**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**МЕХАНИКО-МАТЕМАТИЧЕСКИЙ ФАКУЛЬТЕТ**

**Кафедра дифференциальных уравнений и системного анализа**

**АЛГОРИТМЫ ПОСТРОЕНИЯ МАРШРУТА НА РЕАЛЬНОЙ КАРТЕ**

Курсовая работа

Конаева Кирилла Витальевича

студента 2 курса, специальность 1-31 03 09 Компьютерная математика   
и системный анализ

Научный руководитель:  
старший преподаватель,  
А. В. Кушнеров

Минск, 2024

# ОГЛАВЛЕНИЕ

Оглавление

[ОГЛАВЛЕНИЕ 2](#_Toc166530187)

[ВВЕДЕНИЕ 3](#_Toc166530188)

[ГЛАВА 1 4](#_Toc166530189)

[ПОСТАНОВКА ЗАДАЧИ И МЕТОДОЛОГИЯ 4](#_Toc166530190)

[1.1 Постановка задачи 4](#_Toc166530191)

[1.2 Алгоритмы для построения маршрутов на карте 4](#_Toc166530192)

[1.3 Файлы формата .xml и .pbf 7](#_Toc166530193)

[1.4 Расстояние между объектами 8](#_Toc166530194)

[ГЛАВА 2 9](#_Toc166530196)

[Реализация приложения для построения маршрутов на реальной карте 9](#_Toc166530197)

[2.1 Подготовка датасета. 9](#_Toc166530198)

[2.2 Алгоритм построения графа 10](#_Toc166530199)

[2.3 Вспомогательные функции для построения кратчайшего пути 12](#_Toc166530200)

[2.4 Алгоритм для построения кратчайшего пути 14](#_Toc166530201)

[2.5 Main-функция 16](#_Toc166530202)

[2.6 Библиотеки tkinter и tkintermapview для реализация графического интерфейса пользователя 17](#_Toc166530203)

[ЗАКЛЮЧЕНИЕ 20](#_Toc166530207)

[СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ 22](#_Toc166530208)

[ПРИЛОЖЕНИЕ А. 23](#_Toc166530209)

ВВЕДЕНИЕ

Туризм занимает важную часть нашей жизни. Многие люди либо раз в год, либо несколько раз могут посещать новые для них места: страны, города и иные неизвестные для путешественников места. Соответственно, возникает проблема ориентирования в новой локации. На решение данной проблемы направлена моя курсовая работа. Но на сегодняшний день существует множество различных приложений, которые могут помочь путешественникам. Они могут прокладывать маршруты от текущего местоположения до того, куда хочет добраться человек. Данные приложения многофункциональны, а следовательно, могут проложить как пеший, так и дорожный маршруты, учитывая различные метрики (тип местности, время прохождения, дорожные происшествия). Однако реализация приложения для построения маршрутов на реальной карте является непростой затеей в связи с тем, что приходится работать с файлами больших размеров, которые могут вызывать трудности при их чтении. Поэтому при реализации данного приложения привлекается большое количество программистов.

Также построение маршрута на реальной карте представляет собой сложную задачу, требующую учета множества факторов, таких как географические особенности местности, состояние дорожного покрытия, наличие препятствий и т.д. Эти факторы могут значительно варьироваться в зависимости от конкретного региона и условий движения. В связи с этим, разработка эффективных алгоритмов построения маршрута требует глубокого понимания теоретических основ и практического применения различных методик.

В первой главе данной работы будет проведен анализ существующих методов построения маршрута, их преимуществ и недостатков. Будет рассмотрена теория алгоритмов, используемых для решения задачи построения маршрута, а также методология их применения на практике.

Во второй главе работы будет представлена реализация выбранного алгоритма на примере поставленной задачи построения маршрута на реальной карте.

ГЛАВА 1

ПОСТАНОВКА ЗАДАЧИ И МЕТОДОЛОГИЯ

* 1. Постановка задачи

Данную задачу можно сформулировать следующим образом: реализовать приложение для построения маршрутов на реальной карте, которое на основе данных из файла и выборе пользователя будет строить маршрут, учитывает некоторые метрики: тип маршрута, тип дороги, расстояние. Изучить нюансы работы с моделью реальной географической карты в приложениях.

## Алгоритмы для построения маршрутов на карте

Существуют различные методы для построения маршрутов. Можно выделить некоторые из них:

1. Алгоритм Дейкстры

Алгоритм Дейкстры — это метод нахождения кратчайших путей от одной вершины графа ко всем остальным.

Преимущества:

- простота и понятность алгоритма, что делает его доступным для понимания и реализации;

- эффективность в условиях, когда граф имеет небольшие размеры и малое количество узлов;

- способность находить кратчайший путь между двумя точками в графе.

Недостатки:

- неэффективность в условиях, когда граф имеет большое количество узлов и ребер, что приводит к высокой вычислительной нагрузке;

- не учитывает стоимость перехода через узлы, если она отличается от стоимости ребра;

- имеет ряд проблем при работе с графом, у которого веса рёбер принимают отрицательные значения.

1. Алгоритм A\*

Алгоритм A\* — это модифицированный алгоритм Дейкстры, который позволяет дополнительно учитывать стоимость перехода от одного узла ко второму.

Преимущества:

- учитывает стоимость перехода через узлы, что позволяет более точно определять кратчайший путь;

- использует функцию оценки, что позволяет алгоритму выбирать наиболее перспективные узлы для дальнейшего исследования;

- более эффективен по сравнению с алгоритмом Дейкстры в условиях больших графов благодаря использованию функции оценки.

Недостатки:

- требует предварительной информации о графе для определения функции оценки, что может быть сложно в реальных условиях;

- может быть перегружен, если функция оценки неадекватно выбирает узлы для дальнейшего исследования;

- затраты на вычисление функции оценки могут быть значительными для больших графов.

1. Алгоритм Беллмана-Форда

Алгоритм Беллмана-Форда предназначен для решения задачи поиска кратчайшего пути на графе. Для заданного ориентированного взвешенного графа алгоритм находит кратчайшие расстояния от выделенной вершины-источника до всех остальных вершин графа.

Преимущества:

- обобщает алгоритм Дейкстры, позволяя найти кратчайшие пути между всеми парами узлов в графе;

- учитывает отрицательные веса ребер, что делает его универсальным для различных типов графов;

- не требует предварительной информации о графе, что упрощает его применение.

Недостатки:

- вычислительно сложен для графов с большим количеством узлов и ребер;

- неэффективен в условиях, когда граф имеет большое количество узлов и ребер, поскольку требует проверки всех возможных путей;

- может быть перегружен в графах с большим количеством узлов и ребер, что приводит к высокой вычислительной нагрузке.

Все методы в той или иной ситуации эффективны для нахождения связи между точками в графе, учитывая множество различных факторов. Но в данной работе был использован алгоритм Дейкстры, потому что:

1) для тестирования алгоритма была выбрана страна Лихтенштейн, файл которой содержит небольшое количество данных. Следовательно, алгоритм будет эффективен для работы с текущим набором данных;

2) в файле с данными отсутствуют веса рёбер графа (расстояния между точками). Это позволяет использовать алгоритм Дейкстры без опасения встречи отрицательных весов.

Все алгоритмы для построения маршрутов работают с графами. Пример подобного графа представлен на рисунке 1.

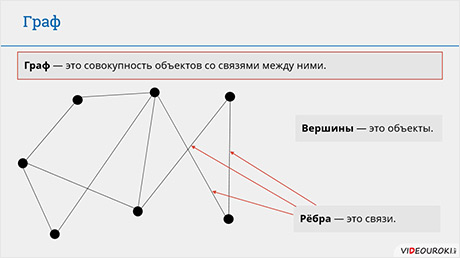


Рисунок 1.1 Граф

Граф — это абстрактная структура данных, состоящая из вершин (или узлов) и ребер, которые соединяют эти вершины. Графы используются для моделирования различных систем и отношений между объектами, такими как социальные сети, сетевые топологии, графы дорог и многие другие. В контексте теории графов, графы могут быть простыми, ориентированными, мультиграфами, смешанными и иметь множество других свойств и характеристик, в зависимости от специфики задачи и требуемого уровня детализации [1].

Но граф в готовом виде в интернете не так легко найти. Поэтому перед тем, как получить готовый граф, необходимо найти и скачать файл с данными.

При работе с картами используются следующие форматы файлом с данными: .xml (eXtensible Markup Language), pbf (Protocolbuffer Binary Format), geojson (GeoJSON), gdb (ESRI File Geodatabase).

## Файлы формата .xml и .pbf

Но для выполнения поставленной задачи производилось взаимодействие с файлами формата .xml и .pbf.

Формат .xml – используется для представления и обмена структурированными данными в электронном виде. В контексте географических данных, .xml часто ассоциируется с форматом OpenStreetMap (OSM), где данные о картах хранятся в формате OSM XML. Этот формат позволяет представлять географические объекты и их атрибуты в текстовой форме, что делает его легко читаемым как людьми, так и машинами.

Формат pbf – используется для хранения географических данных, в частности, данных OpenStreetMap (OSM). Этот формат предназначен для замены формата XML, используемого в OSM, и предлагает ряд преимуществ:

1. **эффективность хранения**: файлы в формате. pbf значительно меньше по размеру по сравнению с их эквивалентами в формате XML;
2. **быстрота записи и чтения**: запись файлов в формате. pbf происходит примерно в 5 раз быстрее, а чтение — в 6 раз быстрее, по сравнению с форматом OSM XML;
3. **расширяемость и гибкость**: формат .pbf был разработан с учетом лучшей расширяемости и гибкости в будущем, что позволяет легче адаптировать и оптимизировать обработку географических данных.

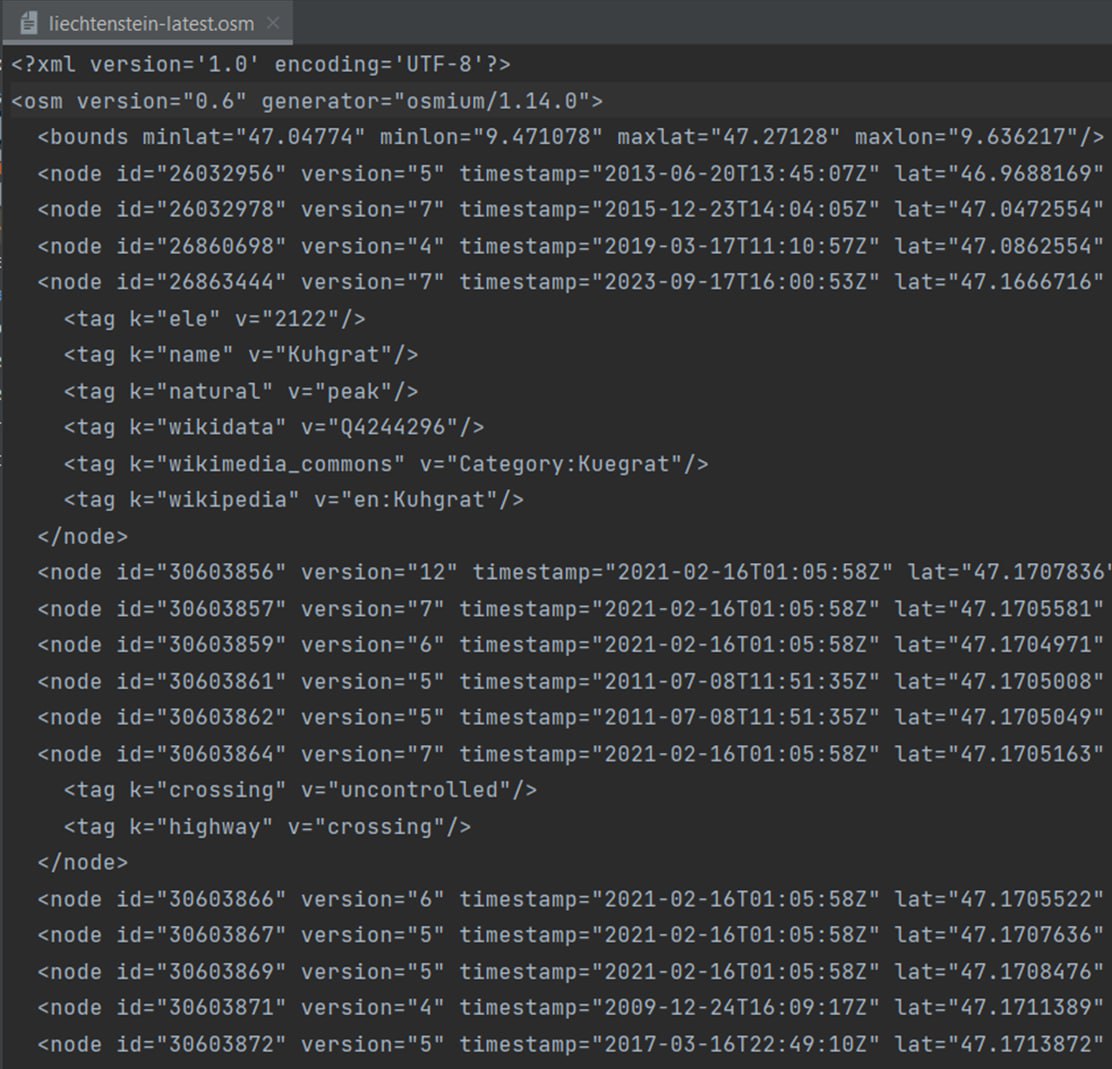
Оба файла имеют одинаковую информацию внутри, но разную структуру. А именно: данные в формате, схожим с HTML, и данные в бинарном виде.

Рисунок 1.2 – п

## 1.4 Расстояние между объектами

В связи с отсутствием каких-либо данных о расстоянии между узлами в файле, использовалась функция гаверсинуса, которая позволяет найти расстояние на сферической поверхности между двумя точками, то есть через долготу и широту. Вид функции представлен формулой 1.

(1)

где R – радиус Земли, который равен 6371 км;

phi1, phi2 – широта в радианах;

lambda1, lambda2 – долгота в радианах.

Использование данной функции является корректным при решении данной задачи, так как точки на карте расположены достаточно близко друг к другу, следовательно, посчитанное расстояние между точками будет точным.

# ГЛАВА 2

# Реализация приложения для построения маршрутов на реальной карте

В рамках данной главы будет представлена реализация основного алгоритма (алгоритм Дейкстры), а также все необходимые для работы с файлом класс и его методы.

Приложение будем тестировать на основе данных страны Лихтенштейн, так как на большие страны не хватает вычислительных мощностей персонального компьютера.

Все упомянутые классы и функции будут приведены в разделе ПРИЛОЖЕНИЯ А. Графический интерфейс пользователя будет реализован через библиотеку *tkinter* и *tkintermapview* (для работы с реальной картой в GUI *tkinter*).

## Подготовка датасета.

Первоначально необходимо получить файл с нужными данными, для этого я использовал сайт Geofabric [2].

Geofabric – это бесплатный сервер для загрузок файлом с данными карт. На сервере хранятся фрагменты данных из проекта OpenStreetMap, которые обычно обновляются каждый день. Сайт предоставляет варианты загрузок файлом различных форматов, например, .osm и .pbf. В частности, данный сервер предоставляет файл для дальнейшей работы с картой, который содержит все новейшие изменения.

OpenStreetMap – некоммерческий веб-картографический проект по созданию силами сообщества участников — пользователей Интернета подробной свободной и бесплатной географической карты мира.

В качестве карт выбран OpenStreetMap, так как получение API Google Maps не является бесплатным, а проект OpenStreetMap по картах точно не уступает Google Maps. Для построения маршрутов была выбрана страна Лихтенштейн.

В процессе сбора данных были скачены файлы с данными страны Лихтенштейн [3]. Файлы представляли собой архивы, внутри которых лежали файлы формата .osm и .pbf., содержащие информацию о следующем: точки на карте (узлы), пути, связи путей и узлов, дополнительную информацию о путях (тип дороги, состояние).

Для работы с данными файлами использовалась библиотека Python **osmium**,

которая предназначена для работы с данными OpenStreetMap (OSM) в быстрый и гибкий манер. Она поддерживает работу с различными форматами данных OSM, включая .osm, .osm.bz2, .osm.pbf.

Первоначально для удобства был реализован класс для работы с файлом, который содержит следующие методы:

1. node (self, n), который предназначен для обработки узлов из соответствующего файла;
2. way (self, w), который предназначен для обработки путей.

Данный файл на момент работы с ним содержал внутри себя 330000 узлов.

Первым шагом были обработаны все узлы файла, посредством метода **node**, которые сохранены в словаре вида:

**Узлы и их координаты**: {‘id’: (lat, lon)} (2)

где id – номер узла в файле с данными;

(lat, lon) – кортеж, который состоит из долготы и широты соответствующего узла.

## Алгоритм построения графа

Процесс построения графа можно представить в виде следующих операций:

* + 1. выбор типа пути;
    2. чтение и обработка файла с данными:

- обработка узлов и создание словаря (обработка географических координаты каждого узла и сохранение их в словаре nodes);

- обработка путей, соответствующих типу пути, и создание словаря (фильтрация путей по типу движения и сохранение информации о них в словаре ways);

- подсчёт расстояний между соседними узлами и создание словаря расстояний;

* + 1. построение графа.

**Выбор типа пути**

Перед началом работы пользователь выбирает тип пути: walking (пеший) либо car (дорожный). Этот первый шаг критически важен для определения того, какие виды путей будут включены в граф. В зависимости от того, какой тип движения будет моделироваться (например, автомобильное движение, пешеходное движение, велосипедное движение), можно выбрать различные типы путей из данных OpenStreetMap. Например, для автомобилей в граф будут включены дороги, автострады, шоссе, а для пешеходов — тротуары, пешеходные дорожки, велосипедные дорожки. Предварительный выбор пути позволяет оперировать с меньшим количеством узлов и путей, что уменьшает нагрузку на вычислительную составляющую персонального компьютера.

## Чтение и обработка файла

Обработка узлов и создание словаря: на данном этапе проводится анализ данных OpenStreetMap, чтобы извлечь информацию о каждом узле (точке на карте). Каждый узел имеет уникальный идентификатор, который сохраняется вместе с географическими координатами (широта и долгота) в словаре nodes. Это позволяет легко обращаться к координатам каждого узла по его идентификатору, что является ключевым для последующего построения графа и расчета расстояний между узлами.

Обработка путей, соответствующих типу пути, и создание словаря: Следующим этапом производится фильтрация путей, которые соответствуют выбранному типу движения, и сохраняется вся информация о них в словаре ways. Каждый путь состоит из набора узлов, также сохраняются идентификаторы этих узлов, дополнительная информация, такая как теги, которые могут указывать на тип дороги, скоростной ограничения и другие характеристики. Это позволяет сосредоточиться на путях, которые наиболее подходящие для конкретного применения, и обеспечивает гибкость в том, какие связи между узлами будут учтены в графе.

Подсчёт расстояний между соседними узлами и создание словаря расстояний: последним этапом обработки файла является этап вычисления расстояний между соседними узлами, используя их географические координаты. Данное расстояние считается с помощью функции гаверсинуса, которая реализована в библиотеке geopy. Далее сохраняются эти расстояния в словаре distances, связывая каждый узел с его соседями и расстояниями до них. Это критически важно для многих алгоритмов, которые могут использовать граф, например, для поиска кратчайшего пути между двумя точками.

## Построение графа

## После обработки узлов, путей и вычисления расстояний между узлами, можно приступать к построению графа. В отличие от традиционного подхода с использованием списка смежности, в нашем случае граф будет представлен в виде словарей, где каждый узел связан с его соседями через ребра, представляющие пути между ними. Это позволяет легко исследовать граф, находить кратчайшие пути, вычислять расстояния и выполнять другие аналитические задачи, сохраняя при этом гибкость и эффективность обработки данных.

## 2.3 Вспомогательные функции для построения кратчайшего пути

После того, как обработаны все необходимые данные и построен граф, можно приступать к следующему этапу. Этот этап подразумевает использование вспомогательных функций для подготовки данных и управления пользовательским интерфейсом. Вспомогательные функции необходимы для корректности работы программы, так как они предполагают собой считывание вводимых пользователем точек и нахождение к ним ближайших, если введённые точки не находятся в графе.

Для выбора точек на карте реализованы следующие функции: **get\_input()**; **find\_nearest\_point(**graph\_move: dict, my\_coords: str**)**.

**Функция get\_input().**

Функция **get\_input()** играет ключевую роль в управлении пользовательским интерфейсом и обработке ввода данных от пользователя в данном приложении. Вот подробное описание того, что делает эта функция:

1) **глобальное объявление переменной** ***points***: сначала функция обращается к глобальной переменной **points**, которая используется для хранения списка координат, введённых пользователем. Это позволяет функции добавлять новые координаты к этому списку;

2) **добавление новой координаты**: следующим шагом является получение новой координаты из поля ввода (***entry.get()****)*, которое является частью графического интерфейса пользователя, созданного с помощью библиотеки *tkinter*. Эта координата затем добавляется в список **points** с помощью метода *append*;

3) **проверка количества введённых координат**: после добавления новой координаты функция проверяет, сколько координат уже было введено. Если количество введённых координат меньше двух, то это означает, что пользователь ещё не ввёл достаточное количество точек для запуска алгоритма построения кратчайшего пути;

4) **запуск основного процесса приложения**: если в список **points** введено две или более координат, это означает, что пользователь ввел достаточное количество точек для выполнения поиска кратчайшего пути. В этом случае функция **get\_input()** завершает свою работу, и вызывается функция **run\_main()**. Эта функция отвечает за выполнение основного процесса приложения, включающего в себя поиск кратчайшего пути между введенными координатами и отображение результата на карте.

Таким образом, функция **get\_input()** обеспечивает простой и интуитивно понятный способ ввода данных пользователем, позволяя ему динамически добавлять новые координаты для поиска кратчайшего пути, а также сигнализируя о готовности к выполнению поиска путем перехода к вызову функции **run\_main()**.

**Функция find\_nearest\_point(**graph\_move: dict, my\_coords: str**).**

Чтобы предоставить пользователю удобство в выборе точек на карте, была реализована данная функция. Она необходима для корректного выполнения алгоритма: функция к двум, выбранным пользователем, точкам находит ближайшую по расстоянию в имеющемся списке узлов. Также предусмотрено то, что если тип движения выбран как дорожный, то к двум выбранным точкам будут найдены двое ближайшие из списка узлов, которые лежат на пути, доступным для движения автомобиля. Сама функция запускается уже в процессе построения кратчайшего маршрута.

Стоит заметить, что функция работает с конкретной точкой, а не со всем списком.

Процесс работы функции можно разделить на несколько пунктов:

1) **инициализация переменных:** функция начинается с инициализации двух переменных: nearest\_node, которая будет хранить идентификатор ближайшего узла, и min\_distance, которая будет хранить минимальное расстояние от заданных координат до узлов графа. Обе переменные изначально устанавливаются в значения, которые гарантируют, что они будут изменены в процессе работы функции: nearest\_node устанавливается в None, а min\_distance — в бесконечность (float("inf")).

2) **преобразование входных координат:** входные координаты пользователя (my\_coords) преобразуются из строки в кортеж чисел с плавающей точкой. Это делается путем разделения строки на отдельные компоненты (широта и долгота), их преобразования в числа с плавающей точкой и объединения в кортеж. Это необходимо для дальнейшего использования этих координат в вычислениях расстояний.

3) **поиск ближайшего узла:** функция проходит по всем узлам графа (graph\_move). Для каждого узла извлекаются его координаты (cur\_coords), и вычисляется расстояние от этих координат до заданных пользователем координат (my\_coords\_tuple). Это расстояние вычисляется с помощью функции geodesic из библиотеки geopy.distance, которая использует формулу гаверсинуса для определения расстояния между двумя географическими точками.

4) **определение ближайшего узла:** если вычисленное расстояние меньше текущего минимального расстояния (min\_distance), то min\_distance обновляется на значение нового расстояния, а nearest\_node обновляется на идентификатор узла, к которому относится это расстояние. Этот процесс повторяется для всех узлов графа, что позволяет определить узел с наименьшим расстоянием от заданных пользователем координат.

5) **возвращение результата:** после завершения перебора всех узлов графа функция возвращает идентификатор узла, который оказался ближайшим к заданным пользователем координатам. Этот идентификатор затем может быть использован для дальнейших операций, например, для поиска кратчайшего пути от этого узла до другого узла.

Таким образом, функция **find\_nearest\_point** обеспечивает удобный способ определения ближайшего узла графа к заданным пользователем координатам, что позволяет упростить процесс выбора точек на карте для последующего поиска кратчайшего пути.

**2.4 Алгоритм для построения кратчайшего пути**

Одной из последних была реализована функция **shortest\_path()**, которая является основной функцией во всей программе, так как с её помощью можно строить кратчайшие маршруты между двумя объектами на карте. А это и есть основная задача данной курсовой работы. Также стоит заметить, что функция реализована на основе алгоритма Дейкстры. Данный алгоритм в качестве параметров принимает следующие 3 параметра:

1. distance\_graph – граф связей всех подходящих узлов, представленный в виде словаря, где ключи - это вершины, а значения - словари со смежными вершинами и весами расстояний;
2. start\_vertex – точка на карте, из которой пользователь хочет строить маршрут;
3. end\_vertex – точка на карте, в которую хочет добраться пользователь.

Данный алгоритм можно разбить на следующие этапы:

1. проверка принадлежности start\_vertex и end\_vertex к графу: первым шагом алгоритм проверяет наличие выбранных пользователем узлов в построенном графе, иначе сообщает о невозможности построения маршрута между двумя заданными точками;
2. если же точки принадлежат графу, то производится инициализация необходимых переменных:

* visited – переменной задаётся значение пустого множества. Данная переменная играет важную роль в проходе по соседям каждого узла;
* distances – переменной на этапе инициализации задаётся значение словаря, где ключи - это вершины, а значения - бесконечно большие числа, представляющие расстояния до каждой вершины. Переменная необходима для хранения связей между узлами и для хранения расстояний. Важно уточнить, что distances используется не только для хранения расстояний, но и для отслеживания минимального расстояния до каждого узла
* paths – задаётся в виде словаря, который будет хранить идентификаторы узлов, по которым в дальнейшем будет построен маршрут. Если точнее, то **paths** будет хранить идентификаторы узлов, а не сами узлы. Также данная переменная необходима не только для хранения узлов, входящих в путь, но также для отображения всего маршрута на карте.

1. запуск цикла while True, внутри производиться проход вершин уже созданного графа.

Подробнее рассмотрим логику работы алгоритма c 3-го пункта:

1. просматриваются все вершины из графа distance\_graph. В одно время с просмотром всех вершин выполняется проверка на то, не принадлежит ли данный узел множеству visited. Ключевым параметром в выборе узла является расстояние. Следовательно, из всех узлов выбирается узел с наименьшим расстоянием до него. Но в нашем случае будет выбран узел start\_vertex в связи с тем, что расстояние от него к нему же уже известно и равно 0, когда расстояние до остальных узлов равно float(‘inf‘) – бесконечно большому числу.;
2. выполняется проверка на то, не является ли текущая вершина конечной. Если является, тогда функция возвращает общее расстояние до данного узла, а также переменную paths, в которой указаны идентификаторы всех узлов, которые входят в путь между начальной точкой и конечной.
3. Если же текущая точка не является конечной, тогда следующим этапом запускается цикл for, в котором рассматриваются все соседи данной точки в графе и их расстояния. Если сосед уже находится в множестве visited, тогда цикл переходит к следующему соседу. Иначе, переменной new\_distance присваивается новое значение, равное сумме . Если новое расстояние меньше того, которое было найдено до, тогда в словаре distances расстояние до текущего узла обновляется значением new\_distance. Также к списку узлов, которые лежат в пути добавляется данный узел.
4. Данный алгоритм заканчивает свою работу либо когда будет выполнен обход всего графа, либо когда будет найдена конечная точка.

**2.5 “Main-функция“**

Основная логика программы находится внутри функции **run\_main().** Данная функция представляет собой обёртку, внутри которой выполняется построение и дальнейшее отображение пути. Работу функции можно разбить на следующие этапы:

1. инициализация глобального списка **points**: сначала функция обращается к глобальному списку **points**, который используется для хранения списка координат, введённых пользователем. Этот список инициализируется как пустой в начале работы функции и пополняется по мере ввода пользователем новых координат;
2. инициализация переменных **near\_start\_node** и **near\_target\_node**: следующим этапом инициализируются переменные **near\_start\_node** и **near\_target\_node**. Они служат для того, чтобы хранить в себе информацию об идентификаторах найденных узлов. Введённые пользователем точки передаются в функцию **find\_nearest\_point**, которая в уже созданном графе ищет ближайшую точку к той, которую передал пользователь. Данная функция поочерёдно применяется к первой и второй точкам, а затем в инициализированные переменные записываются новые значения соответственно; важно отметить, что find\_nearest\_point возвращает идентификаторы узлов, а не сами узлы;
3. запуск основного алгоритма для поиска кратчайшего пути: на данном этапе запускается основной алгоритм **shortest\_path**, который принимает на вход граф связей и расстояний и переменные: **near\_start\_node**, **near\_target\_node.** Общее расстояние записывается в переменную **distance,** а список с кратчайшим маршрутом, в котором находятся идентификаторы узлов, которые содержатся в пути, в переменную **short\_path.** Стоит заметить, что **shortest\_path** возвращает общее расстояние и список узлов, образующих кратчайший путь.;
4. инициализация переменной **wayrepr**: которая служит для хранения географических координат соответствующих узлов списка **short\_path**;
5. отображение пути на карте: если переменная **wayrepr** хранит непустое значение, тогда данный путь отображается на карте посредством метода set\_path(wayrepr). Данный метод принадлежит библиотеке tkintermapview, которая в свою очередь предназначена для работы с картой OSM; заметим, что set\_path используется для отображения пути на карте, представляющего собой последовательность узлов, идентификаторы которых хранятся в **short\_path**;
6. отображение узлов: отображаются начальная и конечная вершины на карте;
7. очистка списка точек **points**: заключительным этапом производится очистка содержимого списка **points**. Это сделано для того, чтобы пользователь имел возможность многократного построения маршрутов на карте. Это позволяет пользователю не перезапускать приложение каждый раз, как ему нужен новый маршрут. Данная часть кода является важной деталью, делающей приложение более удобным для пользователя.
   1. **Библиотеки tkinter и tkintermapview для реализация графического интерфейса пользователя**

Создание приложения подразумевает собой не только алгоритмическую составляющую, а также и визуальную. В любом приложении должны быть как первая, так и вторая составляющие.

Для создания графического интерфейса пользователя и для работы с картой внутри данного GUI были использованы библиотеки: **tkinter, tkintermapview.**

**Библиотека tkinter**

Библиотека *tkinter* является стандартной библиотекой Python для создания графического интерфейса пользователя (GUI). Она позволяет разработчикам создавать окна, виджеты и меню, обеспечивая базовую структуру для взаимодействия пользователя с программой. В нашем приложении tkinter используется для создания основного окна, кнопок и полей ввода, через которые пользователь может взаимодействовать с программой.

Основные компоненты, которые использовались из **tkinter** включают:

1. главное окно (Tk): создается с помощью tkinter.Tk(), которое служит корневым окном приложения. Оно используется для размещения всех остальных элементов интерфейса;
2. поле ввода (Entry): предоставляет пользователю возможность вводить текстовые данные, например, координаты для поиска. В данном коде используется для получения координат от пользователя перед тем, как они будут использоваться для поиска ближайшего узла или конечной точки пути;
3. кнопка (Button): используется для активации определенных действий в приложении, таких как добавление новых координат в список или запуск процесса поиска пути. В коде кнопка "Добавить координаты в список" связана с функцией get\_input(), которая добавляет введенные пользователем координаты в список для дальнейшей обработки;
4. цикл обработки событий (mainloop): начинает цикл обработки событий, который ожидает и обрабатывает события, такие как нажатия клавиш или клики мышью, от пользователя. В коде mainloop запускается после настройки всех элементов интерфейса, что позволяет приложению оставаться открытым и реагировать на действия пользователя.



Рисунок 2.1 – пример дорожного маршрута в окне tkinter

**Библиотека tkintermapview**

Библиотека tkintermapview расширяет возможности *tkinter*, добавляя возможность отображения карт в окне *tkinter*. Она интегрирует различные сервисы карт, такие как Google Maps, OpenStreetMap и другие, позволяя отображать географические данные непосредственно в графическом интерфейсе пользователя. В нашем приложении tkintermapview используется для отображения карты и установки маркеров на ней, что позволяет пользователю визуально оценить расположение точек и пути между ними.

В реализованном коде применяется следующим образом:

1. **отображение карты**: используется для отображения географической карты в окне приложения. Карта загружается с помощью сервиса карт, указанного в параметре set\_tile\_server(), и может быть масштабирована и перемещена с помощью методов set\_zoom() и set\_position() соответственно;
2. **установка маркеров на карте**: Метод set\_marker() используется для размещения маркеров на карте, которые указывают на определенные географические точки. В коде маркеры устанавливаются на начальной и конечной точках пути, которые пользователь хочет найти;
3. **установка пути на карте**: Метод set\_path() используется для отображения пути между двумя точками на карте. В реализованном коде этот метод вызывается с массивом координат, который представляет собой кратчайший

Рисунок 2.2 – графический интерфейс для взаимодействия с картой

путь между начальной и конечной точками, найденный вашим алгоритмом.

Эти компоненты tkintermapview позволяют создать интерактивную и наглядную карту, которая становится частью графического интерфейса пользователя вашего приложения, делая его более удобным и понятным для пользователя.

# ЗАКЛЮЧЕНИЕ

В заключение к выполненной работе хотелось бы выделить следующие пункты:

* Алгоритм Дейкстры действительно обладает рядом преимуществ, делающих его простым для понимания и удобным для работы. Он хорошо работает с графами, в которых нет рёбер с отрицательными значениями.
* Функция гаверсинуса отлично подошла для расчёта расстояний между точками графа. Использование функции гаверсинуса позволяет точно рассчитывать расстояния между географическими точками, учитывая кривизну Земли. Также функция позволяет выбирать единицы измерения.
* Но при построении графа для пешего маршрута возникли некоторые проблемы, которые влияют на работоспособность кода. Основная проблема заключается в том, что при чтении файла могли быть учтены не все теги путей, что привело к ситуации, в которой были выбраны не все доступные для пешехода пути. Данная задача требует детального анализа файла с данными, что является очень трудоёмким процессов. В целом, пеший маршрут также может быть построен, но в некоторых моментах могут возникать ошибки.
* Однако, несмотря на преимущества алгоритма Дейкстры, ему присущи определенные ограничения, включая невозможность работы с графами, содержащими отрицательные веса ребер, и потенциальную медленность при работе с очень большими графами. Вместе с тем, существует множество модификаций и улучшений алгоритма, например, A\*, которые могут предложить дополнительные преимущества. Также можно использовать приоритетную очередь: вместо перебора всех непосещённых вершин на каждом шаге, можно использовать приоритетную очередь. Оптимизация обновлений расстояний: при обновлении расстояний до соседних вершин, можно сразу же добавлять новую вершину в очередь, если она не посещена. Это избавит от необходимости проверять наличие вершины в очереди на каждом шаге.
* Использование библиотеки osmium для обработки OpenStreetMap данных в данном коде позволяет создать мощный инструмент для анализа и манипулирования географической информации. Библиотека предоставляет гибкие возможности для чтения и анализа данных OpenStreetMap в различных форматах, включая PBF, что делает данную библиотеку подходящей для выполнения задания потому, что подходит для работы с большими наборами данных. Также данная библиотека позволяет легко фильтровать данные по различным параметрам, например, типам дорог для определённого типа транспорта, что упрощает подготовку данных.
* В конечном итоге был реализован класс, в котором были реализованы необходимые функции для обработки данных файла OpenStreetMap, функция для поиска ближайших узлов в графе, функция для построения графа, функция для ввода пользователем точек с карты, функция для построения кратчайшего пути и функция для расчёта расстояния между двумя узлами. Также был реализован графический интерфейс пользователя, который позволяет комфортно обращаться с готовым приложением.

# СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Волосатов, Е. Теория графов. Термины и определения в картинках/ Е. Волосатов // Хабр [Электронный ресурс]. – 2021. – Режим доступа: <https://habr.com/ru/companies/otus/articles/568026/>. – Дата доступа: 09.05.2024
2. Geofabric [Электронный ресурс] – Режим доступа: <https://download.geofabrik.de/>. – Дата доступа: 11.03.2024
3. OpenStreetMap Wiki [Электронный ресурс] – Режим доступа: <https://wiki.openstreetmap.org/>. – Дата доступа: 11.03.2024
4. Pyosmium’s documentation [Электронный ресурс] – Режим доступа: <https://docs.osmcode.org/pyosmium/latest/>. – Дата доступа: 13.03.2024
5. Tkinter documentation [Электронный ресурс] – Режим доступа: <https://docs.python.org/3/library/tkinter.html/>. – Дата доступа: 19.03.2024
6. GeoPy documentation [Электронный ресурс] – Режим доступа: [https://geopy.readthedocs.io/en/stable/.](https://geopy.readthedocs.io/en/stable/.-) – Дата доступа: 15.03.2024

# ПРИЛОЖЕНИЕ А.

**Код программы**

Ниже будет приведён основной код программы, который был написан при реализации приложения:

import osmium

import tkinter

import tkintermapview

from geopy.distance import geodesic

class CounterHandler(osmium.SimpleHandler):

def \_\_init\_\_(self, way\_type):

super().\_\_init\_\_()

# osmium.SimpleHandler.\_\_init\_\_(self)

self.type\_of\_way = way\_type

self.nodes = {}

self.ways = {}

self.graph = {}

self.distances = {}

self.highway\_by\_car = {'track', 'path', 'residential', 'primary',

'secondary', 'unclassified', 'tertiary',

'service', 'driveway', 'motorway',

'trunk', 'parking\_aisle'}

self.highway\_by\_walk = {'service', 'pedestrian', 'unclassified',

'footway', 'track', 'path', 'steps',

'cycleway', 'parking\_aisle',

'bridleway', 'crossing'}

def node(self, n):

lat = n.location.lat

lon = n.location.lon

self.nodes[str(n.id)] = (lat, lon)

# метод для парсинга путей из файла

def way(self, w):

way\_id = w.id

nodes\_in\_way = [node.ref for node in w.nodes]

tags = []

if w.tags:

tags = [{'k': tag.k, 'v': tag.v} for tag in w.tags]

if self.type\_of\_way == 'car':

relevant\_tags = self.highway\_by\_car

elif self.type\_of\_way == 'walking':

relevant\_tags = self.highway\_by\_walk

else:

raise ValueError("Неизвестный тип движения")

if not any(tag['k'] == 'highway' and tag['v'] in relevant\_tags for tag in tags):

return

has\_building\_tag = any(tag['k'] == 'building' for tag in tags)

if has\_building\_tag:

# Если ключ 'building' найден, пропускаем этот путь

return

self.ways[way\_id] = {'id': way\_id, 'nodes': nodes\_in\_way, 'tags': tags}

def calculate\_distance(self, node1, node2):

"""Вычисляет расстояние между двумя узлами с использованием формулы гаверсинуса."""

lat1, lon1 = self.nodes[str(node1)]

lat2, lon2 = self.nodes[str(node2)]

return geodesic((lat1, lon1), (lat2, lon2)).kilometers

def build\_graph(self):

# Инициализация списка для хранения всех узлов

all\_nodes = []

# Проходим по всем путям и собираем узлы в список

for way\_data in self.ways.values():

all\_nodes.extend(way\_data['nodes']) # Расширяем список узлов

# Создаем граф, где каждому узлу соответствует пустой список

self.graph = {node: [] for node in all\_nodes}

# Проходим по всем путям еще раз для заполнения графа и словаря расстояний

for way\_data in self.ways.values():

nodes\_in\_way = way\_data['nodes'] # Получаем узлы текущего пути

# Проходим по узлам пути, кроме последнего

for i in range(len(nodes\_in\_way) - 1):

# Проверяем, существует ли уже ребро между текущим и следующим узлом

if nodes\_in\_way[i + 1] not in self.graph[nodes\_in\_way[i]]:

# Добавляем ребро в оба направления для симметрии графа

self.graph[nodes\_in\_way[i]].append(nodes\_in\_way[i + 1])

self.graph[nodes\_in\_way[i + 1]].append(nodes\_in\_way[i])

# Вычисляем расстояние между узлами и сохраняем его

distance = self.calculate\_distance(nodes\_in\_way[i], nodes\_in\_way[i + 1])

# Сохраняем расстояние в словаре расстояний

self.distances[nodes\_in\_way[i]] = self.distances.get(nodes\_in\_way[i], {}) | {

nodes\_in\_way[i + 1]: distance} # Добавляем расстояние до следующего узла

self.distances[nodes\_in\_way[i + 1]] = self.distances.get(nodes\_in\_way[i + 1], {}) | {

nodes\_in\_way[i]: distance} # Добавляем обратное расстояние

def shortest\_path(distance\_graph: dict, start\_vertex: int, end\_vertex: int):

if start\_vertex not in distance\_graph or end\_vertex not in distance\_graph:

return f'{start\_vertex} и {end\_vertex} не находятся в БД карты.'

visited = set()

distances = {elem: float('inf') for elem in distance\_graph}

distances[start\_vertex] = 0

paths = {start\_vertex: [start\_vertex]}

while True:

cur\_vertex = min((v for v in distance\_graph if v not in visited), key=lambda v: distances[v])

if cur\_vertex == end\_vertex:

return distances[end\_vertex], paths[end\_vertex]

visited.add(cur\_vertex)

for neighbor, dist in distance\_graph[cur\_vertex].items():

if neighbor in visited:

continue

new\_distance = distances[cur\_vertex] + dist

if new\_distance < distances[neighbor]:

distances[neighbor] = new\_distance

paths[neighbor] = paths[cur\_vertex] + [neighbor]

def find\_nearest\_point(graph\_move: dict, my\_coords: str):

nearest\_node = None

min\_distance = float("inf")

my\_coords\_tuple = tuple([float(x) for x in my\_coords.split()])

# my\_coords = 47.1846596 9.5516953 - пример полученных координат после выбора кликом на карту

for node\_id in graph\_move:

cur\_coords = h.nodes[str(node\_id)]

dist = geodesic(my\_coords\_tuple, cur\_coords).kilometers

if dist < min\_distance:

min\_distance = dist

nearest\_node = node\_id

return nearest\_node

near\_start\_node = None

near\_target\_node = None

def get\_input():

global points

points.append(entry.get())

if len(points) < 2:

tkinter.Button(root\_window, text='Добавить координаты в список', command=get\_input)

else:

run\_main()

def run\_main():

global points

print(f"Начальная точка: {points[0]}\nКонечная точка: {points[1]}")

near\_start\_node = find\_nearest\_point(h.graph, points[0])

near\_target\_node = find\_nearest\_point(h.graph, points[1])

distance, short\_path, \*\_ = shortest\_path(h.distances, near\_start\_node, near\_target\_node)

print(f"Общее расстояние: {distance} км\nКратчайший путь: {short\_path}")

wayrepr = [h.nodes[str(node)] for node in short\_path]

print(f"Список для построения пути между двумя точками: {wayrepr}")

if wayrepr:

map\_widget.set\_path(wayrepr)

else:

return 'Пути между двумя точками не было найдённо'

map\_widget.set\_marker(h.nodes[str(near\_start\_node)][0], h.nodes[str(near\_start\_node)][1])

map\_widget.set\_marker(h.nodes[str(near\_target\_node )][0], h.nodes[str(near\_target\_node)][1])

map\_widget.set\_position(h.nodes[str(near\_start\_node)][0], h.nodes[str(near\_start\_node)][1])

map\_widget.set\_zoom(16)

points.clear()

if \_\_name\_\_ == '\_\_main\_\_':

points = []

type\_of\_way = 'car' # 'walking' / 'car'

h = CounterHandler(type\_of\_way)

# Лучше использовать.pbf формат (более свежая версия)

h.apply\_file("Map/liechtenstein-latest.osm.pbf")

h.build\_graph()

# Работа с картой

root\_window = tkinter.Tk()

root\_window.geometry(f"{1920}x{1080}")

root\_window.title("Map Testing")

map\_widget = tkintermapview.TkinterMapView(root\_window, width=1920, height=1080, corner\_radius=0)

map\_widget.place(relx=0.5, rely=0.5, anchor=tkinter.CENTER)

map\_widget.set\_tile\_server("https://mt0.google.com/vt/lyrs=s&hl=en&x={x}&y={y}&z={z}&s=Ga")

entry = tkinter.Entry(root\_window)

entry.pack(padx=10, pady=10)

button = tkinter.Button(root\_window, text='Добавить координаты в список', command=get\_input)

button.pack(padx=10, pady=10)

map\_widget.set\_zoom(12)

map\_widget.set\_position(47.12020948185704, 9.56028781452834) # установка карты на точке, которая находится в стране

print(len(h.nodes))

map\_widget.mainloop()