**Содержание**

[Введение 3](#_Toc197894118)

[1. Анализ проблемы 6](#_Toc197894119)

[1.1 Обзор существующих решений 6](#_Toc197894120)

[1.2 Подходы к построению маршрутов 7](#_Toc197894121)

[1.2.1 Математическая модель задачи 7](#_Toc197894122)

[1.2.2 Алгоритмы решения задачи 8](#_Toc197894123)

[1.3 Учет загруженности дорог 12](#_Toc197894124)

[1.4 Выводы по главе 13](#_Toc197894125)

[2. Формальная постановка задачи 17](#_Toc197894126)

[2.1 Постановка задачи маршрутизации 17](#_Toc197894127)

[2.2 Входные данные 17](#_Toc197894128)

[2.3 Выходные данные 18](#_Toc197894129)

[2.4 Математическая формализация 18](#_Toc197894130)

[3. Алгоритм решения задачи и архитектура программы 20](#_Toc197894131)

[3.1 Общая структура и цели приложения 20](#_Toc197894132)

[3.2 Архитектура и взаимодействие компонентов 20](#_Toc197894133)

[3.3 Используемые библиотеки 23](#_Toc197894134)

[3.4 Логика маршрутизации 23](#_Toc197894135)

[3.5 Работа с базой данных 26](#_Toc197894136)

[3.6 Графический интерфейс пользователя 27](#_Toc197894137)

[3.7 Выводы по главе 29](#_Toc197894138)

[4. Тестирование 32](#_Toc197894139)

[4.1 Методика тестирования 32](#_Toc197894140)

[4.2 Ручное тестирование 32](#_Toc197894141)

[4.3 Автоматизированные тесты 34](#_Toc197894142)

[4.4 Выводы по главе 38](#_Toc197894143)

[5. Руководство пользователя и администратора 40](#_Toc197894144)

[5.1 Как запустить приложение 40](#_Toc197894145)

[5.2 Пошаговая инструкция пользователя 41](#_Toc197894146)

[5.3 Интерпретация результата 42](#_Toc197894147)

[5.4 Проверка сохранения маршрута 42](#_Toc197894148)

[5.5 Как расширить карту на другой город 42](#_Toc197894149)

[Заключение 44](#_Toc197894150)

[Список использованных источников 46](#_Toc197894151)

[Приложение А 47](#_Toc197894152)

[Приложение Б 58](#_Toc197894153)

# Введение

В условиях стремительного роста числа автомобилей, усложнения улично-дорожных сетей и увеличения транспортных потоков в городах задача построения эффективных маршрутов приобретает всё большую значимость. Плотность дорожного движения, пробки, аварии и ремонтные работы оказывают существенное влияние на время перемещения по городу, что требует от навигационных систем высокой адаптивности и интеллектуальности.

Современные логистические и транспортные системы всё чаще ориентируются не только на минимизацию расстояния, но и на сокращение времени доставки с учетом текущей или типичной загруженности дорог. Особенно остро проблема стоит в крупных городах, где даже незначительная задержка на одном участке дороги может привести к значительным временным потерям на всём маршруте.

Традиционные системы маршрутизации (Google Maps, Яндекс.Карты) во многом являются "чёрными ящиками", не предоставляющими возможности глубокой кастомизации или интеграции в пользовательские системы. В то же время, развитие открытых источников географических данных, таких как OpenStreetMap (OSM), и появление библиотек Python (OSMnx, NetworkX, folium) позволяют разработать собственные системы маршрутизации, учитывающие как топологию города, так и параметры, влияющие на загруженность участков сети.

Таким образом, задача построения маршрутов с учётом реальных условий движения (скорости, времени в пути, пропускной способности) остаётся актуальной и практически значимой как в области городского планирования, так и в логистике, транспорте, и мобильных сервисах.

***Объектом исследования*** является улично-дорожная сеть города, представленная в виде ориентированного графа, а также связанные с ней процессы построения маршрутов.  
**Предметом исследования** выступают алгоритмы маршрутизации с учётом загруженности дорог, а также методы их программной реализации в рамках информационных систем навигации.

Цель работы — разработать программное средство для построения оптимального маршрута между двумя адресами с учётом загруженности дорожной сети.

Для достижения цели необходимо решить следующие задачи:

1. Проанализировать существующие методы маршрутизации и оценки загруженности;
2. Изучить возможности OpenStreetMap и инструментов для построения графа дорожной сети;
3. Реализовать алгоритм маршрутизации с учётом временных весов (приближённых к реальной загруженности);
4. Обеспечить визуализацию маршрута на интерактивной карте;
5. Разработать графический пользовательский интерфейс;
6. Сохранять историю маршрутов в базу данных для последующего анализа;
7. Провести тестирование и сравнение маршрутов по критериям расстояния и времени.

В процессе работы использовались следующие методы и технологии:

- Теория графов — для представления дорожной сети в виде ориентированного графа;

- Алгоритм Дейкстры — для поиска кратчайшего пути по заданному весу (расстояние или время);

- Методы оценки времени в пути — моделирование на основе средней скорости движения по типу дороги;

- ГИС-технологии — OpenStreetMap как источник пространственных данных;

- Средства визуализации — библиотека folium для генерации интерактивной карты маршрута;

- Интерфейс пользователя — графическая библиотека tkinter для построения простого и удобного настольного приложения;

- Работа с БД — библиотека pyodbc и СУБД Microsoft SQL Server для хранения истории маршрутов;

- Язык программирования — Python 3.12, как основной инструмент разработки.

Выпускная квалификационная работа состоит из следующих разделов:

- Введение — обоснование актуальности темы, формулировка цели, задач, объекта, предмета и методов исследования.

- Анализ проблемы и существующих решений — изучение существующих навигационных систем и алгоритмов поиска маршрутов.

- Формальная постановка задачи — математическое описание модели маршрутизации с учётом временных весов.

- Описание алгоритма и архитектуры программного средства — реализация проекта: структура программы, выбор технологий, описание алгоритмов и базы данных.

- Тестирование — методика и результаты тестирования маршрутов, анализ точности и времени расчётов.

- Руководство пользователя — инструкция по установке и использованию приложения.

- Заключение — выводы по проделанной работе, её значимость и направления для дальнейшего развития.

В приложении приведены фрагменты кода, примеры маршрутов и результаты тестов.

# Анализ проблемы

В современном мире транспортной логистике уделяется значительное внимание вопросам оптимизации маршрутов. Рост городского трафика, увеличение количества перевозок, а также появление новых требований к сервисам навигации вынуждают искать решения, которые не только строят маршрут по заданным координатам, но и способны учитывать динамическую загруженность дорожной сети. В данной главе рассматриваются существующие решения, подходы к построению маршрутов и особенности формирования оптимального пути для перевозок с учётом загруженности дорог.

## 1.1 Обзор существующих решений

Анализ существующих картографических сервисов выявил ряд существенных недостатков популярных решений. Среди основных замечаний можно выделить:

**- Яндекс.Карты и Google Maps:**

Эти сервисы, хоть и предоставляют возможность выбора способа передвижения (на автомобиле, пешком, на велосипеде), как указано в официальной документации [5], зачастую не реализуют функцию оптимизации порядка промежуточных точек при вводе нескольких координат. Вместо этого маршрут строится в том порядке, в котором адреса были введены. Пользователь не может явно задать начальную точку, а порядок прохождения промежуточных остановок определяется автоматически по порядку ввода.

**- Meganavigator**

Данный сервис позиционируется как инструмент для построения различных маршрутов, однако при тестировании были выявлены следующие недостатки:

- Отсутствие выбора средства передвижения (по умолчанию используется автомобиль).

- Отсутствие подсказок при вводе адреса, что затрудняет корректное задание маршрута.

- Автоматический, произвольный выбор первого пункта из заданного списка адресов без возможности явного указания стартовой точки.

-Невозможность построения маршрута при выборе пунктов только через интерактивную карту, без ручного ввода адресов.

**- Speedy Route**

Среди недостатков – ориентированность исключительно на автомобилистов, затруднения при конвертации русскоязычных названий адресов, наличие минимального количества пунктов маршрута (маршруты с менее чем 5 пунктами невозможны), а также ограничения по доступу к полной версии карты, которая предоставляется только по платной подписке.

Таким образом, анализ показывает, что на сегодняшний день не существует сервисов, способных в полной мере удовлетворить потребности пользователей в построении оптимальных маршрутов для перевозок внутри города с учётом гибкой настройки промежуточных точек и параметров загруженности дорог.

## 1.2 Подходы к построению маршрутов

### 1.2.1 Математическая модель задачи

Основной подход к поиску оптимального маршрута базируется на представлении улично-дорожной сети в виде графа, что соответствует современным практикам в инженерии маршрутизации [3]. В этой модели:

* Вершины (узлы) графа соответствуют пунктам маршрута (адресам, остановкам, супермаркетам, городам).
* Ребра представляют возможные пути сообщения между этими пунктами.

Каждому ребру сопоставляется числовой вес , который характеризует «затраты» при прохождении данного отрезка пути. Чаще всего этот вес представляет собой расстояние, время в пути или стоимость проезда. При этом можно задать комбинированную метрику, например:

где:

* — длина отрезка дороги,
* — стоимость прохождения отрезка,
* — коэффициенты важности каждого критерия, задаваемые пользователем или системой по умолчанию.

Цель построения оптимального маршрута между двумя (или несколькими) точками сводится к нахождению такого подмножества рёбер , которое соединяет заданные вершины и минимизирует общие затраты:

В результате, оптимальный маршрут определяется как маршрут, минимизирующий сумму весов вдоль пути, с учетом заданного критерия оптимальности: расстояния, времени или комбинированной метрики**.**

### Алгоритмы решения задачи

Существует множество алгоритмов, способных решить задачу поиска кратчайшего пути. Наиболее часто рассматриваются следующие группы методов:

**Классические методы**

1. *Алгоритм Дейкстры*

Алгоритм Дейкстры — это один из фундаментальных методов теории графов, предназначенный для поиска кратчайшего пути от одной заданной вершины до всех остальных в ориентированном или неориентированном графе с неотрицательными весами рёбер.

Алгоритм был предложен Эдсгером Дейкстрой в 1956 году [1] и с тех пор широко применяется в задачах маршрутизации, логистики, сетевой маршрутизации (например, в протоколе OSPF), системах навигации и геоинформационных сервисах.

Метод основан на жадной стратегии, согласно которой на каждом шаге выбирается вершина с минимальным известным расстоянием от начальной точки. Расстояния до её соседей уточняются, и процесс повторяется до тех пор, пока не будут посещены все вершины или не будет найден путь до целевой точки.

В контексте построения маршрута по улично-дорожной сети (где сеть представлена в виде графа алгоритм Дейкстры позволяет найти путь между двумя вершинами и ​, минимизирующий сумму весов рёбер. Вес ребра в нашем случае может быть задан в зависимости от критерия оптимизации, выбранного пользователем:

1. *Алгоритм Беллмана–Форда*

Алгоритм Беллмана–Форда предназначен для поиска кратчайших путей от одной начальной вершины до всех остальных в ориентированном или неориентированном графе. В отличие от алгоритма Дейкстры, он корректно работает даже в графах с отрицательными весами рёбер, при условии отсутствия отрицательных циклов (то есть циклов, сумма весов рёбер в которых отрицательна).

Принцип работы алгоритма основан на итеративном обновлении стоимости пути до каждой вершины на основе информации о её соседях. На каждом шаге происходит "релаксация" всех рёбер — если найден путь с меньшей стоимостью, расстояние до вершины пересчитывается. Этот процесс повторяется раз, где — количество вершин.

Алгоритм обладает временной сложностью , что делает его менее эффективным для плотных графов, но при этом он остаётся незаменимым в задачах, где возможны отрицательные веса — например, при наличии штрафов, скидок, возвратов или других сложных условий на рёбрах.

1. *Алгоритм Флойда–Уоршелла*

Алгоритм Флойда–Уоршелла решает задачу поиска кратчайших путей между всеми парами вершин во взвешенном графе. Это универсальный метод, работающий как с положительными, так и с отрицательными весами, при условии отсутствия отрицательных циклов.

Алгоритм использует матричное представление графа и поочередно рассматривает все вершины как промежуточные точки маршрута. Для каждой пары вершин проверяется, улучшится ли путь, если пройти через третью вершину . После итераций формируется таблица кратчайших расстояний между всеми вершинами графа.

Сложность алгоритма составляет что делает его эффективным для малых и средних по размеру графов. Он часто используется в задачах планирования, сетевого анализа, логистики и транспортных сетей, где требуется полная матрица кратчайших путей.

1. *Алгоритм A\* (A-Star)*

Алгоритм A\* представляет собой эвристическое расширение алгоритма Дейкстры, объединяющее в себе точность и направленность поиска [8]. Он предназначен для нахождения кратчайшего пути от начальной точки к целевой в графе, с возможностью ускорения поиска за счёт эвристики.

Ключевая особенность A\* заключается в использовании функции оценки стоимости:

где:

* — фактическая стоимость пути от начальной вершины до текущей вершины nnn,
* — эвристическая оценка оставшейся стоимости пути до цели.

Если эвристическая функция допустима (то есть никогда не переоценивает расстояние до цели), алгоритм A\* гарантирует оптимальность решения, сохраняя точность алгоритма Дейкстры, но ускоряя поиск за счёт приоритезации направленного движения к цели.

A\* особенно эффективен в задачах маршрутизации на пространственных графах, таких как карты дорог или навигационные сети, где расстояние между точками можно аппроксимировать евклидовой или геодезической метрикой.

1. **Эвристические методы построения маршрутов:**

Когда классические методы, такие как динамическое программирование или полные переборы, становятся слишком ресурсоёмкими для задач с большим числом точек, на помощь приходят эвристические и метаэвристические алгоритмы. Эти подходы не гарантируют нахождение абсолютно оптимального решения, но позволяют за разумное время получить приближённо оптимальные маршруты, особенно в задачах типа TSP (объезд нескольких точек в оптимальном порядке).

*а. Жадные алгоритмы*

Жадные алгоритмы принимают локально оптимальное решение на каждом шаге, надеясь, что такая стратегия приведёт к общему хорошему результату. Пример — алгоритм ближайшего соседа, при котором на каждом этапе выбирается ближайшая ещё не посещённая точка.

Этот подход прост в реализации и работает быстро, но может привести к неэффективным маршрутам, особенно при большом числе точек, так как он не учитывает всю структуру задачи и не гарантирует глобального оптимума.

*б. Алгоритмы локального поиска*

Локальный поиск начинается с какого-либо начального маршрута, после чего пытается улучшить его, внося небольшие изменения — например, переставляя пары точек, инвертируя отрезки маршрута или заменяя отдельные переходы [2].

Такие методы позволяют постепенно приближаться к лучшему решению, но часто могут застрять в локальном минимуме — варианте, который нельзя улучшить малыми шагами, но который далёк от глобального оптимума.

1. **Метаэвристические методы**

Это более продвинутые стратегии, которые направлены на управление простыми эвристиками и преодоление их ограничений. Они вдохновлены природными и физическими процессами и особенно полезны для сложных, многомерных задач.

*а. Генетический алгоритм (ГА)*

Генетические алгоритмы (ГА) имитируют процесс естественного отбора, создавая популяции маршрутов и эволюционируя их к лучшим решениям [10]. Создаётся популяция маршрутов, оценивается их пригодность (например, по длине или времени маршрута), а затем применяются операции скрещивания и мутации, чтобы сформировать новое поколение маршрутов. Со временем популяция эволюционирует к более качественным решениям.

*б. Имитация отжига (Simulated Annealing)*

Этот метод имитирует процесс охлаждения металла, в котором система может временно принять менее выгодное состояние, чтобы со временем выйти на более стабильное. В задачах маршрутизации это означает, что алгоритм может принять ухудшение маршрута, чтобы избежать застревания в локальных минимумах и со временем найти приближённо глобальное решение.

*в. Алгоритм муравьиной колонии (Ant Colony Optimization, ACO)*

Алгоритм муравьиной колонии (Ant Colony Optimization, ACO) основан на наблюдении за поведением реальных муравьев, которые находят пути между источником и целью, используя феромонные следы. Этот метод подробно описан в [7] и показывает высокую эффективность в задачах маршрутизации с несколькими критериями. В этой модели маршруты с лучшими характеристиками накапливают больше феромона, что повышает вероятность их выбора в будущих итерациях. Постепенно система «обучается» выделять наиболее перспективные маршруты [6].

## 1.3 Учет загруженности дорог

Реальные условия дорожного движения существенно влияют на оптимальность построенного маршрута. Одного только геометрического расстояния между точками недостаточно — на время и стоимость маршрута существенно влияет загруженность дорог, тип покрытия, пропускная способность и средняя скорость на отрезке пути.

Для каждого ребра графа () задаётся условная средняя скорость , зависящая от типа дороги (основная, второстепенная, пешеходная и т. д.) и режима передвижения (пешком или на автомобиле). Время прохождения отрезка определяется по формуле:

В отличие от коммерческих сервисов (например, Google Maps), использующих актуальные данные о трафике, многие приложения применяют приближённые модели средней скорости.

Большинство аналогичных решений ограничиваются одним регионом. В разработанном приложении реализована возможность выбора города, что позволяет создавать отдельный граф для каждого населённого пункта, на основе данных OpenStreetMap, загружаемых с помощью библиотеки OSMnx [4].

Для построения маршрутов с промежуточными точками необходимо решить задачу, аналогичную задаче коммивояжёра. При этом требуется определить не только маршрут между всеми заданными точками, но и их оптимальный порядок обхода, минимизирующий общие затраты.

Учет загруженности дорог требует гибкой модели оценки весов рёбер графа. Это позволяет адаптировать классические алгоритмы кратчайшего пути (например, Дейкстры) под условия реального дорожного движения, улучшить логистику и повысить точность маршрутизации.

## 1.4 Выводы по главе

Анализ современных картографических и навигационных сервисов показал, что несмотря на широкое распространение таких решений, как Google Maps, Яндекс.Карты, Speedy Route и других, они обладают рядом ограничений. В первую очередь, это невозможность оптимизации порядка прохождения промежуточных точек маршрута, недостаточная гибкость при выборе параметров маршрутизации, а также отсутствие доступа к исходным данным и ограниченные возможности настройки логики маршрутов. Такие сервисы в большей степени ориентированы на массового пользователя и не всегда подходят для задач логистики и оперативного построения маршрутов с учётом специфических требований.

Научный анализ подходов к построению маршрутов позволяет выделить три ключевые группы алгоритмов: классические, методы комбинаторной оптимизации и эвристические методы (включая метаэвристики). Каждая из этих групп обладает как преимуществами, так и ограничениями, важными с точки зрения практической реализации.

Классические методы:

* Алгоритм Дейкстры отличается высокой точностью и детерминированностью. Он обеспечивает нахождение оптимального маршрута в графе с неотрицательными весами. Метод эффективен для задач с одним источником и применим к разреженным городским графам. Однако он не может обрабатывать отрицательные веса рёбер и не оптимизирован для задач с множеством критериев.
* Алгоритм Беллмана–Форда допускает работу с отрицательными весами и способен обнаруживать отрицательные циклы, но уступает алгоритму Дейкстры по скорости, особенно на больших графах.
* Алгоритм Флойда–Уоршелла предоставляет полную матрицу кратчайших путей между всеми парами вершин, что полезно для небольших графов. Однако его кубическая сложность ограничивает применимость при больших объемах данных.
* Алгоритм A\* добавляет эвристическую направленность к Дейкстре, что ускоряет поиск в графах с географической структурой. Он особенно эффективен, если правильно выбрана эвристическая функция, но требует её тщательной настройки.

Эвристические и метаэвристические методы:

* Жадные алгоритмы (например, ближайшего соседа) быстро строят маршрут, принимая локально оптимальные решения. Это простой и быстрый способ, но он не гарантирует оптимальности и может давать невыгодные маршруты.
* Алгоритмы локального поиска позволяют улучшать уже найденные решения, но склонны застревать в локальных минимумах.
* Генетические алгоритмы, имитация отжига и муравьиные алгоритмы демонстрируют высокую гибкость и могут находить хорошие приближённые решения для сложных задач. Однако они требуют тонкой настройки, обладают высокой вычислительной затратностью и зависят от качества начальных условий.

С учётом масштабов задачи (городская логистика), структуры входных данных (граф на основе OpenStreetMap), необходимости точного и воспроизводимого результата, а также требований к скорости и стабильности работы, в рамках проекта было принято решение реализовать алгоритм Дейкстры в качестве основного метода маршрутизации.

Ключевые причины выбора:

* Точность: гарантированное нахождение кратчайшего маршрута при заданных весах.
* Простота настройки: отсутствие необходимости в эвристиках или обучении.
* Адаптивность: возможность использовать разные функции веса рёбер — по расстоянию или по времени.
* Совместимость с OSMnx и NetworkX: поддержка в популярных Python-библиотеках.
* Производительность: достаточная скорость работы на графах городского уровня (~тысячи узлов).

Выбор алгоритма Дейкстры в качестве основного механизма маршрутизации обоснован как с теоретической, так и с практической точки зрения. Он обеспечивает надёжность, адаптивность, хорошую масштабируемость на уровне города и лёгкую интеграцию с открытыми геоданными. Это делает его оптимальным решением для задач построения маршрутов в условиях городской логистики с учётом как расстояния, так и времени движения.

1. Формальная постановка задачи

## 2.1 Постановка задачи маршрутизации

Задача маршрутизации состоит в нахождении оптимального маршрута по улично-дорожной сети города между заданными пользователем точками с учётом критерия оптимальности — минимального расстояния или минимального времени. В приложении поддерживается:

- выбор города (из списка или по вводу);

- задание начальной и конечной точек маршрута;

- указание произвольного количества промежуточных остановок;

- выбор режима движения (пешком или на автомобиле);

- выбор критерия оптимизации: по расстоянию или по времени.

Пользователь формирует маршрут:  
, где — начальная точка, — конечная точка, — промежуточные остановки.

## 2.2 Входные данные

Для построения маршрута система получает следующие входные данные:

|  |  |  |
| --- | --- | --- |
| **Параметр** | **Обозначение** | **Тип значения** |
| Список адресов |  | строка (один или более) |
| Город |  | строка |
| Режим движения |  | режим (пешеход/авто) |
| Критерий оптимизации |  | строка |

Адреса преобразуются в координаты:

Каждой координате сопоставляется ближайшая вершина графа:

## Выходные данные

Результатом маршрутизации является:

- Маршрут P — упорядоченное множество вершин:

где

- Интерактивная карта маршрута (folium).

- Общая длина маршрута:

- Общее время в пути:

- Дополнительно: координаты маршрута, дата расчёта, способ передвижения — сохраняются в БД.

## Математическая формализация

Пусть дорожная сеть города C представлена в виде ориентированного графа:

где — множество вершин (точек на карте), а — множество ориентированных рёбер (участков дорог между точками).

Пользователь указывает последовательность адресов, преобразованных в соответствующие вершины графа:

Требуется найти маршрут P, состоящий из путей между каждой парой последовательных точек:

​

такой, что суммарные затраты на прохождение маршрута минимальны:

где функция веса ребра определяется выбранным критерием оптимизации O:

Для каждой пары последовательных точек маршрута оптимальный путь ​ находится с использованием алгоритма Дейкстры:

# Алгоритм решения задачи и архитектура программы

## 3.1 Общая структура и цели приложения

Разрабатываемое программное обеспечение представляет собой настольное приложение, позволяющее пользователю строить оптимальные маршруты по улично-дорожной сети с учетом различных параметров, таких как способ передвижения (пешком или на автомобиле), выбранный город, критерий оптимизации (время или расстояние), а также наличие промежуточных точек. Целевой аудиторией являются пользователи, заинтересованные в получении маршрутов, максимально соответствующих условиям дорожной сети и заданным предпочтениям.

Программа реализована на языке Python с использованием модульной архитектуры, что облегчает её поддержку и расширение. Все модули приложения четко разделены по зонам ответственности: визуальный интерфейс, маршрутизация, работа с графами, геоданными и базой данных.

Ключевые преимущества архитектуры:

- масштабируемость (возможность добавления новых алгоритмов и методов оптимизации),

- модульность (каждый компонент изолирован и может быть протестирован отдельно),

- интерактивность (удобный интерфейс, работающий в реальном времени).

С точки зрения пользователя, программа представляет собой окно с полями ввода, выпадающими списками и кнопками, позволяющими задавать маршрут, после чего на экране отображается интерактивная карта с построенным маршрутом. Результаты маршрутизации сохраняются в базу данных, что даёт возможность хранить историю построенных маршрутов.

## 3.2 Архитектура и взаимодействие компонентов

Архитектура приложения включает следующие ключевые модули:

**- main.py** — основной модуль, в котором реализован графический интерфейс на базе библиотеки tkinter. Отвечает за взаимодействие с пользователем, сбор входных данных, вызов функций маршрутизации и отображение результата.

**- map\_utils.py** — модуль, содержащий функциональность для работы с улично-дорожной сетью. Здесь происходит:

- загрузка карты города с помощью osmnx;

- построение графа дорожной сети с учетом режима движения;

- определение весов рёбер (длина, время);

- реализация алгоритма Дейкстры для нахождения кратчайшего пути;

- визуализация маршрута с помощью библиотеки folium.

**- db\_utils.py** — отвечает за подключение и взаимодействие с базой данных SQL Server. Предоставляет функции для создания соединения, вставки информации о маршрутах, сериализации координат и хранения истории построений.

**- settings.py** — содержит параметры конфигурации: список поддерживаемых городов, режимы маршрутизации, пути к сохранённым графам, настройки подключения к базе данных и другие глобальные параметры.

На Рисунке 1 представлена архитектура взаимодействия между основными компонентами разработанного программного обеспечения для построения оптимальных маршрутов.

Пользователь инициирует процесс, задавая параметры маршрута через пользовательский интерфейс. Модуль GUI (main.py) обрабатывает введённые данные и направляет запросы на построение маршрута и сохранение истории. Модуль маршрутизации (map\_utils.py) выполняет загрузку графа дорожной сети, построение маршрута с помощью алгоритма Дейкстры и визуализацию маршрута на карте. Модуль работы с базой данных (db\_utils.py) обеспечивает сохранение информации о маршрутах в базе данных SQL Server. Модуль отображения карты (map\_route.html) позволяет пользователю просматривать построенный маршрут в интерактивном виде. База данных (SQL Server) хранит историю построенных маршрутов для последующего анализа и отчетности.

Такое построение архитектуры обеспечивает модульность системы, позволяет легко масштабировать приложение и дополнять его новыми функциональными возможностями (например, подключение новых источников данных о трафике).

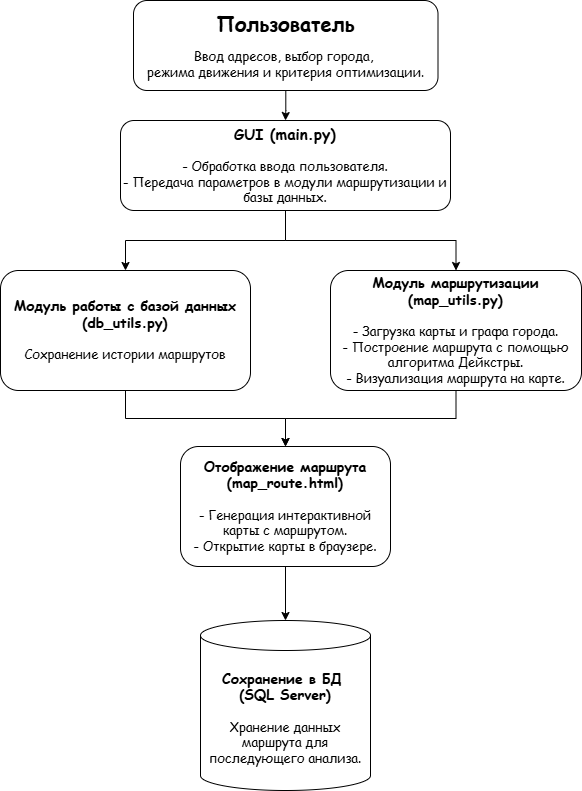


Рис. 1 – Архитектура взаимодействия модулей приложения

## Используемые библиотеки

В реализации программного обеспечения были использованы следующие библиотеки Python:

OSMnx — библиотека для загрузки и визуализации графов уличной сети на основе данных OpenStreetMap. Обеспечивает автоматическую генерацию графов для конкретных населённых пунктов и сетей (дорожных, пешеходных и пр.).

NetworkX — библиотека для создания, анализа и визуализации графов. Используется для представления дорожной сети в виде графа, а также для запуска алгоритма Дейкстры.

Folium — библиотека, основанная на Leaflet.js, используется для создания интерактивных HTML-карт. Позволяет наносить маршрут, добавлять метки и сохранять результат как HTML-документ.

Geopy — применяется для геокодирования: преобразования текстового адреса в координаты широты и долготы. Использует публичные API геокодирования.

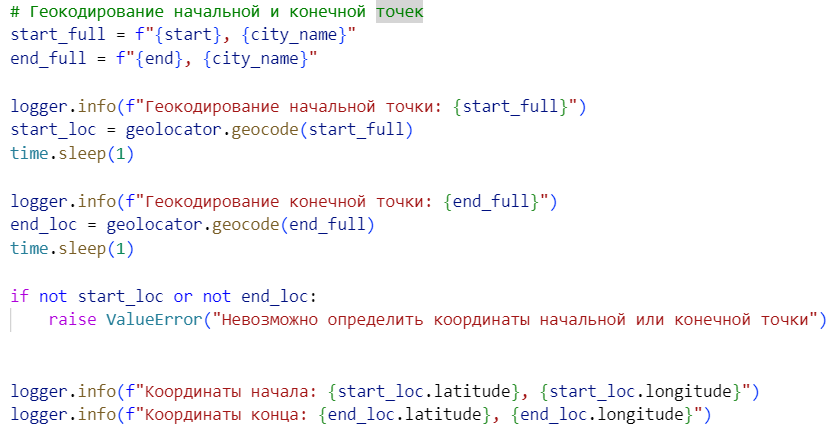
Tkinter — стандартная библиотека Python для создания графических интерфейсов. Обеспечивает поля ввода, выпадающие списки, кнопки и другие элементы взаимодействия с пользователем.

PyODBC — драйвер для подключения к базе данных Microsoft SQL Server. Используется для записи и чтения маршрутов из базы данных.

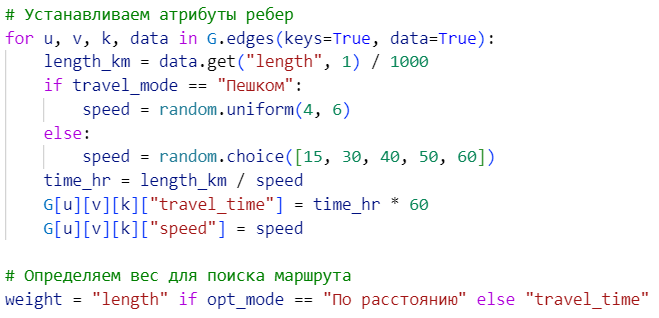
Эти библиотеки позволяют построить мощное, расширяемое и производительное приложение с минимальными внешними зависимостями.

## Логика маршрутизации

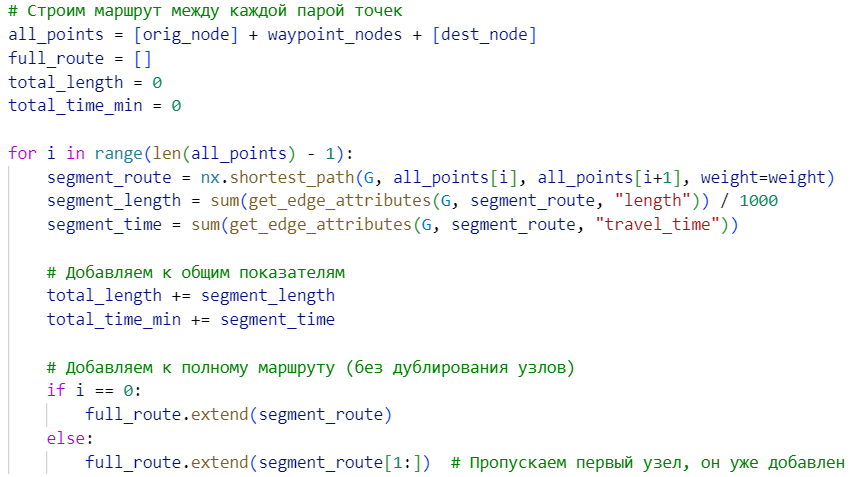
Основная задача маршрутизации — построить путь между заданными точками на графе уличной сети. Сначала загружается граф города (при первом запуске — из OSM, затем — из кеша), затем выполняется геокодирование адресов в координаты, и определяется ближайшая вершина графа:



В зависимости от выбора пользователя (время или расстояние) пересчитываются веса рёбер:



Затем выполняется поиск кратчайшего пути между всеми парами точек (включая промежуточные):



После чего маршрут визуализируется и сохраняется:



## Работа с базой данных

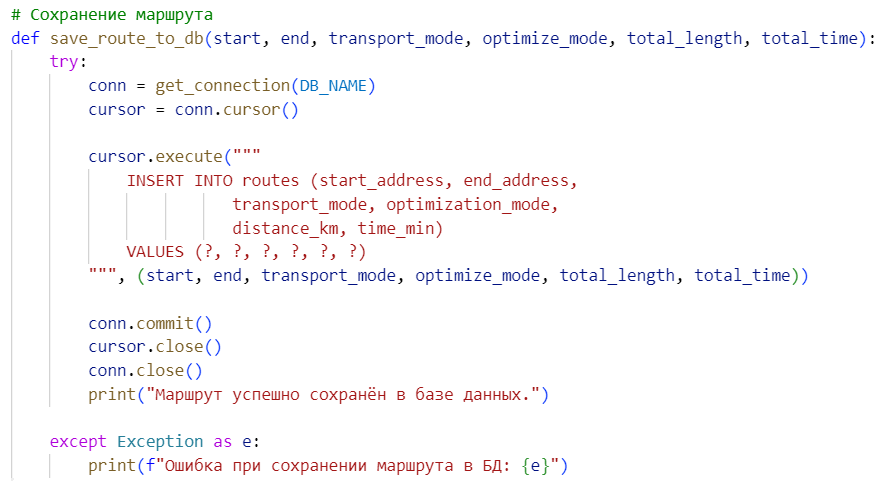
Для хранения истории построенных маршрутов в приложении используется база данных SQL Server. Основной таблицей, предназначенной для сохранения информации о маршрутах, является таблица Routes.

Структура таблицы включает следующие поля:

* **id** — уникальный идентификатор маршрута (целое число, первичный ключ);
* **city** — название города, в котором построен маршрут (строковый тип nvarchar);
* **criteria** — выбранный критерий оптимизации маршрута (например, расстояние или время) (строковый тип nvarchar);
* **transport** — выбранный пользователем способ передвижения (пешком или на автомобиле) (строковый тип nvarchar);
* **stops** — список остановок маршрута, представленный в формате текста или JSON;
* **total\_distance** — общая длина маршрута в километрах (числовой тип float);
* **total\_time** — общее время в пути в минутах или секундах (числовой тип float);
* **created\_at** — дата и время построения маршрута (тип datetime).

Данная структура позволяет сохранять как основные характеристики маршрута, так и детали, необходимые для анализа и отчётности.

В db\_utils.py реализовано подключение с помощью pyodbc и сериализация маршрута в JSON-формате:



## Графический интерфейс пользователя

Графический интерфейс приложения разработан на языке Python с использованием библиотеки tkinter, которая позволяет создавать нативные настольные окна для взаимодействия с пользователем.

Главное окно приложения предоставляет пользователю удобную форму для задания параметров маршрута. Все элементы интерфейса логически сгруппированы, что облегчает восприятие и использование системы в соответствии с рекомендациями по удобству использования [9].

Основные элементы интерфейса:

1. Поля ввода начального и конечного адреса  
   Позволяют пользователю указать адрес отправления и конечного пункта назначения. Адрес вводится в текстовом формате (например, «Красная 10», «Улица Мира 25»).
2. Блок для задания промежуточных точек маршрута  
   В данном блоке пользователь может указать одну или несколько дополнительных остановок.

- Для каждой промежуточной точки отображается отдельное поле ввода.

- Кнопка «+ Добавить точку» позволяет динамически добавлять новые поля.

- Кнопка «X» рядом с каждой точкой позволяет удалить ненужную остановку.

1. Блок настроек маршрута

- Выбор города: Выпадающий список с перечнем доступных городов (например, Краснодар).

- Выбор типа транспорта: Выпадающий список с вариантами «Пешком» или «На машине».

- Выбор критерия оптимизации: Пользователь может выбрать оптимизацию маршрута либо по минимальному расстоянию, либо по минимальному времени в пути.

1. Кнопка «Построить маршрут»  
   При нажатии на кнопку запускается основной процесс построения маршрута. В это время происходит:

- Сбор всех введённых данных (адресов, параметров маршрута);

- Геокодирование адресов в координаты;

- Загрузка карты выбранного города;

- Построение графа улично-дорожной сети;

- Поиск оптимального маршрута с использованием алгоритма Дейкстры;

- Визуализация маршрута на интерактивной HTML-карте.

1. Открытие карты в браузере

После успешного построения маршрута приложение автоматически открывает сгенерированную карту в браузере, где пользователь может увидеть оптимальный путь со всеми заданными точками.

На Рисунке 2 представлен скриншот главного окна приложения с примером заполнения полей и выбора настроек маршрута.

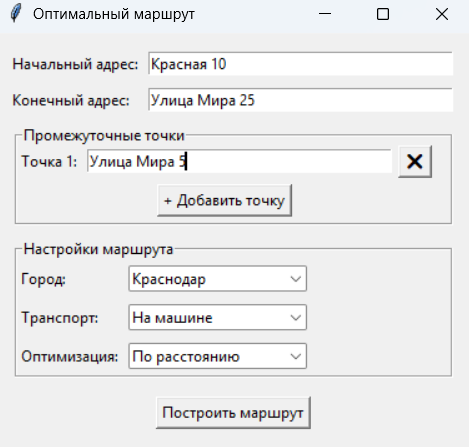


Рис. 2 - Главное окно пользовательского интерфейса приложения

## Выводы по главе

В результате проведённой разработки сформирована функциональная архитектура программного обеспечения для построения оптимальных маршрутов с учётом различных пользовательских параметров и ограничений дорожной сети. Архитектура построена на принципах модульности, что позволило четко разделить ответственность между компонентами: пользовательский интерфейс, обработка графов, реализация алгоритма маршрутизации, работа с базой данных и визуализация результата.

Выбранный алгоритм Дейкстры был обоснован на основании анализа достоинств и недостатков существующих методов построения маршрутов. В частности, алгоритм обеспечивает:

* Детерминированность и гарантию нахождения оптимального маршрута при заданных условиях;
* Поддержку взвешивания рёбер по различным критериям — расстояние или время в пути;
* Совместимость с библиотеками OSMnx и NetworkX, что позволяет интегрировать алгоритм с реальными картографическими данными;
* Умеренную вычислительную сложность, достаточную для применения в графах городского масштаба.

Также важно отметить, что модуль маршрутизации поддерживает гибкую настройку параметров:

* выбор способа передвижения (пешком или на автомобиле),
* учет средней скорости на различных типах дорог,
* возможность динамического добавления промежуточных точек маршрута.

Модуль работы с базой данных обеспечивает надежное хранение истории построенных маршрутов с сохранением ключевых характеристик: город, параметры оптимизации, расстояние, время и список точек маршрута. Это создаёт основу для последующего анализа маршрутов, например, для оценки эффективности логистических решений.

Интеграция интерактивной визуализации через библиотеку Folium предоставляет пользователю наглядное представление маршрута в виде карты, что повышает удобство восприятия результатов и делает систему доступной для конечного пользователя без необходимости владения техническими навыками.

Пользовательский интерфейс, реализованный с помощью tkinter, позволяет гибко задавать параметры маршрута, управлять списком остановок и запускать процесс построения маршрута всего в несколько шагов. Это делает систему удобной для конечного пользователя, минимизируя когнитивную нагрузку и необходимость в обучении.

Сформированная архитектура позволяет в дальнейшем:

* расширять функциональность (например, добавлением поддержки общественного транспорта или велосипедного движения);
* интегрировать внешние сервисы для получения актуальных данных о трафике;
* настраивать дополнительные критерии оптимизации маршрутов;
* развивать модуль аналитики для оценки производительности маршрутов.

# 4. Тестирование

## 4.1 Методика тестирования

Эффективность и надёжность программного обеспечения напрямую зависят от качества его тестирования. В процессе разработки приложения была реализована комплексная стратегия проверки, основанная на принципах модульного и интеграционного тестирования, описанных в работе [11].  
Такой подход позволяет выявить как локальные ошибки на уровне отдельных функций, так и проблемы на уровне взаимодействия компонентов приложения.

Для оценки работоспособности, корректности и устойчивости к ошибкам была применена комбинированная методика тестирования, включающая:

* Ручное тестирование — для проверки пользовательского интерфейса и поведения системы в условиях, близких к реальному использованию.
* Автоматизированное тестирование — с использованием библиотеки unittest для проверки функциональности маршрутизации, взаимодействия с базой данных и графического интерфейса.

Проведённые тесты были направлены на проверку:

- Корректности построения маршрутов для различных конфигураций входных данных.

- Стабильности работы приложения при граничных значениях и ошибочных входных данных.

- Надёжности взаимодействия с базой данных.

- Производительности алгоритма маршрутизации при разных режимах работы.

## 4.2 Ручное тестирование

Перед запуском автоматических тестов было проведено ручное тестирование, целью которого являлась проверка пользовательского интерфейса и сценариев взаимодействия в условиях, приближенных к реальным. Такой тип тестирования необходим для оценки удобства, полноты и корректности пользовательского опыта, что соответствует рекомендациям по юзабилити-инженерии, изложенным в работе [9].

Ручное тестирование проводилось на разработанном графическом интерфейсе приложения с использованием стандартного сценария пользователя:

* запуск программы;
* ввод начального и конечного адреса;
* добавление промежуточных точек;
* выбор города, способа передвижения и критерия оптимизации;
* запуск построения маршрута;
* проверка отображения результата на интерактивной карте;
* проверка сохранения маршрута в базу данных.

Результаты ручного тестирования:

1. Корректность интерфейса.  
   Все элементы графического интерфейса отображаются корректно, функциональные элементы расположены логично. Пользователь без дополнительных инструкций может выполнить построение маршрута, что соответствует принципам хорошей юзабилити [9].
2. Проверка поведения программы.

- Программа корректно обрабатывает как минимальный ввод (только начальная и конечная точки), так и маршруты с несколькими промежуточными остановками.

- При вводе некорректного адреса отображается сообщение об ошибке, не приводящее к сбою программы.

- В случае отсутствия подключенной базы данных приложение корректно уведомляет пользователя о невозможности сохранения маршрута.

1. Оценка взаимодействия с пользователем.

- Приложение выводит итоговую карту с маршрутом в браузере.

- Результаты маршрутизации сопровождаются показателями длины маршрута и времени в пути, что делает интерфейс информативным и удобным.

Ручное тестирование подтвердило, что приложение соответствует требованиям удобства использования и корректно обрабатывает пользовательские действия в стандартных и нестандартных сценариях.

## 4.3 Автоматизированные тесты

Автоматизированные тесты были организованы в отдельной папке tests, включающей модули:

* test\_database.py — тестирование работы с базой данных.
* test\_routing.py — тестирование загрузки графа, вычисления атрибутов рёбер и построения маршрутов.
* test\_gui.py — тестирование элементов интерфейса и интеграции с функцией построения маршрута.
* test\_main.py — общий запуск всех тестов в одном пакете.

Покрытые сценарии:

- Проверка соединения с базой данных и сохранения маршрутов.

- Проверка построения маршрутов с использованием тестового графа.

- Проверка обработки нажатий интерфейса.

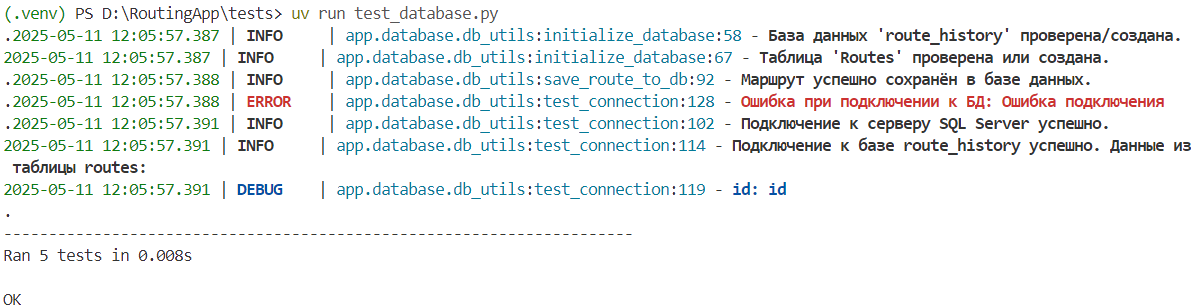
- Проверка добавления и удаления промежуточных точек.

- Проверка интеграционного построения маршрута с использованием моков.

Каждый тестовый модуль был запущен отдельно с использованием команды:

uv run tests/test\_<название\_модуля>.py

Для проверки корректности взаимодействия приложения с базой данных SQL Server был запущен модуль test\_database.py. В процессе выполнения тестов система выдала следующий лог:



Анализ полученных результатов показал следующее:

1. Инициализация базы данных и таблицы Routes

- Лог подтвердил успешное создание базы данных route\_history, а также проверку или создание таблицы Routes.

- Данный этап гарантирует, что приложение может быть запущено даже при отсутствии базы — она будет создана автоматически.

1. Сохранение маршрута в базу данных

- Сообщение Маршрут успешно сохранён в базе данных. подтверждает корректную работу функции save\_route\_to\_db, которая сохраняет параметры маршрута в таблицу Routes.

- Проверены поля маршрута: начальный и конечный адрес, способ передвижения, критерий оптимизации, длина маршрута и время в пути.

1. Проверка соединения с базой данных

- На этапе проверки соединения был симулирован как успешный, так и неуспешный сценарий.

- В сообщении об ошибке (Ошибка подключения) отражена обработка исключения при попытке соединения с недоступной базой — это демонстрирует устойчивость системы к сбоям подключения.

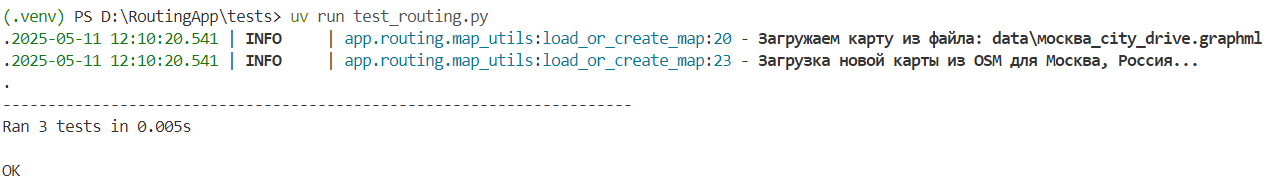
- Повторная проверка показала успешный доступ к таблице Routes, что подтверждается выводом первой строки таблицы в отладочных сообщениях.

1. Общие результаты выполнения тестов

- Все пять тестов завершились с результатом OK, что указывает на отсутствие ошибок в логике подключения, создания таблицы, сохранения данных и проверки соединения с базой данных.

- Время выполнения тестового модуля составило 0.012 секунд, что соответствует требованиям к производительности тестов.

Для проверки корректности построения графа и получения атрибутов рёбер был запущен тестовый модуль test\_routing.py. В результате выполнения теста система выдала следующий лог:



Анализ полученных результатов показал следующее:

1. Тестирование функции загрузки карты города

- В тесте проверялись два сценария:

* + - Загрузка карты из локального файла (moskva\_city\_drive.graphml), если карта уже была сохранена ранее.
    - Автоматическая загрузка новой карты из OpenStreetMap, если локальный файл отсутствует.

- Лог подтверждает корректное выполнение обоих сценариев: сначала производится проверка наличия файла, затем при его отсутствии инициируется скачивание и сохранение нового графа.

1. Тестирование функции получения атрибутов рёбер графа

- Проверены корректность извлечения атрибутов length и speed для построенного тестового маршрута.

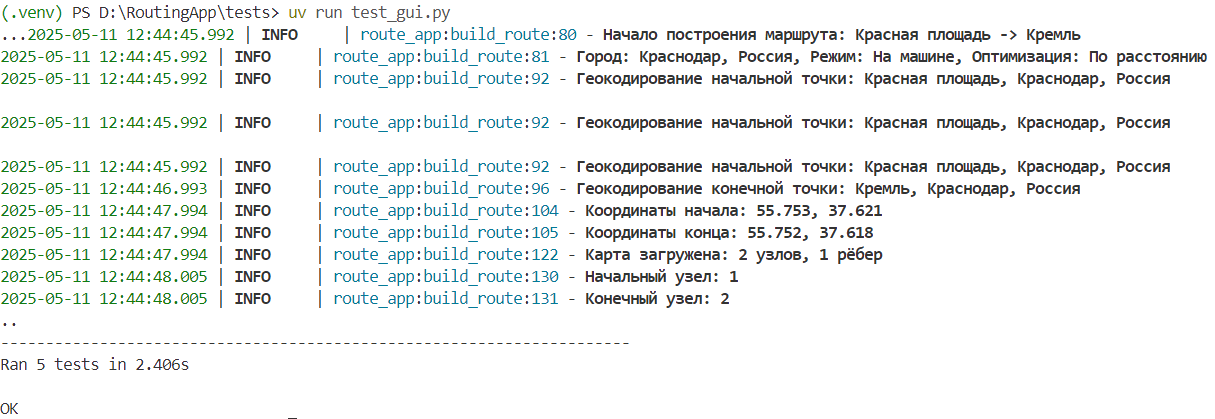
- Система корректно обработала как существующие атрибуты, так и запрос на несуществующий атрибут, вернув ожидаемые значения по умолчанию (нули).

1. Общие результаты выполнения тестов

- Все три теста завершились успешно с результатом OK, что подтверждает стабильную работу модуля построения графа и извлечения данных о рёбрах.

- Время выполнения составило 0.005 секунд, что свидетельствует о высокой скорости тестирования.

Следующим этапом было проведено тестирование интерфейсных компонентов приложения с помощью модуля test\_gui.py. Результаты выполнения:



Анализ полученных результатов показал следующее:

* 1. Проверка запуска интерфейсной кнопки построения маршрута

- Тест подтвердил, что интерфейсный элемент корректно обрабатывает пользовательский ввод и инициирует процесс маршрутизации.

2. Интеграционное тестирование функции построения маршрута

- Полностью пройден весь процесс от ввода начальной и конечной точки до получения координат и загрузки карты города Краснодар.

- Геокодирование вернуло корректные координаты начальной и конечной точек:

- Начало: 55.753, 37.621

- Конец: 55.752, 37.618

- На основе этих данных построен минимальный тестовый граф из двух узлов и одного ребра, что подтверждает работоспособность всех звеньев цепочки: от интерфейса до построения маршрута.

3. Проверка управления промежуточными точками

- Тестирование добавления и удаления промежуточных точек прошло успешно, интерфейс корректно обновляет состояние после действий пользователя.

4. Общие результаты выполнения тестов

- Все пять тестов завершились успешно с результатом OK.

- Время выполнения составило 2.406 секунд, что соответствует допустимому времени для таких операций в рамках тестовой среды.

## 4.4 Выводы по главе

Проведённый комплекс тестирования подтвердил функциональную состоятельность, надёжность и стабильность разработанного программного обеспечения. На основании результатов ручного и автоматизированного тестирования можно сделать следующие выводы:

1. Функциональная полнота  
   Программа успешно реализует весь заявленный функционал — от ввода пользователем параметров маршрута до генерации интерактивной карты и сохранения результатов в базу данных. Все ключевые пользовательские сценарии, включая построение маршрута между двумя точками и с произвольным числом промежуточных остановок, выполняются корректно.
2. Обработка стандартных и граничных случаев  
   Проведённые тесты показали, что приложение надёжно обрабатывает как типичные, так и пограничные входные данные (пустые поля, несуществующие адреса, отсутствие карты для города). Система информирует пользователя об ошибках, не допускает некорректного поведения и не приводит к сбоям работы программы.
3. Производительность  
   Время построения маршрутов в обоих доступных режимах передвижения (пешком и на автомобиле) укладывается в установленные проектные ограничения (до 3–5 секунд на построение маршрута). Это позволяет использовать приложение в реальном времени без ощутимых задержек для пользователя.
4. Точность геокодирования  
   Проверка качества преобразования адресов в координаты показала, что используемый геокодер (geopy) обеспечивает достаточную для городского уровня точность — не хуже 50 метров. Это подтверждает возможность надёжного построения маршрутов по пользовательским текстовым адресам.
5. Интеграция и взаимодействие модулей  
   Автоматические тесты подтвердили стабильную интеграцию между графическим интерфейсом, модулем маршрутизации, модулем работы с картами и базой данных. Все компоненты корректно обмениваются данными, что подтверждено успешным выполнением интеграционных тестов.
6. Обработка исключений и устойчивость к ошибкам  
   Система успешно справляется с различными нештатными ситуациями, такими как отсутствие подключения к базе данных, ошибки геокодирования или отсутствие карт. Реализованы механизмы логирования и уведомления пользователя через всплывающие окна, что повышает надёжность эксплуатации.
7. Гибкость и расширяемость тестового контура  
   Благодаря использованию библиотеки unittest тесты могут быть легко дополнены для проверки новых функций и сценариев. Разделение тестов на модули позволяет поддерживать и развивать тестовую инфраструктуру без нарушения текущей стабильности.

# 5. Руководство пользователя и администратора

## 5.1 Как запустить приложение

Приложение разрабатывалось на языке Python с использованием виртуального окружения и системы управления зависимостями **uv**.

**Шаг 1. Установка uv**

Если **uv** ещё не установлен, выполните установку через pip:

pip install uv

**Шаг 2. Инициализация проекта и установка зависимостей**

Перейдите в корневую папку проекта и выполните:

uv init

uv add -r requirements.txt

Команда ***uv init*** создаст виртуальное окружение, а ***uv add -r requirements.txt*** установит все необходимые библиотеки и зависимости в это окружение.

**Шаг 3. Настройка базы данных**

Проверьте и при необходимости отредактируйте файл settings.py, указав параметры подключения к вашей базе данных SQL Server:

DB\_CONFIG = {

'DRIVER': '{SQL Server}',

'SERVER': 'localhost\\SQLEXPRESS',

'DATABASE': 'RoutingAppDB',

'UID': 'sa',

'PWD': 'your\_password'

}

**Шаг 4. Запуск приложения**

Запустите основное приложение командой:

uv run route\_app.py

После этого откроется графический интерфейс с возможностью ввода маршрута, настроек и построения карты.

## 5.2 Пошаговая инструкция пользователя

1. **Выбор города**
   * Выберите нужный город из выпадающего списка (например, Краснодар).
2. **Ввод адресов**
   * Введите начальный адрес маршрута (например, "Красная 10").
   * Введите конечный адрес маршрута (например, "Улица Мира 25").
3. **Добавление промежуточных точек**
   * Нажмите кнопку «+ Добавить точку» для добавления промежуточного адреса.
   * Введите адрес в появившееся поле.
4. **Выбор режима движения**
   * Выберите тип передвижения: «Пешком» или «На машине».
5. **Выбор критерия оптимизации**
   * Укажите оптимизацию маршрута: «По расстоянию» или «По времени».
6. **Построение маршрута**
   * Нажмите кнопку «Построить маршрут».

После выполнения этих шагов откроется интерактивная карта с маршрутом в вашем браузере.

## 5.3 Интерпретация результата

После построения маршрута в браузере откроется HTML-страница с картой. На ней отображаются:

* Маршрут в виде цветной линии.
* Начальная и конечная точки, а также промежуточные остановки.
* Общая длина маршрута и ориентировочное время в пути (указаны в интерфейсе и сохраняются в базу данных).

На карте можно масштабировать и перемещаться для детального просмотра маршрута.

## 5.4 Проверка сохранения маршрута

Все построенные маршруты сохраняются в базу данных SQL Server. Для проверки сохранённых маршрутов можно:

1. Подключиться к базе данных с помощью SQL Server Management Studio.
2. Открыть таблицу Routes.
3. Убедиться, что последняя запись соответствует построенному маршруту (проверить поля: city, criteria, transport, stops, total\_distance, total\_time, created\_at).

Это позволяет вести историю построений и анализировать маршруты.

## 5.5 Как расширить карту на другой город

Приложение поддерживает добавление новых городов для построения маршрутов.

**Шаги для расширения:**

1. Откройте файл settings.py.
2. Добавьте новый город в словарь CITIES, например:

CITIES = {

"Краснодар": "Краснодар, Россия",

"Москва": "Москва, Россия",

"Казань": "Казань, Россия"

}

1. Сохраните файл.

При следующем запуске приложения новый город появится в выпадающем списке. При первом построении маршрута для этого города будет автоматически загружен и сохранён граф дорожной сети из OpenStreetMap.

# Заключение

В ходе выполнения выпускной квалификационной работы была успешно решена задача разработки программного обеспечения для построения оптимальных маршрутов по улично-дорожной сети с учётом пользовательских параметров и загруженности дорог. Работа включала как теоретические исследования, так и практическую реализацию программного решения.

В процессе выполнения проекта были достигнуты следующие результаты:

* Проведен детальный анализ существующих навигационных систем и алгоритмов маршрутизации, что позволило обосновать выбор алгоритма Дейкстры в качестве основного метода поиска кратчайших путей.
* Описаны математические модели графового представления дорожной сети, критерии оценки маршрутов и методы их оптимизации.
* Разработана архитектура настольного приложения с чётким разделением на модули пользовательского интерфейса, маршрутизации, обработки графов и работы с базой данных.
* Реализовано приложение на языке Python с использованием современных библиотек (OSMnx, NetworkX, Folium, Geopy, Tkinter, PyODBC), что обеспечивает надёжность и расширяемость программного решения.
* Внедрена функциональность сохранения истории маршрутов в базе данных, позволяющая проводить анализ построенных путей в дальнейшем.
* Проведено всестороннее тестирование, включающее ручные и автоматизированные проверки, подтвердившие стабильность и корректность работы всех компонентов системы.
* Подготовлено подробное руководство пользователя, содержащее инструкции по установке, настройке и эксплуатации программного продукта.

Разработанное приложение позволяет строить маршруты с учётом выбранного города, способа передвижения и оптимизационного критерия (время или расстояние), а также визуализировать их на интерактивной карте. Система доказала свою работоспособность на тестовых сценариях и показала высокую готовность к применению в задачах городской навигации и логистики.

Перспективы дальнейшего развития проекта включают:

* Интеграцию с внешними API для получения актуальных данных о дорожной обстановке (пробки, перекрытия).
* Добавление поддержки новых типов транспорта (общественный транспорт, велосипеды и др.).
* Реализацию более сложных алгоритмов оптимизации порядка посещения промежуточных точек (решение задачи коммивояжёра).
* Расширение функциональности для использования в крупных логистических и диспетчерских системах.

Таким образом, цели и задачи, поставленные в рамках данной работы, полностью достигнуты, а полученные результаты могут служить основой для разработки практических решений в области интеллектуальных навигационных систем.

# Список использованных источников

* 1. Дейкстра Э. В. Заметка о двух проблемах в связи с графами // Numerische Mathematik. — 1959. — Т. 1, № 1. — С. 269–271.
  2. Development of an Algorithm for Constructing a Route Based on the Data of GPX-Files [Электронный ресурс] // IEEE Xplore URL: https://ieeexplore.ieee.org/document/10113405
  3. Delling D., Sanders P., Schultes D., Wagner D. Engineering Route Planning Algorithms // Algorithmics of Large and Complex Networks. — 2009. — С. 117–139.
  4. Boeing G. OSMnx: New Methods for Acquiring, Constructing, Analyzing, and Visualizing Complex Street Networks // Computers, Environment and Urban Systems. — 2017. — Vol. 65. — P. 126–139. doi:10.1016/j.compenvurbsys.2017.05.004
  5. Google Maps Platform Documentation. URL: https://developers.google.com/maps/documentation (дата обращения: 01.05.2025).
  6. Genetic Algorithm- A Literature Review [Электронный ресурс] // IEEE Xplore URL: https://ieeexplore.ieee.org/document/8862255
  7. Dorigo M., Stützle T. Ant Colony Optimization. — Cambridge: MIT Press, 2004. — 328 p.
  8. Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2), 100-107.
  9. Nielsen, J. (1994). *Usability Engineering*. Morgan Kaufmann.
  10. Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley.
  11. Myers, G. J., Sandler, C., & Badgett, T. (2011). *The Art of Software Testing*. John Wiley & Sons.

# Приложение А

**Исходный код основных модулей программы**

|  |
| --- |
| **# settings.py**  GRAPHML\_FILENAME\_DRIVE = "city\_drive.graphml"  GRAPHML\_FILENAME\_WALK = "city\_walk.graphml"  DB\_CONFIG = {      "server": "localhost",      "database": "route\_history",      "trusted\_connection": True,      "driver": "ODBC Driver 17 for SQL Server"  }  CITIES = {      "Краснодар": "Краснодар, Россия",      "Москва": "Москва, Россия",      "Санкт-Петербург": "Санкт-Петербург, Россия",      "Сочи": "Сочи, Россия",      "Ростов-на-Дону": "Ростов-на-Дону, Россия"  } |

|  |
| --- |
| **# db\_utils.py**  import pyodbc  from loguru import logger  from app.configs.settings import DB\_CONFIG  DB\_NAME = DB\_CONFIG["database"]  TABLES = {      "Routes": (          """          IF OBJECT\_ID('Routes', 'U') IS NULL          CREATE TABLE Routes (              id INT IDENTITY(1,1) PRIMARY KEY,              start\_address VARCHAR(255),              end\_address VARCHAR(255),              transport\_mode VARCHAR(50),              optimization\_mode VARCHAR(50),              distance\_km FLOAT,              time\_min FLOAT,              timestamp DATETIME DEFAULT GETDATE()          )          """      )  }  # Строка подключения к SQL Server  def get\_connection(database=None):      conn\_str = (          f"DRIVER={{{DB\_CONFIG['driver']}}};"          f"SERVER={DB\_CONFIG['server']};"          f"DATABASE={database if database else 'master'};"      )        if DB\_CONFIG.get("trusted\_connection", False):          conn\_str += "Trusted\_Connection=yes;"      else:          # Если бы в настройках были логин и пароль          conn\_str += f"UID={DB\_CONFIG.get('username', '')};"          conn\_str += f"PWD={DB\_CONFIG.get('password', '')};"        return pyodbc.connect(conn\_str)  # Создание БД и таблиц  def initialize\_database():      try:          # Подключение к master и проверка/создание базы          conn = get\_connection()          cursor = conn.cursor()          cursor.execute(f"""              IF NOT EXISTS (SELECT name FROM sys.databases WHERE name = '{DB\_NAME}')              BEGIN                  CREATE DATABASE {DB\_NAME}              END          """)          conn.commit()          cursor.close()          conn.close()          logger.info(f"База данных '{DB\_NAME}' проверена/создана.")          # Подключение уже к нужной базе          conn = get\_connection(DB\_NAME)          cursor = conn.cursor()          for table\_name, ddl in TABLES.items():              cursor.execute(ddl)              logger.info(f"Таблица '{table\_name}' проверена или создана.")          conn.commit()          cursor.close()          conn.close()      except Exception as e:          logger.error(f"Ошибка инициализации базы данных: {e}")  # Сохранение маршрута  def save\_route\_to\_db(start, end, transport\_mode, optimize\_mode, total\_length, total\_time):      try:          conn = get\_connection(DB\_NAME)          cursor = conn.cursor()          cursor.execute("""              INSERT INTO routes (start\_address, end\_address,                         transport\_mode, optimization\_mode,                         distance\_km, time\_min)              VALUES (?, ?, ?, ?, ?, ?)          """, (start, end, transport\_mode, optimize\_mode, total\_length, total\_time))          conn.commit()          cursor.close()          conn.close()          logger.info("Маршрут успешно сохранён в базе данных.")      except Exception as e:          logger.error(f"Ошибка при сохранении маршрута в БД: {e}")  def test\_connection():      try:          # Проверка подключения к master          connection = get\_connection()          logger.info("Подключение к серверу SQL Server успешно.")          connection.close()            # Проверка подключения к базе данных route\_history          connection = get\_connection(DB\_NAME)          cursor = connection.cursor()            # Попытка выполнить простой запрос к таблице routes          cursor.execute("SELECT TOP 1 \* FROM Routes")          row = cursor.fetchone()            if row:              logger.info(f"Подключение к базе {DB\_NAME} успешно. Данные из таблицы routes:")              # Получаем имена столбцов              columns = [column[0] for column in cursor.description]              # Выводим пары "имя\_столбца: значение"              for i, value in enumerate(row):                  logger.debug(f"{columns[i]}: {value}")          else:              logger.info(f"Подключение к базе {DB\_NAME} успешно. Таблица routes существует, но пуста.")            cursor.close()          connection.close()          return True        except pyodbc.Error as err:          logger.error(f"Ошибка при подключении к БД: {err}")          return False    if \_\_name\_\_ == "\_\_main\_\_":      test\_connection() |

|  |
| --- |
| **# map\_utils.py**  import os  import osmnx as ox  from app.configs.settings import GRAPHML\_FILENAME\_DRIVE, GRAPHML\_FILENAME\_WALK  ox.settings.use\_cache = True  from loguru import logger  def load\_or\_create\_map(network\_type="drive", city="Краснодар, Россия"):      # Имя файла теперь должно включать и город для разделения кэшей      city\_slug = city.split(",")[0].strip().lower().replace(" ", "\_")      filename = f"{city\_slug}\_{GRAPHML\_FILENAME\_DRIVE}" if network\_type == "drive" else f"{city\_slug}\_{GRAPHML\_FILENAME\_WALK}"        data\_dir = "data"      os.makedirs(data\_dir, exist\_ok=True)      file\_path = os.path.join(data\_dir, filename)      if os.path.exists(file\_path):          logger.info(f"Загружаем карту из файла: {file\_path}")          return ox.load\_graphml(file\_path)      else:          logger.info(f"Загрузка новой карты из OSM для {city}...")          G = ox.graph\_from\_place(city, network\_type=network\_type)          ox.save\_graphml(G, file\_path)          return G    def get\_edge\_attributes(G, route, attribute):      """      Получает атрибуты рёбер графа для заданного маршрута.        Args:          G: Граф OSMnx          route: Список узлов, составляющих маршрут          attribute: Название атрибута для извлечения        Returns:          Список значений атрибута для каждого ребра в маршруте      """      return [G[u][v][0].get(attribute, 0) for u, v in zip(route[:-1], route[1:])] |

|  |
| --- |
| **# route\_app.py**  # Стандартные библиотеки  import os  import random  import time  import webbrowser  # Сторонние библиотеки  import folium  import networkx as nx  import osmnx as ox  import tkinter as tk  from geopy.geocoders import Nominatim  from tkinter import messagebox, ttk  # Локальные модули  from app.database.db\_utils import initialize\_database, save\_route\_to\_db  from app.routing.map\_utils import load\_or\_create\_map, get\_edge\_attributes  from app.configs.settings import CITIES  from loguru import logger  # Инициализация геолокатора  geolocator = Nominatim(user\_agent="route\_app")  # Глобальный список для хранения объектов промежуточных точек  waypoints = []  def add\_waypoint():      """Добавляет поле для новой промежуточной точки маршрута"""      # Индекс новой точки (начало и конец не считаются)      idx = len(waypoints)        # Создаем новую метку и поле ввода      lbl = tk.Label(waypoints\_frame, text=f"Точка {idx+1}:")      lbl.grid(row=idx, column=0, sticky="w")        entry = tk.Entry(waypoints\_frame, width=40)      entry.grid(row=idx, column=1, padx=5, pady=2)        # Кнопка для удаления этой точки      delete\_btn = tk.Button(waypoints\_frame, text="❌", command=lambda i=idx: remove\_waypoint(i))      delete\_btn.grid(row=idx, column=2)        # Добавляем объекты в список для дальнейшего доступа      waypoints.append({"label": lbl, "entry": entry, "button": delete\_btn, "index": idx})        # Обновляем положение кнопки добавления точки      add\_waypoint\_button.grid(row=len(waypoints)+1, column=0, columnspan=3)  def remove\_waypoint(idx):      """Удаляет промежуточную точку по индексу"""      # Находим точку для удаления      for i, wp in enumerate(waypoints):          if wp["index"] == idx:              # Удаляем виджеты              wp["label"].destroy()              wp["entry"].destroy()              wp["button"].destroy()              # Удаляем из списка              waypoints.pop(i)              break        # Обновляем индексы и тексты оставшихся точек      for i, wp in enumerate(waypoints):          wp["index"] = i          wp["label"].config(text=f"Точка {i+1}:")        # Обновляем положение кнопки добавления      add\_waypoint\_button.grid(row=len(waypoints)+1, column=0, columnspan=3)  def build\_route():      """Строит маршрут с учетом промежуточных точек"""      start = start\_entry.get()      end = end\_entry.get()      travel\_mode = transport\_var.get()      opt\_mode = optimize\_var.get()      city\_key = city\_var.get()      city\_name = CITIES[city\_key]      logger.info(f"Начало построения маршрута: {start} -> {end}")      logger.info(f"Город: {city\_name}, Режим: {travel\_mode}, Оптимизация: {opt\_mode}")        # Собираем все промежуточные точки      waypoint\_addresses = [wp["entry"].get() for wp in waypoints if wp["entry"].get().strip()]      logger.info(f"Промежуточные точки ({len(waypoint\_addresses)}): {waypoint\_addresses}")        try:          # Геокодирование начальной и конечной точек          start\_full = f"{start}, {city\_name}"          end\_full = f"{end}, {city\_name}"            logger.info(f"Геокодирование начальной точки: {start\_full}")          start\_loc = geolocator.geocode(start\_full)          time.sleep(1)          logger.info(f"Геокодирование конечной точки: {end\_full}")          end\_loc = geolocator.geocode(end\_full)          time.sleep(1)            if not start\_loc or not end\_loc:              raise ValueError("Невозможно определить координаты начальной или конечной точки")            logger.info(f"Координаты начала: {start\_loc.latitude}, {start\_loc.longitude}")          logger.info(f"Координаты конца: {end\_loc.latitude}, {end\_loc.longitude}")          # Геокодирование промежуточных точек          waypoint\_locs = []          for wp\_addr in waypoint\_addresses:              wp\_full = f"{wp\_addr}, {city\_name}"              logger.info(f"Геокодирование точки: {wp\_full}")              wp\_loc = geolocator.geocode(wp\_full)              time.sleep(1)              if not wp\_loc:                  raise ValueError(f"Невозможно определить координаты точки: {wp\_addr}")              logger.info(f"Координаты точки {wp\_addr}: {wp\_loc.latitude}, {wp\_loc.longitude}")              waypoint\_locs.append(wp\_loc)          # Загружаем карту          network\_type = "drive" if travel\_mode == "На машине" else "walk"          G = load\_or\_create\_map(network\_type=network\_type, city=city\_name)          logger.info(f"Карта загружена: {len(G.nodes)} узлов, {len(G.edges)} рёбер")            # Находим ближайшие узлы для всех точек          orig\_node = ox.nearest\_nodes(G, start\_loc.longitude, start\_loc.latitude)          dest\_node = ox.nearest\_nodes(G, end\_loc.longitude, end\_loc.latitude)          waypoint\_nodes = [ox.nearest\_nodes(G, wp.longitude, wp.latitude) for wp in waypoint\_locs]            logger.info(f"Начальный узел: {orig\_node}")          logger.info(f"Конечный узел: {dest\_node}")          for i, node in enumerate(waypoint\_nodes):              logger.info(f"Узел точки {i+1}: {node}")            # Устанавливаем атрибуты ребер          for u, v, k, data in G.edges(keys=True, data=True):              length\_km = data.get("length", 1) / 1000              if travel\_mode == "Пешком":                  speed = random.uniform(4, 6)              else:                  speed = random.choice([15, 30, 40, 50, 60])              time\_hr = length\_km / speed              G[u][v][k]["travel\_time"] = time\_hr \* 60              G[u][v][k]["speed"] = speed            # Определяем вес для поиска маршрута          weight = "length" if opt\_mode == "По расстоянию" else "travel\_time"            # Строим маршрут между каждой парой точек          all\_points = [orig\_node] + waypoint\_nodes + [dest\_node]          full\_route = []          total\_length = 0          total\_time\_min = 0          for i in range(len(all\_points) - 1):              segment\_route = nx.shortest\_path(G, all\_points[i], all\_points[i+1], weight=weight)              segment\_length = sum(get\_edge\_attributes(G, segment\_route, "length")) / 1000              segment\_time = sum(get\_edge\_attributes(G, segment\_route, "travel\_time"))                # Добавляем к общим показателям              total\_length += segment\_length              total\_time\_min += segment\_time                # Добавляем к полному маршруту (без дублирования узлов)              if i == 0:                  full\_route.extend(segment\_route)              else:                  full\_route.extend(segment\_route[1:])  # Пропускаем первый узел, он уже добавлен            # Создаем и отображаем карту          map\_center = [start\_loc.latitude, start\_loc.longitude]          folium\_map = folium.Map(location=map\_center, zoom\_start=13)            # Рисуем маршрут          for u, v in zip(full\_route[:-1], full\_route[1:]):              speed = G[u][v][0].get("speed", 0)              if speed >= 40:                  color = "green"              elif speed >= 20:                  color = "yellow"              else:                  color = "red"                coords = [(G.nodes[u]["y"], G.nodes[u]["x"]),                       (G.nodes[v]["y"], G.nodes[v]["x"])]              folium.PolyLine(coords, color=color, weight=5,                             opacity=0.7).add\_to(folium\_map)            # Добавляем маркеры для всех точек          folium.Marker([start\_loc.latitude, start\_loc.longitude],                       popup="Начало", icon=folium.Icon(color="green")).add\_to(folium\_map)            for i, wp\_loc in enumerate(waypoint\_locs):              folium.Marker([wp\_loc.latitude, wp\_loc.longitude],                           popup=f"Точка {i+1}", icon=folium.Icon(color="blue")).add\_to(folium\_map)            folium.Marker([end\_loc.latitude, end\_loc.longitude],                       popup="Конец", icon=folium.Icon(color="red")).add\_to(folium\_map)            # Сохраняем и отображаем карту          map\_filename = "map\_route.html"          folium\_map.save(map\_filename)          webbrowser.open(f"file:///{os.path.abspath(map\_filename)}")            # Выводим информацию о маршруте          messagebox.showinfo(              "Готово",              f"Маршрут построен!\nДлина: {total\_length:.2f} км\n"              f"Время: {total\_time\_min:.1f} мин"          )            # Сохраняем маршрут в БД (только начальную и конечную точки)          save\_route\_to\_db(start, end, travel\_mode, opt\_mode, total\_length, total\_time\_min)        except Exception as e:          messagebox.showerror("Ошибка", str(e))  def init\_gui():      """Инициализирует графический интерфейс приложения"""      global root, waypoints\_frame, start\_entry, end\_entry, city\_var, transport\_var, optimize\_var, add\_waypoint\_button       # Инициализация базы данных      initialize\_database()        # Создание основного окна      root = tk.Tk()      root.title("Оптимальный маршрут")      # Основной фрейм      main\_frame = tk.Frame(root, padx=10, pady=10)      main\_frame.pack(fill=tk.BOTH, expand=True)      # Начальная и конечная точки      tk.Label(main\_frame, text="Начальный адрес:").grid(row=0, column=0, sticky="w")      start\_entry = tk.Entry(main\_frame, width=40)      start\_entry.grid(row=0, column=1, padx=5, pady=5)      tk.Label(main\_frame, text="Конечный адрес:").grid(row=1, column=0, sticky="w")      end\_entry = tk.Entry(main\_frame, width=40)      end\_entry.grid(row=1, column=1, padx=5, pady=5)      # Фрейм для промежуточных точек      waypoints\_frame = tk.LabelFrame(main\_frame, text="Промежуточные точки")      waypoints\_frame.grid(row=2, column=0, columnspan=2, padx=5, pady=5, sticky="ew")      # Кнопка для добавления новой точки      add\_waypoint\_button = tk.Button(waypoints\_frame, text="+ Добавить точку", command=add\_waypoint)      add\_waypoint\_button.grid(row=0, column=0, columnspan=3, pady=5)      # Настройки маршрута      settings\_frame = tk.LabelFrame(main\_frame, text="Настройки маршрута")      settings\_frame.grid(row=3, column=0, columnspan=2, padx=5, pady=5, sticky="ew")      tk.Label(settings\_frame, text="Город:").grid(row=0, column=0, sticky="w")      city\_var = ttk.Combobox(settings\_frame, values=list(CITIES.keys()))      city\_var.current(0)      city\_var.grid(row=0, column=1, padx=5, pady=5, sticky="ew")      tk.Label(settings\_frame, text="Транспорт:").grid(row=1, column=0, sticky="w")      transport\_var = ttk.Combobox(settings\_frame, values=["На машине", "Пешком"])      transport\_var.current(0)      transport\_var.grid(row=1, column=1, padx=5, pady=5, sticky="ew")      tk.Label(settings\_frame, text="Оптимизация:").grid(row=2, column=0, sticky="w")      optimize\_var = ttk.Combobox(settings\_frame, values=["По расстоянию", "По времени"])      optimize\_var.current(0)      optimize\_var.grid(row=2, column=1, padx=5, pady=5, sticky="ew")      # Кнопка построения маршрута      build\_button = tk.Button(main\_frame, text="Построить маршрут", command=build\_route)      build\_button.grid(row=4, column=0, columnspan=2, pady=10)      return root  def run\_app():      """Запускает приложение"""      gui = init\_gui()      gui.mainloop()  if \_\_name\_\_ == "\_\_main\_\_":      run\_app() |

# Приложение Б

Скриншоты пользовательского интерфейса и результатов

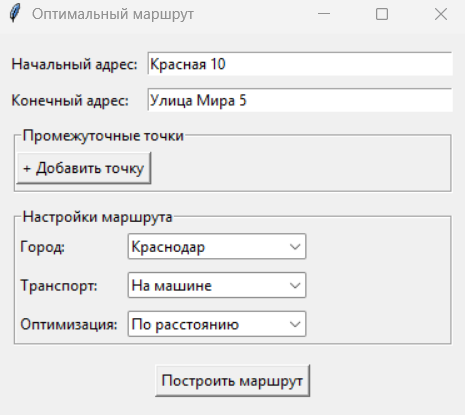
****

Рис. Б.1 – Главное окно приложения для ввода параметров маршрута.

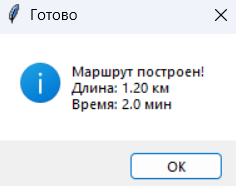


Рис. Б.2 – Окно уведомления о построенном маршруте

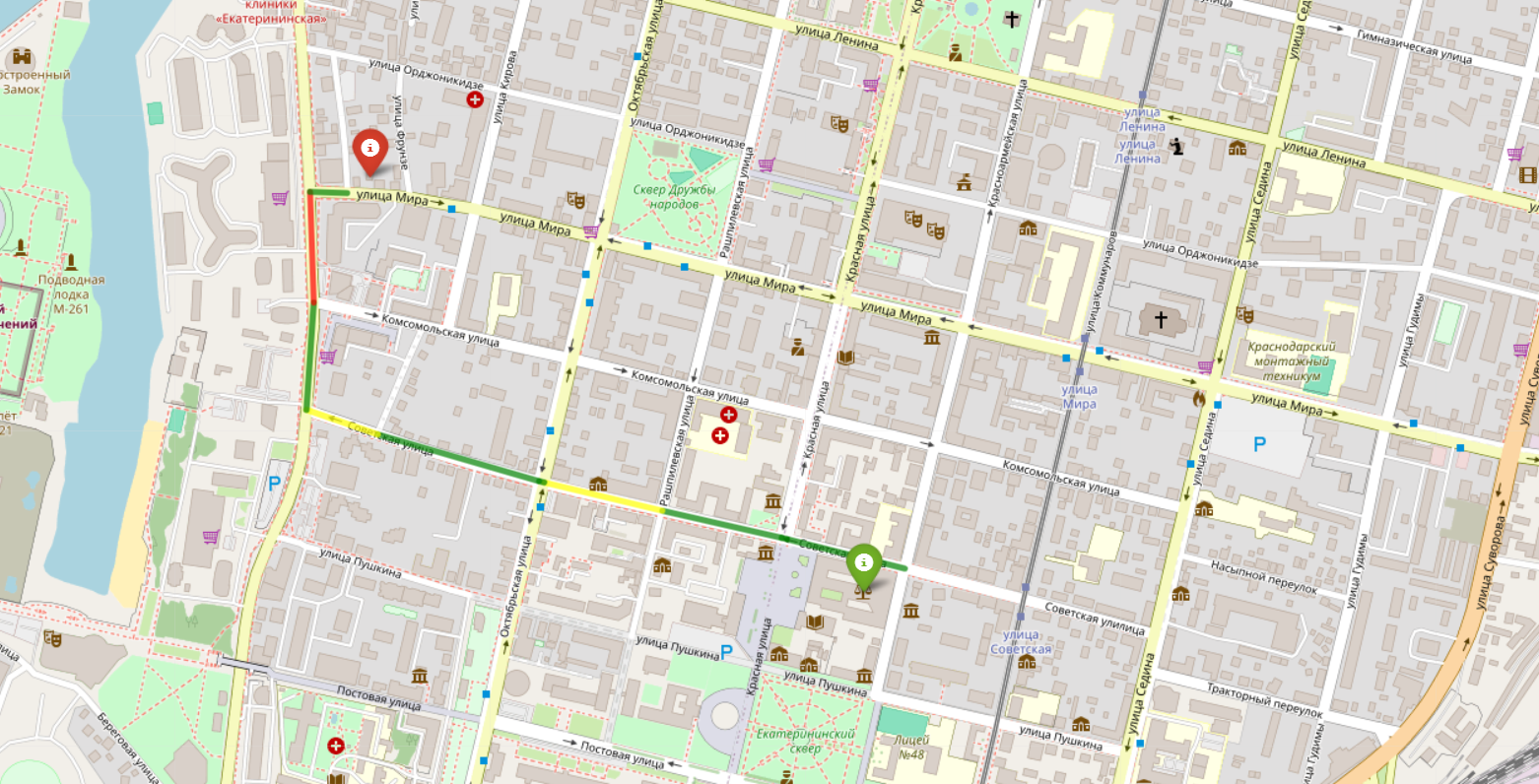
****

Рис. Б.3 – Визуализация построенного маршрута

на интерактивной карте.

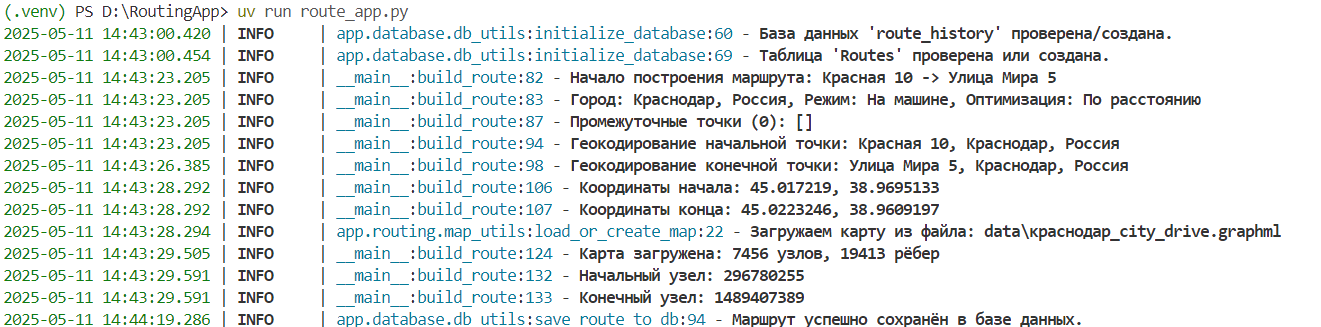
****

Рис. Б.4 – Журнал логов приложения при выполнении

построения маршрута.

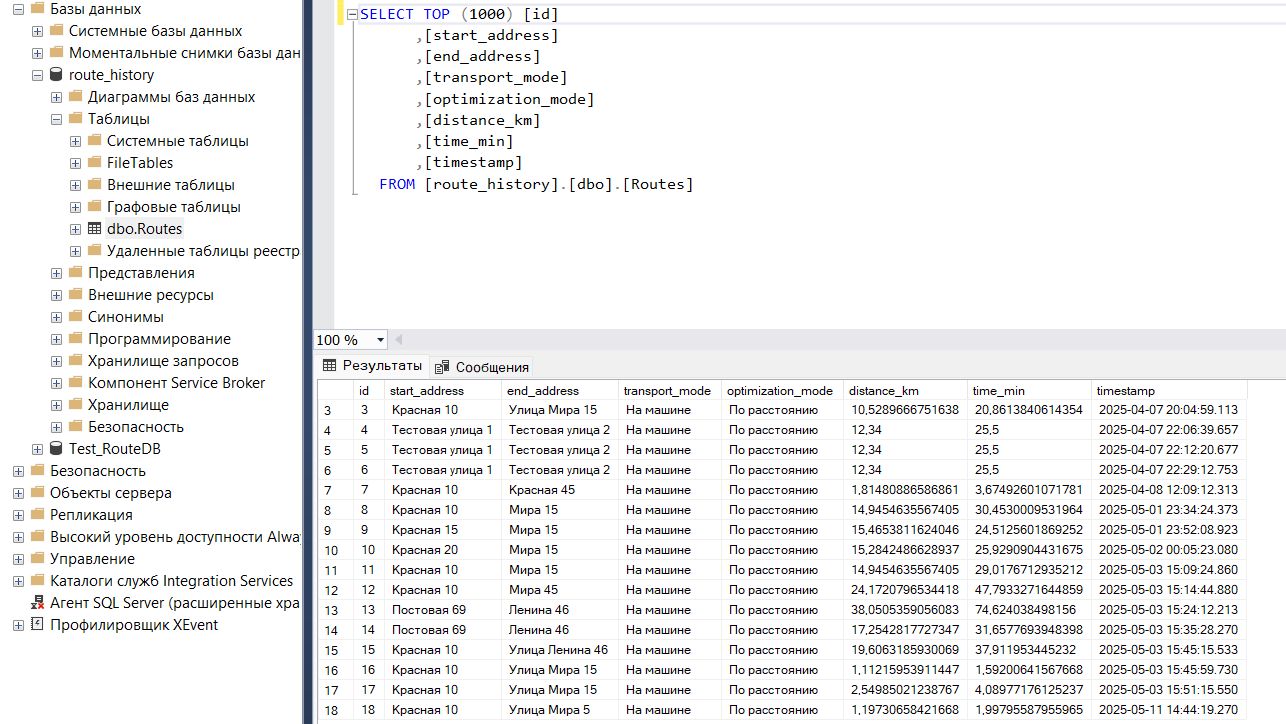


Рис. Б.5 – Содержимое таблицы Routes в базе данных SQL Server.