Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту
(повна назва)
Кафедра Інформатики
(повна назва)

# КВАЛІФІКАЦІЙНА РОБОТА
# Пояснювальна записка

рівень вищої освіти перший (бакалаврський)

**РОЗРОБКА ЗАСТОСУНКУ ДЛЯ ПЛАНУВАННЯ ПОДОРОЖЕЙ З ОПТИМІЗАЦІЄЮ МАРШРУТІВ**
(тема)

Виконав:
здобувач 4 року навчання,
групи ІТІНФ-21-2
Голощапова В. О.
(прізвище, ініціали)

Спеціальність 122 Комп'ютерні науки
(код і повна назва спеціальності)

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва освітньої програми)

Керівник проф. Машталір С. В.
(посада, прізвище, ініціали)

Допускається до захисту

Завідувач кафедри інформатики _____ Кобилін О. А.
(підпис) (прізвище, ініціали)

2025 р.

Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту

Кафедра Інформатики

Рівень вищої освіти перший (бакалаврський)

Спеціальність 122 Комп'ютерні науки
(код і повна назва)

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«____»_____ 2025 р.

# ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Голощаповій Вікторії Олексіївні
(прізвище, ім'я, по батькові)

1. Тема роботи Розробка застосунку для планування подорожей з оптимізацією маршрутів

затверджена наказом університету від 19 травня 2025 року № 381Ст
2. Термін подання здобувачем роботи до екзаменаційної комісії 30 травня 2025 р.
3. Вихідні дані до роботи науково-методична література, технічна документація, дані OpenStreetMap, API Google Maps, бібліотека Leaflet, FastAPI, React.js, Tailwind CSS, JSON-файли з координатами локацій, Concorde TSP Solver, матеріали конференцій, джерела з задачі комівояжера.

4. Перелік питань, що потрібно опрацювати в роботі
1. Аналіз задачі маршрутизації та методів її розв'язання.
2. Проєктування та реалізація вебзастосунку для планування туристичних маршрутів.
3. Тестування, порівняння алгоритмів та оцінка ефективності роботи системи.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри) Актуальність проблеми, постановка задачі, тестові зображення.

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

| Найменування розділу | Консультант (посада, прізвище, ім'я, по батькові) | Позначка консультанта про виконання розділу | |
|---|---|---|---|
| | | підпис | дата |
| | | | |
| | | | |

**КАЛЕНДАРНИЙ ПЛАН**

| № з/п | Назва етапів роботи | Строк / терміни виконання етапів роботи | Примітка |
|---|---|---|---|
| 1 | Отримання завдання на кваліфікаційну роботу | 07.04.2025 | |
| 2 | Аналіз завдання, підбір літератури | 08.04.25-10.04.25 | |
| 3 | Аналіз літератури з досліджуваної проблеми | 11.04.25-14.04.25 | |
| 4 | Аналіз технічних засобів | 15.04.25-20.04.25 | |
| 5 | Пошук та вивчення алгоритмів | 21.04.25-27.04.25 | |
| 6 | Програмна реалізація | 28.04.25-11.05.25 | |
| 7 | Оформлення пояснювальної записки | 12.05.25-20.05.25 | |
| 8 | Перевірка на нормоконтроль | 21.05.25-01.06.25 | |
| 9 | Перевірка на плагіат | 21.05.25-01.06.25 | |
| 10 | Рецензування | 21.05.25-01.06.25 | |
| 11 | Підготовка презентації та доповіді | 21.05.25-18.06.25 | |
| 12 | Занесення роботи в електронний архів | 02.06.25-18.06.25 | |
| 13 | Попередній захист кваліфікаційної роботи | 02.06.25-18.06.25 | |

Дата видачі завдання 7 квітня 2025 р.

Здобувач _____
(підпис)

Керівник роботи _____ проф. Машталір С. В.
(підпис) (посада, прізвище, ініціали)

# РЕФЕРАТ/ABSTRACT

Пояснювальна записка до кваліфікаційної роботи: 77 с., 9 табл., 23 рис., 1 дод., 31 джерело.

ПЛАНУВАННЯ ПОДОРОЖІ, ОПТИМІЗАЦІЯ МАРШРУТІВ, ЗАДАЧА КОМІВОЯЖЕРА, ГРАФ, МАРШРУТ, КОНКОРД, ГІЛОК І МЕЖ, ХЕЛД-КАРП.

Об’єктом роботи є процес побудови маршруту між множиною туристичних об’єктів з однаковою початковою та кінцевою точкою.

Метою роботи є розробка програмного застосунку, який дозволяє користувачеві обрати точки для відвідування на інтерактивній карті міста, після чого будувати оптимальний туристичний маршрут між ними.

У межах роботи досліджено алгоритмічні підходи до розв’язання задачі комівояжера: метод повного перебору, жадібний алгоритм, метод Хелда-Карпа, гілок і меж (Branch and Bound), а також бібліотечне рішення Concorde.

Було створено вебзастосунок для планування подорожей з вибором алгоритму та експортом маршруту. Клієнтська частина застосунку була створена в середовищі WebStorm за допомогою React.js із використанням Tailwind CSS. Для відображення інтерактивної мапи використовувався Leaflet.js, інтегрований з OpenStreetMap. Серверна частина була реалізована в середовищі PyCharm мовою Python з використанням FastAPI.

TRAVEL PLANNING, ROUTE OPTIMIZATION, TRAVELING SALESMAN PROBLEM, GRAPH, ROUTE, CONCORDE TSP, BRANCH AND BOUND, HELD-KARP.

The object of this work is the process of building a route between a set of tourist attractions with the same start and end point.

The aim of the work is to develop a software application that allows the user to select points to visit on an interactive city map and then build an optimal tourist route between them.

As part of the work, algorithmic approaches to solving the traveling salesman problem were investigated: the full search method, the greedy algorithm, the Held-Karp method, Branch and Bound, and the Concorde library solution.

A web application for travel planning with algorithm selection and route export was created. The client side of the application was created in the WebStorm environment using React.js with Tailwind CSS. Leaflet.js integrated with OpenStreetMap was used to display the interactive map. The server side was implemented in the PyCharm environment in Python using FastAPI.

# ЗМІСТ

# ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

TSP – Travelling Salesman Problem (задача комівояжера)

JSON – JavaScript Object Notation (текстовий формат обміну даними)

UI – User Interface (користувацький інтерфейс)

SPA – Single Page Application (односторінковий вебзастосунок)

API – Application Programming Interface (програмний інтерфейс взаємодії між частинами системи)

HTTP – HyperText Transfer Protocol (протокол передачі гіпертексту)

URL – Uniform Resource Locator (уніфікований ідентифікатор ресурсу)

QR-код – Quick Response Code (код швидкої відповіді)

мс – мілісекунда, одиниця вимірювання часу

с (сек) – секунда

м – метр, одиниця довжини

км – кілометр

∞ – символ нескінченності, використовується у матрицях вартості

[lat, lon] – координати точки у форматі широта-довгота

# ВСТУП

В наш час подорожі це невід'ємна частина життя більшості людей, для одних це спосіб відпочинку, для інших необхідність, пов'язана з бізнесом або віддаленими родичами. В минулі століття географічні відкриття дозволили дізнатись про раніше невідомі країни і континенти, а не так давно транспортні мережі надали змогу простому туристу дістатися навіть найвіддаленіших куточків світу. В нашому ж столітті стоїть задача зробити подорожі не лише можливими і доступними, а ще й зручними. Сучасна людина не потребує паперову мапу або компас для подорожей, адже існує безліч інструментів для орієнтування в просторі, планування маршрутів, знаходження шляху до певної точки різними методами, такими як авто, автобус, потяг, пересування пішки або комбінація вище перерахованого. Для цього використовуються просунуті технології GPS та різні алгоритми, які в більшості базуються на теорії графів. На даний момент більшість відомих застосунків для організації подорожей не пропонують опції відсортувати заплановані локації маршруту, таким чином, щоб витратити як можна менше часу в дорозі. Окрім подорожей ця функція була б корисна і для застосунків таксі, коли в замовленні вказано декілька точок, а також для логістичних компаній, які займаються постачанням вантажів або продуктів і повинні поставити вантаж в декілька місць, наприклад, поповнити запаси в усіх філіалах певного супермаркету, яких може налічувати більше 50 локацій в одному місті. Наразі даний функціонал відсутній в таких відомих застосунках, як Google maps, Bolt, Uber та інші.

Актуальність роботи полягає у тому, що завдання пошуку найкоротшого маршруту між кількома точками є класичною проблемою інформатики – задачею комівояжера. Її розв'язання є складним у загальному випадку, проте саме вона лежить в основі логіки побудови оптимального туристичного маршруту. Сучасні програмні засоби здебільшого пропонують формувати черговість точок в маршруті вручну самостійно. Саме тому розробка застосунку, який демонструє роботу класичних алгоритмів для вирішення

задачі комівояжера, є актуальним і корисним як з практичної, так і з навчальної точки зору.

Метою роботи є розробка програмного застосунку для планування подорожей з побудовою оптимальних маршрутів між вибіркою популярних для відвідування місць. У межах проєкту планується розглянути різні підходи до розв'язання задачі комівояжера, реалізувати та порівняти кілька алгоритмів оптимізації, зокрема: метод повного перебору (Brute Force), жадібний алгоритм (Greedy), метод Хелда-Карпа, гілок і меж (Branch and Bound) та бібліотечне рішення Concorde. Застосунок надасть користувачеві змогу взаємодіяти з картою міста (на прикладі міста Братислава), обирати пункти відвідування (кафе, ресторани, пам'ятки, парки тощо), а також встановлювати початкову й кінцеву точку – готель проживання.

Об'єкт роботи – процес побудови маршруту між множиною туристичних об'єктів з однаковою початковою та кінцевою точкою.

Предмет роботи – алгоритми вирішення задачі комівояжера та їх програмна реалізація.

Для досягнення поставленої мети буде використано методи аналізу, моделювання, програмної реалізації, а також порівняння алгоритмів за часом виконання та споживанням пам'яті. Для реалізації програмної складової було обрано мови програмування Python та JavaScript, а також фреймворк React.js із використанням Tailwind CSS. Для відображення інтерактивної мапи буде використовуватись Leaflet.js, інтегрований з OpenStreetMap.

Розроблений застосунок можна буде використовувати для планування різноманітних подорожей. В межах цієї роботи буде використана мапа міста Братислави, але програмну реалізацію можна буде легко використати для інших міст або для навіть інших цілей, замінивши вхідні дані. Крім того, ця система може бути використана як навчальний інструмент для демонстрації роботи алгоритмів.

# 1 АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ДЛЯ ПЛАНУВАННЯ ПОДОРОЖЕЙ ТА ОПТИМІЗАЦІЇ МАРШРУТІВ

## 1.1 Огляд існуючих програмних рішень для планування подорожей

Планування подорожей є важливим етапом для будь-якого мандрівника. Із розвитком цифрових технологій з’явилася велика кількість онлайн-сервісів та мобільних застосунків, які значно спрощують цей процес. Вони дозволяють здійснювати пошук туристичних об’єктів, формувати персоналізовані маршрути, враховувати особисті вподобання користувача. Планується розглянути найбільш популярні програмні рішення, що використовуються для планування подорожей.

Онлайн мапи – це інтерактивні цифрові картографічні сервіси, що функціонують через інтернет і забезпечують користувачів візуалізацією географічної інформації в режимі реального часу. Часто такі сервіси дозволяють дізнатися актуальну інформацію про розклад громадського транспорту, затори, ремонтні роботи. До таких сервісів належать Google Maps [1], Apple Maps [2] та Here WeGo [3]. Вони дозволяють користувачам прокладати маршрути між двома або кількома точками, переглядати транспортні шляхи, оцінювати приблизний час у дорозі та переглядати фотографії об’єктів. Однак, ці сервіси не мають функціональності вирішення задачі комівояжера, тобто не шукають оптимальну послідовність відвідування декількох точок з урахуванням мінімізації відстані або часу.

Туристичні планувальники – це застосунки, призначені допомогти мандрівникам у створенні маршрутів подорожей, виборі туристичних точок інтересу, бронюванні послуг і зберіганні важливої інформації. Вони часто інтегрують мапи, рекомендації, рейтинги, середній час відвідування та можливість формувати персоналізований план поїздки. Прикладами таких є Sygic Travel [4], TripHobo [5], Rome2Rio [6], Wanderlog [7] та Roadtrippers [8]. Вони дозволяють обирати туристичні локації на мапі, переглядати опис та

рейтинги об’єктів, а також будувати маршрут у певному порядку. Деякі з них (наприклад, TripHobo) включають прості евристичні підходи до оптимізації маршруту, однак не дають користувачеві можливості змінювати алгоритми або бачити процес оптимізації.

Застосунки з маршрутизацією для активного відпочинка – це цифрові сервіси, орієнтовані на користувачів, які займаються пішими прогулянками, велосипедними подорожами, бігом або іншими видами активного пересування. Такі застосунки враховують особливості рельєфу, погодні умови, типи доріг та фізичну підготовку користувача. Вони дозволяють будувати маршрути, відстежувати прогрес, аналізувати статистику та ділитися досвідом з іншими. Такі сервіси, як Komoot [9] або AllTrails [10], дозволяють створювати маршрут відповідно до типу активності, рівня складності та особистих уподобань. Ці застосунки мають потужну навігаційну складову, однак також не вирішують задачу оптимізації маршруту.

Інтелектуальні планувальники – це сучасні програмні рішення, які використовують алгоритми оптимізації, машинне навчання або інші методи штучного інтелекту для автоматизації процесу планування маршрутів, розкладів або завдань. Вони аналізують вхідні дані (наприклад, часові обмеження, місця інтересу, ресурси) та генерують оптимальні рішення відповідно до обраних критеріїв – мінімальний час, максимальна ефективність чи баланс навантаження. Існують окремі сервіси, зокрема OptimoRoute [11], RouteXL [12] та Circuit Route Planner [13], які спеціалізуються на вирішенні саме задачі комівояжера або її варіацій. Вони застосовуються переважно в логістиці або комерційних доставках і вимагають від користувача завантаження списку адрес або координат, після чого оптимізують маршрут згідно з заданими обмеженнями. Попри високу точність, такі сервіси рідко використовуються для туристичних цілей, адже не мають зручного візуального інтерфейсу для вибору пам’яток або інтерактивної мапи міста.

Нижче детально розглянемо, який функціонал пропонують вище згадані застосунки (табл. 1.1).

Таблиця 1.1 – Порівняння популярних сервісів для організації подорожей

| **Критерії** | **Оптимізація точок (TSP)** | **Інтерактивна мапа** | **Пошук з автозаповненням** | **Інформація про локації** | **Офлайн-режим** | **Порівняння маршрутів** | **Вибір алгоритму** | **Експорт маршруту** |
|---|---|---|---|---|---|---|---|---|
| Google Maps | Ні | Так | Так | Так | Так | Ні | Ні | Так |
| Apple MapsX | Ні | Так | Так | Так | Так | Ні | Ні | Ні |
| Here WeGo | Ні | Так | Так | Так | Так | Ні | Ні | Ні |
| Sygic Travel | Ні | Так | Так | Так | Так | Ні | Ні | Ні |
| TripHobo | Так | Так | Так | Так | Ні | Так | Ні | Так |
| Rome2Rio | Ні | Так | Так | Так | Ні | Ні | Ні | Ні |
| Wanderlog | Так | Так | Так | Так | Ні | Так | Ні | Так |
| Roadtrippers | Так | Так | Так | Так | Так | Так | Ні | Так |
| Komoot | Ні | Так | Так | Так | Так | Ні | Ні | Ні |
| AllTrails | Ні | Так | Так | Так | Так | Ні | Ні | Ні |
| OptimoRoute | Так | Ні | Ні | Ні | Ні | Так | Ні | Так |
| RouteXL | Так | Ні | Ні | Ні | Ні | Так | Ні | Так |
| Circuit Route Planner | Так | Ні | Ні | Ні | Ні | Так | Ні | Так |

Висновок: за порівняльною таблицею ми можемо чітко бачити дві категорії застосунків. Перша категорія це інтелектуальні планувальники, вони мають оптимізацію та порівняння маршрутів, але відсутні інтерактивна мапа, пошук з автозаповненням та інформація про локації. Друга категорія це всі інші, де є зручний інтерфейс, але немає просунутих функцій планування. Вибір алгоритму ж не доступний в жодному з розглянутих варіантів.

## 1.2 Актуальність задачі оптимізації маршрутів

Задача комівояжера (англ. Travelling Salesman Problem, TSP) є однією з фундаментальних у теорії графів та комбінаторній оптимізації. Вона формулюється як задача знаходження найкоротшого замкненого маршруту, що проходить через задану множину точок рівно один раз і повертається в початкову точку [14]. У контексті планування туристичних маршрутів її застосування є логічним та ефективним рішенням, адже мандрівник зазвичай бажає відвідати низку визначних місць з мінімальними витратами часу або відстані, розпочавши та завершивши маршрут у своєму готелі.

Попри те, що задача TSP формалізована ще в середині XX століття [15], більшість сучасних туристичних застосунків, включаючи популярні сервіси на кшталт Google Maps чи Rome2Rio, не реалізують автоматичне упорядкування маршруту за критерієм оптимальності. Натомість користувач змушений самостійно змінювати порядок відвідування локацій, що ускладнює планування – особливо у випадках із великою кількістю точок або з обмеженим часом перебування у місті.

Актуальність задачі підвищується ще й у зв’язку зі зростанням популярності міського туризму на вихідні дні, коли мандрівники прагнуть охопити максимум визначних місць за мінімальний час. Дослідження показують, що навіть прості алгоритми, які враховують час відвідування і логіку переміщення, дозволяють підвищити ефективність туристичних маршрутів до 30% порівняно з ручним плануванням [16].

Сучасні технології дозволяють реалізовувати як точні методи (метод гілок і меж, динамічне програмування), так і наближені підходи (жадібні алгоритми, генетичні алгоритми, мурашині системи, алгоритм Конкорд) [17]. Більшість із них можуть бути адаптовані до потреб користувача – зокрема, враховуючи тривалість зупинок, категорії об’єктів, години роботи тощо.

### 1.2.1 Прості евристики у туристичних платформах

Як вже було розглянуто більшість сучасних застосунків використовують спрощені підходи до формування маршрутів, більше уваги приділяється зручному інтерфейсу для організації короткого маршруту між двома точками, початковою та кінцевою. І варто визнати, що в більшості користувачам цього достатньо, для них більшою проблемою є організація пересадок між громадським транспортом, час в дорозі найближчий до реального. Для того щоб надати найточнішу інформацію сервіси часто використовують обмін інформацією про розклади всіх видів транспорту та затори в реальному часі, а також підйоми та спуски, пріоритет дороги. Це все допомагає як найточніше визначити орієнтовний час в дорозі. Але для деяких задач цього недостатньо, коли турист планує відвідати багато локацій в новому місті або транспортна компанія доставляє продукцію на багато точок, методи популярних платформ не враховують усі варіанти маршрутів і не гарантують мінімального сумарного шляху, тобто не вирішують задачу оптимізації маршрутів у строгому математичному сенсі.

### 1.2.2 Комерційні логістичні сервіси

Сервіси, орієнтовані на оптимізацію маршрутів для доставки (наприклад, OptimoRoute, RouteXL, Circuit Route Planner) у своїй основі активно використовують точні та наближені алгоритми вирішення задачі комівояжера. Такі системи розраховують маршрути для десятків і сотень точок, враховуючи обмеження по часу, вікна доставки, кількість транспортних засобів тощо. Вони інтегрують комбіновані алгоритми: від класичного Branch and Bound до модифікованих генетичних алгоритмів, симульованого відпалу або ж бібліотек типу Google OR-Tools.

Однак використання цих сервісів для планування особистої подорожі є незручним – інтерфейси орієнтовані на корпоративного користувача, а для

звичайного туриста відсутні функції взаємодії з мапою, фільтрації за типом об’єкта, перегляду описів локацій, тощо.

### 1.2.3 Потенціал для навчання

Окрім необхідності планувати подорожі задача комівояжера також цікава з наукової точки зору, жоден з популярних застосунків не надає користувачеві можливості вибору конкретного алгоритму оптимізації. Це створює певну непрозорість у побудові маршруту: користувач не бачить, який саме метод був використаний, чому був вибраний саме цей порядок відвідування точок і як можна було б отримати кращий варіант. Відсутність можливості побачити і протестувати альтернативні маршрути обмежує потенціал застосунків для навчання, аналізу або досліджень. Дослідивши деякі з популярних існуючих рішень, можна зробити висновок, що існує потреба у створенні програмного застосунку, який би дозволяв не тільки зручно планувати подорожі, а й наочно порівнювати різні алгоритми вирішення задачі комівояжера, обирати оптимальні маршрути, отримувати технічні метрики продуктивності – і все це у візуально зрозумілому інтерфейсі.

## 1.3 Наукові дослідження в галузі задачі комівояжера

Зада́ча комівояже́ра (комівояжер – бродячий торговець; англ. Travelling Salesman Problem, TSP; нім. Problem des Handlungsreisenden) полягає у знаходженні найвигіднішого маршруту, що проходить через вказані міста хоча б по одному разу. В умовах завдання вказуються критерій вигідності маршруту (найкоротший, найдешевший, сукупний критерій тощо) і відповідні матриці відстаней, вартості тощо. Зазвичай задано, що маршрут повинен проходити через кожне місто тільки один раз, в такому випадку розв’язок знаходиться серед гамільтонових циклів [18]. Попри простоту формулювання,

задача є NP-повною, тобто не існує відомого поліноміального алгоритму, що розв’язував би її в загальному випадку.

Перші формулювання задачі зустрічаються ще в роботах початку XX століття. Значного розвитку задача набула у 1950–60-х роках. Серед найвідоміших дослідників у цій галузі варто відзначити деяких науковців.

Меррілла Флуда (Merrill Flood) – один з перших, хто систематично вивчав TSP як приклад логістичної оптимізації.

Річарда Беллмана (Richard Bellman) – науковець, який запропонував підхід динамічного програмування до розв’язання задач комівояжера.

Джека Гельда (Jack Held) і Річарда Карпа (Richard Karp) – автори відомого динамічного алгоритму Хелда-Карпа з експоненційною, але менше факторіальною складністю.

Мартіна Гротшеля (Martin Grötschel) і колеги, які розвинули теорію політопів та застосування лінійного програмування до TSP.

Девіда Апплегейта (David Applegate), Роберта Біклера, Вільяма Кука – дослідники, які створили надзвичайно ефективний TSP-розв’язувач Concorde, що на практиці дозволяє знаходити точні рішення для задач із десятками тисяч точок.

У 1954 році Гарольд В. Кун запропонував алгоритм угорського методу для задачі призначення, який став підґрунтям для розвитку рішень TSP на малих розмірностях. У 1962 році Річард Беллман та Лестер Хелд представили динамічний підхід до задачі комівояжера – відомий як алгоритм Хелда-Карпа, який зменшив експоненційний ріст кількості обчислень до $O(n^2 2^n)$ [19].

Протягом наступних десятиліть TSP стала тестовим полігоном для розробки та аналізу численних евристичних і точних алгоритмів, зокрема: методу гілок і меж (Branch and Bound), метаевристик (генетичних алгоритмів, алгоритмів мурашиної колонії, симульованого відпалу), жадібних алгоритмів, локального пошуку, та комбінованих підходів. У 1976 році була організована бібліотека TSPLIB для уніфікації та збереження тестових випадків задачі [20].

Визначну роль у практичному розв’язанні великих екземплярів TSP відіграє програмний пакет Concorde TSP Solver [21], розроблений Девідом Апплегейтом, Робертом Біксбі, Васеком Чваталом і Вільямом Куком. У своїй монографії The Traveling Salesman Problem: A Computational Study (2007) [22] автори надають глибокий аналіз комбінаторної структури TSP, методів її оптимізації, а також описують, як за допомогою Concorde було точно розв’язано задачі з десятками тисяч вершин [23]. Програма Concorde поєднує метод гілок і меж, лінійне програмування та евристичні методи у межах однієї системи.

Значний внесок у розвиток тематики задачі комівояжера внесено також українськими дослідниками. Зокрема, професор Сергій Володимирович Машталір у своїх наукових роботах досліджував прикладні аспекти задачі комівояжера та споріднених задач маршрутизації у контексті систем підтримки прийняття рішень, логістики й мобільних технологій. Його праці охоплюють як аналіз класичних методів оптимізації, так і реалізацію алгоритмів для веборієнтованих інформаційних систем [24].

Таким чином, TSP є не лише класичною проблемою теоретичної інформатики, але й активно застосовується в практичних задачах – від логістики до туризму та проєктування мікросхем.

### 1.3.1 Класифікація методів розв’язання задачі TSP

Методи розв’язання задачі комівояжера можна умовно поділити на точні, наближені (евристичні) та метаевристичні. Кожен з підходів має свої переваги, недоліки та область застосування.

Точні методи – ці методи гарантують знаходження оптимального маршруту, проте мають високу обчислювальну складність:

– Brute Force (повний перебір) – перевіряє всі можливі перестановки точок. Має факторіальну складність $O(n!)$,практичний лише для задач з $n \leq 10$;

– Held-Karp – метод динамічного програмування з складністю $O(n^2 2^n)$. Практичний для $n \approx 20–25$;

– Branch and Bound – рекурсивний метод, що поєднує перебір з відсіканням гілок, які точно не містять оптимального рішення;

– моделі цілочислового лінійного програмування (ILP) – формулюють задачу у вигляді системи рівнянь та обмежень з цілими змінними;

– Concorde TSP Solver – вважається найшвидшим і найточнішим у світі серед доступних рішень для симетричної задачі TSP. Поєднує розгалуження, обрізання, евристики, лінійне програмування та локальний пошук.

Евристичні методи – ці методи не гарантують оптимального рішення, але дозволяють знайти досить хороше за прийнятний час:

– жадібні алгоритми (Greedy) – на кожному кроці обирається найближча ще не відвідана вершина;

– Nearest Neighbor, Farthest Insertion, Cheapest Insertion – прості евристики, що генерують маршрут шляхом локального прийняття рішень;

– алгоритми локального покращення (2-opt, 3-opt) – намагаються покращити наявний маршрут шляхом заміни ребер.

Метаевристики – використовуються для розв’язання задач великих розмірностей або тоді, коли не потрібно точне рішення:

– генетичні алгоритми – імітують природний добір, комбінуючи та мутуючи маршрути;

– мурашиний алгоритм (Ant Colony Optimization) – заснований на імітації поведінки мурашок;

– симульоване відпалу (Simulated Annealing) – алгоритм випадкового пошуку з поступовим зменшенням ймовірності прийняття гірших рішень;

– алгоритм рою частинок (Particle Swarm Optimization) – алгоритм глобального пошуку на основі кооперативного руху агентів.

### 1.3.2 Практичне значення наукових досліджень

У сучасних наукових і прикладних дослідженнях задача TSP використовується для:

– оптимізації логістики;
– маршрутизації роботів та дронів;
– планування доставки;
– мікросхемного трасування;
– автоматизованого планування завдань;
– туристичних маршрутів тощо.

Дослідження та порівняння методів вирішення задачі TSP також є важливими з точки зору навчання, так як дозволяють проілюструвати поведінку алгоритмів, їх ефективність, витрати ресурсів та адаптивність до змін вхідних даних.

## 1.4 Сучасні технології візуалізації у застосунках для мандрівників

У XXI столітті програмні засоби для планування подорожей повинні не лише ефективно обробляти вхідні дані та будувати маршрути, але й забезпечувати інтуїтивно зрозумілу та привабливу взаємодію з користувачем. Сучасний користувач очікує від туристичних застосунків швидкості, наочності, адаптивності та візуального комфорту. Це зумовлює потребу у використанні технологій інтерактивної візуалізації, анімації та гнучкої компонувальної структури інтерфейсу.

Створюючи застосунок розробник повинен аналізувати та застосовувати найновіші технології дизайну, інтерфейс повинен бути інтуїтивно зрозумілим, здатним масштабуватися. Користувач може навіть не дізнатися про унікальність і корисність продукту, якщо його відштовхне дизайн першої сторінки, тому необхідно звертати увагу на оформлення.

Карта є центральним елементом інтерфейсу більшості застосунків для планування подорожей. Вона не лише відображає локації, а й слугує інструментом взаємодії: користувач повинен мати змогу обирати точки, змінювати порядок відвідування, отримувати інформацію про об’єкти, а також бачити побудований маршрут у реальному часі. Така гнучкість візуалізації досягається завдяки використанню сучасних бібліотек і API, зокрема:

– Mapbox GL JS – JavaScript-бібліотека для побудови високопродуктивних інтерактивних карт з 3D-можливостями;

– Leaflet.js – легка бібліотека для роботи з OpenStreetMap і додавання маркерів, маршрутів та подій на мапі;

– OpenLayers – потужний інструмент для відображення картографічних даних з великою кількістю шарів і типів проєкцій.

Ці інструменти підтримують масштабування, кластери, теплові карти, маршрутизацію та адаптивність до екранів різного розміру.

## 1.5 Постановка задачі

Розробка застосунку для планування подорожей та оптимізації туристичних маршрутів є актуальним завданням у сучасному світі активного туризму, урбаністики та цифрової трансформації способів пересування. У зв’язку з цим ставиться завдання створення інтерактивного вебзастосунку, який дозволить користувачеві швидко й ефективно сформувати маршрут по туристичних локаціях міста з урахуванням часу, відстані та обраного алгоритму маршрутизації. Особливістю застосунка буде можливість обрати час перебування на кожній з локацій та отримати в результаті не лише час в дорозі але і час всієї подорожі з урахуванням витрачено часу на відвідування кафе та музеїв по дорозі. Застосунок має об’єднувати передові технології веброзробки, алгоритмічну оптимізацію та сучасний дизайн.

Об’єктом роботи є процес побудови маршруту між множиною туристичних об’єктів з однаковою початковою та кінцевою точкою.

Метою роботи є розробка програмного застосунку, який дозволяє користувачеві обрати точки для відвідування на інтерактивній карті міста, після чого будувати оптимальний туристичний маршрут між ними.

Необхідно створити вебзастосунок, що дозволить користувачеві:

– обирати свій готель як початкову та кінцеву точку маршруту;

– обирати туристичні локації міста;

– знаходити оптимальний маршрут;

– візуалізувати маршрут на мапі;

– зберігати або експортувати результати через QR-код або посилання у Google Maps.

Для досягнення мети необхідно вирішити такі завдання:

– дослідити алгоритми маршрутизації (Brute Force, Greedy, Held-Karp, Branch and Bound, Concorde), надати їх математичне обґрунтування та порівняння;

– визначити функціональні та нефункціональні вимоги до майбутнього застосунку;

– обґрунтувати вибір мов програмування, архітектури та бібліотек для реалізації системи;

– реалізувати клієнтську частину з інтерактивною мапою, пошуком локацій та виведенням маршрутів;

– реалізувати серверну частину з обробкою запитів користувача, запуском обраних алгоритмів маршрутизації та передачею даних назад;

– розробити приємний та інтуїтивно зрозумілий інтерфейс, у якому користувач зможе обирати точки маршруту, бачити приблизний час на подолання марштуру;

– протестувати працездатність реалізованої системи та порівняти ефективність алгоритмів на реальних сценаріях використання.

# 2 ЗАДАЧА КОМІВОЯЖЕРА ТА АЛГОРИТМИ ВИРІШЕННЯ

## 2.1 Формалізація задачі маршрутизації

Проблема побудови оптимального маршруту між множиною заданих точок є класичним прикладом комбінаторної задачі, що належить до NP-повних. У контексті туристичного планування така задача має практичну інтерпретацію: користувач обирає кілька туристичних локацій, які бажає відвідати, і хоче побудувати найкоротший маршрут, що починається і закінчується в готелі, проходячи через усі вибрані точки.

Формально ця задача зводиться до задачі комівояжера (TSP, Traveling Salesman Problem). У загальному вигляді вона формулюється наступним чином:

Дано:

– множина n міст (в нашому випадку – туристичних локацій);

– відстані між кожною парою міст *d(i, j)*, які задовольняють нерівність трикутника і можуть бути евклідовими або взятими з реальних картографічних даних.

Знайти: такий маршрут, що проходить через кожне місто рівно один раз, починається і закінчується в одній точці, та має мінімальну загальну довжину:

$$\min_{\sigma \in S_n} \left( d\left(\sigma(n), \sigma(1)\right) + \sum_{i=1}^{n-1} d\left(\sigma(i), \sigma(i+1)\right) \right), \qquad (2.1)$$

де $\sigma \in S_n$ – перестановка порядку відвідування міст.

Нижче наведено найкоротший шлях комівояжера через 15 міст Німеччини (рис. 2.1). Всього існує 43589145600 різних шляхів.


Рисунок 2.1 – Задача комівояжера на повнозв’язному графі туристичних точок

Задача TSP має надзвичайно широке практичне застосування – від логістики та маршрутизації транспорту до планування чипів у мікроелектроніці. Через високу обчислювальну складність точні методи рішення ефективні лише для невеликих розмірів n (до 10–12). Тому з’являється потреба в евристичних або комбінаторних підходах.

У цьому проєкті буде розглянуто декілька алгоритмів розв’язання задачі TSP:

– повний перебір (Brute Force);

– жадібний алгоритм (Greedy);

– динамічне програмування (алгоритм Хелда-Карпа);

– метод гілок і меж (Branch and Bound);

– Concorde TSP Solver – оптимізаційний пакет на основі лінійного програмування.

## 2.2 Порівняльна характеристика алгоритмів маршрутизації за часом та пам’яттю

Оскільки задача комівояжера є NP-повною, практичне її розв’язання передбачає пошук компромісу між точністю результату та ресурсами, які витрачає система. Існує кілька підходів до маршрутизації, що використовуються як у наукових дослідженнях, так і в прикладних системах – від повного перебору до евристичних та оптимізаційних алгоритмів.

Наведемо порівняльну характеристику базових алгоритмів маршрутизації (табл. 2.1), які планується реалізувати в програмному застосунку: Brute Force, Greedy, Held-Karp, Branch and Bound, Concorde. Оцінка проводиться на основі доступних емпіричних досліджень [1, 2, 3] і аналітичних оцінок складності.

Таблиця 2.1 – Порівняння алгоритмів маршрутизації

| **Алгоритм** | **Тип алгоритму** | **Часова складність** | **Пам’ять (складність)** | **Оптимальність результату** |
|---|---|---|---|---|
| Brute Force | Точний | $O(n!)$ | $O(n)$ | Гарантовано оптимальний |
| Greedy | Евристичний | $O(n^2)$ | $O(n)$ | Не гарантує оптимум |
| Held-Karp | Динамічне програмування | $O(n^2 2^n)$ | $O(n2^n)$ | Гарантовано оптимальний |
| Branch and Bound | Комбінаторний | Найгірше – $O(n!)$ | Залежить від реалізації | Гарантує оптимум при повному переборі |
| Concorde TSP | Оптимізаційний | Практично швидкий | Висока | Гарантовано оптимальний |

Коротко охарактеризуємо кожен з алгоритмів.

Brute Force дозволяє повний перебір всіх можливих перестановок, але експоненційне зростання часу обмежує його застосування розмірністю.

Greedy є швидким і простим у реалізації, проте може повертати не найкоротший шлях, особливо в асиметричних графах.

Алгоритм Хелда-Карпа є найбільш ефективним з точних методів, однак при  починає споживати занадто багато ресурсів. Час виконання роботи цього алгоритму значно кращий за метод повного перебору, порівняти складність цих алгоритмів можно на графіку (рис. 2.2).

Branch and Bound значно зменшує кількість перебраних варіантів завдяки відсіченню неефективних гілок, але продуктивність залежить від структури графа.

Concorde TSP Solver є промисловим стандартом і використовується у багатьох наукових роботах. Завдяки застосуванню лінійного програмування (LP) і гілок з відсіканням, він вирішує задачі з великого масштабу, але потребує потужних серверів і оптимізованої реалізації на C [25].

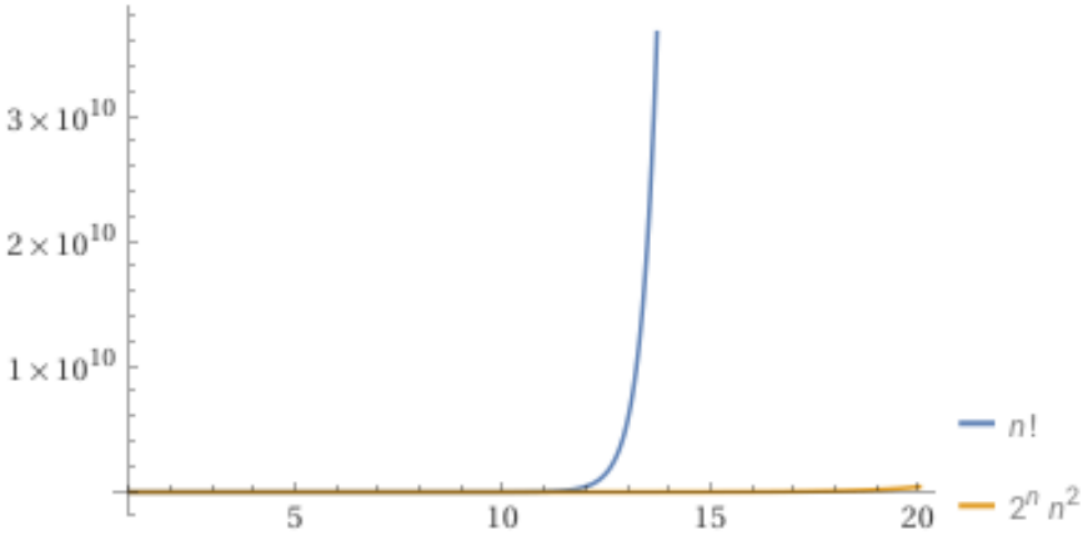



Рисунок 2.2 – Залежність часу виконання від кількості точок для Brute Force та Held-Karp

Під час тестування застосунку планується порівняти алгоритми на прикладі реальної мапи міста.

## 2.3 Математичний опис та реалізація алгоритмів

### 2.3.1 Метод перебору (Brute Force)

Алгоритм повного перебору (Brute Force) є найпростішим способом розв'язання задачі комівояжера. Його суть полягає в переборі всіх можливих перестановок точок, розрахунку довжини маршруту для кожної з них та виборі найкоротшого.

Нехай маємо множину з n локацій $\{u_1, u_2, \ldots, u_n\}$, де відстань між будь-якими двома локаціями позначається як $d(u_i, u_j)$. Задача полягає в тому, щоб знайти перестановку $\sigma$, яка мінімізує сумарну довжину маршруту:

$$D(\sigma) = d\left(u_{\sigma(n)}, u_{\sigma(1)}\right) + \sum_{i=1}^{n-1} d(u_{\sigma(i)}, u_{\sigma(i+1)}). \qquad (2.2)$$

Підхід Brute Force передбачає обчислення цієї суми для всіх перестановок (оскільки першу точку можна вважати фіксованою для зменшення симетрії).

Лістинг 2.1 Псевдокод алгоритму Brute Force:

```
best_distance = infinity
for each permutation p in permutations(points[1:]):
    route = [points[0]] + p + [points[0]]
    dist = compute_route_distance(route)
    if dist < best_distance:
        best_distance = dist
        best_route = route
```

Часова складність: $O(n!)$ – експоненційне зростання, непридатне для великих $n$.

Пам'ятна складність: $O(n)$ – лише один маршрут у пам'яті одночасно.

Переваги:

– гарантує знаходження глобального оптимуму;
– проста реалізація;
– зручна для демонстрації правильності рішень на малих графах.

Недоліки:

– не масштабується при збільшенні n;
– потребує багато часу для обчислення $n > 10$.

Приклад: Нижче зображено всі можливі маршрути для 4 точок, де найкоротший виділено червоною лінією (рис. 2.3).

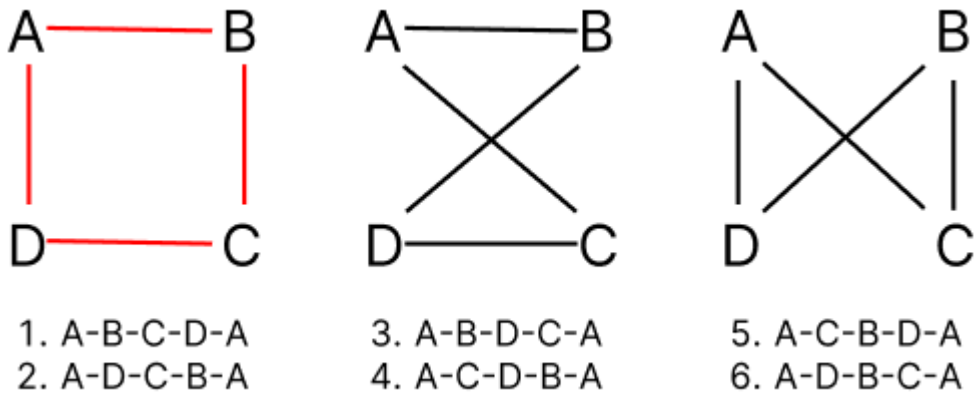



Рисунок 2.3 – Повний перебір усіх маршрутів для 4 точок

У програмній реалізації цей алгоритм використовується як базовий еталон для перевірки точності інших методів. Через обмеження у швидкодії, його застосування доцільне лише для малих підмножин туристичних локацій (до 8–10 точок).

### 2.3.2 Жадібний алгоритм (Greedy)

Жадібний алгоритм є одним з найпростіших евристичних підходів до розв’язання задачі комівояжера. Його суть полягає в тому, щоб на кожному кроці вибирати наступну найближчу ще не відвідану точку, не повертаючись

назад. Такий підхід не гарантує оптимального рішення, проте дозволяє швидко отримати задовільний результат при великих об’ємах вхідних даних.

Опис алгоритму:

Крок 1. Обрати початкову точку (наприклад, готель користувача).

Крок 2. Знайти найближчу до поточної точку, яка ще не була відвідана.

Крок 3. Перейти до неї та повторити Крок 2, поки не буде відвідано всі точки.

Крок 4. Повернутися у початкову точку, завершивши цикл.

Формально: Нехай *V = {*$u_1, u_2, \ldots, u_n$*}, d (i, j)* – відстань між локаціями. Алгоритм будує послідовність $\sigma = [u_1, u_{i2}, \ldots, u_{in}, u_1]$, де $u_1$ – початкова точка, а $u_{ik}$ – найближча невідвідана до попередньої.

Лістинг 2.2 Псевдокод жадібного алгоритму:

```
current = start
unvisited = set(points) - {start}
route = [start]
while unvisited:
    next = argmin([d(current, p) for p in unvisited])
    route.append(next)
    unvisited.remove(next)
    current = next
route.append(start)
```

Часова складність: *O(*$n^2$*)* – на кожному кроці виконується пошук серед *O(n)* точок.

Пам’ятна складність: *O(n)* – зберігається маршрут і множина невідвіданих.

Переваги:

– дуже швидкий навіть при великій кількості даних;

– простий у реалізації;

– добре працює на щільних евклідових графах.

Недоліки:

– може повертати далеко не оптимальний маршрут;

– не враховує глобальну структуру графа;

– залежить від вибору початкової точки.

Якщо взяти умовний граф ABCD та побудувати найкоротший маршрут за жадібним алгоритмом, то отримаємо сумарно:

$$sum(ABCDA) = 17 + 20 + 43 + 60 = 140.$$

Тоді як побудувавши маршрут за методом повного перебору отримаємо сумарно:

$$sum(ABCDA) = 17 + 40 + 43 + 18 = 118.$$

За допомогою графу продемонстровано приклад роботи жадібного алгоритму, де видно локально оптимальні кроки, що в підсумку не дають глобального оптимуму (рис. 2.4).

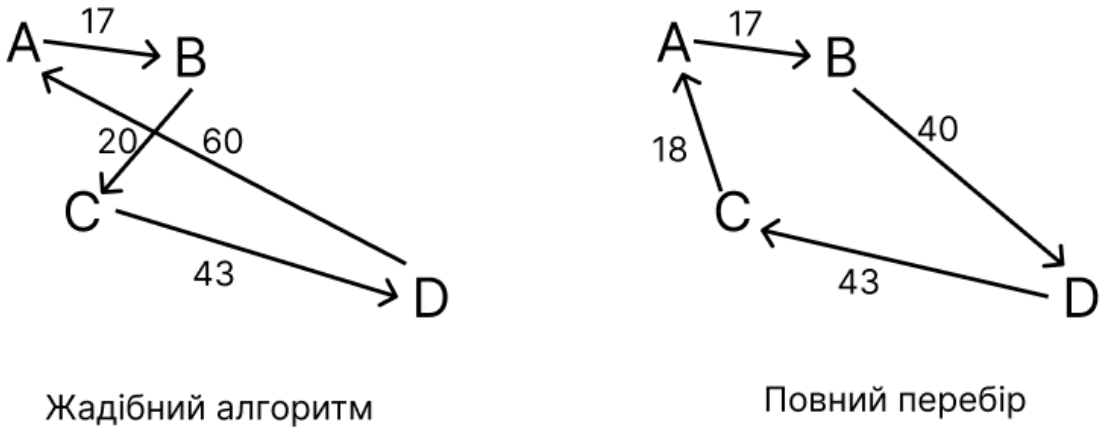



Рисунок 2.4 – Приклад маршруту, побудованого за жадібним алгоритмом та методом повного перебору

### 2.3.3 Алгоритм Хелда-Карпа

Алгоритм Хелда-Карпа є ефективним точним методом розв'язання задачі комівояжера на основі динамічного програмування. Його головна ідея полягає в збереженні проміжних результатів для підмножин міст, що значно знижує кількість необхідних обчислень у порівнянні з повним перебором.

Цей алгоритм був запропонований у 1962 році Майклом Хелдом і Річардом Карпом і залишається одним із найефективніших точних підходів для задач з розмірністю до 25–30 точок.

Математичне формулювання [26]:

Дано $G = (V, D)$, де $V = \{1, 2, \dots, n\}$ це перелік всіх міст, а D – це матриця найкоротших шляхів. При чому $\forall i, j \in V,\ i \neq j,\ d_{ij} > 0$

Необхідно знайти найкоротший маршрут, який проходить через усі $n$ міст і повертається в перше місто.

Рекурсивне вирішення задачі:

$$C(S,k) = \begin{cases} d_{1,m}, & \text{якщо } S = \{1,k\} \\ \min\limits_{m \neq k, m \in S} [C(S - \{k\}, m) + d(m,k)]\cdot \end{cases} \quad (2.3)$$

Часова складність:

$$(n-1)\sum_{k=1}^{n-3}\left(\frac{n-2}{k}\right) + 2(n-1) \sim O(n^2 \times 2^n) \ll O(n!). \quad (2.4)$$

Пам'ятна складність:

$$\sum_{k=1}^{n-1} k\left(\frac{n-1}{k}\right) = (n-1) \times 2^{n-2} \sim O(n \times 2^n). \quad (2.5)$$

Складність описана на основі практичних дослідів, але по факту варіативна для різних випадків.

Лістинг 2.3 Псевдокод алгоритму Хелд-Карпа [26]:

```
function algorithm TSP (G, n) is
    for k := 2 to n do
        g({k}, k) := d(1, k)
    end for

    for s := 2 to n−1 do
        for all S ⊆ {2, ..., n}, |S| = s do
            for all k ∈ S do
                g(S, k) := minm≠k,m∈S [g(S\{k}, m) + d(m, k)]
            end for
        end for
    end for

    opt := mink≠1 [g({2, 3, ..., n}, k) + d(k, 1)]
    return (opt)
end function
```

Переваги:

– значно ефективніший за повний перебір;
– гарантує оптимальне рішення;
– добре підходить для задач середнього розміру.

Недоліки:

– не масштабується при n > 25;
– складна реалізація для новачків;
– потребує ефективної структури даних для роботи з підмножинами.

У програмній реалізації алгоритм Хелда-Карпа буде використаний для демонстрації точного рішення з ефективнішою швидкодією, ніж Brute Force, при розмірності до 15 точок.

### 2.3.4 Метод гілок і меж (Branch and Bound)

Метод гілок і меж (Branch and Bound, BnB) – це класичний алгоритм комбінаторної оптимізації, який дозволяє знайти точне рішення задачі комівояжера шляхом систематичного дослідження простору можливих перестановок з відсіканням явно неефективних гілок. Основна ідея полягає в тому, щоб уникнути перебору всіх можливих маршрутів, якщо вже на якомусь етапі очевидно, що поточна частина маршруту не приведе до кращого результату.

Принцип роботи:

Крок 1. Побудова дерева рішень, де кожна вершина – частковий маршрут.

Крок 2. Для кожної гілки оцінюється нижня межа вартості завершення маршруту.

Крок 3. Якщо межа більша або дорівнює вартості вже знайденого кращого рішення, то ця гілка відсікається (не розглядається далі).

Крок 4. Алгоритм продовжується доти, доки не буде досліджено всі потенційно кращі маршрути.

Оцінка межі:

Найпоширеніший підхід – мінімізація вартості за допомогою редукції матриці відстаней:

Крок 1. з кожного рядка матриці віднімається мінімальне значення;

Крок 2. з кожного стовпця матриці віднімається мінімальне значення;

Крок 3. сума цих зменшень використовується як нижня межа поточного вузла.

Часова складність від $O(n^2)$ до $O(n!)$ залежно від ефективності відсікання;

Пам’ятна складність є високою, бо зберігається багато часткових рішень.

Лістинг 2.4 Узагальнений псевдокод алгоритму гілок і меж:

```
queue = PriorityQueue()
queue.put((lower_bound(start_node), start_node))
best_cost = infinity
while not queue.empty():
   cost, node = queue.get()
   if cost >= best_cost:
      continue
   if node is complete:
      best_cost = min(best_cost, cost)
   for child in expand(node):
      queue.put((lower_bound(child), child))
```

Матриця M називається зведеною, якщо в кожному її рядку та стовпчику є хоча б один нульовий елемент або весь рядок чи весь стовпець має значення ∞. Нехай M – матриця відстаней між 5 містами. Матрицю M можна скоротити наступним чином [27]:

$$M_{RowRed} = \{M_{ij} - \min\{M_{ij} | 1 \le j \le n, \text{та } M_{ij} < \infty\}\}. \quad (2.6)$$

Роздивимось наступну матрицю (рис. 2.5):

$$M = \begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

Рисунок 2.5 – Початкова матриця відстаней

Знайдемо мінімальні елементи кожного рядка матриці (рис. 2.6):

$$M = \begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix} \begin{matrix} \rightarrow 10 \\ \rightarrow 2 \\ \rightarrow 2 \\ \rightarrow 3 \\ \rightarrow 4 \end{matrix}$$

Рисунок 2.6 – Мінімальні елементи кожного рядка матриці

Віднімемо мінімальний елемент від кожного елементу рядка (рис. 2.7):

$$M_{RowRed} = \begin{bmatrix} \infty & 10 & 20 & 0 & 1 \\ 13 & \infty & 14 & 2 & 0 \\ 1 & 3 & \infty & 0 & 2 \\ 16 & 3 & 15 & \infty & 0 \\ 12 & 0 & 3 & 12 & \infty \end{bmatrix}$$

Рисунок 2.7 – Проміжний результат зменшеної матриці

Поточна вартість *(M) = 10 + 2 + 2 + 3 + 4 = 21*.

Матриця $M_{RowRed}$ є скороченою за рядками, але не за стовпцями. Матриця називається зведеною до стовпців, якщо у кожному її стовпці є хоча б один нульовий елемент або всі ∞ елементів.

Для того, щоб зменшити уся матрицю необхідно знайти найменший елемент кожного стовпця також.

$$M_{ColRed} = \{M_{ji} - \min\{M_{ji} | 1 \le j \le n, \text{та } M_{ji} < \infty\}\}. \quad (2.7)$$

Тепер знайдемо мінімальний елемент кожного стовпця (рис. 2.8):

$M_{RowRed} =$

| | | | | |
|---|---|---|---|---|
| ∞ | 10 | 20 | 0 | 1 |
| 13 | ∞ | 14 | 2 | 0 |
| 1 | 3 | ∞ | 0 | 2 |
| 16 | 3 | 15 | ∞ | 0 |
| 12 | 0 | 3 | 12 | ∞ |
| ↓ | ↓ | ↓ | ↓ | ↓ |
| 1 | 0 | 3 | 0 | 0 |

Рисунок 2.8 – Мінімальні елементи стовпців матриці

І віднімемо їх від кожного елементу стовпця (рис. 2.9):

$M_{ColRed} =$

| | | | | |
|---|---|---|---|---|
| ∞ | 10 | 17 | 0 | 1 |
| 12 | ∞ | 11 | 2 | 0 |
| 0 | 3 | ∞ | 0 | 2 |
| 15 | 3 | 12 | ∞ | 0 |
| 11 | 0 | 0 | 12 | ∞ |

Рисунок 2.9 – Проміжний результат зменшеної матриці

*Вартість скорочення рядка + вартість скорочення стовпця = (10 + 2 + 2 + 3 + 4) + (1 + 3) = 25.*

Це означає, що всі маршрути на графі мають довжину не менше 25. Це оптимальна вартість шляху.

Тепер необхідно передивитися усі варіанти розвитку подій, якщо з точки 1 пересунутись в точку 2, 3, 4 або 5 і далі пересуватись за найвигіднішим маршрутом (рис. 2.10).

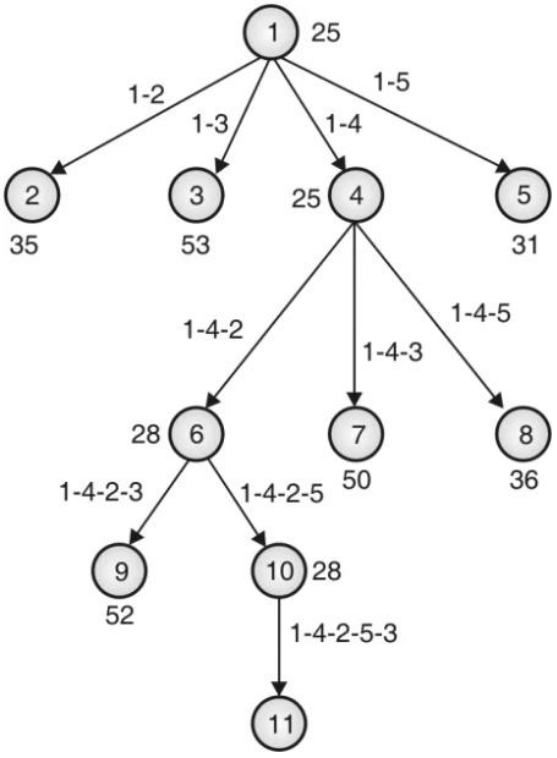



Рисунок 2.10 – Перебір найкращих варіацій маршруту

Таким чином ми з’ясували, що найкращим (найкоротшим) маршрутом буде маршрут 1-4-2-5-3-1.

Переваги алгоритму:

– гарантує знаходження глобального оптимуму;

– ефективніше за повний перебір за рахунок розумного відсікання;

– на практиці працює швидше, ніж Brute Force для середніх розмірів задачі.

Недоліки алготирму:

– залежить від якості функції нижньої межі;

– складний для реалізації без попереднього досвіду;

– при великому кількість вузлів усе одно може стати експоненційною.

У програмній реалізації метод гілок і меж буде використано для демонстрації стратегічного скорочення обчислень у точних алгоритмах. Це дозволить порівняти його з Brute Force та Хелдом-Карпом з точки зору ефективності та складності реалізації.

### 2.3.5 Алгоритм Concorde: структура, ефективність, особливості реалізації

Concorde TSP Solver є однією з найвідоміших і найпотужніших реалізацій точного розв’язання задачі комівояжера. Його було створено командою дослідників – Девідом Апплегейтом, Робертом Біксбі, Васеком Хваталом та Вільямом Куком – для демонстрації практичної розв’язуваності навіть великих інстансів TSP. У 2006 році на базі цього алгоритму було знайдено оптимальний маршрут для 85 900 точок [28].

Concorde не покладається лише на один метод, а реалізує комбінований підхід, що включає:

– розв’язання задачі цілочислового програмування (ILP);

– LP-релаксації (лінійне програмування зі зняттям цілочислових обмежень);

– гілок і меж;

– edge elimination – попередню обробку для зменшення простору пошуку;

– евристики на кшталт Lin-Kernighan для пошуку гарного стартового рішення;

– ефективну реалізацію на C з прямим доступом до пам’яті та структур низького рівня.

Архітектура Concorde:

Крок 1. Попередня обробка: редукція графа шляхом відсікання ребер з високою вагою.

Крок 2. Побудова LP-релаксації: для наближеного оцінювання вартості.

Крок 3. Запуск евристик: для отримання первинного наближення маршруту.

Крок 4. Використання методу гілок і меж: для підтвердження оптимальності.

Крок 5. Постпроцесинг: валідація та запис результатів.

Переваги:

– є одним із найефективніших точних TSP-розв’язувачів у світі;

– дозволяє працювати з тисячами точок;

– використовується в наукових дослідженнях, логістиці, геоінформатиці.

Недоліки:

– складність компіляції (Linux або Cygwin-середовище);

– складна інтеграція в сучасні клієнт-серверні додатки;

– відсутність API для взаємодії в реальному часі.

У межах програмної реалізації алгоритм Concorde використовується як зовнішній інструмент через командний рядок.

# 3 РОЗРОБКА ТА ТЕСТУВАННЯ ЗАСТОСУНКУ ДЛЯ ПЛАНУВАННЯ ПОДОРОЖЕЙ

## 3.1 Специфікація вимог до застосунку

Напередодні створення будь-чого треба визначити мету і вимоги до результату, для того, щоб як можна точніше завчасно спланувати і отримати наприкінці результат, який задовільнить усі виставленні умови і виправдає очікування. Саме тому перед початком планування розробки застосунка треба сформувати чітке уявлення про кінцевий результат. Для цього буде сформовано та описано перелік функцій, які сервіс повинен надавати, а також функціональні та нефункціональні вимоги до вебзастосунку для організації подорожей.

Створимо діаграму варіантів використання (рис. 3.1), що демонструє основні дії користувача під час взаємодії із застосунком:

– створення подорожі;
– вибір готелю та туристичних точок на мапі міста;
– вибір алгоритму і сортування за алгоритмом;
– формування маршруту та його експорт;

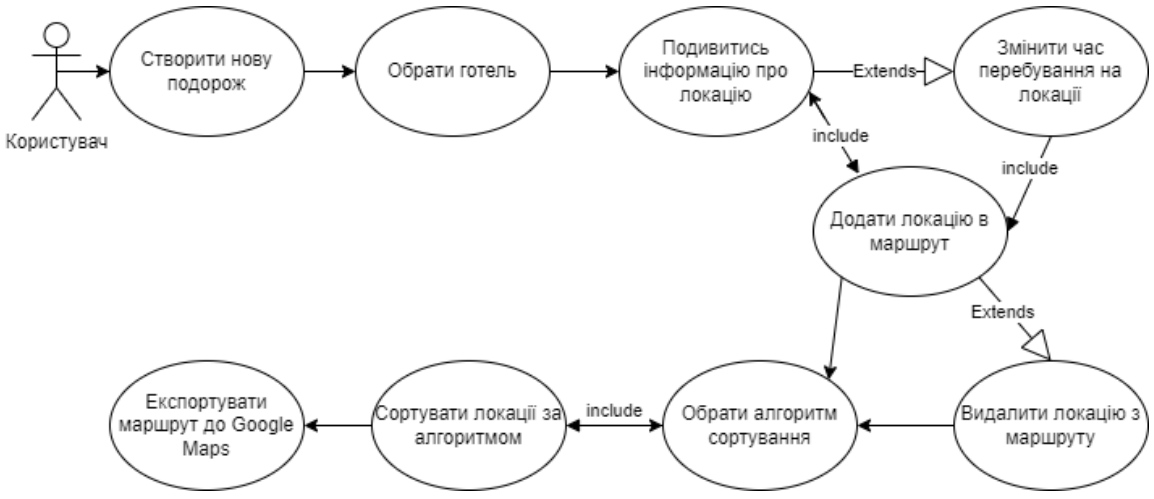



Рисунок 3.1 – Діаграма варіантів використання застосунку для планування подорожей

На підставі проведеного аналізу функціоналу аналогічних сервісів та згідно цілей кваліфікаційної роботи, було визначено наступні системні, функціональні та нефункціональні вимоги.

Системні вимоги:

– браузер з підтримкою JavaScript (рекомендовано Chrome, Firefox, Edge);

– сервер із Python ≥ 3.10, FastAPI, pandas, NumPy;

– клієнтська частина: Node.js ≥ 18, React ≥ 18, Tailwind CSS ≥ 4.

Функціональні вимоги:

– підтримка вибору початкової та кінцевої точки маршруту (готель);

– можливість встановити орієнтовний час перебування на локації;

– можливість вибору до 50 туристичних локацій;

– наявність пошуку з автозаповненням за назвою локації;

– побудова маршруту за одним з обраних алгоритмів (Brute Force, Greedy, Held-Karp, Branch and Bound або Concorde);

– відображення маршруту на мапі;

– в повідомленні про побудований маршрут повинно бути вказано орієнтовний час в дорозі, =орієнтовний час разом з перебуванням на локаціях та час, витрачений на опрацювання алгоритму;

– збереження результату у вигляді посилання на Google Maps.

Нефункціональні вимоги:

– інтерфейс має бути інтуїтивно зрозумілим для користувача без потреби в інструкціях;

– маршрут можливо побудувати лише для 2 і більше різних точок;

– інтерфейс повинен реагувати до 200 мс на кліки (перегляд та додавання/видалення локації);

– система повинна дозволяти додавати алгоритми, змінювати локації та міста без суттєвих змін в архітектурі;

– при помилках побудови маршруту повинні з’являтись зрозумілі повідомлення з пропозицією побудувати заново.

## 3.2 Обґрунтування вибору технологій

### 3.2.1 Мови програмування та середовище для розробки

Перед початком створення вебзастосунку необхідно обрати відповідний технологічний стек, що забезпечить оптимальну продуктивність, гнучкість і зручність у подальшій підтримці проєкту. Для реалізації цього проєкта було прийнято рішення поділити систему на дві частини – клієнтську та серверну.

Клієнтська частина реалізована з використанням React.js, однієї з найпопулярніших бібліотек для створення односторінкових застосунків (SPA). React (також відомий як React.js або ReactJS) – це безкоштовна бібліотека JavaScript з відкритим вихідним кодом, яка має на меті зробити побудову користувацьких інтерфейсів на основі компонентів більш «безшовною». Вона підтримується компанією Meta (раніше Facebook) та спільнотою індивідуальних розробників і компаній [29]. React дозволяє будувати інтерфейс на основі компонентного підходу, що спрощує масштабування, повторне використання коду та забезпечує високу реактивність і швидкість відображення змін у DOM-дереві без повного перерендеру сторінки. Додатково, підтримка бібліотек, таких як React Router (для маршрутизації) та React Query або Zustand (для управління станом), дозволяє гнучко будувати логіку та структуру проєкту.

Серверна частина створена з використанням механізмів Python, зокрема FastAPI – сучасного високопродуктивного вебфреймворку для розробки REST API. Серед переваг FastAPI – автоматична генерація документації, підтримка асинхронності та висока продуктивність, порівняно з Node.js та Go. Оскільки більшість алгоритмів маршрутизації, таких як Held-Karp, Brute Force або метод гілок і меж, потребують значних обчислювальних ресурсів і зручної математичної реалізації, Python є природним вибором завдяки своїй математичній бібліотеці NumPy, інструментам Matplotlib та підтримці великої кількості алгоритмічних фреймворків [30].

Середовищем розробки для фронтенду було обрано WebStorm, яке забезпечує розширені можливості автодоповнення, перевірки типів та інтеграції з Git. Для бекенду використовується PyCharm, завдяки його простоті, великій кількості розширень для Python та вбудованої підтримки систем контролю версій.

Для наочності наведено порівняльну таблицю найпоширеніших мов програмування для розробки фронтенду вебзастосунків (табл. 3.1).

Таблиця 3.1 – Порівняння мов програмування для фронтенду

| **Мова** | **Підтримка UI-фреймворків** | **Простота навчання** | **Популярність** | **Спільнота** |
|---|---|---|---|---|
| JavaScript | React, Vue, Angular | Висока | Дуже висока | Дуже активна |
| TypeScript | React, Angular | Середня | Висока | Активна |
| Dart | Flutter Web | Середня | Помірна | Менш активна |

Далі розглянемо порівняння популярних мов для бекенду (табл. 3.2).

Таблиця 3.2 – Порівняння мов програмування для бекенду

| **Мова** | **Продуктивність** | **Бібліотеки для обчислень** | **Простота інтеграції з API** | **Простота реалізації алгоритмів** |
|---|---|---|---|---|
| Python | Середня | Багатий вибір | Висока | Висока |
| Java | Висока | Помірна кількість | Висока | Середня |
| C++ | Дуже висока | Обмежена | Низька | Висока |
| JavaScript | Середня | Обмежена | Висока | Низька |

### 3.2.2 Методи зберігання та опрацювання інформації

Зберігання інформації про туристичні локації, маршрути, час відвідування та інші метадані можна реалізувати за допомогою різних підходів: у вигляді статичного JSON-файлу або у вигляді повноцінної бази даних (SQL чи NoSQL). Проведемо детальний порівняльний аналіз обох рішень з точки зору продуктивності, гнучкості, складності розгортання та підтримки масштабування (табл. 3.3).

JSON-файл – це простий текстовий формат, що дозволяє структурувати дані у вигляді вкладених об’єктів і масивів. Він ідеально підходить для невеликих проєктів, MVP та офлайн-рішень.

Бази даних – це системи, що дозволяють виконувати запити, фільтрацію, зв’язки між сутностями, збереження змін та багато іншого.

Таблиця 3.3 – Порівняння зберігання у JSON-файлі та базі даних

| **Критерій** | **JSON-файл** | **База даних(SQL/NoSQL)** |
|---|---|---|
| Простота налаштування | Дуже проста | Складніша |
| Продуктивність при малих обсягах | Висока | Висока |
| Масштабованість | Обмежена | Висока |
| Можливість оновлення даних | Вручну або повністю | Запити можна оновлювати частково |
| Підтримка складних запитів | Немає | Є ( SQL або агрегування) |
| Паралельний доступ | Обмежений | Повна підтримка транзакцій |
| Залежність від серверної частини | Не обов’язкова | Обов’язкова |
| Ускладнення для розробника | Мінімальне | Вищий поріг входу |

### 3.2.3 Технології побудови маршруту на реальній мапі

Побудова маршруту між туристичними точками на реальній мапі є центральним функціональним елементом застосунку. На відміну від теоретичної задачі комівояжера, яка базується на евклідових відстанях або довільних графах, реальні мапи мають складну структуру доріг, обмежень руху, односторонніх вулиць тощо. Це потребує використання картографічних сервісів, які здатні побудувати маршрут відповідно до географічних координат та існуючої дорожньої мережі.

Для реалізації побудови маршруту було обрано (табл. 3.4) використання картографічного API Mapbox Directions, який базується на рушії OSRM (Open Source Routing Machine).

Перевагою такого підходу є те, що маршрут не просто з'єднує точки прямими лініями, а враховує реальну дорожню інфраструктуру міста. Це створює реалістичний досвід для користувача та дозволяє точно оцінити час подорожі між об'єктами. Безкоштовного план Mapbox накладає обмеження на кількість локацій, що можуть бути одночасно маршрутизовані. Це питання вирішується шляхом попередньої оптимізації порядку відвідування точок одним із алгоритмів TSP.

Таблиця 3.4 – Порівняння методів побудови маршруту

| **Метод** | **Реалістичність** | **Швидкість реалізації** | **Залежність від серверу** | **Обмеження** |
|---|---|---|---|---|
| Пряма лінія (Haversine) | Низька | Дуже висока | Ні | Немає |
| Граф без карт(ручний) | Середня | Середня | Ні | Немає |
| Mapbox Directions | Висока | Висока | Так | 25 точок |

### 3.2.4 Методи зберігання користувачем інформації про маршрут

Для зручності і практичності застосування необхідно було додати функцію збереження маршруту для подальшого використання. Так як в цьому проєкті найбільше уваги звернено на алгоритми побудови маршруту і він не представляє собою повноцінного застосунку з багатим функціоналом для повсякденного використання, то не планується реалізовувати реєстрацію і аутентифікацію, відповідно маршрути неможливо прив’язати до акаунту, також обмеже використання на смартфонах, які частіше за все використовують при подорожах і немає опції використовувати застосунок без інтернету. Саме тому буде запропоновано використовувати застосунок, знаходячись в готелі або ще вдома і планувати подорожі завчасно, а потім зберегти посилання на маршрут і використати його в більш адаптованому сервісі Google Maps. Порівняймо цей та інші можливі підходи до збереження маршруту користувачем (табл. 3.5).

Таблиця 3.5 – Порівняння методів зберігання маршрутів

| **Метод** | **Зрозумілість для користувача** | **Підтримка мобільних** | **Зовнішні залежності** | **Можливість відкриття маршруту** |
|---|---|---|---|---|
| Google Maps URL | Дуже висока | Так | Ні | Так |
| QR – код (Google URL) | Висока | Так | Ні | Так |
| Збереження у базу | Низька (потребує інтерфейсу) | Ні | Так (сервер) | Ні (без клієнта) |
| Експорт у GPX/KML | Середня | Частково | Так (парсинг) | Так (але не інтерактивно) |

## 3.3 Структура клієнт-серверної взаємодії

### 3.3.1 Основні компоненти системи

Структура застосунку реалізована за принципом клієнт-серверної архітектури, яка забезпечує гнучкий розподіл функціональних обов’язків, масштабованість та безпечний обмін даними між частинами системи.

Клієнтська частина реалізована як односторінковий застосунок (SPA) з використанням бібліотеки React, що дозволяє користувачеві взаємодіяти з картою, шукати локації, запускати алгоритми оптимізації та переглядати результати. Клієнт здійснює HTTP-запити до серверної частини через REST API.

Серверна частина розроблена з використанням FastAPI – сучасного високошвидкісного вебфреймворку для Python. Сервер відповідає за обробку запитів з фронтенду, виконання алгоритмів маршрутизації, зчитування та обробку файлів, в яких зберігається інформація про локації, а також повернення результатів (маршрутів, статистики тощо).

Основні компоненти системи (табл. 3.6):

Frontend (React):

– відображення інтерактивної карти (Mapbox/Leaflet);

– компоненти для вибору точок, перегляду маршрутів;

– панель керування та інтерфейс порівняння результатів.

Backend (FastAPI):

– обробка запитів з клієнта;

– реалізація маршрутних алгоритмів (Brute Force, Held-Karp, Greedy тощо);

– вимірювання часу/пам’яті виконання.

Зовнішні сервіси:

– OpenStreetMap через Mapbox/Leaflet (візуалізація карти);

– Google Maps API (опційне зовнішнє посилання для відкриття маршруту).

Таблиця 3.6 – Відповідність функціоналу та компонентів системи

| № | Компонент | Технологія | Функція |
|---|---|---|---|
| 1 | UI/Карта | React+Mapbox/Leaflet | Вибір точок, перегляд маршруту |
| 2 | Пошук/Вибір точок | React, JSON | Автозаповнення, фільтрація |
| 3 | API-запит | Fetch/Axios | Надсилання параметрів маршруту на сервер |
| 4 | Сервер | FastAPI | Виконання обчислень, логіка оптимізації |
| 5 | JSON-файл | Python (json) | Зчитування локацій, зберігання інформації |
| 6 | QR/Google Maps | qrcode, Google URL | Генерація посилань і кодів для поділитись результатами |

Такий розподіл дозволяє легко модифікувати окремі частини системи, зокрема:

– розширювати набір алгоритмів на сервері;

– адаптувати UI без зміни логіки обчислень;

– замінити джерело даних або формат обміну.

Це забезпечує масштабованість та зручність супроводу проєкту у майбутньому.

### 3.3.2 Формат зберігання даних

Для забезпечення ефективної роботи застосунку з туристичними об’єктами та маршрутами було обрано формат зберігання у вигляді структурованого JSON-файлу. Це рішення обумовлено простотою інтеграції з JavaScript-клієнтом, відсутністю потреби у сервері баз даних на початковому етапі та легкістю обробки масивів даних без складних запитів.

JSON файли bratislava_places.json та bratislava_hotels.json містять масив об’єктів, кожен з яких описує окрему туристичну точку. Нижче надано приклад структури запису, представленого у вигляді ключ-значення з наступними атрибутами.

Лістинг 3.1 приклад структури опису туристичних об’єктів:

```
{
   "id": 1,
   "name": "Bratislava Castle",
   "category": "landmark",
   "duration": 90,
   "lat": 48.142109,
   "lon": 17.100235
  }
```

Ця структура дозволяє застосунку здійснювати фільтрацію за категоріями або за ключовими словами, формувати маршрути з урахуванням часу перебування та будувати маршрут між обраними координатами.

Для зберігання маршруту використовується окремий об’єкт, який створюється динамічно після оптимізації.

Лістинг 3.2 приклад структури збереженого маршруту:

```
{
  "start_hotel": 5,
  "route_order": [12, 4, 17, 9],
  "total_distance_km": 6.3,
  "total_duration_min": 85
},
```

Дане рішення є оптимальним для MVP-застосунку, де обсяг даних невеликий і змінюється нечасто. У майбутньому структура даних може бути перенесена до повноцінної бази даних (наприклад, PostgreSQL або MongoDB) із можливістю додавання/редагування записів через адміністративну панель.

### 3.3.3 Взаємодія між користувачем, клієнтом на сервером

Типовий сценарій користувача має такий вигляд:

– користувач відкриває застосунок або створює нову подорож;

– користувач вводить назву локації в пошуку і клієнт надсилає запит /search_locations або обирає точку на карті кліком;

– обирає час перебування на локації і додає точку до маршруту;

– після додавання стартової та фінішної точки (готелю), а також хоча б ще одну іншу користувач обирає алгоритм і запускає побудову маршруту;

– клієнт надсилає POST-запит /solve-route з масивом об’єктів { id, lat, lon, duration };

– сервер обробляє запит, використовуючи функцію solve_route з модуля solver.py, що викликає один з алгоритмів (Brute Force, Greedy тощо);

– у відповідь клієнт отримує обчислений маршрут, відображає його на мапі і формує посилання на Google Maps та QR-код.

Клієнт приймає вхідні дані від користувача і передає їх серверу, а сервер в свою чергу формує відповідь клієнту для передачі інформації користувачеві.

Інформація, що передається клієнтом уявляє собою масив точок маршруту з полями:

– ідентифікаційний номер локації *(id);*

– географічну широту геолокації *(lat);*

– географічну довготу геолокації *(lon);*

– час перебування на локації *(duration);*

– і обраний алгоритм (*algorithm*).

А так виглядає інформація, що повертається у відповіді сервера:

Лістинг 3.3 HTTP – відповідь від сервера клієнту:

*{*
*path: [[lat, lon], ...]*
*segments: [{from, to, distance_km}, ...]*
*computation_time_sec*
*total_distance_km*
*total_time_with_stops_min*
*}*

Детальніше взаємодію користувач – клієнт – сервер можна розглянути на відповідній діаграмі (рис. 3.2).

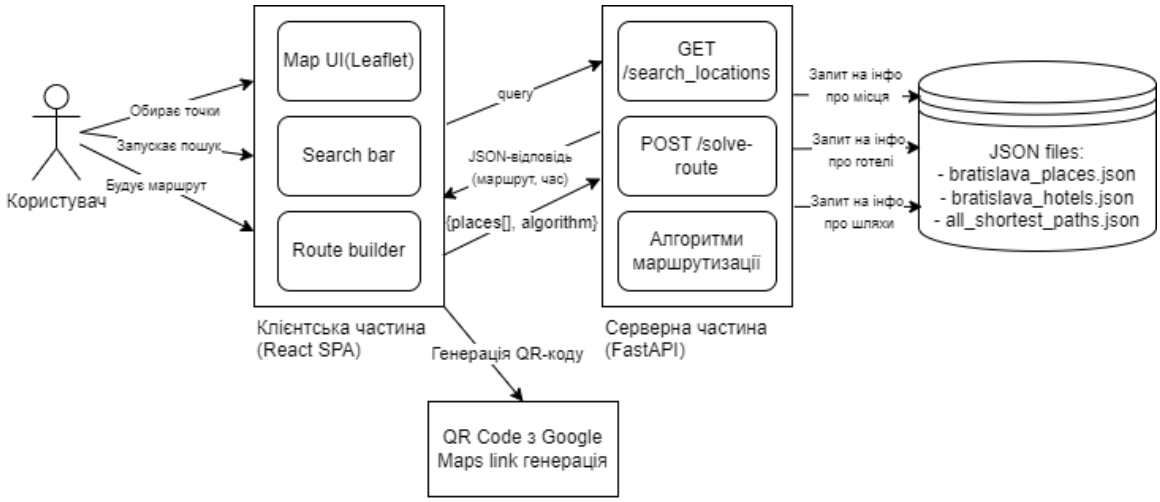



Рисунок 3.2 – Діаграма взаємодії між користувачем, клієнтом та сервером

Така організація взаємодії забезпечує чітке розмежування відповідальності між клієнтом і сервером, що сприяє модульності та дозволяє легко розширювати функціонал у майбутньому без необхідності зміни структури запитів. Завдяки цьому застосунок може масштабуватися та інтегрувати нові джерела даних і алгоритми маршрутизації.

## 3.4 Реалізація клієнтської частини

Клієнтська частина застосунку для планування подорожей реалізована у вигляді односторінкового вебінтерфейсу з використанням бібліотеки React, що дозволяє будувати багатокомпонентні інтерфейси з високим рівнем інтерактивності та повторним використанням логіки відображення. Уся логіка взаємодії з серверною частиною, обробка даних, а також відображення інформації на карті відбуваються без перезавантаження сторінки, що значно покращує користувацький досвід.

Основною структурною одиницею React-застосунку є компоненти. Залежно від функціоналу, що реалізується, компоненти поділяються на функціональні (без стану) та класові або з використанням хуків (useState, useEffect, useRef). У межах застосунку створено окремі компоненти для:

– відображення мапи з маркерами (MapComponent);
– пошукової панелі з автозаповненням (SearchBar);
– списку обраних локацій (SelectedPlacesList);
– параметрів оптимізації маршруту (AlgorithmSelector);
– результатів обчислень маршруту та його візуалізації (RouteInfo, RouteMapDrawer).

Для реалізації карти було використано React-Leaflet, що забезпечує гнучке API для інтеграції OpenStreetMap у компоненти React. Користувач може додавати точки маршруту кліком по карті або обираючи з пошукового поля. Усі точки автоматично зберігаються в стані застосунку та візуально позначаються маркерами.

Пошук реалізовано із підтримкою автозаповнення, що обробляється на клієнтській стороні за допомогою методу фільтрації по bratislava_places.json. При введенні запиту, відображається список результатів, які відповідають частковому збігу за назвою. Карта автоматично центрує відповідну координату з обраною локацією.

Передача даних до серверної частини здійснюється через HTTP-запити бібліотекою Axios. Усі запити виконуються асинхронно, а результат обробки маршруту (оптимізований порядок відвідування точок, довжина маршруту, час обчислення) отримується у форматі JSON.

Для візуалізації маршруту використовується бібліотека leaflet-routing-machine, яка дозволяє побудувати маршрут за набором точок з урахуванням дорожньої мережі. Результат маршруту зчитується у вигляді координат, що використовуються для побудови полігону (Polyline) на карті. Окремі стилі відображення полігонів залежать від обраного алгоритму, що дозволяє користувачу візуально порівняти різні варіанти маршруту.

У підсумку, клієнтська частина застосунку є гнучкою, динамічною та зручною у використанні. Реалізація на базі React забезпечує високу продуктивність, а обрані бібліотеки дозволяють інтегрувати картографічний функціонал та ефективно відображати результати алгоритмічної обробки маршрутів.

### 3.4.1 Проєктування дизайну застосунку

Першим кроком розробки клієнтської частини стала побудова макету користувацького інтерфейсу в інструменті Figma. Це дозволило візуалізувати логіку роботи застосунку ще до написання коду, а також забезпечити узгодженість стилів, кольорової палітри та адаптивності.

Застосунок має наступні основні інтерфейсні екрани:

– головна сторінка з мапою і віконця з інформацією про кожну локацію (рис. 3.3);

– вікно, де формується список доданих користувачем точок (рис. 3.4);

– вікно з результатом про побудований маршрут (рис. 3.5);

– вікно з QR-кодом для збереження або відкриття маршруту.

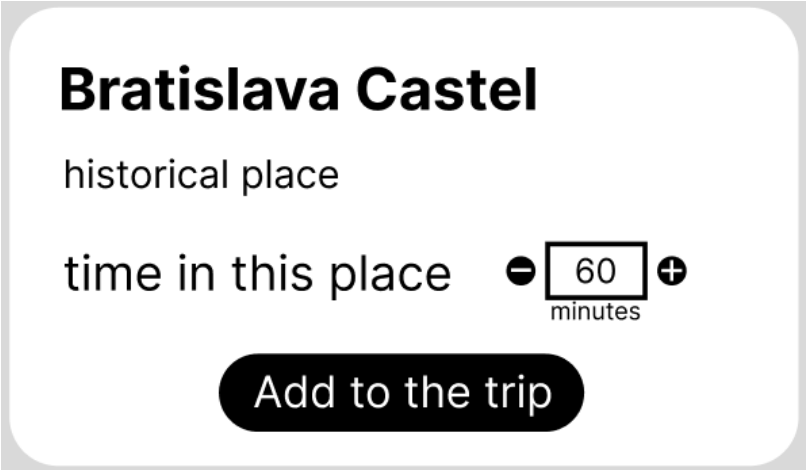



Рисунок 3.3 – Дизайн-макет віконця з інформацією про локацію

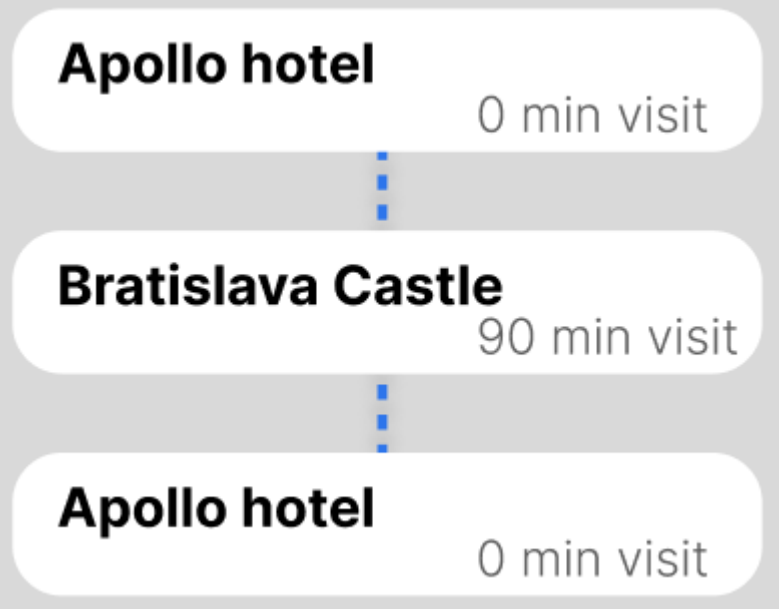



Рисунок 3.4 – Дизайн-макет віконця зі списком доданих в маршрут точок

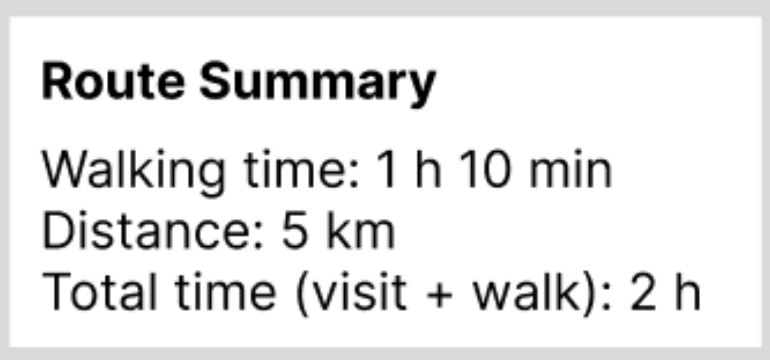



Рисунок 3.5 – Дизайн-макет віконця з результатом про побудований маршрут

Ключовими принципами, які використовувалися при створенні макету в Figma, стали:

– мінімалізм – залишено лише найважливіші елементи;

– навігаційна логіка – всі дії користувача мають інтуїтивно зрозуміле розташування;

– адаптивність – елементи автоматично масштабуються залежно від розміру вікна.

Для того, щоб зобразити локації, було створено 6 іконок для різних категорій локацій (рис. 3.6), також було перефарбовано ці іконки в синій колір для того, щоб відрізняти локації, що вже додані в маршрут.


Рисунок 3.6 – Іконки локацій за категоріями

### 3.4.2 Впровадження Tailwind CSS та інтеграція OpenStreetMap

Після затвердження макету було розпочато реалізацію інтерфейсу за допомогою React. Для стилізації застосовано Tailwind CSS 4, що дозволило швидко створювати адаптивні компоненти без надмірного CSS-коду. Кожен візуальний компонент був реалізований як окремий React-компонент. Для

роботи з мапою було використано бібліотеку React-Leaflet, яка забезпечує повноцінну інтеграцію OpenStreetMap.

Додаткові бібліотеки, що застосовувалися:

– *leaflet-geosearch* – для реалізації пошуку місць на мапі;

– *react-qr-code* – для генерації QR-коду;

– *react-toastify* – для виведення повідомлень про помилки чи успішні дії;

– *axios* – для надсилання запитів на сервер з обчисленням маршруту.

Особливу увагу було приділено адаптивності – інтерфейс (рис. 3.7) виглядає коректно на екранах різних розмірів. Для цього активно використовувалися медіа-запити Tailwind та компоненти, що автоматично змінюють розташування елементів на малих екранах.

Навігація між різними секціями здійснюється без перезавантаження сторінки завдяки використанню React Router, а глобальний стан (наприклад, обрані локації) зберігається за допомогою React Context.

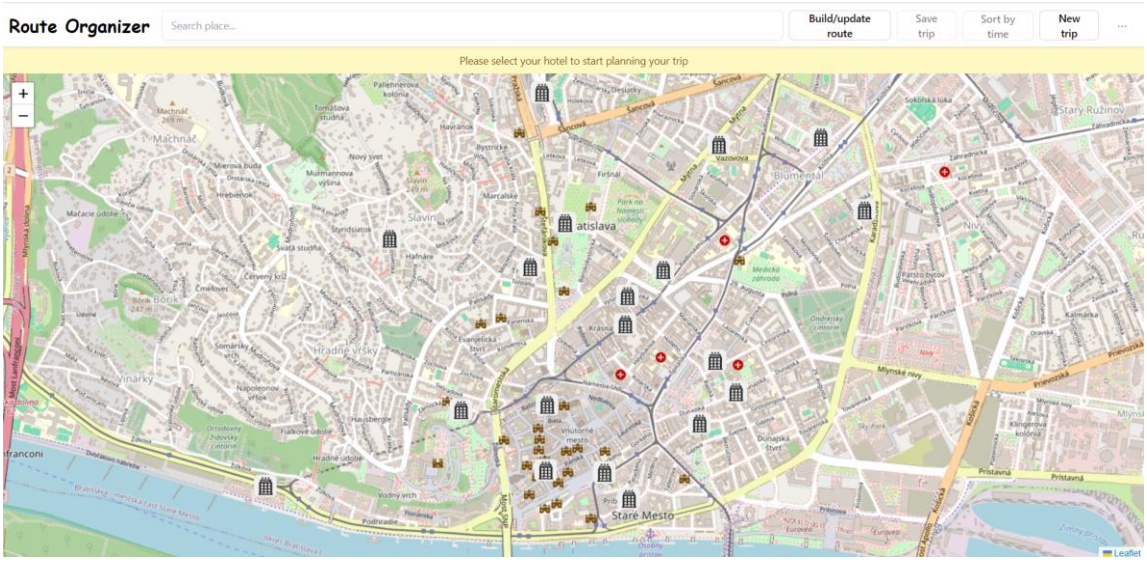



Рисунок 3.7 – Реалізований інтерфейс головної сторінки у браузері

В результаті формування маршруту користувач отримує інформацію про маршрут (рис. 3.8) з часом та відстанню, а також часом опрацювання запиту та сам маршрут, графічно зображений на мапі (рис. 3.9)

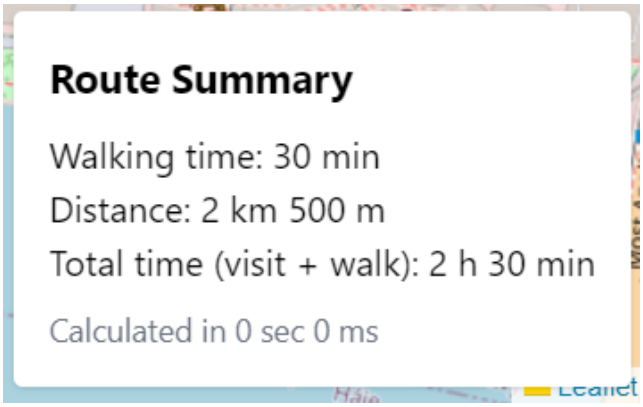



Рисунок 3.8 – Отримана інформація про побудований маршрут

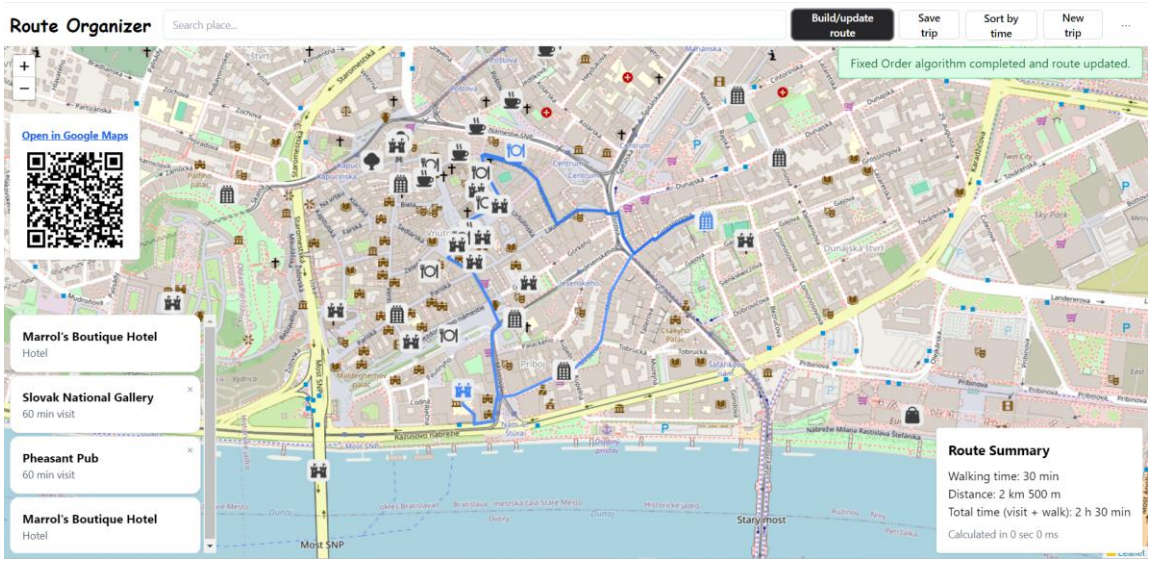



Рисунок 3.9 – Зображення побудованого маршруту на мапі

### 3.4.3 Пошукова система локацій

У системі реалізовано функціональність пошуку туристичних локацій, що є важливим елементом зручності для користувача. Відповідний інтерфейс розміщено у верхній панелі програми у вигляді текстового поля введення (рис. 3.10). Після введення перших символів назви локації користувачеві автоматично пропонуються найбільш релевантні варіанти з бази даних.


Рисунок 3.10 – Форма пошуку з автозаповненням

Автозаповнення реалізовано на клієнтській стороні за допомогою асинхронного запиту до серверного API /search_locations, який повертає список не більше ніж з 10 релевантних результатів. Алгоритм пошуку включає кілька етапів:

– нормалізація вхідного рядка (переведення до нижнього регістру, видалення діакритики для уніфікації латинських і словацьких символів);

– пошук локацій, де перше слово в назві починається з введеного рядка (найвища релевантність);

– пошук локацій, де інші слова в назві починаються з введеного рядка (середня релевантність).

Результати сортуються за рівнем відповідності, і користувач може клікнути на будь-який із запропонованих варіантів. При кліку на локацію відбувається наступне:

– текст пошукового поля заповнюється назвою вибраної локації;

– карта масштабується і переміщується так, щоб обрана точка була в центрі екрану;

– відкривається вікно з інформацією про локацію, з можливістю додавання її до маршруту.

Для зберігання локацій використано локальні JSON-файли bratislava_places.json та bratislava_hotels.json. Сервер обробляє запити пошуку, не змінюючи ці файли, що спрощує масштабування системи та додає гнучкість у роботі з даними.

### 3.4.4 Експорт маршруту до Google Maps

Однією з важливих функцій застосунку для планування подорожей є можливість збереження побудованого маршруту з подальшим його відтворенням або відкриттям у сторонньому сервісі. Це дозволяє користувачу не тільки зафіксувати оптимізований маршрут, але й поділитися ним з іншими або швидко переглянути на мобільному пристрої. У рамках проєкту реалізовано два методи збереження маршруту: у вигляді зовнішнього посилання Google Maps та QR-коду, що містить це посилання.

Формування посилання відбувається на основі масиву координат, які були оптимізовані одним із алгоритмів TSP. Відповідно до специфікації Google Maps URL [31], можливо створити запит у форматі:

*https://www.google.com/maps/dir/{lat1},{lon1}/{lat2},{lon2}/.../{lat},{lon}*

Для зручного переносу посилання на мобільний пристрій реалізовано генерацію QR-коду. Створення коду здійснюється за допомогою бібліотеки qrcode для Python, яка перетворює текстову URL-строку у графічне зображення. На клієнтській стороні воно виводиться поруч із результатом маршрутизації. Таким чином, користувач може відсканувати код (рис. 3.11) зі свого екрану та миттєво відкрити маршрут на смартфоні (рис. 3.12).


Рисунок 3.11 – Приклад QR-коду для збереження посилання на маршрут в Google Maps

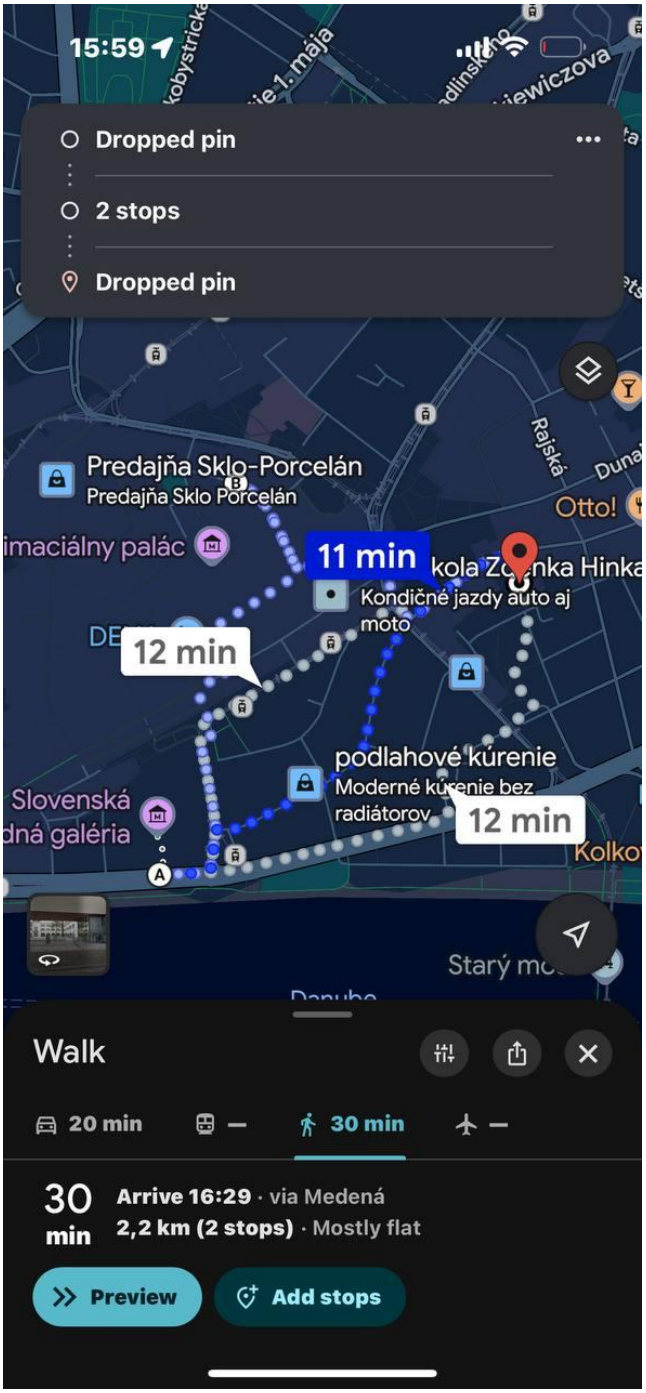



Рисунок 3.12 – Результат переходу в Google Maps

Якщо порівняти побудований маршрут в застосунку для планування подорожей (рис. 3.9) та маршрут побудований за допомогою сервісу Google Maps (рис. 3.12), то видно, що це майже той самий маршрут, лише з несуттєвою різницею в часі і відстані.

## 3.5 Реалізація серверної частини

### 3.5.1 Архітектура серверу

Серверна частина застосунку реалізована з використанням мови програмування Python та вебфреймворку FastAPI, що забезпечує високу швидкодію та підтримку сучасних стандартів розробки RESTful API. Вибір FastAPI зумовлений його простотою в реалізації асинхронних обчислень, широкою підтримкою типізації та зручним автоматичним створенням документації до API, що особливо актуально при інтеграції з фронтендом і зовнішніми сервісами.

Основним завданням серверної частини є обробка запитів на оптимізацію маршрутів, а також обслуговування запитів до інформації про туристичні об’єкти.

Файлова структура серверної частини має наступний вигляд:

– *main.py* – основна точка входу, в якій підключаються маршрути та налаштовується FastAPI;

– *solver.py* – містить виклики алгоритмів оптимізації маршрутів;

– *models.py* – pydantic-моделі для валідації вхідних та вихідних даних;

– *algorithms/* – директорія, яка містить реалізацію всіх алгоритмів оптимізації маршрутів;

– *routers/search.py* – окремий модуль для обробки GET-запитів пошуку локацій;

– *data/* – директорія з JSON-файлами, що зберігають інформацію про місця, готелі та попередньо обчислені маршрути між точками;

– *data/bratislava_places.json* – JSON-файл, що зберігає інформацію про місця на мапі міста;

– *data/bratislava_hotels.json* – JSON-файл, що зберігає інформацію про готелі на мапі міста;

– *data/all_shortest_paths.json* – JSON-файл, що зберігає інформацію про найкоротші маршрути між кожними двома точками.

У серверній частині реалізовано два основних маршрути запитів:

– *POST /solve-route* – приймає список вибраних точок і обраний алгоритм маршрутизації, повертає оптимізований маршрут із відповідною статистикою;

– *GET /search_locations?query=...* – повертає список локацій, релевантних до пошукового запиту користувача.

Лістинг 3.4 Запит на розрахунок маршруту:

```
POST /solve-route
Content-Type: application/json

{
  "places": [
    { "id": 1, "lat": 48.1421, "lon": 17.1002, "duration": 60 },
    { "id": 7, "lat": 48.1488, "lon": 17.1074, "duration": 45 },
    { "id": 2, "lat": 48.1396, "lon": 17.1101, "duration": 30 }
  ],
  "algorithm": "Greedy"
}
```

Лістинг 3.5 Відповідь на запит розрахунок маршруту:

```
{
  "path": [
    [48.1421, 17.1002],
    [48.1396, 17.1101],
    [48.1488, 17.1074]
  ],
  "segments": [
    { "from": 1, "to": 2, "distance_km": 1.1 },
```

*{ "from": 2, "to": 7, "distance_km": 0.9 }*
*],*
*"walking_time_min": 21,*
*"total_time_with_stops_min": 156,*
*"total_distance_km": 2.0,*
*"computation_time_sec": 0.03*
*}*

Якщо ж відбувається запит на пошук локації, наприклад *GET /search_locations?query=castle*, то відповідь представляє перелік з 10 елементів, кожен з яких дає всю інформацію про точку, обрану для результату пошуку.

### 3.5.2 Реалізація алгоритмів обчислення маршрутів

Для знаходження оптимального маршруту між вибраними туристичними локаціями в застосунку реалізовано декілька алгоритмів. Вони відрізняються як за точністю результату, так і за обчислювальною складністю. Це дає змогу користувачеві вибрати найбільш прийнятний варіант залежно від кількості точок маршруту, обмежень по часу чи бажання порівняти результати різних підходів.

Реалізовано наступні алгоритми маршрутизації:

– Fixed Order – побудова маршруту відповідно до порядку вибраних точок;

– Brute Force – перебір усіх можливих перестановок (факторіальна складність, використовується тільки для невеликої кількості точок);

– Greedy – жадібний підхід: з кожної точки обирається найближча наступна;

– Held-Karp – динамічне програмування для задачі комівояжера (TSP);

– Branch and Bound – оптимізація перебору з відсіченням гілок;

– Concorde – виклик стороннього оптимізованого TSP-алгоритму.

Усі алгоритми реалізовані або адаптовані в директорії *algorithms/* та викликаються в модулі *solver.py*. Вибір алгоритму здійснюється на стороні клієнта, і відповідне ім’я передається у тілі запиту */solve-route*.

На вхід алгоритми отримують список об’єктів, кожен з яких має такі поля:

– *id* – унікальний ідентифікатор точки;

– *lat, lon* – географічні координати;

– *duration* – час перебування в точці в хвилинах.

Крім того, маршрути між усіма парами точок уже попередньо розраховані й збережені у файлі all_shortest_paths.json, що значно зменшує навантаження на сервер під час запитів.

## 3.6 Тестування та аналіз отриманих результатів

### 3.6.1 Тестування працездатності

З метою перевірки коректності функціонування вебзастосунку для планування подорожей було проведено комплексне ручне тестування основних компонентів користувацького інтерфейсу. Тестування охоплювало перевірку дій користувача від моменту завантаження сторінки до завершення побудови маршруту та його збереження у вигляді посилання або QR-коду.

Одним з ключових функціональних елементів є інтерактивна карта, реалізована на основі бібліотеки Leaflet. Під час тестування було успішно перевірено масштабування мапи, інформаційні вікна кожної з локацій, функціонал додавання локації в маршрут з різним часом відвідування.

Особлива увага була приділена функції пошуку з автозаповненням. Було протестовано нечутливість до регістру (castle, Castle, CASTLE – однаковий результат), нечутливість до діакритики (введення cafeteria та café повертає

однаковий результат), автозаповнення за другим або третім словом у назві локації, пріоритет локацій, у яких збіг починається з першого слова.

Після вибору готелю як стартової точки та додавання кількох локацій маршрут формується одним із доступних алгоритмів. Після натискання кнопки «Build/update route», карта автоматично масштабується до побудованого маршруту, а на екрані відображаються уся потрібна інформація.

Користувач має можливість зберегти побудований маршрут у вигляді посилання на Google Maps. Для зручності посилання дублюється як QR-код, який генерується на стороні клієнта.

Під час багаторазових сеансів тестування не було виявлено критичних помилок. Усі компоненти UI працюють узгоджено, що дозволяє перейти до глибшого аналізу ефективності побудованих маршрутів залежно від обраного алгоритму.

### 3.6.2 Порівняння маршрутів, отриманих різними алгоритмами

Для оцінки ефективності реалізованих алгоритмів маршрутизації було проведено серію порівняльних експериментів (рис. А.1 – рис. А.5). Результати записано до таблиці 3.7.

Таблиця 3.7 – Результати маршрутизації для 11 точок

| **Алгоритм** | **Час в дорозі** | **Відстань** | **Час опрацювання** |
|---|---|---|---|
| Порядок без змін | 2 г. 57 хв | 14 км 820 м | 0 с 0 мс |
| Greedy | 2 г. 9 хв | 10 км 790 м | 0 с 0 мс |
| Brute Force | 1 г. 48 хв | 9 км 10 м | 459 с 23 мс |
| Branch and bound | 1 г. 48 хв | 9 км 10 м | 98 с 194 мс |
| Held-Karp | 1 г. 48 хв | 9 км 10 м | 0 с 55 мс |

Як видно з результатів найкращим алгоритмом виявився Held-Karp, а найгіршим рішенням було б не змінювати порядок.

### 3.6.3 Порівняння побудованих маршрутів з Google Maps

Для оцінки практичної корисності побудованих маршрутів було проведено порівняння результатів, отриманих у створеному застосунку для планування подорожей (рис. А.6 – А.8), із маршрутами, згенерованими за допомогою сервісу Google Maps (рис. А.9 – А.11) для 6, 8 та 10 точок в маршруті, побудованому за допомогою алгоритма Held-Karp. Аналіз здійснювався на тих самих наборах локацій, за однакових умов.

Порівнявши результати побудови маршруту локально в застосунку і за допомогою сервісу Google Maps видно, що результати Google Maps за часом в дорозі зазвичай більші і складають 110% – 132% від результату в застосунку, це відбувається через те, що Google Maps враховує зміну швидкості в залежності від підйомів та спусків. Відстань же майже однакова і результати складають 94% –112% від результатів в застосунку.

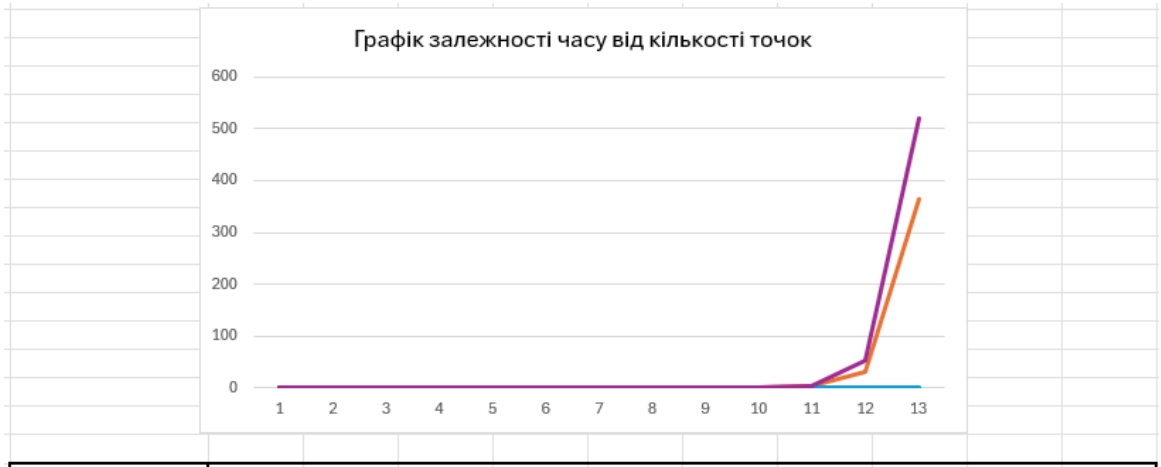
### 3.6.4 Аналіз часу побудови маршруту

Було проведено серію вимірювань часу обчислення маршрутів за допомогою кожного з реалізованих алгоритмів: Brute Force, Greedy, Held-Karp, Branch and Bound та Concorde.

Метою аналізу є оцінка продуктивності кожного з підходів залежно від кількості точок у маршруті, а також побудова загального уявлення про придатність алгоритмів для реального застосування.

Усі вимірювання (рис. 3.13) виконувались на однаковому комп’ютері з процесором Intel Core i7-10510U CPU @ 1.80GHz у локальному середовищі. Для кожної кількості точок (від 3 до 12) маршрут будувався 10 разів – результати усереднювались. Окремо фіксувались:

– час обчислення (секунди);
– відстань (км);




| Алгоритми/кількість точок | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|
| Brute Force | 0 | 1.00E-04 | 1.50E-04 | 4.33E-04 | 3.32E-03 | 2.82E-02 | 0.3 | 2.8 | 30.6 | 364.0 |
| Greedy | 0 | 1.04E-04 | 0 | 1.19E-04 | 0 | 1.04E-04 | 2.08E-04 | 1.31E-04 | 1.19E-04 | 1.04E-04 |
| Held - Karp | 1.04E-04 | 0 | 1.04E-04 | 3.38E-04 | 8.41E-04 | 2.10E-03 | 5.30E-03 | 1.33E-02 | 4.46E-02 | 0.1 |
| Branch and Bound | 1.04E-04 | 1.04E-04 | 4.87E-04 | 2.06E-03 | 1.33E-02 | 0.1 | 0.7 | 3.8 | 54.1 | 519.4 |

Рисунок 3.13 – Усереднені результати симуляції алгоритмів на 3-12 точках, обраховані в Exsel

Brute Force швидко стає непридатним для практичного використання при кількості точок >10. Час експоненціально зростає, навіть у 11 точках перевищує 30 секунд, що є критичним для інтерактивного застосунку.

Greedy показує стабільно низький час обчислення — менше 50 мс у всіх випадках. Це робить його найпридатнішим для моментального оновлення маршруту при кожній зміні точок.

Held-Karp забезпечує оптимальний баланс – час зростає лінійно, навіть при 10 точках не перевищує 0.1 сек, що комфортно для користувача.

Branch and Bound – здивував і в середньому виявився навіть повільніший за метод повного перебору на локальній симуляції. Його використання доцільне при кількості точок ≤8, якщо потрібна підвищена точність.

Алгоритми Greedy, Held-Karp та Branch and Bound було порівняно за правильністю з алгоритмом Brute Force і виявлено, що алгоритми Held-Karp та Branch and Bound коректні в 100% випадків, тоді як Greedy може забезпечити лише 22% дійсно найкоротших маршрутів за випадкових вхідних даних.

# ВИСНОВКИ

У межах даної кваліфікаційної роботи було спроєктовано та реалізовано прототип вебзастосунку для планування подорожей та оптимізації туристичних маршрутів на основі популярних алгоритмів розв’язання задач комівояжера. Застосунок орієнтований на користувачів, які бажають ефективно організувати подорож містом, обираючи найцікавіші локації та будуючи оптимальний маршрут з урахуванням часу, кількості точок і логіки переміщення реальними дорогами.

У процесі розробки були досягнуті наступні результати:

– виконано детальний аналіз існуючих сервісів для планування маршрутів, таких як Google Maps, Rome2Rio, Wanderlog, Sygic Travel та інших, що дозволило визначити ключові функції, які мають бути реалізовані у застосунку, а також сформувати вимоги до інтерфейсу та архітектури системи;

– спроєктовано загальну архітектуру застосунку, що поєднує клієнтську частину на основі React та серверну частину з використанням FastAPI. Передбачено чіткий обмін даними через REST API, реалізовано передачу структурованої інформації у форматі JSON;

– обґрунтовано вибір мов програмування та бібліотек, зокрема використання React, Tailwind CSS, Leaflet, FastAPI, NetworkX та інших інструментів, які забезпечують зручність у розробці, масштабованість і високий рівень інтерактивності користувацького інтерфейсу;

– реалізовано функціонал для пошуку локацій із автозаповненням, побудову маршруту реальними дорогами з використанням OpenStreetMap і бібліотеки osmnx, а також відображення оптимального шляху на карті з підрахунком тривалості подорожі;

– досліджено, математично описано та реалізовано популярні алгоритми маршрутизації, такі як повний перебір (Brute Force), жадібний метод, алгоритм Хелда-Карпа, метод гілок і меж, а також використання Concorde для порівняння з теоретично оптимальними результатами;

– проведено аналіз ефективності алгоритмів за критеріями часу виконання та споживання пам’яті, а також побудовано графіки, які наочно ілюструють залежності продуктивності від кількості точок маршруту;

– розроблено систему збереження та передачі маршруту за допомогою QR-кодів та зовнішніх посилань, що дозволяє швидко поділитися маршрутом з іншими користувачами або перейти до нього з мобільного пристрою;

– виконано моделювання структури зберігання даних про туристичні об’єкти у форматі JSON із можливістю розширення, описані переваги такого підходу над використанням повноцінної БД на етапі MVP (minimum viable product);

– створено адаптивний і зручний інтерфейс з інтерактивною мапою, полем пошуку з підказками, виводом інформації про точки маршруту та вибором бажаного алгоритму оптимізації;

– виконано тестування основних сценаріїв взаємодії користувача з системою, зокрема пошук локацій, формування маршруту, його збереження та перегляд, а також перевірено працездатність усіх реалізованих алгоритмів на тестових вибірках.

Таким чином, у ході написання кваліфікаційної роботи було досягнуто поставлених цілей і виконано всі передбачені задачі. Розроблений застосунок демонструє високий рівень функціональності, гнучкості та технічної якості, а також створює основу для подальшого масштабування: зокрема, додавання персоналізованих рекомендацій, реєстрації користувачів, збереження історії подорожей та комерційної інтеграції з зовнішніми сервісами (готелі, музеї, ресторани тощо). Застосунок може бути використаний як база для стартапу у сфері смарт-туризму або як навчальний інструмент для студентів технічних спеціальностей.

# ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Google. Google Maps. URL: https://www.google.com/maps (дата звернення: 14.04.2025).

2. Apple Inc. Apple Maps. URL: https://www.apple.com/maps (дата звернення: 14.04.2025).

3. HERE Technologies. HERE WeGo. URL: https://wego.here.com (дата звернення: 14.04.2025).

4. Sygic a.s. Sygic Travel. URL: https://www.sygic.com/travel (дата звернення: 14.04.2025).

5. TripHobo. TripHobo. URL: https://www.triphobo.com (дата звернення: 14.05.2025).

6. Rome2Rio Pty Ltd. Rome2Rio. URL: https://www.rome2rio.com (дата звернення: 15.04.2025).

7. Wanderlog, Inc. Wanderlog. URL: https://wanderlog.com (дата звернення: 18.04.2025).

8. Roadtrippers LLC. Roadtrippers. URL: https://roadtrippers.com (дата звернення: 18.04.2025).

9. Komoot GmbH. Komoot. URL: https://www.komoot.com (дата звернення: 21.04.2025).

10. AllTrails, LLC. AllTrails. URL: https://www.alltrails.com (дата звернення: 21.04.2025).

11. OptimoRoute Inc. OptimoRoute. URL: https://optimoroute.com (дата звернення: 21.04.2025).

12. RouteXL. RouteXL. URL: https://www.routexl.com (дата звернення: 21.04.2025).

13. Circuit Routing Limited. Circuit Route Planner. URL: https://getcircuit.com/route-planner (дата звернення: 21.04.2025).

14. Applegate, D. L. (2006). The traveling salesman problem: a computational study (Vol. 17). Princeton university press.

15. Held, M., & Karp, R. M. (1962). A dynamic programming approach to sequencing problems. Journal of the Society for Industrial and Applied mathematics, 10(1), 196-210.

16. Reinelt, G. (1991). TSPLIB—A traveling salesman problem library. ORSA journal on computing, 3(4), 376-384.

17. Cook, W. Concorde TSP Solver. URL: http://www.math.uwaterloo.ca/tsp/concorde.html (дата звернення: 14.05.2025).

18. Машталір, С. В. (2018) Методи оптимізації в логістиці: навч. посібник. Харків: ХНУРЕ.

19. Google Developers. Google Maps URLs. URL: https://developers.google.com/maps/documentation/urls (дата звернення: 01.05.2025).

20. Meta Platforms, Inc. React – A JavaScript library for building user interfaces. URL: https://react.dev (дата звернення: 01.05.2025).

21. Tiangolo. FastAPI – The fast web framework for building APIs with Python 3.6+. URL: https://fastapi.tiangolo.com (дата звернення: 01.05.2025).

22. Harris, C. R., Millman, K. J., Van Der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., ... & Oliphant, T. E. (2020). Array programming with NumPy. Nature, 585(7825), 357-362.

23. Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. Computing in science & engineering, 9(03), 90-95.

24. TSP Solver and Generator. URL: http://www.math.uwaterloo.ca/tsp/concorde.html (дата звернення: 03.05.2025).

25. Google Maps Platform Documentation. URL: https://developers.google.com/maps/documentation (дата звернення: 03.05.2025).

26. Python Software Foundation. Python 3 Documentation. URL: https://docs.python.org/3/ (дата звернення: 03.05.2025).

27. NumPy Documentation. URL: https://numpy.org/doc/ (дата звернення: 03.05.2025).

28. Matplotlib Documentation. URL: https://matplotlib.org/stable/contents.html (дата звернення: 04.05.2025).

29. FastAPI Documentation. URL: https://fastapi.tiangolo.com/ (дата звернення: 04.05.2025).

30. React Documentation. URL: https://react.dev/learn (дата звернення: 04.05.2025).

31. Concorde TSP Solver: Downloads. URL: http://www.math.uwaterloo.ca/tsp/concorde.html (дата звернення: 04.05.2025).