hoos, 1997. – 309-314 с.;
10. Land A.H. An autоmatic method of solving discrete programming problems. Econometrica / А.Н. Land, A.G. Doig, 1960. – 497-520 р.;
11. Little J. D. С. An algorithm for the Traveling Salesman Problem / J. D. С. Little, K. G. Murty, D. W. Sweeney, С. Karel // Operations Research, 1963. No. 11. – 972–989 р.

# ПРИЛОЖЕНИЕ А

# Код реализации муравьиного алгоритма и метода ветвей и границ на языке программирования Java

**А 1.1 Класс Main**

package com.company;  
import com.company.creator.Creator;  
import com.company.entity.Ant;  
import com.company.entity.BranchAndBound;  
import com.company.handler.MatrixHandler;  
import com.company.reader.InfoReader;  
import com.company.way.WayType;  
import java.util.ArrayList;  
public class Main {  
 public static void main(String[] args) {  
 int n;  
 InfoReader reader = new InfoReader();  
 n = reader.readVertex();  
 Creator creator = new Creator(n);  
 for (ArrayList<Double> d : creator.getDistance()) {  
 System.*out*.print("{");  
 for (double item : d) {  
 System.*out*.printf("%8.0f ", item);  
 }  
 System.*out*.println("}");  
 }  
 // Ant algorithm  
 long start = System.*currentTimeMillis*();  
 WayType wayType = new WayType(n);  
 Ant ant = wayType.findShortestWay(creator, n);  
 long timeSpent = System.*currentTimeMillis*() - start;  
 System.*out*.println("\nAnt algorithm\nBest path: " + ant.getPath());  
 System.*out*.printf("Length: %1.0f", ant.getLength());  
 System.*out*.println("\nTime spent: " + timeSpent + " milliseconds");  
  
 // Branch and bound algorithm  
 start = System.*currentTimeMillis*();  
 BranchAndBound branch = new BranchAndBound(creator.getDistance());  
 MatrixHandler matrixHandler = new MatrixHandler();  
 for (int i = 1; i < n; i++) {  
 ArrayList<Integer> path = new ArrayList<>();  
 ArrayList<ArrayList<Double>> newMatrix = new ArrayList<>();  
 matrixHandler.copyMatrix(branch.getMatrix(), newMatrix);  
 matrixHandler.setY(newMatrix, 0, i);  
  
 path.add(1);  
 path.add(i + 1);  
 branch.matrixUpdate(matrixHandler.getY(), path, branch.getMatrix().get(0).get(i));  
 }  
 timeSpent = System.*currentTimeMillis*() - start;  
 System.*out*.println("\nBranch and bound algorithm\nBest path: " + branch.getRecord());  
 System.*out*.printf("Length: %1.0f", branch.getRecordWeight());  
 System.*out*.println("\nTime spent: " + timeSpent + " milliseconds");  
 }  
}

**А 1.2 Класс InfoReader**

package com.company.reader;  
import java.io.InputStream;  
import java.util.Scanner;  
public class InfoReader {  
 public static int *MIN\_VERTEX* = 3;  
 public static int *MAX\_VERTEX* = 20;  
 public static final String *REGEX\_NATURAL* = "[1-9]\\d?";  
 public int intReader(InputStream input) {  
 Scanner scanner = new Scanner(input);  
 String line = scanner.nextLine();  
 line = line.trim();  
 while (!line.matches(*REGEX\_NATURAL*)) {  
 System.*out*.println("Input natural number!");  
 line = scanner.nextLine();  
 line = line.trim();  
 }  
 return Integer.*parseInt*(line);  
 }  
 public int readVertex() {  
 int n = 0;  
 while (n < *MIN\_VERTEX* || n > *MAX\_VERTEX*) {  
 System.*out*.println("Input amount of vertices between 3 and 20: ");  
 n = intReader(System.*in*);  
 }  
 return n;  
 }  
}

**А 1.3 Класс Ant**

package com.company.entity;  
import com.company.creator.Creator;  
import com.company.way.WayType;  
import java.util.ArrayList;  
public class Ant {  
 private ArrayList<Integer> path;  
 private double length;  
 public Ant() {  
 this.path = new ArrayList<>();  
 this.path.add(1);  
 }  
 public ArrayList<Integer> getPath() {  
 return path;  
 }  
 public void setPath(ArrayList<Integer> path) {  
 this.path = path;  
 }  
 public double getLength() {  
 return length;  
 }  
 public void setLength(double length) {  
 this.length = length;  
 }  
 public void chooseVertex(Creator creator, Ant ant, WayType wayType) {  
 int prob = (int) ( 1 + Math.*random*() \* 99 );  
 ArrayList<Integer> P = creator.createProbabilityArray(ant, wayType);  
 ArrayList<Integer> unvisited = wayType.getUnvisited();  
 for(int i = 0; i < P.size() - 1; i++) {  
 if(prob >= P.get(i) && prob <= P.get(i + 1)) {  
 this.path.add(unvisited.get(i));  
 unvisited.remove(i);  
 break;  
 }  
 }  
 }  
}

**А 1.4 Класс Creator**

package com.company.creator;  
import com.company.entity.Ant;  
import com.company.way.WayType;  
import java.util.ArrayList;  
public class Creator {  
 private ArrayList<ArrayList<Double>> distance;  
 private ArrayList<ArrayList<Double>> pheromone;  
 public Creator(int n) {  
 this.distance = createDistanceMatrix(n);  
 this.pheromone = createPheromoneMatrix(n);  
 }  
 public ArrayList<ArrayList<Double>> getDistance() {  
 return distance;  
 }  
 public void setDistance(ArrayList<ArrayList<Double>> distance) {  
 this.distance = distance;  
 }  
 public ArrayList<ArrayList<Double>> getPheromone() {  
 return pheromone;  
 }  
 public void setPheromone(ArrayList<ArrayList<Double>> pheromone) {  
 this.pheromone = pheromone;  
 }  
 public ArrayList<ArrayList<Double>> createDistanceMatrix(int n) {  
 ArrayList<ArrayList<Double>> distance = new ArrayList<>();  
 for (int i = 0; i < n; i++) {  
 distance.add(new ArrayList<>());  
 }  
 for (int i = 0; i < n; i++) {  
 for (int j = i; j < n; j++) {  
 if (i == j) {  
 distance.get(i).add(Double.*POSITIVE\_INFINITY*);  
 } else {  
 distance.get(i).add((double) Math.*round*(Math.*random*() \* 30 + 1));  
 distance.get(j).add(distance.get(i).get(j));  
 }  
 }  
 }  
 return distance;  
 }  
 public ArrayList<ArrayList<Double>> createPheromoneMatrix(int n) {  
 ArrayList<ArrayList<Double>> pheromone = new ArrayList<>(n);  
 for (int i = 0; i < n; i++) {  
 pheromone.add(new ArrayList<>());  
 }  
 for (int i = 0; i < n; i++) {  
 for (int j = 0; j < n; j++) {  
 if (i == j) {  
 pheromone.get(i).add(0.0);  
 } else {  
 pheromone.get(i).add(1.0);  
 }  
 }  
 }  
 return pheromone;  
 }  
 public ArrayList<Integer> createProbabilityArray(Ant ant, WayType wayType) {  
 ArrayList<Integer> P = new ArrayList<>();  
 ArrayList<Integer> unvisited = wayType.getUnvisited();  
 int percentage = 0;  
 P.add(0);  
 for (int item : unvisited) {  
 percentage += wayType.probability(ant, item, pheromone, distance);  
 P.add(percentage);  
 }  
 P.set(P.size() - 1, 100);  
 return P;  
 }  
 public void createPath(Ant ant, WayType wayType, Creator creator) {  
 while (wayType.getUnvisited().size() != 0) {  
 ant.chooseVertex(creator, ant, wayType);  
 }  
 ant.getPath().add(1);  
 for (int i = 0; i < ant.getPath().size() - 1; i++) {  
 ant.setLength(ant.getLength() + distance.get(ant.getPath().get(i) - 1).get(ant.getPath().get(i + 1) - 1));  
 }  
 }  
}

**А 1.5 Класс WayType**

package com.company.way;  
import com.company.creator.Creator;  
import com.company.entity.Ant;  
import com.company.update.Update;  
import java.util.ArrayList;  
public class WayType {  
 public static double *ALPHA* = 1;  
 public static int *BETTA* = 1;  
 public static int *ANT\_AMOUNT* = 20;  
 private ArrayList<Integer> unvisited;  
 public WayType(int n) {  
 this.unvisited = new ArrayList<>();  
 for (int i = 2; i <= n; i++) {  
 this.unvisited.add(i);  
 }  
 }  
 public ArrayList<Integer> getUnvisited() {  
 return unvisited;  
 }  
 public void setUnvisited(ArrayList<Integer> unvisited) {  
 this.unvisited = unvisited;  
 }  
 // defines probability of going to vertex 'to' from the last visited one  
 public int probability(Ant ant, int to, ArrayList<ArrayList<Double>> pheromone, ArrayList<ArrayList<Double>> distance) {  
 int from = ant.getPath().get(ant.getPath().size() - 1);  
 double sum = 0;  
 for (int item : unvisited) {  
 sum += Math.*pow*(pheromone.get(from - 1).get(item - 1), *ALPHA*) \* Math.*pow*(1.0 / distance.get(from - 1).get(item - 1), *BETTA*);  
 }  
 double numerator = Math.*pow*(pheromone.get(from - 1).get(to - 1), *ALPHA*) \* Math.*pow*(1.0 / distance.get(from - 1).get(to - 1), *BETTA*);  
 double Pij = numerator / sum;  
 return (int) Math.*round*(Pij \* 100);  
 }  
 public Ant findShortestWay(Creator creator, int n) {  
 Update update = new Update();  
 double length = 0;  
 Ant bestAnt = new Ant();  
 for (int k = 0; k < *ANT\_AMOUNT*; k++) {  
 Ant ant = new Ant();  
 WayType wayType = new WayType(n);  
 creator.createPath(ant, wayType, creator);  
 update.updatePheromone(ant, creator, n);  
 if (length == 0) {  
 bestAnt = ant;  
 length = ant.getLength();  
 } else if (ant.getLength() < length) {  
 length = ant.getLength();  
 bestAnt = ant;  
 }  
 }  
 return bestAnt;  
 }  
}

**А 1.6 Класс Update**

package com.company.update;  
import com.company.creator.Creator;  
import com.company.entity.Ant;  
import java.util.ArrayList;  
public class Update {  
 public static double *RHO* = 0.2;  
  
 public void updatePheromone(Ant ant, Creator creator, int n) {  
 double delta = 1.0 / ant.getLength();  
 for (int i = 0; i < n; i++) {  
 for (int j = i; j < n; j++) {  
 if (i != j){  
 creator.getPheromone().get(i).set(j, (1 - *RHO*) \* creator.getPheromone().get(i).get(j));  
 creator.getPheromone().get(j).set(i, creator.getPheromone().get(i).get(j));  
 }  
 }  
 }  
 ArrayList<Integer> path = ant.getPath();  
 for (int i = 0; i < path.size() - 1; i++) {  
 creator.getPheromone().get(path.get(i) - 1).set(path.get(i + 1) - 1, creator.getPheromone().get(path.get(i) - 1).get(path.get(i + 1) - 1) + delta);  
 creator.getPheromone().get(path.get(i + 1) - 1).set(path.get(i) - 1, creator.getPheromone().get(path.get(i) - 1).get(path.get(i + 1) - 1));  
 }  
 }  
}

**А 1.7 Класс BranchAndBound**

package com.company.entity;  
import com.company.handler.MatrixHandler;  
import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.Collections;  
public class BranchAndBound {  
 private ArrayList<ArrayList<Double>> matrix;  
 private ArrayList<Integer> record;  
 private double recordWeight;  
 public BranchAndBound(ArrayList<ArrayList<Double>> distance) {  
 this.matrix = new ArrayList<>();  
 for (int i=0; i<distance.size(); i++) {  
 this.matrix.add(new ArrayList<>());  
 for (int j=0; j<distance.size(); j++) {  
 this.matrix.get(i).add(distance.get(i).get(j));  
 }  
 }  
 setRecord();  
 setRecordWeight();  
 }  
 public BranchAndBound(ArrayList<ArrayList<Double>> distance, ArrayList<Integer> record, double record\_weight) {  
 this.matrix = new ArrayList<>();  
 for (int i = 0; i < distance.size(); i++) {  
 this.matrix.add(new ArrayList<>());  
 for (int j = 0; j < distance.get(i).size(); j++) {  
 this.matrix.get(i).add(distance.get(i).get(j));  
 }  
 }  
 this.record = new ArrayList<>();  
 this.record.addAll(record);  
 this.recordWeight = record\_weight;  
 }  
 public ArrayList<ArrayList<Double>> getMatrix() {  
 return matrix;  
 }  
 public void setMatrix(ArrayList<ArrayList<Double>> matrix) {  
 this.matrix = matrix;  
 }  
 public ArrayList<Integer> getRecord() {  
 return record;  
 }  
 public void setRecord() {  
 this.record = new ArrayList<>();  
 for (int i = 1; i <= this.matrix.size(); i++) {  
 this.record.add(i);  
 }  
 this.record.add(1);  
 }  
 public double getRecordWeight() {  
 return recordWeight;  
 }  
 public void setRecordWeight() {  
 this.recordWeight = 0;  
 for (int i = 0; i < this.record.size() - 1; i++) {  
 this.recordWeight += this.matrix.get(this.record.get(i) - 1).get(this.record.get(i + 1) - 1);  
 }  
 }  
 public void matrixUpdate(ArrayList<ArrayList<Double>> matrix, ArrayList<Integer> path, double lowerBound) {  
 ArrayList<Double> minRow = new ArrayList<>();  
 ArrayList<Double> minColumn = new ArrayList<>();  
 lowerBound = findMin(matrix, minRow, lowerBound);  
 transpose(matrix);  
 lowerBound = findMin(matrix, minColumn, lowerBound);  
 transpose(matrix);  
 // cut branch if it is longer, than record  
 if (lowerBound >= recordWeight) {  
 return;  
 }  
 if (path.size() == record.size()) {  
 record.clear();  
 record.addAll(path);  
 this.recordWeight = lowerBound;  
 return;  
 }  
 if (findZeros(matrix, path).size() != 0) {  
 ArrayList<Integer> currentZero = findZeros(matrix, path).get(0);  
 MatrixHandler handler = new MatrixHandler(matrix, currentZero.get(0), currentZero.get(1));  
 ArrayList<Integer> newPath = new ArrayList<>(path);  
 newPath.add(currentZero.get(1) + 1);  
  
 BranchAndBound Y = new BranchAndBound(handler.getY(), newPath, lowerBound);  
 matrixUpdate(Y.matrix, Y.record, Y.recordWeight);  
  
 BranchAndBound N = new BranchAndBound(handler.getN(), path, lowerBound);  
 matrixUpdate(N.matrix, N.record, N.recordWeight);  
 }  
 }  
 private double findMin(ArrayList<ArrayList<Double>> matrix, ArrayList<Double> minimum, double lowerBound) {  
 for (ArrayList<Double> d : matrix) {  
 if (Collections.*min*(d) == Double.*POSITIVE\_INFINITY*) {  
 minimum.add(0.0);  
 } else {  
 minimum.add(Collections.*min*(d));  
 }  
 lowerBound += minimum.get(minimum.size() - 1);  
  
 for (int i = 0; i < d.size(); i++) {  
 if (d.get(i) != Double.*POSITIVE\_INFINITY*) {  
 d.set(i, d.get(i) - minimum.get(minimum.size() - 1));  
 }  
 }  
 }  
 return lowerBound;  
 }  
 private void transpose(ArrayList<ArrayList<Double>> a) {  
 for (int i = 0; i < a.size(); i++) {  
 for (int j = i + 1; j < a.size(); j++) {  
 double temp = a.get(i).get(j);  
 a.get(i).set(j, a.get(j).get(i));  
 a.get(j).set(i, temp);  
 }  
 }  
 }  
 // finds fine (coefficient) of zero  
 public double findCoefficients(ArrayList<ArrayList<Double>> matr, int row, int col) {  
 ArrayList<Double> rowElements = new ArrayList<>();  
 ArrayList<Double> colElements = new ArrayList<>();  
 for (int i = 0; i < matr.size(); i++) {  
 if (i != row) {  
 colElements.add(matr.get(i).get(col));  
 } else {  
 for (int j = 0; j < matr.get(i).size(); j++) {  
 if (j != col) {  
 rowElements.add(matr.get(i).get(j));  
 }  
 }  
 }  
 }  
 return Collections.*min*(rowElements) + Collections.*min*(colElements);  
 }  
 public ArrayList<ArrayList<Integer>> findZeros(ArrayList<ArrayList<Double>> matr, ArrayList<Integer> Path) {  
 ArrayList<ArrayList<Integer>> zeros = new ArrayList<>();  
 ArrayList<Double> coeffList = new ArrayList<>();  
 for (int i = 0; i < matr.size(); i++) {  
 for (int j = 0; j < matr.size(); j++) {  
 if (matr.get(i).get(j) == 0 && i == Path.get(Path.size() - 1) - 1) {  
 zeros.add(new ArrayList<>(Arrays.*asList*(i, j)));  
 coeffList.add(findCoefficients(matr, i, j));  
 }  
 }  
 }  
 if (zeros.size() != 0) {  
 findMaxCoeff(zeros, coeffList);  
 }  
 return zeros;  
 }  
 public void findMaxCoeff(ArrayList<ArrayList<Integer>> zeros, ArrayList<Double> coeffList) {  
 double maxCoeff = Collections.*max*(coeffList);  
 for (int i = 0; i < coeffList.size(); i++) {  
 if (coeffList.get(i) != maxCoeff) {  
 coeffList.remove(i);  
 zeros.remove(i);  
 i--;  
 }  
 }  
 }  
}

**А 1.8 Класс MatrixHandler**

package com.company.handler;  
import java.util.ArrayList;  
public class MatrixHandler {  
 private ArrayList<ArrayList<Double>> Y;  
 private ArrayList<ArrayList<Double>> N;  
 public MatrixHandler(ArrayList<ArrayList<Double>> matrix, int row, int col) {  
 setY(matrix, row, col);  
 setN(matrix, row, col);  
 }  
 public MatrixHandler() {}  
 public ArrayList<ArrayList<Double>> getY() {  
 return Y;  
 }  
 public void setY(ArrayList<ArrayList<Double>> matrix, int row, int col) {  
 this.Y = new ArrayList<>();  
 for (int i = 0; i < matrix.size(); i++) {  
 this.Y.add(new ArrayList<>());  
 for (int j = 0; j < matrix.size(); j++) {  
 if (i == row || j == col) {  
 this.Y.get(i).add(Double.*POSITIVE\_INFINITY*);  
 } else {  
 this.Y.get(i).add(matrix.get(i).get(j));  
 }  
 }  
 }  
 this.Y.get(col).set(row, Double.*POSITIVE\_INFINITY*);  
 }  
 public ArrayList<ArrayList<Double>> getN() {  
 return N;  
 }  
 public void setN(ArrayList<ArrayList<Double>> matrix, int row, int col) {  
 this.N = new ArrayList<>();  
 for (int i = 0; i < matrix.size(); i++) {  
 this.N.add(new ArrayList<>());  
 for (int j = 0; j < matrix.size(); j++) {  
 if (i == row && j == col) {  
 this.N.get(i).add(Double.*POSITIVE\_INFINITY*);  
 } else {  
 this.N.get(i).add(matrix.get(i).get(j));  
 }  
 }  
 }  
 }  
 public void copyMatrix(ArrayList<ArrayList<Double>> A, ArrayList<ArrayList<Double>> B) {  
 for (int i = 0; i < A.size(); i++) {  
 B.add(new ArrayList<>());  
 for (int j = 0; j < A.get(i).size(); j++) {  
 B.get(i).add(A.get(i).get(j));  
 }  
 }  
 }  
}