

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи освітнього ступеня «магістр»
за спеціальністю 121 «Інженерія програмного забезпечення»
(освітня програма «Інженерія програмного забезпечення»)

на тему:

**«Застосунок модифікованої задачі пошуку
оптимального маршруту з урахуванням обмежень»**

Виконала студентка групи ІПЗм-24-1
БІЛЬКЕВИЧ Дар'я Володимирівна

Керівник роботи:
ЧИЖМОТРЯ Олексій Володимирович

Рецензент:
ЛОБАНЧИКОВА Надія Миколаївна

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

«ЗАТВЕРДЖУЮ»

зав. кафедри інженерії
програмного забезпечення
Тетяна ВАКАЛЮК
«18» вересня 2025 р.

ЗАВДАННЯ
на кваліфікаційну роботу

Здобувач вищої освіти: **БІЛЬКЕВИЧ Дар'я Володимирівна**

Керівник роботи: **ЧИЖМОТРЯ Олексій Володимирович**

Тема роботи: **«Застосунок модифікованої задачі пошуку оптимального маршруту з урахуванням обмежень»**,

затверджена Наказом закладу вищої освіти від **«18» вересня 2025 р. № 491/с**

Вихідні дані для роботи: **планування автомобільних маршрутів з урахуванням обмежень на запас палива на дорожній мережі України; алгоритми пошуку найкоротшого шляху на графах великого розміру; методи оптимізації транспортного графа з використанням просторової індексації; автоматичне планування точок дозаправки з урахуванням параметрів транспортного засобу; інтеграція даних про трафік для міських маршрутів.**

Консультанти з магістерської кваліфікаційної роботи із зазначенням розділів, що їх стосуються:

Розділ	Консультант	Завдання видав	Завдання прийняв
1	Чижмотря О.В.	22.09.2025р.	22.09.2025р.
2	Чижмотря О.В.	22.09.2025р.	22.09.2025р.
3	Чижмотря О.В.	22.09.2025р.	22.09.2025р.

РЕФЕРАТ

Випускна кваліфікаційна робота магістра складається з опису дослідження методів планування оптимальних автомобільних маршрутів з урахуванням обмежень на запас палива, розробки та реалізації алгоритмів оптимізації графа дорожньої мережі, створення системи автоматичного планування точок дозаправки та пояснювальної записки. Пояснювальна записка до кваліфікаційної роботи складається з 87 сторінок, 29 ілюстрацій, 1 таблиці, 1 додатку та 40 літературних джерел.

Мета роботи – розробка та реалізація системи планування оптимальних автомобільних маршрутів з автоматичним визначенням точок дозаправки на основі модифікованих алгоритмів пошуку найкоротшого шляху з урахуванням обмежень на запас палива та інтеграції даних про реальний стан трафіку для міських маршрутів на дорожній мережі України.

У роботі було описано цілі та поставлено задачі до досліджень та розробок. Було проаналізовано існуючі системи планування маршрутів та базові алгоритми пошуку найкоротшого шляху на графах. Розроблено методи оптимізації транспортного графа з використанням просторової індексації та dual-graph стратегії для скорочення часу обчислень. Реалізовано адаптивний вибір алгоритмів пошуку шляху залежно від типу маршруту з інтеграцією даних про трафік для міських маршрутів. Розроблено алгоритм планування точок дозаправки, який забезпечує мінімальну кількість зупинок при гарантованій досяжності пункту призначення. Спроековано та реалізовано функціональну систему, що складається з серверної частини та мобільного застосунку, проведено тестування та аналіз ефективності.

У висновках було отримано результати досліджень та розробок за час виконання кваліфікаційної роботи.

КЛЮЧОВІ СЛОВА: ГРАФ, ПЛАНУВАННЯ МАРШРУТІВ, ОПТИМІЗАЦІЯ ГРАФА, АЛГОРИТМИ ПОШУКУ ШЛЯХУ, ДОЗАПРАВКИ, ПРОСТОРОВА ІНДЕКСАЦІЯ.

ABSTRACT

The master's qualification work consists of a description of the study of methods for planning optimal automobile routes considering fuel capacity constraints, development and implementation of road network graph optimization algorithms, creation of an automatic refueling point planning system, and an explanatory note. The explanatory note to the qualification work consists of 87 pages, 29 illustrations, 1 table, 1 appendix and 40 literature sources.

The aim of the work is to develop and implement a system for planning optimal automobile routes with automatic determination of refueling points based on modified shortest path search algorithms considering fuel capacity constraints and integration of data on real traffic conditions for urban routes on the Ukrainian road network.

The work described the objectives and set tasks for research and development. Existing route planning systems and basic shortest path search algorithms on graphs were analyzed. Methods for optimizing the transport graph using spatial indexing and dual-graph strategy were developed to reduce computation time. Adaptive selection of path search algorithms depending on the route type with traffic data integration for urban routes was implemented. A refueling point planning algorithm was developed that ensures the minimum number of stops while guaranteeing destination reachability. A functional system consisting of a server-side component and a mobile application was designed and implemented, testing and efficiency analysis were conducted.

The conclusion presents the results of research and development obtained during the completion of the qualification work.

KEYWORDS: GRAPH, ROUTE PLANNING, GRAPH OPTIMIZATION, PATH SEARCH ALGORITHMS, REFUELING, SPATIAL INDEXING.

ЗМІСТ

РЕФЕРАТ	3
ABSTRACT	4
ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ	6
ВСТУП	7
РОЗДІЛ 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ	10
1.1 Постановка завдання	10
1.2 Огляд аналогів	12
1.3 Вибір програмних засобів для реалізації	20
1.4 Аналіз базових алгоритмів пошуку оптимальних маршрутів	23
Висновки до першого розділу	28
РОЗДІЛ 2 РОЗРОБКА АЛГОРИТМУ ОПТИМАЛЬНОГО МАРШРУТУ	30
2.1 Формування графа мережі та оптимізація простору пошуку	30
2.2 Побудова оптимального шляху на підграфі	40
2.3 Побудова маршруту з урахуванням дозаправок	48
Висновки до другого розділу	52
РОЗДІЛ 3 РЕАЛІЗАЦІЯ ТА ІНТЕГРАЦІЯ ПРОГРАМНОЇ СИСТЕМИ	54
3.1 Архітектура та взаємодія компонентів системи	54
3.2 Опис роботи програмного забезпечення	58
Висновки до третього розділу	72
ВИСНОВКИ	73
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	75
ДОДАТКИ	80

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

API – Application Programming Interface (Прикладний програмний інтерфейс)

BBOX – Bounding Box (Обмежувальна рамка / Обмежувальний прямокутник)

ETA – Estimated Time of Arrival (Очікуваний час прибуття)

HTTP – HyperText Transfer Protocol (Протокол передачі гіпертексту)

JWT – JSON Web Token (Веб-токен JSON)

MBR – Minimum Bounding Rectangle (Мінімальний обмежувальний прямокутник)

OSM – OpenStreetMap (Відкрита карта вулиць)

OTP – One-Time Password (Одноразовий пароль)

REST – Representational State Transfer (Передача репрезентативного стану)

SMS – Short Message Service (Служба коротких повідомлень)

ВСТУП

Актуальність теми. Автомобільні подорожі залишаються одним з найпопулярніших способів пересування завдяки свободі планування маршруту та незалежності від розкладу громадського транспорту. Однак планування довгих автомобільних маршрутів супроводжується необхідністю врахування обмежень, пов'язаних із запасом палива в автомобілі. Водії змушені постійно моніторити рівень палива, шукати заправні станції вздовж маршруту та приймати рішення про необхідність зупинки для заправки, що відволікає увагу від дороги та може призводити до неоптимального планування зупинок. Неправильне планування може призвести до ризику залишитися без палива на віддалених ділянках маршруту або до надмірної кількості зупинок, що збільшує загальний час подорожі. Додатковим фактором, який впливає на ефективність планування, є стан дорожнього руху, особливо на міських маршрутах, де затори можуть суттєво змінювати час подорожі та витрати палива. Автоматизація процесу планування зупинок для заправки з урахуванням параметрів конкретного автомобіля та актуального стану доріг дозволить користувачам заздалегідь отримувати детальні плани подорожей з оптимальними точками заправки, мінімізуючи кількість зупинок при гарантованій досяжності пункту призначення.

Ця проблема набуває додаткової актуальності в контексті зростаючої популярності електромобілів, для яких планування зарядних станцій є ще більш критичним через обмежений запас ходу та значно довший час зарядки порівняно з заправкою традиційних автомобілів. Хоча поточна робота зосереджена на автомобілях з двигунами внутрішнього згоряння, розроблені алгоритми та методи оптимізації мають універсальний характер і можуть бути адаптовані для електромобілів, що робить дослідження перспективним для майбутнього розвитку транспортних систем.

Розробка ефективного алгоритмічного рішення для автоматичного планування маршрутів з урахуванням обмежень на запас палива дозволить покращити досвід користувачів при плануванні подорожей, забезпечить

можливість аналізу різних варіантів маршрутів та оцінки ефективності планування, що є важливим для оптимізації витрат часу та палива.

Мета роботи полягає у розробці та реалізації системи планування оптимальних автомобільних маршрутів з автоматичним визначенням точок дозаправки на основі модифікованих алгоритмів пошуку найкоротшого шляху з урахуванням обмежень на запас палива та інтеграції даних про реальний стан трафіку для міських маршрутів на дорожній мережі України.

Для досягнення мети виокремлено такі основні **завдання**:

- провести аналіз існуючих систем планування маршрутів та базових алгоритмів пошуку найкоротшого шляху на графах;
- розробити методи оптимізації графа дорожньої мережі;
- реалізувати адаптивну стратегію вибору алгоритмів пошуку шляху з інтеграцією даних про трафік;
- розробити багатокритеріальний алгоритм планування точок дозаправки;
- спроектувати та реалізувати систему у вигляді серверної частини та мобільного застосунку;
- провести тестування системи та проаналізувати ефективність алгоритмів.

Об'єктом дослідження є процес планування автомобільних маршрутів з урахуванням обмежень на запас палива на дорожній мережі України.

Предметом дослідження є алгоритми та методи оптимізації пошуку маршрутів на графах великого розміру з автоматичним плануванням точок дозаправки та інтеграцією даних про трафік для міських маршрутів.

Наукова новизна роботи полягає у розробці методу автоматичного планування маршрутів з урахуванням обмежень на запас палива, який поєднує класичні алгоритми пошуку шляху з багатокритеріальною логікою визначення точок дозаправки. Запропоновано систему з чотирма критеріями прийняття рішень та адаптивним порогом безпеки, що забезпечує мінімальну кількість зупинок при гарантованій досяжності пункту призначення, а також гібридний

підхід до оптимізації обчислень, який дозволив зберегти прийнятний час формування маршруту.

Практичне значення роботи полягає у розробці функціональної системи для планування автомобільних маршрутів, яка може бути використана водіями для ефективного планування подорожей з автоматичним визначенням оптимальних точок заправки. Розроблені алгоритми та методи оптимізації можуть бути адаптовані для різних типів транспортних засобів, включаючи електромобілі, що робить систему перспективною для майбутнього розвитку. Реалізована система надає користувачам можливість заздалегідь отримувати детальні плани подорожей з аналізом витрат часу та палива, що дозволяє приймати обґрунтовані рішення при виборі маршруту.

Апробація. За темою дослідження кваліфікаційної магістерської роботи було опубліковано 2 тези конференцій [1, 2].

РОЗДІЛ 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Постановка завдання

Автомобільні подорожі стають дедалі популярнішими через свою свободу у виборі планування маршруту, зручність та незалежність. Одним з ключових викликів при плануванні довгих автомобільних маршрутів є необхідність врахування обмежень, пов'язаних із запасом пального в автомобілі та поточною дорожньою ситуацією. Основна задача дослідження полягає у створенні ефективного алгоритмічного рішення для оптимізації маршрутів з автоматичним плануванням зупинок для заправки та врахуванням трафіку на міських маршрутах, що забезпечить користувачам можливість ефективного планування подорожей з мінімальними витратами часу та пального).

Виходячи з даних цілей, необхідно виконати наступні завдання:

1. Дослідження предметної області та аналіз існуючих рішень:
 - 1.1. Провести аналіз популярних систем планування маршрутів, визначити їхні переваги та недоліки.
 - 1.2. Дослідити алгоритми пошуку найкоротшого шляху на графах (алгоритм Дейкстри, A*).
 - 1.3. Проаналізувати підходи до врахування обмежень при плануванні маршрутів (запас пального, трафік).
 - 1.4. Дослідити методи оптимізації графів великого розміру для швидкої обробки запитів.
2. Вибір технологічного стеку та інтеграція зовнішніх сервісів:
 - 2.1. Визначити багат шарову архітектуру системи та обґрунтувати вибір технологій для алгоритмічної частини, API-шару та мобільного клієнта.
 - 2.2. Інтегрувати API для отримання даних про поточний стан трафіку.
 - 2.3. Інтегрувати дані про розташування заправних станцій.
 - 2.4. Вибрати картографічне API для візуалізації маршрутів.

3. Розробка структури даних та алгоритмів оптимізації:
 - 3.1. Розробити ефективне представлення графа дорожньої мережі України (близько мільйона вузлів).
 - 3.2. Реалізувати оптимізований граф з виділенням основних доріг для обробки довгих маршрутів.
 - 3.3. Реалізувати алгоритм пошуку оптимального маршруту з автоматичним плануванням зупинок для заправки.
 - 3.4. Розробити гібридний підхід: використання повного графа з трафіком для коротких маршрутів та оптимізованого графа для довгих маршрутів.
 - 3.5. Реалізувати просторове індексування (R-tree) для ефективної побудови підграфів у заданій географічній області.
 - 3.6. Провести оптимізацію обчислювальних процесів для забезпечення прийняттого часу побудови маршрутів.
4. Вибір технологічного стеку та інтеграція зовнішніх сервісів:
 - 4.1. Обрати технології для серверної частини та обґрунтувати вибір.
 - 4.2. Визначити архітектуру системи (серверна частина, API-шар, мобільний клієнт).
 - 4.3. Обрати платформу для розробки мобільного застосунку.
 - 4.4. Інтегрувати API для отримання даних про поточний стан трафіку.
 - 4.5. Інтегрувати дані про розташування заправних станцій.
 - 4.6. Вибрати картографічне API для візуалізації маршрутів.
5. Розробка мобільного застосунку:
 - 5.1. Створити інтерфейс для введення параметрів маршруту (початкова та кінцева точки, параметри автомобіля).
 - 5.2. Реалізувати візуалізацію побудованого маршруту з відображенням запланованих зупинок для заправки, сегментів трафіку.
 - 5.3. Розробити функціонал збереження побудованих маршрутів.

5.4. Реалізувати перегляд історії маршрутів з детальною статистикою (відстань, час, зупинки, трафік, час побудови).

6. Тестування та оптимізація продукту:

6.1. Провести тестування коректності роботи алгоритмів на різних типах маршрутів.

6.2. Виконати аналіз продуктивності системи (час побудови, використання пам'яті).

6.3. Оптимізувати вузькі місця системи на основі результатів тестування.

6.4. Провести порівняльний аналіз з існуючими рішеннями.

Очікуваним результатом буде реалізована система для планування оптимальних маршрутів з урахуванням обмежень, що складається з серверної частини для виконання обчислень на графі дорожньої мережі України та мобільного застосунку для взаємодії з користувачем. Система забезпечить побудову оптимальних маршрутів з урахуванням запасу пального в автомобілі, поточного стану трафіку та автоматичним визначенням оптимальних точок для заправки. Користувачі зможуть зберігати побудовані маршрути та переглядати їх історію з детальною статистикою для аналізу ефективності планування.

Науковою цінністю роботи є розробка методу розширення класичних алгоритмів пошуку найкоротшого шляху для врахування обмеження на запас пального з автоматичним плануванням оптимальних зупинок для заправки, а також реалізація гібридного підходу до обробки маршрутів різної довжини на графах великого розміру.

1.2 Огляд аналогів

У процесі розробки застосунку для оптимізації маршрутів з урахуванням обмежень важливим етапом є аналіз існуючих рішень, які вже представлені на ринку. Огляд аналогів дозволяє визначити переваги та недоліки популярних

систем планування маршрутів, оцінити функціональність щодо планування заправок, а також дослідити підходи до реалізації алгоритмів пошуку оптимальних шляхів. Це, у свою чергу, дає змогу виявити ефективні практики, які можна адаптувати в новому продукті, зрозуміти обмеження існуючих рішень у контексті автоматичного планування зупинок для заправки та визначити напрямки для вдосконалення розроблюваної системи.

Google Maps

Google Maps був офіційно запущений 8 лютого 2005 року після того, як Google придбала у жовтні 2004 року компанію Where 2 Technologies, засновану датськими братами Ларсом та Єнсом Расмуссен. Застосунок створювався як веб-базований картографічний сервіс для надання користувачам можливості планування маршрутів та навігації. На сьогодні Google Maps налічує понад мільярд активних користувачів щомісяця та охоплює понад 220 країн і територій, що робить його найпопулярнішою системою планування маршрутів у світі. Основне призначення застосунку – забезпечення користувачів точною навігацією з урахуванням поточного трафіку, надання інформації про об'єкти інфраструктури та допомога у плануванні подорожей.

Google Maps використовує потужні алгоритми для пошуку найкоротшого шляху, але офіційно не розкриває деталі. Ймовірніше за все, він використовує комбінацію алгоритму Дейкстри та алгоритму A* (A-star) для обчислення оптимальних маршрутів. Алгоритм Дейкстри є базовим алгоритмом пошуку найкоротшого шляху на графі, що гарантує знаходження оптимального маршруту для графів з невід'ємними вагами ребер A* покращує алгоритм Дейкстри, використовуючи евристичну функцію для направлення пошуку в сторону пункту призначення, що зменшує кількість непотрібних шляхів для дослідження та робить його швидшим на практиці [3].

Google Maps також ймовірно використовує ієрархічні підходи до оптимізації [4], які дозволяють системі спочатку отримати загальний вигляд карти, а потім заповнювати деталі лише за необхідності, що забезпечує майже

миттєві маршрути навіть для поїздок через цілі країни. Це досягається шляхом використання методів контракції графів та просторового індексування, які дозволяють ефективно обробляти величезні дорожні мережі.

Попри високу ефективність маршрутизації, функціональні можливості Google Maps щодо врахування індивідуальних параметрів користувача залишаються обмеженими. Що стосується планування зупинок для заправки, сервіс надає лише базовий функціонал – дозволяє користувачам під час активної навігації шукати заправки вздовж маршруту та додавати їх як проміжні зупинки вручну. Однак, система не враховує запас пального в автомобілі та не планує зупинки для заправки автоматично.

До переваг даного програмного продукту можна віднести точні дані про трафік у реальному часі завдяки мільйонам користувачів Android-пристроїв, високу якість та регулярне оновлення картографічних даних, широке покриття практично всіх регіонів світу, інтеграцію з іншими сервісами для отримання інформації про розташування заправних станцій, їхні ціни та графіки роботи, а також потужну алгоритмічну базу на основі A^* та оптимізованих структур даних.

Водночас, система має ряд недоліків, зокрема неможливість введення параметрів автомобіля (витрата палива, об'єм баку) для врахування при побудові маршруту, обмежене планування на етапі підготовки – користувачі не можуть заздалегідь отримати повний маршрут з усіма запланованими зупинками для заправки, відсутність історії поїздок з детальною статистикою, а також фокус на покроковій навігації в реальному часі, а не на детальному плануванні маршруту з обмеженнями.

Інтерфейс сервісу Google Maps показано нижче (рис. 1.1):

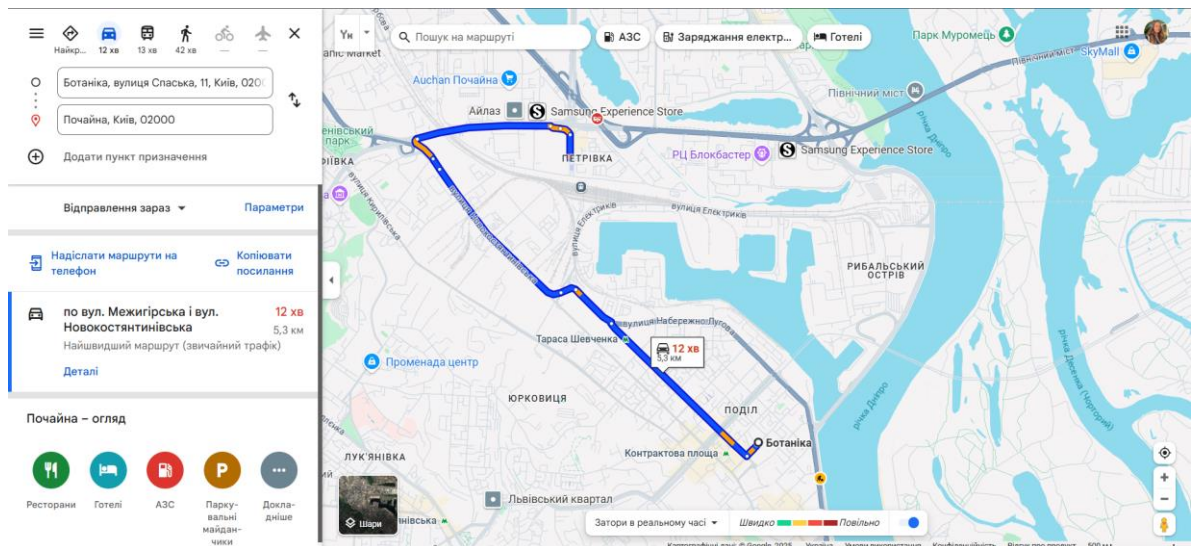


Рис. 1.1 – Інтерфейс сервісу Google Maps

Waze

Waze розпочинався у 2006 році як проєкт FreeMap Israel, створений ізраїльським програмістом Ехудом Шабтаєм з метою розробки безкоштовної цифрової карти Ізраїлю з можливістю оновлення спільноту. У 2008 році проєкт було комерціалізовано та перейменовано на Waze Mobile Ltd. У червні 2013 року Google придбав Waze за 1,3 мільярда доларів США. На відміну від Google Maps, Waze є застосунком, орієнтованим на спільноту користувачів, де кожен водій може вносити дані про дорожню ситуацію в реальному часі. Основне призначення застосунку – надання навігації з урахуванням актуального трафіку та попередження про небезпеки на дорозі на основі даних від інших користувачів.

Waze використовує алгоритм A^* для пошуку шляху, який дозволяє обчислювати найкоротший маршрут від точки А до точки В [5]. Алгоритм маршрутизації використовує історичні та реальні дані для швидкого реагування та навіть прогнозування умов руху наперед. Алгоритм віддає перевагу дорогам вищого типу, але менш обережний щодо використання місцевих доріг порівняно з альтернативними системами.

Waze використовує серверну маршрутизацію, яка враховує майбутній очікуваний час прибуття на сегмент дороги для розрахунку очікуваної швидкості на цьому сегменті. Для довгих поїздок (приблизно 30 хвилин або

більше) система включає майбутні проміжки часу, де поточні дані про трафік недоступні на момент початкового розрахунку. Система також використовує прунінг маршрутів для довгих дистанцій, видаляючи дороги нижчого типу з розгляду на певних відстанях від початку або кінця маршруту для оптимізації обчислень.

Waze надає функціонал пошуку заправних станцій вздовж маршруту. Застосунок дозволяє користувачам встановлювати переваги щодо типу палива та бренду заправки, а також шукати заправні станції на маршруті з найкращими цінами на основі даних, наданих спільнотою. Користувачі можуть сортувати заправні станції за відстанню, ціною або брендом, а також переглядати години роботи та послуги на станціях Android Central. Однак, як і Google Maps, Waze не пропонує автоматичного планування зупинок для заправки.

До переваг даного програмного продукту можна віднести оперативне отримання даних про трафік від активної спільноти користувачів, наявність інформації про ціни на паливо, що регулярно оновлюється, попереджень про небезпеки на дорозі, дорожні роботи та інші перешкоди, можливість пошуку заправок за ціною, що допомагає економити кошти, а також систему колективного внесення змін до картографічних даних про дороги та заправні станції.

Серед недоліків системи слід виділити відсутність автоматичного планування зупинок для заправки з урахуванням запасу палива та параметрів автомобіля (витрата палива, об'єм баку), фокус виключно на навігації в реальному часі без збереження історії маршрутів та їх аналітики для подальшого аналізу, а також високе споживання інтернет-трафіку та заряду батареї через постійну передачу даних про трафік у реальному часі.

Інтерфейс сервісу Waze показано нижче (рис. 1.2):

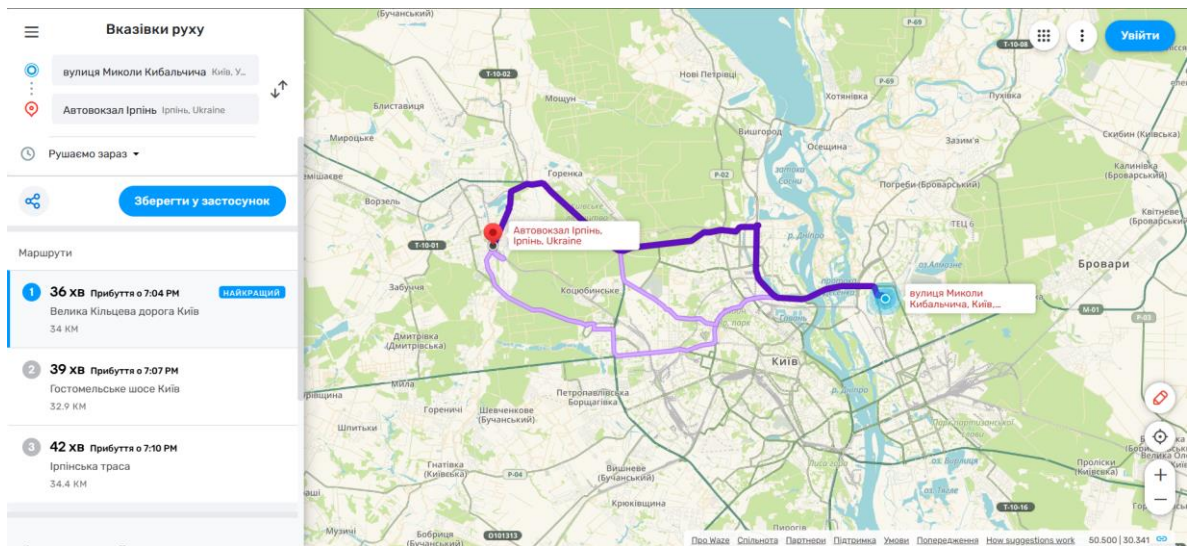


Рис. 1.2 – Інтерфейс сервісу Waze

Here WeGo

Here WeGo розпочинався у 2007 році як Nokia Maps для платформи Symbian та Windows Mobile. У 2012 році застосунок був перейменований на HERE Maps, а 27 липня 2016 року отримав нинішню назву HERE WeGo. HERE Technologies належить автомобільним виробникам Audi, BMW та Mercedes-Benz, а застосунок надає картографічні дані та навігаційні послуги для понад 200 країн. HERE WeGo є безкоштовним мобільним застосунком та планувальником подорожей, призначеним для підтримки подорожей від простих поїздок до складних мультимодальних маршрутів. Основне призначення застосунку – надання навігації з акцентом на приватність користувачів та можливість роботи в офлайн-режимі.

HERE WeGo дозволяє користувачам вибирати пріоритет при побудові маршруту: найкоротший час подорожі або найменшу відстань [6]. На відміну від Google Maps, який за замовчуванням шукає найшвидший маршрут, HERE WeGo надає більше контролю над параметрами маршрутизації. Застосунок дозволяє відключити автоматичну перемаршрутизацію в реальному часі, що дає можливість дотримуватися оригінального маршруту, запланованого перед початком навігації. Система використовує динамічні розрахунки очікуваного часу прибуття (ETA) [7] з урахуванням оновлень трафіку в реальному часі. HERE WeGo також має професійну версію для вантажівок (HERE WeGo Pro)

[8], яка враховує специфічні обмеження транспортних засобів та оптимізує маршрути для комерційних перевезень.

HERE WeGo надає можливість додавати зупинки вздовж маршруту, включаючи заправні станції (Gas), парковки, ресторани та туалети. Застосунок інформує користувачів про найкращі зупинки для відпочинку на маршруті, включаючи заправні станції, готелі та ресторани, надаючи інформацію про відстань та деталі про майбутні зупинки. Однак, як і в попередніх застосунках, HERE WeGo не пропонує автоматичного планування зупинок для заправки з урахуванням запасу палива. Користувач повинен самостійно додавати заправні станції як проміжні точки на маршруті.

Серед переваг даного програмного продукту слід виділити можливість завантаження офлайн-карт для навігації без інтернет-з'єднання, гнучкі налаштування маршрутизації з можливістю вибору пріоритету (час або відстань), контроль над автоматичною перемаршрутизацією, що дає змогу дотримуватися запланованого маршруту, а також підтримку мультимодальних подорожей (автомобіль, громадський транспорт, пішки, велосипед).

До недоліків системи належать застарілі картографічні дані в деяких регіонах, нижча точність даних про трафік порівняно з Google Maps та Waze через меншу базу активних користувачів, відсутність автоматичного планування зупинок для заправки з урахуванням поточного рівня палива в автомобілі (користувач має самостійно додавати заправні станції як проміжні точки), а також складність інтерфейсу для деяких користувачів після останніх оновлень застосунку.

Інтерфейс додатку HERE WeGo показано нижче (рис. 1.3):

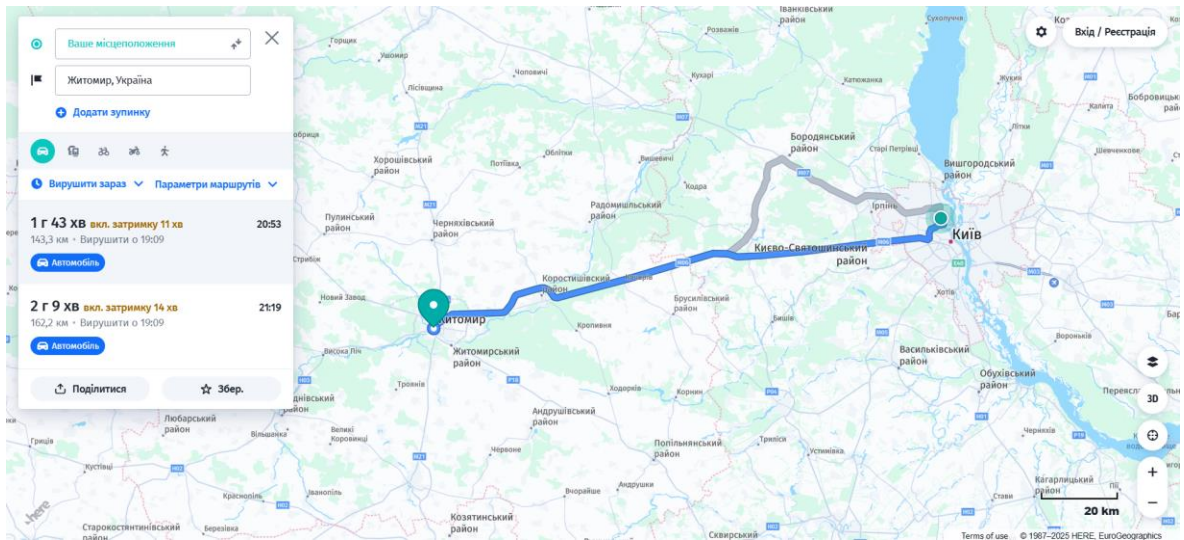


Рис. 1.3 – Інтерфейс додатку HERE WeGo

Враховуючи переваги та недоліки існуючих застосунків, розроблюваний продукт має на меті заповнити прогалину у сфері планування довгих автомобільних подорожей з автоматичним врахуванням обмежень на запас палива. На відміну від Google Maps, Waze та HERE WeGo, які надають лише можливість вручну шукати та додавати заправні станції як проміжні точки, наша система буде автоматично розраховувати оптимальні зупинки для заправки на основі параметрів конкретного автомобіля (витрата палива, об'єм баку).

Основна відмінність розроблюваного рішення полягає у зміщенні акценту з real-time навігації на детальне планування маршруту на етапі підготовки до поїздки. Система забезпечує користувачам можливість отримати повний маршрут з усіма запланованими зупинками для заправки ще до початку подорожі, а також зберігати історію побудованих маршрутів з детальною статистикою (загальна відстань, час побудови, кількість та пояснення вибору зупинок, дані про трафік) для подальшого аналізу ефективності планування. Таким чином, розроблюваний застосунок не конкурує з існуючими навігаторами, а доповнює їх функціонал, надаючи спеціалізований інструмент для планування подорожей з урахуванням обмежень.

1.3 Вибір програмних засобів для реалізації

Реалізація системи оптимізації маршрутів з урахуванням обмежень вимагає ретельного підбору технологічного стеку, який здатен забезпечити ефективну обробку великих графів дорожньої мережі, швидкий пошук оптимальних маршрутів та зручний користувацький інтерфейс. Архітектура системи побудована за багатошаровим принципом, де кожен компонент відповідає за певну функціональність: алгоритмічна частина виконує складні обчислення на графі дорожньої мережі України, серверна частина забезпечує зберігання та управління даними про маршрути, а мобільний застосунок надає інтерфейс для взаємодії з користувачем та візуалізації результатів. Вибір технологій для кожного рівня системи обґрунтовується специфічними вимогами до продуктивності, зручності розробки та можливості інтеграції з геопросторовими даними.

Алгоритмічна частина, реалізована на **Python** з використанням **Flask** – легковагого веб-фреймворку для створення RESTful API. Python, як основну мову для реалізації, було обрано через наявність потужних бібліотек для роботи з графами та просторовими даними, а також через простоту інтеграції з існуючими геопросторовими інструментами. Flask забезпечує простий та ефективний спосіб створення HTTP API для взаємодії між алгоритмічною частиною на Python та серверною частиною на TypeScript.

Для отримання даних про дорожню мережу України було обрано **OpenStreetMap (OSM)** [9, 10] – відкриту платформу картографічних даних, яка підтримується спільнотою волонтерів по всьому світу. OSM надає детальну інформацію про дороги, включаючи їхні типи (автомагістралі, головні дороги, другорядні дороги тощо), обмеження швидкості, напрямки руху та інші атрибути, необхідні для побудови графа дорожньої мережі. Вибір OSM обґрунтовується декількома факторами: дані є безкоштовними та доступними для комерційного використання, а також OSM забезпечує високу деталізацію дорожньої мережі України з регулярними оновленнями від

спільноти. Також існують зручні бібліотеки для роботи з даними OSM, зокрема **OSMnx** для Python, яка дозволяє завантажувати, обробляти та аналізувати дорожні мережі у вигляді графів. Завантажений граф дорожньої мережі України містить близько мільйона вузлів, що відображає всі доступні для проїзду дороги країни.

Дані про розташування заправних станцій також отримуються з OpenStreetMap, що забезпечує узгодженість між картографічними даними дорожньої мережі та об'єктами інфраструктури. OSM містить інформацію про понад 3,000 заправних станцій на території України з координатами їхнього розташування, що є достатнім для ефективного планування зупинок для заправки на маршрутах будь-якої довжини. Використання єдиного джерела даних (OSM) для доріг та заправок спрощує інтеграцію та забезпечує географічну узгодженість між елементами системи.

Для реалізації алгоритмічної частини системи використовується бібліотека **NetworkX** [11] – потужний інструмент для створення, маніпулювання та аналізу складних мережевих структур у Python. NetworkX надає готові реалізації класичних алгоритмів пошуку найкоротшого шляху, зокрема алгоритму Дейкстри, який використовується як базовий алгоритм для побудови маршрутів. Бібліотека OSMnx інтегрується з NetworkX, перетворюючи дані з OpenStreetMap у графи NetworkX з автоматичним розрахунком ваг ребер на основі довжини доріг та обмежень швидкості. Для оптимізації продуктивності при роботі з великим графом України використовується кешування завантажених даних за допомогою модуля **pickle**, що дозволяє уникнути повторного завантаження графа з OSM при кожному запуску системи.

Для підвищення точності розрахунку часу подорожі система інтегрується з **TomTom Traffic API**, який надає дані про трафік у реальному часі при обчисленні на малих маршрутах. Його обрано через високу точність даних про дорожню ситуацію в Україні, широке покриття доріг та регулярне оновлення інформації. API надає дані про поточну швидкість руху на дорожніх

сегментах, що дозволяє динамічно коригувати ваги ребер графа з урахуванням реальної дорожньої ситуації. Інтеграція TomTom Traffic API забезпечує більш реалістичні оцінки часу прибуття та дозволяє уникати маршрутів з високим рівнем завантаженості.

Для ефективного пошуку вузлів графа у заданій географічній області (bbox) та побудови підграфів використовується бібліотека **rtree** [12], яка реалізує просторове індексування. Вона дозволяє швидко знаходити об'єкти у певній географічній зоні, що критично важливо для оптимізації роботи з графом великого розміру. Завдяки R-tree індексу система може ефективно створювати підграфи для коридорного пошуку маршрутів, зменшуючи час побудови маршруту. R-tree індекс будується один раз при завантаженні графа та зберігається окремо для швидкого доступу при наступних запусках системи.

Серверна частина системи реалізована на TypeScript, що забезпечує типізацію коду та зменшує кількість помилок під час розробки. Цю мову обрано через її потужну систему типів, відмінну підтримку у сучасних IDE та можливість компіляції у JavaScript для виконання у середовищі Node.js. Для зберігання даних про побудовані маршрути використовується **MongoDB** [13, 14] – документо-орієнтована NoSQL база даних, яка має вбудовану підтримку геопросторових даних через спеціальний тип GeoJSON. MongoDB дозволяє ефективно зберігати координати маршрутів, точок заправок та іншу географічну інформацію, а також надає можливість виконувати геопросторові запити для аналізу збережених маршрутів. Вибір MongoDB обґрунтовується гнучкістю схеми даних, що важливо для зберігання маршрутів різної складності, та високою продуктивністю при роботі з таким типом даних.

Для розробки мобільного застосунку обрано **React Native** з використанням Expo (bare workflow) – популярний фреймворк для створення крос-платформних мобільних додатків. React Native дозволяє розробляти застосунки одночасно для iOS та Android, використовуючи єдину кодову базу на JavaScript/TypeScript, що значно прискорює розробку та спрощує підтримку

коду. Expo bare workflow надає доступ до нативних модулів при збереженні зручності інструментів Expo для розробки та тестування. Для візуалізації маршрутів на карті використовується **Mapbox** – сучасна картографічна платформа, яка надає високоякісні інтерактивні карти з можливістю відображення користувацьких маршрутів, маркерів та інших геопросторових об'єктів. Mapbox обрано через його гнучкість у налаштуванні зовнішнього вигляду карт, відмінну продуктивність на мобільних пристроях та зручний API для React Native.

1.4 Аналіз базових алгоритмів пошуку оптимальних маршрутів

Для пошуку оптимального шляху в системі планування маршрутів з урахуванням обмежень на запас палива слід розглянути алгоритми, які ефективно працюють з великими графами дорожніх мереж. Оскільки розроблюваний застосунок орієнтований на попереднє планування маршруту перед поїздкою, а не на покрокову навігацію в реальному часі, алгоритм повинен забезпечувати точний розрахунок маршруту з урахуванням статичних параметрів (довжина доріг, обмеження швидкості) та, для маршрутів невеликої довжини, поточного стану трафіку на момент запиту. Крім того, алгоритм має бути достатньо ефективним для роботи з графом дорожньої мережі всієї України та дозволяти інтеграцію додаткових обмежень, зокрема автоматичне планування зупинок для заправки.

Алгоритм Дейкстри (Dijkstra's algorithm)

Алгоритм Дейкстри – це один з найвідоміших алгоритмів для пошуку найкоротшого шляху в графах [3, 15]. Він був розроблений нідерландським вченим Едсгером Дейкстрою у 1956 році і використовується для знаходження найкоротшого шляху від однієї вершини (джерела) до всіх інших вершин у графі, де всі ребра мають невід'ємні ваги. Алгоритм працює за жадібним принципом (greedy approach) [16], що означає, що він завжди обирає поточний

найбільш перспективний крок, і використовує пріоритетну чергу для ефективного вибору вершини з найменшою відстанню.

Алгоритм Дейкстри широко застосовується в системах планування маршрутів для визначення оптимальних шляхів на основі довжини доріг та обмежень швидкості. Це робить його особливо придатним для застосунків, які розраховують маршрути на основі статичних або повільно змінюваних параметрів дорожньої мережі.

Розглянемо покрокове пояснення алгоритму:

1. Ініціалізація:

- 1.1. Встановлюємо початкову вершину.
- 1.2. Встановлюємо її відстань до самої себе рівною 0.
- 1.3. Для всіх інших вершин відстань встановлюємо нескінченність, оскільки шлях поки що невідомий.
- 1.4. Додаємо всі вершини до множини невідвіданих вершин (пріоритетна черга).

2. Вибір поточної вершини:

- 2.1. Вибираємо серед невідвіданих вершин ту, яка має найменшу поточну відстань.
- 2.2. Якщо ця відстань дорівнює ∞ , алгоритм завершується (до даної вершини немає шляху).

3. Оновлення сусідів:

- 3.1. Перебираємо всі вершини, які з'єднані з поточною.
- 3.2. Якщо через поточну вершину можна дістатися до сусіда швидше, ніж відомий на цей момент найкоротший шлях, оновлюємо цю відстань.

4. Повторення:

- 4.1. Позначаємо поточну вершину як оброблену (видаляємо з множини невідвіданих).
- 4.2. Переходимо до наступної вершини з найменшою поточною відстанню.

4.3. Повторюємо процес, поки всі вершини не будуть оброблені або поки не знайдемо шлях до необхідної вершини.

Для задачі планування маршрутів з обмеженнями алгоритм Дейкстри має низку переваг. Він гарантує знаходження оптимального маршруту, якщо всі ваги ребер невід’ємні, що виконується для дорожніх мереж з часом або відстанню як вагами. Алгоритм добре масштабується на великі графи та може ефективно працювати з дорожньою мережею України за умови застосування оптимізацій простору пошуку. Крім того, його просто інтегрувати з додатковими обмеженнями, такими як планування зупинок для заправки, оскільки базова логіка пошуку залишається незмінною.

Основним недоліком алгоритму Дейкстри є те, що він обчислює найкоротші шляхи від початкової вершини до всіх інших вершин графа, навіть якщо потрібен шлях лише до однієї конкретної точки призначення. Це може призвести до зайвих обчислень, особливо на великих графах. На графі розміром мільйон вузлів без оптимізацій простору пошуку час розрахунку маршруту може сягати декількох хвилин, що є неприйнятним для інтерактивного застосунку. Також алгоритм не має вбудованих механізмів для врахування напрямку до цільової точки, що може призводити до дослідження вершин у напрямках, протилежних кінцевій меті.

Алгоритм A* (A-star)

Алгоритм A* є одним із найефективніших алгоритмів для пошуку найкоротшого шляху у зважених графах. Він був розроблений у 1968 році Пітером Гартом, Нільсом Нільсоном і Бертом Рафелем і поєднує в собі елементи алгоритму Дейкстри та жадібного пошуку за евристикою. Основна ідея A* полягає в тому, щоб комбінувати фактичну відстань пройденого шляху (g) і передбачувану відстань до цілі (h), що дозволяє ефективно направляти пошук у бік цільової точки [17].

Алгоритм A* використовує функцію оцінки вартості шляху:

$$f(n) = g(n) + h(n), \quad (1.1)$$

де $g(n)$ – фактична вартість шляху від початкової вершини до поточної,

$h(n)$ – евристична оцінка вартості шляху від поточної вершини до кінцевої точки.

Евристична функція (зазвичай евклідова або манхеттенська відстань) допомагає пріоритизувати найбільш перспективні вузли, що дозволяє алгоритму уникати зайвих обчислень у напрямках, які віддаляються від цілі.

Розглянемо покрокове пояснення алгоритму:

1. Ініціалізація:

1.1. Встановлюємо початкову вершину.

1.2. Встановлюємо її вартість шляху $g = 0$.

1.3. Для всіх інших вершин $g = \infty$.

1.4. Додаємо початкову вершину в чергу пріоритетів.

2. Вибір поточної вершини:

2.1. Обираємо вершину з найменшим значенням $f(n)$ у відкритій множині.

2.2. Якщо ця вершина – цільова, алгоритм завершується.

3. Оновлення сусідів:

3.1. Для кожного сусіда поточної вершини обчислюємо нове значення $g(n)$.

3.2. Якщо новий шлях до сусіда коротший, ніж записаний раніше, оновлюємо його вартість і додаємо до черги пріоритетів.

4. Повторення:

4.1. Продовжуємо вибір наступної вершини з найменшим $f(n)$, поки не знайдемо шлях або не вичерпаємо всі можливі варіанти.

Головною перевагою алгоритму A^* є те, що він направляє пошук у бік цільової точки, що зменшує кількість досліджуваних вершин порівняно з алгоритмом Дейкстри. Це робить його особливо ефективним для пошуку маршруту між двома конкретними точками. У системах планування маршрутів A^* може значно прискорити обчислення, особливо на довгих маршрутах, де евристика ефективно відсіває непотрібні напрямки пошуку. Алгоритм також може динамічно враховувати різні параметри дорожньої

мережі, такі як поточний стан трафіку, якщо ці дані доступні на момент розрахунку.

Проте A^* має певні обмеження, які можуть вплинути на його застосування в системі планування маршрутів з обмеженнями. Ефективність алгоритму сильно залежить від вибору евристичної функції – для складних дорожніх мереж з різними типами доріг та обмеженнями швидкості проста евклідова відстань може давати неточні оцінки, що знижує ефективність пошуку [18]. Алгоритм може споживати більше пам'яті порівняно з Дейкстрою через необхідність зберігання додаткових евристичних оцінок для кожної вершини. На графах великого розміру, таких як дорожня мережа України, A^* все одно потребує значних обчислювальних ресурсів, особливо якщо евристика недостатньо точна або граф не оптимізований.

Для розроблюваної системи необхідно обрати базовий алгоритм пошуку найкоротшого шляху, який буде використовуватися як фундамент для подальшої інтеграції додаткової логіки врахування обмежень. Враховуючи специфіку задачі, де маршрут розраховується один раз на етапі планування, а основний акцент робиться на автоматичному плануванні зупинок для заправки, ключовими критеріями вибору є: гарантована оптимальність результату, ефективність роботи з графами різного розміру, та можливість інтеграції з додатковою логікою обмежень.

Було обрано реалізувати комбінацію двох алгоритмів з автоматичним вибором залежно від характеристик маршруту. Алгоритм A^* з евклідовою евристикою обрано як основний базовий алгоритм для більшості маршрутів. Використання A^* виправдане тим, що евристична функція на основі відстані Гаверсайна ефективно направляє пошук до цільової точки, що зменшує кількість досліджуваних вузлів порівняно з чистим алгоритмом Дейкстри. Це надзвичайно важливо для маршрутів середньої та великої довжини, де навіть при застосуванні методів оптимізації простору пошуку підграф все ще може містити значну кількість вузлів. Евклідова відстань як евристична функція є

допустимою, оскільки вона ніколи не переоцінює реальну відстань до цілі, що гарантує знаходження оптимального маршруту.

Алгоритм Дейкстри використовується як альтернатива для специфічних випадків коротких міських маршрутів, де планується інтеграція з даними про трафік від TomTom Traffic API. У таких маршрутах після застосування методів оптимізації простору пошуку (які розглядаються в наступному підрозділі) підграф містить невелику кількість вузлів, тому евристична функція A^* не дає суттєвого виграшу в швидкості. Водночас, для коротких маршрутів важливою є можливість точного врахування поточної дорожньої ситуації, що досягається шляхом інтеграції з зовнішнім API. Простота алгоритму Дейкстри полегшує таку інтеграцію та забезпечує передбачувану поведінку при роботі з динамічними вагами ребер, що відображають реальний стан трафіку.

Важливо наголосити, що вибір між Дейкстрою та A^* не є основним методом оптимізації швидкодії системи для графів великого розміру. На графі дорожньої мережі України навіть A^* з оптимальною евристикою не може забезпечити прийнятний час розрахунку без додаткових оптимізацій простору пошуку. Тому критично важливим є застосування методів оптимізації, які зменшують кількість вузлів, що розглядаються алгоритмом, до прийнятного рівня, які ми розглянемо в наступному розділі.

Таким чином, для базового алгоритму пошуку шляху обрано алгоритм A^* як основний (для маршрутів середньої та великої довжини з оптимізацією заправок) та алгоритм Дейкстри як допоміжний (для коротких міських маршрутів з інтеграцією трафіку).

Висновки до першого розділу

У першому розділі було проведено комплексний аналіз предметної області планування автомобільних маршрутів з урахуванням обмежень на запас палива та визначено основні напрямки розробки системи.

Сформульовано основне завдання дослідження, яке полягає у створенні ефективного алгоритмічного рішення для оптимізації маршрутів з автоматичним плануванням зупинок для заправки та врахуванням трафіку на міських маршрутах. Визначено ключові етапи роботи та очікувані результати – систему для планування оптимальних маршрутів, що складається з серверної частини для обчислень на графі дорожньої мережі України та мобільного застосунку для взаємодії з користувачем.

Проведено огляд існуючих систем планування та виявлено, що жодна з них не надає автоматичного планування зупинок для заправки з урахуванням параметрів конкретного автомобіля. Визначено основну відмінність розроблюваного рішення – фокус на детальному плануванні маршруту перед поїздкою з можливістю збереження історії та аналізу статистики.

Обґрунтовано вибір технологічного стеку для багатоваріантної архітектури системи: Python з Flask та NetworkX для алгоритмічної частини, OpenStreetMap як джерело даних про дорожню мережу України та заправні станції, TomTom Traffic API для отримання даних про трафік, TypeScript з MongoDB для серверної частини, React Native з Mapbox для мобільного застосунку.

Проаналізовано базові алгоритми пошуку оптимальних маршрутів та обрано комбінований підхід: алгоритм A* з евристикою Гаверсайна як основний для маршрутів середньої та великої довжини, алгоритм Дейкстри для коротких міських маршрутів з інтеграцією трафіку.

РОЗДІЛ 2 РОЗРОБКА АЛГОРИТМУ ОПТИМАЛЬНОГО МАРШРУТУ

2.1 Формування графа мережі та оптимізація простору пошуку

Для побудови оптимального маршруту з урахуванням обмежень на паливо необхідно сформувати транспортний граф, який відображає дорожню мережу України. Якість та повнота вихідних даних безпосередньо впливають на точність розрахунків і ефективність алгоритмів пошуку маршруту. Основною проблемою при роботі з великомасштабними дорожніми мережами є обчислювальна складність – повний граф доріг України містить близько мільйона вузлів, що робить застосування класичних алгоритмів пошуку шляху надто повільним для практичного використання. Вирішення цієї проблеми потребує комплексного підходу, що включає вибір оптимального джерела даних, ефективні методи кешування та застосування просторових оптимізацій, які дозволяють скоротити час обчислень з декількох хвилин до 3-15 секунд на реальних маршрутах.

Для формування транспортного графа використовується OpenStreetMap (OSM) як джерело даних про дорожню мережу України та бібліотеки OSMnx і NetworkX для їх обробки (детальне обґрунтування вибору наведено у розділі 1.2). Завантажений граф містить вузли, що відображають перехрестя та поворотні точки, та відповідну кількість ребер, що представляють дорожні сегменти різних типів. Структура даних OSM, включаючи атрибути вузлів (географічні координати, кількість вулиць що сходяться) та ребер (довжина сегменту, тип дороги, обмеження швидкості, напрямок руху), забезпечує повноту інформації для навігаційних застосувань [19, 20]. Кожне ребро графа має вагу, розраховану на основі довжини дорожнього сегменту та дозволеної швидкості руху, що дозволяє алгоритмам пошуку шляху оптимізувати маршрут за часом подорожі.

Дані про розташування заправних станцій також отримуються з OpenStreetMap через теги {"amenity": "fuel"}. OSM містить інформацію про понад 3000 заправних станцій на території України з географічними

координатами, що є достатнім для ефективного планування зупинок на маршрутах будь-якої довжини. Використання єдиного джерела даних для доріг та заправок забезпечує географічну узгодженість та спрощує інтеграцію цих елементів у єдину систему планування маршрутів.

Процес завантаження графа з OSM є ресурсоємним і може займати від декількох хвилин до години, тому при першому запуску, тому для оптимізації повторних запусків системи реалізовано механізм кешування з використанням модуля `pickle`, який серіалізує граф у бінарний формат та зберігає для швидкого подальшого доступу:

```
def download_and_cache_ukraine_graph(cache_file='ukraine_graph.pkl'):
    if os.path.exists(cache_file):
        print(f"Граф з кешу...")
        with open(cache_file, 'rb') as f:
            G = pickle.load(f)
        print(f"    ✓ {len(G.nodes):,} вузлів")
        return G

    print("Завантаження графа з OSM...")
    G = ox.graph_from_place('Ukraine', network_type='drive', simplify=True)

    with open(cache_file, 'wb') as f:
        pickle.dump(G, f, protocol=pickle.HIGHEST_PROTOCOL)

    return G
```

Параметр `network_type='drive'` забезпечує вибір лише доріг, доступних для автомобільного руху, а `simplify=True` виконує спрощення графа шляхом видалення проміжних вузлів на прямих дорожніх сегментах без перехресть, що зменшує розмір графа без втрати точності маршрутів.

Окрім дорожньої мережі, система потребує інформації про розташування заправних станцій для планування зупинок на паливо. Дані про заправки також отримуються з OSM через теги `{"amenity": "fuel"}`, що забезпечує узгодженість координатних систем та актуальність інформації. Всі заправні станції містять географічні координати. Використання єдиного джерела даних як для доріг, так і для заправок є критично важливим для коректної прив'язки заправних станцій до вузлів дорожньої мережі.

Робота з повним графом дорожньої мережі України створює серйозні обчислювальні проблеми. Наївні алгоритми пошуку найкоротшого шляху, такі як Dijkstra або A*, навіть при оптимальній реалізації, вимагають чимало часу (як для використання в застосунку) для обчислення маршруту між віддаленими містами. Основна причина полягає у необхідності обробляти надмірно велику кількість вузлів, більшість з яких не є релевантними для конкретного маршруту. Наприклад, при плануванні маршруту Київ-Львів немає потреби аналізувати дороги у Харкові чи Одесі. Тому ключовою оптимізацією є зменшення простору пошуку шляхом виділення підграфа, який містить лише релевантні для маршруту вузли.

Просторова індексація з використанням R-tree

Для ефективного виділення підграфа застосовано просторову індексацію з використанням структури даних R-tree. R-tree – це деревоподібна структура даних, розроблена спеціально для індексації просторових об'єктів, яка широко використовується в географічних інформаційних системах та базах даних для оптимізації просторових запитів [12]. Основна ідея R-tree полягає в ієрархічній організації просторових об'єктів, де кожен вузел дерева містить мінімальний обмежувальний прямокутник (Minimum Bounding Rectangle, далі MBR) для групи дочірніх об'єктів. Листові вузли містять посилання на фактичні просторові об'єкти (у нашому випадку – вузли дорожнього графа), а внутрішні вузли формують ієрархію MBR, що дозволяє ефективно відсіювати нерелевантні області при пошуку.

При виконанні просторового запиту «знайти всі вузли в межах заданої географічної області», R-tree дозволяє швидко відкинути цілі гілки дерева, чий MBR не перетинаються з областю пошуку, без необхідності перевіряти кожен окремий об'єкт. Це забезпечує логарифмічну складність пошуку $O(\log n)$ у середньому випадку, на відміну від лінійної складності $O(n)$ при наївному переборі всіх вузлів графа. Для графа з такою великою кількістю вузлів це означає різницю між перевіркою приблизно 20 вузлів дерева замість мільйону

окремих точок, що дає прискорення у десятки тисяч разів для операції пошуку вузлів у заданій області (рис. 2.1).

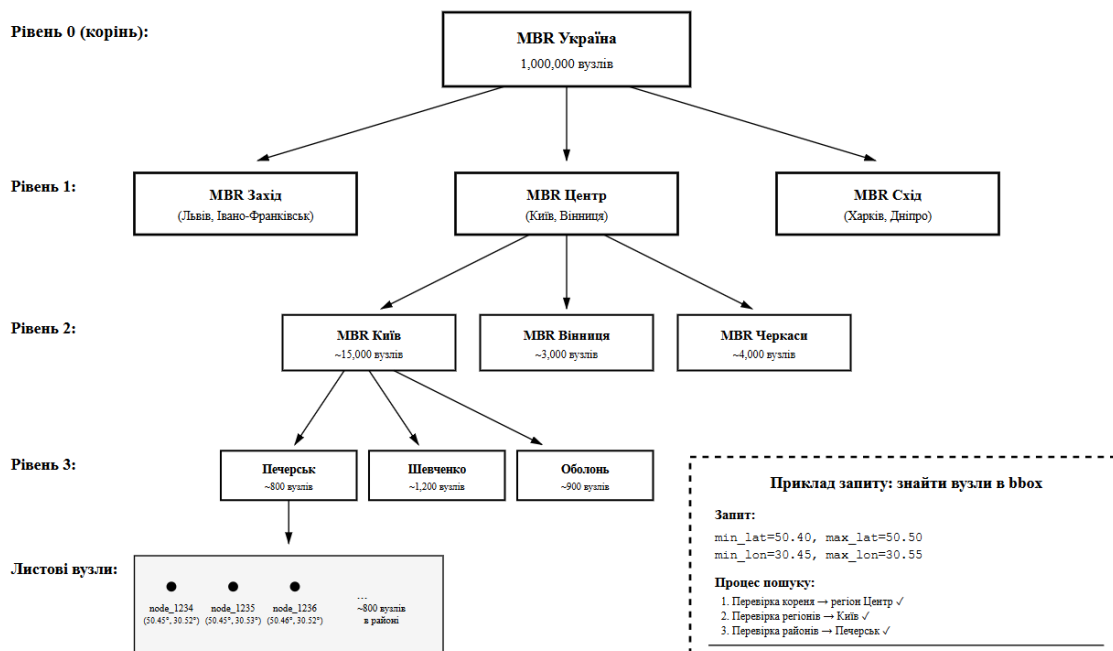


Рис. 2.1 – Ієрархічна структура R-tree для просторового пошуку вузлів графа

У розробленій системі R-tree індекс будується один раз при завантаженні графа і містить всі вузли дорожньої мережі з їх географічними координатами. Індекс зберігається на диску у форматі .dat та .idx файлів, що дозволяє уникнути перебудови при наступних запусках системи:

```
from rtree import index

def build_rtree_index(G):
    print("Створення R-tree індексу...")
    idx = index.Index()

    for node, data in G.nodes(data=True):
        try:
            lon, lat = data['x'], data['y']
            idx.insert(int(node), (lon, lat, lon, lat))
        except (KeyError, ValueError):
            continue

    return idx
```

Стратегія обмежувальної рамки з адаптивним буфером

Завдяки R-tree індексу система може швидко виділити підграф, що містить лише вузли в межах географічної області навколо планованого маршруту. Ключовим питанням є визначення форми та розміру цієї області.

Для цього використовується підхід обмежувальної рамки (bbox) – прямокутної географічної області, що охоплює початкову та кінцеву точки маршруту з додатковим буфером навколо них. Розмір буфера визначає, наскільки далеко від прямої лінії між точками система шукатиме альтернативні шляхи.

В процесі розробки було розглянуто два альтернативні підходи до формування області пошуку: стандартна bbox та corridor (вузький коридор вздовж напрямку руху) [21]. Теоретично, corridor-підхід обіцяв значне зменшення площі пошуку – для прикладу маршруту довжиною 1000 км, bbox з буфером 25 км охоплює площу близько 50 000 км², тоді як вузький corridor міг би зменшити це до 25 000 км² (50% економії). Однак експериментальні випробування на реальних маршрутах України виявили критичну проблему з цим підходом.

Основна проблема corridor-стратегії полягає у геометричній обмеженості для складних маршрутів. Реальні оптимальні шляхи між містами України часто суттєво відхиляються від прямої лінії через географічні особливості – гори, річки, міста, які потребують об'їзду. Наприклад, для маршруту Ужгород-Одеса пряма відстань становить 676 км, але реальний оптимальний шлях відхиляється на 35-65 км від прямої через необхідність проїзду через гірські перевали, обхід річок та проходження через міста з розвиненою дорожньою інфраструктурою. Вузький коридор фіксованої ширини не здатний передбачити всі такі відхилення, що призводить до проблем зв'язності графа – підграф може не містити шляху між початковою та кінцевою точками, хоча такий шлях існує у повному графі. Забезпечення топологічної зв'язності є критичним фактором для надійності систем планування маршрутів [22, 23].

Спроби збільшити ширину corridor для компенсації потенційних об'їздів нівелюють його теоретичну перевагу над bbox, оскільки широкий corridor за площею наближається до bbox, але при цьому вносить додаткову складність у реалізацію. Більше того, для забезпечення надійності знаходження маршруту corridor-підхід потребував би додаткової логіки динамічного розширення

ширини або fallback-механізмів, що знижує передбачуваність продуктивності системи. Експериментальні результати показали, що альтернативний corridor підхід, незважаючи на теоретичні переваги, виявився ненадійним для складних маршрутів через проблеми зв'язності графа та не дав значного приросту продуктивності (різниця <10%) порівняно зі стабільною роботою bbox-підходу. Нижче наведено порівняння обох підходів (рис. 2.2):

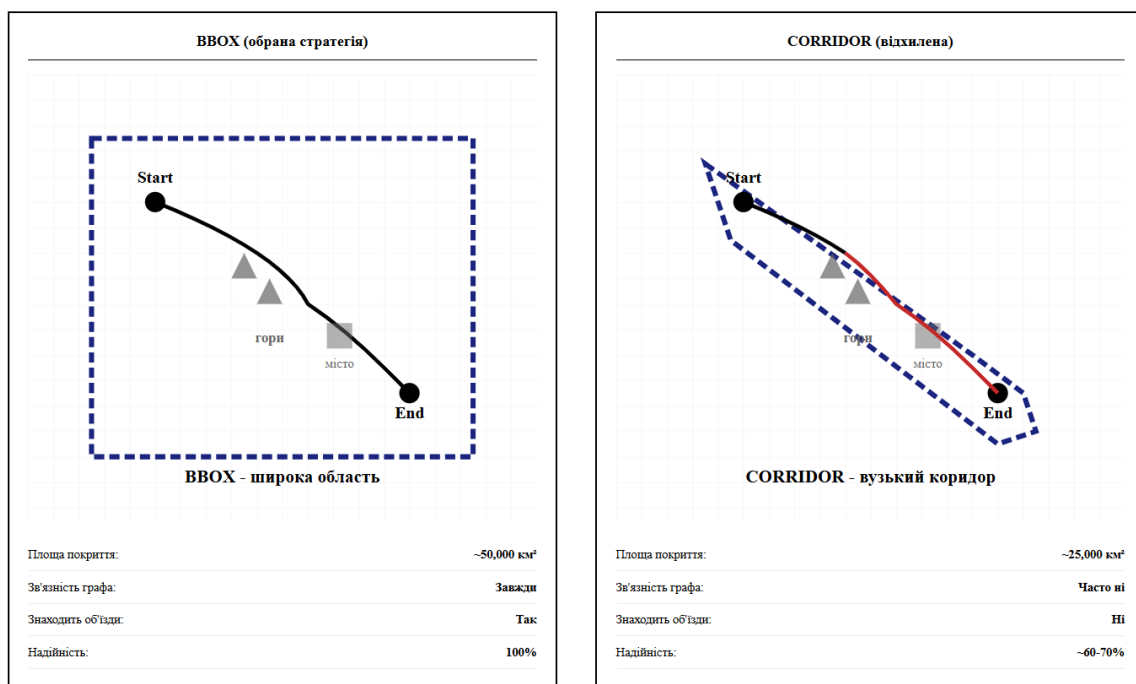


Рис. 2.2 – Порівняння bbox та corridor, візуалізація проблеми corridor

Тому для системи було обрано bbox-підхід з адаптивним буфером як оптимальне рішення, що забезпечує баланс між ефективністю та надійністю. Адаптивний підхід до вибору розміру буфера базується на прямій відстані між початковою та кінцевою точками, розрахованій за формулою гаверсинусів [24]. Аналіз реальних маршрутів по Україні показав, що для коротких маршрутів (<50 км) достатньо буфера 10 км, для середніх маршрутів (50-150 км) – 15 км, а для довгих маршрутів (>150 км) можна використовувати буфер 20-25 км.

Такий адаптивний підхід забезпечує гарантовану топологічну зв'язність, що працює для всіх маршрутів завдяки достатній ширині буфера для покриття реальних відхилень шляху. Обчислювальна ефективність досягається через єдиний R-tree запит замість множинних перевірок, а геометрична повнота

покриває всі можливі шляхи в заданій області, включаючи об'їзди та альтернативні траси. Практична надійність методу підтверджується стабільною продуктивністю 2-10 секунд на створення підграфа, що дозволяє легко масштабувати рішення для різних регіонів шляхом коригування параметрів буфера.

Створення підграфа через R-tree виконується наступним чином:

```
# Адаптивний вибір розміру буфера
if straight_distance < 50:
    buffer_km = 10
elif straight_distance < 150:
    buffer_km = 15
else:
    buffer_km = 20

# Створення bbox з буфером
buffer_deg = buffer_km / 111.0 # Конвертація км в градуси
min_lat = min(lat1, lat2) - buffer_deg
max_lat = max(lat1, lat2) + buffer_deg
min_lon = min(lon1, lon2) - buffer_deg
max_lon = max(lon1, lon2) + buffer_deg

# Швидкий пошук вузлів через R-tree
nodes_in_bbox = list(rtree_idx.intersection(
    (min_lon, min_lat, max_lon, max_lon)
))

# Створення підграфа
G_sub = G.subgraph(nodes_in_bbox).copy()
```

Для забезпечення надійності системи реалізовано механізм перевірки зв'язності підграфа. Якщо обраний буфер не забезпечує існування шляху між початковою та кінцевою точками (що може статися у виняткових випадках з нестандартною топологією доріг), система автоматично розширює буфер на 10 км або переключається на використання повного графа з ширшим буфером. Це гарантує, що система завжди знайде маршрут, навіть у складних випадках.

Стратегія оптимізації подвійного графа

Аналіз реальних маршрутів показав ще одну важливу закономірність: короткі міські маршрути (<100 км) потребують детального врахування всіх доріг, включаючи місцеві вулиці та провулки, тоді як довгі міжміські маршрути (>100 км) переважно використовують основні траси –

автомагістралі, національні та регіональні дороги. Виходячи з цього спостереження, було реалізовано логіку dual-graph стратегії (стратегії подвійного графа), яка передбачає використання двох різних наборів даних залежно від характеру маршруту.

Основна ідея dual-graph підходу полягає у попередній фільтрації дорожньої мережі за категоріями доріг. Замість того, щоб зберігати один великий граф з усіма дорогами і потім фільтрувати його під час кожного запиту, система формує два варіанти графа один раз при завантаженні: повний граф з усіма дорогами для коротких маршрутів та відфільтрований граф лише з основними дорогами (~120 000 вузлів) для довгих маршрутів. Кожен з графів зберігається окремо і має власний R-tree індекс для швидкого просторового пошуку.

Відфільтрований граф створюється шляхом відбору ребер за типом дороги в класифікації OSM. До основних доріг відносяться категорії motorway (автомагістралі), trunk (траси державного значення), primary (дороги першої категорії), secondary (дороги другої категорії), tertiary (дороги третьої категорії) та відповідні з'єднувальні ділянки (links). Процес фільтрації перебирає всі ребра повного графа і копіює у новий граф лише ті, що відповідають зазначеним категоріям:

```
major_types = {
    'motorway', 'trunk', 'primary', 'secondary', 'tertiary',
    'motorway_link', 'trunk_link', 'primary_link',
    'secondary_link', 'tertiary_link'
}

def filter_major_roads_graph(G):
    print("Фільтрація основних доріг...")
    G_major = nx.MultiDiGraph()

    for u, v, k, data in G.edges(keys=True, data=True):
        highway = data.get('highway', '')

        if isinstance(highway, list):
            highway = highway[0] if highway else ''

        if highway in major_types:
            if not G_major.has_node(u):
```

```

        G_major.add_node(u, **G.nodes[u])
    if not G_major.has_node(v):
        G_major.add_node(v, **G.nodes[v])

    G_major.add_edge(u, v, k, **data)

return G_major

```

Процес фільтрації виконується один раз при першому завантаженні системи і займає 1-2 хвилини, після чого результат зберігається в окремий кеш-файл для швидкого завантаження при наступних запусках. Фільтрація зменшує кількість вузлів приблизно на 75-85%, що суттєво прискорює обчислення маршрутів на великі відстані, оскільки алгоритми пошуку шляху працюють з набагато меншим простором вузлів.

Система автоматично обирає відповідний граф на основі прямої відстані між початковою та кінцевою точками маршруту. Якщо пряма відстань менша за 100 км, використовується повний граф з усіма дорогами, оскільки короткі маршрути часто проходять через міські райони, де важливі деталі місцевих вулиць і де оптимальний шлях може пролягати через другорядні дороги для обходу заторів чи скорочення відстані. Якщо пряма відстань становить 100 км або більше, використовується відфільтрований граф основних доріг, оскільки довгі міжміські маршрути переважно проходять магістральними трасами, а врахування місцевих вулиць лише сповільнює обчислення без покращення якості результату – водії на довгих дистанціях не їдуть провулками, а використовують швидкісні траси.

Ця адаптивна логіка дозволяє оптимізувати як швидкість обчислень, так і точність побудованих маршрутів. Для коротких маршрутів система враховує всі можливі альтернативи включно з місцевими дорогами, що важливо для міських поїздок. Для довгих маршрутів система працює швидше завдяки меншому графу, при цьому не втрачаючи якості, оскільки оптимальний міжміський маршрут завжди проходить основними трасами. Кожен з двох графів має власний R-tree індекс, що забезпечує швидкий просторовий пошук в обох випадках без необхідності перебудови індексу або додаткових обчислень.

Інтеграція заправних станцій

Завершальним етапом формування графа є інтеграція заправних станцій у дорожню мережу. Координати заправок з OSM необхідно прив'язати до вузлів дорожнього графа, оскільки алгоритми планування маршрутів працюють з графовими структурами, а не з довільними географічними точками. Для кожної заправної станції визначається найближчий вузол графа за допомогою функцій бібліотеки OSMnx, що забезпечує коректне включення заправок у процес планування зупинок.

При великій кількості заправок (понад 100) використовується пакетна обробка, яка суттєво швидша завдяки векторизованим обчисленням – замість послідовного пошуку найближчого вузла для кожної заправки окремо, система обробляє всі заправки одним запитом. Крім того, для кожного конкретного маршруту виконується попередня фільтрація заправок за географічною областю bbox – використовуються лише ті заправки, що знаходяться в межах обмежувальної рамки навколо маршруту. Це зменшує обсяг обчислень при плануванні зупинок на паливо, оскільки система не розглядає заправки, що знаходяться далеко від планованого шляху і не можуть бути релевантними для маршруту.

Таким чином, формування та оптимізація транспортного графа включає кілька послідовних етапів: завантаження даних з OSM та їх кешування, побудову R-tree індексів для швидкого просторового пошуку, створення відфільтрованого графа основних доріг для реалізації dual-graph стратегії, адаптивне виділення підграфів через bbox з оптимальним буфером та інтеграцію заправних станцій. Комбінація цих оптимізацій дозволила скоротити час обчислення маршрутів до прийнятного на реальних даних дорожньої мережі України [25], забезпечуючи при цьому якість та надійність результатів за рахунок гарантованої зв'язності графа та повноти покриття можливих шляхів. Це створило основу для стабільної та передбачуваної роботи алгоритмів пошуку маршруту незалежно від регіону чи щільності дорожньої інфраструктури.

2.2 Побудова оптимального шляху на підграфі

Після формування оптимізованого підграфа наступним етапом є обчислення оптимального шляху між початковою та кінцевою точками. Як було обґрунтовано в підрозділі 1.4, для базового пошуку шляху обрано комбінацію алгоритмів Дейкстри та A^* з адаптивним вибором залежно від характеристик маршруту.

Для їх реалізації використовується бібліотека NetworkX, яка надає оптимізовані реалізації класичних алгоритмів на графах. Це дозволяє зосередитися на логіці вибору стратегії та інтеграції додаткових обмежень, не витрачаючи ресурси на низькорівневу реалізацію базових алгоритмів.

Алгоритм Дейкстри реалізовано через функцію `shortest_path` бібліотеки NetworkX [26]:

```
def find_route_simple(G, start_node, end_node):  
    """Пошук найкоротшого шляху алгоритмом Дейкстри"""  
    return nx.shortest_path(G, start_node, end_node, weight='length')
```

Параметр `weight='length'` вказує, що алгоритм мінімізує сумарну довжину ребер у метрах. Функція повертає список ідентифікаторів вузлів, які утворюють оптимальний шлях від початкової до кінцевої точки. Ця реалізація використовується для коротких міських маршрутів (до 50 км), де підграф після застосування методів оптимізації містить відносно невелику кількість вузлів (5,000-15,000) і алгоритм виконується за 0.5-1.5 секунди.

Алгоритм A^* реалізовано через функцію `astar_path` з власною евристичною функцією:

```
def find_route_medium(G, start_node, end_node):  
    """Пошук шляху алгоритмом  $A^*$  з евристикою Гаверсайна"""  
  
    def euclidean_heuristic(node, target):  
        """Евристична функція на основі відстані Гаверсайна"""  
        n_lat, n_lon = G.nodes[node]['y'], G.nodes[node]['x']  
        t_lat, t_lon = G.nodes[target]['y'], G.nodes[target]['x']  
        return haversine_coords(n_lat, n_lon, t_lat, t_lon)  
  
    return nx.astar_path(G, start_node, end_node,  
                        heuristic=euclidean_heuristic,  
                        weight='length')
```

Евристична функція обчислює пряму відстань між поточним вузлом та цільовою точкою за формулою Гаверсайна, яка враховує сферичність Землі. Вибір саме формули Гаверсайна як евристичної функції обумовлений кількома факторами [26]. По-перше, для алгоритму A^* критично важливою є допустимість евристики – вона ніколи не повинна переоцінювати реальну відстань до цілі, що гарантує знаходження оптимального маршруту. Формула Гаверсайна обчислює найкоротшу відстань по поверхні сфери (геодезичну лінію), яка завжди менша або дорівнює реальній відстані по дорогах, оскільки дороги не можуть бути коротшими за пряму лінію між двома точками. По-друге, на відміну від простої евклідової відстані в декартових координатах, формула Гаверсайна коректно враховує кривизну Землі, що особливо важливо для міжміських маршрутів довжиною понад 100-200 км, де спотворення від проекції географічних координат на площину стають суттєвими. По-третє, обчислення за формулою Гаверсайна є достатньо швидким (кілька тригонометричних операцій), що не створює значних накладних витрат при обчисленні евристики для тисяч вузлів під час пошуку маршруту.

Для двох точок з координатами (φ_1, λ_1) та (φ_2, λ_2) , де φ – широта, λ – довгота, відстань обчислюється за формулою:

$$a = \sin^2\left(\frac{\Delta\varphi}{2}\right) + \cos\varphi_1 \cdot \cos\varphi_2 \cdot \sin^2\left(\frac{\Delta\lambda}{2}\right), \quad (2.1)$$

$$c = 2 \cdot \operatorname{atan2}(\sqrt{a}, \sqrt{1-a}), \quad (2.2)$$

$$d = R \cdot c, \quad (2.3)$$

де $R = 6,371,000$ м – середній радіус Землі, $\Delta\varphi = \varphi_2 - \varphi_1$, $\Delta\lambda = \lambda_2 - \lambda_1$.

Реалізація формули Гаверсайна має наступний вигляд:

```
def haversine_coords(lat1, lon1, lat2, lon2):
    """Обчислення відстані між координатами за формулою Гаверсайна"""
    R = 6371000

    phi1, phi2 = radians(lat1), radians(lat2)
    dphi = radians(lat2 - lat1)
    dlambd = radians(lon2 - lon1)

    a = sin(dphi/2)**2 + cos(phi1) * cos(phi2) * sin(dlambd/2)**2
    c = 2 * atan2(sqrt(a), sqrt(1-a))
```

```
return R * c
```

Ця евристика є допустимою, оскільки пряма відстань між двома точками на поверхні Землі завжди менша або дорівнює реальній відстані по дорогах, що гарантує знаходження оптимального маршруту алгоритмом A*. Функція `haversine_coords` використовується не лише як евристика для цього алгоритму, але й для швидкої оцінки прямої відстані між точками при виборі стратегії оптимізації, а також для розрахунку запасу палива на основі відстані та витрати палива автомобіля.

Вибір між алгоритмом Дейкстри та A* здійснюється автоматично на основі характеристик маршруту. Оптимальний підхід для кожного запиту визначається через функцію `select_optimization_strategy`, яка аналізує пряму відстань між точками та поточний запас палива:

```
def select_optimization_strategy(distance_km, initial_fuel_range_km):
    if distance_km < 50:
        if TOMTOM_AVAILABLE:
            return 'simple', True, f"Міський маршрут ({distance_km:.0f} км)
            - TomTom з реальним трафіком"
        else:
            return 'simple', False, f"Міський маршрут ({distance_km:.0f} км)
            - Dijkstra (TomTom недоступний)"

    if distance_km <= initial_fuel_range_km:
        return 'simple', False, f"Без дозаправок ({distance_km:.0f} км <
        {initial_fuel_range_km:.0f} км запасу)"

    return 'advanced', False, f"Довгий маршрут ({distance_km:.0f} км) - з
    оптимізацією дозаправок"
```

Функція повертає три значення: тип стратегії ('simple' або 'advanced'), прапорець використання TomTom API та текстове пояснення причини вибору. Критерії вибору стратегії наведено в таблиці 2.1. Для коротких міських маршрутів (до 50 км) маршрут обчислюється безпосередньо через TomTom Traffic API, який використовує власні алгоритми оптимізації з урахуванням реального трафіку. У разі недоступності TomTom API система автоматично переходить до резервного варіанту – алгоритму Дейкстри на графі OSM. Для довгих міжміських маршрутів (понад 50 км) обирається алгоритм A*, оскільки

навіть після оптимізації підграф містить значну кількість вузлів (30,000-70,000), і евристична функція ефективно направляє пошук до цільової точки, зменшуючи час обчислень у 2-3 рази порівняно з алгоритмом Дейкстри. Для таких маршрутів дані про трафік не використовуються, оскільки стан трафіку на довгих дистанціях значно змінюється протягом поїздки, що робить прогнози менш точними.

Таблиця 2.1

Критерії вибору стратегії обчислення маршруту

Відстань	Запас палива	Основний алгоритм	Резервний варіант	Обґрунтування
<50 км	Достатньо	TomTom Traffic API	Дейкстра	Реальний трафік критичний для міста
<50 км	Недостатньо	TomTom Traffic API	Дейкстра	FAST-PATH через найближчу заправку
≥50 км	Достатньо	A*	-	Евристика ефективна на великих підграфах
≥50 км	Недостатньо	A*	-	Оптимізація множинних заправок

Для коротких міських маршрутів система інтегрується з TomTom Traffic API для отримання маршрутів з урахуванням реального стану трафіку. Дане API надає не лише координати оптимального маршруту, але й детальну інформацію про час у дорозі з урахуванням поточних заторів, швидкість руху на окремих сегментах та прогнозовані затримки [27]. Інтеграція з TomTom здійснюється через функцію `find_route_tomtom`:

```
def find_route_tomtom(start_coords, end_coords, departure_time=None):
    """Отримання маршруту з реальним трафіком через TomTom API"""

    if not TOMTOM_AVAILABLE:
        return None

    result = tomtom_api.get_route_with_traffic(
        start_coords=start_coords,
        end_coords=end_coords,
        departure_time=departure_time or datetime.now(),
        route_type='fastest'
```

```
)

if result.get('success'):
    return result
else:
    return None
```

Параметр `route_type`: 'fastest' вказує API обирати найшвидший маршрут з урахуванням трафіку, а не найкоротший за відстанню, що відповідає потребам користувачів, які зазвичай прагнуть мінімізувати час у дорозі. TomTom API повертає структуровану відповідь з детальною інформацією про маршрут:

```
{
    'distance_km': 45.2,
    'duration_minutes': 52.0,                # Час з урахуванням трафіку
    'duration_no_traffic_minutes': 38.0,     # Час без заторів
    'traffic_delay_minutes': 14.0,           # Затримка через затори
    'avg_speed_kmh': 52.0,
    'route_points': [(50.4501, 30.5234), ...], # Координати маршруту
    'traffic_segments': [                    # Деталі по сегментах
        {
            'start_idx': 0,
            'end_idx': 15,
            'speed_kmh': 35,
            'delay_seconds': 180,
            'delay_magnitude': 2,            # Рівень затору (0-4)
            'category': 'MODERATE'
        }
    ]
}
```

Оскільки TomTom повертає маршрут у вигляді списку географічних координат, а система працює з графом OSM, де вузли ідентифікуються числовими ID, необхідно виконати прив'язку координат TomTom до вузлів графа OSM. Для цього використовується векторизований пошук найближчих вузлів:

```
def map_tomtom_route_to_osm(G, tomtom_route_points):
    # Кожна 5-та точка для міських маршрутів
    sample_points = tomtom_route_points[::5] if len(tomtom_route_points) > 20
    else tomtom_route_points

    # Завжди включити початок і кінець
    if len(tomtom_route_points) > 0:
        if sample_points[0] != tomtom_route_points[0]:
            sample_points.insert(0, tomtom_route_points[0])
```

```

    if sample_points[-1] != tomtom_route_points[-1]:
        sample_points.append(tomtom_route_points[-1])

try:
    # Розділити точки на масиви широт та довгот
    lats = [point[0] for point in sample_points]
    lons = [point[1] for point in sample_points]

    # Векторизований пошук найближчих вузлів (один виклик для всіх точок)
    nearest_nodes = ox.distance.nearest_nodes(G, X=lons, Y=lats)

    # Конвертація у список цілих чисел
    osm_route = [int(node) for node in nearest_nodes]

    return osm_route

except Exception as e:
    # Резервний варіант: послідовний пошук
    osm_route = []
    for lat, lon in sample_points:
        try:
            nearest_node = ox.distance.nearest_nodes(G, X=lon, Y=lat)
            osm_route.append(int(nearest_node))
        except Exception as e:
            continue

    return osm_route

```

Векторизований підхід, коли функція `nearest_nodes` обробляє всі точки одночасно, працює в 10-20 разів швидше порівняно з послідовним пошуком найближчого вузла для кожної точки окремо [28]. Для типового міського маршруту з 200-300 точками від TomTom векторизований пошук виконується швидше, ніж послідовний підхід. Додатково застосовується оптимізація через семплювання точок маршруту: для міських маршрутів береться кожна 5-та точка, що зменшує кількість точок з 200-300 до 40-60 без суттєвої втрати точності маршруту, при цьому початкова та кінцева точки завжди включаються до результату. У разі виникнення помилки при векторизованому пошуку система автоматично переходить до резервного варіанту з послідовним пошуком вузлів для кожної точки окремо.

Для міських маршрутів, де передбачається необхідність заправки, застосовується додаткова оптимізація – FAST-PATH підхід, який базується на принципах просторової оптимізації [29]. Замість того, щоб спочатку будувати

маршрут від початку до кінця, потім виявляти необхідність заправки, знаходити заправну станцію та перебудовувати маршрут, система одразу будує маршрут через найближчу від старту заправну станцію. Передбачальна оцінка необхідності заправки виконується наступним чином:

```
estimated_route_dist = estimated_dist * 1.3
needs_refuel_prediction = estimated_route_dist * 1000 > initial_range_m * 0.8

if needs_refuel_prediction and use_tomtom and estimated_dist < 50:
    # Знайти найближчу заправку від точки старту
    nearest_fuel = find_nearest_fuel_station(start_coords, fuel_filtered)

    # Одразу побудувати маршрут: Start → Fuel → End
    waypoints = [start_coords, nearest_fuel, end_coords]
    return find_route_tomtom_multi_waypoint(waypoints)
```

Коефіцієнт 1.3 враховує той факт, що реальна відстань по дорогах зазвичай на 20-40% більша за пряму відстань між точками через звивистість доріг та особливості міської забудови. Поріг 0.8 від запасу палива забезпечує запас надійності для випадків, коли маршрут виявиться дещо довшим за очікуваний і дозволяє завчасно виявити необхідність заправки без ризику вичерпання палива.

Цей підхід економить час обчислень, оскільки замість двох запитів до TomTom API виконується лише один запит з проміжною точкою. Для реалізації такого підходу TomTom Traffic API підтримує нативний multi-waypoint режим [30], який дозволяє отримати оптимальний маршрут через множину точок одним API запитом:

```
def find_route_tomtom_multi_waypoint(waypoints, departure_time=None):
    """Оптимізована маршрутизація через множину точок"""

    if not TOMTOM_AVAILABLE:
        return None

    # Перевірка підтримки нативного multi-waypoint API
    if hasattr(tomtom_api, 'get_route_with_waypoints'):
        result = tomtom_api.get_route_with_waypoints(
            waypoints=waypoints,
            departure_time=departure_time or datetime.now(),
            route_type='fastest'
        )

        if result.get('success'):
```

```
return result
```

```
# Резервний варіант: послідовні запити сегмент за сегментом  
return _build_route_sequential(waypoints, departure_time)
```

Замість послідовних запитів (Start→Fuel1, Fuel1→Fuel2, Fuel2→End), які потребують трьох окремих HTTP запитів з відповідною мережевою затримкою, multi-waypoint виконує один запит та повертає оптимізований маршрут через всі точки. Для маршруту з двома заправками це дає вигравш у часі приблизно в 2-3 рази з урахуванням мережесих затримок.

Критично важливим етапом після побудови маршруту є валідація його досяжності з урахуванням місткості паливного бака. Навіть якщо система правильно розмістила заправні станції на маршруті, може виникнути ситуація, коли відстань між двома послідовними заправками перевищує максимальний запас ходу автомобіля на повному баку. Така ситуація особливо ймовірна на гірських дорогах або у віддалених регіонах з низькою щільністю заправних станцій. Система виконує перевірку кожного сегмента маршруту між заправками:

```
segments_data = tomtom_result.get('segments_data', [])  
  
for i, segment in enumerate(segments_data):  
    seg_dist = segment['distance_km'] * 1000  
  
    if seg_dist > tank_range_m:  
        print(f"КРИТИЧНА ПОМИЛКА: Сегмент #{i+1}")  
        print(f"    Відстань: {seg_dist/1000:.1f} км")  
        print(f"    Місткість бака: {tank_range_m/1000:.1f} км")  
        print(f"    Маршрут НЕДОСЯЖНИЙ навіть з дозаправками!")  
        return None
```

Якщо хоча б один сегмент перевищує максимальний запас ходу, система повертає помилку та інформує користувача про неможливість досягти пункту призначення з поточними параметрами автомобіля.

Таким чином, система обчислення оптимального шляху на підграфі поєднує класичні алгоритми пошуку шляху (Дейкстра, A*) з сучасними джерелами даних про трафік та адаптивною стратегією вибору підходу залежно від характеристик маршруту. Інтеграція з TomTom Traffic API забезпечує точні прогнози часу в дорозі для міських маршрутів, а оптимізації

через векторизований пошук вузлів та multi-waypoint режим мінімізують накладні витрати на обчислення.

2.3 Побудова маршруту з урахуванням дозаправок

Наступним етапом є визначення оптимальних точок для дозаправки з урахуванням обмеження на запас палива. Це є ключовою модифікацією класичної задачі пошуку найкоротшого шляху, оскільки система повинна автоматично планувати зупинки для заправки таким чином, щоб гарантувати досяжність кінцевого пункту призначення при мінімальній кількості дозаправок.

Алгоритм розрахунку точок дозаправки отримує на вхід базовий маршрут у вигляді послідовності вузлів графа, поточний запас палива, місткість паливного бака та множину заправних станцій, що знаходяться на маршруті або в його околицях. Основною метою є визначення мінімальної кількості заправок, достатньої для завершення маршруту [31], з урахуванням запасу надійності для непередбачуваних ситуацій. Повний код алгоритму наведено в додатку А.

Перший етап алгоритму полягає у розрахунку кумулятивних відстаней вздовж маршруту. Для кожного вузла маршруту обчислюється загальна відстань від початкової точки до цього вузла шляхом підсумовування довжин ребер графа. Після цього виконується ідентифікація заправних станцій, які розташовані безпосередньо на маршруті, з уникненням дублікатів у випадку, коли маршрут проходить через один і той же вузол декілька разів.

Критично важливою є перевірка досяжності першої заправної станції на маршруті:

```
first_station_dist = fuel_on_route[0][2]
if first_station_dist > initial_fuel_m:
    return None
```

Якщо відстань до першої заправки перевищує поточний запас палива, маршрут є недосяжним з поточними параметрами, і система інформує

користувача про необхідність збільшити початковий запас палива або обрати альтернативний маршрут.

Наступним етапом є розрахунок адаптивного порогу безпеки, який залежить від густоти заправних станцій на маршруті [32]. Даний параметр обчислюється як кількість станцій на 100 км маршруту:

```
stations_density = len(fuel_on_route) / (total_distance / 100000)
safety_threshold = 0.25 if stations_density < 2 else 0.20
```

Для маршрутів з низькою густиною заправок (менше 2 станцій на 100 км) встановлюється більш консервативний поріг безпеки 25% від місткості бака, що забезпечує додатковий запас надійності у віддалених регіонах. Для маршрутів з вищою густиною заправок використовується поріг 20%, оскільки ризик залишитися без палива є нижчим.

Загальна структура алгоритму планування дозаправок наведена нижче (рис. 2.3), яка показує послідовність основних етапів від отримання вхідних даних до формування фінального списку точок дозаправки.

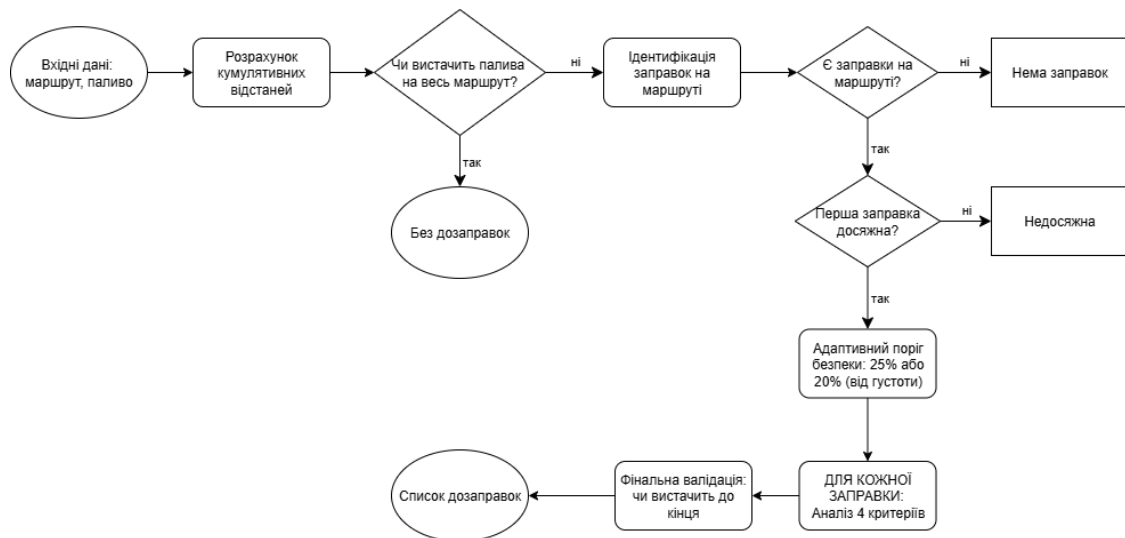


Рис. 2.3 – Загальна структура алгоритму

Основною частиною алгоритму є багатокритеріальна логіка прийняття рішення про дозаправку. Для кожної заправної станції на маршруті система послідовно перевіряє чотири критерії у порядку пріоритету. Ці критерії були розроблені на основі аналізу різних сценаріїв подорожей та забезпечують оптимальний баланс між мінімізацією кількості зупинок та гарантією безпечного завершення маршруту.

Критерій 1: Критична ситуація – заправка є обов’язковою, якщо після проїзду поточної станції неможливо дістатися ні кінцевого пункту призначення, ні наступної заправної станції. Цей критерій має найвищий пріоритет, оскільки відмова від заправки в такій ситуації призведе до неможливості завершення маршруту.

Критерій 2: Критично низький рівень палива – заправка необхідна, якщо рівень палива впаде нижче порогу безпеки і при цьому неможливо дістатися наступної заправної станції. Цей критерій забезпечує дотримання мінімального запасу палива для непередбачуваних ситуацій, таких як об’їзди або затримки в русі.

Критерій 3: Нemoжливість досягнення наступної станції – заправка виконується, якщо після проїзду поточної станції недостатньо палива для досягнення наступної заправки, навіть якщо поточний рівень палива вище порогу безпеки. Цей критерій запобігає ситуаціям, коли відстань між двома послідовними заправками перевищує поточний запас палива.

Критерій 4: Передбачальна логіка – найбільш складний критерій, який аналізує не лише можливість досягнення наступної заправки, але й прогнозує рівень палива після неї. Критерій спрацьовує у двох випадках:

- поточний рівень палива нижче 15% від місткості бака і після досягнення наступної заправки залишиться менше 10%;
- відстань до наступної заправки перевищує 40 км і після її досягнення залишиться менше 15% палива.

Ці підкритерії дозволяють уникнути ситуацій, коли пропуск поточної заправки призведе до критично низького рівня палива на наступному сегменті маршруту.

Послідовність перевірки чотирьох критеріїв прийняття рішення про дозаправку детально представлена нижче (рис. 2.4). Як видно з діаграми, критерії перевіряються послідовно, і якщо будь-який з них повертає позитивний результат, виконується дозаправка без перевірки наступних критеріїв.

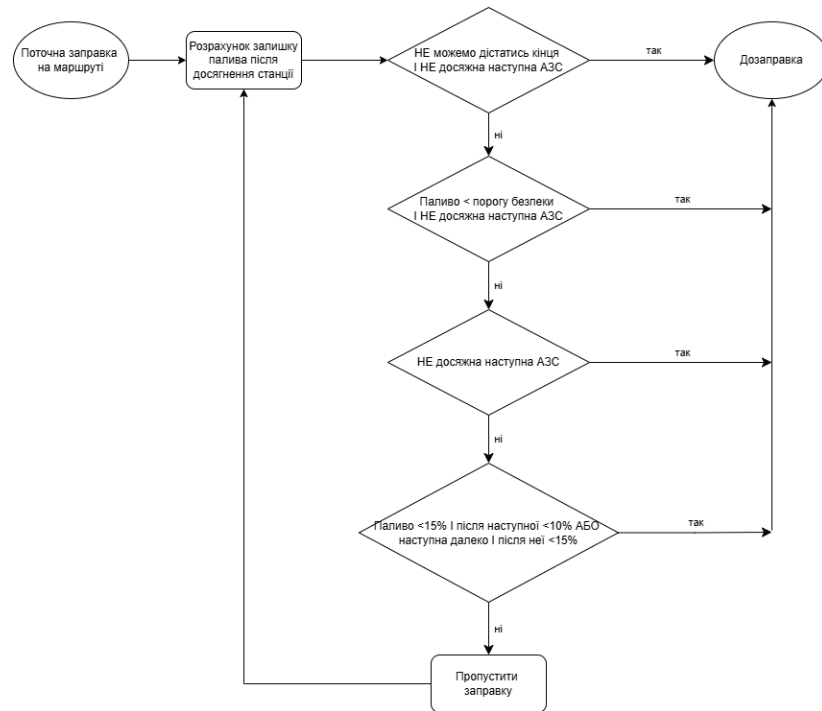


Рис. 2.4 – Логіка прийняття рішення

Алгоритм також включає аварійну логіку для випадків, коли поточний запас палива недостатній для досягнення найближчої заправки. У такій ситуації система виконує зворотний пошук попередніх заправних станцій, які ще не були використані, і вибирає найближчу досяжну. Якщо жодна з попередніх заправок не є досяжною, маршрут визнається неможливим для виконання з поточними параметрами автомобіля.

Після визначення всіх точок дозаправки виконується фінальна валідація маршруту – перевірка достатності палива для досягнення кінцевого пункту призначення від останньої точки заправки. У разі успішного проходження всіх перевірок алгоритм повертає список точок дозаправки, загальну відстань маршруту та детальну статистику, яка включає кількість дозаправок, причину кожної зупинки, густоту заправних станцій на маршруті та прогнозований залишок палива після завершення подорожі.

Таким чином, багатокритеріальний алгоритм планування дозаправок забезпечує надійне виконання маршруту з мінімальною кількістю зупинок, автоматично адаптуючись до характеристик конкретного маршруту через динамічний розрахунок порогів безпеки та передбачальну логіку аналізу

майбутніх сегментів подорожі. застосування ефективних методів оптимізації простору пошуку на великих графах.

Висновки до другого розділу

У другому розділі було розроблено комплексний алгоритм побудови оптимального маршруту з урахуванням обмежень на запас палива, який складається з трьох основних етапів: формування та оптимізації транспортного графа, пошуку оптимального шляху та планування точок дозаправки.

Розроблено методи формування та оптимізації транспортного графа на основі даних OpenStreetMap для дорожньої мережі України з використанням dual-graph стратегії, яка створює окремі підграфи для коротких міських та довгих міжміських маршрутів. Застосування просторової індексації через структуру R-tree та методу обмеження області пошуку через VBOX дозволило скоротити час формування підграфа з декількох хвилин до 3-15 секунд при зменшенні кількості вузлів без втрати релевантних заправних станцій.

Описано алгоритми пошуку оптимального шляху з адаптивним вибором стратегії залежно від характеристик маршруту. Для коротких міських маршрутів реалізовано інтеграцію з TomTom Traffic API для врахування реального стану трафіку, а для довгих міжміських маршрутів застосовано алгоритм A* з евристикою Гаверсайна, що забезпечує зменшення часу обчислень у 2-3 рази. Додатково впроваджено векторизований пошук вузлів та multi-waypoint режим для оптимізації маршрутів через множину проміжних точок.

Розроблено багатокритеріальний алгоритм планування точок дозаправки, який є ключовою модифікацією класичної задачі пошуку найкоротшого шляху. Алгоритм використовує адаптивний поріг безпеки залежно від густоти заправних станцій та послідовно перевіряє чотири критерії прийняття рішення: критична ситуація, критично низький рівень

палива, неможливість досягнення наступної станції та передбачальна логіка. Це забезпечує мінімальну кількість зупинок при гарантованій досяжності кінцевого пункту призначення з урахуванням непередбачуваних ситуацій.

РОЗДІЛ 3 РЕАЛІЗАЦІЯ ТА ІНТЕГРАЦІЯ ПРОГРАМНОЇ СИСТЕМИ

3.1 Архітектура та взаємодія компонентів системи

Система планування оптимальних маршрутів з урахуванням обмежень побудована за принципами багатошарової архітектури [33, 34] з чітким розділенням відповідальності між окремими рівнями: обчислювальний рівень для алгоритмічних операцій, рівень бізнес-логіки для управління даними та автентифікації та рівень представлення для взаємодії з користувачем.

Рішення про використання багатошарової архітектури обумовлене кількома міркуваннями. Одна з основних причин – можливість ізолювати обчислювально складні операції побудови маршрутів від бізнес-логіки управління користувачами та збереження даних. Алгоритмічна частина системи вимагає завантаження великих обсягів даних (граф дорожньої мережі, R-tree індекси, множина заправних станцій) в оперативну пам'ять та їх постійного зберігання для швидкого доступу, тоді як бізнес-логіка працює з невеликими об'ємами даних про користувачів та історію їх маршрутів.

Розділення на окремі рівні забезпечує можливість використання найбільш підходящих технологій для кожної задачі, що обумовлювало вибір технологічного стеку, описаного в розділі 1. Така архітектура природно підтримує використання різних мов програмування та фреймворків на кожному рівні через чітко визначені API між ними.

Така архітектура спрощує подальший розвиток системи. Алгоритмічне ядро може бути вдосконалене або замінене без впливу на серверну частину, якщо зберігається контракт API. Аналогічно, мобільний застосунок може бути переписаний на іншу платформу без змін у серверних компонентах.

Структура серверної частини

Серверна частина складається з двох незалежних API сервісів, які взаємодіють між собою через HTTP протокол [35] згідно з принципами REST архітектури [36]. Algorithm API відповідає виключно за обчислювальні операції: формування підграфів за допомогою R-tree індексу, пошук

найкоротшого шляху алгоритмами Дейкстри або A^* , та розрахунок оптимальних точок дозаправки. При запуску цього сервісу відбувається одноразове завантаження всіх необхідних структур даних в оперативну пам'ять, після чого вони залишаються доступними протягом всього життєвого циклу застосунку.

App API виконує роль посередника між мобільним застосунком та алгоритмічним ядром. Його відповідальність включає автентифікацію користувачів через перевірку JWT токенів [37, 38], валідацію параметрів запитів, проксіювання запитів до Algorithm API, збереження історії маршрутів в MongoDB базі даних, та надання точок доступу для отримання списку попередніх маршрутів користувача.

Архітектура App API організована за шаблоном “Controller-Service-Model”. Контролер маршрутів обробляє HTTP запити та містить основну логіку взаємодії з Algorithm API:

```
export const calculateRoute = async (req: Request, res: Response) => {
  const userId = req.userId;

  const pythonResponse = await axios.post(`${PYTHON_API_URL}/route`, {
    start, end, fuel_liters, current_fuel, consumption
  });

  const newRoute = new Route({
    userId,
    route: { coordinates: routeData.route.coordinates },
    fuelStations: routeData.fuel_stations,
    refueling: routeData.refueling
  });
  await newRoute.save();

  res.status(200).json({ success: true, routeId: newRoute._id });
};
```

Окрім контролеру маршрутів, система включає контролер користувачів для управління їх реєстрацією та автентифікацією. MongoDB моделі User та Route визначають схеми збереження даних про користувачів та їх маршрути відповідно. Таким чином, контролери та моделі забезпечують чітке розмежування відповідальностей і створюють основу для надійної та передбачуваної роботи App API.

Взаємодія компонентів при обробці запиту

Життєвий цикл запиту на побудову маршруту включає послідовну взаємодію всіх рівнів системи. Мобільний застосунок формує HTTP POST запит до App API endpoint `/routes/calculate` з параметрами маршруту у форматі JSON:

```
{
  "start": [50.4501, 30.5234],
  "end": [49.9935, 36.2304],
  "fuel_liters": 50,
  "current_fuel": 35,
  "consumption": 7.5
}
```

Запит проходить через послідовність `middleware`: спочатку `authenticate` перевіряє JWT токен та ідентифікує користувача, потім `validate` перевіряє коректність формату вхідних даних. Після успішної валідації контролер маршрутів формує внутрішній запит до Algorithm API ендпоінту `/api/route`, передаючи ті ж параметри.

Algorithm API виконує послідовність операцій з побудови маршруту, описану в розділі 2: R-tree запит для формування підграфа, пошук найкоротшого шляху, ідентифікацію заправних станцій на маршруті та розрахунок оптимальних точок дозаправки. Результат повертається у форматі JSON:

```
{
  "success": true,
  "route": {
    "coordinates": [[50.4501, 30.5234], ...],
    "distance_km": 412.5
  },
  "fuel_stations": [...],
  "refueling": {
    "total_refuels": 2,
    "refuel_details": [...]
  }
}
```

App API отримавши результат зберігає маршрут в MongoDB разом з ідентифікатором користувача та часовою міткою, після чого відправляє відповідь мобільному застосунку. Збереження в базі даних дозволяє

користувачу переглядати історію своїх маршрутів та повторно використовувати попередньо розраховані шляхи.

Мобільний застосунок використовує бібліотеку React Query для управління станом запитів. Хук `useCalculateRoute` інкапсулює логіку відправки запиту та автоматичного оновлення кешу:

```
export const useCalculateRoute = () => {
  return useMutation<CalculateRouteResponse, Error, CalculateRoutePayload>({
    mutationFn: async (payload) => {
      const response = await privateApi.post("/routes/calculate", payload);
      return response.data;
    },
    onSuccess: () => {
      queryClient.invalidateQueries({ queryKey: routeKeys.lists() });
    }
  });
};
```

Отриманий результат передається компоненту візуалізації, який відображає маршрут на карті Mapbox та додає маркери точок дозаправки. Детальна діаграма послідовності взаємодії компонентів представлена нижче (рис. 3.1):

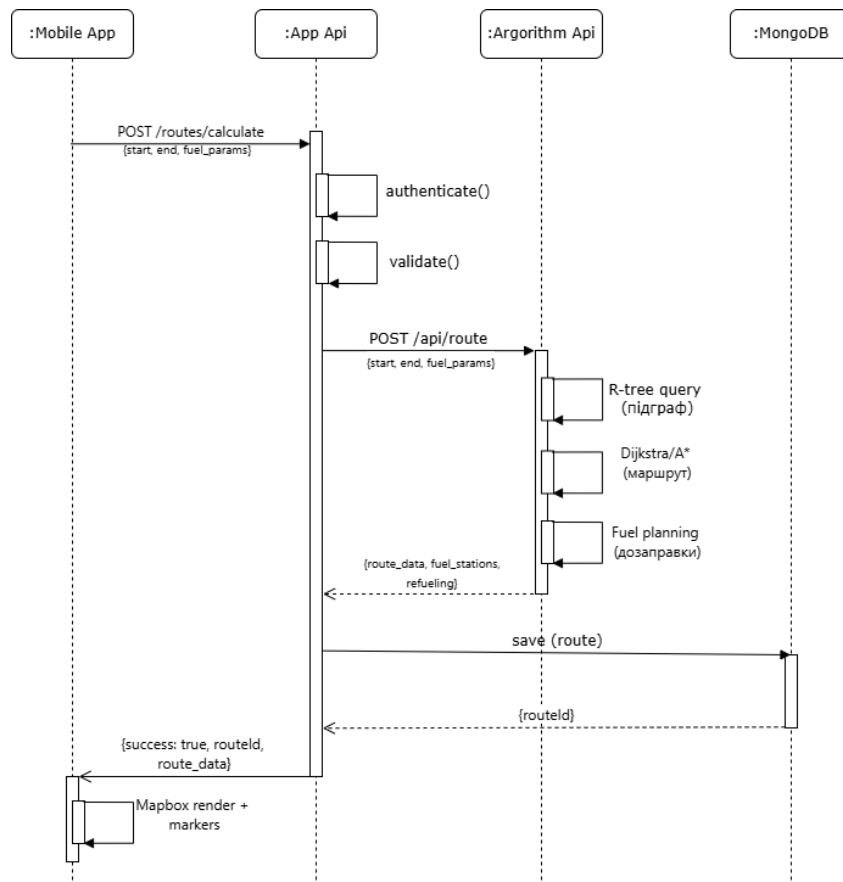


Рис. 3.1 – Діаграма послідовності взаємодії компонентів системи

3.2 Опис роботи програмного забезпечення

Програмне забезпечення реалізує повний цикл роботи з маршрутами: від автентифікації користувача до перегляду аналітики розрахованих маршрутів. Мобільний застосунок на базі React Native [39] забезпечує інтуїтивний інтерфейс для взаємодії з системою, приховуючи складність серверних обчислень за простими екранними формами.

Робота з застосунком починається з екрану входу (рис. 3.2), який пропонує користувачу два варіанти: реєстрацію нового облікового запису або автентифікацію в існуючий. Інтерфейс входу виконаний у мінімалістичному стилі з чітким поділом функціоналу, що дозволяє користувачу швидко зорієнтуватися та обрати потрібну дію.

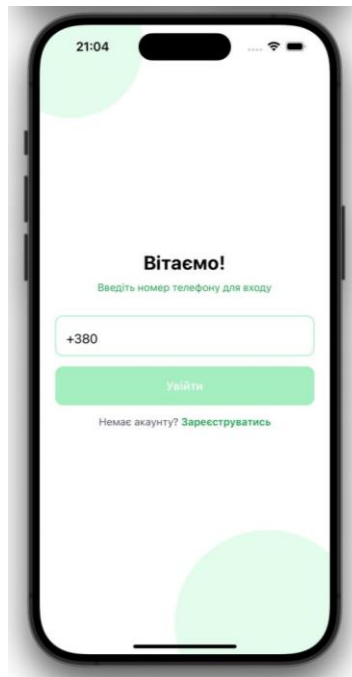


Рис. 3.2 – Екран входу в систему з можливістю реєстрації або автентифікації

Розглянемо сценарій першого використання системи, коли користувач ще не має облікового запису. Для створення акаунту необхідно перейти на екран реєстрації, де на першому етапі система запитує ім'я користувача (рис. 3.3). Це поле є обов'язковим для заповнення та використовується для персоналізації інтерфейсу застосунку. Система валідує введене ім'я на предмет мінімальної довжини та допустимих символів, забезпечуючи коректність збережених даних.

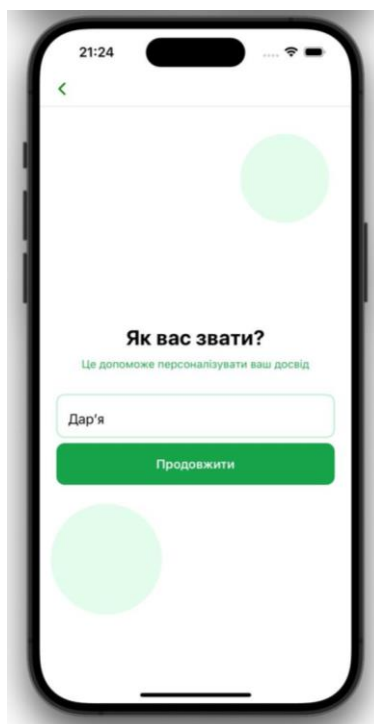


Рис. 3.3 – Екрани реєстрації користувача з введеними іменем

Після введення імені користувач натискає кнопку «Продовжити», що переводить його на наступний крок реєстрації – введення номера телефону (рис. 3.4). Він є унікальним ідентифікатором користувача в системі та використовується для автентифікації через одноразові паролі.

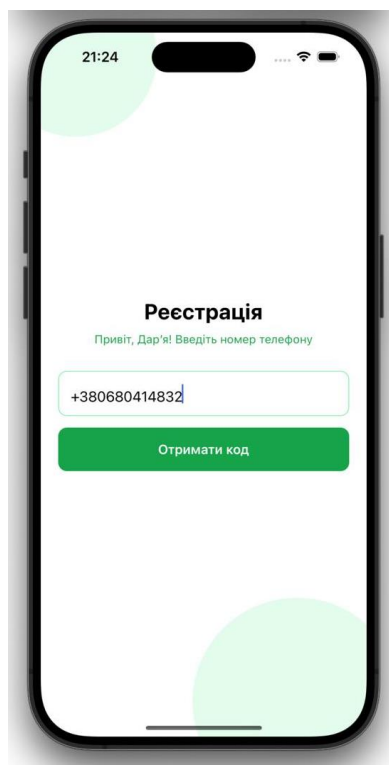


Рис. 3.4 – Екран реєстрації користувача з введеним номером телефону

При натиску на кнопку «Отримати код» система ініціює процес верифікації номера телефону, надсилаючи одноразовий пароль (OTP) через SMS або Viber (рис. 3.5). Використання OTP-автентифікації забезпечує високий рівень захисту облікового запису без необхідності запам'ятовування складних паролів.

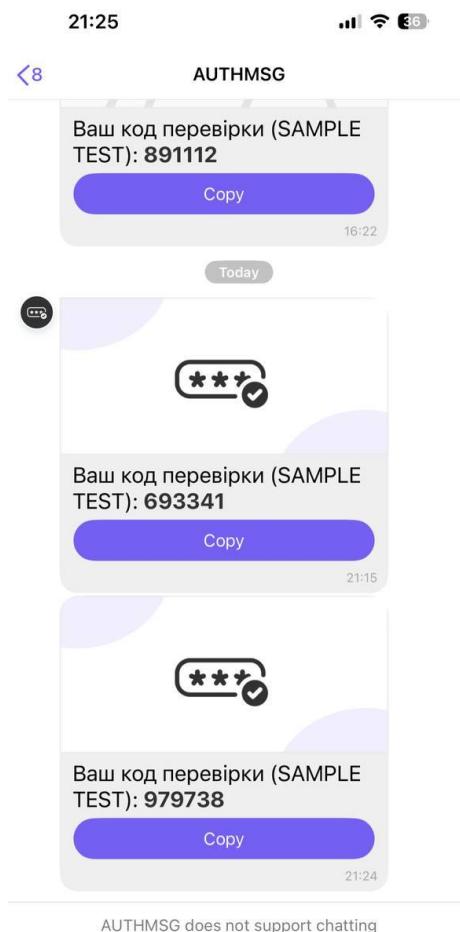


Рис. 3.5 – Отриманий код у Viber

Отриманий код користувач вводить у спеціальне поле верифікації (рис. 3.6). Система автоматично валідує його, перевіряючи відповідність згенерованому на сервері значенню та термін дії. Якщо код введено невірно або минув термін його дії, система пропонує повторно запросити новий код верифікації.

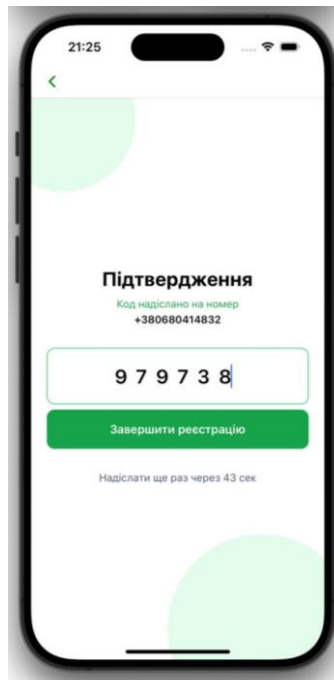


Рис. 3.6 – Екран для вводу коду для завершення аутентифікації

Процес автентифікації для існуючих користувачів відбувається аналогічним чином: користувач вводить свій номер телефону, система перевіряє його наявність у базі даних, після чого надсилає одноразовий код верифікації. Після успішного введення коду користувач отримує доступ до головного екрану застосунку (рис. 3.7).

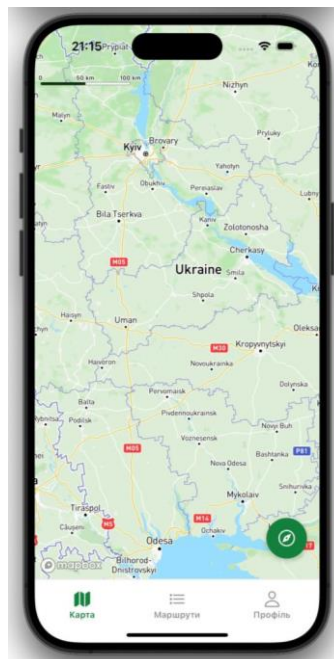


Рис. 3.7 – Головний екран застосунку

Головний екран застосунку є центральним робочим простором для планування маршрутів. Він містить інтерактивну карту [40] з кнопкою в

правому нижньому куті екрану, натиск на яку відкриває панель введення параметрів маршруту (рис. 3.8). Вона містить всі необхідні поля для налаштування розрахунку: стартову і кінцеву точки маршруту, а також параметри автомобіля – об'єм бака, поточний рівень палива та витрату пального на 100 км.

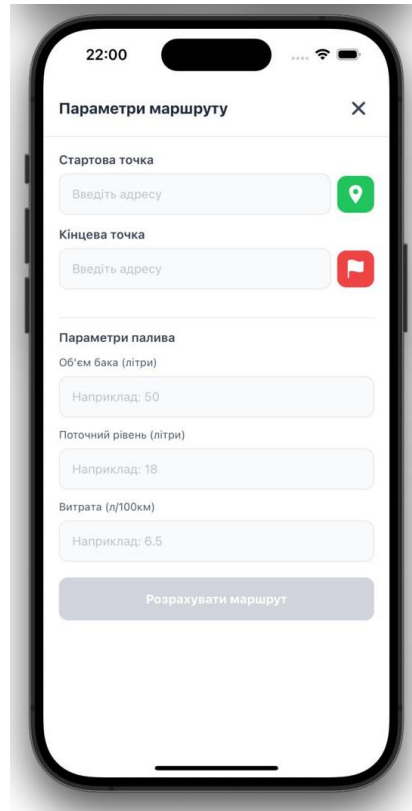


Рис. 3.8 – Панель для введення параметрів для обрахунку маршруту

Система надає користувачу два способи введення стартової точки маршруту. Перший з них – це ручне введення адреси через текстове поле з автодоповненням (рис. 3.9). При введенні тексту система пропонує список релевантних адрес на основі введених символів і користувач обирає потрібну.

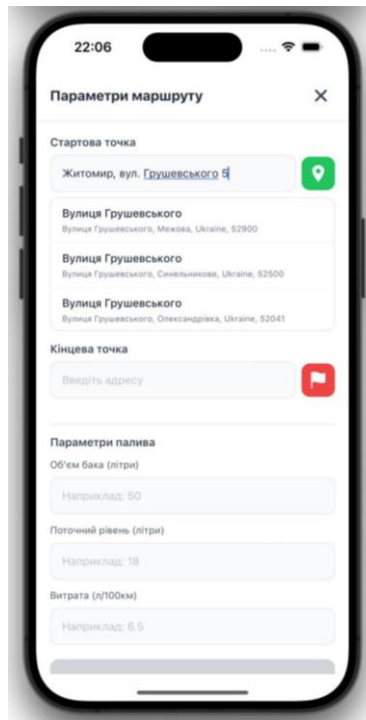


Рис. 3.9 – Введення стартової точки вручну

Другий спосіб введення стартової точки – візуальний вибір на карті (рис. 3.10). При натиску на кнопку з іконкою маркера біля поля вводу відкривається повноекранний режим карти з інтерактивним маркером у центрі. Користувач може переміщувати маркер, обираючи потрібну локацію, після чого підтвердити вибір натиском відповідної кнопки.

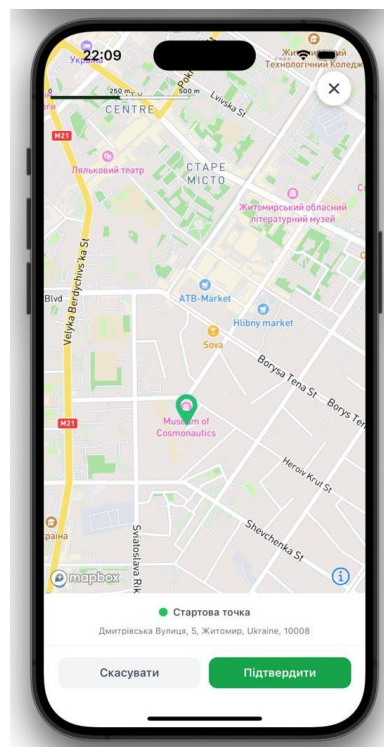


Рис. 3.10 – Обрання стартової точки на карті

Аналогічним чином заповнюються решта параметрів маршруту (рис. 3.11). Поле кінцевої точки підтримує ті самі способи введення – текстовий пошук та вибір на карті. Параметри автомобіля вводяться у відповідні числові поля.

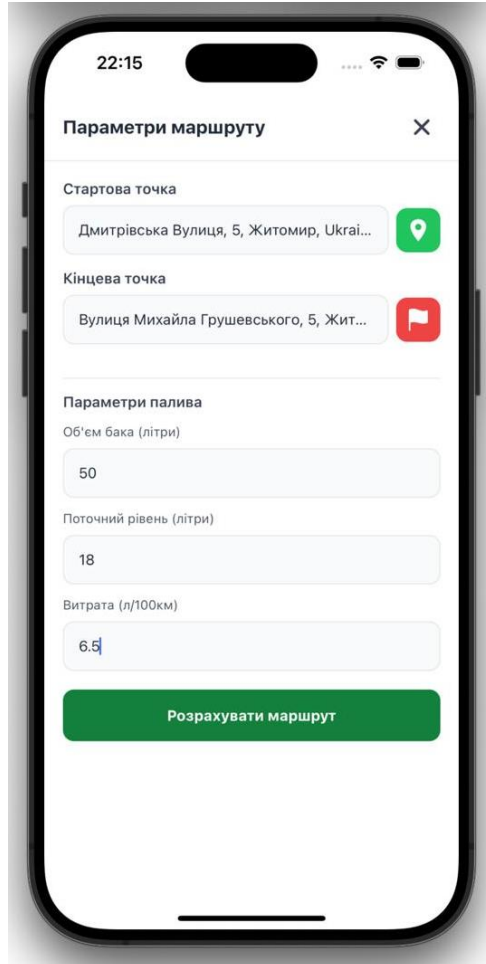


Рис. 3.11 – Заповнення всіх параметрів для обрахунку маршруту

Після заповнення всіх необхідних параметрів користувач ініціює розрахунок маршруту натисканням відповідної кнопки. Система відображає екран процесу обчислень (рис. 3.12) з індикатором прогресу та текстовим повідомленням про те, що відбувається розрахунок маршруту.

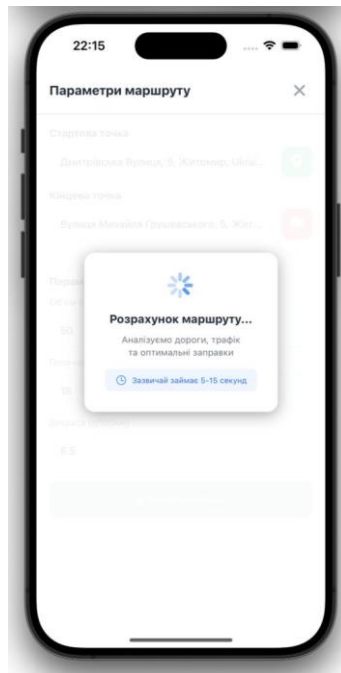


Рис. 3.12 – Процес обрахунку маршруту

Після успішного завершення обчислень система відображає модальне повідомлення з узагальненою інформацією про розрахований маршрут (рис. 3.13). Для міських маршрутів повідомлення містить загальну відстань, кількість дозаправок, кількість заправок на маршруті загалом, час розрахунку та час в дорозі.

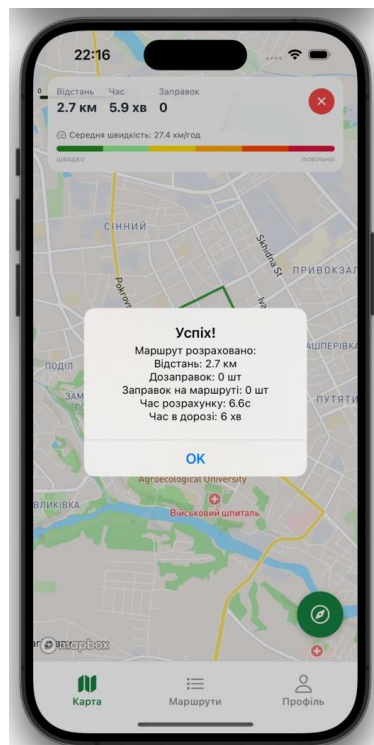


Рис. 3.13 – Повідомлення про успішний розрахунок міського маршруту

Візуалізація міського маршруту на карті (рис. 3.14) використовує кольорове кодування для відображення інтенсивності трафіку на різних сегментах шляху. Зелений колір позначає вільний рух, жовтий – помірний трафік, оранжевий – затори, а червоний – значні затримки або дорожні роботи. Така кольорова диференціація дозволяє користувачу швидко оцінити проблемні ділянки маршруту ще до початку поїздки. У верхній частині екрану відображається панель з ключовими метриками: загальна відстань маршруту, очікуваний час у дорозі, кольорова шкала для розуміння трафіку та середня швидкість руху. Як бачимо, для даного міського маршруту заправки для заправки не були заплановані, оскільки запасу палива було достатньо для досягнення пункту призначення.

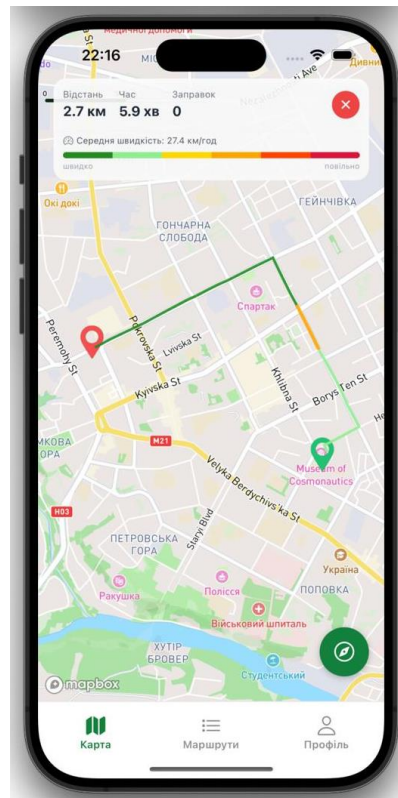


Рис. 3.14 – Результат обрахунку міського маршруту

Для демонстрації функціоналу планування дозаправок розглянемо приклад міжміського маршруту. Користувач заповнює параметри аналогічно попередньому випадку (рис. 3.15), але обирає пункти відправлення та призначення, що знаходяться на значній відстані один від одного, яка перевищує можливий пробіг на одному баку палива.

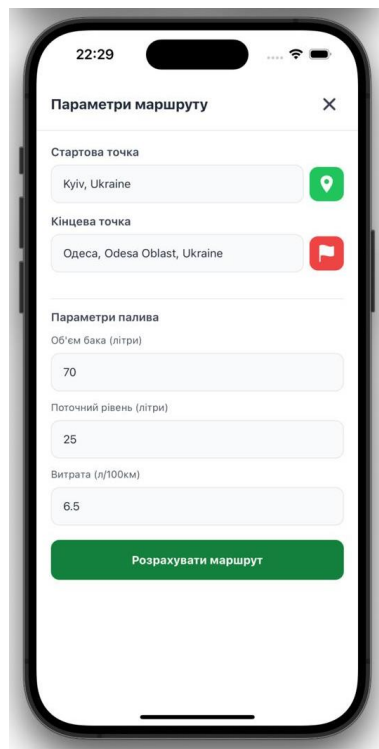


Рис. 3.15 – Заповнення параметрів для міжміського маршруту

Після завершення розрахунку система відображає повідомлення з результатами (рис. 3.16), яке подібне по метрикам міському маршруту, проте не містить часу у дорозі. Тут ми вже бачимо, що є певна кількість заправок та розміщена 1 підзаправка для оптимального маршруту.

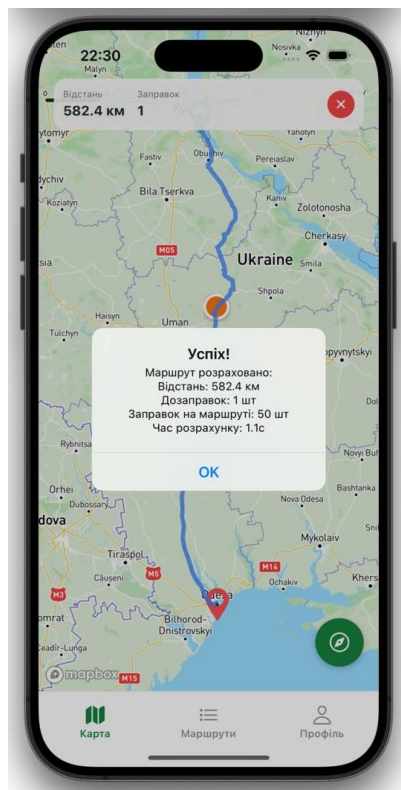


Рис. 3.16 – Повідомлення про успішний розрахунок міжміського маршруту

Візуалізація міжміського маршруту на карті (рис. 3.17) відрізняється від міського тим, що не містить кольорового кодування трафіку, оскільки для міжміських маршрутів дані про трафік не інтегруються. Натомість маршрут відображається суцільною лінією, на якій позначені точки запланованих дозаправок спеціальними маркерами. У верхній панелі може побачити загальну відстань та кількість підзаправок.

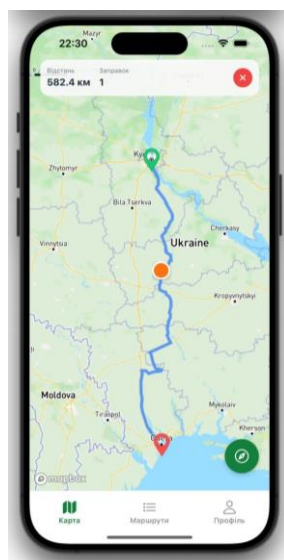


Рис. 3.17 – Отримані результати міжміського маршруту

Для перегляду історії попередньо розрахованих маршрутів та їх детальної аналітики передбачено окремий екран «Маршрути» (рис. 3.18), доступ до якого здійснюється через нижню навігаційну панель застосунку. Даний екран відображає список усіх маршрутів користувача у зворотному хронологічному порядку (найновіші зверху). Кожен елемент списку представлений у вигляді картки, що містить базову інформацію: стартову та кінцеву точки, загальну відстань та дату створення маршруту. Картки мають згорнутий вигляд за замовчуванням, що дозволяє переглянути більшу кількість маршрутів на одному екрані. Для доступу до детальної аналітики користувач може розгорнути потрібну картку натисканням на неї або на спеціальну іконку розкриття.

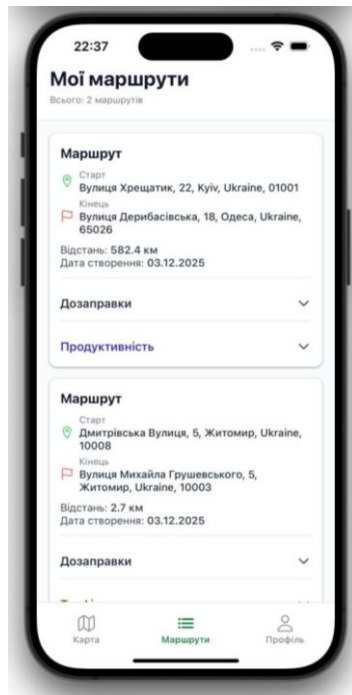


Рис. 3.18 – Екран для перегляду історії маршрутів

При розгортанні картки міжміського маршруту (рис. 3.19) відображається детальна аналітика, структурована у три основні блоки: дозаправки, продуктивність та загальна інформація. Блок дозаправок містить загальну інформація по заправкам та детальну про кожну заплановану зупинку: її порядковий номер, відстань від початку маршруту, на якій вона розташована, залишок пального до дозаправки та причину зупинки.

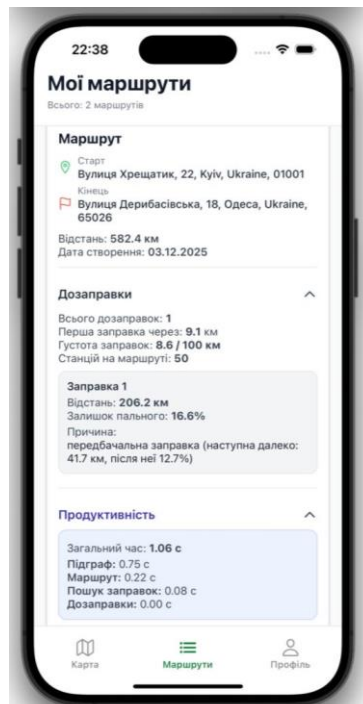


Рис. 3.19 – Аналітика по міжміському маршруту

Для міських маршрутів аналітична картка (рис. 3.20) має дещо іншу структуру. Якщо дозаправки не були заплановані, замість детального блоку дозаправок відображається пояснення у форматі «загальна відстань маршруту < максимально можлива відстань на поточному запасі палива», що підтверджує достатність наявного палива. Натомість додається блок аналітики трафіку, який містить три ключові метрики: час у дорозі без урахування трафіку (базовий розрахунок), час з урахуванням поточного трафіку (реальний прогноз) та середню швидкість на маршруті.

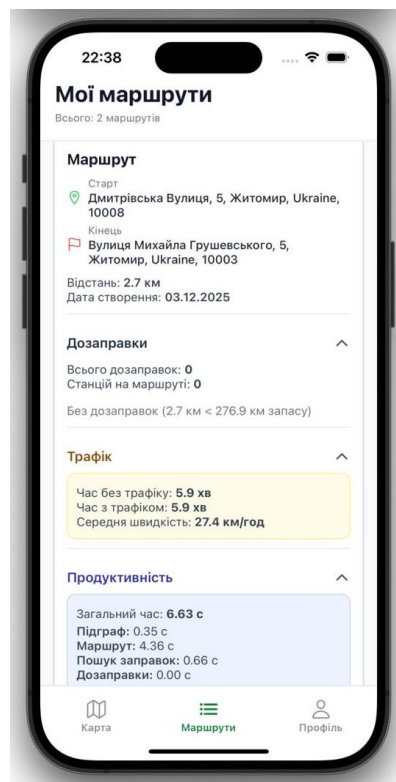


Рис. 3.20 – Аналітика по міському маршрутах

Блок продуктивності, присутній в аналітиці обох типів маршрутів, надає технічну інформацію про швидкість обчислень. Він показує загальний час розрахунку маршруту та деталізацію за окремими етапами: час виділення підграфа, час власне пошуку шляху, час пошуку заправних станцій у радіусі від маршруту та час оптимізації дозаправок.

Окремо в застосунку передбачено екран профілю користувача (рис. 3.21), який відображає персональну інформацію: ім'я користувача та верифікований номер телефону, прив'язаний до облікового запису. Даний

екран є простим інформаційним розділом, що дозволяє користувачу переглянути свої облікові дані.

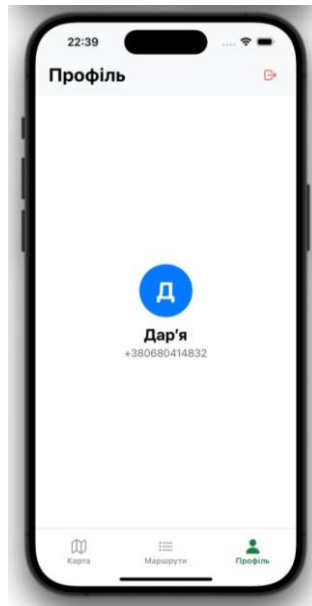


Рис. 3.21 – Екран профілю користувача

Для завершення сесії роботи з застосунком передбачено функцію виходу з облікового запису (рис. 3.22), доступ до якої здійснюється через кнопку у правому верхньому куті екрану профілю. При натиску на цю кнопку система видає підтверджуюче повідомлення, що запобігає випадковому виходу з акаунту. Після підтвердження виконується деавторизація і сесія користувача закінчується.

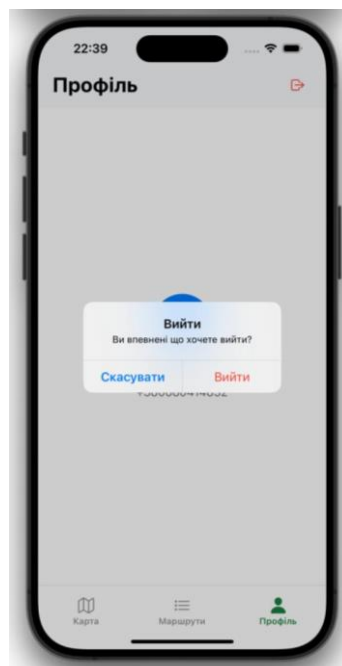


Рис. 3.22 – Опція виходу з акаунту

Висновки до третього розділу

У третьому розділі було описано архітектуру системи, побудовану за принципами багат шарової організації з розділенням на обчислювальний рівень (Algorithm API), рівень бізнес-логіки (App API) та рівень представлення (мобільний застосунок).

Представлено структуру серверної частини з двома незалежними API сервісами, які взаємодіють через HTTP протокол. Детально описано життєвий цикл запиту на побудову маршруту, який включає послідовну взаємодію всіх компонентів системи: від прийому параметрів через App API, передачі їх до Algorithm API для виконання обчислень, до збереження результату в базі даних MongoDB та відправки відповіді клієнту.

Наведено опис роботи програмного забезпечення з точки зору користувача, включаючи процес входу в систему, введення параметрів маршруту, візуалізацію результатів на інтерактивній карті та перегляд детальної аналітичної інформації про розраховані маршрути в історії.

ВИСНОВКИ

Під час виконання кваліфікаційної магістерської роботи було детально досліджено методи та алгоритми планування оптимальних автомобільних маршрутів з урахуванням обмежень на запас палива та інтеграцією даних про трафік. Результатом роботи стала функціональна система, що складається з серверної частини для обчислень на графі дорожньої мережі України та мобільного застосунку для взаємодії з користувачем. Основними завданнями було проведення аналізу існуючих систем планування маршрутів, розробка методів оптимізації графа дорожньої мережі, реалізація адаптивної стратегії вибору алгоритмів пошуку шляху, створення багатокритеріального алгоритму планування точок дозаправки та проектування програмної системи.

У першому розділі було проаналізовано предметну область планування автомобільних маршрутів, проведено огляд існуючих систем та виявлено, що жодна з них не надає автоматичного планування зупинок для заправки з урахуванням параметрів конкретного автомобіля. Проаналізовано базові алгоритми пошуку найкоротшого шляху на графах, зокрема алгоритми Дейкстри та A^* , та визначено критерії їх вибору залежно від характеристик маршруту. Обґрунтовано вибір технологічного стеку для архітектури системи, що включає Python з Flask для алгоритмічної частини, OpenStreetMap як джерело даних про дорожню мережу, TomTom Traffic API для інтеграції трафіку, TypeScript з MongoDB для серверної частини та React Native з Mapbox для мобільного застосунку.

У другому розділі було розроблено методи оптимізації транспортного графа з використанням просторової індексації через структуру R-tree та dual-graph стратегії, що дозволило значно скоротити час формування підграфа. Реалізовано адаптивну стратегію вибору алгоритмів пошуку шляху: алгоритм Дейкстри з інтеграцією TomTom Traffic API для коротких міських маршрутів та алгоритм A^* з евристикою Гаверсайна для довгих міжміських маршрутів, що забезпечило зменшення часу обчислень у 2-3 рази порівняно з базовим алгоритмом Дейкстри. Ключовим досягненням стала розробка

багатокритеріального алгоритму планування точок дозаправки з чотирма критеріями прийняття рішень та адаптивним порогом безпеки залежно від густоти заправних станцій, що забезпечує мінімальну кількість зупинок при гарантованій досяжності пункту призначення.

У третьому розділі було здійснено проектування та реалізацію програмної системи з архітектурою, що включає обчислювальний рівень для виконання алгоритмічних операцій, рівень бізнес-логіки для управління даними та автентифікації, та рівень представлення у вигляді мобільного застосунку. Детально представлено життєвий цикл запиту на побудову маршруту від прийому параметрів до виконання обчислень та збереження результату в базі даних. Наведено опис роботи програмного забезпечення з точки зору користувача, включаючи процес автентифікації, введення параметрів транспортного засобу, візуалізацію маршруту на інтерактивній карті та перегляд детальної аналітичної інформації про розраховані маршрути в історії.

Розроблена система демонструє ефективність запропонованих алгоритмічних рішень та має потенціал для подальшого розвитку. Розроблені алгоритми та методи оптимізації мають універсальний характер і можуть бути адаптовані для різних типів транспортних засобів у майбутньому. Система надає користувачам інструмент для ефективного планування подорожей з автоматичним визначенням оптимальних точок заправки та аналізом витрат, що дозволяє приймати обґрунтовані рішення при виборі маршруту.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Bilkevych D., Vakaliuk T. Artificial intelligence in gps navigation system. Сучасні інформаційні системи та технології: матеріали VII Всеукр. наук.-практ. інтернет-конф. за тематикою «Сучасні комп'ютерні системи та мережі в управлінні» (29 листопада 2024 р., м. Херсон, м. Хмельницький) / за ред. А. А. Григорової. – Херсон: Книжкове видавництво ФОП Вишемирський В. С., 2024. С. 153-154.
2. D.V. Bilkevych, T.A. Vakaliuk. The Use of Dijkstra's Algorithm in Navigation Applications. XVIII Міжнародна науково-практична конференція магістрантів та аспірантів «Теоретичні та практичні дослідження молодих вчених» (19–22 листопада 2024 року): матеріали конференції / за ред. проф. Є.І. Сокола. – Харків : НТУ «ХПІ», 2024. – С. 58-59.
3. Rachmawati D., Gustin L. Analysis of Dijkstra's Algorithm and A* Algorithm in Shortest Path Problem. Journal of Physics: Conference Series, Vol. 1566, No. 1, 2020, <https://doi.org/10.1088/1742-6596/1566/1/012061>.
4. Ferguson D., Stentz A. A Systematic Literature Review of A* Pathfinding. Procedia Computer Science, Vol. 179, 2021, pp. 507-514, <https://doi.org/10.1016/j.procs.2021.01.035>.
5. Wu N., Zhao X.W., Wang J., Pan D. Learning Effective Road Network Representation with Hierarchical Graph Neural Networks. Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, 2020, pp. 6-14, <https://doi.org/10.1145/3394486.3403043>.
6. HERE Technologies. HERE WeGo: Maps & Navigation [Електронний ресурс] – Режим доступу до ресурсу: <https://www.here.com/products/wego>.
7. Sasson, R. (2023). ETA prediction at Waze using Deep Learning Models. Data Science Economics. Режим доступу до ресурсу: <https://www.datascienceeconomics.com/post/eta-prediction-at-waze-using-deep-learning-models>.

8. HERE Technologies unveils HERE WeGo Pro: a truck navigation app for safer, more efficient fleet operations. Press Release [Электронный ресурс] – Режим доступа до ресурсу: <https://www.globenewswire.com/news-release/2025/02/11/3024224/0/en/HERE-Technologies-unveils-HERE-WeGo-Pro-a-truck-navigation-app-for-safer-more-efficient-fleet-operations.html>.

9. Klinkhardt C., Woerle T., Briem L., Heilig M., Kagerbauer M., Vortisch P. Using OpenStreetMap as a Data Source for Attractiveness in Travel Demand Models. Transportation Research Record, Vol. 2675, No. 5, 2021, pp. 294-303, <https://doi.org/10.1177/0361198121997415>.

10. Haklay M., Antoniou V., Basiouka S., Soden R., Mooney P. OSM Science – The Academic Study of the OpenStreetMap Project, Data, Contributors, Community, and Applications. ISPRS International Journal of Geo-Information, Vol. 11, No. 4, 2022, <https://doi.org/10.3390/ijgi11040230>.

11. Boeing G. Modeling and Analyzing Urban Networks and Amenities with OSMnx. Geographical Analysis, 2025, <https://doi.org/10.1111/gean.70009>.

12. Kwak H., Choi Y., Choi J. SGIR-Tree: Integrating R-Tree Spatial Indexing as Subgraphs in Graph Database Management Systems. ISPRS International Journal of Geo-Information, Vol. 13, No. 10, 2024, <https://doi.org/10.3390/ijgi13100346>.

13. Mohebi A., Sattari M., Haghi Kashani M., Mahdipour E. A Comparative Study of MongoDB and Document-Based MySQL for Big Data Application Data Management. Future Internet, Vol. 14, No. 5, 2022, <https://doi.org/10.3390/fi14050134>.

14. Abdellaoui S., Abbaoui W., Meziane L., Bhiri B.E., Ziti S. SQL and NoSQL Databases: A Comparative Study with Perspectives on IA-Based Migration Approach. Engineering Proceedings, Vol. 112, No. 1, 2025, <https://doi.org/10.3390/engproc2025112072>.

15. Cormen T.H., Leiserson C.E., Rivest R.L., Stein C. Introduction to Algorithms. 3rd ed. Cambridge: MIT Press, 2009, pp. 643–671.

16. Wang Y. Review on greedy algorithm. Theoretical and Natural Science, 2023, pp. 233-239, <https://doi.org/10.54254/2753-8818/14/20241041>.
17. Yan Y. Research on the A Star Algorithm for Finding Shortest Path. Highlights in Science, Engineering and Technology, 2023, pp. 154-161, <https://doi.org/10.54097/hset.v46i.7697>.
18. Thompson J. Discovering Heuristics. Impact, Vol. 2020, No. 1, 2020, pp. 25–29, <https://doi.org/10.1080/2058802X.2020.1739866>.
19. Lotfian M., Ingensand J., Brovelli M.A. Exploring five indicators for the quality of OpenStreetMap road networks: a case study of Québec, Canada. Geomatica, Vol. 76, No. 1, 2022, pp. 1-19, <https://doi.org/10.1139/geomat-2021-0012>.
20. Hosseini R., Tong D., Lim S., Sohn G., Gidófalvi G. A framework for performance analysis of OpenStreetMap data in navigation applications: the case of a well-developed road network in Australia. International Journal of Digital Earth, Vol. 18, 2025, pp. 233-250, <https://doi.org/10.1080/19475683.2025.2468184>.
21. Fraire Huacuja H.J., Carpio Valadez J.M., Cruz Chávez M.A., Gómez Fuentes M.C., Reyes Pimentel T.A. GIS Spatial Optimization for Corridor Alignment Using Simulated Annealing. Applied Sciences, Vol. 10, No. 18, 2020, <https://doi.org/10.3390/app10186190>.
22. Huynh H.N., Shin Y., Son B. A road network simplification algorithm that preserves topological properties. Applied Network Science, Vol. 7, No. 93, 2022, <https://doi.org/10.1007/s41109-022-00521-8>.
23. Zhang S., Wang L., Zhang J., Yu P.S. What If London Bridge Is Closed? Feature-Aware Subgraph Augmentation for Modeling Road Network Structure Changes. IEEE Transactions on Knowledge and Data Engineering, 2024, <https://doi.org/10.1109/TKDE.2024.3416774>.
24. Maria E., Wulandari S., Wibowo A. Measure distance locating nearest public facilities using Haversine and Euclidean Methods. Journal of Physics: Conference Series, Vol. 1450, No. 1, 2020, <https://doi.org/10.1088/1742-6596/1450/1/012080>.

25. Zhang X., Li Q., Wang Y. Data Science in Transportation Networks with Graph Neural Networks: A Review and Outlook. *Data Science for Transportation*, Vol. 7, No. 14, 2025, <https://doi.org/10.1007/s42421-025-00124-6>.
26. Pham H.D., Narasimhamurthy S.M., Mehran B., Manley E., Ashraf A. Reinforcement learning based estimation of shortest paths in dynamically changing transportation networks. *Frontiers in Future Transportation*, Vol. 6, 2025, <https://doi.org/10.3389/ffutr.2025.1524232>.
27. TomTom International B.V. Traffic API Documentation: Traffic Flow and Routing Services. TomTom Developer Portal, 2025 [Электронный ресурс] – Режим доступа до ресурсу: <https://developer.tomtom.com/traffic-api/documentation/tomtom-maps/product-information/introduction>.
28. Yan Z., Zhang J., Tang J. Big spatial vector data management: a review. *International Journal of Digital Earth*, Vol. 12, No. 1, 2019, pp. 18-38, <https://doi.org/10.1080/20964471.2018.1432115>.
29. Lei Z., Lei T.L. Solving Spatial Optimization Problems via Lagrangian Relaxation and Automatic Gradient Computation. *ISPRS International Journal of Geo-Information*, Vol. 14, No. 1, 2025, <https://doi.org/10.3390/ijgi14010015>.
30. Xu Z., Li Y., Wang H. A Review of Last-Mile Delivery Optimization: Strategies, Technologies, Drone Integration, and Future Trends. *Vehicles*, Vol. 7, No. 1, 2025, <https://doi.org/10.3390/vehicles7010019>.
31. Lin J., Lee W., Vairaktarakis G.L. A linear-time algorithm for finding optimal vehicle refueling policies. *Operations Research Letters*, Vol. 35, No. 3, 2007, pp. 290-296, <https://doi.org/10.1016/j.orl.2006.03.005>.
32. Kobvölgyi A.N., Gokalp G.G., Chioua M., Hellendoorn H., Schulze E. An efficient heuristic for the fixed-route vehicle-refueling problem. *Transportation Research Part E: Logistics and Transportation Review*, Vol. 169, 2023, <https://doi.org/10.1016/j.tre.2022.102980>.
33. Wang H. Research on the Application of Layered Architecture in Computer Software Development. *Journal of Computing and Electronic Information*

Management, Vol. 9, No. 6, 2023, pp. 88-92,
<https://doi.org/10.26480/jceim.06.2023.88.92>

34. Marmsoler D., Malkis A., Eckhardt J. A Model of Layered Architectures. arXiv preprint, 2024, <https://doi.org/10.48550/arXiv.1503.04916>

35. Fielding R., Nottingham M., Reschke J. HTTP Semantics. RFC 9110 [Электронный ресурс] – Режим доступа до ресурсу: <https://www.rfc-editor.org/info/rfc9110>

36. Microsoft Azure Architecture Center. Web API Design Best Practices [Электронный ресурс] – Режим доступа до ресурсу: <https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design>

37. Jones M., Bradley J., Sakimura N. JSON Web Token (JWT). RFC 7519 [Электронный ресурс] – Режим доступа до ресурсу: <https://tools.ietf.org/html/rfc7519>

38. Bertocci V. JSON Web Token (JWT) Profile for OAuth 2.0 Access Tokens. RFC 9068 [Электронный ресурс] – Режим доступа до ресурсу: <https://www.rfc-editor.org/rfc/rfc9068.html>

39. Kansal A., Sarangal H., Kumar A. Cross-Platform Empirical Analysis of Mobile Application Development frameworks: Kotlin, React Native and Flutter. Proceedings of the 4th International Conference on Information Management & Machine Intelligence (ICIMMI 2022), 2022, <https://doi.org/10.1145/3590837.3590897>

40. Li C.-Y., Liang C.-H., Chou C.-C., Wu F.-G. Spatial Knowledge Acquisition with Mobile Maps: Effects of Map Size on Users' Wayfinding Performance with Interactive Interfaces. ISPRS International Journal of Geo-Information, Vol. 9, No. 11, 2020, <https://doi.org/10.3390/ijgi9110614>

ДОДАТКИ

Лістинг алгоритму для пошуку підзаправок на маршруті

```

"""
Optimal refueling points calculation module
"""

def calculate_refuels_optimized(base_route, G, initial_fuel_m, tank_capacity_m,
                               fuel_stations, node_to_coord,
                               total_distance_override=None):
    """
    Optimized refueling calculation with predictive logic.

    4 refueling criteria (priority order):
    1. Cannot reach end or next station (CRITICAL)
    2. Fuel critically low AND cannot reach next station
    3. Cannot reach next station (even if fuel > threshold)
    4. Predictive: remaining <15% AND after next <10%, OR next >40km AND after
    <15%

    Args:
        base_route: List of route nodes
        G: NetworkX graph
        initial_fuel_m: Initial fuel range (meters)
        tank_capacity_m: Tank capacity range (meters)
        fuel_stations: Set of fuel station nodes
        node_to_coord: Dict mapping node -> coordinates
        total_distance_override: Real route distance (for TomTom, optional)

    Returns:
        tuple: (list of refuel nodes, total distance, refuel_stats) or (None,
        distance, None)
        where refuel_stats is a dict with detailed refueling information
    """

    # Initialize statistics collection
    refuel_details = []

    # Calculate cumulative distances
    cumulative_distances = [0]
    for i in range(len(base_route) - 1):
        curr = base_route[i]
        next_node = base_route[i + 1]

        edge_data = G.get_edge_data(curr, next_node)
        if edge_data is None:
            edge_length = 0
        elif isinstance(edge_data, dict):
            if 0 in edge_data:
                edge_length = min(e.get('length', 0) for e in edge_data.values())

```

```

        else:
            edge_length = edge_data.get('length', 0)
        else:
            edge_length = 0

        cumulative_distances.append(cumulative_distances[-1] + edge_length)

    total_distance = cumulative_distances[-1]

    if total_distance_override is not None:
        print(f"    OSM відстань: {total_distance/1000:.1f} км")
        print(f"    ✓ Використання ТомТом відстані: {total_distance_override/1000:.1f} км")
        total_distance = total_distance_override

    if total_distance <= initial_fuel_m:
        message = f"Без дозаправок ({total_distance/1000:.1f} км < {initial_fuel_m/1000:.1f} км запасу)"
        print(f"    ✓ {message}")

    refuel_stats = {
        "total_refuels": 0,
        "refuel_details": [],
        "stations_on_route": 0,
        "safety_threshold_percent": 0,
        "first_station_reachable": True,
        "message": message
    }

    return [], total_distance, refuel_stats

# Find fuel stations on route
fuel_on_route = []
seen_nodes = set()

for i, node in enumerate(base_route):
    if node in fuel_stations and node not in seen_nodes:
        fuel_on_route.append((i, node, cumulative_distances[i],
node_to_coord[node]))
        seen_nodes.add(node)

if not fuel_on_route:
    print("    X Немає заправок на маршруті!")
    return None, total_distance, None

print(f"    {len(fuel_on_route)} заправок на маршруті (без дублікатів)")

# Check if first station reachable
first_station_dist = fuel_on_route[0][2]
if first_station_dist > initial_fuel_m:
    print(f"\n    X КРИТИЧНА ПОМИЛКА: Не можемо дістатися першої заправки!")

```

```

        print(f"        Перша заправка на: {first_station_dist/1000:.1f} км")
        print(f"        Запас палива: {initial_fuel_m/1000:.1f} км")
        print(f"        Не вистачає: {(first_station_dist -
initial_fuel_m)/1000:.1f} км")
        print(f"\n        РІШЕННЯ:")
        print(f"        1. Збільшити current_fuel (мінімум
{(first_station_dist/1000) * (8.0/100):.2f}L)")
        print(f"        2. Зменшити consumption")
        print(f"        3. Обрати інший маршрут з ближчими заправками")
        return None, total_distance, None

    print(f"    ✓ Перша заправка досяжна: {first_station_dist/1000:.1f} км <
{initial_fuel_m/1000:.1f} км запасу")

    # Calculate safety threshold
    stations_density = len(fuel_on_route) / (total_distance / 100000)
    safety_threshold = 0.25 if stations_density < 2 else 0.20
    safety_margin_m = tank_capacity_m * safety_threshold

    print(f"    Густота заправок: {stations_density:.1f}/100км")
    print(f"    Попір безпеки: {safety_threshold*100:.0f}%
({safety_margin_m/1000:.1f} км)\n")

    # Main refueling algorithm
    refuel_points = []
    current_fuel = initial_fuel_m
    current_position = 0

    print(f"АНАЛІЗ ПЕРШИХ 10 ЗАПРАВOK:\n")

    for idx, (route_idx, node, dist_from_start, coords) in
enumerate(fuel_on_route):
        distance_to_station = dist_from_start - current_position
        fuel_after_reaching = current_fuel - distance_to_station

        if idx < 10:
            fuel_percent = fuel_after_reaching / tank_capacity_m * 100 if
fuel_after_reaching >= 0 else -1
            print(f"    {idx+1:2d}. Заправка на {dist_from_start/1000:6.1f} км →
залишок {fuel_percent:5.1f}%", end="")

            # Can't reach this station - emergency refuel at previous
            if fuel_after_reaching < 0:
                if idx < 10:
                    print(f"    X НЕ МОЖЕМО ДІСТАТИСЯ!")

                    print(f"    X КРИТИЧНА ПОМИЛКА: Не можемо дістатися заправки на
{dist_from_start/1000:.1f} км!")
                    print(f"        Потрібно: {distance_to_station/1000:.1f} км, є:
{current_fuel/1000:.1f} км")

```

```

    for prev_idx in range(idx - 1, -1, -1):
        _, prev_node, prev_dist, prev_coords = fuel_on_route[prev_idx]
        dist_to_prev = prev_dist - current_position

        if dist_to_prev <= current_fuel and prev_node not in
refuel_points:
            print(f"    Аварійна дозаправка на {prev_dist/1000:.1f} км")
            refuel_points.append(prev_node)
            current_fuel = tank_capacity_m
            current_position = prev_dist
            break
        else:
            print(f"    X Маршрут неможливий!")
            return None, total_distance, None

    continue

# Analyze situation
distance_to_end = total_distance - dist_from_start
can_reach_end = distance_to_end <= fuel_after_reaching

can_reach_next = False
next_station_dist = None

if idx + 1 < len(fuel_on_route):
    next_station_dist = fuel_on_route[idx + 1][2]
    distance_to_next = next_station_dist - dist_from_start
    can_reach_next = distance_to_next <= fuel_after_reaching

is_fuel_critically_low = fuel_after_reaching < safety_margin_m

# Decision logic (4 criteria)
should_refuel = False
reason = ""

if not can_reach_end and not can_reach_next:
    should_refuel = True
    reason = "не можемо дістатися кінця і наступної заправки"
    if idx < 10:
        print(f" → ✓ КРИТЕРІЙ 1!")

elif is_fuel_critically_low and not can_reach_next:
    should_refuel = True
    reason = f"критично мало палива (< {safety_threshold*100:.0f}%) і не
досягнемо наступної заправки"
    if idx < 10:
        print(f" → ✓ КРИТЕРІЙ 2!")

elif not can_reach_next and idx + 1 < len(fuel_on_route):
    should_refuel = True

```

```

        reason = f"не досягнено наступної заправки (на
{next_station_dist/1000:.1f} км)"
        if idx < 10:
            print(f" → ✓ КРИТЕРІЙ 3!")

    elif can_reach_next and idx + 1 < len(fuel_on_route):
        next_station_dist = fuel_on_route[idx + 1][2]
        distance_to_next = next_station_dist - dist_from_start
        fuel_after_next = fuel_after_reaching - distance_to_next

        is_very_low_now = fuel_after_reaching < tank_capacity_m * 0.15
        will_be_critical_after_next = fuel_after_next < tank_capacity_m *
0.10

        is_long_distance_ahead = distance_to_next > 40000
        will_be_low_after_next = fuel_after_next < tank_capacity_m * 0.15

        if (is_very_low_now and will_be_critical_after_next) or \
            (is_long_distance_ahead and will_be_low_after_next):
            should_refuel = True
            fuel_percent_after_next = fuel_after_next / tank_capacity_m * 100

            if is_long_distance_ahead and will_be_low_after_next and not
is_very_low_now:
                reason = f"передбачальна заправка (наступна далеко:
{distance_to_next/1000:.1f} км, після неї {fuel_percent_after_next:.1f}%)"
            else:
                reason = f"передбачальна заправка (без неї після наступної
було б {fuel_percent_after_next:.1f}%)"

            if idx < 10:
                print(f" → ✓ КРИТЕРІЙ 4! (після наступної
{fuel_percent_after_next:.1f}%)")
            else:
                if idx < 10:
                    if not is_very_low_now:
                        print(f" → пропуск (залишок
{fuel_after_reaching/tank_capacity_m*100:.1f}% > 15%)")
                    elif not will_be_critical_after_next:
                        fuel_percent_after_next = fuel_after_next /
tank_capacity_m * 100
                        print(f" → пропуск (після наступної
{fuel_percent_after_next:.1f}% > 10%)")
                    else:
                        if idx < 10:
                            print(f" → пропуск (інші причини)")

        # Execute refueling
        if should_refuel and node not in refuel_points:
            fuel_percent = fuel_after_reaching / tank_capacity_m * 100

```

```

        if idx == 10:
            print(f"\n    ... (решта заправок приховано)\n")

            print(f"    Дозаправка на {dist_from_start/1000:.1f} км (залишок
{fuel_percent:.1f}%)")
            print(f"        Причина: {reason}")

            # Store refuel details for API response
            refuel_details.append({
                "distance_km": round(dist_from_start / 1000, 1),
                "fuel_remaining_percent": round(fuel_percent, 1),
                "reason": reason,
                "coordinates": coords
            })

            refuel_points.append(node)
            current_fuel = tank_capacity_m
            current_position = dist_from_start
            print()

        elif not should_refuel:
            current_fuel = fuel_after_reaching
            current_position = dist_from_start

    # Final check
    final_distance = total_distance - current_position
    final_fuel_percent = None

    if final_distance > current_fuel:
        print(f"        ПОПЕРЕДЖЕННЯ: Може не вистачити палива до кінця!")
        print(f"        Потрібно: {final_distance/1000:.1f} км")
        print(f"        Є: {current_fuel/1000:.1f} км")
        print(f"        Дефіцит: {(final_distance - current_fuel)/1000:.1f} км")
    else:
        final_fuel_percent = (current_fuel - final_distance) / tank_capacity_m *
100

        print(f"    ✓ Фінальна перевірка: залишиться {final_fuel_percent:.1f}%
палива")

    # Build statistics object
    refuel_stats = {
        "total_refuels": len(refuel_points),
        "refuel_details": refuel_details,
        "stations_on_route": len(fuel_on_route),
        "stations_density_per_100km": round(stations_density, 1),
        "safety_threshold_percent": int(safety_threshold * 100),
        "safety_margin_km": round(safety_margin_m / 1000, 1),
        "first_station_reachable": True,
        "first_station_distance_km": round(first_station_dist / 1000, 1),
        "final_fuel_percent": round(final_fuel_percent, 1) if final_fuel_percent
is not None else None

```

```
}  
  
return refuel_points, total_distance, refuel_stats
```