МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»**

**(ФГБОУ ВО «КубГУ»)**

**Факультет компьютерных технологий и прикладной математики**

**Кафедра вычислительных технологий**

**КУРСОВАЯ РАБОТА**

**ПРОГРАММНАЯ ОПТИМИЗАЦИЯ АЛГОРИТМА СЛИЗЕВИКА ПОСТРОЕНИЯ МАРШРУТНОЙ СЕТИ**

Работу выполнил \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Д.Е. Гиренко

(подпись)

Направление подготовки 02.03.02 — «Фундаментальная информатика и\_\_\_\_\_

(код, наименование)

информационные технологии»\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_курс\_\_\_\_\_\_\_\_3\_\_\_\_\_\_\_\_

Направленность (профиль) \_\_\_\_\_\_\_\_\_\_\_\_ Математическое и программное обеспечение компьтерных технологий \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Научный руководитель

канд. техн. наук, доц. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Т.А. Приходько

(подпись, дата)

Нормоконтролер

ассистент \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Е.А. Нигодин

(подпись, дата)

Краснодар

2023

**РЕФЕРАТ**

Курсовая работа 26 стр., 3 ч., 5 рис., 4 источника, 1 приложение.

АЛГОРИТМ СЛИЗЕВИКА, МАРШРУТНЫЕ СЕТИ, МНОГОАГЕНТНЫЕ СИСТЕМЫ, ГРАФЫ, FLUTTER

Цель работы курсовой работы – оптимизировать работу алгоритма слизевика построения маршрутной сети и разработать удобный интерфейс для его использования.

Объект исследования: Алгоритмы построения маршрутных сетей.

Предмет исследования: Улучшенный алгоритм работы слизевика для построения модели маршрутной сети с последующим выделением лучшего мартшрута.

В качестве методов исследования использовались эмпирический метод (наблюдение, тестирование), количественный и качественный анализ, методы математического моделирования и программирования. В результате работы были изучены методы минимизации графов, способы вычисления эффективности маршрутных сетей, принципы проектирования и разработки десктопных приложений.

Практическая новизна работы заключается в формировании и разработке нового приложения удобного и быстрого проектирования маршрутных сетей.

По результатам работы, было спроектировано и разработано кроссплатформенное приложение, предназначенное для проектирования маршрутных сетей.

**СОДЕРЖАНИЕ**

[Введение 4](#_Toc135650175)

[1 Алгоритм слизевика в задаче построения маршрутных сетей 6](#_Toc135650176)

[1.1 Задача построения маршрутной сети 6](#_Toc135650177)

[1.2 Использование алгоритма слизевика в задаче построения 8](#_Toc135650178)

[1.3 Необходимость модернизирования алгоритма 9](#_Toc135650179)

[2 Описание используемых алгоритмов минимизации маршрутной сети и разработанных метрик 11](#_Toc135650180)

[2.1 Реализация результатов работы алгоритма слизевика 11](#_Toc135650181)

[2.2 Алгоритм выделения вершин сети и алгоритм загрубления 11](#_Toc135650182)

[2.3 Определение метрических параметров маршрутных сетей 15](#_Toc135650183)

[2.4 Выделение лучшей маршрутной сети 18](#_Toc135650184)

[3 Разработка приложения на Flutter 19](#_Toc135650185)

[3.1 Необходимость разработки приложения 19](#_Toc135650186)

[3.2 Причина использования Flutter 19](#_Toc135650187)

[3.2 Описание функциональности приложения 20](#_Toc135650188)

[Заключение 24](#_Toc135650189)

[Список использованных источников 26](#_Toc135650190)

[Приложение А Основные функции загрубления, минимизации графов и функции вычисления метрических показателей 27](#_Toc135650191)

# ВВЕДЕНИЕ

Задача построения маршрута является одной из самых часто встречаемых задач на графах. Из-за её фундаментальности с точки зрения транспорта она будет актуальна всегда. Однако для компьютерного моделирования эта задача является крайне сложной по причине нп-полноты алгоритмов перебора и из-за дополнительных потребностей бизнеса. Для решения проблемы алгоритмической сложности в прошлой курсовой было предложено использовать алгоритм слизевика, который в последствии был реализован и показал неплохой результат.

Однако суждение о неплохом результате работы слизевика было сформировано на основе личного опыта. Конечно, задача оптимизации алгоритмической сложности была выполнена. Но самостоятельная настройка параметров требует знаний принципов работы алгоритма слизевика, генерирование множества результатов работы требует зрительного осмотра каждого из них, а итоговое решение о лучшей маршрутной сети из всех созданных невозможно без знаний о том, как должна выглядеть подобная. К тому же у программы не имеется удобного интерфейса и управление происходит с помощью txt файла и проводника. Подобный метод использования далёк от понятия чёрный ящик.

По этой причине в данной курсовой работе было решено продолжить разработку алгоритмического решения задачи построения маршрутной сети. Имеющаяся главная проблема нынешнего решения – большая роль человека для отбора результатов работы алгоритмов и неудобство использования программы. Для решения оной было решено разработать алгоритмы графов построения маршрутной сети из результатов работы алгоритма слизевика и некие математические метрики, которые будут использоваться для сравнения оных. После того, как проблема выбора будет решена алгоритмически, все результаты стоит обернуть в интерфейс для удобства пользования.

Основная цель работы – реализовать программную автоматизацию формирования лучшего решения задачи построения маршрутной сети при помощи алгоритма слизевика и предоставить удобный интерфейс пользователю.

Для реализации поставленной цели предполагается решить следующие задачи:

* изучить потребности бизнеса в задаче построения маршрутной сети и методы алгоритмического построения оных;
* реализовать алгоритмы построения маршрутной сети и методы вычисления метрических показателей;
* реализовать десктопное приложение для удобного управления работы алгоритма;

Объектом исследования в работе является программное построения маршрутных сетей.

Предметом исследования является математическое преобразование результатов работы алгоритма слизевика в модель маршрутной сети с последующим выделением лучшего варианта и удобным предоставлением пользователю.

Информационная база исследования включает в себя несколько видов учебных материалов таких как книги по маршрутным сетям, статьи от авторитетных сервисов и документация. В качестве методов исследования использовались абстрагирование, наблюдение и моделирование.

Научная новизна работы заключается в том, что разработанное новое решение задачи построения оптимальной маршрутной сети автоматизируется для исключения столь большой роли человека в задаче.

Практическая значимость исследования курсовой работы заключается в результатах исследования, которые пригодятся для дальнейшей работы над решением поставленной задачи и реализацией полного алгоритма до модели чёрного ящика.

# Алгоритм слизевика в задаче построения маршрутных сетей

## Задача построения маршрутной сети

Маршрутная сеть — это графическое представление системы транспортных маршрутов, связывающих различные местоположения в определенной территориальной области. Маршрутная сеть включает в себя узлы (точки) и ребра (пути), которые представляют собой маршруты между этими узлами. Она может включать различные виды транспорта, такие как дороги, железные дороги, авиалинии, водные пути и общественный транспорт.

Задача построения маршрутной сети заключается в определении наиболее эффективных и оптимальных маршрутов для перемещения между различными местоположениями внутри заданной территориальной области. Целью задачи является разработка алгоритмов и моделей, которые могут оптимизировать использование доступных транспортных маршрутов, учитывая различные факторы, такие как время, расстояние, пропускная способность и предпочтения пользователей [1]. Как раз первые три фактора являются основными параметрами для того, чтобы разработать на их основе метрические показатели, что в реализации алгоритма будут использоваться для определения лучшего из всех сгенерированных вариантов маршрутных сетей.

Результаты, полученные в ходе выполнения задачи построения маршрутной сети, представляют собой ценную информацию, которая может быть использована для различных целей и приложений. Одним из важных применений этих результатов является планирование городского развития [1]. Имея информацию о маршрутах и связях между различными местами в городе, можно определить оптимальное размещение объектов, развивать транспортную инфраструктуру и создавать удобные условия для горожан.

Оптимизация транспортной логистики также является важным применением результатов построения маршрутной сети. Зная оптимальные маршруты доставки товаров или организации транспортных потоков, можно снизить издержки и улучшить эффективность процессов логистики, что приведет к экономическим выгодам и повышению уровня обслуживания.

Разработка навигационных систем является еще одним важным применением результатов задачи построения маршрутной сети. На основе полученных данных можно создать точные и надежные системы навигации, которые помогут пользователям быстро и безопасно достигать своих пунктов назначения. Это особенно важно для автомобильной и пеших навигационных систем, которые облегчают перемещение в неизвестных местах и сокращают время поиска нужного пути.

Кроме того, результаты задачи построения маршрутной сети могут быть применены при разработке систем общественного транспорта. Зная оптимальные маршруты и расписание движения транспорта, можно повысить эффективность общественного транспорта, улучшить доступность и удобство его использования, а также снизить транспортные проблемы в городе.

Таким образом, результаты задачи построения маршрутной сети имеют широкий спектр применений, связанных с перемещением и транспортом. Они могут быть использованы для планирования городского развития [1], оптимизации транспортной логистики, разработки навигационных систем и улучшения систем общественного транспорта, способствуя эффективному и удобному перемещению людей и грузов.

Так как в прошлой курсовой для определения эффективности результатов работы алгоритма слизевика использовалась карта железнодорожных путей Краснодарского края, то и в рамках этой научных работы основная цель – построение карты железнодорожных путей. В связи с этим на задачу накладываются следующие изменения:

* 1. Имеется только один тип дорог;
  2. У каждой вершины есть лишь параметр приоритета, который отвечает и за то, сколько потока создаёт эта вершина и то, как много потока других вершин должно останавливаться в ней;
  3. Дополнительный параметр определения оптимальной маршрутной сети – устойчивость, который показывает, как сильно пострадает маршрутная сеть при исключении одного из рёбер. Под исключением может подразумеваться и авария, и стихийное бедствие, и ремонтные работы.

## Использование алгоритма слизевика в задаче построения

Алгоритм Physarum mold — это мультиагентный алгоритм, инспирированный поведением сети слизевиком Physarum polycephalum. Этот алгоритм моделирует процессы поиска оптимальных путей, основываясь на физической структуре и поведении физаровой плесени.

В алгоритме Physarum mold моделируется распределение потока в сети, где каждый псевдоподий физаровой плесени соответствует ребру в графе. Используя принципы обратной связи и оптимизации, алгоритм регулирует поток вдоль ребер сети, чтобы находить оптимальные пути. Поскольку поток в сети распределяется на основе того, какие пути являются более привлекательными, алгоритм может находить оптимальные маршруты, учитывая различные факторы, такие как длина пути или стоимость прохождения через ребра.

Именно этот алгоритм был реализован и протестирован в прошлой курсовой для решения задачи построения оптимальной маршрутной сети. Результатом его работы является множество bmp изображений локации с расположенными на них агентами, занимающими по одному пикселю. Так как в моделировании использовались тысячи агентов и не очень большие карты локации, то глядя на рисунок можно было вполне чётко определить те маршруты и пути, которые проложили своими телами в момент оцифровки в изображение агенты. Так как сеть постоянно изменялась и, причём, не обязательно в лучшую сторону, то принцип жадности алгоритма не работал и для выделения лучшего результата было необходимо оцифровывать локацию и агентов на ней каждые n итераций. В результате человек мог осмотреть все созданные изображения и выбрать лучший вариант маршрутной сети, самостоятельно в последствие проложив на нём чёткие рёбра.

## 1.3 Необходимость модернизирования алгоритма

Результаты тестов хоть и были удовлетворительными, однако они дались путём достаточно долгой настройки работы алгоритма. Одной из главных проблем той реализации алгоритма была неустойчивость во времени сети, из-за чего приходилось достаточно долго ждать, чтобы поймать момент оптимального расположения элементов всей сети.

В связи с этим было решено изменить сам алгоритм слизевика следующим образом:

* 1. Сделать фактор разложения линейным. Это было сделано потому, что зоны с большим содержанием следа слишком быстро за счёт умножения теряли накопления, что мешало формированию чётких маршрутов.
  2. Диффузия была полностью отключена по той же причине – пути, что формируются агентами во время симуляции после определённой некой достаточной популярности, должны перестать изменяться.
  3. Теперь у каждого агента имеется ещё один атрибут – массив пути. Это массив, в который последовательно записывается все посещённые пиксели локации. В случае, если агент прибывает к другому генератору, то каждой из посещённых им точек в массиве путей добавляется пятикратный след за шаг. В случае, если у агента заканчивается время жизни, то след каждого пикселя пути уменьшается на 1.8 следа за шаг агента. Аналогично всем предыдущим изменениям, массив путей необходим для того, чтобы формировать стойкие и чёткие маршруты агентов, достаточно сильно их подкрепляя. Так как не всегда агенты способны пройти по уже готовому маршруту и закончить свою жизнь в другом генераторе, то штраф пути практически в 2.5 раза меньше бонуса за правильный путь.

С данными изменения маршрутные сети, что начали формировать агенты, стали намного более устойчивыми во времени и самоорганизующимися, однако при этом немного потеряли в гибкости. Также было замечено, что генераторы с малым приоритетом и находящие не в центре сети практически всегда становятся висячими вершинами в формирующимся графе, что является правильным решение в организации оптимальной маршрутной сети.

Для того, чтобы продвинуться дальше в автоматизации алгоритма необходимо научить программу переводить результаты работы алгоритма слизевика в маршрутную сеть так, как это делает взглядом человек. То есть соединять те линии, что образованы большим потоком агентов в единое ребро. Кроме того, одним из преимуществ алгоритма слизевика перед обычным перебором является то, что он способен формировать дополнительны вершины в графе, а значит алгоритм формирования маршрутной сети должен учитывать то, что количество вершин должно быть больше, чем количество генераторов.

Задача выбора лучшей из множества маршрутных сетей решается при помощи определения функции сравнения на этом множестве. Главные параметры маршрутных сетей – время, расстояние, устойчивость и поток – уже выбраны, а значит нам необходимо представить их в виде некоторого вещественного числа. Научившись сравнивать маршрутные сети по отдельным параметрам, будет достаточно легко написать функцию сравнения.

# Описание используемых алгоритмов минимизации маршрутной сети и разработанных метрик

## 2.1 Реализация результатов работы алгоритма слизевика

На вход в программу подаются настройки симуляции, а также генераторы, которые определяются координатой в пространстве локации, а также приоритетностью. Приоритетность определяет количество агентов, что создаются генератором в секунду.

Через некоторое количество итераций получается карта локации с расположением на ней множеством агентов, которые для обычного человека похожи на маршрутную сеть. Такой эффект похожести достигается за счёт того, что агенты расположены достаточно кучно и вдоль линий между вершинами. Места, где агенты расположены с разрывами или же где агентов совсем мало слабо похожи на пути. Кроме того, агентов очень много и большинство из них совсем не нужны для того, чтобы создать граф, однако они пригодятся для определения, достаточной ли «толщины» ребро. Кроме того, сами генераторы очень важные точки и их ни при каких условиях нельзя не отобразить в сети, и они все должны находиться в одной компонент-связности. Так что на вход алгоритму минимизации маршрутной сети подаётся массив координат агентов и массив координат генераторов.

## 2.2 Алгоритм выделения вершин сети и алгоритм загрубления

Загрубление графа — это процесс упрощения или уменьшения сложности исходного графа путем объединения вершин или ребер в более крупные сущности [2]. Цель загрубления графа заключается в создании нового графа меньшего размера, который сохраняет некоторые важные характеристики и структуру исходного графа. Именно загрубление нам и необходимо использовать, чтобы уменьшить количество вершин.

В начале полученные на вход координаты точек и собираются в один массив, сохраняя информацию о том, является ли точка генератором, после чего они сортируются. Было решено применить метод точного агрегирования для минимизации графа, а значит придётся пройтись по каждой паре вершин [2]. Две вершины становятся одной, находящейся по середине между исходными вершинами в случае, если расстояние между изначально заданного параметра – вершинное расстояние. Если одна из вершин генератор, то эта вершина остаётся недвижима, а вторая стирается. Сортировка вершин по координатам нужна была для того, чтобы оптимизировать поиск вершин, что должны соединиться во едино. Ведь, если вершины находятся слишком далеко друг от друга по одной из координат, то и расстояние между ними будет больше или равно.

Когда останутся вершины, что находятся на расстоянии больше, чем вершинное расстояние, алгоритм выделит из всего множества вершин все те, которые влили не меньше определённого количества вершин, которое задано параметром минимального веса вершин. Города остаются всегда.

Теперь из множества оставшихся вершин создаётся граф тем способом, что каждую пару вершин, что находится на расстоянии больше, чем параметр рёберного расстояния, соединяют. Параметр рёберного расстояния обязательно должен быть больше значения вершинного расстояния, причём, желательно в два раза больше, чтобы избежать потери связности получившегося графа.

На данном этапе вполне можно проследить рисунок маршрутной сети:

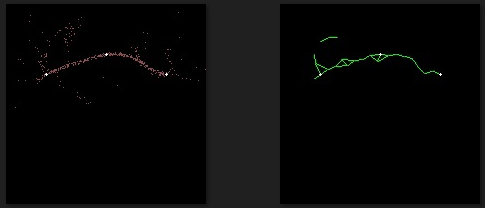


Рисунок 1 – Результат работы алгоритма загрубления графа

Однако такая маршрутная сеть всё ещё имеет слишком много лишних вершин и лишних рёбер, что сильно отразится на её эффективности. Особенно сильно это заметно на больших графах:

Изображение выглядит как зеленый

Автоматически созданное описание

Рисунок 2 – Результат работы алгоритма загрубления графа на большом графе

Лишние вершины, рёбра и структуры в получившемся графе было решено назвать аномалиями. Аномалии образованы рёбрами и теми вершинами, которые не являются городами. Все аномалии можно разделить на 4 типа:

* 1. Одинокая вершина. Это дополнительная вершина, что создана алгоритмом и к которой ведёт лишь одно ребро. Такая вершина абсолютно не нужна маршрутной сети. Кроме того, удаление таких вершин помогает избавиться от компонент связностей, что получились случайно и вне главной компонент связности;
  2. Треугольник. Вершина, что имеет только два ребра, что соединяются с вершинами, что тоже соединены ребром. Аналогично одинокой вершине, это бесполезно для маршрутной сети, кроме того, помогает очищать сеть от ненужных циклов;
  3. Ломаная. Если три вершины расположены в ряд и центральная соединена только с двумя крайними вершинами, то стоит удалить центральную и соединить крайние, чтобы в маршрутной сети было меньше лишних вершин. Алгоритм определяет, что вершины можно соединить, если между рёбрами угол не больше ещё одного параметра – минимального угла рёбер;
  4. Квадраты. Большая часть аномалий на больших графах, именно возникающие и наслаивающиеся друг на друга «квадраты». Эта аномалия обнаруживается, если у некоторой вершины есть такие две смежные вершины, которые имеют ещё одну смежную обоим вершину и при этом являются смежными друг для друга. Первая и последняя вершина в данном определении тоже могут быть смежными, это влияет на то, каким образом они стянутся. В случае, если ребро есть только между второй и третьей вершинами, то выполняется операция стягивания этого ребра. В случае, если ребро есть как между второй и третьей, так и между первой и четвёртой, то есть имеется полный граф, то все четыре вершины стягиваются в одну, находящуюся по центру. Стоит упомянуть, что вершины-генераторы не подвержены изменению положения и, если аномалия образуется с их участием, то либо все вершины стягиваются в единственный генератор, либо вовсе эта аномалия остаётся без изменений.

Алгоритм рассматривает каждую из имеющихся вершин, а после рассматривает все те вершины, которые были затронуты изменениями, для оптимизации вычислений. Обработка графа останавливается лишь тогда, когда больше нет ни одной аномалии. В конце из исходного графа удаляются все вершины, что были слиты в другие и на выходе получается маршрутная сеть удовлетворительного качества.

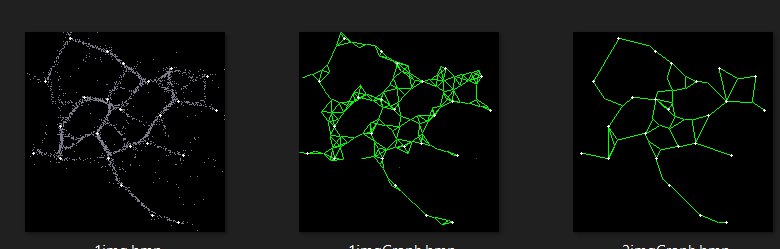


Рисунок 3 – Результат работы алгоритма минимизации графа

## 2.3 Определение метрических параметров маршрутных сетей

Теперь, когда результат работы алгоритма слизевика можно преобразить не в изображение, а в математический граф без лишних вершин и рёбер, необходимо определить несколько метрик, по которым графы будут сравниваться. В теоретической части было сказано, что самые главные параметры для маршрутной сети – длина рёбер, время перемещения, устойчивость и величина потока.

В дальнейших рассуждениях будет применяться понятие потока между двумя вершинами. Под ними имеются в виду только те вершины, что являются генераторами, потому что только у них есть параметр приоритета. Для каждой пары вершин можно вычислить то, какая величина потока проходит от первой ко второй. В этом случае приоритет первой означает то, какую часть от значения приоритета второй вершины она должна получить.

Вернёмся к рассуждениям о метрических параметрах. Главное условие маршрутной сети – она должна являться связным графов. Бизнесу интересно то, чтобы можно было построить как можно меньше километров дорожного полотна, чтобы уменьшить затраты на строительство и амортизацию. Решение этой проблемы – минимальное остовное дерево, а значим будем вычислять эту метрику, основываясь на нём. Пусть единица – это сумма длин всех рёбер минимального остовного дерева. Тогда для вычисления метрики, определяющей экономность маршрутной сети, достаточно вычислить сумму рёбер нынешнего графа и разделить на сумму рёбер минимального остовного дерева для данных вершин.

Кроме того, бизнесу важно, чтобы из любого пункта в любой другой пункт можно было попасть за минимальное время. Так как скорость движения транспорта не зависит от построения маршрутной сети, задача алгоритма минимизировать расстояние между каждой вершиной. Оптимальное решение этой задачи – полный граф, так как в нём все вершины соединены на прямую. Так что за единица данной метрики – соответствует полному графу. Кроме того, чтобы далёкие и мало приоритетные вершины не влияли на данный показатель так же, как и вершины с большим приоритетом, вычисление строится таким образом:

* 1. Для каждой пары вершин строятся кратчайшие пути между ними;
  2. Для каждого из путей считается его длина, а также то, какой поток проходит по нему между двумя вершинами на его концах;
  3. Вычисляется то, во сколько раз длина пути между вершинами больше расстояния между ними, умноженное на то, какую часть от всей величины потока данной маршрутной сети составляет поток между этими двумя вершинами.
  4. Значения, вычисленные в третьем пункте, складываются, образуя значение метрики

Аналогично наиболее быстрой маршрутной сети, самым устойчивым графом будет являться полный граф за счёт того, что при удалении любого ребра останется достаточно много способов проложить пути между разделёнными вершинами через третью. При рассмотрении существующих карт путей было замечено, что инженеры не стоят свои дороги так, чтобы при удалении одного из ребер все пути, проходящие через него, перестроили так, чтобы не сильно уж и возрасти по расстоянию. Это связано с тем, что зачастую ломается лишь один железнодорожный путь в то время, как параллельный ему начинает обслуживать транспорт в обе стороны. Однако абсолютно все дороги построены таким образом, что при удалении любого ребра граф не разделится на две большие компоненты связности. Только лишь удалённые вершины с низким приоритетом подвержены такой опасности. Чтобы учитывать этот факт в виде метрики, было решено рассмотреть каждое ребро. Алгоритм рассматривает, является и связным граф, если исключить возможность проходить по данному ребру. В том случае, если граф делится на две компоненты связности, то все вершины в той компоненте, которая суммарно имеет меньший приоритет, помечаются как «опасные». Значение метрики равно единице плюс сумма приоритетов опасных вершин, делённая на сумму приоритетов всех вершин. Единица необходима, чтобы все метрики имели приближенные значения.

Последняя метрика связана с величиной потока. Любая маршрутная сеть должна предусматривать возможность роста, проходящего по ней потока. В случай с железнодорожным полотном, цель инженеров распределить всю нагрузку таким образом, чтобы даже в случае повышения суммарной величины потока, нигде не появлялось место, которое не способно больше принять на себя ещё нагрузку. Решением этой проблемы вновь будет полный граф – так как на каждое ребро будет приходиться поток только между двумя вершинами. Обозначим тогда за единицу величину потока между двумя наиболее приоритетными вершинами в полном графе. После этого найдём кратчайшие пути между каждой вершиной с приоритетом и к каждому ребру каждого пути добавим значение проходящего потока. Таким образом мы вычислили функцию нагрузки для каждого ребра маршрутной сети. Значением метрики будет являться то, во сколько раз наиболее загруженное ребро больше значения наиболее загруженного ребра полного графа.

## 2.4 Выделение лучшей маршрутной сети

В результате рассуждений были разработаны четыре метрики, для определения эффективности той или иной маршрутной сети в каждой из важных сфер. Стоит упомянуть, что данные метрики можно сравнивать только для тех графов, которые построены из одних и тех же генераторов. Они служат для того, чтобы из множества решений алгоритма слизевика выбрать лучшее.

Чтобы упростить задачу и предоставить пользователю удобное управление той сетью, что формируется алгоритм, было решено объединить все четыре вещественных значения в одно число путём умножения метрик на соответствующий приоритет, значение которого лежит от 0 до 1, и последующего суммирования каждой. Приоритет задаётся пользователем. Таким образом, на множестве всех решений задачи построения маршрутной сети с помощью алгоритма слизевика была определена функция сравнения, гибкость которой в зависимости от предметной области может определять пользователь. Однако, стоит упомянуть, что в данный момент функция задания приоритета метрика не реализована в программе.

# Разработка приложения на Flutter

## Необходимость разработки приложения

Ещё во время работы над прошлой курсовой работой было замечено, что удобство пользования программой оставляет желать лучшего. Для ввода настроек и проверки, всё ли правильно считалось, использовался txt файл и отдельная функция. Кроме того, часто были проблемы с тем, что было достаточно сложно то, что происходит внутри симуляции. Данная проблема была временно решена использованием bmp файлов, однако это явно не было серьёзным решением проблемы. В купе с тем, что без знания самого алгоритма было невозможно понять, что означают цифры, разделённые пробелами в файле настроек и осознанием необходимости в будущем представлять программу, было решено написать в рамках этой курсовой работы удобный для человеческого восприятия интерфейс.

Его основной задачей было не уменьшить функциональность нынешней реализации алгоритма и при этом представить удобство пользованием. Так как даже после всего множества модификаций алгоритма ещё виднеются пути, куда можно дальше улучшать программу, то и цели создать готовый продукт не ставилось. В связи с этим было решено разработать интерфейс на фреймворке Flutter и только для Windows.

## 3.2 Причина использования Flutter

Flutter является мощным и популярным фреймворком для разработки кроссплатформенных приложений. Он предлагает удобный и эффективный способ создания красивого и отзывчивого пользовательского интерфейса, который может быть запущен как на устройствах Windows, Linux, MacOS, Android, iOS и других [3]. Flutter использует язык программирования Dart, который является основным языком разработки приложений во фреймворке. Dart предлагает современный и эффективный синтаксис, поддержку асинхронного программирования и множество библиотек и инструментов для разработки мобильных приложений.

Flutter также обладает гибкостью и простотой в использовании при разработке различных видов элементов интерфейса. Разработчики могут легко настраивать внешний вид и поведение элементов, чтобы они соответствовали специфическим потребностям и требованиям приложения, в том числе и для отображения маршрутных сетей.

Кроме того, Flutter обеспечивает возможность создания высокопроизводительных приложений, так как он использует собственный движок рендеринга [3]. Архитектура Flutter строится на принципах эффективной отрисовки пользовательского интерфейса, управления состоянием и удобного API для разработки приложений. Но самой главной причиной, по которой было решено использовать Flutter – это библиотека FFI от разработчиков фреймворка, которая позволяет синхронно и асинхронно вызывать функции, написанные на языке С++ [4].

Таким образом можно с удобством воспользоваться всеми преимуществами Flutter-а для разработки интерфейса приложения, удовлетворяющего всем потребностям пользователя и при этом сохранить быстродействие и среду программирования алгоритмов от С++ в одном проекте без необходимости переписывать код.

## Описание функциональности приложения

На данный момент приложение имеет 4 основные части интерфейса, которых достаточно для того, чтобы использовать весь имеющийся функционал, необходимый для решения задачи построения маршрутной сети, за исключением возможности вывести результаты из программы.

Изображение выглядит как текст, снимок экрана, программное обеспечение, компьютер

Автоматически созданное описание

Рисунок 4 – Вид главного экрана программы вне рабочего состояния

Слева находится поле, на которую будет выводиться результат работы алгоритма построения маршрутной сети по множеству агентов для нынешнего шага итерации. Однако изначально это поле используется для установки на нём генераторов. Клик левой кнопкой мыши устанавливает генератор в пустой клетке, либо увеличивает его приоритет на единицу, клик правой кнопкой мыши уменьшает приоритет генератора или удаляет его, если приоритет становится равным нулю.

Справа находятся две вкладки TabBar, отвечающие за нынешние настройки симуляции. На вкладке с вершинами отображается список нынешних генераторов и их приоритет. В будущем будет добавлена возможность редактирования списка генераторов не только при помощи кликов мыши, но и вводом через файлы или же путём использования интерфейса приложения. На вкладке с расширенными настройками находятся все возможные настройки симуляции, которые раньше вводились через txt файл. Теперь они перед началом симуляции передаются во внутрь среды выполнения С++.

По центру находится главная панель управления приложением. Поле ввода необходимо для того, чтобы указать какое количество итераций должна сделать программа. Число внутри этого поля каждый шаг будет уменьшаться. За каждый шаг программа выполняет 10 итераций алгоритма слизевика, после чего применяет алгоритм построения маршрутной сети.

Кнопка «Выполнить» загружает список генераторов и все настройки в программу, после чего запускает симуляцию. Также нажатие на эту кнопку переводит приложение в режим работы, который блокирует возможность изменения настроек, закрывая их Drawer-ом справа, на котором расположено ещё одно поле. На нём изображается наилучшая, согласно вычисленным метрикам, маршрутная сеть.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание

Рисунок 5 – Вид главного экрана программы во время рабочего состояния

Кнопка «Остановить» прекращает работу симуляции, однако не сбрасывает её. Кнопка «Сбросить» полностью обнуляет работу программы. Работа с состояниями настроена так, что любая последовательность нажатий кнопок не может вызвать ошибку. Внизу выводятся метрики наилучшей на данный момент маршрутной сети, изображённой справа.

В виду того, что выполнение алгоритма требует достаточно много процессорного времени, при помощи средств языка Dart создаётся отдельный изолят, имеющего свою область памяти и свой процессорный поток, внутри которого и запускаются итерации. Благодаря этому интерфейс не зависает при работе алгоритма и продолжает работать без провисаний. К сожалению, по этой причине не получилось воспользоваться библиотекой OMP для С++ для организации многопоточности, ибо Flutter выделяет вызываемым через FFI функциям лишь один поток [4]. Однако, скорее всего это временная трудность, которую можно решить, немного изменив код и объединив несколько изолятов Dart-а.

# ЗАКЛЮЧЕНИЕ

В данной курсовой работе было представлено десктопное приложение на фреймворке Flutter, разработанное для решения задачи построения оптимальной маршрутной сети. Целью работы было создание приложения с удобным интерфейсом, позволяющего с помощью алгоритма слизевика, загрубления графа, минимизации маршрутной сети вычислять чёткие математические метрики, чтобы в итоге показывать пользователю наилучший результат работы алгоритмов.

В процессе выполнения работы были разработаны алгоритмически оптимизированные алгоритмы построения маршрутной сети по результатам работы алгоритма слизевика и математические метрики для определения функции сравнения на множестве маршрутных сетей. Были изучены основные критерии оценки маршрутных сетей, особенности работы с библиотекой FFI фреймворка Flutter. Также были освещены принципы проектирования пользовательского интерфейса, организации бизнес-логики и взаимодействия с данными.

В ходе разработки приложения были поставлены и успешно решены следующие задачи:

1) Изучение и применены для разработки метрик потребности бизнеса в задаче построения маршрутной сети и методы алгоритмического построения оных.

2) Реализованы алгоритмы построения маршрутной сети и методы вычисления метрических показателей;

3) Создано десктопное приложение для удобного управления работы алгоритма.

Однако, как любой проект, разработка данного приложения также имеет потенциал для дальнейшего улучшения и расширения, причём достаточно обширный. С появлением математических строгих показателей результативности можно вновь вернуться к настройкам алгоритма слизевика с целью оценки пользы или вреда вводимых механик. Новые параметры, которые появились вместе с алгоритмами загрубления и минимизации графов ещё не проверялись на практике, чтобы определить какие настройки являются более оптимальными. Кроме того, подбор самих параметров, которые вводятся в программу, можно попытаться автоматизировать, чтобы минимизировать роль человека в поиске оптимального решения.

С появлением нормального и удобного интерфейса появилась возможность выводить на экран любую полезную или интересную информацию, например, карта следа локации или историю изменений карты агентов.

Кроме того, всегда можно вернуться к написанному коду алгоритмов и провести дополнительный рефакторинг, чтобы уменьшить его время работы. Также в будущем стоит разобраться с проблемой потери многопоточности из-за особенностей работы библиотек.

В итоге, разработанное десктопное приложение на фреймворке Flutter для решения задачи построения маршрутной сети предоставляет удобный и функциональный инструмент для проектирования оных и в будущем может вполне использоваться на реальных задача после того, как пройдёт испытания и улучшения.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Формирование маршрутных сетей городского общественного транспорта // Текст : электронный // Научная электронная библиотека «КиберЛенинка» : официальный сайт. –– 2021 –– URL: https://cyberleninka.ru/article/n/formirovanie-marshrutnyh-setey-gorodskogo-obschestvennogo-transporta (дата обращения 05.04.2023).

2. Графы, процесс загрубления, кластеризация // Текст : электронный // Сообщество IT специалистов : официальный сайт. –– 2022 –– URL: http://vbystricky.ru/2021/07/graph\_coarsing\_clustering.html (дата обращения 17.04.2023).

3. Flutter documentation // Текст : электронный // Сообщество IT специалистов : официальный сайт. –– 2023 –– URL: https://docs.flutter.dev/ (дата обращения 10.05.2023).

4. Flutter FFI Practice // Текст : электронный // Сообщество IT специалистов : официальный сайт. –– 2023 –– URL: https://www.programmersought.com/article/267810188709/ (дата обращения 12.05.2023).

# ПРИЛОЖЕНИЕ A

# Основные функции загрубления, минимизации графов и функции вычисления метрических показателей

#include "SlimeMoldClass.h"

void AgentGraphAnalyser::makeGraph(vector<SlimeAgent\*> particles, vector<Generator\*> generators) {

clear();

// добавляем все точки и генераторы в однин вектор, различая их с помощью второго параметра

vector<pair<pair<it,it>, it>> position;

for (auto u : particles) {

position.push\_back(make\_pair(make\_pair(u->pixelVector[0], u->pixelVector[1]), 0));

}

for (auto u : generators) {

position.push\_back(make\_pair(make\_pair(u->position[0], u->position[1]), u->agentsPerStep));

}

sort(position.begin(), position.end());

// делаем из вектора односвязный список. В i-том элементе находится индекс вектора следующего активного элемента

vector<it> next(position.size(), -1);

for (int i = 0; i < position.size()-1; i++) {

next[i] = i + 1;

}

// количество точек, которые забрала данная точка

vector<it> pointMass(position.size(), 1);

bool hasChange = true;

while (hasChange) {

hasChange = false;

for (int i = 0; i != -1; i = next[i]) {

int prevIndex = i;

for (int j = next[i]; j != -1; j = next[j]) {

pair<it, it> firstPoint = position[i].first;

pair<it, it> secondPoint = position[j].first;

bool isNeedDelete = false;

//благодаря сортировке из n^2 переходим в n\*log(n) в случае с нашими графами

if (abs(firstPoint.first - secondPoint.first) > vertexRange) {

break;

}

// если точки достаточно близко, стираем точку j

if (distance(firstPoint, secondPoint) <= vertexRange) {

//если вторая точка - город, то мы не можем её убрать. Чтобы в итоге получился связный граф, мы просто останавливаем обработку этой точки. Проблема лишь в бОльшем количестве точек, которая решится в некст функции

if (position[j].second) {

// в случае, если обе точки стоят в одном месте, то их стоит поменять местами, после чего слить воедино.

if (firstPoint.first == secondPoint.first && firstPoint.second == secondPoint.second) {

it tmp = position[j].second;

position[j].second = position[i].second;

position[i].second = tmp;

}

else {

break;

}

}

isNeedDelete = true;

hasChange = true;

}

if (isNeedDelete) {

next[prevIndex] = next[j];

pointMass[i] += pointMass[j];

}

else {

prevIndex = j;

}

}

}

}

vector<pair<it,it>> exitPoints;

vector<it> towns;

for (int i = 0; i != -1; i = next[i]) {

if (pointMass[i] >= minVertexMass || position[i].second) {

exitPoints.push\_back(position[i].first);

towns.push\_back(position[i].second);

}

}

// дебаг count

int count = 0;

vector<vector<it>> exitGraph(exitPoints.size(), vector<it>());

for (int i = 0; i < exitPoints.size(); i++) {

for (int j = i + 1; j < exitPoints.size(); j++) {

if (abs(exitPoints[i].first - exitPoints[j].first) > edgesRange) {

break;

}

ft tmpDistance = distance(exitPoints[i], exitPoints[j]);

if (tmpDistance <= edgesRange) {

exitGraph[i].push\_back(j);

exitGraph[j].push\_back(i);

count++;

}

}

}

this->towns = towns;

this->graph = exitGraph;

this->exitPoints = exitPoints;

}

void AgentGraphAnalyser::minimizeGraph() {

// если нужно что-то удалить, то запоминаем номер

vector<it> vertexToDelete;

// TODO можно переписать на односвязный список, чтобы меньше по памяти было

vector<bool> vertexWhatDeleted(exitPoints.size(), false);

// вначале чекаем все вершины, потом те вершины, что были затронуты удалением, потом те вершины, что были затронуты ещё одним удалением...

queue<it> vertexToCheck;

for (int i = 0; i < exitPoints.size(); i++) {

vertexToCheck.push(i);

}

while (!vertexToCheck.empty()) {

int i = vertexToCheck.front(); vertexToCheck.pop();

if (!vertexWhatDeleted[i]) {

if (!towns[i]) {

// одинокая вершина

if (graph[i].size() <= 1) {

vertexToDelete.push\_back(i);

vertexWhatDeleted[i] = true;

if (graph[i].size() == 1) {

eraseEdge(graph[i][0], i);

vertexToCheck.push(graph[i][0]);

}

}

// треугольник

else if (graph[i].size() == 2 && findEdge(graph[i][0], graph[i][1])) {

it firstIndex = graph[i][0];

it secondIndex = graph[i][1];

vertexToDelete.push\_back(i);

vertexWhatDeleted[i] = true;

eraseEdge(firstIndex, i);

vertexToCheck.push(firstIndex);

eraseEdge(secondIndex, i);

vertexToCheck.push(secondIndex);

}

// ломаная

else if (graph[i].size() == 2 && canConnectEdges(i)) {

it firstIndex = graph[i][0];

it secondIndex = graph[i][1];

// соединяем

vertexToDelete.push\_back(i);

vertexWhatDeleted[i] = true;

// удаляем рёбра с i-й вершиной

eraseEdge(firstIndex, i);

eraseEdge(secondIndex, i);

// если вершины не соединены, соединяем

if (!findEdge(firstIndex, secondIndex)) {

graph[firstIndex].push\_back(secondIndex);

graph[secondIndex].push\_back(firstIndex);

}

// проверим эти вершины

vertexToCheck.push(firstIndex);

vertexToCheck.push(secondIndex);

}

}

// ромб, Причём i-я вершина может быть городом

{

vector<it> rombVertex = checkRect(i);

if (!rombVertex.empty()) {

// если обе диалогнали есть

if (findEdge(i, rombVertex[2])) {

int townCount = 0;

for (auto u : rombVertex) {

if (towns[u]) {

townCount++;

}

}

townCount += towns[i] != 0 ? 1 : 0;

if (townCount >= 2) {

continue;

}

else {

// если находим хотя бы один город - фиксируем положение вершины

bool hasTown = towns[i];

// делаем смежную вершину из четырёх

for (int k: rombVertex) {

// образуем вершину по середине или с центром в городе

if (!hasTown) {

if (towns[k]) {

exitPoints[i] = exitPoints[k];

towns[i] = towns[k];

towns[k] = 0;

hasTown = true;

}

else {

exitPoints[i] = average(exitPoints[i], exitPoints[k]);

}

}

for (int v: graph[k]) {

// соединяем первую вершину со смежными вершинами

if (i != v && !findEdge(i, v)) {

graph[i].push\_back(v);

graph[v].push\_back(i);

}

// стираем рёбра к k-вершине и проверяем все смежные вершины.

eraseEdge(v, k);

vertexToCheck.push(v);

}

vertexToDelete.push\_back(k);

vertexWhatDeleted[k] = true;

}

}

}

else {

if (towns[rombVertex[0]] && towns[rombVertex[1]]) {

continue;

}

// делаем смежную вершину из двух, соединённых диагональю, изменяя первую и удаляя вторую

it firstIndex = -1;

it secondIndex = -1;

if (towns[rombVertex[1]]) {

firstIndex = rombVertex[1];

secondIndex = rombVertex[0];

}

else {

firstIndex = rombVertex[0];

secondIndex = rombVertex[1];

}

// если они не города, то образуем вершину по середине

if (!towns[firstIndex] && !towns[secondIndex]) {

exitPoints[firstIndex] = average(exitPoints[firstIndex], exitPoints[secondIndex]);

}

// для каждой смежной вершины: соединяем её с первой вершиной, удаляем рёбра со второй вершиной и проверяем их в последствии

for (int j: graph[secondIndex]) {

if (firstIndex != j && !findEdge(firstIndex, j)) {

graph[firstIndex].push\_back(j);

graph[j].push\_back(firstIndex);

}

eraseEdge(j, secondIndex);

vertexToCheck.push(j);

}

vertexToDelete.push\_back(secondIndex);

vertexWhatDeleted[secondIndex] = true;

}

}

}

}

}

// нужно из исходного графа удалить все вершины, что были слиты. Для этого пересобираем меньший граф и сохраняем, для какого индекса сколько уже было удалённых вершин

// чтобы после просто вычесть это числа из номера ребра. sort(vertexToDelete.begin(), vertexToDelete.end());

// удаление выполняется за O(e+n)

vector<it> shiftVector;

vector<vector<it>> newGraph;

vector<pair<it, it>> newExitPoints;

vector<it> newTowns;

vector<it> townIndexes;

int lastDeleteIndex = 0;

sort(vertexToDelete.begin(), vertexToDelete.end());

for (int i = 0; i < exitPoints.size(); i++) {

if (lastDeleteIndex != vertexToDelete.size() && i == vertexToDelete[lastDeleteIndex]) {

lastDeleteIndex++;

}

else {

newExitPoints.push\_back(exitPoints[i]);

newGraph.push\_back(graph[i]);

newTowns.push\_back(towns[i]);

if (towns[i]) {

townIndexes.push\_back(newTowns.size()-1);

}

}

// запоминаем сдвиг

shiftVector.push\_back(lastDeleteIndex);

}

// Пример: было ребро 1-3 и 3-1. Удалили вершину 2. Теперь это будет ребро 1-2 2-1. Ребро 2-1 само становится на место, так как мы пропустили прошлую вершину 2 в новом графе.

// Ребро 1-2 мы должны сделать из ребра 1-3 тем, что из "3" вычтем кол-во удалённых до неё вершин - т.е. 1

for (int i = 0; i < newGraph.size(); i++) {

for (int j = 0; j < newGraph[i].size(); j++) {

newGraph[i][j] -= shiftVector[newGraph[i][j]];

if (newGraph[i][j] == i) {

cout << 1;

}

}

}

this->towns = newTowns;

this->graph = newGraph;

this->townIndexes = townIndexes;

this->exitPoints = newExitPoints;

}

ft AgentGraphAnalyser::calculateWeigth() {

if (weigthGraph.empty()) {

buildWeigth();

}

ft minWeigth = 0;

// первое число - длина ребра, второе - вершина, куда идёт ребро

priority\_queue<pair<ft, it>> edges;

vector<int> visited(townIndexes.size(), 0); visited[0] = 1;

for (it i = 1; i < townIndexes.size(); i++) {

edges.push(make\_pair(-distance(exitPoints[townIndexes[0]], exitPoints[townIndexes[i]]), i));

}

int countOfVertex = townIndexes.size()-1;

while (countOfVertex) {

auto edge = edges.top(); edges.pop();

if (visited[edge.second]) {

continue;

}

minWeigth -= edge.first;

countOfVertex--;

visited[edge.second] = 1;

// фу так делать через for

for (it i = 1; i < townIndexes.size(); i++) {

if (!visited[i]) {

edges.push(make\_pair(-distance(exitPoints[townIndexes[edge.second]], exitPoints[townIndexes[i]]), i));

}

}

}

ft result = 0;

for (int i = 0; i < graph.size(); i++) {

for (int j : graph[i]) {

if (i < j) {

result += weigthGraph[i][j];

}

}

}

result /= minWeigth;

return result;;

}

ft AgentGraphAnalyser::calculateOverDistance() {

if (waysGraph.empty()) {

buildWays();

}

ft waysCount = 0; ft overMult = 0;

for (int i = 0; i < waysGraph.size(); i++) {

// проходимся лишь по одной копии пути

for (int j = i + 1; j < waysGraph[i].size(); j++) {

ft minDistance = distance(exitPoints[townIndexes[i]], exitPoints[townIndexes[j]]);

ft nowDistance = 0;

ft waysFlow = ft(towns[townIndexes[i]] \* towns[townIndexes[j]]) / (sumOfPriority() - towns[townIndexes[i]]);// поток из i в j

waysFlow += ft(towns[townIndexes[j]] \* towns[townIndexes[i]]) / (sumOfPriority() - towns[townIndexes[j]]);// поток обратно

it first;

it second = waysGraph[i][j][0];

for (int k = 1; k < waysGraph[i][j].size(); k++) {

first = second;

second = waysGraph[i][j][k];

nowDistance += weigthGraph[first][second];

}

waysCount++;

overMult += nowDistance \* waysFlow / minDistance;

}

}

overMult /= sumOfPriority();

return overMult;

}

ft AgentGraphAnalyser::calculateResistance() {

//it maxCycle = 0;

vector<int> stressed(graph.size(), 0);

// выбираем все возможные рёбра. Удаляем это ребро и пытается найти путь из i в j. Если путь можно найти, то получается, что мы имеем цикл, иначе мост

for (int i = 0; i < graph.size(); i++) {

for (it j : graph[i]) {

if (i >= j) {

continue;

}

ft nowSum = sumOfPriority() - towns[i];

vector<int> visited(graph.size(), 0); visited[i] = 1;

queue<pair<it, it>> vertex;

// заранее проходимся по всем рёбрам из i, чтобы исключить условие на проверку ребра i j

for (it k : graph[i]) {

if (k != j) {

vertex.push(make\_pair(k, 1));

visited[k] = 1;

nowSum -= towns[k];

}

}

it index;

it size;

while (vertex.size()) {

index = vertex.front().first;

size = vertex.front().second; vertex.pop();

for (it k : graph[index]) {

/\*if (k == j) {

visited[j] = 1;

goto endwhile;

}\*/

if (!visited[k]) {

vertex.push(make\_pair(k, size + 1));

visited[k] = 1;

nowSum -= towns[k];

}

}

}

//endWhile:

bool isConnected = true;

for (int k = 0; k < graph.size(); k++) {

if (!visited[k]) {

isConnected = false;

break;

}

}

//// если смогли прийти в j вершину, значит есть цикл

//if (visited[j]) {

// maxCycle = max(maxCycle, size + 1);

//}

//// иначе граф уже не связный и значит i j - мост

//else

if (!isConnected)

{

// если мы прошли меньшую часть из компонент связностей, то все посещённые вершины считаем в зоне риска, иначе все непосещённые

bool smallerPart = nowSum > sumOfPriority() / 2;

for (int k = 0; k < visited.size(); k++) {

if (smallerPart == visited[k]) {

stressed[k] = 1;

}

}

}

}

}

ft sumOfStressedPriority = 0;

for (int i = 0; i < towns.size(); i++) {

if (stressed[i]) {

sumOfStressedPriority += towns[i];

}

}

// добавляем +1 потому, что все метрики больше 1

return 1 + sumOfStressedPriority/ sumOfPriority();

}

ft AgentGraphAnalyser::calculateDeltaFlow() {

if (flowGraph.empty()) {

buildFlow();

}

ft maxPriority = max(towns[0], towns[1]);

ft secondMaxPriority = min(towns[0], towns[1]);

for (int i = 0; i < towns.size(); i++) {

if (towns[i] > maxPriority) {

secondMaxPriority = maxPriority;

maxPriority = towns[i];

}

else if (towns[i] > secondMaxPriority) {

secondMaxPriority = towns[i];

}

}

ft minMaxEdgeFlow = maxPriority + secondMaxPriority;

ft maxEdgeFlow = 0;

for (int i = 0; i < graph.size(); i++) {

for (int j : graph[i]) {

if (i < j) {

if (flowGraph[i][j] > maxEdgeFlow) {

maxEdgeFlow = flowGraph[i][j];

}

}

}

}

ft result = maxEdgeFlow / minMaxEdgeFlow;

return result;

}