

УНИВЕРСИТЕТ НАУКИ И ТЕХНОЛОГИЙ «МИСИС»

ИНСТИТУТ КОМПЬЮТЕРНЫХ НАУК

Курсовая работа

По дисциплине «Алгоритмы дискретной математики» на тему:

**Решение задачи коммивояжера на основе минимального
остовного дерева**

Работу выполнил:

Студент 2 курса

Группы БИВТ-22-2

Матяж А. И.

Научный руководитель:

Валова А. А.

Москва, 2024 г.

ВВЕДЕНИЕ

Особое место в классе транспортных задач занимает задача коммивояжера, заключающаяся в нахождении самого выгодного маршрута, проходящего через заданные пункты. Задача коммивояжера на сегодняшний день применяется в различных сферах, таких как: маршрутизация транспортных средств, выбор оптимальной траектории движения рабочего инструмента.

Целью курсовой работы является знакомство с алгоритмом решения задачи коммивояжера на основе минимального остовного дерева, его реализация на Python и применение решения в качестве модуля клиент-серверного приложения.

Задачи работы:

1. Изучение постановки задачи и понятий, связанных с решением задачи коммивояжера
2. Изучение математической модели
3. Реализация решения на основе минимального остовного дерева на языке программирования Python
4. Сравнения решения с другими способами решения задачи коммивояжера
5. Создание клиент-серверного приложения для демонстрации работы алгоритма

Объект исследования курсовой работы – задача коммивояжера. Предметами исследования является решение задачи на основе минимального остовного дерева, которое подразумевает нахождение замкнутого цикла, и клиент-серверное приложение, которое показывало практическое применение работы алгоритма.

1. ПОСТАНОВКА ЗАДАЧИ

1.1 СОДЕРЖАТЕЛЬНАЯ ПОСТАНОВКА ЗАДАЧИ

Рассматривается следующая формулировка задачи. Дрон должен доставить некие заказы в определенное число мест (городов) и вернуться в пункт

отправления. Маршрутами между пунктами доставки являются прямые, их соединяющие. Необходимо определить последовательность облета клиентов таким образом, чтобы пройденное расстояние оказалось минимальным (путь занял наименьшее время), а каждый клиент был посещен только один раз. Решением задачи является оптимальная последовательность пунктов, последовательно пройдя по которым заказы будут доставлены при кратчайшем пройденном расстоянии.

Классическая постановка задачи следующая: Дан список городов и расстояния между каждой парой городов. Каков кратчайший возможный маршрут, который посещает каждый город ровно один раз и возвращается в исходный город?

На рисунках 1,2 представлен пример графа, на котором найдено решение задачи коммивояжера (отмечено красным)

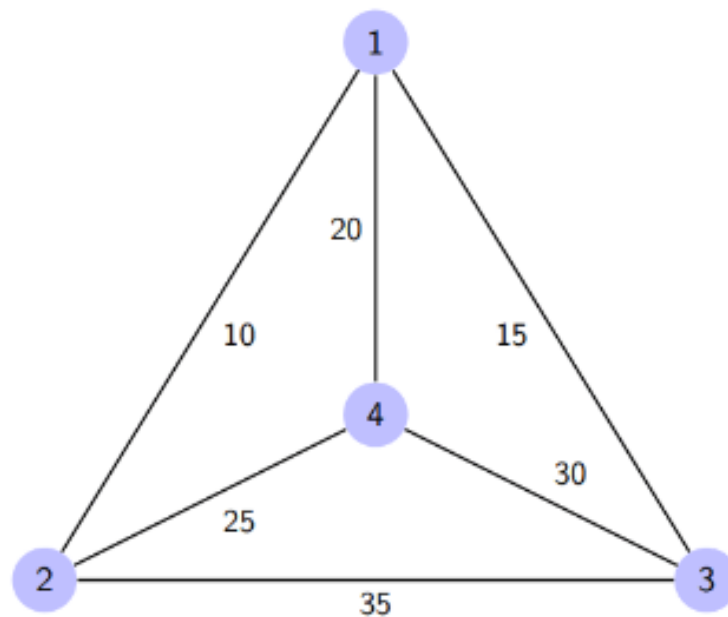


Рисунок 1. Пример графа, на котором нужно решить задачу коммивояжера

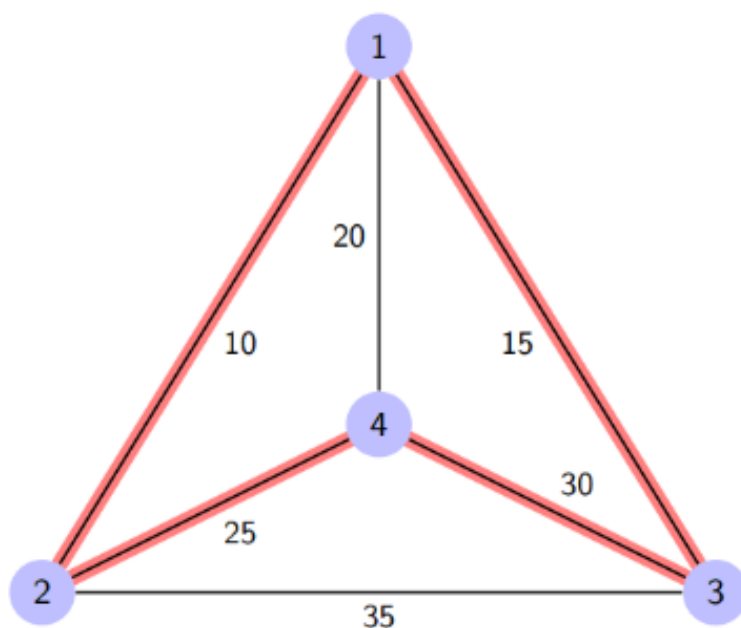


Рисунок 2. Оптимальное решения задачи коммивояжера (суммарный итоговый маршрут минимален)

1.2 МАТЕМАТИЧЕСКАЯ ПОСТАНОВКА ЗАДАЧИ

Классическая задача TSP представляет собой задачу, состоящую из целых чисел. Пусть $x_{ij} = 1$, когда торговец переходит из города i в город j . И, если $x_{ij} = 0$, следовательно перехода между городами нет. Введем город $n+1$, расположенный в том же городе, где начинает свое путешествие торговец. Теперь из первого города можно только выйти, а в город $n+1$ можно только зайти. Дополнительное целочисленное значение равно количеству способов доступа к городу $u_1 = 1, u_n + 1 = n$.

Чтобы избежать замкнутых путей, торговец должен покинуть первый город и вернуться в $n+1$. Определим дополнительные ограничения, которые связывают переменные x_{ij} и u_i $i = \{1 \dots n\}$ – данный массив представляет собой количество всех городов, которые должен обойти торговец. Матрица C_{ij} состоит из затраченного времени между городами, где $1 \leq i, j \leq n$. Задача называется симметричной, если $C_{ij} = C_{ji}$ для всех i и j , то есть стоимость проезда или вес ребра на графе между двумя города не зависит от направления движения. Математическая постановка задачи коммивояжера формулируется следующим образом:

$$\sum_{i=1}^n \sum_{j=1}^n x_{ij} c_{ij} \rightarrow \min \mid x \in \delta_\beta$$

где δ_β – это множество допустимых альтернатив, и оно представляется следующей системой ограничений:

$$\sum_{j=1}^k x_{ij} = 1 \quad \forall i \in 1, 2, \dots, n$$

$$\sum_{i=1}^n x_{ij} = 1 \quad \forall i \in 1, 2, \dots, n$$

Это ограничения, которые выполняют условие: искомый путь проходит через каждую вершину графа ровно по одному разу.

$$u_i - u_j x_{ij} \leq n - 1 \quad \forall i, j \in 2 \dots n \quad i \neq j$$

Это ограничение на то, что искомый путь должен представлять из себя единый цикл, то есть не должен распадаться на отдельные циклы.

$$x_{ij} \in 0, 1 \quad \forall i \in 1, 2, \dots, n$$

Данное ограничение гарантирует, что принимает только булевы значения, то есть 0 или 1.

$$u_i \in R^1 (\forall i \in 2, \dots, n)$$

Это то ограничение, которое гарантирует, что должны принимать вещественные значения

Задача Коммивояжера является NP – полной. Это доказывается сведением задачи к нахождению гамильтонова цикла, которая находится в списке NP-полных задач Гэри и Джонсона и представляет собой задачу о нахождении цикла, проходящего через каждую вершину графа ровно один раз.

2. РЕАЛИЗАЦИЯ

2.1 АЛГОРИТМИЧЕСКАЯ РЕАЛИЗАЦИЯ

Алгоритм решения задачи коммивояжера проделывает следующие шаги (на изображениях можно наблюдать пример работы. Исходный граф на Рисунке 3):

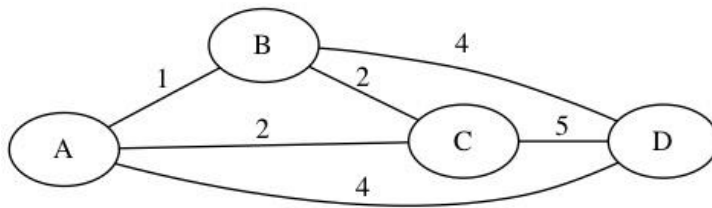


Рисунок 3. Пример работы алгоритма. Исходный граф.

1. Строим минимальное остовное дерево. (был использован алгоритм Прима, но можно использовать и алгоритм Краскала)

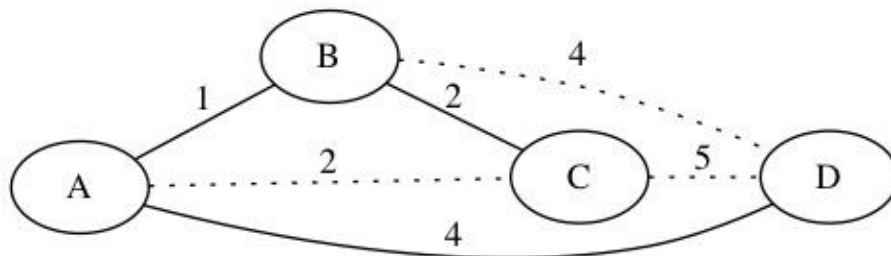


Рисунок 4. Пример работы алгоритма. Построение минимального остовного дерева.

2. Удвоить каждое ребро полученного дерева, чтобы получился цикл.

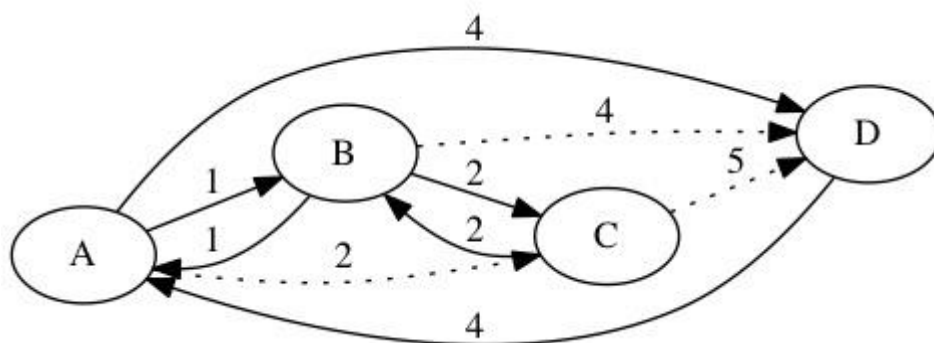


Рисунок 5. Пример работы алгоритма. Удвоение ребер в минимальном остовном дереве.

3. По полученному графу с удвоенными вершинами начать обход в глубину. Если мы начинаем использовать вершины, которые получились в результате удвоения вершин, то подпути, основанные на дубликатах пытаемся на каждой пройденной вершине, заменяем на

прямой путь. (На рисунке 6 путь, отмеченный синим цветом CBAD был преобразован в путь CD)

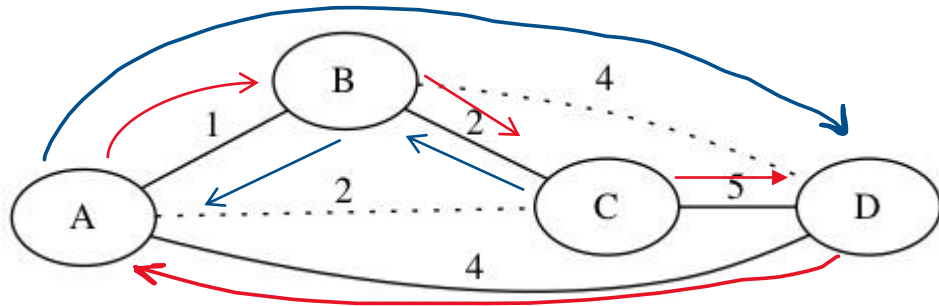


Рисунок 6. Пример работы алгоритма. Построение цикла.

2.2 ОЦЕНКА АСИМПТОТИЧЕСКОЙ СЛОЖНОСТИ

Пусть исходный граф содержит n вершин. В таком случае максимальное количество ребер составляет n^2 .

Реализация алгоритма Прима на графе с N вершинами и M ребрами, заданном с помощью списка смежности работает за $O(M \log N)$, значит, построение минимального остовного дерева при этом имеет асимптотику $O(n^2 \log n)$.

Количество ребер в минимальном остовном дереве составит $(n - 1)$, значит обход дерева займет $O(n)$ операций даже с учетом возврата по дереву составит не более $2(n-1)$.

Таким образом итоговая оценка сложности составит $O(n^2 \log n)$. (Заметим, что решение, основанное на полном переборе, работает за $O(n!)$, что значительно хуже)

2.3 ПРОГРАММНАЯ РЕАЛИЗАЦИЯ

Реализуем алгоритм, описанный в пункте 2.1 на языке Python.

Важное замечание - расстояние между двумя городами считается как декартово расстояние.

Для начала реализуем построение минимального остовного дерева MST – Minimum Spanning Tree (Рисунок 7)

```
def mst(vertexes: List[City]) -> Dict[City, List[City]]:
    """Given a set of vertexes, build a minimum spanning tree: a dict of the form {parent: [child...]}, spanning all
    vertexes."""
    tree = {vertexes[0]: []} # the first city is the root of the tree.
    links = shortest_links_first(vertexes)
    while len(tree) < len(vertexes):
        (A, B) = next((A, B) for (A, B) in links if (A in tree) ^ (B in tree))
        if A not in tree:
            A, B = B, A
        tree[A].append(B)
        tree[B] = []
    return tree
```

Рисунок 7. Функция mst

Вспомогательная функция shortest_links_first сортирует все ребра по возрастанию (Рисунок 8)

```
def shortest_links_first(cities: List[City]):
    """Return all links between cities, sorted shortest first."""
    return sorted(itertools.combinations(cities, 2), key=lambda link: distance(*link))
```

Рисунок 8. Вспомогательная функция для построения mst

Далее организуем рекурсивный проход по дереву начиная с корня (наша стартовая и конечная вершина (Рисунок 9)).

```
def preorder_traversal(tree: Dict[City, List[City]], root: City):
    """Traverse tree in pre-order, starting at root of tree."""
    yield root
    for child in tree.get(root, ()):
        yield from preorder_traversal(tree, child)
```

Рисунок 9. Функция обхода дерева

И организуем последовательный вызов этих функций, подсчет длины маршрута и вывод последовательности вершин, составляющих искомый путь коммивояжера (Рисунок 10)

```
def mst_tsp(cities_coords: List[Tuple[float, float]], cities_names: Dict[Tuple[float, float], str]):
    """Create a minimum spanning tree and walk it in pre-order."""
    cities = list(cities_coords)
    ans = list(preorder_traversal(mst(cities), cities[0]))

    order_of_travelling_coords = [(i[0], i[1]) for i in ans]
    order_of_travelling_names = [cities_names[elem] for elem in order_of_travelling_coords]
    return order_of_travelling_coords, tour_length(ans), order_of_travelling_names, ans
```

Рисунок 10. Основная функция для нахождения искомого пути

2.4 СРАВНЕНИЕ С ДРУГИМИ АЛГОРИТМАМИ

Для решения задачи коммивояжера существует очень большое количество альтернативных решений. В курсе АДМ был представлен также метод ближайшего соседа, но на самом деле решений больше.

1. Точные методы решения:

- a. Полный перебор всех путей: Асимптотика $O(n!)$
- b. Алгоритм Хелда-Карпа: Асимптотика $O(2^n n^2)$

2. Эвристические методы решения

- a. Метод ближайшего соседа

Пункты обхода последовательно включаются в маршрут, причём каждый очередной включаемый пункт должен быть ближайшим к последнему выбранному пункту среди всех остальных, ещё не включенных в состав маршрута.

- b. Муравьиный алгоритм

- c. Метод ветвей и границ.

Идет построение бинарного дерева, где каждый узел означает берем ли мы в обход какое-то из ребер. После построения дерева оценивается его глубина и отсекаются те, которые длиннее остальных.

3. Вероятностные методы решения

- a. Метод имитации отжига

2.5 ПРИМЕРЫ РАБОТЫ АЛГОРИТМА

Рассмотрим работу алгоритма в клиент-серверном приложении. Начнем с добавления городов, через которые будет проходить путь коммивояжера-дрона.

Интерфейс программы позволяет добавить в список городов новый город: достаточно ввести его название и координаты и нажать кнопку «Add City» (Рисунок 11). После этого он сохранится в базе данных.

The screenshot shows a web application interface. At the top, there is a dark grey notification box with the title 'Сообщение на этой странице' and the text 'City added successfully'. Below the notification, the heading 'Traveling Salesman' is partially visible. Underneath, the text 'Current Cities:' is followed by a bulleted list of two cities: 'Moscow (55.75222, 37.61556)' and 'Vladimir (56.1446, 40.41787)'. Below the list, there are input fields for 'City Name:', 'Latitude:', and 'Longitude:'. The 'City Name' field contains 'Nizhny Novgorod', the 'Latitude' field contains '56.2965', and the 'Longitude' field contains '43.93606'. To the right of the 'City Name' field is a button labeled 'Add City'. Below these fields is a button labeled 'Find Optimal Route'.

Сообщение на этой странице

City added successfully

OK

Traveling Salesman

Current Cities:

- Moscow (55.75222, 37.61556)
- Vladimir (56.1446, 40.41787)

City Name: Nizhny Novgorod Latitude: 56.2965 Longitude: 43.93606

Add City

Find Optimal Route

Рисунок 11. Добавление городов для последующего построения оптимального маршрута коммивояжера

После добавления всех нужных городов можно запустить поиск оптимального маршрута. На клиентскую часть вернется список городов в порядке их обхода, а также будет нарисован граф, представляющий собой схему, соответствующую оптимальному пути коммивояжера. Также будет произведен расчет суммарного расстояния для всего маршрута.

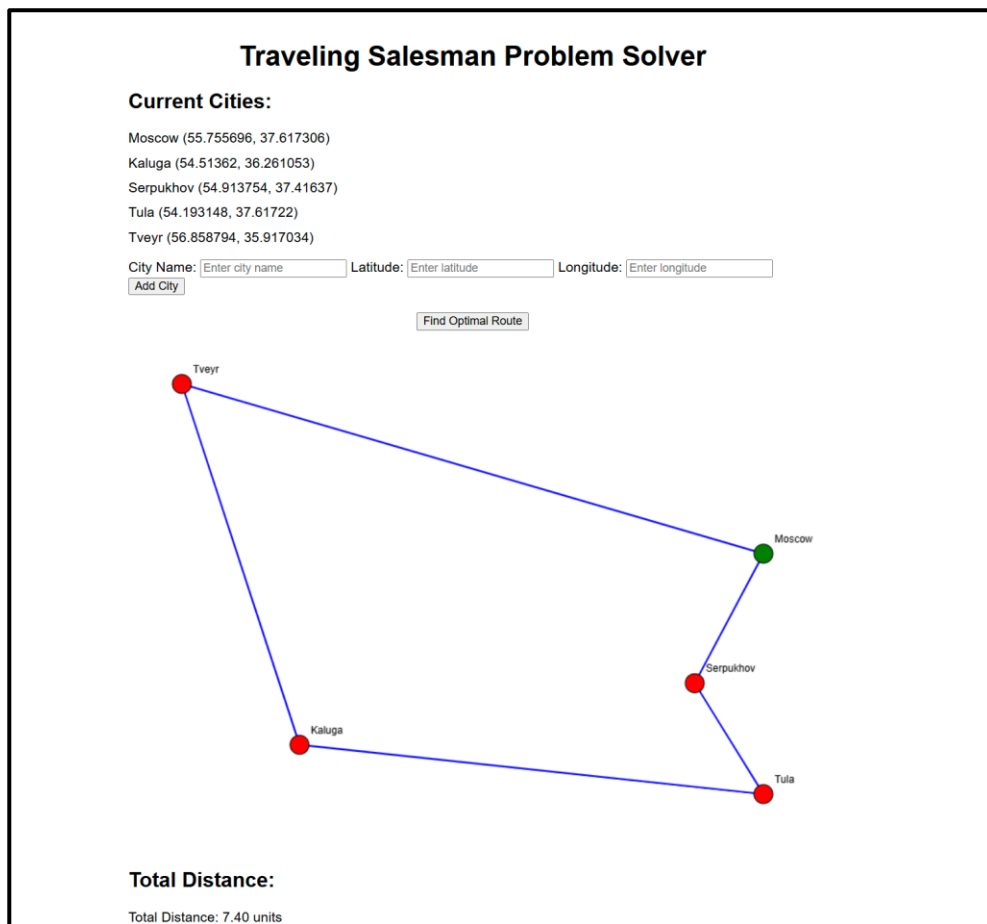


Рисунок 12. Вывод на клиенте



Рисунок 13. Продолжение вывода (Список порядка обхода вершин)

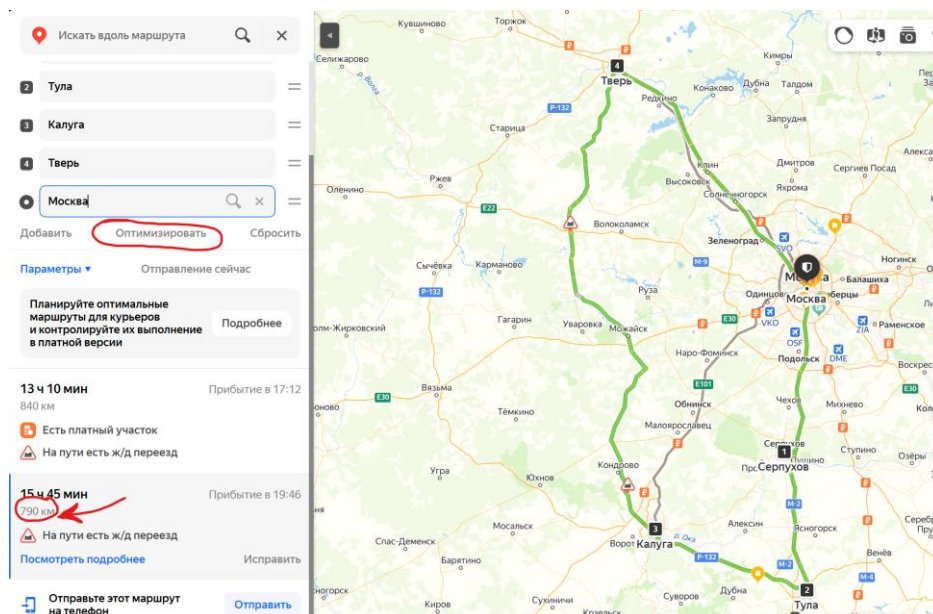


Рисунок 14. Сравнение работы алгоритма с ситуацией, когда необходимо передвигаться по дорогам общего пользования

Для наглядности посмотрим, как будет проходить пеший маршрут между указанными городами, если попросить Яндекс карты построить оптимальный маршрут.

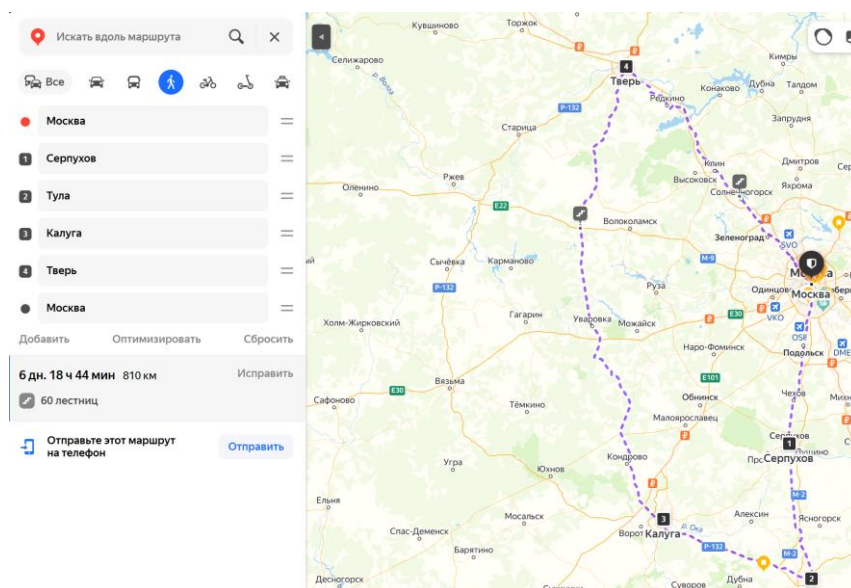


Рисунок 15. Сравнение с оптимальным пешим маршрутом по мнению Яндекса.Карт

3. РЕАЛИЗАЦИЯ КЛИЕНТ-СЕРВЕРНОГО ПРИЛОЖЕНИЯ

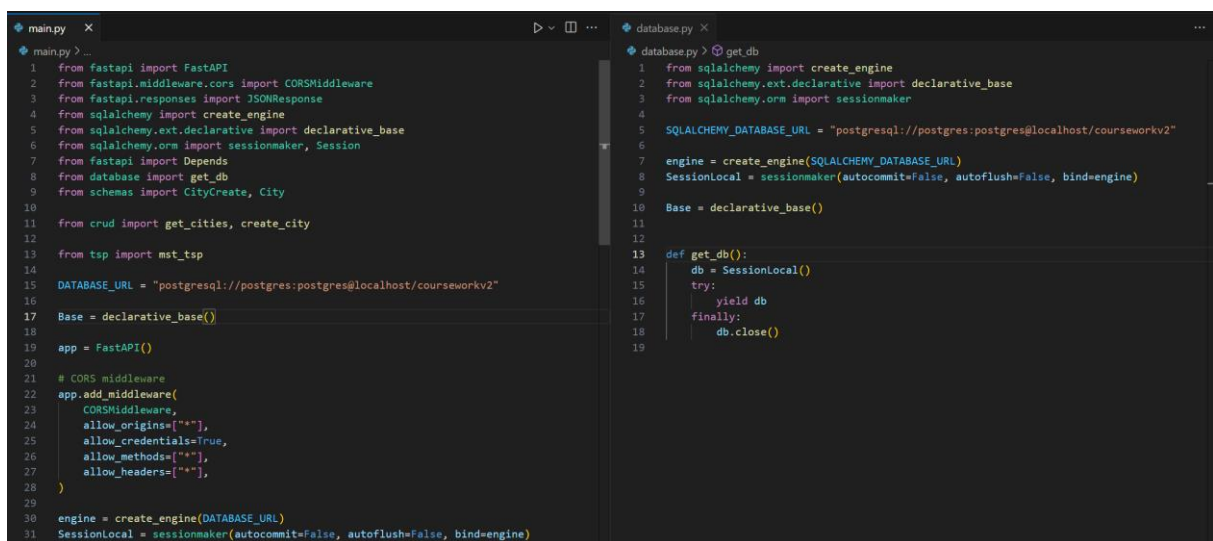
Клиент-серверное приложение будет представлять собой сервер на языке Python с фреймворком FastApi и клиент в виде браузера, html-страницы и JavaScript скрипта для того, чтобы обмениваться запросами с сервером

Реализация сервер. Для реализации сервера понадобится библиотека `fastapi`, а также следующие зависимости (Рисунок 16)

```
PS C:\Users\Mi\VSCodeProjects\courseworkv2> pip install fastapi uvicorn sqlalchemy psycopg2
>> |
```

Рисунок 16. Установка зависимостей

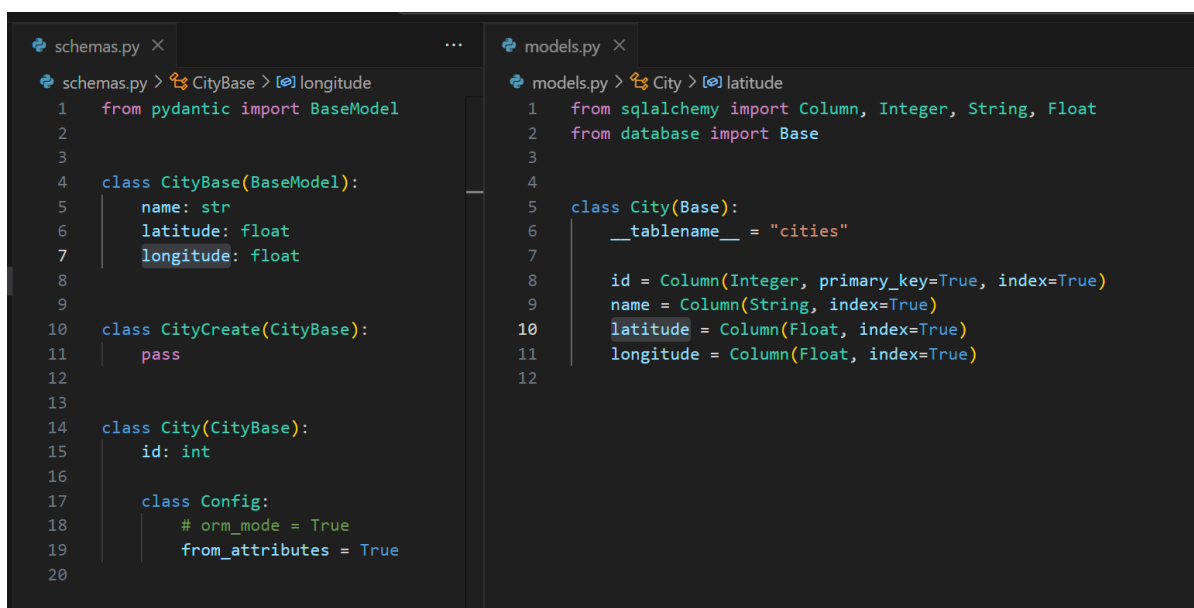
Далее инициализация приложения и подключение к источнику данных, а также объявление сущностей и объектов передачи данных (Рисунки 17-18):



```
main.py
1 from fastapi import FastAPI
2 from fastapi.middleware.cors import CORSMiddleware
3 from fastapi.responses import JSONResponse
4 from sqlalchemy import create_engine
5 from sqlalchemy.ext.declarative import declarative_base
6 from sqlalchemy.orm import sessionmaker, Session
7 from fastapi import Depends
8 from database import get_db
9 from schemas import CityCreate, City
10
11 from crud import get_cities, create_city
12
13 from tsp import mst_tsp
14
15 DATABASE_URL = "postgresql://postgres:postgres@localhost/courseworkv2"
16
17 Base = declarative_base()
18
19 app = FastAPI()
20
21 # CORS middleware
22 app.add_middleware(
23     CORSMiddleware,
24     allow_origins=["*"],
25     allow_credentials=True,
26     allow_methods=["*"],
27     allow_headers=["*"],
28 )
29
30 engine = create_engine(DATABASE_URL)
31 SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
```

```
database.py
1 from sqlalchemy import create_engine
2 from sqlalchemy.ext.declarative import declarative_base
3 from sqlalchemy.orm import sessionmaker
4
5 SQLALCHEMY_DATABASE_URL = "postgresql://postgres:postgres@localhost/courseworkv2"
6
7 engine = create_engine(SQLALCHEMY_DATABASE_URL)
8 SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
9
10 Base = declarative_base()
11
12
13 def get_db():
14     db = SessionLocal()
15     try:
16         yield db
17     finally:
18         db.close()
19
```

Рисунок 17



```
schemas.py
1 from pydantic import BaseModel
2
3
4 class CityBase(BaseModel):
5     name: str
6     latitude: float
7     longitude: float
8
9
10 class CityCreate(CityBase):
11     pass
12
13
14 class City(CityBase):
15     id: int
16
17     class Config:
18         orm_mode = True
19         from_attributes = True
20
```

```
models.py
1 from sqlalchemy import Column, Integer, String, Float
2 from database import Base
3
4
5 class City(Base):
6     __tablename__ = "cities"
7
8     id = Column(Integer, primary_key=True, index=True)
9     name = Column(String, index=True)
10    latitude = Column(Float, index=True)
11    longitude = Column(Float, index=True)
12
```

Рисунок 18

Обработка событий запроса и добавления новых городов (Рисунки 19-20)

```
@app.get("/cities/", response_model=list[City])
def read_cities(skip: int = 0, limit: int = 10, db: Session = Depends(get_db)):
    return get_cities(db, skip=skip, limit=limit)

@app.post("/cities/", response_model=City)
def create_city_view(city: CityCreate, db: Session = Depends(get_db)):
    return create_city(db=db, city=city)
```

Рисунок 19

```
crud.py > ...
1  from sqlalchemy.orm import Session
2  from models import City
3  from schemas import CityCreate
4
5
6  def get_city(db: Session, city_id: int):
7      return db.query(City).filter(City.id == city_id).first()
8
9
10 def get_cities(db: Session, skip: int = 0, limit: int = 10):
11     return db.query(City).offset(skip).limit(limit).all()
12
13
14 def create_city(db: Session, city: CityCreate):
15     db_city = City(**city.dict())
16     db.add(db_city)
17     db.commit()
18     db.refresh(db_city)
19     return db_city
20
```

Рисунок 20

Основная часть – обработка события по эндпоинту «/solve-tsp». Происходит обращение к модулю tsp.py где и реализован алгоритм решения задачи коммивояжера на основе минимального остовного дерева (Рисунок 21).

```

@app.get("/solve-tsp/")
def solve_tsp_route(db: Session = Depends(get_db)):
    all_cities = get_cities(db=db)
    cities_coords = [(city.latitude), (city.longitude)] for city in all_cities
    cities_names = dict()
    for city in all_cities:
        cities_names[(city.latitude, city.longitude)] = city.name

    order_of_travelling_coords, total_distance, order_of_travelling_names, ans = mst_tsp(cities_coords, cities_names)

    response_data = {
        "order_of_travelling_coords": order_of_travelling_coords,
        "total_distance": total_distance,
        "order_of_travelling_names": order_of_travelling_names,
        "ans": ans
    }
    return JSONResponse(content=response_data)

if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=8000)

```

Рисунок 21. Обработка события по запросу построения оптимального маршрута

Результат работы забирает себе уже клиентская часть в лице script.js.

Выводится список обхода городов и суммарное расстояние маршрута (Рисунок 22).

```

frontend > JS script.js > solveTSP
39 async function solveTSP() {
40     const response = await fetch("http://localhost:8000/solve-tsp");
41     const data = await response.json();
42
43     // Отобразить результат на странице
44     const tspCityList = document.getElementById("tspCityList");
45     tspCityList.innerHTML = ""; // Очистить список перед добавлением новых городов
46 > data.order_of_travelling_names.forEach(city => { ...
50 });
51 const totalDistanceElement = document.getElementById("totalDistance");
52 totalDistanceElement.textContent = `Total Distance: ${data.total_distance.toFixed(2)} units`;

```

Рисунок 22. Обработка на клиенте

Кроме того, клиентская часть рисует по полученным координатам сам маршрут (Рисунок 23)

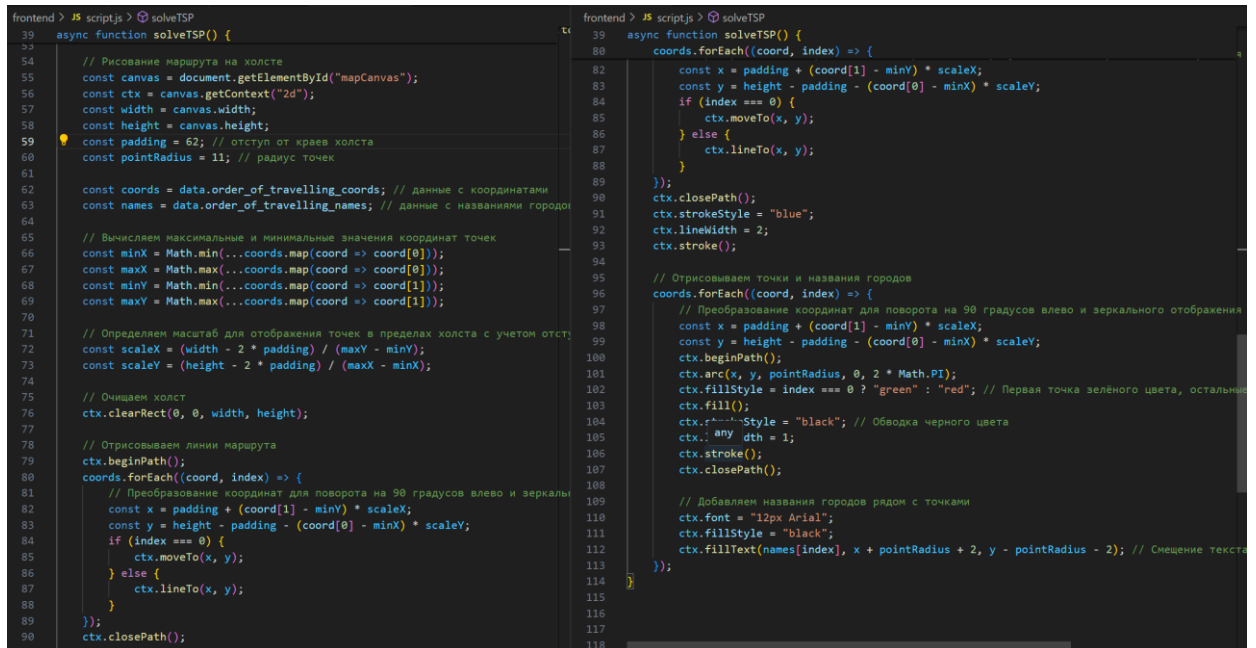


Рисунок 23 Отрисовка графа на клиенте. (Файл один и тот же - открыт в двух окнах)

Тело веб-страницы описано следующим образом (Рисунок 24):

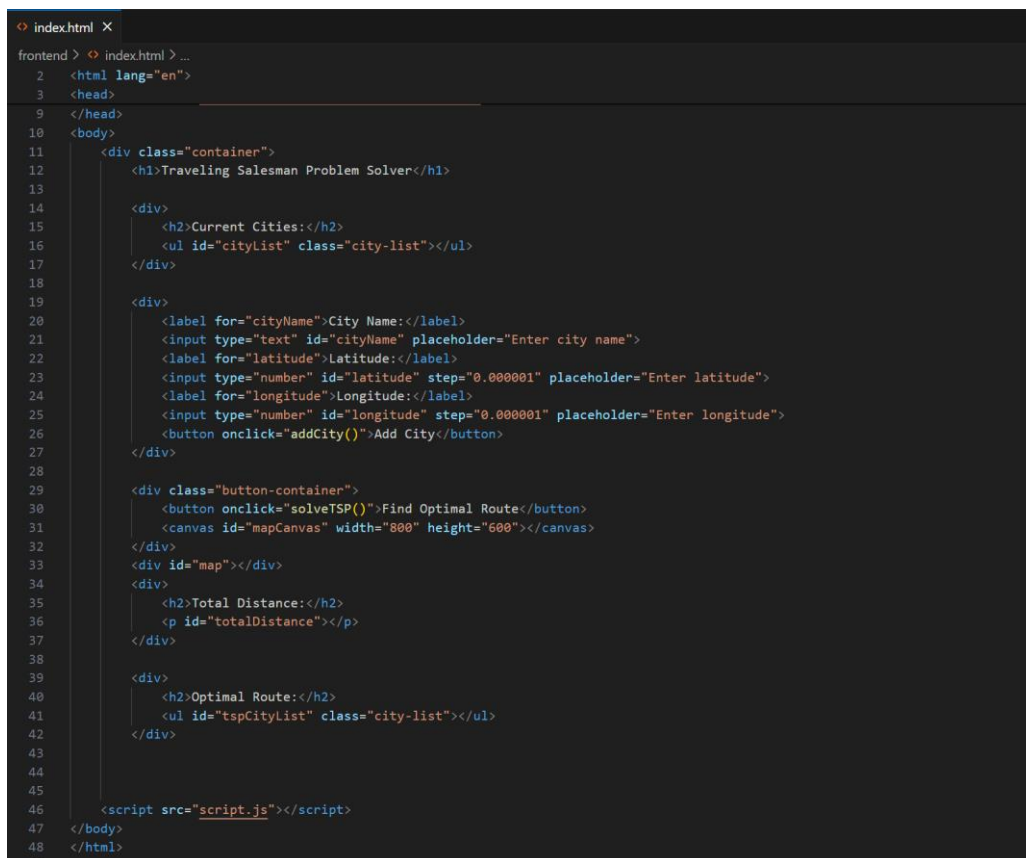


Рисунок 24. HTML-страница

Для улучшения внешнего вида веб-страницы добавлены стили CSS

```
frontend > # styles.css > body
1  body {
2      font-family: Arial, sans-serif;
3      margin: 0;
4      padding: 0;
5  }
6
7  .container {
8      max-width: 800px;
9      margin: 0 auto;
10     padding: 20px;
11 }
12
13 h1 {
14     text-align: center;
15     margin-bottom: 20px;
16 }
17
18 .city-list {
19     list-style-type: none;
20     padding: 0;
21 }
22
23 .city-list li {
24     margin-bottom: 10px;
25 }
26
27 .button-container {
28     text-align: center;
29     margin-top: 20px;
30 }
31
```

Рисунок 25. styles.css

ЗАКЛЮЧЕНИЕ

В ходе курсовой работы были успешно выполнены все поставленные задачи. Решение задачи коммивояжера с помощью минимального остовного дерева является актуальным и предоставляет достаточное приближение при несравнимо лучшей скорости работы по сравнению с существующими точными методами решения.

Кроме того, по сравнению с другими способами решения задачи коммивояжера (например, метод ближайшего соседа, метод ветвей и границ) метод решения с помощью минимального остовного дерева гарантирует длину маршрута коммивояжера отличающуюся от оптимального не более чем в два раза, что на некоторых примерах может быть крайне важно. При этом, несмотря

на такую гарантию в теории, жадные алгоритмы зачастую работают лучше на реальных примерах за счет некоторых допущений и оптимизаций.

В заключение, алгоритм A^* является одним из наиболее эффективных и распространенных алгоритмов для решения задачи поиска минимального по стоимости пути между двумя вершинами в графе. Он основывается на выборе вершины с наименьшей суммой удаленностей от начальной и конечной вершины.

В процессе исследования данного алгоритма мы обнаружили множество его преимуществ. Во-первых, алгоритм A^* гарантирует построение минимального по стоимости пути, в отличие от жадных алгоритмов. Во-вторых, он обладает линейно-логарифмической сложностью по времени, что позволяет эффективно обрабатывать даже большие графы с тысячами вершин и ребер.

В процессе изучения алгоритма A^* мы также обратили внимание на его возможные ограничения и недостатки. Например, он уступает по временной эффективности жадным алгоритмам. Это, например, тестировал в своем исследовании решений алгоритмов решения задачи коммивояжера Питер Норвиг.

В курсовой работе было реализовано клиент-серверное приложение, которое наглядно демонстрирует то, как может быть применено решение задачи коммивояжера в случае, когда между точками на плоскости существует прямой маршрут. Пользователь может протестировать работу алгоритма, добавить неограниченное количество вершин, необходимых, для посещения и получить не только порядок обхода вершин, но и визуализацию этого обхода.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. [Minimum Spanning Tree \(MST\). Two algorithms / by Jimmy \(xiaoke\) Shen / Medium](#)
2. [Minimum Spanning Trees — Algorithmic Foundations of Computer Science \(boadycs.gitlab.io\)](#)
3. [The Traveling Salesperson Problem. Solutions Comparison \(Peter Norvig, 2015–2023\)](#)
4. [Prim's Algorithm for Minimum Spanning Tree \(MST\)](#)
5. [Открытый курс лекций Курсанова Н.И. - YouTube](#)
6. [Held–Karp algorithm - Wikipedia](#)
7. [Travelling Salesman Problem: Python, C++ Algorithm \(guru99.com\)](#)

ПРОГРАММНЫЕ ПРИЛОЖЕНИЯ

[dimbag-alex/courseworkv2 \(GitHub.com\)](#) – репозиторий со всем кодом,
написанным в рамках курсовой работы