КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет радіофізики,електроніки та комп'ютерних систем
Кафедра комп'ютерної інженерії

Імплементація симплекс-методу для оптимізації витрат в задачах логістики

Дипломна робота магістра

студента 2 року навчання

Спеціальність: 123 «Комп'ютерна інженерія»

Глушка Гліба Федоровича

Науковий керівник:

Кандидат фізико-математичних наук,

асистент кафедри МТРФ

Іваненко Дмитро Олександрович

Рецензент
д. ф.-м. н., професор кафедри

теорії ймовірностей,

статистики та актуарної математики

механіко-математичного факультету

Шевченко Георгій Михайлович

До захисту допускаю

Протокол засідання кафедри від
«___»_____2022р. №_____

Завідувач кафедрою

Бойко Ю.В.

КИЇВ 2022

РЕФЕРАТ

Дипломна робота за об’ємом складає 60 сторінок, містить 20 рисунків, 2 таблиці, 4 додатки, використано 19 інформаційних джерел.

Розглянуто: алгоритми для знаходження шляху з мінімальною вартістю, можливість автоматичної побудови маршруту з урахуванням заправних станцій, характеристик транспортних засобів.

Зроблено: реалізовано алгоритм, в основі якого лежить симплекс-метод для оптимізації витрат на пальне, розроблено прототип веб-застосування для побудови та оптимізації маршруту.

Ключові слова: Симплекс-метод, побудова маршруту, оптимізація, сайт.

ЗМІСТ

1. ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

АЗС – автозаправна станція;

ТЗ – транспортний засіб;

ОС – операційна система;

Фреймворк – програмна платформа, що визначає структуру програмної системи та спрощує розробку шляхом об’єднання різних компонентів програмного проекту;

Front-end – клієнтська сторона користувацького інтерфейсу;

Back-end – програмно-апаратна частина сервісу;

API – application programming interface, опис способів, якими комп’ютерна програма взаємодіє з іншою програмою;

ORM – object-relational mapping, технологія програмування, що пов’язує бази даних з концепціями об’єктно-орієнтованих мов програмування;

MSA – Micro Service Architecture – мікросервісна архітектура;

DDD – Domain-Driven Architecture - предметно-орієнтована архітектура;

2. ВСТУП

Сьогодні як ніколи наше життя тим чи іншим чином пов’язане з автомобільним транспортом. Щоденно люди використовують ТЗ для пересування містом, перевезення вантажів, подорожей тощо.

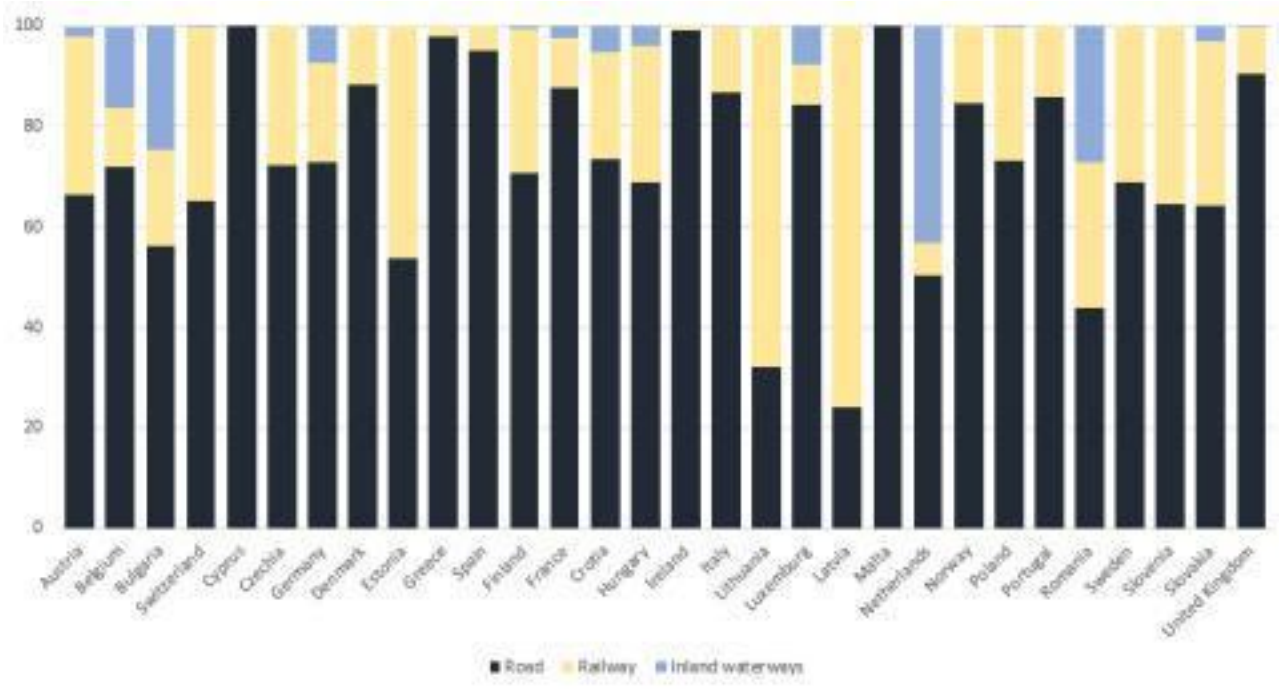

Рисунок 1 – графік розподілу вантажних перевезень у Європі за 2020[1]

Так, за даними статистичної служби Європейського Союзу Євростат серед вантажних перевезень у країнах Європейського Союзу, переважна більшість вантажу перевозиться саме автомобільними дорогами.

Аналогічна ситуація спостерігається з наземними пасажирськими переміщеннями. Автомобілі та автобуси мають найбільшу частку пасажирських перевезень у Європейських країнах.

З цього можна зробити висновок, що дорожні транспортні засоби, такі як машини, автобуси, вантажівки грають важливу роль у пасажирських та вантажних перевезеннях у країнах Європи, тому оптимізація логістичних операцій у автоперевезеннях відіграє важливу роль.

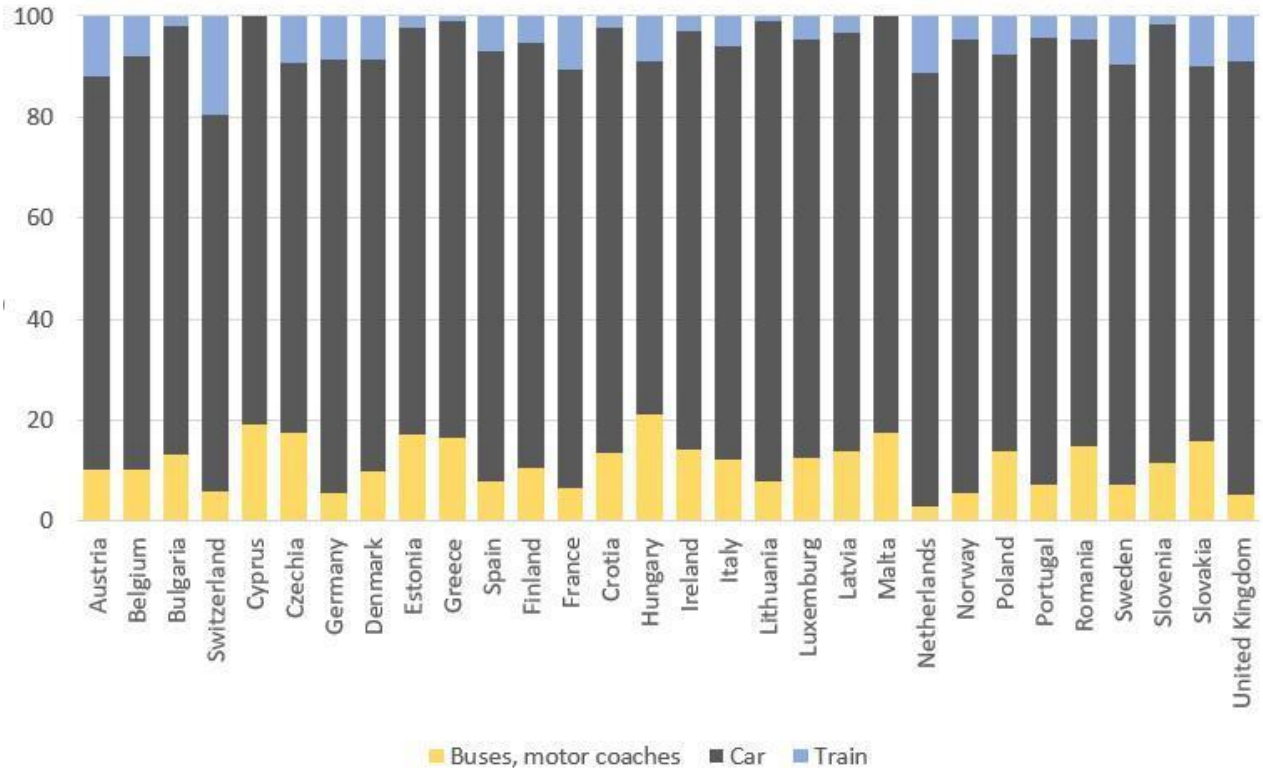

Рисунок 2 – графік розподілу пасажирських перевезень у Європі за 2020[2]

Собівартість перевезень складається з постійних і змінних витрат. До постійних витрат зокрема належить зарплата водіїв, диспетчерів, спеціалістів по ремонту, обслуговування транспортних засобів, паркування. До змінних витрат відносяться вартість палива, шин, запасних деталей, амортизації транспорту.

Основна економія досягається за рахунок зменшення змінних витрат, зокрема палива. Зменшення витрат на пальне можна досягти різними способами, наприклад, покращуючи ТЗ, модифікуючи двигун, збільшення кількості транспортуючих ресурсів або планування маршруту.

Дана робота присвячена автоматичній побудові маршруту, розрахунку необхідної кількості пального та вибору місць заправки оптимізуючи витрати на

пальне. Для повного розуміння проблематики та можливих рішень в роботі зроблений аналітичний огляд вже наявних застосувань для побудови маршрутів.

Актуальність роботи полягає в отриманні маршруту, при якому, витрати на пальне будуть мінімальні та відповідати обмеженням які накладаються на ТЗ, вибір мереж АЗС тощо.

Новизною наукових досліджень є застосування симплекс-методу із обмеженнями типу нерівностей до розв’язання транспортної задачі оптимізації витрат на пальне.

Метою випускної кваліфікаційної роботи є побудова алгоритму для побудови оптимального за витратами на пальне маршруту та його програмна імплементація зокрема розробка веб застосування

3. ОГЛЯД НАУКОВИХ ПУБЛІКАЦІЙ ТА ЛІТЕРАТУРНИХ ДЖЕРЕЛ

З розвитком технологій люди намагались вирішити проблему побудови оптимального маршруту. Раніше, до буму технологій, люди декілька століть використовували паперові мапи для навігації, але мапи треба постійно оновлювати, оскільки будуються нові дороги, паперові мапи «не знають» про поточну ситуацію на дорозі і не мають повної інформації щодо місцезнаходження АЗС та цін на них.

Зараз у кожної людини в кишені є смартфон з встановленим застосуванням, у якому можна знайти кафе чи АЗС поряд, можна побудувати маршрут, вибираючи точки старту/призначення, зупинки, враховуючи вимоги до доріг відповідно до задачі.

Наразі є декілька готових рішень, які надають можливості побудови маршруту та пошуку АЗС. Було розглянуто більше 10 застосувань, серед яких було вибрано найбільш популярне застосування Google Maps та застосування Uta Stations Finder, яке має найбільш схожі можливості.

Застосування та сайт Google Maps[3] – дуже популярний сайт для знаходження потрібних місць на карті та побудови маршруту.

Сайт не дозволяє переглянути АЗС на маршруті, їх можна знайти лише в радіусі певної точки. Сервіс не розрахований для побудови маршруту вантажного транспорту. Сервіс не розраховує необхідну кількість палива. Google Maps пропонує декілька альтернативних маршрутів, оптимізованих лише за часом, або за відстанню

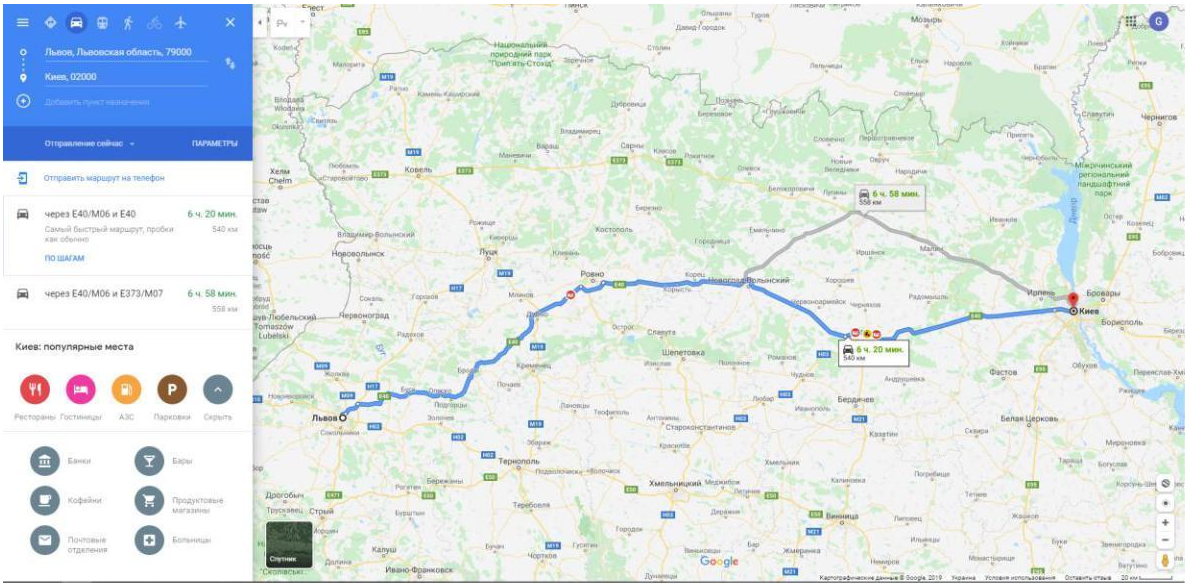

Рисунок 3 – Сайт Google Maps

Мобільний додаток Uta Stations Finder[4] – додаток спрямований на пошук АЗС, та надає інформацію про АЗС мережі UTA. У додатку можна побудувати маршрут та переглянути АЗС, які знаходяться поруч з маршрутом. Зауважимо, що цей додаток використовує API Google Maps і є розширенням цього застосунку, тому його функціонал побудови маршруту має ті ж переваги і недоліки, що і Google Maps.

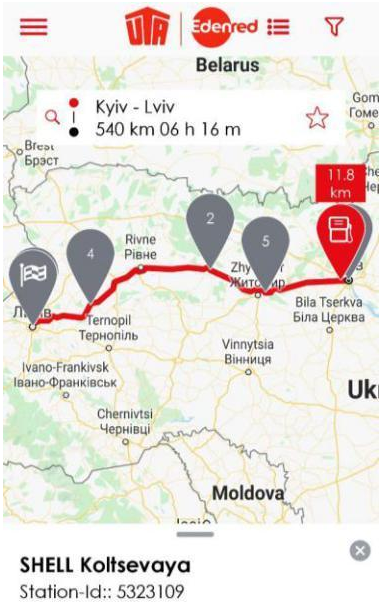

Рисунок 4 – Додаток Uta Stations Finder

Додаток не надає інформації про АЗС, які не належать до мережі UTA. Додаток не розрахований для побудови маршруту вантажного транспорту. Додаток не розраховує необхідну кількість палива. Додаток не пропонує альтернативних маршрутів і невідомо, чи враховує поточну ситуацію на дорозі.

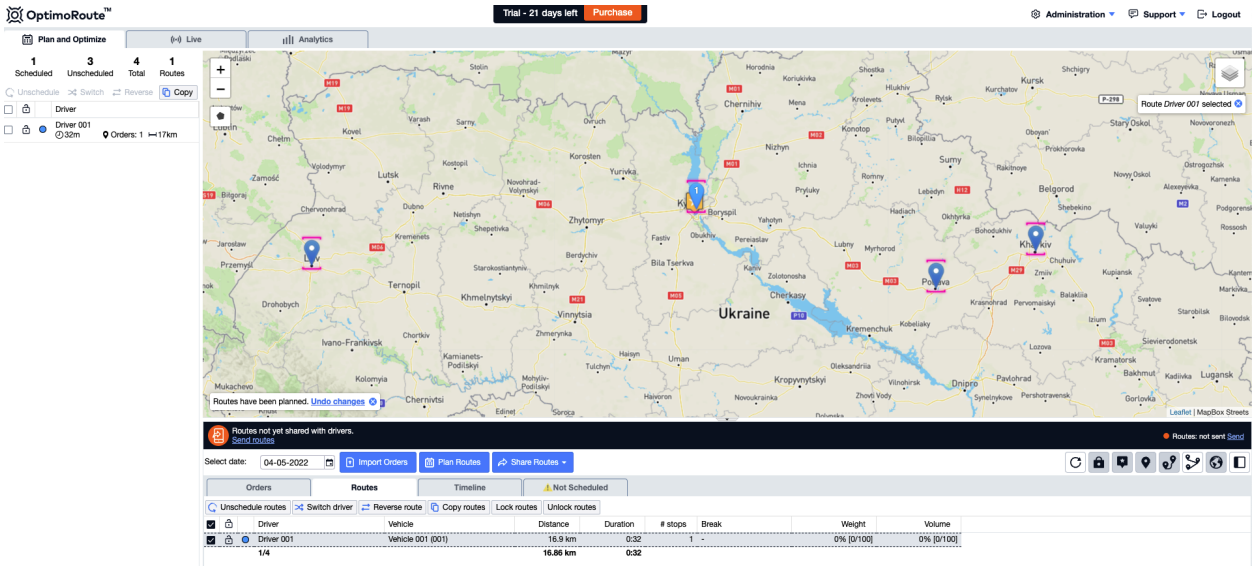

Рисунок 5 – Додаток OptimoRoute

Додаток OptimoRoute[5] - додаток спрямований на оптимізацію маршрутів доставки, у додатку можна задавши точки доставки створити та оптимізувати маршрут, з прив'язкою до завантаження/відвантаження грузу, додаток дає можливість вибрати водія, налаштувати час, налаштувати транспортний засіб. Додаток працює тільки для маршрутів, які можна виконати за 1 робочий день (по одному місту), маршрут оптимізується лише за часом і відстаню, немає можливості пошуку АЗС, не враховується ціна пального.

4. ПОСТАНОВКА ЗАДАЧІ

Для вирішення транспортної задачі необхідно визначити перелік всіх можливих вимог задачі, а також вказати, які з вимог є посутніми, а які ні. Всі вимоги можна розбити на такі групи:

- вимоги до ТЗ;
- вимоги до місць, які треба відвідати на маршруті;
- вимоги до пошуку логістичної карти маршруту;
- інші вимоги

4.1 Вимоги до транспортних засобів:

Тип ТЗ. ТЗ бувають легкові, вантажні та громадські. Від типу транспорту залежать витрати пального і обмеження пересування транспорту деякими ділянками маршруту, наприклад на паромах, на мостах, під мостами, пересування у центрі міст, що проходять взводж маршруту.

Витрати пального. Різні моделі і типи ТЗ мають різні витрати пального на пройдену відстань. Якщо ТЗ вантажного типу, то витрати пального залежать ще й від вантажу.

4.2 Вимоги до місць, які треба відвідати на маршруті:

Розташування місця. Це основний параметр транспортної задачі, від нього залежить, які поруч є заправні станції, і власне ціни на пальне. Також цей параметр визначає можливість дістатись до місця різними видами транспорту.

4.3 Вимоги до пошуку логістичної карти маршрутів:

Критерій оптимізації логістичної карти маршруту. Оптимізувати можна за часом, за відстанню і/або за витратами на пальне. В цій роботі розглядається задача оптимізації витрат на пальне, навіть якщо при цьому виростуть час і пройдена відстань.

Урахування топології доріг при пошуку карти маршруту. Від топології залежить можливість ТЗ дістатись з однієї точки до іншої, зокрема може знадобитись використання платних доріг.

4.4 Інші вимоги.

До інших вимог можна віднести податкову політику в тій чи іншій країні, можливість використання платних доріг, використання паромів, максимальний час безперервної роботи водія, витрати на амортизацію транспортного засобу та зарплати водія.

5. ПРОЕКТУВАННЯ ТА РЕАЛІЗАЦІЯ

5.1 Архітектура застосування

Так як застосування складається з різних, незалежних між собою частин, то для його реалізації було обрано мікросервісну архітектуру (MSA).

MSA – принципи організації розподіленої системи на основі мікросервісів та їх взаємодії один з одним та з середовищем через мережу, а також принципи, що визначають проектування архітектури, її створення та розвиток. Мікросервіс – основна складова частина MSA, яка має такі властивості:

- Відповідає лише за один окремий фрагмент функціональності;
- Працює незалежно від інших мікросервісів;
- Має власне сховище даних;
- Невеликий – вся логіка мікросервісу і весь його код може знати/пам’ятати одна людина;
- Легко замінити або переписати;

MSA надає такі переваги як легка масштабованість системи, уникнення складних залежностей, можливість розробляти кожен сервіс з використанням різних більш доречних технологій, можливість розробляти сервіси різними людьми, усуває проблему недієздатності при виникненні проблеми в одному з сервісів та підтримує безпеку на високому рівні. Але вибір такої архітектури додає інші проблеми і складності, такі як високе використання апаратних ресурсів (оперативна пам’ять, постійна пам’ять, ресурси процесора), ускладнення тестування, розробки і відлагодження, необхідність точного опису інтерфейсів і формату передачі даних, велике навантаження на мережу та налаштування мережі, якою сервіси обмінюються даними.

На рисунку зображена архітектура застосування, зеленими шестикутниками позначені сервіси, з яких складається застосування, синіми хмарами позначено сторонні сервіси, стрілки означають зв’язок між сервісами.

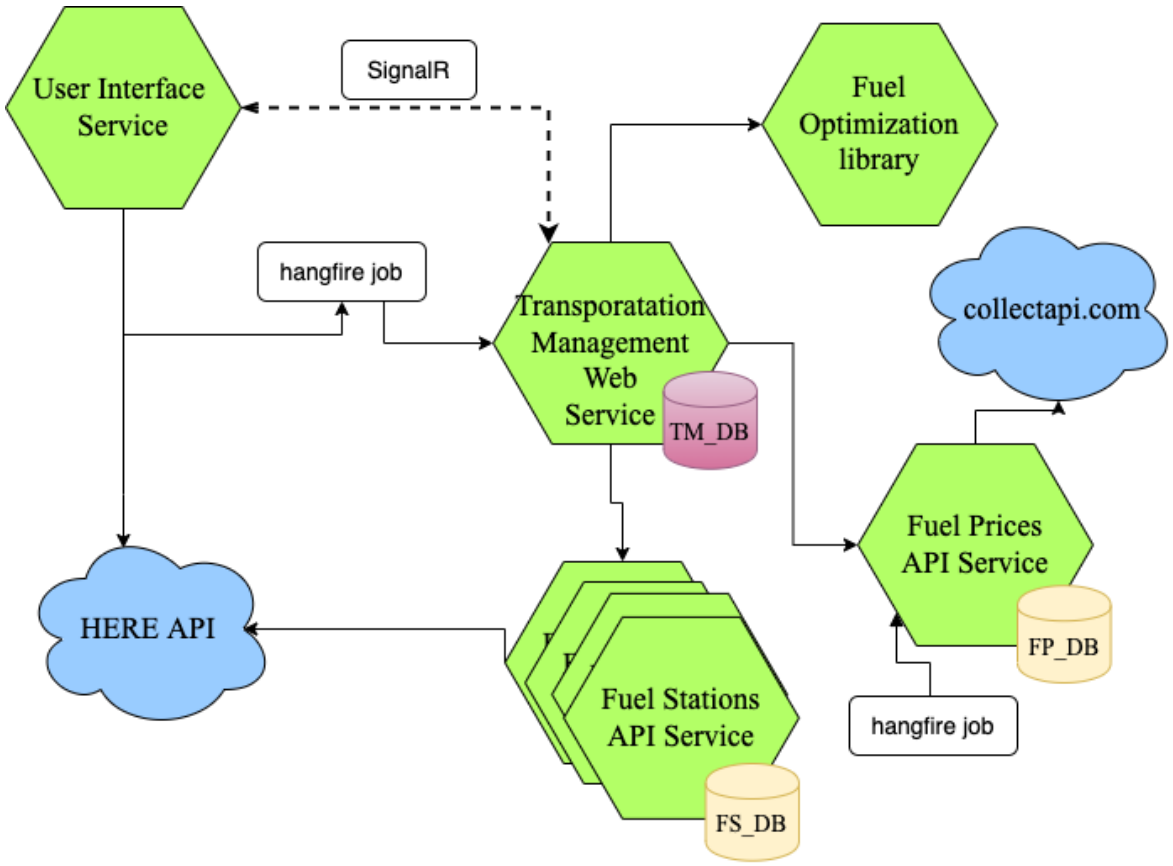

Рисунок 6 – діаграма архітектури застосування

5.2 Обрані технології та інструменти

5.2.1 Розгортання застосування

Деплой (deploy) або розгортання – процес переводу коду проекту до робочого стану на сервері, який складається з кількох частин:

- Компіляція коду
- Вивантаження коду на сервер
- Підтягування залежностей проекту
- Виконання сценаріїв налаштування на сервері
- Запуск застосування

Для розгортання застосування було використано хмарну платформу DigitalOcean[6], яка надає послуги хмарного хостингу застосувань. Для

розгортання застосування треба орендувати віртуальну машину (Dropblet), яка має наперед встановлені інструменти для запуску застосування. Також DigitalOcean надає інструменти для моніторингу, масштабування та менеджменту Droplets.

Для сайту було встановлено SSL-сертифікат. SSL-сертифікат – це цифровий підпис, що дозволяє забезпечити безпечне шифрування між веб-сервером та веб-клієнтом(браузером) шляхом використання протоколу HTTPS. Використання протоколу HTTPS гарантує шифрування даних, що передаються мережею, це потрібно для того, щоб зловмисники, що перехопили інформацію, що пересилається, не змогли її розшифрувати та використати.

В основі будь-якого метода шифрування лежить ключ. Ключ – спосіб зашифрувати або розшифрувати повідомлення. У роботі SSL-сертифікату беруть участь 3 типи ключів:

- публічний ключ – використовується для шифрування повідомлення. Браузер використовує його, коли треба відправити дані користувача на сервер;
- приватний ключ – використовується для дешифрування повідомлення, він знаходиться на сервері і сервер використовує його, коли отримує повідомлення;
- сеансовий ключ – використовується для шифрування та дешифрування повідомлення, цей тип ключа генерується для кожного сеансу, тобто браузер генерує ключ на той час, коли користувач знаходиться на сайті, коли користувач виходить з сайту сеанс завершується і ключ інвалідується;

Для отримання сертифікату було використано можливості центру сертифікації Let’s Encrypt. Let’s Encrypt – це центр сертифікації, що автоматизовано надає безкоштовні криптографічні сертифікати терміном на 3 місяці. Зараз результат дипломної роботи у вигляді сайту можна знайти за адресою 167.71.33.134.

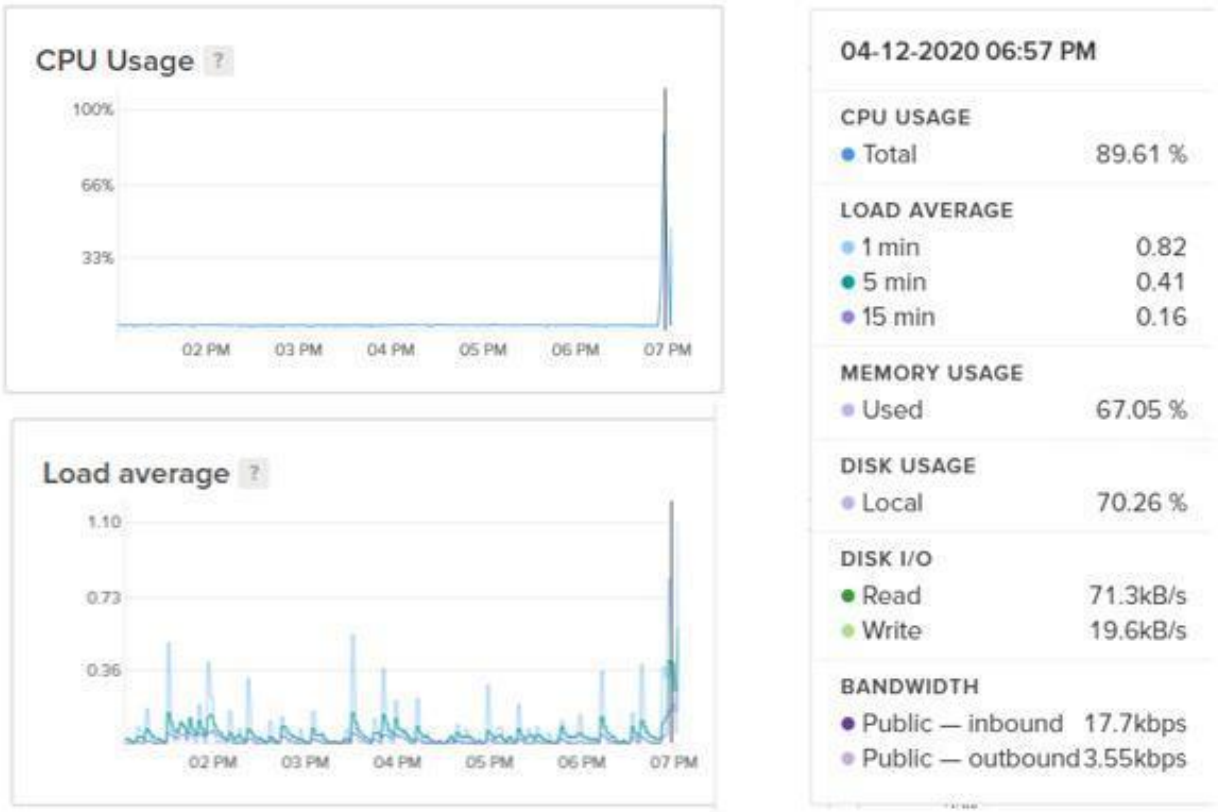

Рисунок 7 – моніторинг використання ресурсів застосуванням

Для автоматизації розгортання та керування сервісами було використано Docker.

Docker[7] - програмне забезпечення для розробки, тестування доставки і виконання застосувань. Docker упаковує ПО в стандартизовані блоки, які називаються контейнерами. Кожен контейнер включає в себе все необхідне для роботи застосування: бібліотеки, системні інструменти, код і середовище виконання. В основі Docker лежить стандартизований спосіб виконання коду, контейнери створюють віртуальне представлення серверної операційної системи

Переваги Docker:

- Масштабування – Docker дозволяє легко масштабувати застосування створюючи нові контейнери, швидке і легке розгортання і налаштування контейнерів на інших серверах;
- Абстрагування застосування від хосту – контейнери з’єднуються з хостом через певний інтерфейс, а саме застосування не залежить від архітектури або ресурсів хоста;
- Ізолювання середовища виконання – ізоляція у контейнерах відноситься до ізоляції на рівні процесів, при цьому контейнери працюють на самому ядрі, що дозволяє їх швидко працювати та не використовує додаткові ресурси на ОС як при віртуалізації
- Використання прошарків – якщо декілька контейнерів використовують однаковий прошарок, Docker дозволяє їм використовувати один прошарок не дублюючи його, що зменшує використання дискового простору;
- Ефективна міграція – застосування на основі Docker-контейнерів дозволяє швидко і ефективно переносити застосування на новий сервер або хмару;

Для створення, налаштування та автоматизації застосування на платформі Docker було створено файли Dockerfile та docker-compose.yml. Dockerfile – файл, що містить в собі всі команди, що необхідні для збірки контейнеру

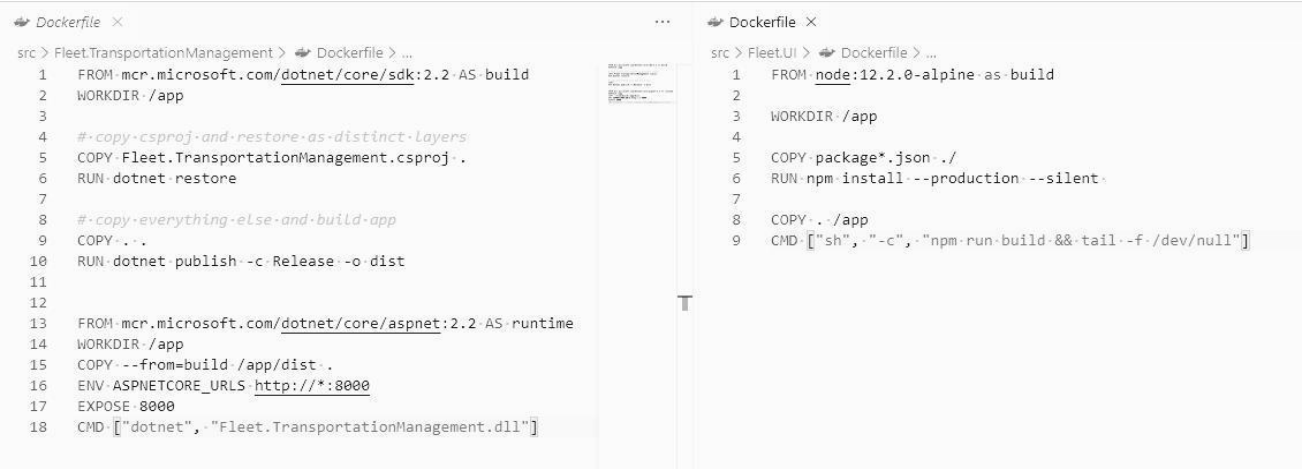

Рисунок 8 – приклад Dockerfiles для запуску сервісів
на платформах ASP.NET Core та Node.js

docker-compose.yml – файл, що використовується застосунком docker-compose для визначення і запуску багатоконтейнерних застосувань. У docker-compose.yml визначаються сервіси, що повинні запускатися в одному середовищі, а також визначається налаштування середовища, у якій буде працювати кожен окремий контейнер.

Серед основних налаштувань docker-compose.yml:

- image – образ контейнеру;
- context – розташування файлів, які необхідні для збірки;
- port – співвідношення портів в середині контейнера та на хості;
- environment –змінні, які передаються у середовище виконання;
- volumes – співвідношення файлів і директорій у контейнері і на хості;
- command – команда, що виконується після збірки контейнера;

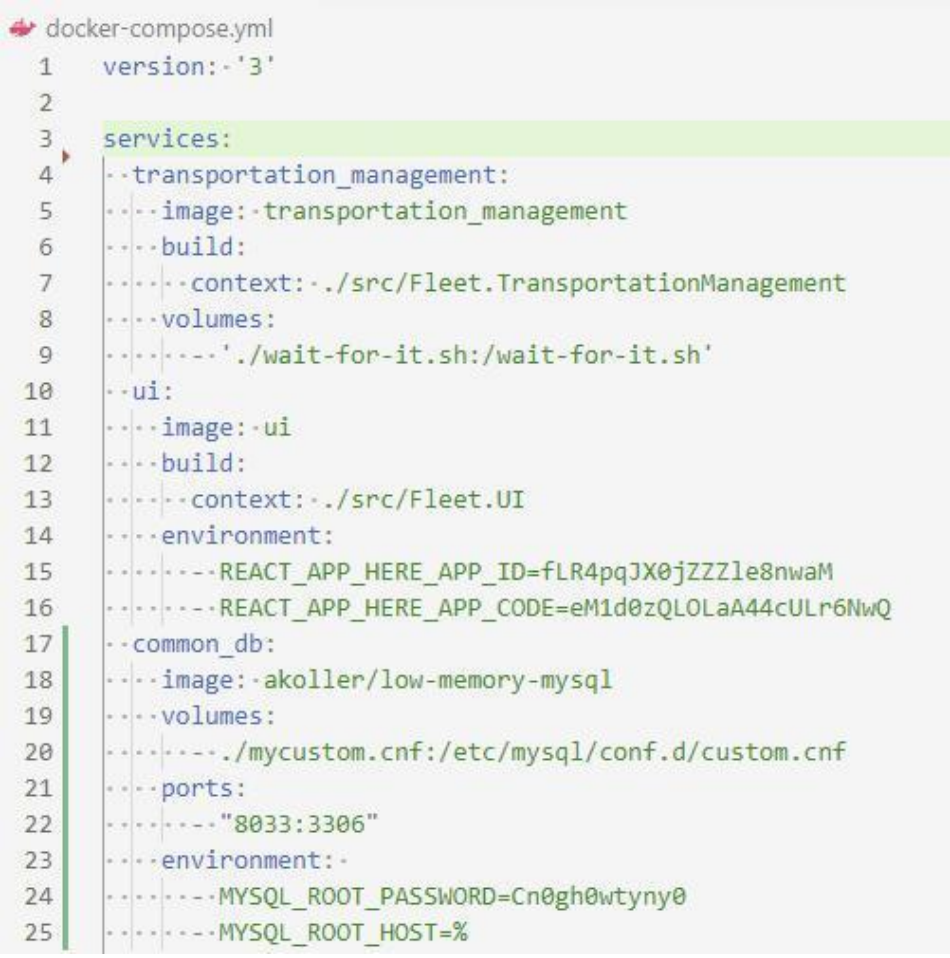

Рисунок 9 – приклад частини файлу docker-compose.yml

Continuous integration and continuous delivery (CI/CD) - безперервна інтеграція і безперервне розгортання. Безперервна інтеграція - концепція, при якій кожна зміна подана до програми, тестується автоматично і безперервно. Для різних типів гілок можна налаштувани різні процеси автоматизації.

Безперервне розгортання - наступний крок безперервної інтеграції. Окрім того, що на кожну зміну відбувається тестування і створюється одразу версія застосунку і автоматично розгортається на середовище роботи програми.

Для реалізації CI/CD було обрано систему для безперевних інтеграцій Gitlab. При створенні релізної гілки, запускається автоматичний процес, який

спочатку компілює і публікує дзеркало програми, запускає тестування, і при успішному завершенню тестів відбувається автоматичне розгортання на сервері DigitalOcean.

Приклад автоматизації зображено на схемі:

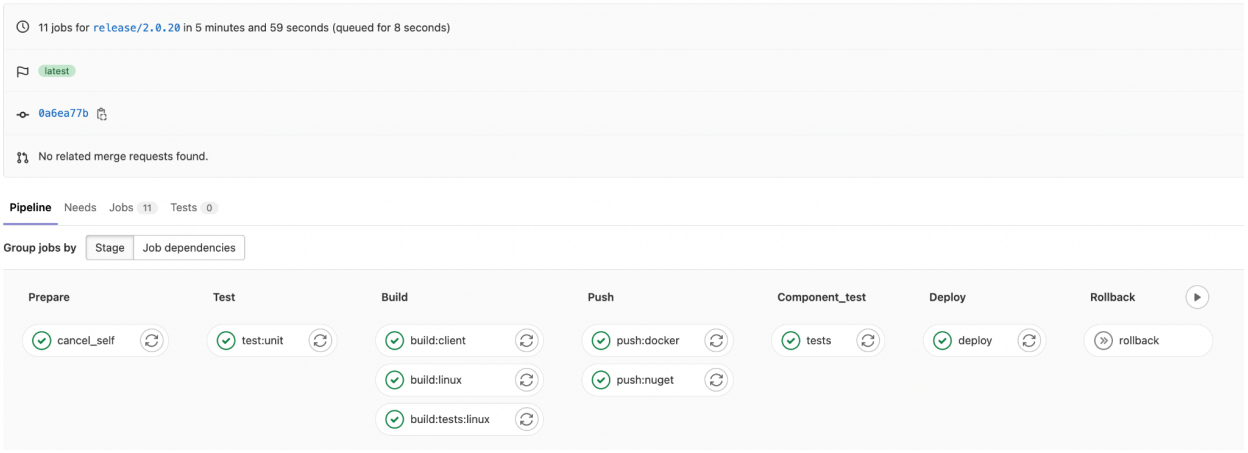

Рисунок 10 - Автоматизовані процеси CI/CD

5.2.2 Користувацький інтерфейс застосування (Front-end)

Для реалізації Front-End була вибрана мова програмування JavaScript та бібліотека React.Js.

JavaScript – крос-платформна скриптова мова програмування, призначена для підготовки сценаріїв виконання сторінок, що надає можливість зробити Web-сторінки більш інтерактивними і функціональними.

React.Js[8] – бібліотека для створення користувацьких інтерфейсів. Основні переваги React.Js:

- *Декларативність.* Декларативні представлення роблять код більш зрозумілим, передбачуваним та спрощують відлагодження
- *В основі лежать компоненти.* Інкапсульовані компоненти не залежать між собою, а залежать лише від внутрішнього стану, тому достатньо лише описати як компоненти виглядають у різних станах.
- *Життєвий цикл компоненту.* Зрозумілий життєвий цикл компоненту та надання найважливіших функцій. Якщо стан компоненту змінений, то він буде автоматично відрендерений заново. Надається

можливість визначати що відбувається при першому створенні компоненту ComponentDidMount, при видалені компоненту ComponentWillUnmount, при зміні стану ShouldComponentUpdate та інші.

- *Легкість.* React.js це лише бібліотека, а не повноцінний фреймворк, це лише рівень представлення над мовою JavaScript. Він надає список шаблонів і деякі callback-функції, які потім транслюються в Html, Css
  та Javascript.

5.2.3 Програмно-апаратна частина застосування (Back-end)

Для реалізації Back-End була вибрана мова програмування C# та фреймворк .NET Core.

C# - високорівнева об’єктно-орієнтована мова програмування, що підтримує поліморфізм, спадкування, інкапсуляцію. Об’єктно-орієнтований підхід надає можливість для розробки великих, але гнучких та легкомасштабованих.

.NET Core[9] – крос-платформний фреймворк з відкритим вихідним кодом, що використовується для розробки додатків. Основні переваги .NET Core:

- *Підтримка декількох мов програмування.* В основі фреймворку лежить загальномовне середовище виконання CLR(Common Language Runtime), завдяки чому .NET підтримує декілька мов: C# VB.NET, C++, F#. При компіляції код на будь-якій з цих мов
- *Кросплатформність.* .NET Core – адаптивна платформа, яка підтримується на більшості сучасних ОС таких як: Windows, MacOs, Linux. Використовуючи додаткових технологій на платформі .NET можна розробляти додатки для Android, IOS, Tizen.
- *Потужна бібліотека класів.* .NET має єдину, для всіх підтримуваних мов, бібліотеку класів, яка використовується у всіх додатках на мові C# від консольного застосування до веб-сайту.

- *Підтримка різних технологій.* Загальномовне середовище виконання CLR та базова бібліотека класів – основа для повного стеку технологій, які можуть використовуватись розробниками для побудови різних застосувань. Наприклад, для роботи з базами даних в цьому стеці є технології ADO.NET та Entity Framework. Для створення графічних застосувань - WPF і UWP. Для створення мобільних додатків – Xamarin.
- *Автоматичний збір сміття.* Garbage Collection (GC) – автоматично керує пам’яттю. CLR керує виділенням та звільненням пам’яті, що унеможливлює витік пам’яті.

5.2.4 Веб-сервер

Веб-сервер є прошарком між клієнтською частиною застосування та серверною чистиною. При відкритті сторінки, браузер з’єднується з с сервером сайту(застосування), потім сервер шукає дані, які потрібні для сторінки, це можуть бути html-сторінки, зображення, дані різних форматів(JSON, XML), і відправляє їх в браузер. Це найпростіший тип запиту і він розглядається форматі одного потоку. Традиційні веб-сервери створюють окремий потік для кожного запиту.

Для реалізації функцій веб-серверу було використано Ngninx. Ngninx – веб-сервер, основні функції якого: веб-обслуговування HTTP, зворотній проксі-сервер, балансувальник навантаження і поштовий проксі-сервер. Основна відмінність Ngninx від інших веб-серверів – асинхронна архітектура, що керується подіями, тобто аналогічні потоки можуть управлятися одним процесом, а кожний процес складається з менших блоків, які називаються робочими з’єднаннями. Робочі з’єднання обробляють запити і доставляють їх в головний процес, а головний процес в свою чергу надає результат запитів браузеру або іншим сервісам, що надсилають запити.

5.2.5 База даних

В цій задачі дані мають фіксований формат, наприклад маршрут містить інформацію про точки, кількість пального і типу автомобіля. Кожна точка теж має фіксовані властивості – адреса, координати, та тип. Якщо тип – АЗС, то додаються додаткові поля: назва мережі та вартість пального. Виходячи з цього, було обрано реляційну базу даних, оскільки нереляційні бази даних (NoSQL) мають інші формати даних, які не є фіксованими, наприклад документо-орієнтована база даних MongoDB зберігає дані у вигляді документу форматі JSON. Redis – нереляційна база даних, що зберігає дані у форматі пари ключ-значення.

Для реалізації сховища даних було обрано систему керування MySQL. MySQL – вільна реляційна система керування базами даних, розроблена компанією Oracle.

- *Масштабованість.* MySQL підтримує швидкий процесор запитів, який забезпечує підтримку баз даних дуже великих розмірів.
- *Продуктивність.* MySQL використовує складові індекси, нові алгоритми хешування та злиття, розподіленні та паралельні запити.
- *Ціна.* MySQL безкоштовна для некомерційного використання
- *Використання ресурсів.* MySQL не потребує великої кількості ресурсів процесора та оперативної пам’яті.

Для роботи з базою даних на стороні Back-end було використано Entity Framework Core. Фреймворк Entity Framework Core підтримує систему керування базами даних MySQL.

Entity Framework Core[10] – ORM інструмент, що надає високий рівень абстракції, який дозволяє абстрагуватись від певної системи керування базами даних, та надає інтерфейси для оперування таблицями та індексами на концептуальному рівні.

За вибраною архітектурою, а саме мікросервісною архітектурою (MSA), кожен сервіс повинен мати своє окреме сховище даних. На жаль, через низьку вартість орендованого серверу, оперативної пам’яті не вистачає на 5 сервісів і 3 окремих сервери баз даних, тому було прийнято рішення запустити лише один сервер баз даних MySQL та створити 3 окремі бази даних на цьому сервері. Також для оптимального використання ресурсів, було використано неофіційний образ серверу MySQL, a low-memory-build-mysql[11] образ автора akoller, який оптимізовано для серверів з обмеженою оперативною пам’яттю. Також було реалізовано конфігурацію docker-compose.yml, що використовує окремий сервер, для того, щоб при необхідності запустити його на сервері с більшими об’ємами ресурсів.

5.2.6 Проектування і реалізація сервісів

5.2.6.1 Сервіс користувацького інтерфейсу

Користувацький інтерфейс реалізований мовою JavaScript з використанням бібліотеки React.Js.

Користувацький інтерфейс складається з декількох сторінок. Дві основні з них: створення нового маршруту та перегляд оптимізованого маршруту з додатковою інформацією.

Сторінка створення нового маршруту складається з форми введення. На цій формі вибирається машина з існуючого набору, або створюється нова з потрібними характеристиками, які надалі зберігається у локальному сховищі браузера для подальшого використання, також вводиться кількість палива, що залишилось. Далі є інтерфейс введення маршруту, маршрут складається з кількох точок (максимум 15), точки можна створювати вводячи адресу, або подвійним клацанням на мапі. Маршрут можна редагувати перетягуючи лінію, яка автоматично перебудовується.

Для побудови, відображення і редагування маршруту було використано платформу HERE[12], що надає різні API для роботи з мапами. Було використано такі API:

- Routing API – для побудови маршруту та отримання даних для відображення маршруту на мапі;
- Geocoding and Search API – для отримання інформації про локацію на мапі за її координатами та отримання координат за адресою;
- Map Tile API – для отримання зображень мапи;

Для розробки інтерфейсу було використано бібліотеку Material-UI, що надає можливість створювати користувацькі інтерфейси, використовуючи готові компоненти та API для роботи з ними.

5.2.6.2 Сервіс оновлення цін на пальне

Для коректної побудови маршруту та оптимального вибору АЗС необхідні актуальні ціни на пальне. Набір сервісів HERE APIs включає в себе Fuel Prices API, що надає інформацію про АЗС та ціни на них, але цей сервіс не входить у безкоштовну підписку на сервіси HERE

Інформацію про ціни на заправних станція надають багато сервісів, але всі вони платні, а також, їх використання є неможливим, через обмеження компанії HERE. Ліцензія, за якою використовуються сервіси компанії HERE забороняє використовувати її сервіси з сервісами, що конкурують з компанією. Тобто, заборонено використовувати зображення мапи від сервісу HERE та отримувати інформацію про заправні станції від компанії Google Maps.

Єдиний варіант для отримання цін на пальне це використання безкоштовних агрегаторів цін. Сайт CollectAPI[13] надає інформацію про середні ціни на АЗС у європейских країнах.

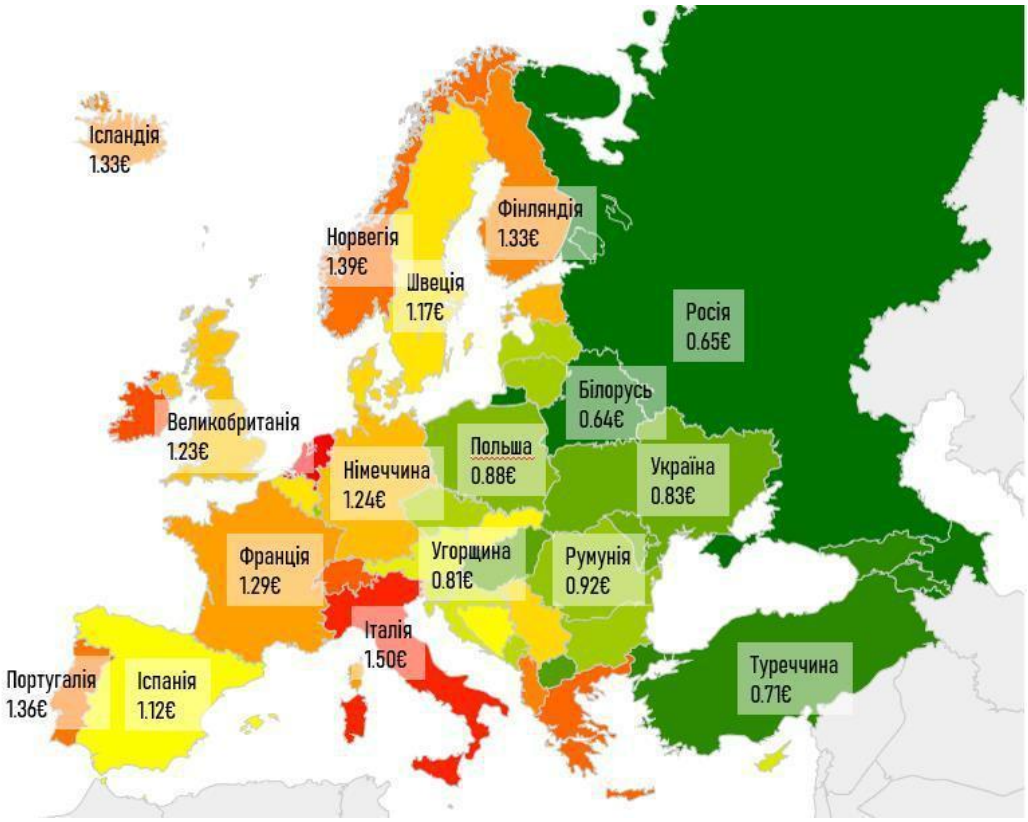

Рисунок 11 – мапа Європи з цінами на пальне (17.02.2022)

Мікросервіс для оновлення цін на пальне, реалізований на мові програмування C# на платформі .NET Core, оновлює ціни на пальне при першому запуску мікросервісу і далі кожну добу

5.2.6.3 Сервіс керування поїздками

Сервіс керує усією логікою пов’язаною з створенням маршрутів, попередньою обробкою даних, зберіганням та видачою результату.

Сервіс виконує такі функції: створення нового маршруту, зберігання початкового маршруту, виклик сервісу з пошуку АЗС, розрахунку пального, необхідного для подолання кожного відрізку маршруту, виклик сервісу з оптимізації вибору АЗС, створення результату, що містить всі необхідні дані та зберігання даних з всіх етапів виконання.

Сервіс реалізований мовою програмування C# на платформі .NET Core, а також використовує бібліотеку SignalR для оновлення статусу виконання оптимізації в режимі реального часу через Websocket.

При реалізації сервісу керування поїздками було використано Domain-Driven Architecture[14] (предметно-орієнтовану архітектуру).

Domain-Driven Design — це підхід до проектування програмного забезпечення, який поєднує реалізацію системи з моделлю, що постійно розвивається, залишаючи осторонь невідповідні деталі, такі як мови програмування, інфраструктурні технології.

Основні концепції предметно-орієнтовної архітектури, які застовні до даної задачі:

Bounded context (обмежений контекст) — це концептуальна межа навколо частин програми та проекту з точки зору сфери бізнесу і коду. Він групує пов’язані компоненти та поняття та уникає двозначності, оскільки деякі з них можуть мати подібне значення без чіткого контексту.

В даній задачі було виділено обмежений контекст маршруту, який і є основною бізнес моделю в даній задачі.

Context Mapping (планування контексту) - це концепція, яка описує взаємодію між різними частинами проєкту, як обмеженні контексти взаємопов’язанні і як вони взаємодіють між собою. В даній задачі одним із прикладів планування контекстів є взаємодія між контекстом маршруту, контекстом автозаправних станцій. Ці контексти очевидно пов’язані між собою, але неможна сказати, що один з них включає в собі інший, тому правильним рішенням є розділити їх та описати їхню взаємодію.

Anti-corruption Layer (шар захисту від зловживання) - ця концепція розділяє різні контексти, щоб уникнути некоректних даних та неправильних взаємодій. Нижній обмежений контекст реалізує рівень, який перекладає дані або об’єкти, що надходять із контексту вище по потоку, забезпечуючи підтримку внутрішньої моделі. В даній задачі ця концепція відповідає за захист та розмежування контекстів, наприклад контекст маршруту включає інформацію про довжину маршруту, в той час як інший контекст (контекст мапи) відповідає за пошук і прорахунок відстаней, тому якщо будуть необхідні зміни в контексті мапи, контекст маршруту буде захищений від

невалідних даних з контексту мапи, наприклад: при зміні сервісу для розрахунку відстаней між точками, сервіс почав повертати відстані у милях, а в контесті маршруту ми використовуємо кілометри, то шар захисту від зловживання буде реалізовувати валідацію даних, і при необхідності конвертувати дані в потрібний вигляд.

Aggregates (агрегати) - це концепція пов'язаних сутностей та об'яктів значень об'єднаних у групи, що представляють межу транзакції. Кожен агрегат має сутність, яка звернена назовні і контролює доступ до об'єктів всередині межі. Жодні об'єкти всередині агрегату не можуть бути викликанні або зміненні безпосередньо з зовнішнього контексту, таким чином зберігаючи узгодженість всередині, що дозволяє зберігати валідний стан агрегату у кожен момент його життя. Наприклад, агрегат маршруту містить інформацію про відстані та загальну вартість, при вільному доступі до кожного з цих об'єктів з зовнішнього контексту можливий невалідний стан агрегату, зміна відстаней між точками на маршруті напряму впливає на вартість, тому доступ до об'єкту відстаней необхідно закрити, і надати методи зміни цих відстаней, при яких буде відбуватися перерахунок вартості, таким чином, зберігаючи валідний стан агрегату.

Repositories (репозиторії) - це концепція можливості отримувати і зберігати об'єкти з постійної пам'яті, це може бути файлова система, база даних, кеш, або інше. Для реалізації цієї концепції треба надати інтерфейс, який приховує деталі реалізації клієнта, щоб це не залежало від особливостей інфраструктури, а лише від абстракції. Репозиторії забезпечують інтерфейс, який рівень домену може використовувати для отримання збережених об'єктів, уникаючи тісного зв'язку з логікою зберігання.

Services (сервіси) - ця концепція дозволяє логічно розділити властивості об'єктів та операцій з цими об'єктами. Часто модель домену вимагає певних дій або операцій, які не пов'язані з сутністю, і можуть призвести до спотворення визначення цих сутностей. Служби - це класи, які пропонують операції з об'єктами, без збереження стану.

Отже предметно-орієнтована архітектура пропонує багатошарову архітектуру як спосіб розділення проблем і уникнення неоднозначності. Це досягається шляхом розділення кодової бази на 4 основні рівні: користувацький інтерфейс, шар застосування, домен і інфраструктура.

Основне правило тут полягає в тому, що компоненти в кожному шарі повинні залежати лише від компонентів того ж самого шару, або будь-якого шару під ним. Верхні рівні можуть використовувати компоненти нижчих, просто викликаючи публічні інтерфейси, а нижчі рівні можуть спілкуватися вгору лише за допомогою інверсії контролю (IoC Inversion of Control).

Підсумовуючи:

- ***User interface*** (інтерфейс користувача) - найвищий рівень, відповідає за відображення даний і запис команд користувача, виступає посередником між користувачем і програмою.
- ***Application Layer*** (Шар сервісів) - служить організатором роботи домену, цей шар не відповідає за правила домену, але організовує та делегує об’єкти домену для виконання своєї роботи.
- ***Domain layer*** (шар домену) - шар, який містить в собі бізнес-логіку та правила, часто зберігає бізнес стан і відповідає за його консистентність.
- ***Infrastructure layer*** (шар інфраструктури) - реалізує всі технічні функції необхідні для підтримки роботи вищих рівнів, збереження, обміну повідомленнями, зв’язку між рівнями і іншим.

Псевдокод і взаємодія агрегатів і сервісів описанні в додатку Б і В.

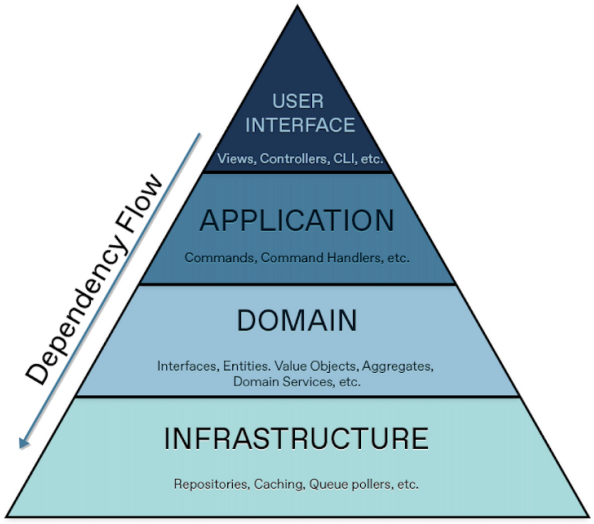

Рисунок 12 - схема залежностей між шарами в предметно-орієнтованій архітектурі

5.2.6.4 Сервіс пошуку заправних станцій

Сервіс відповідає за пошук АЗС та збирає інформацію, про їх місце розташування, назву, мережу, до якої відноситься АЗС, відстань, яку необхідну подолати, щоб доїхати.

Так як, через певні причини, АЗС у кожній країні мають однакову ціну на бензин, шукати заправні станції, що потребують значного відхилення від маршруту не є оптимальним, через додаткову відстань, яку треба подолати до АЗС, пошук заправних станцій на маршруті відбувається на відстані не більше 500 метрів від оригінального маршруту. 500 метрів – оптимальна відстань, так велика імовірність того, що буде знайдено достатня кількість АЗС і не буде виконано забагато ресурсоємних операцій.

Для пошуку заправних станцій на маршруті використовується API від компанії HERE. Алгоритм пошуку АЗС на маршруті: спочатку будується маршрут між двома послідовними точками за допомогою Routing API, далі від цього маршруту будується ізолінія**.**

Ізолінія[16] – умовне позначення на карті, яке представляється у вигляді лінії, у кожній точці якої, вимірювана величина зберігає однакове значення.

Ця ізолінія потрібна для того, щоб відсікти ті заправні станції, які знаходяться дуже далеко від маршруту і їхати до яких не оптимально. Також Here API має обмеження на кількість місць, які можна отримати за один запит.

Далі для кожної з заправних станцій шукається маршрут і відстань, яку треба проїхати, щоб дістатись до станції. Заправні станції сортуються в порядку того, як транспортний засіб буде їхати.

Сервіс реалізований мовою програмування JavaScript на платформі Node.js. Незвична платформа вибрана через наявність бібліотеки turf.js, що реалізована мовою JavaScript.

turf.js[17] – бібліотека з відкритим кодом, що використовується для просторового аналізу. Вона включає реалізовані методи для просторового аналізу, допоміжні функція для отримання даних у форматі GeoJSON, а також інструменти для статистики.

Ця бібліотека використана для побудови ізолінії, пошуку перетину дороги до АЗС з початковою дорогою та отримання даних про АЗС, що знаходяться в межах ізолії.

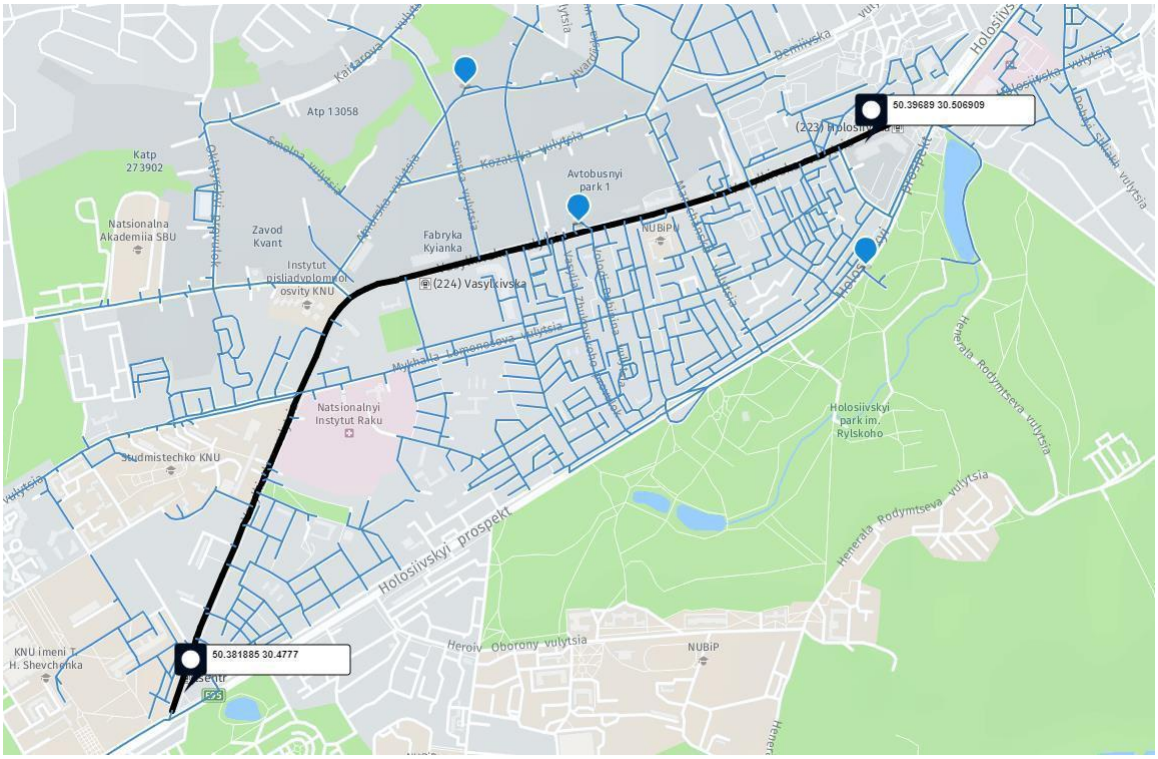

Рисунок 13 – Побудова маршруту та ізолінії із значенням 1км та відображення АЗС

5.2.6.5 Сервіс оптимізації

Сервіс оптимізації – це бібліотека, що приймає на вхід інформацію про маршрут: відстані між точками та АЗС, вартість палива на АЗС, розмір баку машини, інформацію про витрати палива автомобілем і на виході видає оптимальний план заправок.

Порівняння різних алгоритмів оптимізації. Для вирішення даної задачі було розглянуто різні види алгоритмів:

1. *Алгоритми пошуку оптимального маршруту з використанням теорії графів.*

   Ідея полягає у тому, щоб побудувати граф, вершини якого це АЗС та точки, які треба заїхати, а ребра це чисельна характеристика, що може відповідати значенням відстані між точками, кількістю пального, необхідного для подолання відстані або вартість переїзду з однієї точки до

іншої. Так як послідовність точок на маршруті фіксована, задача не зводиться до проблеми комівояжера і її можна вирішити за поліноміальний час такими алгоритмами як пошук у ширину або алгоритм Дейкстри. Нажаль, алгоритми пошуку найкоротшого шляху з використанням теорії графів не підходять, через те, що побудувати відповідний граф неможливо. Так як основна задача це оптимізувати витрати на пальне, не є можливим реалізувати функцію оцінки вартості проїзду між двома точками, бо вартість не є статичною і складається з декількох незалежних параметрів. Наприклад, важко оцінити вартість переїзду тому що, невідомо заздалегідь на якій АЗС буде здійснена заправка, а від цього залежить ціна на пальне, тому вартість переїзду буде залежати від кількості пального, що залишилось і вибору місця заправки. А вибір заправки і є проблемою, яку треба вирішити.

2. *Алгоритм пошуку оптимального маршруту з використанням динамічного програмування.*

   Динамічне програмування – метод вирішення задачі шляхом розбиття її на декілька однакових підзадач, що рекурентно пов’язані між собою. Тобто побудову цілого маршруту можна розбити на підзадачі побудови маршруту між точками, а далі ці частини розбити на маршрути, що включають в собі АЗС, так як кожна частина маршруту в такому випадку є незалежною і не перетинається з іншими частинами, якщо побудувати їх оптимально, то шляхом поєднання оптимальних частин отримується оптимальний план. Технічна реалізація такого алгоритму може виглядати приблизно так. ***dp[point1][point2][current_volume] = cost,*** де

   point1 – поточна точка, це може бути точка, що треба відвідати, або АЗС;

   point2 – наступна точка;

   current_volume – поточна кількість пального у баці;

   cost – вартість переїзду.

Тобто алгоритм буде зберігати у трьохвимірному масиві вартість переїзду з точки *point1* до точки *point2*, таким чином, що у баці залишиться *current_volume* кількість пального.

Псевдо-код реалізація цього алгоритму може бути реалізовано таким чинном:

```
forearch point1 in points:
    foreach point2 in points:
        foreach volume in range(0, tank):
            dp[point1][point2][volume]=
            min(dp[any_point][point1][any_volume])
+ cost(point1, point2, any_volume, volume)
```

Де min(dp[any_point][point1][any_volume]) – мінільна вартість доїхати до точки point1 з будь-якої точки any_point з будь-якою кількістю палива any_volume, а cost(point1, point2, any_volume, volume) – функція, що обраховує вартість доїхати з точки point1 до точки point2, маючи на старті палива any_volume і маючи на фініші палива volume.

Складність реалізації пошуку оптимального маршруту з використанням динамічного програмування в тому, що дуже складна логіка обрахунку вартості пального, тобто складна реалізація методу cost(point1, point2, any_volume, volume). Бо не з усякою кількістю пального any_volume можна проїхати необхідну відстань, а також є багато обмежень, таких як, не у кожній точці можна придбати пальне, тобто варіантів проїхати з однієї точки до іншої, якщо це не АЗС може бути не більше одного, а також складно в подальшому накладати додаткові обмеження, такі, як переїзд платними дорогами, або коли не між всіма точками існує пряма дорога, тобто з однієї точки до другої можна дістатись лише через третю, а повний перебір усіх точок і об’ємів палива лежить в основі цього алгоритму.

Також за найменшими оцінками, складність алгоритму дуже важка, для реалізації швидкого оброблення маршрутів, якщо у нас є N точок, які треба відвідати, між якими знаходяться M заправних станцій, а бак автомобіля V літрів, складність алгоритму виходить $(N+M)^3 * V^2$, тобто якщо є 10 точок, що треба відвідати, 90 АЗС, що знаходяться на маршруті між точками, а бак автомобіля 100л, кількість операцій, нехтуючи складністю обрахунку функції cost() складає $(10+90)^3 * 100^2 = 100^5 = 10^{10}$, при можливій роботі процесора в середньому 10^8 операцій за секунду, тобто 100 секунд, на кожний опрацювання. До цього ще додається час, який треба на пошук відстаней на мапі, а це може тривати декілька хвилин, тому що треба зробити 10^4 HTTP-запитів до сервісу з картами. При цьому кількість заправних станцій збільшується пропорційно відстані між точками, а не кількістю точок. Тобто використовувати цей алгоритм можна, але недоцільно.

3. *Пошук оптимального маршруту з використанням «жадібного» алгоритму* Жадібний алгоритм – алгоритм, ідея якого в прийнятті локально оптимальних рішень на кожному етапі, допускаючи, що кінцеве значення теж буде оптимальним. В даній задачі, жадібним алгоритмом буде алгоритм, який заправляється на мінімальну кількість пального, якої вистачить для подолання відстані до наступної точки. Очевидно, що таким чином глобальний оптимум досяжний лише у випадку однакових цін на всіх АЗС. Але так як цей алгоритм простий у реалізації, він буде використаний для порівняння оптимізації з фінальним алгоритмом.
4. *Пошук оптимального маршруту генетичними алгоритмами*

Генетичний алгоритм - це еврестичний метод пошуку оптимізованих розв’язків для пошуку проблем на основі теорії природного відбору та еволюційної біології. Генетичний алгоритм базується на таких методах, як відбір, мутація, успадковування та інші. Переваги генетичних/еволюційних алгоритмів, що можна оптимізовувати для будь-яких вхідних данних, та

визначати будь-які правила та обмеження. До недоліків можна віднести значну залежність від вхідних даних, залежність від гіперпараметрів, різні гіперпараметри можуть показати різні результати. Генетичні алгоритми часто мають повільну збіжність, що впливає на час оптимізації та результат. Навіть при довгому виконанні оптимізації, генетичні алгоритми не гарантують глобальний мінімум, якщо неправильно підібраний параметр який впливає на випадкові мутації, то алгоритм може зайти в локальний мінімум і ніколи звідти не вийти.

5. *Пошук оптимального маршруту з використанням симплекс-методу*

   Симплекс-метод[15] – алгоритм вирішення задачі оптимізації лінійного програмування методом перебору вершин опуклого багатокутника у багатовимірному просторі. Основна ідея методу полягає у побудові базисних рішень, на яких монотонно зменшується лінійний функціонал, до того моменту, доки виконуються необхідні обмеження локальної оптимальності.

   Алгоритм симплекс-методу такий: спочатку всі задачі модифікуються згідно з наступними правилами: обмеження типу «≤» зводяться до рівності з додаванням додаткової змінної з коефіцієнтом +1. Обмеження типу «≥» доповнюються змінною з коефіцієнтом -1, так як негативні змінні не можуть використовуватися в початковому базисі, додається ще одна, допоміжна змінна. Допоміжні змінні завжди додаються з коефіцієнтом +1. Обмеження типу «=» доповнюються однією допоміжною змінною.

   Допоміжні і додаткові змінні створюються штучно для побудови початкового базису, але їх роль різна. Додаткові змінні означають, що відповідне їм обмеження недовикористане, якщо значення додаткової змінної дорівнює нулю, це означає, що виконана рівність правої і лівої частини, якщо значення більше нуля, це означає, що рішення може бути, але необов’язково є неоптимальним. Додатнє значення допоміжної змінної означає недопустимість рішення.

Після цього проводиться симплекс-метод відносно цільової функції. Так як всі додаткові змінні збільшують значення функції *y,* в процесі алгоритму вони будуть почергово виводитись з базису, при цьому після кожного переходу нове рішення буде все ближче до множини допустимих рішень.

Отже, в даній задачі для пошуку оптимального маршруту необхідно мати такі дані**:**

- Costs – ціни на кожній заправці, що зустрічаються на маршруті;
- Volumes – об’єм пального, необхідний для подолання відстані між двома сусідніми заправними станціями;
- Tank – об’єм баку транспортного засобу;
- Remainder – залишок палива на початку маршруту;

Нехай Plan – масив, який зберігає дані про об’єм придбаного палива на кожній заправній станції, тоді список обмежень для симплекс-методу:
де, n – кількість заправних станцій на маршруті

Обмеження у вигляді нерівностей переписуються в обмеження рівності за допомогою фіктивних параметрів і далі вирішуються стандартно для симплекс-методу.

Оскільки даний підхід пропонує оптимальний план, який не є єдиним, наприклад, враховуючи, що ціни на кожній АЗС у межі однієї країни однакові, то оптимальний план, це будь-який план, який має найкоротшу пройдену відстань, тому множина оптимальних розв’язків може мати не один, а декілька варіантів. Це спричинено тим, що при однаковій ціні на декількох АЗС немає значення де і скільки запритись, тому дозаправка відбувається на кожній АЗС на невеликий об’єм палива, щоб вирішити цю проблему алгоритм було доповнено методом спрощення маршруту Simplify**.**

Ідея методу Simplify полягає в тому, що якщо ціна на поточній заправній станції дорівнює ціні на наступній заправній станції і в баку достатньо вільного об’єму, то додатково заправляємся на поточній станції і пропускаємо наступну.

Сервіс оптимізації реалізовано у вигляді Web API мовою програмування C# на платформі .NET Core.

6. РЕЗУЛЬТАТИ ТА ЇХ АНАЛІЗ

Згідно з описаною вище архітектурою та спроектованими мікросервісами було реалізовано веб-застосування, де можна використати описані функції.

Структура проекту виглядає так:

Рисунок 14 – Структура проекту

ngnix – директорія, що містить налаштування та шаблони для веб-серверу.

data – директорія, що зберігає дані з баз даних. Вона потрібна тому що, сервіс баз даних запускається у внутрішньому середовищі Docker і дані зберігаються там, так як після змін у застосуванні треба перерозгортати проект, для цього треба зупиняти всі сервіси та при великих змінах створювати всі контейнери з початку, в такому випадку дані не збережуться, тому щоб не втрачати дані було вирішено створити локальну директорію, яка буде закріплена до контейнеру і там будуть зберігатися дані навіть після оновлення всіх сервісів.

src – директорія, що містить код усіх мікросервісів, а саме:

- FO.FuelOptimization – Web-API сервіс оптимізації;
- FO.FuelStations – Web-API сервіс, який по заданому маршруту будує ізолінію, шукає АЗС та знаходить відстань між точками;
- FO.FuelStationsCore – Collect-API сервіс, що оновлює ціни на АЗС;
- FO.TransportationManagement – Web-API сервіс, що приймає дані введенні користувачем, зберігає їх, та запускає сервіс оптимізації;
- FO.UI – сервіс користувацького інтерфейсу;

docker-compose.yml, docker-compose.dev.yml, docker-compose.prod.yml – файли конфігурації Docker, що містять налаштування для різних варіантів застосування.

docker-compose.yml – файл конфігурації, що містить спільні налаштування для версії для розробки та для розгорання, це такі налаштування, як ключи до сторонніх API, директорії, що містять код та інше. docker-compose.dev.yml, docker-compose.prod.yml – файли конфігурації, для розробки та для розгортання відповідно. Вони містять налаштування, що відрізняються для кожної збірки, це порти, деякі файли та налаштування середовища запуску.

foscript.sh – Bash-скрипт, що використовується для автоматичної збірки проекту у різних режимах. Скрипт створює необхідні директорії, для правильної роботи застосування на новому робочому місці або сервері, а також запускає застосування у потрібній конфігурації:

- dev – режим для розробки, він підтримує hot-reloading (компіляція та оновлення програми, при змінні відповідних файлів), а також відображає всі помилки у режимі розробника і зберігає логи подій;
- prod – режим розгортання, він компілює і запускає застосування у режимі розгортання, використовуючи оптимізовані збірки проектів, а також зберігає, але не відображає помилки, які трапилися при роботі користувача зі застосуванням;

- ui – режим розробки користувацького інтерфейсу, в цьому режимі запускається лише сервіс користувацького інтерфейсу, при цьому сервіси, що використовуються користувацьким інтерфейсом mocked, тобто вони мають «заглушки» у вигляді статичних даних або файлів;

mycustom.cnf – файл конфігурації баз даних, необхідний для підтримки базами даних символів формату Unicode.

Для тестування було створено маршрут, у якому обрано легковий ТЗ Ford Mondeo. Характеристики автомобіля:

- витрати палива на 100км – 5л;
- паливний бак – 63л;
- об'єм палива на початку маршруту – 10л;

Маршрут складається з шести точок: Львів ➜ Краків ➜ Берлін ➜ Штутгарт ➜ Турин ➜ Барселона. Загальна довжина маршруту 3055км. Очікувана кількість пального:

Fuel Cost= 3055 * (5 / 100) = 152.75 (літрів)

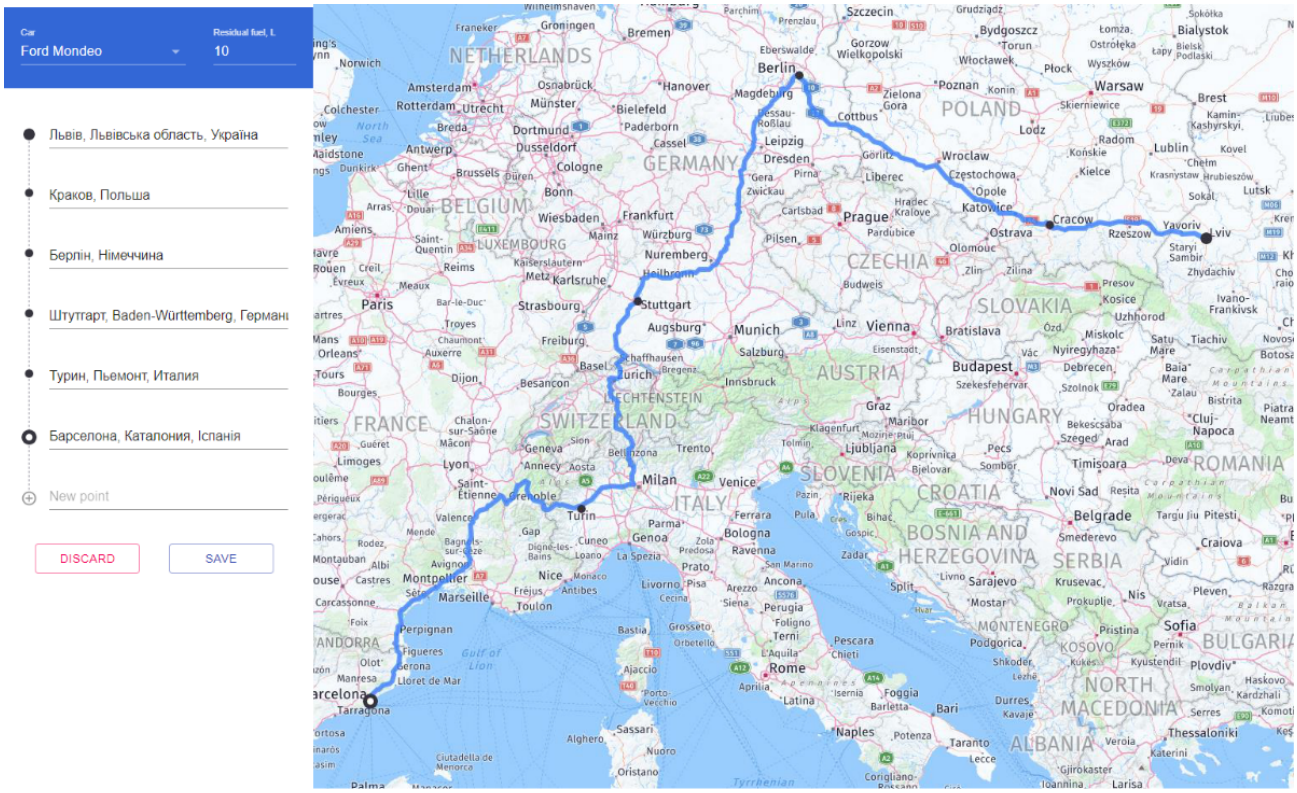

Рисунок 15 – інтерфейс створення нового маршруту

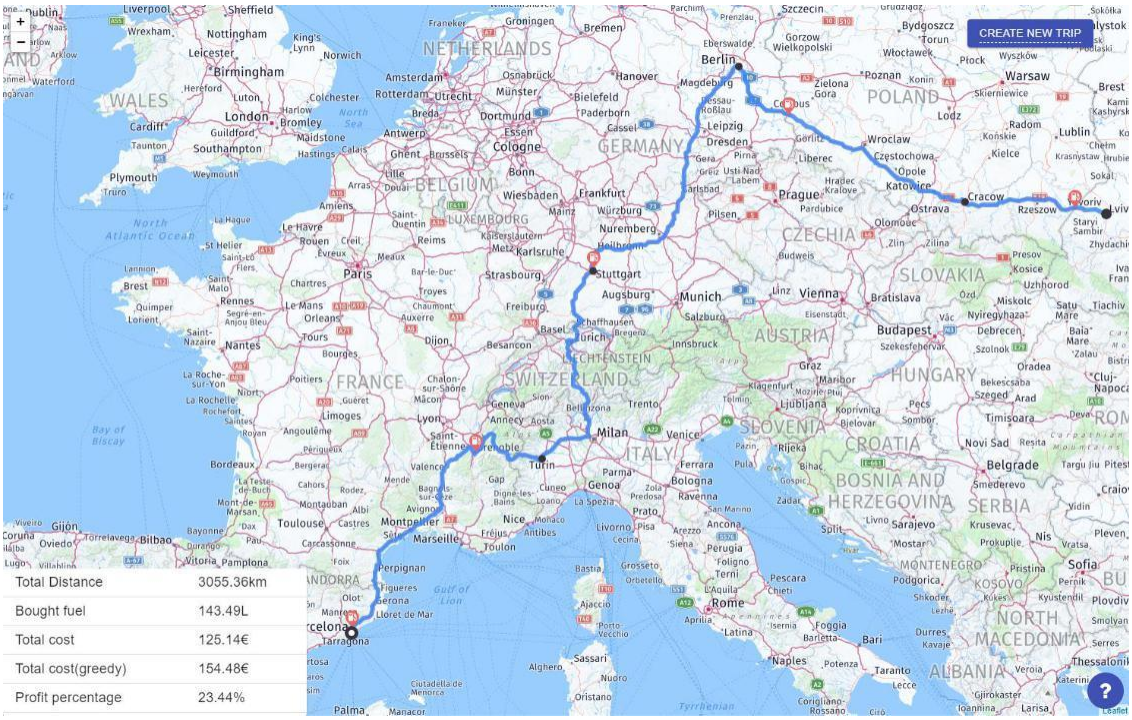

Рисунок 16 – інтерфейс оптимізованого маршруту

Результат розрахунку програми - 143 літри. Різниця майже на 10 літрів пояснюється тим, що на старті подорожі у баці було 10л, які вже були придбані. Тобто розрахунок кількості пального правильний.

Total Distance	3055.36km
Bought fuel	143.49L
Total cost	125.14€
Total cost(greedy)	154.48€
Profit percentage	23.44%

Рисунок 17 – коротка інформація про маршрут отримана з сайту

В загальному було зроблено 5 дозаправок:

Країна	Кількість пального(Л)	Ціна(євро)	Назва АЗС	Вартість(євро)
Україна	57.422	0.737	WOG	43.32,
Польща	34.474	0.765	BP	26.37
Німеччина	39.494	1.07	Avia	42.26
Франція	11.279	1.184	Avia	13.35
Іспанія	0.825	1.016	Cepsa	0.84
Загалом	143.49			125.14

Таблиця 1 – результат розрахунку маршруту

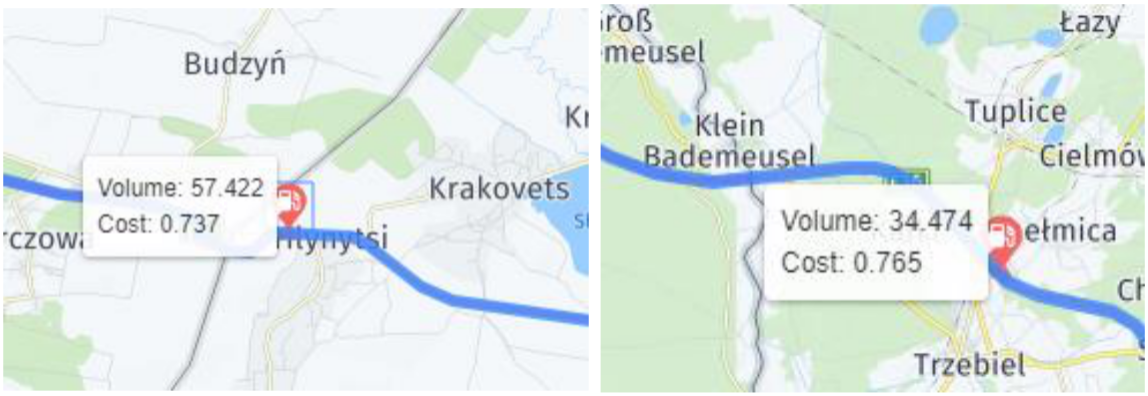

Рисунок 18 – місця заправки перед переїздом за кордон

Як можна побачити з зображень вище, при переїзді з однієї країни до іншої, де ціна на пальне вище, алгоритм заправляє машину на максимально

можливу або потрібну кількість пального, щоб їхати країною з високою ціною не заправляючись(перше зображення – перетин кордону з України до Польщі, друге зображення – перетин кордону з Польщі до Німеччини.

Рисунок 19 – місця заправки у Франції та Іспанії

Як можна побачити з зображень вище, у Франції та Іспанії було придбано невеликий об’єм пального всього 11.28 та 0.8 літрів відповідно. Це зумовлено тим, що після Німеччини маршрут проходить такі країни як Швейцарія, Італія та Франція, де ціни на пальне вище ніж у Німеччині, тому при виїзді з Німеччини було заправлено бак на максимальний об’єм, якого вистачило проїхати до Франції, де потрібна дозаправка. У Франції ціна на пальне вище ніж у Іспанії, тому було придбано лише 11.28 літри, чого вистачає щоб доїхати до найближчої АЗС у Іспанії, де було придбано всього 0.8 літри, чого вистачило, щоб доїхати до кінцевої точки – Барселони, з таким планом заправок побудований маршрут є оптимальним.

Також було реалізовано жадібний алгоритм для порівняння. Принцип роботи жадібного алгоритму – заправлятися завжди на мінімальну кількість пального, якого вистачить щоб доїхати до наступної АЗС або кінцевої точки. Як можна побачити такий принцип неоптимальний, маршрут побудований таким способом має загальну вартість 154.48 євро, оптимізований маршрут дешевше на 29.34 євро (23.45%).

Також було побудовано тестовий маршрут для вантажного ТЗ. У вантажних ТЗ більше бак і більший витрати палива, тому і оптимізація може мати більш суттєве значення. Вибраний вантажний ТЗ – DAF XF 105 з характеристиками:

- витрати палива на 100км – 22л;
- паливний бак – 400л;
- об'єм палива на початку маршруту – 50л;

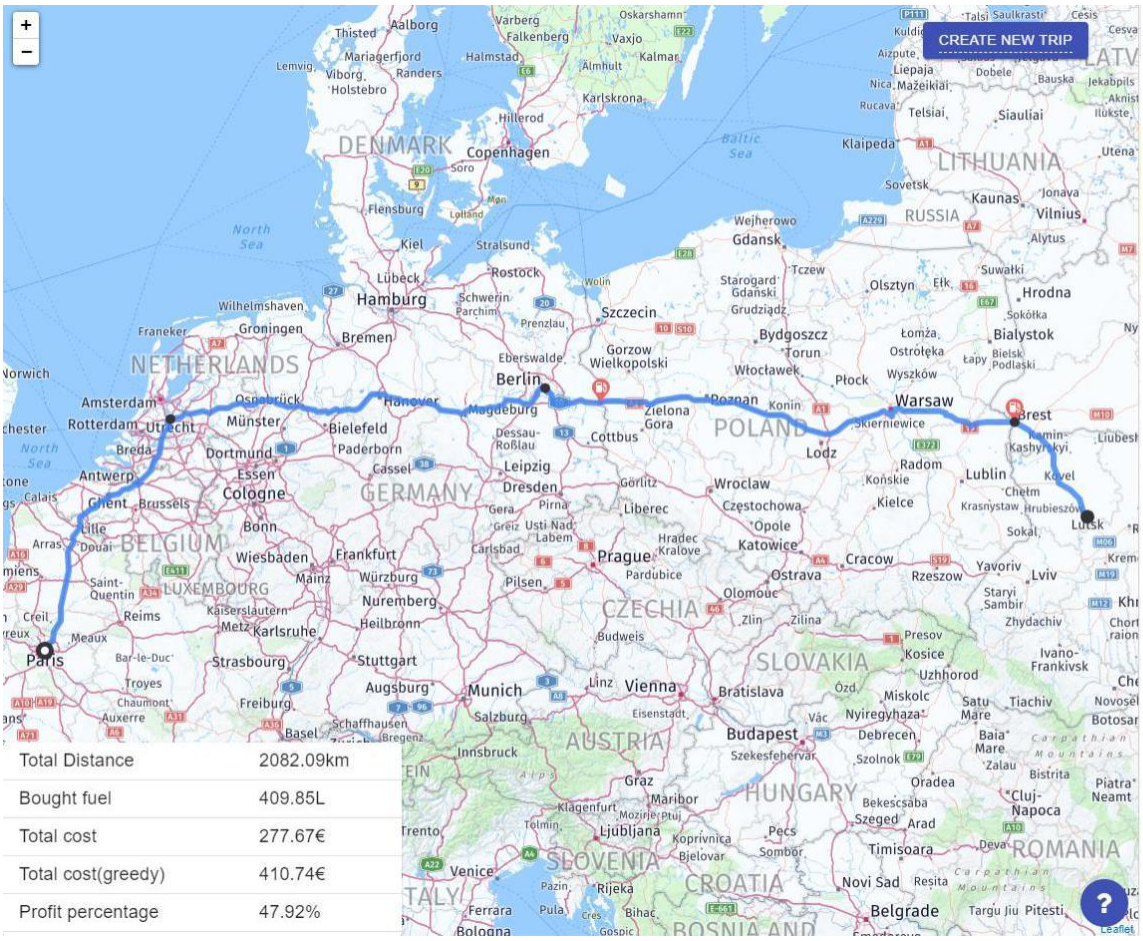

Рисунок 20 – результат оптимізованого маршруту вантажного ТЗ

Маршрут складається з п’яти точок: Луцьк ➜ Брест ➜ Берлін ➜ Утрехт ➜ Париж. Загальна довжина маршруту 2080км. Придбано пального – 409.85 літрів.

В загальному було зроблено 2 заправки:

Країна	Кількість пального(Л)	Ціна(євро)	Назва АЗС	Вартість(євро)
Білорусь	394.14	0.674	Белоруснефть	265.65
Польща	15.71	0.836	Orlen	13.13
Загалом	409.85			277.67

Таблиця 2 – результат розрахунку маршруту

Згідно плану, щоб доїхати з Луцьку до Бресту було достатньо 50 літрів, що були спочатку, далі було заправлено 394 літрів до повного баку(6 літрів залишилося зі старту). І лише, коли було витрачено деяку кількість пального було придбано ще 16 літрів у Польщі, де ціни на паливо найменші серед країн, які зустрічаються на шляху і цього палива було достатньо щоб доїхати до Парижу.

Маршрут побудований жадібним алгоритмом має загальну вартість 410.74 євро, що більше оптимального маршруту на 133.07 євро, що складає 47.92% від вартості.

Маршрути доступні за посиланнями:

Львів➜Барселона: gradwork.site/trip/fb31a4c4-8b08-40ef-8ef5-adf86bbd6359

Луцьк➜Париж: gradwork.site/trip/019e4c28-d682-457f-ad63-9f3fb0accc6e

7. ВИСНОВКИ

В ході виконання дипломної роботи було проаналізовано програмні рішення для побудови маршруту та пошуку АЗС, враховано знайдені недоліки та підходи при проектуванні, на основі чого, було створено прототип веб-застосування, що вирішує поставлену задачу. Вихідний код розробленого застосування опублікований на публічному ресурсі Gitlab[16].

Виявлено, що поширені рішення не мають можливості підтримувати різні види ТЗ, не мають інформації про АЗС і не підтримують гнучкість при побудові маршруту.

Було розглянуто різні технології для реалізації прототипу застосування та вибрано найбільш відповідні для даної задачі.

Було розглянуто різні можливі алгоритми вирішення задачі оптимізації, порівняно їх за визначеними критеріями (важкість реалізації, час виконання оптимізація залежно від вхідних даних, використання пам’яті). Було вибрано і реалізовано найбільш оптимальний алгоритм.

Було розроблено прототип програми і проведено тестування на декількох маршрутах різної довжини і з різними типами ТЗ. Отримані результати демонструють значне зменшення вартості маршруту симплекс методом відносно інших алгоритмів.

Було проведено виступ виступ на 16 міжнародній конференції електроніки та прикладної фізики в розділі Математичні проблеми прикладної фізики та комп'ютерних технологій[19] (Додаток Г).

8. СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

[1] Modal split of freight transport [Електронний ресурс]. – Режим доступу до ресурсу: https://ec.europa.eu/eurostat/databrowser/view/t2020_rk320/default/table.

[2] Modal split of passenger transport [Електронний ресурс] – Режим доступу до ресурсу: https://ec.europa.eu/eurostat/databrowser/view/t2020_rk310/default/table.

[3] Google Maps About [Електронний ресурс] – Режим доступу до ресурсу: https://www.google.com/maps/about/.

[4] The UTA Station Finder: Service station information available online. [Електронний ресурс] – Режим доступу до ресурсу: https://www.uta.com/tankkarte/tindex/en_Stationfinder-petrolstation.htm.

[5] - OptimoRoute: Key Features - [Електронний ресурс] Режим доступу до ресурсу: https://optimoroute.com/features/

[6] DigitialOcean Droplets [Електронний ресурс] – Режим доступу до ресурсу: https://www.digitalocean.com/products/droplets/.

[7] Docker What is Container [Електронний ресурс] – Режим доступу до ресурсу: https://www.docker.com/resources/what-container

[8] React Js Documentation and Tutorial [Електронний ресурс] – Режим доступу до ресурсу: https://reactjs.org/docs/getting-started.html

[9] .NET Core A developer platform for build all your [Електронний ресурс] – Режим доступу до ресурсу: https://dotnet.microsoft.com/learn/dotnet/hello-world-tutorial/intro

[10] Entity Framework Core Overview [Електронний ресурс] – Режим доступу до ресурсу: https://docs.microsoft.com/en-us/ef/core/

[11] Low memory MySQL [Електронний ресурс] – Режим доступу до ресурсу:

https://hub.docker.com/r/akoller/low-memory-mysql/

[12] HERE REST APIs [Електронний ресурс] – Режим доступу до ресурсу: https://developer.here.com/develop/rest-api

[13] CollectAPI Gas Prices API [Електронний ресурс] – Режим доступу до ресурсу: https://collectapi.com/api/gasPrice/gas-prices-api

[14] - Implementing Domain-Driven Design / Vaughn Vernon - Addison-Wesley Professional - 2013

[15] What Is Mathematics?: An Elementary Approach to Ideas and Methods / C.Robbins, R. Robbins, H. Robbins, S. Ian. – New York: Oxford University Press, 1996. – 344 с.

[16] Turf.js Advanced geospatial [Електронний ресурс] – Режим доступу до ресурсу: https://turfjs.org/

[17] Хемді А. Таха. Глава 3. Симплекс-метод // Введення в дослідження операцій Operations Research: An Introduction. — 7-е видання. — М.: «Вільямс», 2007

[18] Репозиторій з вихідним кодом програми [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/GlebGlushko/fuel_optimization

[19] Simplex method implementation for optimization in logistics tasks / Hlib Hlushko, Dmytro Ivanenko, Dec 2020.

ДОДАТКИ

Додаток А

Код програми

```
using Accord.Math;
using System.Collections.Generic;
namespace Fleet.FuelOptimization.Services {
public class Optimization : IOptimization
{
double[] costs;//ціна палива в і-й країні
double[] volumes;//об'єм палива, що потрібен для доїзду від країни і до
кордону країни і+1
int tank;//об'єм баку автовоза - обов'язковий залишок int remainder;//залишок
палива на початок маршруту
int minimum_remainder;//мінімальний залишок палива у баку
public double[,] Table
{
get
{
int points_number = costs.Length;
double[,] table = new double[2 * points_number + 1, points_number +
1];
table[0, 0] = -volumes[0] - minimum_remainder + remainder; table[0, 1] = -1;
for (int i = 1; i < points_number; i++)

{
table[i, 0] = table[i - 1, 0] - volumes[i];
for (int j = 1; j <= i + 1; j++)
```

```
{
table[i, j] = -1;
}
}
table[points_number, 0] = tank - remainder + minimum_remainder;
table[points_number, 1] = 1;
for (int i = points_number + 1; i < 2 * points_number; i++)
{
table[i, 0] = table[i - 1, 0] + volumes[i - points_number - 1]; for (int j = 1; j <= i +
1 - points_number; j++) {
table[i, j] = 1;
}
}
for (int j = 0; j < points_number; j++)
{
table[2 * points_number, j + 1] = costs[j];
}
return table;
}
}//вхідні дані для симплекс-таблиці (обмеження, цільова функція) з
урахуванням залишку на початок і мінімального необхідного залишку
public double[] Plan()//об'єм купленого палива в і-й країні
{

int points_number = costs.Length;
double[] result = new double[points_number]; double[,] table_result;
Simplex S = new Simplex(Table);
table_result = S.Calculate(result);
//return result;
```

```
return Simplify1(result);
}
public Optimization()
{
costs = new double[] { 31 };
tank = 600;
volumes = new double[] { 0 };
}
public void Init(double[] Costs, double[] Volumes, int Tank, int Remainder, int
Minimum_remainder)
{
remainder = Remainder;
minimum_remainder = Minimum_remainder; costs = Costs;
tank = Tank - Minimum_remainder;
volumes = Volumes;
}
double[] Simplify(double[] plane)
{
bool change = true;
int n = plane.Length;
48
double[] simple = new double[n];
for (int i = 0; i < n; i++)
{
simple[i] = plane[i];
}
double[] volumeremainders = new double[n];//кількість палива в баку у
точці і
volumeremainders[0] = remainder - minimum_remainder + plane[0];
```

```
for (int i = 1; i < n; i++)
{
volumeremainders[i] = volumeremainders[i - 1] + plane[i] - volumes[i -
1];
//Console.WriteLine(volumeremainders[i]);
}
int j = 0;
while (change && j < n)
{
change = false;
for (int i = j; i < n - 1; i++)
{
if (volumeremainders[i] + simple[i + 1] < tank)
{
if (simple[i + 1] != 0 && costs[i + 1] == costs[i])
{
simple[i] += simple[i + 1]; volumeremainders[i] += simple[i + 1];
volumeremainders[i + 1] -= simple[i + 1];
49
simple[i + 1] = 0;
change = true;
}
}
else { j++; }
}
}
return simple;
}
double[,] Standardization(double[,] M)
```

```
{
double[,] StandardSours = Matrix.Copy(M);
int changeVarRow = findMainNegativeRow(StandardSours); int changeVarCol;
while (changeVarRow != -1)
{
changeVarCol = findMainNegativeCol(changeVarRow, StandardSours);
basis[changeVarRow] = changeVarCol;
StandardSours = RewriteSimplexTable(changeVarRow, changeVarCol,
StandardSours);
changeVarRow = findMainNegativeRow(StandardSours);
}
return StandardSours;
}
private int findMainNegativeRow(double[,] M)
{
50
int mainRow = 0;
int rows = M.GetLength(0);
for (int i = 1; i < rows - 2; i++)
if (M[mainRow, 0] > M[i, 0])
{
mainRow = i;
}
if (M[mainRow, 0] >= 0) return -1;
return mainRow;
}
```

Додаток Б

```
// псевдокод агрегату маршруту
public class Route : Aggregate
{
   // Тут поля приватні, вони можуть оновлюватись лише через публічні методи
   private readonly Points _destinations;
   private Distances _distances;
   // Кожен агрегат містить список домених івентів, які генеруються у відповідний час
   // На ці доменні івенти можуть підписатися різні івент хендлери
   // і залежно від івенту ініцювати наступний крок алгоритму
   private List<DomainEvent> _events = new List<DomainEvent>;
   // конструктор агрегату
   public Route(Points destinations){
     _destinations = destinations;
   }
   // окрім конструктору агрегату необхідно реалізувати можливість
   // відтворити доменну модель зі збереної інформації в базі данних
   public Route FromDataAccessLayer(RouteDal dal){
     return new Route(dal);
   }
   // Деякі поля агрегату можуть бути корисні в інших зовнішніх сервісах
   // тому до них є публічний доступ через властивості,
   // при чому властивості доступні на читання, а не на запис
   public Destinations Destinations { get { return _destinations; }; private set; }
   public Distances Distances { get { return _distances; }; private set; }
   // Агрегат містить інкапсулює в собі методи, які описують дії які з ним можуть відбуватися
```

```
    public void CalculateDistances()
    {
      var distances = await
_external_routing_service.CalculateDistancesAsync(_destinations);
      _distances = distances;
      // при необхідності, при змінні агрегатів можуть генеруватися доменні
івенти
      _destinations.Add(new DistancesCalculatedDomainEvent(distances));
    }
    public void FindGasStations()
    {
       var gasStations = await
_external_routing_service.FindGasStationsAsync(_destinations);
       _distances = distances;
      _destinations.Add(new
RouteOptimizationCompletedDomainEvent(distances));
    }
    public void GetSummary()
    {
       return this.ToSummary();
    }
}
```

Додаток В

```
// псевдокод обробника доменних івентів,
// на кожен івент може бути безліч обробників
public class EventHandler<DomainEvent>{
    // при отриманні івенту, що відстані між точками було прораховано
    // дістаємо маршрут із бази даних і ініціюємо пошук заправних станцій
    public void HandleDistancesCalculated(DistancesCalculatedDomainEvent
@event)
    {
        var route = _routing_managment_service.GetRoute(@event.RouteId);
        route.FindGasStations();
    }
    // при отриманні івенту, що оптимізація маршруту завершена
    // отримуємо маршрут, формуємо загальну інформацію і відправляємо
користувачу
    public void
HandleRouteOptimizationCompleted(RouteOptimizationCompletedDomainEvent
@event)
    {
        var route = _routing_managment_service.GetRoute(@event.RouteId);
        var result = route.GetSummary();
        _message_bus.SendOptimizedRoute(result);
    }
}
```

Додаток Г

SIMPLEX METHOD IMPLEMENTATION FOR OPTIMIZATION IN LOGISTICS TASKS

Hlib Hlushko*, Dmytro Ivanenko**
Taras Shevchenko National University of Kyiv,
Faculty of Radiophysics, Electronics and Computer Systems,
64/13, Volodymyrska Street, Kyiv city, Ukraine 01601
*email: gleb.glushko10@gmail.com **email: divanenko1979@gmail.com

Abstract: This work proposes using a simplex method algorithm to optimize fuel consumption of different types of vehicles by defining constraints of a vehicle and a route. The simplex method algorithm allows us to add new constraints easily and adapt to new requirements with fewer alterations.

Introduction

Automatization of building a route plan is an important task and the main problem of building an optimal route is taking into consideration a whole bunch of parameters, like the weight of a cargo, loading/unloading cargo on the way, fuel prices, tax refunds, vehicle specifications, and others. Nowadays most planning is done by people, who cannot not only consider all factors properly but also are expensive workers. This problem can be solved by building an objective function and minimizing its value considering given requirements and constraints.

Factors that affect the route plan

There is an enormous number of factors that affect the optimal route plan. The list of these factors is displayed below, but it can be easily expanded or shrunk. Factors considered in this work: fuel prices, cargo weight, loading and unloading cargo on the way, vehicle specifications: the size of the vehicle, fuel consumption, tank volume, heating system; driver license and permissions to drive in different

countries, tax refunds, remained fuel, road limits, obligatory of the driver to rest, toll cost.

Simplex method algorithm

The simplex algorithm operates on linear programs in the canonical form[1]:

minimize $cTx = z$

subject $Ax \leq b$ and $x \geq 0$

Here A is an $m \times n$ matrix whose jth column is aj. This matrix corresponds to the coefficients on x1, x2 … xn in the constraints of a linear programming problem. The vector x is a vector of solutions to the problem, b is the right-hand-side vector, and c is the cost coefficient vector.

This representation of the algorithm can be transformed to meet our problem[2]. The coefficients of the objective function would be fuel price at each fuel station, variables of the problem would be the amount of fuel bought at each station and nonnegative constants would be constraints that bought amount of fuel is less or equal to tank volume, remained fuel at each moment is greater than 0 and less than tank volume. Other factors could be taken into account while computing distances(e.g. vehicle size, road limits, etc.), fuel consumption(e.g. heating system, the weight of the cargo), fuel cost(e.g. tax refunds, toll cost, etc.), and others intermediate calculations that affect either costs or the refueling plan.

This way we can build an objective function, which looks like:

$L(costs, volumes) = \sum costs[i] * plan[i]$

with n – number of fuel stations,

cost[i] – fuel price at the i fuel station,

plan[i] – bought amount of fuel at the i fuel station.

Results

The proposed algorithm is implemented and shows the reduction of expenses up to 25% compared to the manually planned route. Even though it shows great results in theory, in reality, the expenses differ from the cost predicted by the

algorithm due to lack of data, human factor, and other minor inaccuracies. Although the algorithm doesn't provide an accurate cost of the route plan it performs better than people who build and estimate route manually or following simple instructions, the algorithm can provide an explanation for the selected route and refueling plan, and on average can be used by drivers themselves without additional assistants like transportation managers.

References:

Catherine Lewis "Linear Programming: Theory and Applications" pages 8-9, May, 2008;

Sandra Eriksson Barman "Modeling and solving vehicle routing problems with many available vehicle types" pages 8-12, Gothenburg, Sweden, 2014;