МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение
высшего образования
«Тольяттинский государственный университет»

Институт Математики, физики и информационных технологий
(наименование института полностью)

Кафедра «Прикладная математика и информатика»
(наименование)

01.03.02 Прикладная математика и информатика
(код и наименование направления подготовки, специальности)

Компьютерные технологии и математическое моделирование
(направленность (профиль)/специализация)

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА (БАКАЛАВРСКАЯ РАБОТА)

на тему Разработка алгоритма поиска оптимального городского маршрута с использованием эволюционных вычислений

Обучающийся Е. С. Пичугина
(И.О. Фамилия) (личная подпись)

Руководитель к.т.н., В. С. Климов
(ученая степень, звание, И.О. Фамилия)

Консультант Т. С. Якушева
(ученая степень, звание, И.О. Фамилия)

Тольятти 2022

Аннотация

Тема бакалаврской работы: «Разработка алгоритма поиска оптимального городского маршрута с использованием эволюционных вычислений».

В данной бакалаврской работе исследуется вопрос программной реализации эволюционного алгоритма в задаче поиска кратчайшего маршрута.

Объект исследования – поиск оптимального маршрута.

Предмет – разработка алгоритма поиска оптимального городского маршрута с использованием эволюционных вычислений.

Целью данной выпускной квалификационной работы является проектирование и разработка алгоритма маршрутизации в городских условиях с использованием эволюционных вычислений.

Структура бакалаврской работы представлена введением, тремя разделами, заключением, списком литературы и приложением.

Во введении раскрывается актуальность выбранной темы, описываются цель и задачи, которые необходимо решить.

В первом разделе рассматриваются существующие решения задачи и рассматриваются технологии ее решения.

Во втором разделе описываются реализация алгоритма поиска кратчайшего пути и результаты тестирования.

В третьем разделе проводится анализ полученных данных и вносятся корректировки в разработанный алгоритм.

В заключении представляются выводы по проделанной работе.

В работе – 68 страниц, содержащих 71 рисунок, 8 таблиц, 3 раздела, заключение, 1 приложение, 15 библиографических источников.

Abstract

The topic of the graduation work is «Development of an algorithm for finding the optimal urban route using evolutionary calculations».

The senior paper consists of an introduction, three parts, a conclusion, tables, list of references including foreign sources and an appendix.

In this bachelor's work, the issue of software implementation of the evolutionary algorithm in the problem of finding the shortest route is investigated.

The aim of the work is to design and development of a routing algorithm in urban environments using evolutionary computing.

The object of the graduation work is the search for the optimal route.

The subject is the development of an algorithm for finding the optimal urban route using evolutionary calculations.

The graduation work may be divided into several logically connected parts which are: analysis of existing solutions; rationale for choosing an algorithm; program development; testing the developed algorithm and comparing the results with another algorithm; summarizing.

Overall, the results suggest that the genetic algorithm strongly depends on the parameters, and even a small change in one of them can lead to both a strong deterioration and an improvement in the obtained solutions. Based on the performance estimates of the genetic algorithm obtained as a result of testing, it will be possible to improve it in the future in order to speed up its work while maintaining the accuracy of the solutions obtained.

Содержание

Введение

В современных городах, особенно в крупных, чтобы добраться из одной точки в другую, можно воспользоваться несколькими вариантами маршрута. Если человеку некуда спешить, то можно выбрать любой из них, а что, если наоборот: ему необходимо добраться до назначенного пункта как можно быстрее или выбрать наиболее дешевый маршрут? В таком случае ему предстоит просмотреть несколько маршрутов вручную и выбрать наиболее подходящий вариант.

Для решения таких транспортных задач используются приложения, которые автоматически строят маршрут до указанного места. Однако они чаще всего предоставляют своим пользователям самые короткие маршруты, но это не значит, что они самые быстрые или же менее затратные.

Просчет маршрута с учетом не только расстояния, но и затраченного на него времени, а также средств – это трудноreализуемая чисто математическая задача. Для решения такого рода задач часто применяют эволюционные алгоритмы [15][13].

Эволюционные вычисления основаны на идеи применения дарвинской эволюции к компьютерной программе, концепция выживания которой заключается в том, что выживает наиболее приспособленная особь. Задачей эволюции является с помощью случайных мутаций в результате прийти к решению задачи с нужной точностью [1][14][12].

Эволюционные алгоритмы разделены на несколько видов:

- генетические алгоритмы;
- эволюционное программирование;
- программирование экспрессии генов;
- дифференциальная эволюция;
- нейроэволюция.

Все они используют базовые эволюционные этапы – отбор, скрещивание и мутации. Поведение особи (агента) определяется

окружающей средой в результате чего определяется его пригодность к ней. Особи с подходящей приспособленностью участвуют в размножении, формируя следующее поколение (популяцию). Такие алгоритмы относятся к адаптивным поисковым механизмам [24].

Эволюционные алгоритмы используются при поиске оптимального решения задачи в конечном множестве решений, например при решении NP-трудных задач, таких как задача коммивояжера, задача упаковки ранца и зарисовка графов. NP-трудные задачи – это задачи, оптимальное решение которых невозможно вычислить за разумное время никакими способами и методами, но с помощью эволюционных алгоритмов стало возможным получение приблизительного оптимального решения. Обычно оптимальность полученного результата оценивается приспособленностью, которая должна быть не ниже установленного значения. Наглядным примером успешности работы эволюционных вычислений является космическая антенна NASA. Задача состояла в определении формы антенны такой, чтобы передаваемый ею сигнал был как можно лучше. Ее пытались решить точными методами, но только эволюционный алгоритм смог найти оптимальную форму [17].

Объект исследования – поиск оптимального маршрута.

Предмет – разработка алгоритма поиска оптимального городского маршрута с использованием эволюционных вычислений.

Целью данной выпускной квалификационной работы является проектирование и разработка алгоритма маршрутизации в городских условиях с использованием эволюционных вычислений.

1 АНАЛИЗ ТЕХНОЛОГИЙ ПОИСКА КРАТЧАЙШЕГО ПУТИ

1.1 Описание исследуемой задачи

Есть различные задачи по нахождению наилучшего маршрута, которые решаются в таких приложениях, как 2ГИС и Яндекс.Карты, которые далее будут рассмотрены.

Воспользуемся 2ГИС, указав, что нам нужно добраться в городе Самара из Московского шоссе 43 до Гагарина 99. В результате приложение показывает пользователю самый быстрый по его расчету маршрут с учетом пробок на дороге. Данные о ситуациях на дорогах собираются с помощью пользователей приложения – чем их больше, тем точнее будет информация.

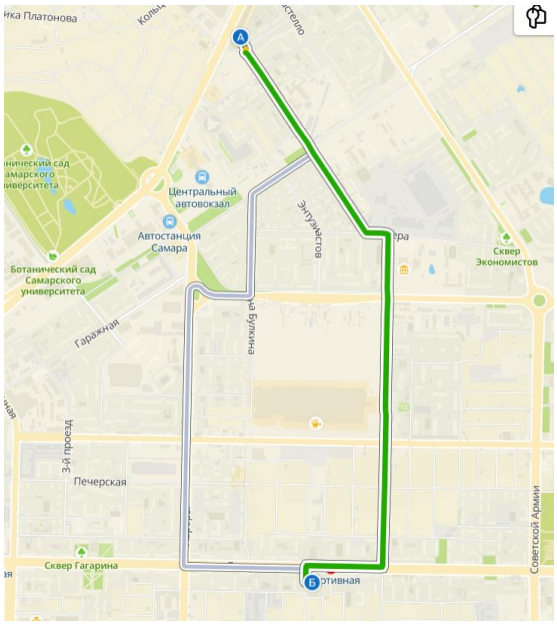

Рисунок 1.1.1 – Предложенные 2ГИС автомобильные маршруты

Так же в 2ГИС доступны фильтры, где можно указать желаемый вид транспорта.

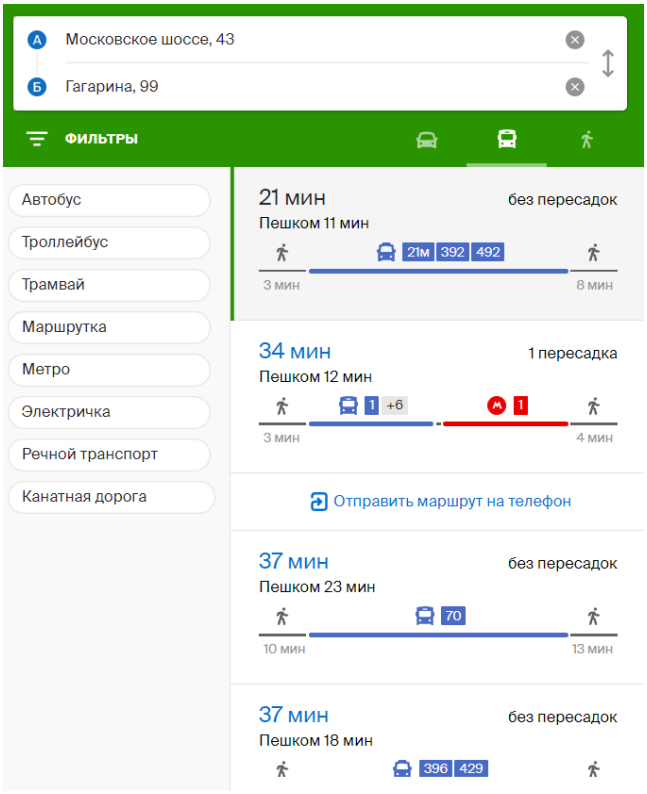

Рисунок 1.1.2 – Фильтры в 2ГИС

Далее воспользуемся приложением Яндекс.Карты с таким же запросом и увидим, что результат такой же – оба приложения предложили одинаковый маршрут.

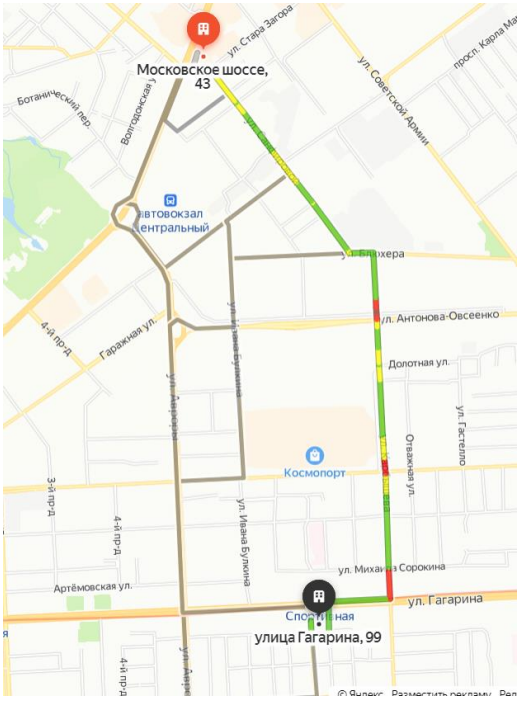

Рисунок 1.1.3 – Построенный Яндекс.Карты маршрут

Это приложение тоже предоставляет выбор желаемого вида транспорта.

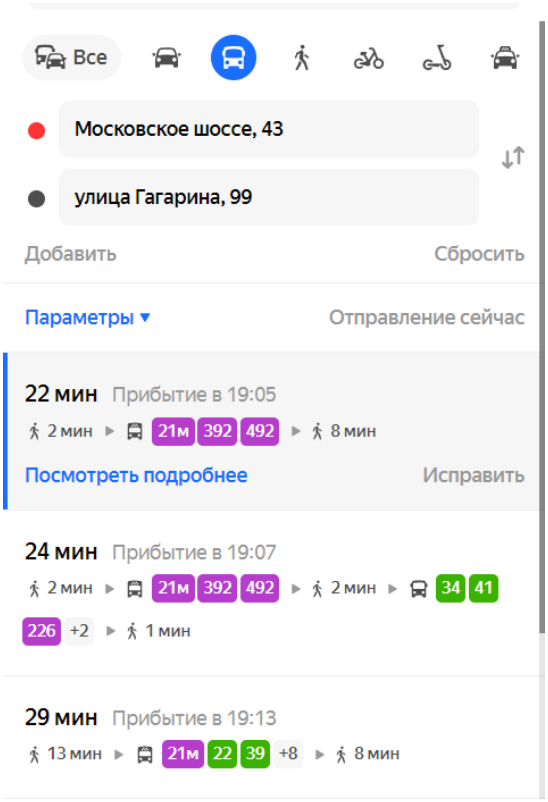

Рисунок 1.1.4 – Фильтры Яндекс.Карты

В целом видно, что два этих приложения работают похоже и выполняют свою главную задачу – строят самый быстрый маршрут с учетом выбранного транспорта.

Задача маршрутизации – это типичная задача компьютерной оптимизации, относящаяся к классу NP-трудных задач, в которых сложность вычисления ответа задачи экспоненциально зависит от размера входных данных. В таких случаях возможно использование эвристических методов, которые находят приближенные решения, однако их точность достаточно точна. Такого рода задачи являются ключевыми в областях транспортных перевозок и перемещения [4].

Таким образом, используя эволюционные вычисления, можно расширить список возможных параметров для пользователей, предоставив им больше выбора при составлении своего маршрута. Использование эвристических методов позволит не только улучшить фильтры приложений, но и сделать это без сильного увеличения времени, необходимого на расчеты

– приближенные результаты все равно будут достаточно точны для обычных пользователей.

Известно, что алгоритм Дейкстры является ядром в работе таких навигационных приложений, как Google Maps, Apple maps, OpenStreetMap, Яндекс.Карты и многих других [7][22]. Данный алгоритм не относится к эволюционным, а суть его работы заключается в прохождении всех узлов графа. Чем больше вариантов при построении маршрута, тем дольше будет работать алгоритм. Он так же не способен будет решить задачу коммивояжера с большим количеством данных. Суть задачи коммивояжера состоит в том, чтобы найти такой кратчайший маршрут, начинающийся в неком стартовом городе и заканчивающийся тоже в нем, в котором все города будут посещены только один раз. Поэтому на практике для ускорения работы алгоритма Дейкстры на больших графах его используют с модификациями [23]. Например, вместо того, чтобы алгоритм отрабатывал один раз от начальной позиции к месту назначения, он начинается с каждого конца и расширяется по обе стороны, пока они не встретится посередине, что в итоге устраняет примерно половину работы. Чтобы не исследовать закоулки каждого города между начальным местом и пунктом назначения, можно иметь несколько слоев картографических данных: слой «шоссе», который содержит только шоссе, вторичный слой, который содержит только второстепенные улицы, и так далее. Затем исследуются только меньшие участки более подробных слоев, расширяя их по мере необходимости.

Яндекс.Карты и 2ГИС достаточно эффективны внутри города и между городами (но тогда становятся недоступны общественные виды транспорта при построении пути), однако при расширении их работы и фильтров в них, могут возникать ситуации, похожие на NP-трудные задачи с решением которых они уже не будут столь эффективны. Чтобы этого избежать, можно использовать эволюционные вычисления, которые эффективно могут построить маршрут как внутри города, так и между ними.

Таким образом, целью моей дипломной работы является с помощью эволюционных вычислений решить задачу маршрутизации в городских условиях. Эффективность работы алгоритма будет оцениваться временем работы, т. е. производительностью и точностью его расчетов, а также эти результаты будут сравниваться с уже используемым в таких задачах алгоритмом Дейкстры, но без его модификаций.

1.2 Анализ существующих решений

Для просчитывания маршрута существует несколько вариантов алгоритмов, как, например, поиск в ширину, который ищет кратчайший путь между двумя объектами и определяет, существует ли путь между ними [9][21]. Предположим, пользователь находится в точке А и хочет добраться до места Б, собираясь доехать на автобусе с наименьшим количеством пересадок. Решим задачу с помощью алгоритма поиска в ширину [2]. При построении маршрута система дорог представляется в программе в виде дорожного графа (сетки дорог), с которым дальше уже работает алгоритм. Для этого представим возможные варианты на графе ниже, где узлы обозначают пересадки.

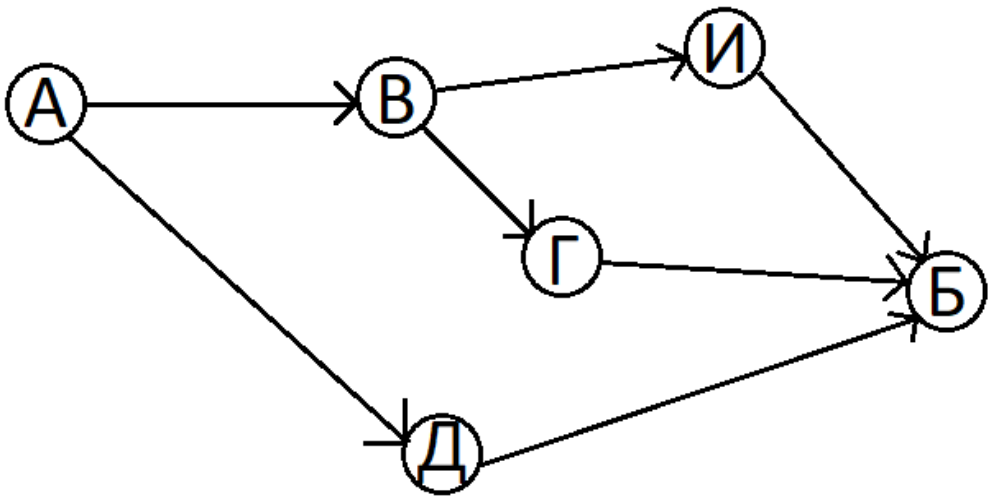

Рисунок 1.2.1 – Ориентированный граф

На рисунке 1.2.2 закрашены все вершины, в которые можно перейти за один шаг из вершины А.

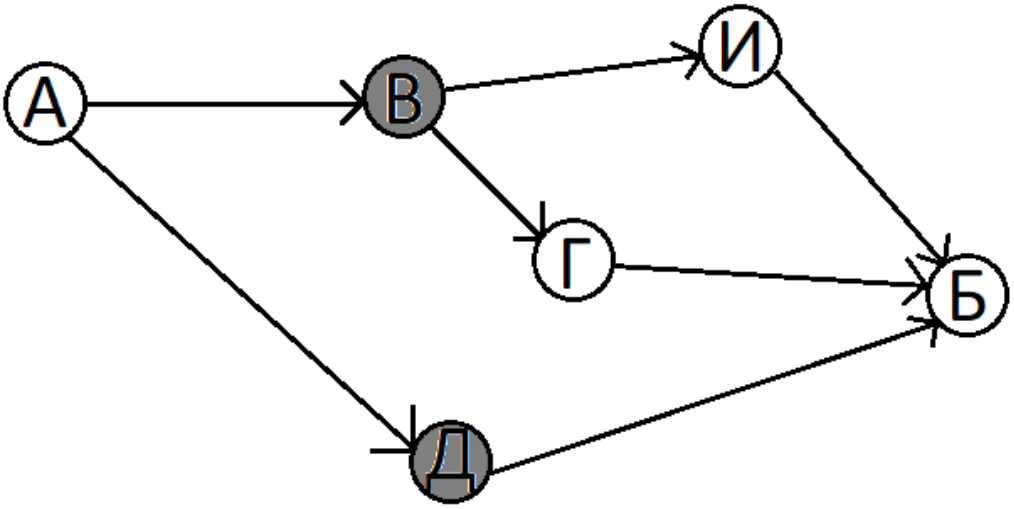

Рисунок 1.2.2 – Граф с выделенными вершинами, до которых можно добраться за 1 шаг из А

Поскольку вершина Б так и не была достигнута, это означает, что за один шаг невозможно добраться из А в Б. На рисунке 1.2.3 представлен граф с выделенными вершинами, до которых можно добраться за два шага от узла А.

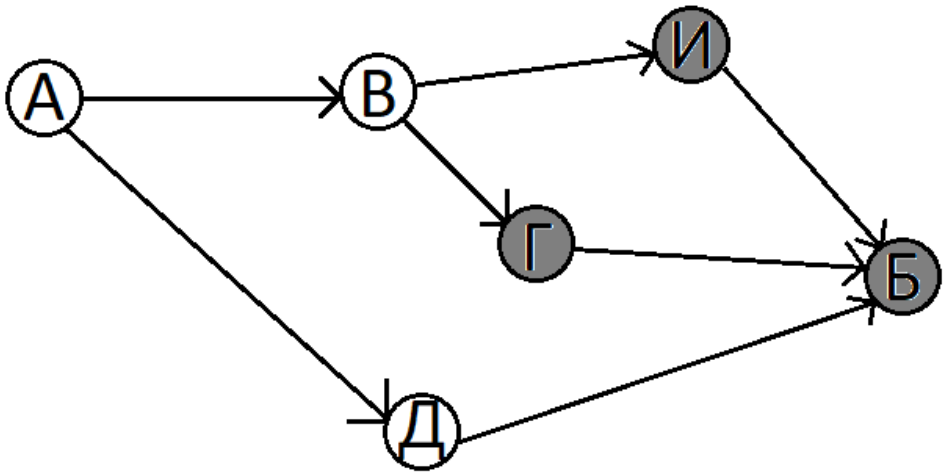

Рисунок 1.2.3 – Граф с выделенными вершинами, до которых можно добраться за два шага от вершины А

На этот раз вершина Б была достигнута, а это значит, что для того, чтобы добраться из А в Б нужно сделать минимум два шага. Есть и другие пути, которые так же приведут к узлу Б, но они длиннее. Таким образом,

алгоритм поиска в ширину просчитал, что кратчайший путь от А к Б потребует две пересадки.

Однако у данного алгоритма есть существенный недостаток: найденный путь может быть не самым быстрым. Меньшее количество пересадок не гарантирует, что в итоге маршрут будет короче и быстрее. Если учесть, что ребра графа будут обозначены продолжительностью перемещения, то тогда можно будет вычислить более быстрый маршрут. Для этого уже будет использоваться другой алгоритм. Например, всем известный продукт Яндекс.Карты, с помощью которого прокладывать маршруты, использует при построении пути алгоритм Дейкстры. То есть для вычисления кратчайшего пути в невзвешенном графе используют поиск в ширину, а во взвешенном – алгоритм Дейкстры. Далее рассмотрим принцип работы второго. На рисунке 1.2.4 изображен граф с отмеченными на ребрах времени, которое будет занимать данный сегмент на его прохождение.

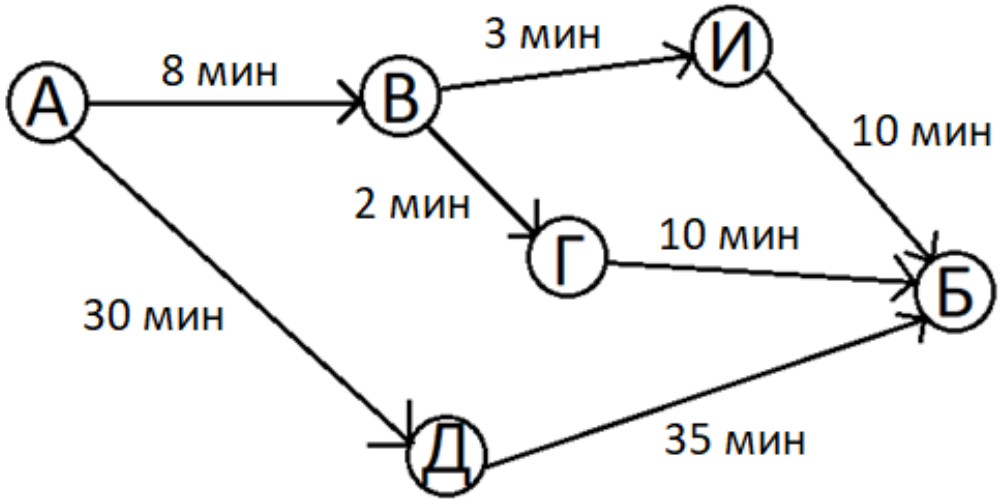

Рисунок 1.2.4 – Взвешенный граф с отмеченными продолжительностями перемещения

Сначала находится узел с наименьшей стоимостью. От вершины А до вершины Д путь займет 30 минут, а до вершины В – 8 минут. Поскольку пока неизвестно время достижения других узлов, то оно считается бесконечным. Путь до узла В пока считается более быстрым.

Таблица 1.2.1 – Результат первого шага алгоритма Дейкстры

Узел	Время до узла
В	8
Д	30
И	∞
Г	∞
Б	∞

На втором шаге вычисляется необходимое время до соседей В при переходе по ребру из вершины В.

Таблица 1.2.2 – Результат второго шага алгоритма Дейкстры

Узел	Время до узла
В	8
Д	30
И	11
Г	10
Б	∞

В результате были найдены более быстрые пути до вершин И и Г, теперь время до них сокращено от ∞ до 11 и 10 минут соответственно. Их время обновилось в таблице 1.2.2.

На следующем шаге переходим к рассмотрению соседей узла с наименьшим временем – узла Г. У вершины Г остался всего один не пройдённый сосед – узел Б. От Г до Б путь занимает 10 минут, значит весь путь до Б через вершину Г займет 20 минут, что меньше ∞, поэтому обновляем данные в таблице.

Таблица 1.2.3 – Результат третьего шага алгоритма Дейкстры

Узел	Время до узла
В	8
Д	30
И	11
Г	10
Б	20

Теперь путь до конечного узла Б составляет 20 минут. От узла И до Б путь займет также 10 минут, но в сумме на маршрут через И уйдет 21 минута, что больше известных 20 через узел Г, поэтому эти данные не записываются в таблицу. Аналогично с вершиной Д. Теперь, когда все узлы пройдены, получается итоговая таблица, по которой вычисляется итоговый путь.

Таблица 1.2.3 – Итоговый результат

Узел	Время до узла
В	8
Д	30
И	11
Г	10
Д	30
Б	20

В результате получается, что самый быстрый путь будет выглядеть так: А-В-Г-Б, как показано на рисунке 1.2.5.

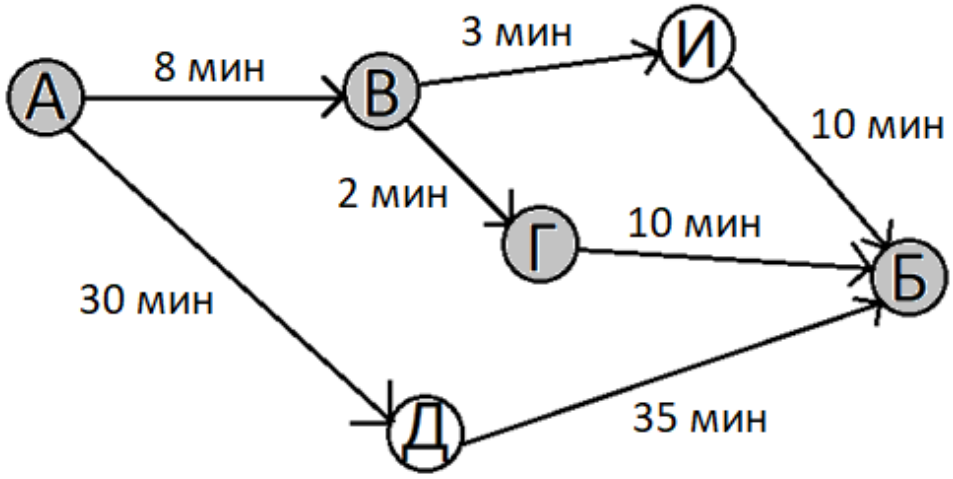

Рисунок 1.2.5 – Граф с отмеченным итоговым результатом

Стоит упомянуть, что алгоритм Дейкстры не работает в графах с отрицательными весами, в таком случае используется другой алгоритм – Беллмана-Форда, однако здесь он не будет рассмотрен, поскольку при построении маршрута отрицательных значений ребер быть не может [11].

В большинстве своем алгоритм Дейкстры хорошо применим во многих задачах, решаемых с помощью графов, однако он сильно теряет свою эффективность в решении NP-трудных задач, таких, как задача о коммивояжере. Время ее решения растет с большой скоростью, а на просчет всех возможных вариантов с учетом современных компьютерных мощностей может уйти слишком много времени. В общем случае для вычисления результата при n элементах необходимо произвести n-факториал (n!) операций, то есть время выполнения составит O(n!). Для такой задачи на данный момент не существует алгоритма, который бы смог дать очень точный результат за приемлемое время. По этой причине самые эффективные алгоритмы решения задачи коммивояжера находят приближенное решение, а не самое точное [6].

Рассмотрим один из таких методов – генетический алгоритм, который является одним из видов эволюционных алгоритмов [5]. Идея, заложенная в основу этого алгоритма, заключается в имитации эволюционного процесса, т. е. применяется принцип выживания наиболее приспособленных особей. В случае с вычислением наиболее подходящего маршрута приспособленность особи будет определяться эффективностью полученного результата, насколько удовлетворительным получилось приблизительное решение [3][8]. Рассмотрим основные понятия:

- Хромосома – это последовательность вершин, которые образуют путь;
- Особь – это набор хромосом, которые удовлетворяют решению, т. е. просчитанный маршрут, удовлетворяющий определенным критериям;
- Популяция – это множество особей, т. е. маршрутов;
- Скрещивание – это операция, при которой две хромосомы обмениваются своими частями;
- Мутация – это случайное изменение одной или нескольких позиций в хромосоме. Они нужны для того, чтобы дать достаточное разнообразие с точки зрения результатов, чего может не хватать при использовании только скрещивания.

Генетический алгоритм обычно начинается с генерации начальной популяции – вариантов маршрута до нужного места. Затем для каждой особи в своей популяции вычисляется ее приспособленность, т. е. то, насколько она удовлетворяет выбранным условиям, как, например, длина маршрута. После чего случайным образом выбираются два родителя, которые будут участвовать в процессе размножения. После выполнения алгоритма скрещивания родители заменяются двумя новыми потомками, полученными в результате прошлой операции. После скрещивания выполняется мутация, которая с некоторым процентом вероятности меняет случайно выбранные гены местами. Для нахождения оптимального маршрута жизненный цикл популяции длится пока не сменит друг друга заданное число поколений, после чего выбирается вариант, который больше всего удовлетворяет выбранным условиям. Большими плюсами генетического алгоритма являются то, что он хорошо работает при решении крупномасштабных проблем оптимизации и обрабатывает одновременно несколько точек поискового пространства.

Генетический алгоритм используется планировщиком запросов Postgres [20]; компания Biotech использует алгоритм для оптимизации функций подсчета очков, которые используются для виртуального скрининга в рамках своего приложения для поиска лекарств; вплоть до версии 3.0 программное обеспечение SpamAssassin, которое широко используется для отделения нежелательной электронной почты от полезной, использовало генетический алгоритм для определения веса баллов для различных эвристик спама, которые оно включило [16]. Также он широко используется в логистике (задачах оптимизации цепочек поставок) [18].

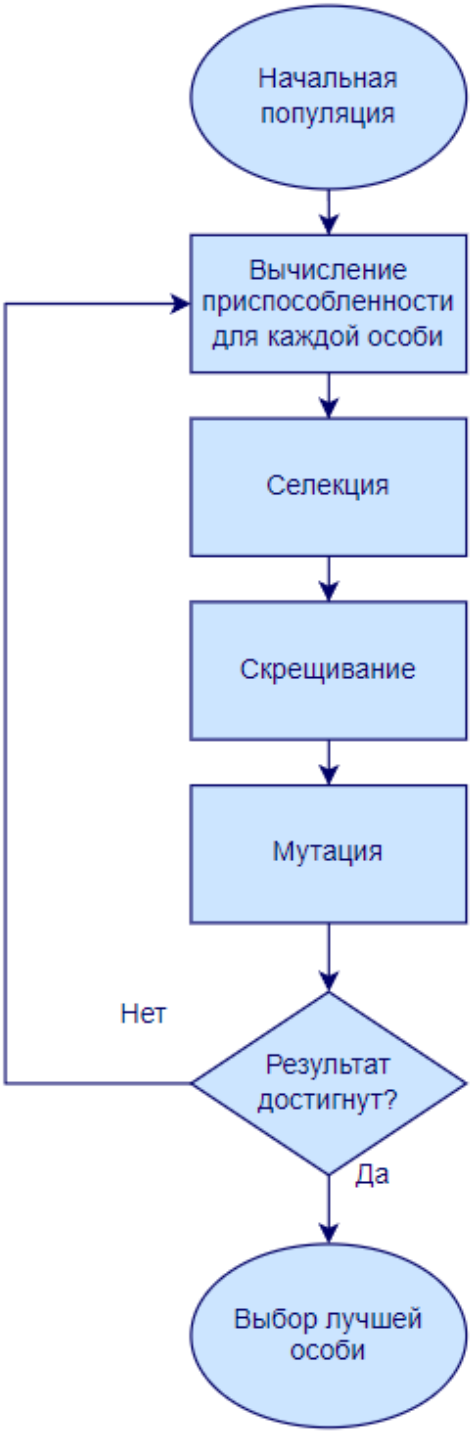

Рисунок 1.2.6 – Схема работы генетического алгоритма

Выводы по разделу 1

Приведем выводы по первому разделу бакалаврской работы:

– алгоритм Дейкстры является ядром в работе таких навигационных приложений, как Google Maps, Apple maps, OpenStreetMap,

Яндекс.Карты и многих других. На практике он используется с модификациями;

– рассмотренный алгоритм поиска в ширину не подходит для решения поставленной задачи, так как он работает с невзвешенными графами;

– среди эволюционных алгоритмов был рассмотрен генетический, который широко используется в логистике и в решении NP-трудных задач, таких, как задача о коммивояжере, время решения которой растет с большой скоростью.

2 ПРОЕКТИРОВАНИЕ АЛГОРИТМА ДЛЯ ПОИСКА КРАТЧАЙШЕГО ПУТИ

2.1 Анализ и выбор алгоритма

В предыдущем разделе были рассмотрены некоторые плюсы и минусы алгоритмов, а именно:

Плюсы алгоритма Дейкстры:

- легок для понимания;
- высокая точность результата.

Минусы алгоритма Дейкстры:

- полный перебор всех вершин графа;
- работает только с метриками положительных значений.

А также была рассмотрена работа одного из эволюционных алгоритмов, которые разделены на несколько видов:

- генетические алгоритмы;
- эволюционное программирование;
- программирование экспрессии генов;
- дифференциальная эволюция;
- нейроэволюция.

Среди них мною был выбран генетический алгоритм, поскольку у меня уже был небольшой опыт работы с ним. В целом, они работают по одной схеме – имитируют эволюцию, но со своими тонкостями и нюансами.

Плюсы генетического алгоритма:

- хорошо работает при решении крупномасштабных проблем оптимизации, где ответ неочевиден;
- обрабатывает одновременно несколько точек поискового пространства;
- можно использовать с многопоточностью.

Минусы генетического алгоритма:

- не всегда можно представить задачу в виде хромосом и генов;

– не гарантирует получение оптимального решения.

Хорошо известно, что алгоритм Дейкстры с модификациями активно используется в приложениях для построения кратчайших маршрутов, в то время как генетический алгоритм используют в логистике, что так же относится к задачам маршрутизации. Однако я не смогла найти явного сравнения алгоритма Дейкстры и одного из эволюционных при решении одинаковой задачи, как, например, задачи поиска кратчайшего пути. Поэтому мною был выбран уже знакомый генетический алгоритм из эволюционных, чтобы в дальнейшем провести тесты и выяснить насколько хорошо он справляется с задачей поиска кратчайшего пути в сравнении с алгоритмом Дейкстры.

2.2 Общая структура алгоритма

Генетический алгоритм разработан согласно схеме рисунка 1.2.6. Код программы написан на языке Python, поскольку данный язык предоставляет огромное количество модулей и пакетов, что позволит при реализации генетического алгоритма опустить часть кода, который уже реализован в библиотеках и хорошо оптимизирован профессиональными программистами. К тому же язык Python универсальный – он прекрасно подходит для решения многих задач, включая машинное обучение. В качестве среды разработки была выбрана бесплатная среда Colaboratory с которой у меня уже есть небольшой опыт работы [25]. Она позволяет выполнять код Python в браузере, который исполняется на облачных серверах Google, независимо от мощности вашего устройства. Colaboratory активно используется в области машинного обучения, включая разработки и обучения нейронных сетей, распространения исследований в области искусственного интеллекта и т. п.

Входные данные будут представлены в виде таблиц далее. В качестве таких данных будут взяты следующие: расстояние между точками в километрах, цена за проезд и средняя скорость проезда на данном участке. С помощью последнего параметра и известного расстояния, можно будет посчитать необходимое на проезд время на случай, если пользователь

захочет поехать самой быстрой по времени дороге. Обосновано это тем, что может быть такая ситуация, когда на меньшем участке скорость перемещения настолько мала по каким-нибудь причинам (например, пробка или строительные работы), чем на участке с большим расстоянием, что проезд на нем займет больше времени, несмотря на меньшее расстояние. Такие приложения как Яндекс.Карты используют для оценки ситуаций на дорогах данные от пользователей, а также знают среднюю скорость преодоления участка в нормальное состояние дороги, чтобы далее можно было оценить то, насколько дорога загружена по сравнению с обычным состоянием на ней.

Для начала представим данные в виде графа на рисунке 2.2.1.

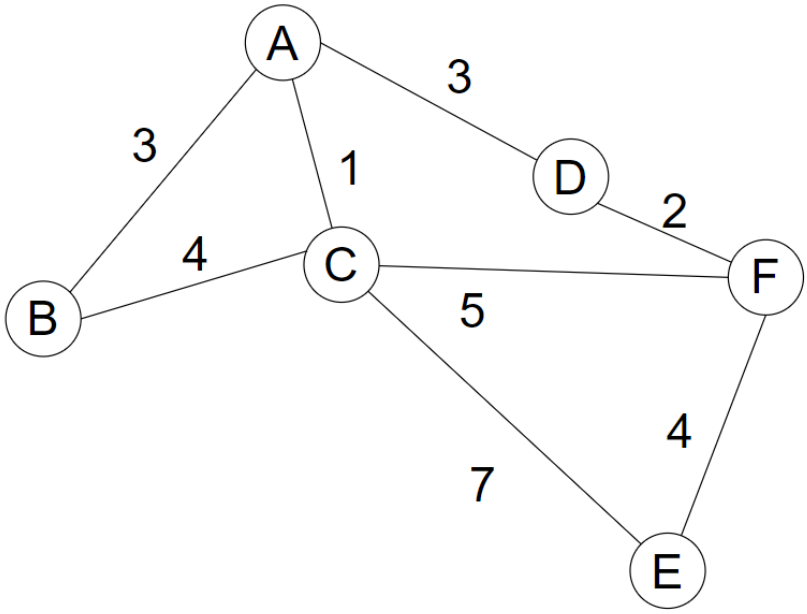

Рисунок 2.2.1 – Граф входных данных с отмеченными расстояниями между вершинами

Входные данные представляют собой некоторые города R, O, M, A, Q, L, B, N, расстояние, скорость перемещения и стоимость проезда между которыми известно. Бесконечность означает, что между данными городами нет известного маршрута. График симметричен, то есть, если существует путь от вершины A до B со значением J, то существует и путь от вершины B

до A со значением J. Представим данные рисунка 2.2.1 в виде таблицы 2.2.1 ниже.

Таблица 2.2.1 – Расстояние между вершинами

	A	B	C	D	E	F
A	0	3	1	3	∞	∞
B	3	0	4	∞	∞	∞
C	1	4	0	∞	7	5
D	3	∞	∞	0	∞	2
E	∞	∞	7	∞	0	4
F	∞	∞	5	2	4	0

Аналогично с данными о цене на проезд. Стоит уточнить, что если пользователь выбирает в качестве желательного вида транспорта общественный, то цена за проезд на всех участках будет фиксированной, поэтому возьмем в качестве цены за проезд на общественном транспорте равным 8. Если же транспорт не будет общественным, то цена за проезд будет разной для каждого участка пути. Цена за проезд на таких транспортах указана на рисунке 2.2.2 и таблице 2.2.2 ниже.

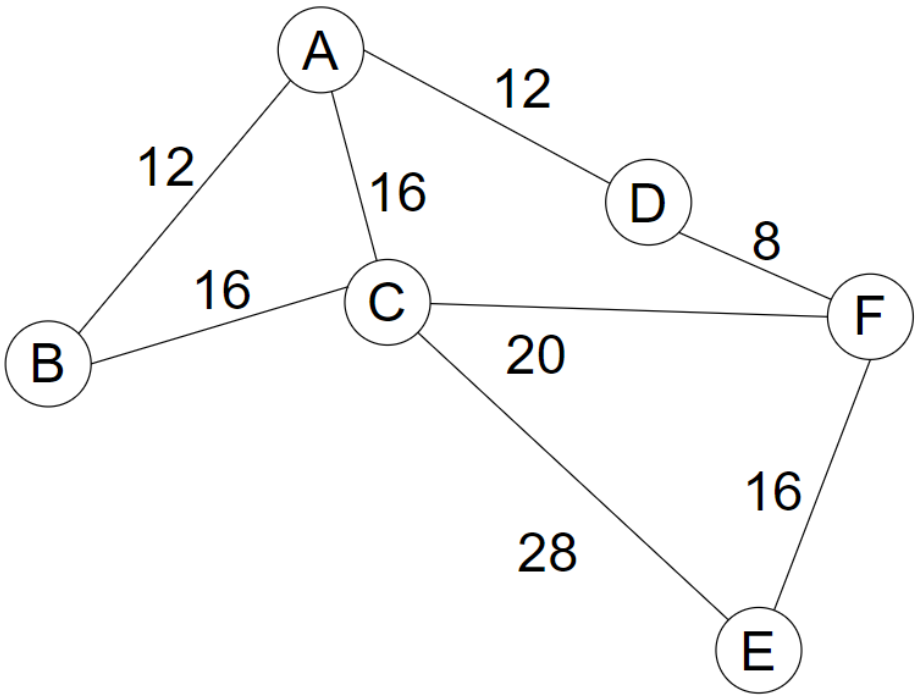

Рисунок 2.2.2 – Граф входных данных с отмеченными ценами за проезд между узлами (пересадки)

Таблица 2.2.2 – Цена за проезд между вершинами

	A	B	C	D	E	F
A	0	12	16	12	∞	∞
B	12	0	16	∞	∞	∞
C	16	16	0	∞	28	20
D	12	∞	∞	0	∞	8
E	∞	∞	28	∞	0	16
F	∞	∞	20	8	16	0

Аналогично для средней скорости на участках указано на рисунке 2.2.3 и в таблице 2.2.3.

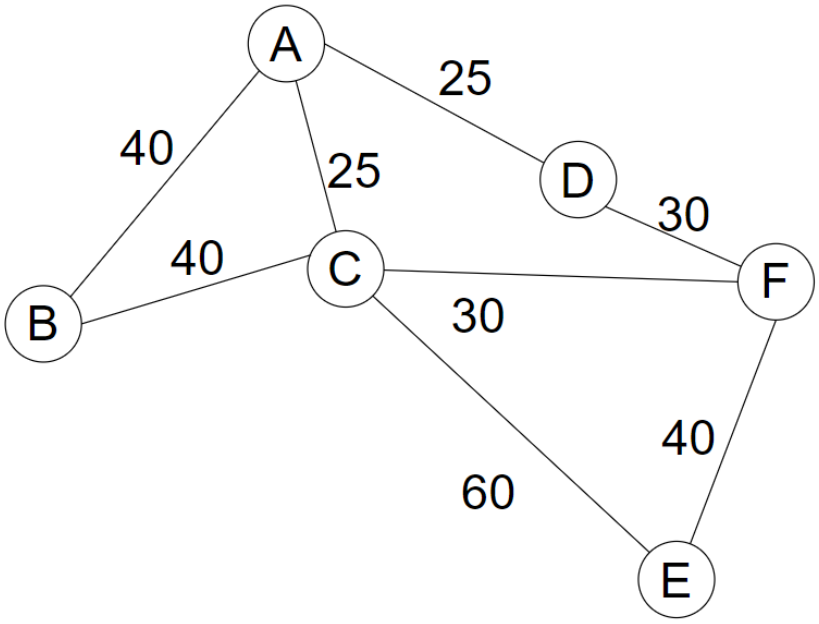

Рисунок 2.2.3 – Граф входных данных с отмеченными на нем средними скоростями перемещения между узлами

Таблица 2.2.3 – Цена за проезд между вершинами

	A	B	C	D	E	F
A	0	40	25	25	∞	∞
B	40	0	40	∞	∞	∞
C	25	40	0	∞	60	30

D	25	∞	∞	0	∞	30
E	∞	∞	60	∞	0	40
F	∞	∞	30	30	40	0

При написании кода будет использоваться DEAP (Distributed Evolutionary Algorithms in Python) – пакет для создания генетических алгоритмов [19]. С помощью команды !pip install deap устанавливаем ее. Если установка прошла успешно, то будут доступны следующие импорты, изображенные на рисунке 2.2.4.

```
!pip install deap
from deap import base, algorithms, creator, tools
import numpy as np
import random
import matplotlib.pyplot as plt
```

Рисунок 2.2.4 – Библиотеки программы

Рассмотрим их более подробно далее. Библиотека numpy добавляет поддержку больших матриц и многомерных массивов, включая высокоуровневые математические функции для работы с ними. Пакет matplotlib.pyplot понадобится для вывода графика в конце. Библиотека random нужна будет для генерации случайных значений. Модуль creator позволяет с помощью функции create(<название класса>, <базовый класс>, [атрибуты нового класса]) создавать новые классы в программе. Например, если написать creator.create("Foo", object, a=list, b =1, c = 3), то в модуле creator будет создан класс с именем Foo, наследуемый от object, как показано на рисунке 2.2.5.

```
class Foo(object):
    b = 1
    c = 3

    def __init__(self):
        self.a = list()
```

Рисунок 2.2.5 – Созданный в ветке creator класс Foo

Для создания экземпляра такого класса используется следующий код: f = creator.Foo(). Модуль creator понадобится для создания классов генетического алгоритма. С помощью модуля tools можно воспользоваться классом base.Toolbox для регистрации псевдонима функции, а также для указания дополнительных параметров, как значения по умолчанию. Тогда при обращении к функции по псевдониму, если не передать ей новые значения, то она получит те, что были указаны как по умолчанию при создании этого псевдонима. Модуль tool так же включает в себя функцию initRepeat, которая помогает в формировании списков. Остальные модули будут рассмотрены уже непосредственно в коде для лучшего понимания причины их применения.

Чтобы представить данные графа в виде хромосом, можно придумать несколько разных способов, я выбрала следующий: каждая хромосома хранит в себе все возможные пути от начальной вершины ко всем остальным, т. е. список вершин. Лучшей хромосомой будет считаться в итоге та, которая будет хранить в себе лучшие по определенным параметрам маршруты по сравнению с другими хромосомами.

Данные матриц смежности, представленные в таблицах 2.2.1, 2.2.2 и 2.2.3, в коде будут объединены в класс Weight, где поле km – это длина дуги, price – цена за проезд, km_h – средняя скорость передвижения на данном участке. Зная длину маршрута и среднюю скорость, можно рассчитать время, которое будет затрачено на этот маршрут (Время = Расстояние : Скорость). Однако, поскольку в матрице смежности по главной диагонали все значения равны нулю, то, чтобы избежать деления на нуль, в класс был добавлен логический оператор. Теперь, если расстояние (а, значит, и средняя скорость перемещения) равно нулю, то и время, необходимое на проезд, тоже равно нулю.

```
class Weight(object):
    def __init__(self, km, price, km_h):
        self.km = km
        self.price = price
        self.km_h = km_h

        if (km != 0): self.time = km/km_h
        else:         self.time = 0
```

Рисунок 2.2.6 – Класс Weigth

Чтобы в конце увидеть хромосому лучшей особи последней итерации, будет добавлен объект HallOfFame().

```
BEST_CHROMOSOME_SIZE = 1

hof = tools.HallOfFame(BEST_CHROMOSOME_SIZE)
```

Рисунок 2.2.7 – Инициализация «зала славы» для хранения данных о лучшей особи в конце итерации

Переменная BEST_CHROMOSOME_SIZE отвечает за размер зала славы, то есть, в конце будет выбрана только одна лучшая особь. Объект hof будет хранить в себе ссылку на лучшего индивидуума.

Далее идет инициализация матрицы смежности объектами класса Weight, как показано на рисунке 2.2.8, где inf равна 1000, что означает, что маршрут не существует. В таблицах выше данный параметр был равен бесконечности. Здесь он равен всего лишь 1000, т. к. он все равно достаточно велик по сравнению с известными значениями в таблицах. Переменная price_pub хранит цену за проезд на общественном транспорте, поскольку цена за проезд на таких видах транспорта обычно фиксирована.

```
inf = 1000
price_pub = 8

#Матрица смежности
#     A                   B                   C
D = ((Weight(0,0,0),       Weight(3,12,40),    Weight(1,16,25),
     (Weight(3,12,40),     Weight(0,0,0),      Weight(4,16,40),
     (Weight(1,16,25),     Weight(4,16,40),    Weight(0,0,0),
     (Weight(3,12,25),     Weight(inf,inf,inf), Weight(inf,inf,inf),
     (Weight(inf,inf,inf), Weight(inf,inf,inf), Weight(7,28,60),
     (Weight(inf,inf,inf), Weight(inf,inf,inf), Weight(5,20,30),
```

Рисунок 2.2.8 – Инициализация матрицы смежности. Часть 1

```
D                     E                     F
Weight(3,12,25),      Weight(inf,inf,inf), Weight(inf,inf,inf)),
Weight(inf,inf,inf), Weight(inf,inf,inf), Weight(inf,inf,inf)),
Weight(inf,inf,inf), Weight(7,28,60),      Weight(5,20,30)),
Weight(0,0,0),        Weight(inf,inf,inf), Weight(2,8,30)),
Weight(inf,inf,inf), Weight(0,0,0),        Weight(4,16,40)),
Weight(2,8,30),       Weight(4,16,40),     Weight(0,0,0)))
```

Рисунок 2.2.9 – Инициализация матрицы смежности. Часть 2

Далее будут определены необходимые для генетического алгоритма константы.

```
startV = 0
LENGTH_D = len(D)
LENGTH_CHROM = len(D)*len(D[0])

POPULATION_SIZE = 500
P_CROSSOVER = 0.9
P_MUTATION  = 0.1
MAX_GENERATONS = 30
```

Рисунок 2.2.10 – Инициализация параметров для генетического алгоритма

Переменная startV – это стартовая вершина, в нашем случае это будет первая вершина в массиве, то есть узел A. LENGTH_D хранит в себе размер матрицы смежности. LENGTH_CHROM – это длина хромосомы, которая хранит в себе все маршруты от начальной вершины ко всем остальным. За количество особей в популяции отвечает параметр POPULATION_SIZE. За вероятность скрещивания и мутации отвечают переменные P_CROSSOVER и P_MUTATION соответственно. Параметр MAX_GENERATORS отвечает за максимальное количество поколений, то есть итераций. Чем больше это значение, тем лучше будет результат, но тем больше уйдет времени на расчет. Далее переменную RANDOM_SEED инициализируем любым случайным числом для генератора случайных чисел. Чем удачнее будет число, тем лучше будет конечный результат. Методом подбора было выбрано число 42.

Это нужно для того, чтобы результат работы генетического алгоритма можно было повторить.

```
RANDOM_SEED = 42
random.seed(RANDOM_SEED)
```

Рисунок 2.2.11 – Инициализация зерна

Затем создадим два класса, как показано на рисунке 2.2.12. Принцип работы функции create был рассмотрен выше. Класс FitnessMin хранит в себе информацию о приспособленности индивидуумов, а класс Individual представляет собой отдельную особь популяции.

```
creator.create("FitnessMin", base.Fitness, weights = (-1.0,))
creator.create("Individual", list, fitness = creator.FitnessMin)
```

Рисунок 2.2.12 – Создание классов FitnessMin и Individual

На рисунке 2.2.12 видно, что при создании класса FitnessMin параметр weights равен отрицательному значению. Это обосновано тем, что в данной задаче цель – минимизация функции приспособленности, и чтобы сохранить общую логику генетического алгоритма, указывается отрицательный вес, чтобы наименьшее значение становилось наибольшим. Процесс эволюции основан на том, что выживают наиболее приспособленные особи, то есть те, у которых функция приспособленности имеет наибольшее значение, а в моей задаче нужно находить такие пути, при которых значения будут наименьшими, т. е. меньше расстояние, времени на путь и стоимость проезда.

Теперь, когда определены необходимые классы, можно перейти к созданию начальной популяции. Для этого с помощью класса Toolbox зарегистрируем следующие функции:

```
toolbox = base.Toolbox()
toolbox.register("randomOrder", random.sample, range(LENGTH_D), LENGTH_D)
toolbox.register("individualCreator", tools.initRepeat, creator.Individual,
                 toolbox.randomOrder, LENGTH_D)
toolbox.register("populationCreator", tools.initRepeat, list, toolbox.individualCreator)
```

Рисунок 2.2.13 – Регистрация функций для создания начальной популяции

Функция randomOrder отвечает за генерацию случайных списков со значениями от нуля до LENGTH_D-1 длиной LENGTH_D. Функция individualCreator на основе класса creator.Individual создает отдельных особей, а toolsbox.randomOrder возвращает объекты для инициализации его значений. LENGTH_D – число таких объектов. Все это создает хромосому. Функция populationCreator формирует список особей. Для создания популяции используется следующая функция, где в качестве параметра указывается ее количество особей:

```
population = toolbox.populationCreator(n=POPULATION_SIZE)
```

Рисунок 2.2.14 – Создание начальной популяции

Далее нужно описать функции выполнения скрещивания, мутации и определения приспособленности. Начнем с последней.

```
def dikstryFitness(individual):
    s = 0
    for n, path in enumerate(individual):
        path = path[:path.index(n) + 1]

        si = startV
        for j in path:
            s += D[si][j].km            #рассматриваемый параметр
            si = j

    return s,
```

Рисунок 2.2.15 – Описание функции расчета приспособленности особи

Данная функция высчитывает сумму весов дуг от начальной вершины ко всех остальным. В качестве веса дуги в данном случае выбрано расстояние в километрах, и если пользователь захочет по другому параметру искать маршрут, то менять его нужно в этом месте. Переменная s хранит в себе сумму маршрутов до всех узлов графа. Затем перебираем списки в каждой хромосоме, после чего выделаем узлы, которые относятся к текущему маршруту. Во вложенном цикле уже вычисляется сумма весов с известными значениями матрицы смежности. Функция возвращает кортеж значений.

Далее идет определение функции скрещивания.

```
def cxOrder(ind1, ind2):
    for p1, p2 in zip(ind1, ind2):
        tools.cxOrdered(p1, p2)
    return ind1, ind2
```

Рисунок 2.2.16 – Определение функции скрещивания

На рисунке 2.2.16 видно, что в качестве входных параметров функция принимает двух особей, которых потом будет скрещивать. В ней в цикле перебираются пары маршрутов из хромосом первого и второго индивидуума, затем вызывается встроенная в библиотеку DEAP функция cxOrdered. Данная функция реализует алгоритм упорядоченного скрещивания, то есть в итоге все номера вершин графа в хромосомах будут сохранены.

На рисунке 2.2.17 показано описание функции мутации.

```
def mutShuffleIndexes(individual, indpb):
    for ind in individual:
        tools.mutShuffleIndexes(ind, indpb)

    return individual,
```

Рисунок 2.2.17 – Определение функции мутации

Данная функция определяет порядок мутации генов в хромосомах, то есть идет перебор списков в хромосоме и с некоторой вероятностью indpb происходит мутация – перемешивание индексов (номеров вершин графа). Функция возвращает результат мутации.

Далее необходимо описанные выше функции для генетического алгоритма зарегистрировать, как показано на рисунке 2.2.18.

```
toolbox.register("evaluate", dikstryFitness)  # эволюция
toolbox.register("select", tools.selTournament, tournsize=3)  # отбор
toolbox.register("mate", cxOrder)  # скрещивание
toolbox.register("mutate", mutShuffleIndexes, indpb=1.0 / LENGTH_CHROM / 10)  # мутация
```

Рисунок 2.2.18 – Регистрация функций

Названия, указанные в кавычках, должны быть именно такими, т. к. этого требует пакет DEAP. Функция evaluate вычисляет приспособленность

отдельного индивидуума; select – выбирает родителей для следующего поколения, то есть делает отбор. Встроенная в библиотеку DEAP функция selTournament проводит турнирный отбор для трех случайно выбранных индивидуумов. Функция mate выполняет скрещивание двух особей, а функция mutate – мутацию.

Далее, для сбора статистики нужно определить объект класса Statistics, как показано на рисунке 2.2.19.

```
stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("min", np.min)  # минимальное значение приспособленности особи
stats.register("avg", np.mean) # среднее значение присособленности всех особей в популяциях
```

Рисунок 2.2.19 – Определение объекта stats для сбора статистики

С его помощью будут вычисляться минимальное и среднее значения приспособленности каждого поколения.

Чтобы запустить генетический алгоритм, нужно воспользоваться встроенной в библиотеку DEAP функцией eaSimple:

```
population, logbook = algorithms.eaSimple(population, toolbox,
                                          cxpb=P_CROSSOVER / LENGTH_D,
                                          mutpb=P_MUTATION / LENGTH_D,
                                          ngen=MAX_GENERATONS, halloffame=hof,
                                          stats=stats, verbose=True)
```

Рисунок 2.2.20 – Запуск генетического алгоритма

Для вывода статистики в виде графа, нужно добавить следующий код:

```
# собираем статистику
maxFitnessValues, meanFitnessValues = logbook.select("min", "avg")

# выводим результат
plt.plot(maxFitnessValues, color='red')
plt.plot(meanFitnessValues, color='green')
plt.xlabel('Поколение')
plt.ylabel('Макс/средняя приспособленность')
plt.title('Зависимость максимальной и средней приспособленности поколений')
plt.show()
```

Рисунок 2.2.21 – Сбор и вывод статистики

В итоге получается вывод, изображенный на рисунках 2.2.22 и 2.2.23. Целиком код можно посмотреть в приложении [Приложение А].

gen	nevals	min	avg
0	500	1017	9042.05
1	96	2037	6848.57
2	73	2025	5386.98
3	84	1026	4394.77
4	73	1026	3485.25
5	76	1026	3010.77
6	91	1020	2681.21
7	70	1020	2161.82
8	80	1015	2185.44
9	83	29	1908.84
10	70	26	1482.41
11	94	20	1123.54
12	58	20	979.99
13	66	20	787.31
14	98	20	605.02
15	78	20	269.564
16	76	20	173.866
17	75	20	134.988
18	65	20	112.23
19	51	20	96.152
20	93	20	100.27
21	77	20	82.072
22	95	20	110.194
23	90	20	86.092
24	67	20	106.112
25	88	20	116.192
26	91	20	86.1
27	97	20	110.182
28	101	20	108.178
29	78	20	68.192
30	87	20	150.272

Рисунок 2.2.22 – Вывод статистики

На рисунке 2.2.22 видно, что первый столбец (gen) – это номер поколений; nevals – количество особей в популяции; min – найденное в данном поколении минимальное значение, а avg – среднее значение.

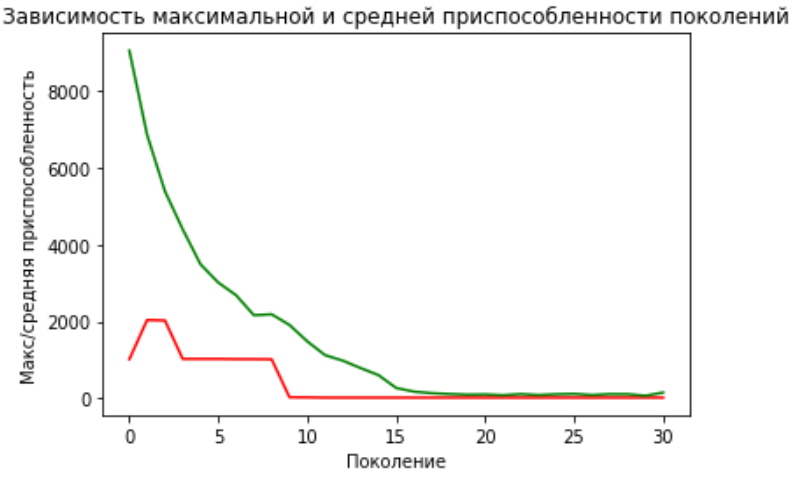

Рисунок 2.2.23 – График зависимостей средней и максимальной приспособленностей от поколений

На рисунке 2.2.23 видно, что в итоге было достигнуто некоторое минимальное значение, равное 20 – суммарный минимальный маршрут. На

рисунке 2.2.22 и на графике выше, смотря на красную линию, видно, что это значение было достигнуто на 9 итерации, то есть в этом случае было достаточно 9 поколений, чтобы достичь желаемого результата. Чтобы вывести списки лучшей итоговой хромосомы, нужно добавить следующий код:

```
best = hof.items[0]
print("Хромосома лучшей особи, которая хранит следующие списки кратчайших маршрутов:")
print(best)
```

Рисунок 2.2.24 – Код для вывода списков лучших маршрутов

Результат работы этой части кода показан на рисунке 2.2.25 ниже. Стоит упомянуть, что счет здесь начинается с нуля. Первый список – это маршруты из 0 вершины в 0, второй – из нулевой в первую вершину и т. д. Так как длина пути изначально неизвестна, то он считается законченным при встрече узла с номером назначения. Это хорошо видно в первом списке, где первое же значение равно нулю, то есть это и есть кратчайший путь из вершины А в А. Из А в В кратчайшим будет прямой путь, т. к. во втором списке первое значение равно единице, то есть равно индексу узла В.

```
Хромосома лучшей особи, которая хранит следующие списки кратчайших маршрутов:
[[0, 1, 4, 3, 5, 2], [1, 4, 3, 0, 5, 2], [2, 0, 5, 1, 4, 3], [3, 2, 4, 5, 1, 0],
 [2, 4, 1, 5, 3, 0], [3, 5, 1, 4, 0, 2]]
```

Рисунок 2.2.25 – Вывод списков маршрутов лучшей особи

Проверить верность решения можно по рисунку 2.2.26, где изображен начальный граф с отмеченным маршрутом от А до Е, который алгоритм посчитал как самый эффективный. Серыми цифрами обозначены индексы вершин начиная с нуля.

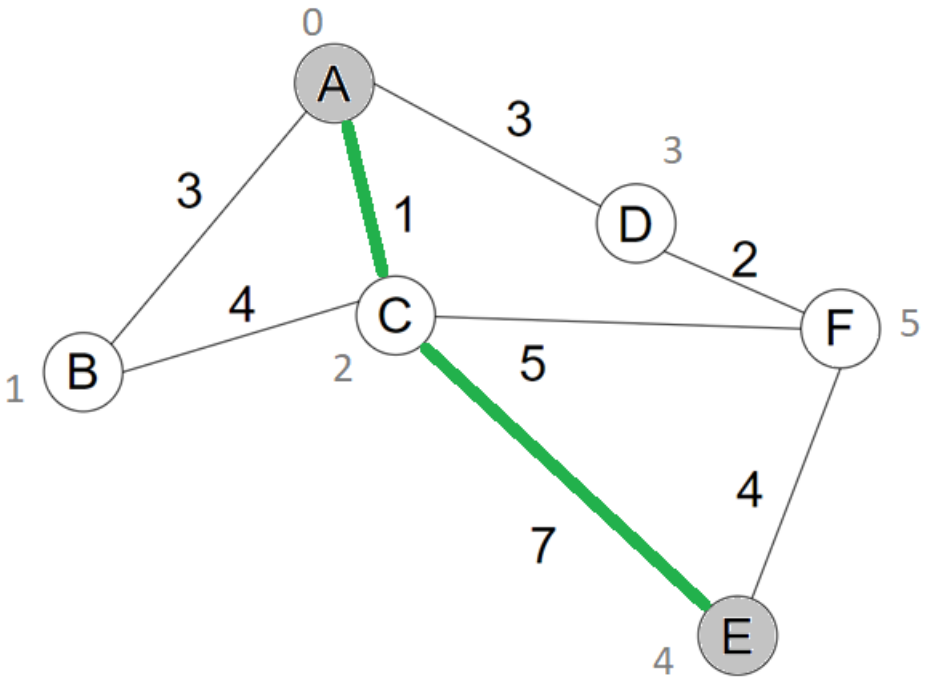

Рисунок 2.2.25 – Кратчайший путь от вершины A до E по результату работы генетического алгоритма

И действительно: если пойти от A к E не через вершину C, а, например, через D, затем F, то нужно будет проехать не 8 километров, 9, что уже больше полученного алгоритмом результата.

2.3 Анализ реализованного алгоритма

Для проверки верности работы алгоритма запустим его для других известных параметров смежной матрицы, например, для стоимости. Для этого нужно в функции dikstryFitness поменять рассматриваемый параметр на price. Тогда результат работы алгоритма будет следующий:

gen	nevals	min	avg
0	500	1116	9173.6
1	96	2172	6966.37
2	73	2100	5508.9
3	84	1128	4502.56
4	73	1128	3573.6
5	76	1128	3126.32
6	91	1104	2730.4
7	70	1104	2274.53
8	80	1072	2263.9
9	83	1072	2002.45
10	70	184	1570.02
11	94	104	1306.56
12	58	104	1168.87
13	66	104	1112.46
14	98	104	1268.18
15	78	104	1066.75
16	76	104	790.216
17	75	104	368.512
18	65	104	186.68
19	51	104	136.624
20	93	104	254.12
21	77	104	142.056
22	95	104	181.032
23	90	104	166.448
24	67	104	193.032
25	88	104	191.272
26	91	104	179.12
27	97	104	171.312
28	101	104	191.4
29	78	104	130.328
30	87	104	195.36

Рисунок 2.3.1 – Результат работы алгоритма с ценовым параметром.

Часть 1

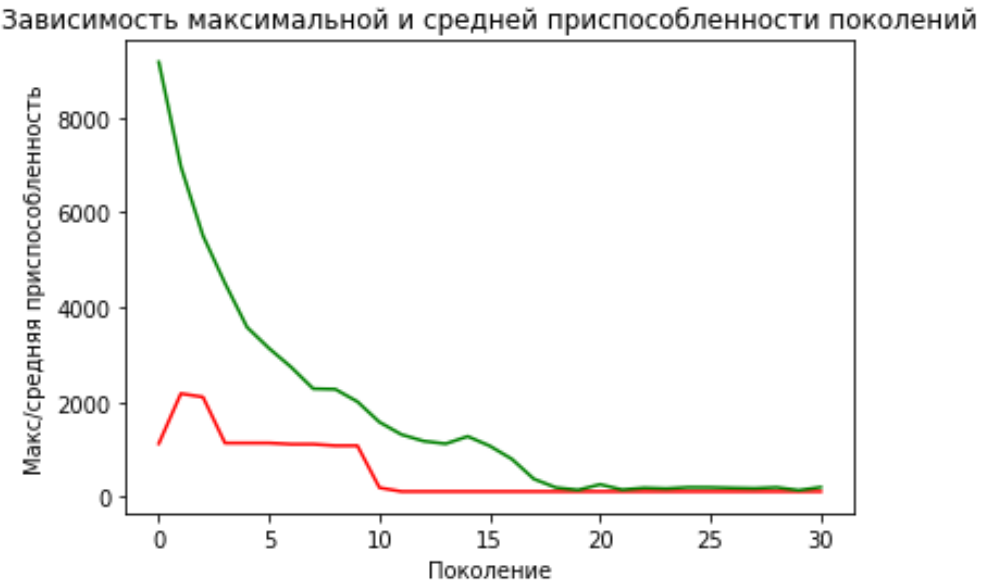

Рисунок 2.3.2 – Результат работы алгоритма с ценовым параметром.

Часть 2

```
Хромосома лучшей особи, которая хранит следующие списки кратчайших маршрутов:
[[0, 1, 3, 4, 2, 5], [1, 4, 3, 0, 5, 2], [2, 0, 5, 1, 4, 3], [3, 2, 4, 5, 1, 0],
 [2, 4, 1, 3, 0, 5], [3, 5, 1, 2, 4, 0]]
```

Рисунок 2.3.2 – Результат работы алгоритма с ценовым параметром.

Часть 3

На основе полученных генетическим алгоритмом результатов отметим решения на графе, где вес ребер – стоимость проезда. Результат представлен на рисунках 2.2.3–2.2.7.

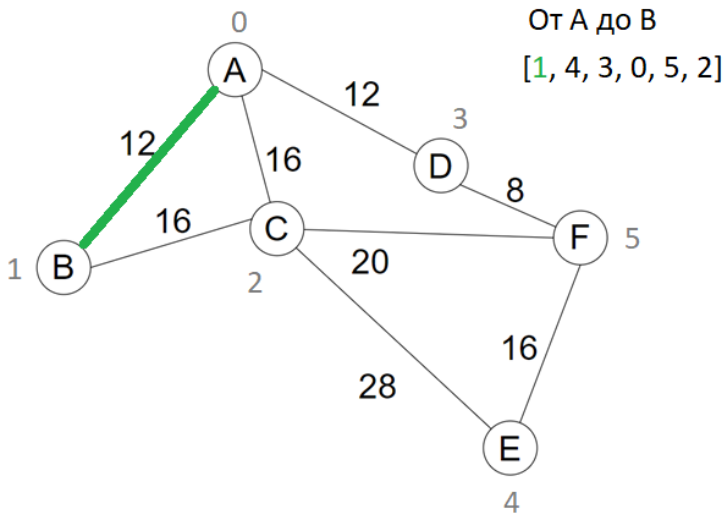

Рисунок 2.3.3 – Лучший по стоимости путь от A до B по расчетам генетического алгоритма

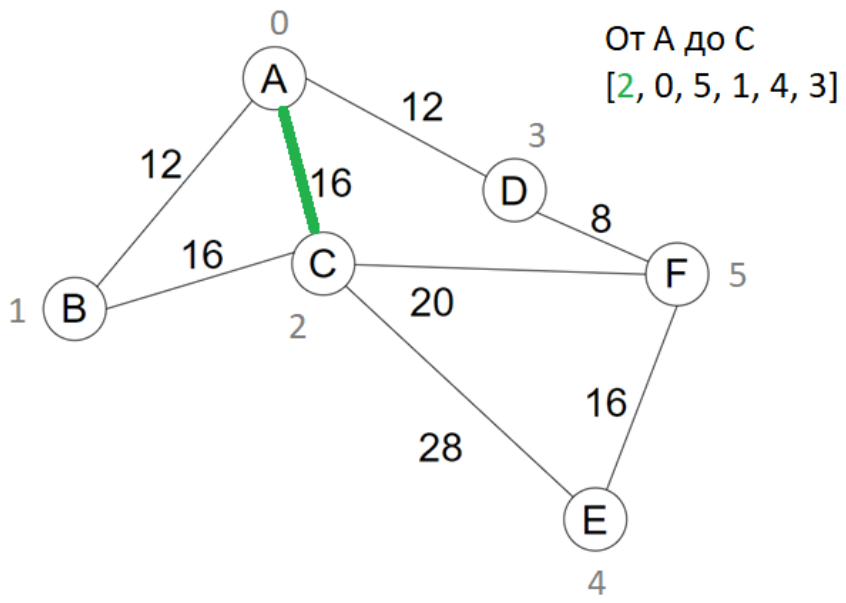

Рисунок 2.3.4 – Лучший по стоимости путь от A до C по расчетам генетического алгоритма

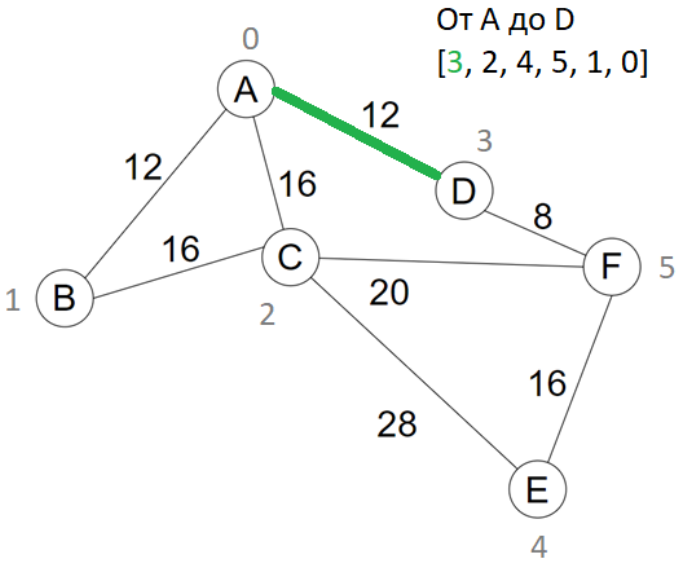

Рисунок 2.3.5 – Лучший по стоимости путь от A до D по расчетам генетического алгоритма

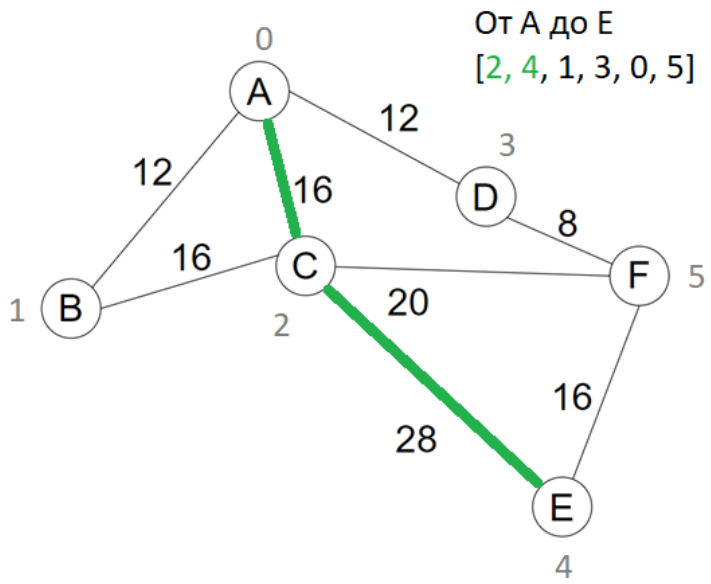

Рисунок 2.3.6 – Лучший по стоимости путь от A до E по расчетам генетического алгоритма

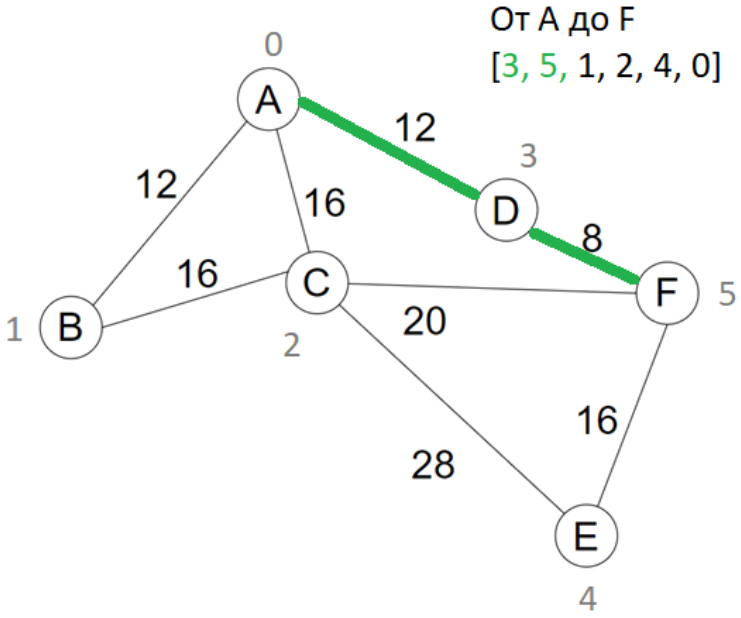

Рисунок 2.3.7 – Лучший по стоимости путь от A до F по расчетам генетического алгоритма

Аналогично сделаем по результатам алгоритма для параметра времени, которое необходимо потратить на маршрут. Данный параметр вычисляется при инициализации данных матрицы смежности по формуле Время = Расстояние : Скорость. Чтобы было удобнее проверить валидность полученных результатов, вручную посчитаем время для каждого участка и

отобразим полученные результаты на графе, изображенном на рисунке 2.3.8. Чтобы время было в минутах, полученный результат надо умножить на 60, для этого нужно добавить это в класс Weight в строку, где считается время.

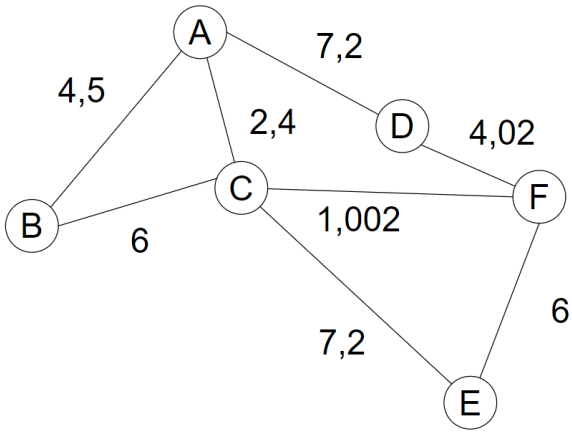

Рисунок 2.3.8 – Граф с отмеченными значениями в минутах

Теперь запустим алгоритм и сверим результаты.

gen	nevals	min	avg
0	500	88.3	601.986
1	96	180.2	468.214
2	73	163.1	377.514
3	84	106.4	315.711
4	73	106.4	257.654
5	76	106.4	228.471
6	91	101.7	206.568
7	70	101.7	176.911
8	80	92.5	171.851
9	83	55.9	153.37
10	70	43.7	133.223
11	94	34.7	112.302
12	58	34.7	100.956
13	66	34.7	93.3812
14	98	34.7	80.3566
15	78	34.7	62.678
16	76	34.7	50.062
17	75	34.7	41.7492
18	65	34.7	42.0074
19	51	34.7	38.2736
20	93	34.7	43.0576
21	77	34.7	37.6184
22	95	34.7	40.946
23	90	34.7	37.505
24	67	34.7	40.2088
25	88	34.7	38.9248
26	91	34.7	42.1
27	97	34.7	44.9962
28	101	34.7	41.1652
29	78	34.7	36.4396
30	87	34.7	43.8652

Рисунок 2.3.9 – Результат работы алгоритма с временным параметром. Часть 1

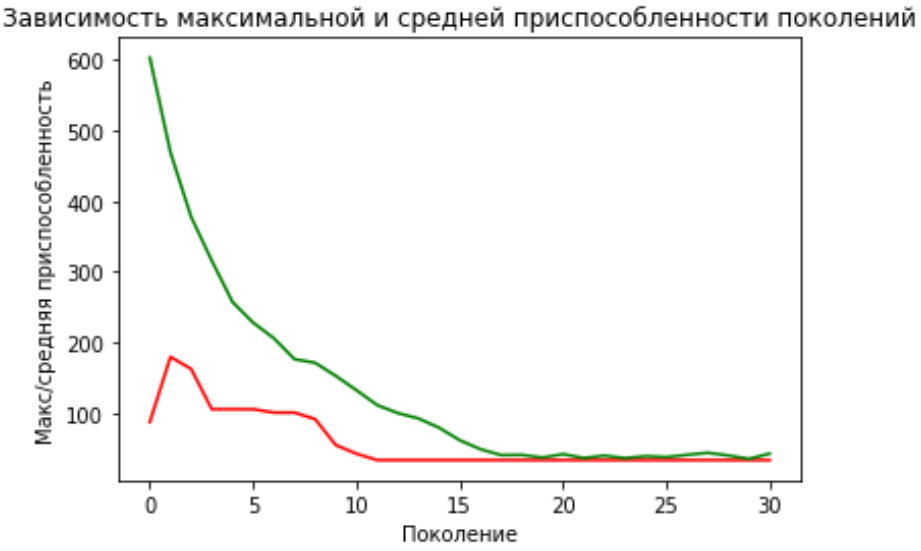

Рисунок 2.3.10 – Результат работы алгоритма с временным параметром. Часть 2

```
Хромосома лучшей особи, которая хранит следующие списки кратчайших маршрутов:
[[0, 1, 4, 3, 5, 2], [1, 4, 3, 0, 5, 2], [2, 0, 5, 1, 4, 3], [3, 2, 4, 5, 1, 0],
 [2, 4, 1, 5, 3, 0], [3, 5, 1, 4, 0, 2]]
```

Рисунок 2.3.11 – Результат работы алгоритма с временным параметром. Часть 3

Далее отметим полученные алгоритмом результаты на графе.

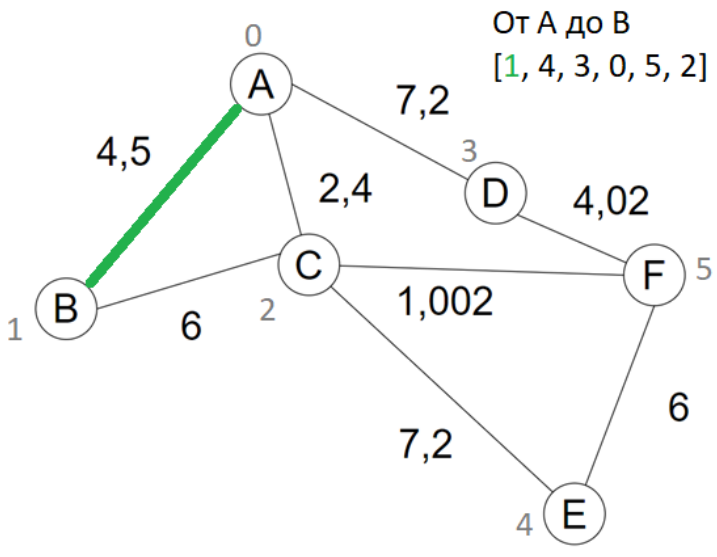

Рисунок 2.3.12 – Лучший по времени путь от A до B по расчетам генетического алгоритма

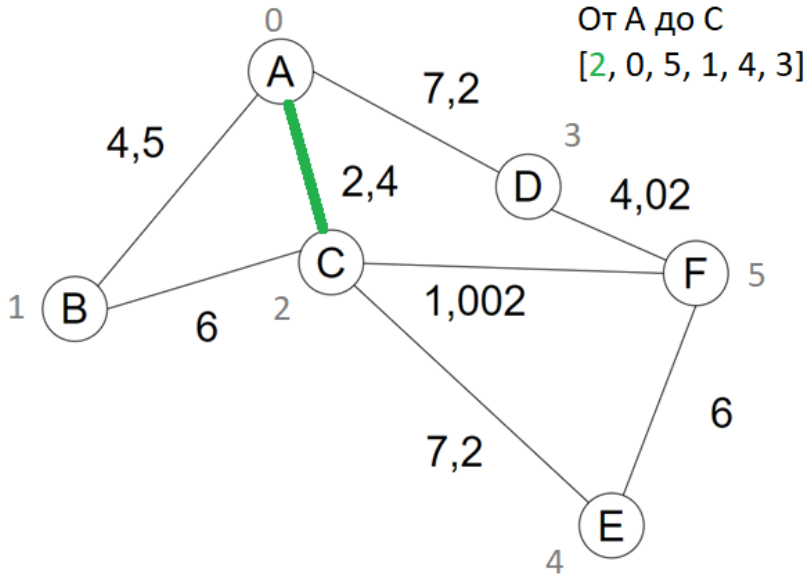

Рисунок 2.3.13 – Лучший по времени путь от A до C по расчетам генетического алгоритма

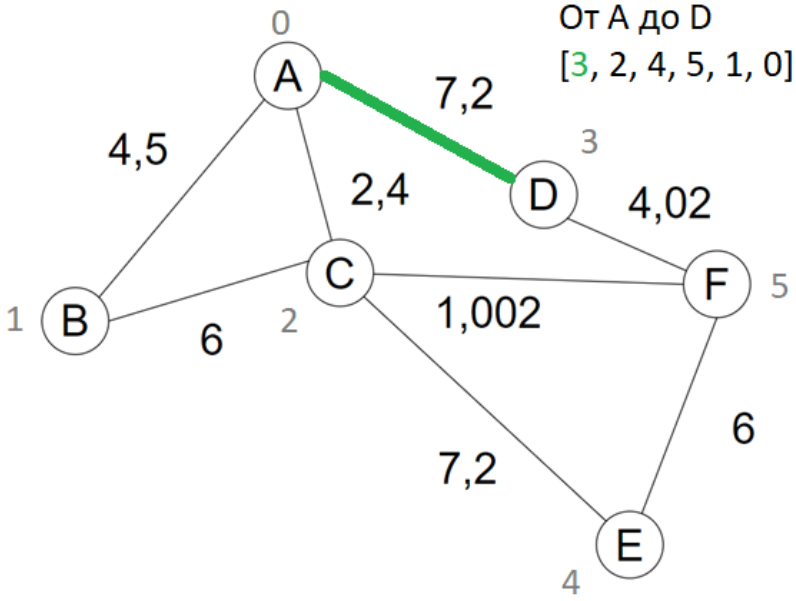

Рисунок 2.3.14 – Лучший по времени путь от A до D по расчетам генетического алгоритма

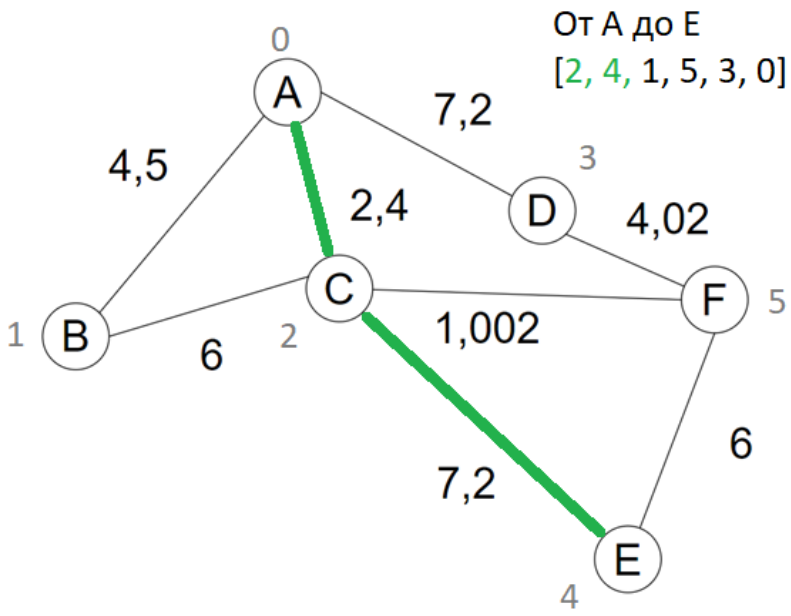

Рисунок 2.3.15 – Лучший по времени путь от A до E по расчетам генетического алгоритма

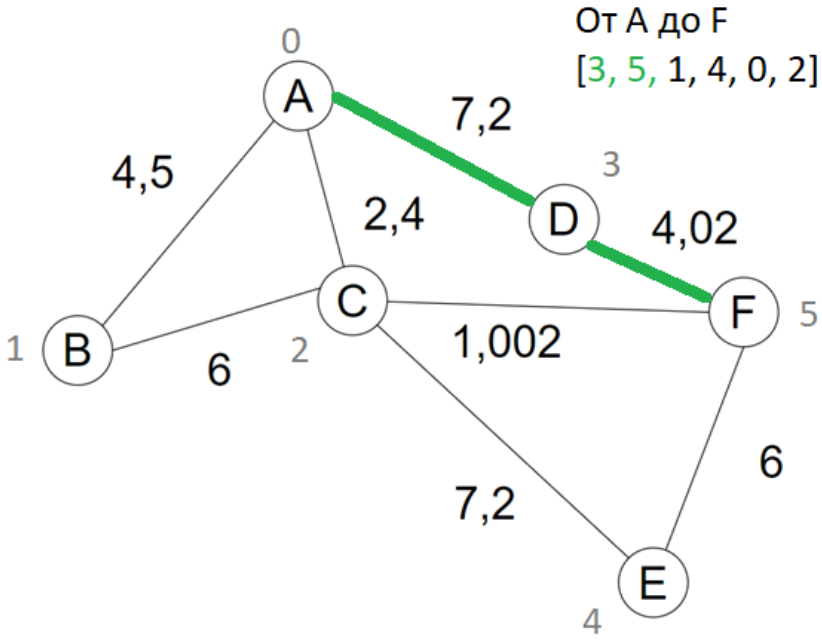

Рисунок 2.3.16 – Лучший по времени путь от A до F по расчетам генетического алгоритма

Теперь, когда мы убедились в валидности полученных результатов, можно перейти к сравнению генетического алгоритма с уже используемым Яндекс.Карты алгоритмом Дейкстры. Для этого в код будут добавлены две библиотеки: datetime и time, чтобы можно было засекать время работы

алгоритмов. Для этого перед запуском генетического алгоритма надо добавить строку start_time = datetime.now(), чтобы запомнить время начало работы. А после окончания работы алгоритма добавить строку res_time = datetime.now() – start_time, т. е. время выполнения считает как текущее время минус время начало работы. В итоге в коде это будет выглядеть следующим образом:

```
start_time = datetime.now() #засекаем время, запоминая время начала работы

# запуск генетического алгоритма
population, logbook = algorithms.eaSimple(population, toolbox,
                                          cxpb=P_CROSSOVER / LENGTH_D,
                                          mutpb=P_MUTATION / LENGTH_D,
                                          ngen=MAX_GENERATONS, halloffame=hof,
                                          stats=stats, verbose=True)

res_time = datetime.now() - start_time #время выполнения = текущее время - время начала работы
```

Рисунок 2.3.17 – Добавление таймера в код

В конце кода для вывода результата надо добавить print("Время работы алгоритма: {}.".format(res_time)). В результате расчета кратчайшего пути алгоритм выведет:

```
Хромосома лучшей особи, которая хранит следующие списки кратчайших маршрутов:
[[0, 1, 4, 3, 5, 2], [1, 4, 3, 0, 5, 2], [2, 0, 5, 1, 4, 3], [3, 2, 4, 5, 1, 0],
 [2, 4, 1, 5, 3, 0], [3, 5, 1, 4, 0, 2]]
Время работы алгоритма: 0:00:01.729274.
```

Рисунок 2.3.18 – Вывод затраченного на расчет генетическим алгоритмом времени

Взяв одну из реализаций алгоритма Дейкстры [10], передаем ему только матрицу смежности с известными параметрами расстояния в километрах и опускаем расчет по времени и цене, поскольку если алгоритм хорошо справляется с одним типом параметров, то, скорее всего, он так же хорошо посчитает и другие.

Входные данные для алгоритма Дейкстры изображены на рисунке 2.3.19.

```
inf = 1000

graph_list = ((0, 3, 1, 3, inf, inf),
              (3, 0, 4, inf, inf, inf),
              (1, 4, 0, inf, 7, 5),
              (3, inf, inf, 0,inf, 2),
              (inf,inf, 7, inf, 0, 4),
              (inf, inf, 5, 2, 4, 0))
```

Рисунок 2.3.18 – Входные данные для алгоритма Дейкстры на основе таблицы 2.2.1

Для запуска алгоритма функции dijkstra передаются два параметра: первый – это матрица смежности; второй – начальная вершина. На рисунке 2.3.19 представлен код алгоритма.

```
def dijkstra(graph, src):
    visited = []
    distance = {src: 0}
    node = list(range(len(graph[0])))
    if src in node:
        node.remove(src)
        visited.append(src)
    else:
        return None
    for i in node:
        distance[i] = graph[src][i]
    prefer = src
    while node:
        _distance = float('inf')
        for i in visited:
            for j in node:
                if graph[i][j] > 0:
                    if _distance > distance[i] + graph[i][j]:
                        _distance = distance[j] = distance[i] + graph[i][j]
                        prefer = j
        visited.append(prefer)
        node.remove(prefer)
    return distance


start_time = datetime.now()
distance = dijkstra(graph_list, 0) #запуск алгоритма
res_time = datetime.now() - start_time
print("Время работы алгоритма: {}.".format(res_time))

print(distance)
```

Рисунок 2.3.19 – Алгоритм Дейкстры

Результат его работы представлен на рисунке ниже.

```
Время работы алгоритма: 0:00:00.000068.
{0: 0, 1: 3, 2: 1, 3: 3, 4: 8, 5: 5}
```

Рисунок 2.3.20 – Результат работы алгоритма Дейкстры

Программа возвращает список суммы весов кратчайших путей от первой (нулевой) вершины ко всем остальным. Например, запись 1: 3 означает, что до вершины с индексом 1, то есть до узла B, в сумме нужно проехать 3 километра, и это будет кратчайший путь в данном графе до него. При таком малом количестве вершин графа алгоритм Дейкстры пока справляется быстрее генетического даже несмотря на то, что ему нужно пройти все узлы графа для получения ответа. Далее увеличим число вершин и запустим оба алгоритма снова. Для этого следуют написать код, который будет инициализировать матрицу смежности самостоятельно с заданным количеством вершин. На рисунке 2.3.21 представлена его реализация.

```
#инициализация матрицы смежности рандомными значениями
N = 50
inf = 1000

D = [[0] * N for i in range(N)] #создание списка
#заполнение симметричного двумерного массива рандомными значениями
for i in range(N):
     for j in range(N):
       if i == j: D[i][j]=0
       elif i<j:
         if random.randint(0,1) == 0: D[i][j] = random.randint(1, 11)
         else:                        D[i][j] = inf
       else:
          D[i][j] = D[j][i]

#вывод массива
for i in range(len(D)):
    for j in range(len(D[i])):
        print(D[i][j], end=' ')
    print()
```

Рисунок 2.3.21 – Инициализация двумерного массива случайными значениями и его вывод

Переменная N отвечает за количество строк и столбцов массива. Данный код обеспечивает симметричность матрицы и инициализирует матрицу значениями от 1 до 11, имея некоторый шанс присвоить значение inf, то есть, как упоминалось ранее, не существует известного пути между данными вершинами. Значения главной диагонали равны нулю. Результат работы кода представлен на рисунке 2.3.22.

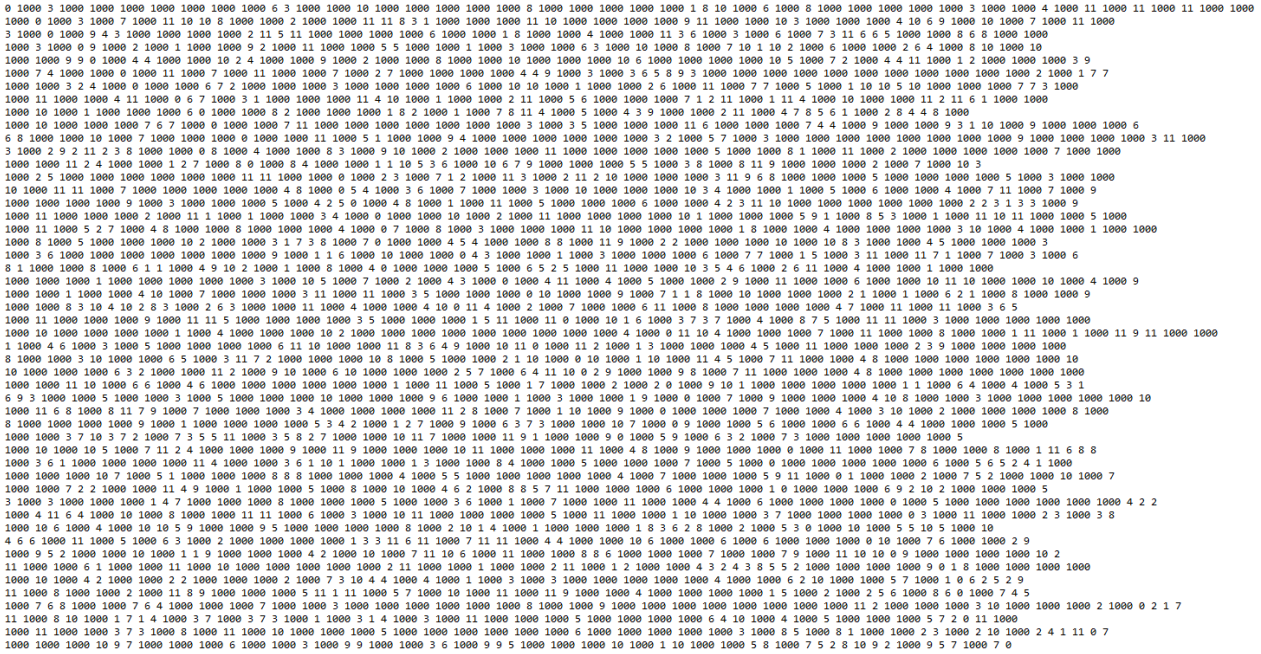

Рисунок 2.3.22 – Инициализированный двумерный массив размером 50 на 50 элементов

Теперь запустим алгоритм Дейкстры для данного массива.

```
Время работы алгоритма: 0:00:00.007392.
{0: 0, 1: 7, 2: 3, 3: 5, 4: 4, 5: 4, 6: 4,
7: 5, 8: 5, 9: 7, 10: 5, 11: 3, 12: 5, 13: 5,
14: 5, 15: 5, 16: 4, 17: 5, 18: 6, 19: 4, 20: 6,
21: 4, 22: 4, 23: 5, 24: 5, 25: 4, 26: 1, 27: 6,
28: 5, 29: 3, 30: 5, 31: 2, 32: 4, 33: 4, 34: 7,
35: 5, 36: 5, 37: 4, 38: 3, 39: 4, 40: 4, 41: 4,
42: 6, 43: 3, 44: 4, 45: 6, 46: 6, 47: 5, 48: 5, 49: 4}
```

Рисунок 2.3.23 – Результат работы алгоритма Дейкстры с массивом 50 на 50 элементов

С увеличением количества вершин с 6 до 50 время работы данного алгоритма увеличилось приблизительно в 108 раз. Увеличим значение N с 50 до 100 и запустим алгоритм еще раз.

```
Время работы алгоритма: 0:00:00.065513.
{0: 0, 1: 1, 2: 4, 3: 3, 4: 3, 5: 3, 6: 4,
 7: 3, 8: 3, 9: 4, 10: 3, 11: 3, 12: 3, 13: 3,
 14: 4, 15: 1, 16: 3, 17: 3, 18: 4, 19: 2,
 20: 3, 21: 3, 22: 2, 23: 1, 24: 2, 25: 2,
26: 3, 27: 5, 28: 3, 29: 3, 30: 3, 31: 3,
32: 3, 33: 3, 34: 4, 35: 2, 36: 3, 37: 3,
38: 3, 39: 3, 40: 3, 41: 1, 42: 3, 43: 2,
44: 2, 45: 3, 46: 3, 47: 3, 48: 2, 49: 4,
50: 2, 51: 1, 52: 3, 53: 3, 54: 3, 55: 3,
56: 3, 57: 2, 58: 3, 59: 2, 60: 2, 61: 3,
62: 2, 63: 3, 64: 3, 65: 2, 66: 3, 67: 3,
68: 3, 69: 3, 70: 3, 71: 3, 72: 2, 73: 3,
 74: 3, 75: 4, 76: 1, 77: 3, 78: 1, 79: 3,
80: 2, 81: 2, 82: 3, 83: 2, 84: 2, 85: 3,
86: 3, 87: 1, 88: 3, 89: 4, 90: 2, 91: 4,
92: 3, 93: 2, 94: 1, 95: 5, 96: 3, 97: 2, 98: 2, 99: 3}
```

Рисунок 2.3.24 – Результат работы алгоритма Дейкстры с массивом 100 на 100 элементов

После увеличения размера матрицы в два раза, время на расчет ответа увеличилось почти в 9 раз. Теперь проведем такие же тесты для генетического алгоритма.

Результат работы генетического алгоритма для 50 вершин. Опытным путем было выяснено, что количество поколений достаточно 5, количество особей 600. При увеличении их числа конечный результат не менялся.

gen	nevals	min	avg
0	600	510115	653099
1	17	510115	608339
2	13	502239	576414
3	11	492046	551558
4	11	434027	533423
5	10	434027	519605

Рисунок 2.3.25 – Результат работы генетического алгоритма с матрицей 50 на 50 элементов. Часть 1

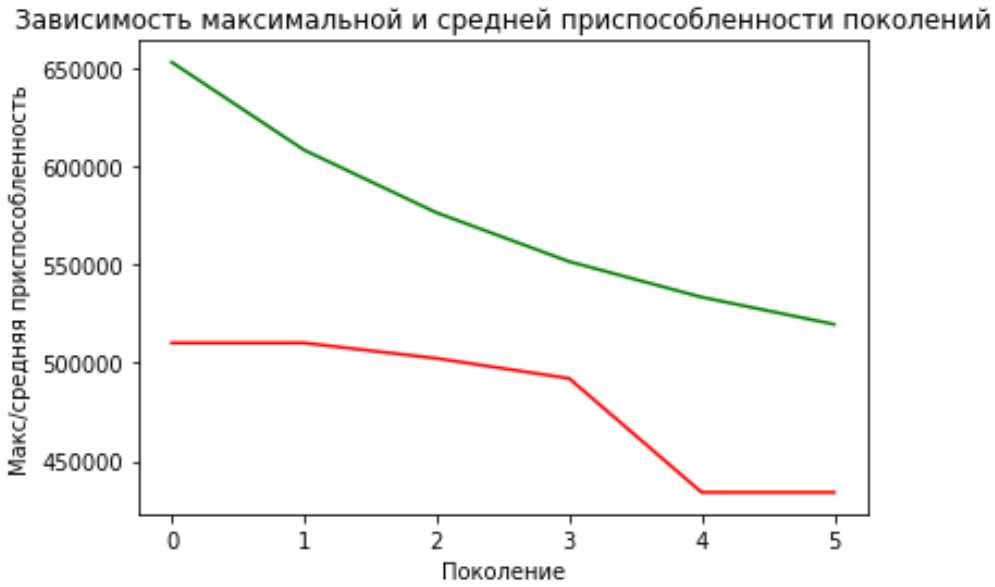

Рисунок 2.3.25 – Результат работы генетического алгоритма с матрицей 50 на 50 элементов. Часть 2

Время работы алгоритма 6.460788 секунд. То есть время работы генетического алгоритма выросло в 2 раза, что гораздо меньше, чем у алгоритма Дейкстры.

Теперь запустим генетический алгоритм для 100 вершин с такими же значениями весов, как было для Дейкстры.

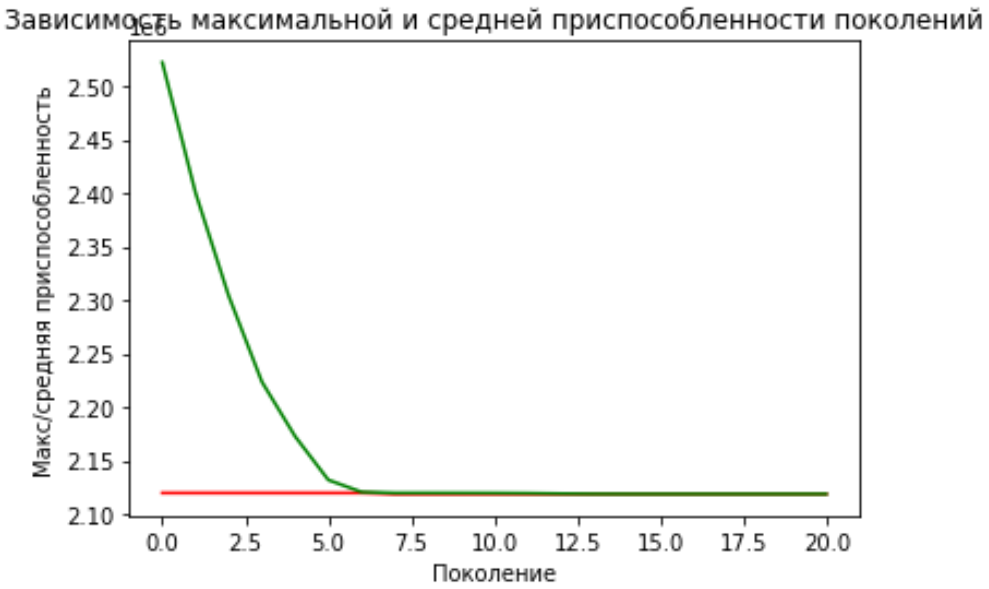

Рисунок 2.3.26 – Результат работы генетического алгоритма с графом со 100 вершинами. 300 особей. Часть 1

```
gen     nevals  min             avg
0       300     2.11958e+06     2.522e+06
1       4       2.11958e+06     2.40016e+06
2       4       2.11958e+06     2.30377e+06
3       0       2.11958e+06     2.22342e+06
4       5       2.11958e+06     2.17247e+06
5       0       2.11958e+06     2.13173e+06
6       6       2.11958e+06     2.12045e+06
7       5       2.11859e+06     2.11957e+06
8       4       2.11859e+06     2.11956e+06
9       5       2.11859e+06     2.11953e+06
10      2       2.11859e+06     2.11947e+06
11      3       2.11859e+06     2.1193e+06
12      7       2.11859e+06     2.11894e+06
13      2       2.11859e+06     2.11864e+06
14      2       2.11859e+06     2.11859e+06
15      2       2.11859e+06     2.11859e+06
16      0       2.11859e+06     2.11859e+06
17      4       2.11859e+06     2.11859e+06
18      2       2.11859e+06     2.11859e+06
19      3       2.11859e+06     2.11859e+06
20      2       2.11859e+06     2.11859e+06
Время работы алгоритма: 0:00:39.916063.
Хромосома лучшей особи, которая хранит следующие списки кратчайших маршрутов:
[[50, 10, 68, 11, 81, 84, 71, 22, 27, 51, 45, 2, 62, 83, 24, 37, 93, 14, 95, 69, 49, 59, 85,
 13, 15, 80, 87, 53, 36, 7, 96, 70, 43, 28, 67, 98 ... ]
```

Рисунок 2.3.27 – Результат работы генетического алгоритма с графом со 100 вершинами. 300 особей. Часть 2

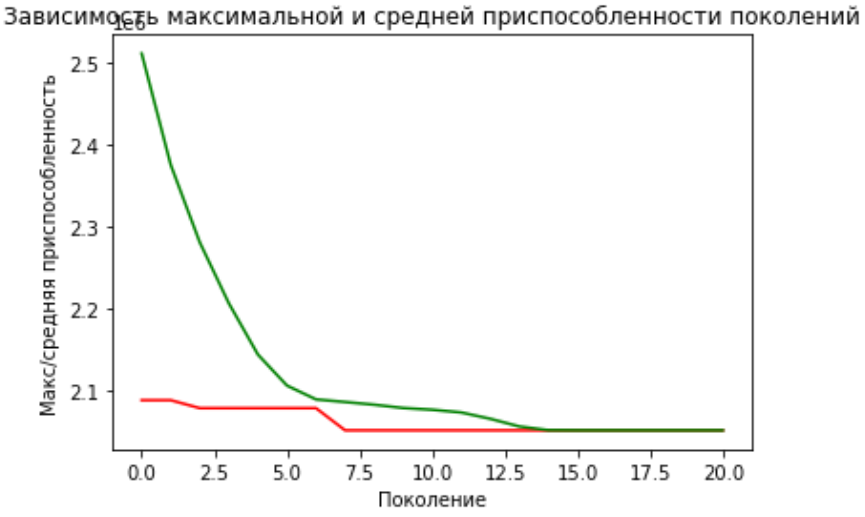

Рисунок 2.3.28 – Результат работы генетического алгоритма с графом со 100 вершинами. 600 особей. Часть 1

```
gen     nevals  min             avg
0       600     2.08712e+06     2.51085e+06
1       3       2.08712e+06     2.37467e+06
2       4       2.07739e+06     2.27995e+06
3       10      2.07739e+06     2.2055e+06
4       1       2.07739e+06     2.14287e+06
5       10      2.07739e+06     2.10498e+06
6       5       2.07739e+06     2.08811e+06
7       4       2.0503e+06      2.0849e+06
8       10      2.0503e+06      2.08161e+06
9       10      2.0503e+06      2.07754e+06
10      0       2.0503e+06      2.07545e+06
11      5       2.0503e+06      2.07215e+06
12      4       2.0503e+06      2.06436e+06
13      4       2.0503e+06      2.05497e+06
14      1       2.0503e+06      2.05048e+06
15      4       2.0503e+06      2.0503e+06
16      2       2.0503e+06      2.0503e+06
17      5       2.0503e+06      2.0503e+06
18      6       2.0503e+06      2.0503e+06
19      7       2.0503e+06      2.0503e+06
20      4       2.0503e+06      2.0503e+06
Время работы алгоритма: 0:01:18.517268.
Хромосома лучшей особи, которая хранит следующие списки кратчайших маршрутов:
[[36, 41, 40, 71, 15, 90, 44, 66, 12, 72, 99, 83, 34, 61, 25, 37, 42, 69, 49, 23,
 46, 92, 10, 76, 39, 93, 54, 7, 14, 70, 64, 2, 58, 68, 21, 9, 55, ... ]
```

Рисунок 2.3.29 – Результат работы генетического алгоритма с графом со 100 вершинами. 600 особей. Часть 2

На рисунках 2.3.26 – 2.3.27 показаны результаты работы генетического алгоритма с 300 особями в начальном поколении, а на рисунках 2.3.28 – 2.3.29 с 600 особями. Внимательно изучив результаты, можно заметить, что хромосомы лучших особей получились у них разными. Чем больше будет сделано итераций (максимальное число поколений), тем лучше будет конечный результат. Также хорошо видно, что с увеличением числа особей значительно увеличилось время выполнения алгоритма.

Далее были проведены еще несколько тестов (на них у генетического алгоритма 300 особей и 10 поколений) с разным количеством вершин графа и их результаты отображены на графиках ниже. На рисунках 2.3.30 – 2.3.31 отображены зависимости времени работы от количества узлов в графах.

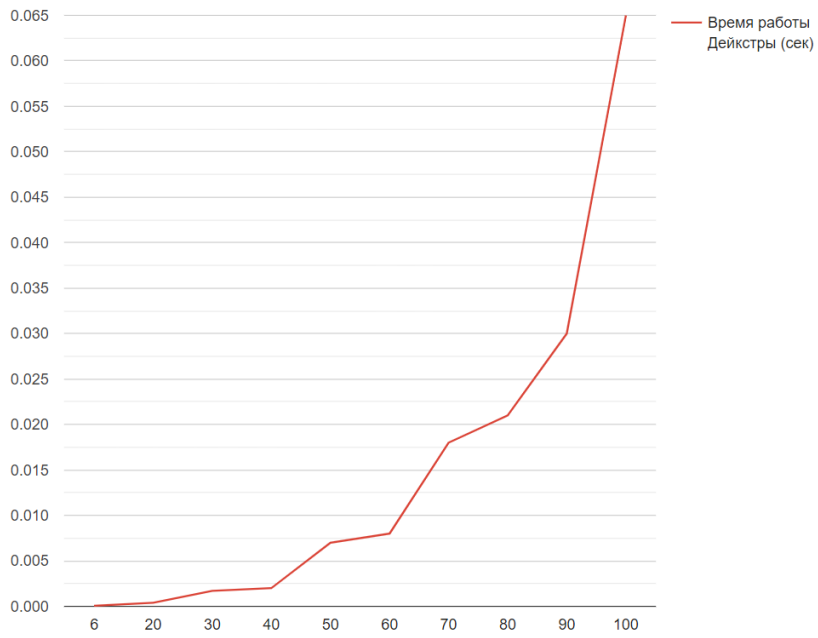

Рисунок 2.3.30 – Зависимость времени работы от количества вершин для алгоритма Дейкстры

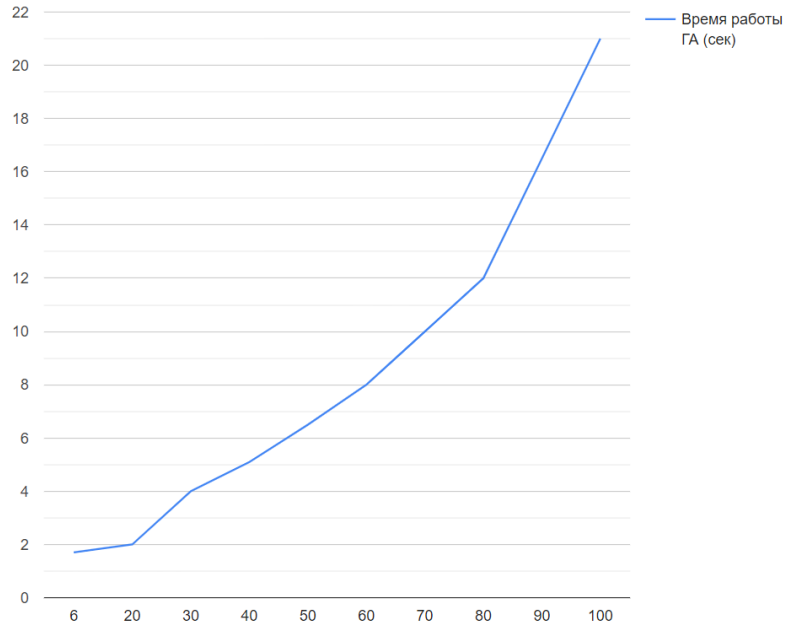

Рисунок 2.3.31 – Зависимость времени работы от количества вершин для генетического алгоритма

На рисунке 2.3.32 изображен график, на котором наглядно видно, что у алгоритма Дейкстры уходило гораздо меньше времени на работу, чем у генетического при одинаковом количестве вершин.

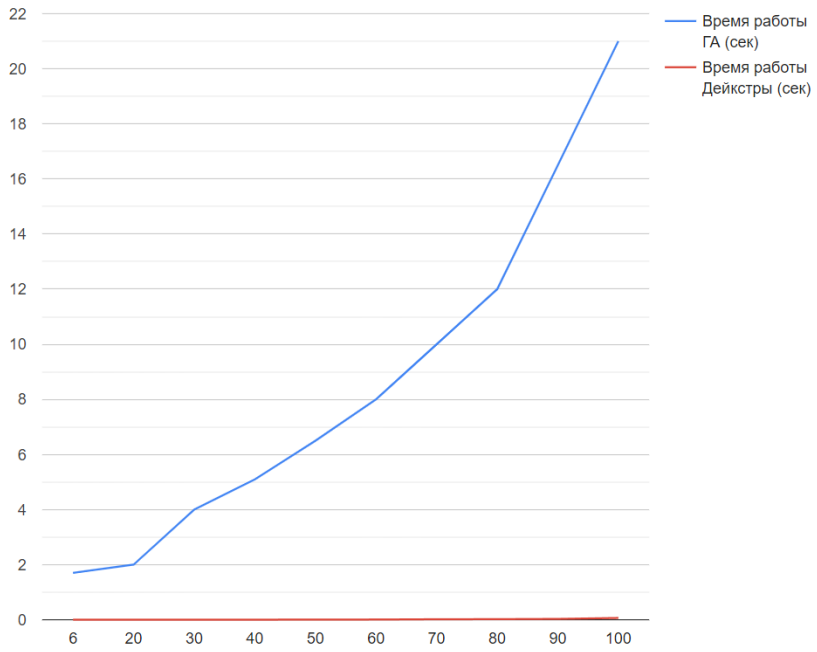

Рисунок 2.3.32 – График отношений затрат времени от количества узлов для двух рассматриваемых алгоритмов

Как видно на графиках выше, у генетического алгоритма на работу уходит больше времени, чем у Дейкстры, и это обосновывается тем, что первый сильно зависит от начальных данных (начальной популяции), а также от числа поколений и особей в них. Большое число особей в поколениях увеличивают шанс получения лучшего результата, но тогда приходится тратить больше ресурсов и времени на их обработку.

Выводы по разделу 2

Приведем выводы по второму разделу бакалаврской работы:

– были рассмотрены плюсы и минусы алгоритмов Дейкстры и генетического;

– была более подробно рассмотрена структура генетического алгоритма и по ней написан код на языке Python в среде Colaboratory с использованием пакета DEAP;

– были проведены тесты реализованного генетического алгоритма и алгоритма Дейкстры, а после приведены сравнения их результатов. В итоге

было выяснено, что генетический алгоритм тратит на решение задачи больше времени и это обосновывается тем, что он сильно зависит от заданных параметров.

3 АНАЛИЗ И ВЕРИФИКАЦИЯ ПОЛУЧЕННЫХ ДАННЫХ

3.1 Проведение вычислительного эксперимента

В предыдущем разделе было проведено несколько тестов, где сравнивались генетический алгоритм и алгоритм Дейкстры. В результате их сравнения было выяснено, что имитационное моделирование биологической эволюции тратит больше времени на расчет ответа задачи, чем посещение всех узлов графа в задаче поиска кратчайшего пути.

Сравнение данных двух алгоритмов проводилось с разным количеством вершин в графах, которые генерировались случайным образом с помощью дополнительно написанной функции. Данные в графах для двух рассматриваемых алгоритмов и их размер были одинаковы, чтобы обеспечить корректное получение результатов тестов.

При количестве шести вершин, генетический алгоритм справился с расчетом за 1,7 секунд, а Дейкстра за 0,00007 секунд. При 50 узлах у генетического алгоритма ушло 6,5 секунд, у Дейкстры – 0,007 секунд. В графе со 100 вершинами генетическому алгоритму понадобилось 21 секунда на расчет ответа, у алгоритма Дейкстры ушло 0,07 секунд.

Таблица 3.1.1 – Результаты тестов

Количество вершин	Время работы генетического алгоритма (сек)	Время работы алгоритма Дейкстры (сек)
6	1,7	0,000068
20	2	0,0004
30	4	0,0017
40	5,1	0,002
50	6,5	0,007
60	8	0,008
70	10	0,018
80	12	0,021
90	16,5	0,03
100	21	0,065

3.2 Корректировка разработанного алгоритма

При выполнении исследования была выполнена главная поставленная в начале работы задача – разработка алгоритма для поиска наилучшего маршрута в городских условиях с использованием эволюционных вычислений, среди которых был выбран генетический алгоритм. На основе проведенных тестов видно, что программа выдает верные результаты, то есть, кратчайший путь.

В приложениях, который ищут кратчайший путь, можно выбирать параметры, чтобы настроить свой маршрут. Сейчас код написан так, что рассматриваемый параметр нужно менять непосредственно в функции, где идет расчет вес дуг ко всем остальным узлам. Чтобы это исправить, можно воспользоваться модулем enum – тип для перечисления значений, который можно использовать для понятных обозначений. В качестве таких обозначений будут выступать рассматриваемые пользователем параметры.

```
import enum

# Перечисление рассматриваемых параметров
class Param(enum.Enum):
  km = 0
  price = 1
  time = 2

# Здесь меняется рассматриваемый параметр
param = Param.km
```

Рисунок 3.2.1 – Добавление enum в код

Для использования модуля enum необходимо было импортировать его библиотеку. Затем был создан класс с именем Param, который является наследником класса enum.Enum. Атрибутами его класса являются рассматриваемые при нахождении кратчайшего пути параметры. К ним можно обращаться либо по имени, либо по номерам. Чтобы поменять параметр на нужный, теперь необходимо будет поменять значение переменной param. На рисунке 3.2.1 в качестве параметра указано расстояние

в километрах. Чтобы поменять это значение, нужно изменить атрибут km на доступный из перечисления Param другой атрибут. Также частично был изменен код функции dikstryFitness, как показано на рисунке 3.2.2.

```
# расчитываем длину/цену/время маршрута до всех остальных вершин
def dikstryFitness(individual):
    s = 0
    for n, path in enumerate(individual):
        path = path[:path.index(n) + 1]

        si = startV
        for j in path:
          if (param.name == "km"):
              s += D[si][j].km
          elif (param.name == "price"):
              s += D[si][j].price
          else: s += D[si][j].time
        si = j

    return s,
```

Рисунок 3.2.2 – Изменение функции dikstryFitness

В функцию были добавлены операторы условия, которые в зависимости от выбранного параметра в переменной param определяют, какой параметр необходимо считать. Теперь, если нужно будет поменять рассматриваемый параметр, то это нужно будет сделать только в одном месте – в переменной param, не изменяя саму функцию.

Запустив код, можно увидеть, что результат выдается верный.

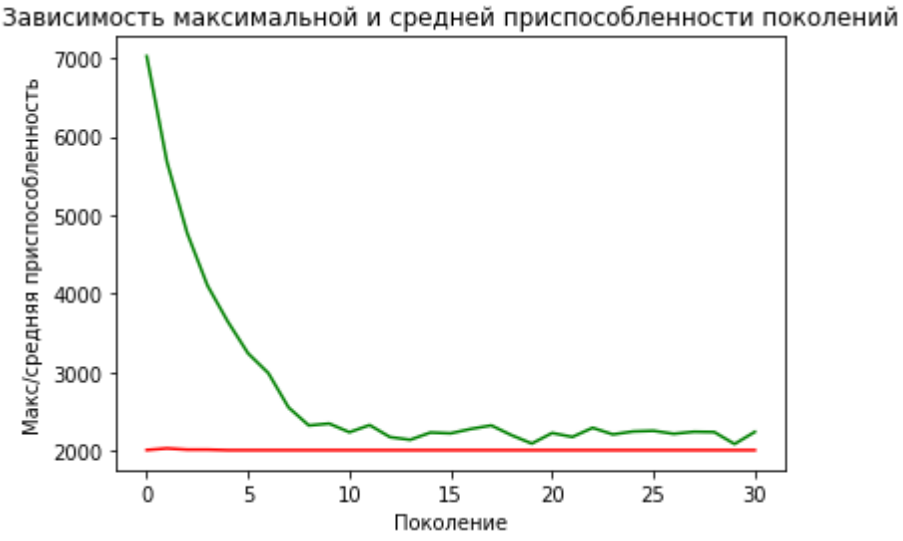

Рисунок 3.2.3 – Результат работы генетического алгоритма после внесения правок. Часть 1

```
Хромосома лучшей особи, которая хранит следующие списки кратчайших маршрутов:
[[0, 1, 4, 3, 5, 2], [1, 4, 3, 0, 5, 2], [2, 0, 5, 1, 4, 3], [3, 2, 4, 5, 1, 0],
 [2, 4, 1, 5, 3, 0], [3, 5, 1, 4, 0, 2]]
Время работы алгоритма: 0:00:01.729274.
```

Рисунок 3.2.4 – Результат работы генетического алгоритма после внесения правок. Часть 2

Выводы по разделу 3

Приведем выводы по третьему разделу бакалаврской работы:

– по полученным данным из прошлого раздела, была составлена таблица результатов, где указано время работы каждого из рассматриваемых алгоритмов в зависимости от количества вершин в графах;

– был добавлен модуль enum (тип для перечисления значений), с помощью которого теперь можно проще переключать рассматриваемые параметры для решения задачи.

Заключение

Для решения задач маршрутизации используют различные алгоритмы, которые наиболее всего подходят к поставленной задаче. Известно, что эволюционные алгоритмы используются в решении NP-сложных задач, включая задачи маршрутизации. А в поиске кратчайшего пути активно используют модифицированный алгоритм Дейкстры. В литературе можно найти сильные и слабые стороны этих алгоритмов, но нигде не приводятся результаты сравнения в их производительности на одинаковых задачах – обычно эволюционные алгоритмы сравниваются только между собой. Среди эволюционных алгоритмов мною был выбран генетический и на задаче поиска кратчайшего пути его производительность сравнивалась с активно используемым в решении таких задач алгоритмом Дейкстры.

Генетический алгоритм, как было выяснено в результате исследований, сильно зависит от параметров и даже небольшое изменение одного из них может привести как к сильному ухудшению, так и к улучшению полученных решений. Влияние начальных параметров хорошо заметно на популяциях, полученных в результате имитации эволюции начальной популяции, которая в моей работе генерировалась случайными значениями, где каждая особь популяции хранила в себе лучшие, по ее мнению, маршруты, хранящиеся у нее в хромосоме. На основе этих хромосом выбирались лучшие, т. е. более приспособленные по мнению алгоритма особи, которые с заданной вероятностью могли скрещиваться, а затем с некоторой вероятностью могла происходить мутация полученных новых особей. Вероятности мутации и скрещивания также сильно влияют на результирующий ответ. Чем больше популяций и особей в них, тем больше вероятность получения хорошего результата. Тем не менее, если задача состоит в поиске оптимального пути для, например, 20 и более городов, то возникает задача коммивояжера, с решением которой генетический алгоритм покажется себя лучше Дейкстры.

На основе полученных в результате тестирования оценки производительности генетического алгоритма в дальнейшем можно будет его усовершенствовать, чтобы ускорить его работу с сохранением точности полученных решений. Например, можно будет добавить в программу многопоточность, чтобы оказалось возможным обрабатывать несколько переменных (особей в популяции) за раз. Еще одним вариантом его усовершенствования можно быть внедрение в него частей других алгоритмов (гибридизация) и внесение в него модификаций, как было сделано с алгоритмом Дейкстры в таких приложениях как Google Maps, Apple maps, OpenStreetMap, Яндекс.Карты и многих других. Возможно, добавление сильной стороны генетического алгоритма в алгоритм Дейкстры в качестве модификации поможет ускорить его еще больше, особенно в тех моментах, когда решение задачи может быть не очевидным (т. е. невозможно получить точный ответ), либо же ее расчет займет настолько много времени, что достаточным будет получение приблизительно оптимального решения.

Список используемой литературы и источников

1. Александр Ершов. Азбука ИИ: «Эволюционные алгоритмы» // N+1. 2016. URL: https://nplus1.ru/material/2016/07/06/evodevo (дата обращения: 10.4.2022).
2. Александр Гришутин, Станислав Алексеевич. Поиск в ширину // Алгоритмика. URL: https://ru.algorithmica.org/cs/shortest-paths/bfs/ (дата обращения: 16.4.2022).
3. Генетический алгоритм // Википедия. [2022]. Дата обновления: 23.02.2022. URL: https://ru.wikipedia.org/?curid=16652&oldid=120270996 (дата обращения: 25.03.2022).
4. Дмитрий Трофимов, Анатолий Федуков. Задача маршрутизации транспорта. Дискретная математика: алгоритмы // Lobanov Logist. URL: https://www.lobanov-logist.ru/library/all_articles/55059/ (дата обращения: 25.3.2022).
5. Е. Г. Климко. Генетический алгоритм как разновидность эволюционного алгоритма // Cyber Leninka. URL: https://cyberleninka.ru/article/n/geneticheskiy-algoritm-kak-raznovidnost-evolyutsionnogo-algoritma (дата обращения: 12.3.2022).
6. Задача коммивояжёра // Википедия. [2022]. Дата обновления: 06.04.2022. URL: https://ru.wikipedia.org/?curid=45128&oldid=121246343 (дата обращения: 16.04.2022).
7. Компания Яндекс – Технологии – Маршрутизация // Яндекс. URL: https://yandex.ru/company/technologies/routes/ (дата обращения: 15.3.2022).
8. Классический генетический алгоритм. Часть 1. Краткий обзор // AIportal. URL: http://www.aiportal.ru/articles/genetic-algorithms/classic-alg-part1.html (дата обращения: 13.3.2022).
9. Поиск в ширину // Википедия. [2021]. Дата обновления: 10.10.2021.

URL: https://ru.wikipedia.org/?curid=227593&oldid=117122792 (дата обращения: 16.04.2022).

10. Реализация алгоритма Дейкстры на Python // Русские Блоги. URL: https://russianblogs.com/article/90771009032/ (дата обращения: 18.4.2022).

11. Роберт Седжвик. Алгоритмы на c++. Лекция 21: Кратчайшие пути // ИНТУИТ. URL: https://intuit.ru/studies/courses/12181/1174/lecture/25268?page=9 (дата обращения: 8.4.2022).

12. Эволюционные алгоритмы // Википедия. [2020]. Дата обновления: 01.01.2020. URL: https://ru.wikipedia.org/?curid=1304904&oldid=104297578 (дата обращения: 15.3.2022).

13. Эволюционные алгоритмы // Постнаука. 2016. URL: https://postnauka.ru/tv/69879 (дата обращения: 16.4.2022).

14. Эволюционный алгоритм (Evalutionary algorithm) // Loginom. URL: https://wiki.loginom.ru/articles/evolution-algorithm.html (дата обращения: 12.4.2022).

15. Эволюционные методы для решений задач проектирования и логистики // БиГОР. URL: http://bigor.bmstu.ru/?cnt/?doc=Default/Evolution.cou (дата обращения: 5.3.2022).

16. ASF Infrabot. .GeneticAlgorithm // Apache. 2009. URL: https://cwiki.apache.org/confluence/display/SPAMASSASSIN/GeneticAlgorithm (дата обращения: 14.3.2022).

17. Automated Antenna Design with Evolutionary Algorithms // Ti. URL: https://ti.arc.nasa.gov/m/pub-archive/1244h/1244%20(Hornby).pdf (дата обращения: 17.4.2022).

18. Somasundaram Kumanan. A Genetic Algorithm for Optimization Of Supply Chain Logistics Network // ResearchGate. 2005. URL: https://www.researchgate.net/publication/267212044_A_Genetic_Algorithm_for_

Optimization_Of_Supply_Chain_Logistics_Network (дата обращения: 22.4.2022).

19. Fmder. DEAP documentation. // Github. 2020. URL: documentation https://deap.readthedocs.io/en/master/ (дата обращения: 10.3.2022).

20. Genetic Query Optimization (GEQO) in PostgreSQL // PostgreSQL. URL: https://www.postgresql.org/docs/9.1/geqo-pg-intro.html (дата обращения: 16.4.2022).

21. Mafulechka. Поиск в ширину (Breadth first search, BFS) // Evileg. 2019. URL: https://evileg.com/ru/post/512/ (дата обращения: 18.4.2022).

22. Muthukumar Kumar. The famous algorithm that made navigation in Google Maps a reality // GEO Awesome. 2015. URL: https://geoawesomeness.com/the-famous-algorithm-that-made-navigation-in-google-maps-a-reality/ (дата обращения: 15.4.2022).

23. Nick Johnson. What algorithms compute directions from point A to point B on a map? // Stack overflow. 2009. URL: https://stackoverflow.com/questions/430142/what-algorithms-compute-directions-from-point-a-to-point-b-on-a-map (дата обращения: 16.4.2022).

24. PatientZero. Эволюционные вычисления: учим табуретку ходить // Хабр. 2017. URL: https://habr.com/ru/post/340772/ (дата обращения: 15.4.2022).

25. Welcome To Colaboratory // Google. URL: https://colab.research.google.com/?utm_source=scs-index (дата обращения: 10.3.2022).

Приложение А

Листинг программы

```
!pip install deap
from deap import base, algorithms, creator, tools
import numpy as np
import random
import matplotlib.pyplot as plt
from datetime import datetime
import time
import enum
# Перечисление рассматриваемых параметров
class Param(enum.Enum):
  km = 0
  price = 1
  time = 2
# Здесь меняется рассматриваемый параметр
param = Param.km
#вес дуги, который включает в себя км, цену и среднюю скорость
class Weight(object):
    def __init__(self, km, price, km_h):
        self.km = km
        self.price = price
        self.km_h = km_h
        if (km != 0): self.time = (km/km_h)*60
        else:         self.time = 0

#Кол-во хромосом с лучшим результатом
BEST_CHROMOSOME_SIZE = 1
hof = tools.HallOfFame(BEST_CHROMOSOME_SIZE)
```

```
inf = 1000
price_pub = 8 #цена за проезд в общественном транспорте
#Матрица смежности
#     A                B                C                D                E
F
D  =  ((Weight(0,0,0),          Weight(3,12,40),          Weight(1,16,25),
Weight(3,12,25),    Weight(inf,inf,inf), Weight(inf,inf,inf)),  # A
       (Weight(3,12,40),          Weight(0,0,0),          Weight(4,16,40),
Weight(inf,inf,inf), Weight(inf,inf,inf), Weight(inf,inf,inf)),  # B
       (Weight(1,16,25),          Weight(4,16,40),          Weight(0,0,0),
Weight(inf,inf,inf), Weight(7,28,60),    Weight(5,20,30)),    # C
       (Weight(3,12,25),          Weight(inf,inf,inf),      Weight(inf,inf,inf),
Weight(0,0,0),      Weight(inf,inf,inf), Weight(2,8,30)),     # D
       (Weight(inf,inf,inf),      Weight(inf,inf,inf),      Weight(7,28,60),
Weight(inf,inf,inf), Weight(0,0,0),      Weight(4,16,40)),    # E
       (Weight(inf,inf,inf),      Weight(inf,inf,inf),      Weight(5,20,30),
Weight(2,8,30),     Weight(4,16,40),    Weight(0,0,0)))      # F
startV = 0                  #стартовая вершина
LENGTH_D = len(D)
LENGTH_CHROM = len(D)*len(D[0]) #длина хромосомы, подлежащей
оптимизации
#константы генетического алгоритма
POPULATION_SIZE = 500   #кол-во индивидуумов в популяции
P_CROSSOVER = 0.9       #вероятность скрещивания
P_MUTATION  = 0.1       #вероятность муиацмм индивидуума
MAX_GENERATONS = 30     #максимаьное кол-во поколений
#Случайное число (зерно) для генерации случайных чисел
RANDOM_SEED = 42
random.seed(RANDOM_SEED)
```

#FitnessMin - класс хранит информация о приспособленности каждого индивидуума

```
creator.create("FitnessMin", base.Fitness, weights = (-1.0,))
creator.create("Individual", list, fitness = creator.FitnessMin)
toolbox = base.Toolbox()
toolbox.register("randomOrder", random.sample, range(LENGTH_D), LENGTH_D)
toolbox.register("individualCreator", tools.initRepeat, creator.Individual, toolbox.randomOrder, LENGTH_D)
toolbox.register("populationCreator", tools.initRepeat, list, toolbox.individualCreator)
population = toolbox.populationCreator(n=POPULATION_SIZE)
# расчитываем длину/цену/время маршрута до всех остальных вершин
def dikstryFitness(individual):
    s = 0
    for n, path in enumerate(individual):
        path = path[:path.index(n) + 1]
        si = startV
        for j in path:
          if (param.name == "km"):
            s += D[si][j].km
          elif (param.name == "price"):
            s += D[si][j].price
          else: s += D[si][j].time
        si = j
    return s,
# определяет способ скрещивания двух особей
```

```
def cxOrder(ind1, ind2):
```

Продолжение Приложения А

```
    for p1, p2 in zip(ind1, ind2):
        tools.cxOrdered(p1, p2)
    return ind1, ind2
# определяет порядок мутаций генов в хромосоме. Перемешиваем индексы в списке
def mutShuffleIndexes(individual, indpb):
    for ind in individual:
        tools.mutShuffleIndexes(ind, indpb)
    return individual,
# регистрируем необходимые функции для реализации ген. алгоритма
toolbox.register("evaluate", dikstryFitness)  # эволюция
toolbox.register("select", tools.selTournament, tournsize=3)  # отбор
toolbox.register("mate", cxOrder)  # скрещивание
toolbox.register("mutate", mutShuffleIndexes, indpb=1.0 / LENGTH_CHROM / 10)  # мутация
# Сбор статистики
stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("min", np.min)  # мин. значение приспособленности особи
stats.register("avg", np.mean)  # среднее значение присособленности всех особей в популяциях
start_time = datetime.now() #засекаем время, запоминая время начала работы
# запуск генетического алгоритма
population, logbook = algorithms.eaSimple(population, toolbox, cxpb=P_CROSSOVER / LENGTH_D, mutpb=P_MUTATION / LENGTH_D, ngen=MAX_GENERATONS, halloffame=hof, stats=stats, verbose=True)
```

res_time = datetime.now() - start_time #время выполнения = текущее время - время начала работы

Продолжение Приложения А

```
# собираем статистику
maxFitnnessValues, meanFitnessValues = logbook.select("min", "avg")
# выводим результат
plt.plot(maxFitnnessValues, color='red')
plt.plot(meanFitnessValues, color='green')
plt.xlabel('Поколение')
plt.ylabel('Макс/средняя приспособленность')
plt.title('Зависимость максимальной и средней приспособленности поколений')
plt.show()
best = hof.items[0]
print("Хромосома лучшей особи, которая хранит следующие списки кратчайших маршрутов:")
print(best)
print("Время работы алгоритма: {}.".format(res_time))
```