Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет _____Комп'ютерних наук_____
(повна назва)

Кафедра _____Програмної інженерії_____
(повна назва)

АТЕСТАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти – другий (магістерський)

Дослідження методів планування перевезень за критерієм кількісті шкідливих викидів
(тема)

Виконав: студент 2 курсу, групи ІПЗм-18-4
_____Руденко Д.Б._____
(прізвище, ініціали)
спеціальності 121- Інженерія програмного _ .
забезпечення_____
(код і повна назва спеціальності)
Освітньо-наукової програми Інженерія_
програмного забезпечення _____
(прізвище, ініціали)
Керівник _____доц. Чуприна А.С._____
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри, проф. _____ З.В.Дудар

2020 p.

ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ РАДІОЕЛЕКТРОНІКИ

Факультет_____Комп'ютерних наук_____

Кафедра_____Програмної інженерії_____

Рівень вищої освіти – другий (магістерський)

Спеціальність _____121- Інженерія програмного забезпечення_____
(код і повна назва)

Тип програми освітньо-наукова програма

Освітньо програма ____Інженерія програмного забезпечення_____
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри_____
(підпис)
«____»_____ 20 ___ р.

ЗАВДАННЯ
НА АТЕСТАЦІЙНУ РОБОТУ

студентові _____Руденко Денису Борисовичу_____
(прізвище, ім'я, по батькові)

1. Тема роботи _Дослідження методів планування перевезень за критерієм кількісті шкідливих викидів_____
затверджена наказом по університету від "___" _____20 ___ р № _____
заповнюється вручну після отримання наказу
2. Термін подання студентом роботи до екзаменаційної комісії
_____ 2020 р.
3. Вихідні дані до роботи *моделі, методи та існуючи алгоритми теорії графів для планування перевезень, знаходження найкоротшого шляху, середовище об'єктно-орієнтованого проектування Visual Studio 2019*

4. Перелік питань, що потрібно опрацювати в роботі *мета роботи, аналіз проблемної галузі і постановка задачі, огляд методів машинного зору, використання машинного навчання і розпізнавання образів, реалізація програмної системи*_____
_____

5. Консультанти розділів роботи

Найменування розділу	Консультант (посада, прізвище, ім’я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата
Спецчастина	доц. Чуприна А.С.		

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Аналіз проблемної галузі	27.01.2020	
2	Розробка моделі предметної галузі	9.02.2020	
3	Розробка структури зберігання даних	02.03.2020	
4	Створення коду програми	16.03.2020	
5	Тестування і налагодження програми	3.04.2020	
6	Підготовка пояснювальної записки	20.04.2020	
7	Підготовка презентації та доповіді	28.04.2020	
8	Нормоконтроль, рецензування	9.05.2020	
9	Занесення диплома в електронний архів	9.05.2020	
10	Попередній захист	9.05.2020	
11	Допуск до захисту у зав. кафедри	19.05.2020	

Дата видачі завдання _____*2020 р*.

Студент _____
(підпис)

Керівник роботи _____ доц. Чуприна А.С._____
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка до атестаційної роботи: 62 с., 23 рис., 2 додатки, 20 джерел.

ПРОГРАМНА СИСТЕМА, ПРОГРАМНИЙ ПРОДУКТ, ТЕОРІЯ ГРАФІВ, НАВІГАЦІЯ, ЗНАХОДЖЕННЯ ОПТИМАЛЬНОГО МАРШРУТУ

Об’єкт дослідження – методи будування маршрутів перевезень з мінімальною кількістю шкідливих викидів.

Мета роботи – дослідити основні методи, технології та алгоритми для будування ршрутів перевезень з мінімальною кількістю шкідливих викидів, розробити програму на основі проведених досліджень.

В роботі проведений аналіз предметної галузі, досліджені основні методи роботи з графами, проведений аналіз актуальних технологій.

SOFTWARE SYSTEM, SOFTWARE PRODUCT, GRAPH THEORY, NAVIGATION, FINDING THE OPTIMAL ROUTE

The Object is an investigation of methods of construction of transportation routes with the minimum amount of harmful emissions.

The purpose of the work is to investigate the basic methods, technologies and algorithms for the construction of transport routes with the minimum amount of harmful emissions, to develop the program on the basis of the conducted researches.

The article analyzes the subject area, investigates the main methods of graph usage, conducted the analysis of current technologies.

ЗМІСТ

ВСТУП

Логістика завжди мала велику роль у житті людства. Одною з найважливіших задач сьогодення – це зменшення викидів шкідливих речовин до атмосфери при експлуатації транспортних засобів. Над вирішенням даної проблеми працюють фахівці усіх провідних країн світу.

Розвиток нових технологій стає все важче і важче без врахування впливу на екологію. Із зростанням обсягів перевезень ми все більш чуємо про шкоду, яку людство наносить Землі, як локально, так і глобально.

Знаходження маршруту з найменшою кількістю шкідливих викидів – це комплексна задача, яка відноситься до задач комбінаторної оптимізації, яка ставить своєю ціллю знаходження кращого шляху не за критеріями найкоротшої дистанції, як наприклад у задачі комівояжера, а за можливістю знаходження шляху пересування з можливістю створення найменшої кількості викидів шкідливих речовин. При цьому потрібно враховувати розподіл пересування транспорту за часом, щоб уникати заторів у місті, простоїв кораблів у морі та підвищення льотного часу літака через брак пропускних можливостей аеропортів.

З цього можна зробити висновок, що контроль викидів потребує аналізу не тільки факторів витрати пального при переміщенні, я ще й аналізу непрямих факторів.

Тому метою дослідження є можливості використання існуючих алгоритмів теорії графів для будування оптимального маршруту з погляду екології.

Дану проблему можна вирішувати на різних рівнях та зазираючи на різні існуючи галузі. Сьогодні існує багато різних видів транспорту, які використовуються для перевезень як вантажів, так і пасажирів. Можна виділити наступні основні види транспорту: морський, авіа, залізничний, автомобільний.

Для кожної з категорій можна виділити особистий підхід створення оптимального маршруту. Та умовно можна виділити види існуючих проблем забруднення. Забруднення в умовах перевезень на великі відстані та забруднення в умовах міста.

Ще з вісімдесятих років 20-ого сторіччя традиційним підходом збереження даних є реляційна модель. Вона сприймається користувачем як набір нормалізованих відношень різного ступеню. Інформація у таких базах структуровано зберігається у вигляді таблиць з наперед визначеними колонками конкретних типів. Розробникам доводиться створювати строгу структуру даних. Швидкі темпи збільшення обсягу інформації вимагають нових підходів до вирішення проблеми її збереження.

Часом вдається модернізувати вже наявні рішення, але інколи потрібні нові засоби, які принципово відмінні від попередників. Більшою мірою мають попит продукти, які використовують реляційну базу даних, але як альтернатива баз даних SQL десь з початку 2000-х розвивається напрямок NoSQL.

Останні розділяють на декілька типів, залежно від їх масштабування, систем збереження даних, моделей даних та запитів. До єдиної класифікації не дійшли, тому виокремимо основні: ключ/значення, документоорієнтовані, стовпчиково-орієнтовані та графові.

Робота з графами потребує специфічного підходу, тому при розробці системи, яка використовує графи потрібно дослідити який підхід збереження графу більш усього підходе для результуючої системи.

Таким чином під час розробки схеми бази даних необхідно допускати різні підходи проектування. Можливо у деяких випадках має сенс застосування графових схем і це може істотно скоротити доступ до даних, а на великих обсягах значно зменшити час обробки інформації.

Все вищесказане визначило актуальність теми роботи - використання теорії графів для будування маршрутів перевезень з мінімальною кількістю шкідливих викидів.

Отже, тема та засоби для виконання роботи є актуальними. Метою роботи є дослідження основних методів, технологій та алгоритмів для будування маршрутів перевезень з мінімальною кількістю шкідливих викидів.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Історія та розвиток перевезень

Перевезення вантажів – одна з найдавніших тем, з якою люди зустрічалися на протязі усієї історії. Можна стверджувати, що без перевезень прогрес людства був би значно повільнішим, ніж він є зараз. Можна багато говорити про те, як змінив наше життя інтернет, але яке значення інтернету порівнюючи з залізницею? Залізниця – це саме те, що стало тригером індустріальної революції. Там можна сміливо стверджувати, що без залізниці не було би не тільки інтернету та й інших галузей таких як: металургія, горна справа, енергетика та інші. І також саме порівняння можна зробити між залізницею та конями. Якщо не було би коней, то людство ще довго би не могло дійти до створення залізниці. Прикладом є корені цивілізації двох Америк. Саме те що, в інків та ацтеків не було достатньо великої в'ючної тварини, не надало можливості розвиватися у тому самому темпі як люди зі Старого світу.

Транспорт часто асоціюється з економічним розвитком. Підключення міста чи регіону до основних транспортних мереж може дати початковий імпульс місцевій економіці та створити нові робочі місця. Однак, як тільки регіон досяг певного рівня зв'язку, додаткова транспортна інфраструктура не дає порівнянних переваг. Також це може спричинити значний вплив на навколишнє середовище.

Використання транспорту призводить до викидів забруднюючих речовин, які можуть поширюватися поза досяжністю транспортних мереж. Вони можуть сприяти фоновим концентраціям твердих часток, озону та діоксиду азоту, впливаючи на людей, рослини та тварин. Деякі райони, включаючи гірські регіони, прибережні зони та моря, можуть бути особливо вразливими до забруднення транспорту[1].

Так само розливи нафти або викид небезпечних речовин у море можуть завдати значної шкоди морському життю.

Забруднення шумом від транспорту викликає ще одну проблему, і його вплив не обмежується лише наземними екосистемами. Великі кораблі видають значну кількість шуму. Їх корпуси, як правило, підсилюють механічний шум від двигуна і гвинтів. Через низьку частоту цей тип шуму поширюється дуже далеко у воді і порушує морське життя. Дослідження показують, що кити та інші види, які спілкуються та орієнтуються через звук, особливо страждають. Потенційні впливи, які зазнають популяції риб та морських безхребетних, також стають зрозумілішими завдяки постійним дослідженням.

Деякі рішення вже існують та є досить ефективними для зменшення шумового забруднення у морі та на суші. Наприклад, кораблі можуть бути сконструйовані з їх двигунами, розміщеними далі від корпусу (наприклад, електродвигуни в контейнерах поза корпусом), щоб мінімізувати посилення шуму. Аналогічно, двигуни та деталі автомобілів (наприклад, шини) можуть бути перероблені для зменшення рівня шуму на місцевості, або можуть бути розширені шумозахисні смуги вздовж автомобільних доріг.

Сьогодні існує багато програмних рішень, які допомагають, як планувальникам міст так і звичайним людям, побачити актуальну ситуацію на дорозі. Google Maps дозволяє переглядати існуючий трафік при плануванні маршуту та обирати шлях через менш завантажені ділянки. Uber – також використовує аналіз поточного стану для коригування цін та розрахунку часу у дорозі1.2 Існуюче забруднення за видами транспорт

1.2 Огляд існуючих видів транспорту

Розглядаючи різні існуючи види транспорту можна дати коротку характеристики для кожного з них.

Автомобільний транспорт – використовується, як у містах так і для регіонального сполучення. Найшкідливіший вид транспорту у місті.

Міський транспорт – перевозить велику кількість людей, роблячи це дуже ефективно, порівняно з авто транспортом. Кращій вибір транспорту у місті.

Морський транспорт – великі танкери можуть причиняти шкоду як кілька мільйонів автомобілів, при цьому забрудненню підлягає не тільки повітря, а й вода.

Авіа транспорт – дуже шкодить атмосфері через великі витрати пального для здійснення польоту.

Найбільший інтерес має автомобільний транспорт, тому що через велику кількість проблемних ділянок на шляху, можна отримати багато варіантів оптимізації руху для зменшення шкідливих викидів в умовах міста.

1.3 Огляд існуючих систем з аналізу автомобільного трафіку

Сьогодні ведеться активна розробка засобів моніторингу забруднення навколишнього середовища, особливо в умовах міста. Дані, які можна отримати за допомогою такого моніторингу є дуже цінними та потребують відповідної обробки та належного застосування для отримання користі з зібраних даних. Тому окрім збору необхідних показників по забрудненню середовища, актуальним стає питання обробки та застосування отриманих даних.

На даний час існує кілька проектів, які ставлять за ціль оптимізацію пересування в місті. Багато з цих проектів є експериментальними комплексними рішеннями, які націлені на сбір максимального можливої кількості даних для їх подальшої обробки. У більшості випадків існуючи рішення розробляються за підтримки державних закладів та наукових організацій. Та направлені як на рішення існуючих проблем міста, так і на вивчення можливих підходів та збору даних для подальшого аналізу. Тому проблема переміщення в місті стосується не стільки планування переміщення за наявними даними, скільки з отриманням цих даних.

Розглянемо декілька існуючих ресурсів пов’язаних з вивченням забруднення міста(викидами транспорту):

- веб-ресурс www. arrayofthings.github.io;
- веб-ресурс www.511ga.org;
- веб-ресурс www.hal24k.com.

ArrayOfThings (AoT) - це експериментальний міський проект вимірювання, що включає мережу інтерактивних модульних пристроїв або "вузлів", які встановлюються навколо Чикаго для збору даних у реальному часі про місто, інфраструктуру та діяльність міста. Ці вимірювання публікуються як відкриті дані для досліджень та громадського використання. AoT по суті служить "фітнес-трекером" для міста, вимірюючи фактори, що впливають на життєздатність Чикаго, такі як клімат, якість повітря та шум.

Проект фінансувався Національним науковим фондом США, і до кінця 2019 року було встановлено приблизно 130 вузлів у всьому місті, як правило, на вуличних перехрестях, 22-24 фути над вулицею. План проекту передбачає розміщення приблизно 150 вузлів до середини 2020 року, включаючи як заміну старих вузлів (деякі з яких були встановлені ще в 2016 році), так і додавання місць розташування [2].

Технологічна платформа, яка використовується для побудови вузлів, називається Waggle - апаратно-програмною системою з відкритим кодом, розробленою в Національній лабораторії Argonne. Вузли AoT виробляються в Чикаголанді.

ArrayOfThings надає можливість:

- швидко отримувати актуальну інформацію щодо ситуації на окремих ділянках міста;
- переробляти дані в потрібний формат, робити базові розрахунки;
- знаходити оптимальні маршрути пересування містом;
- автоматично знаходити потенційні проблемні ділянки доріг для превентивного усування можливих проблем з перенавантаженням трафіку;

– створювати «макроси», автоматизуючи обробку даних на ділянках;
– 1 велика перевага ArrayOfThings - його все об’ємність. Розроблена та запроваджена система дозволяє отримувати повне уявлення.

Схема розташування трекерів ArrayOfThings представлено на рисунку 1.1.

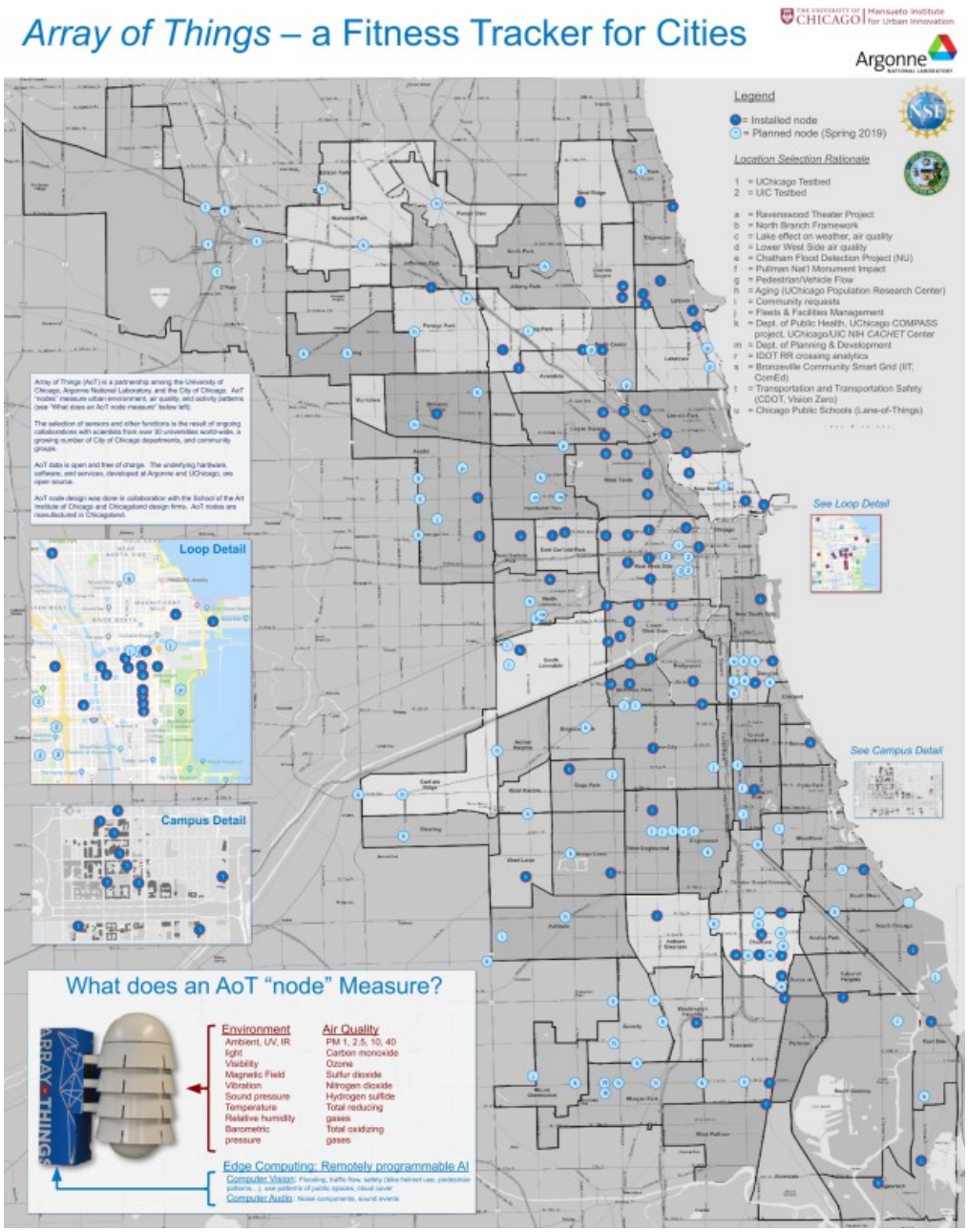


Рисунок 1.1 – Робота додатку OpenRefine

З основних недоліків можна виділити недолік кастомізації вхідних даних та подальшого використання за власними правилами для таблиць.

GDOT - це система аналізу та відображення трафіку. У ньому представлено стан доріг у реальному часі, присутнє відображення навантаження на окремі ділянки доріг.

Orange - це веб-система, яка була створена для використання у штаті Джорджия, США. Версії до 3.0 включають основні компоненти в C++ із обгортками в Python, доступні на GitHub. Починаючи з версії 3.0, Orange використовує поширені бібліотеки з відкритим кодом Python для наукових обчислень, такі як numpy, scipy та scikit-learn, в той час як його графічний інтерфейс користувача працює в рамках платформи Qt між платформами [3].

GDOT має наступну функціональність:

– віджети для відображення, фільтрації трафіку;
– віджети для загальної візуалізації (графічна скринька, аналіз перешкод, середня швідкість);
– перехресна валідація, процедури на основі вибірки трафіку, оцінка методів прогнозування.

Приклад візуалізації даних за допомогою GDOT представлено на рисунку 1.2.

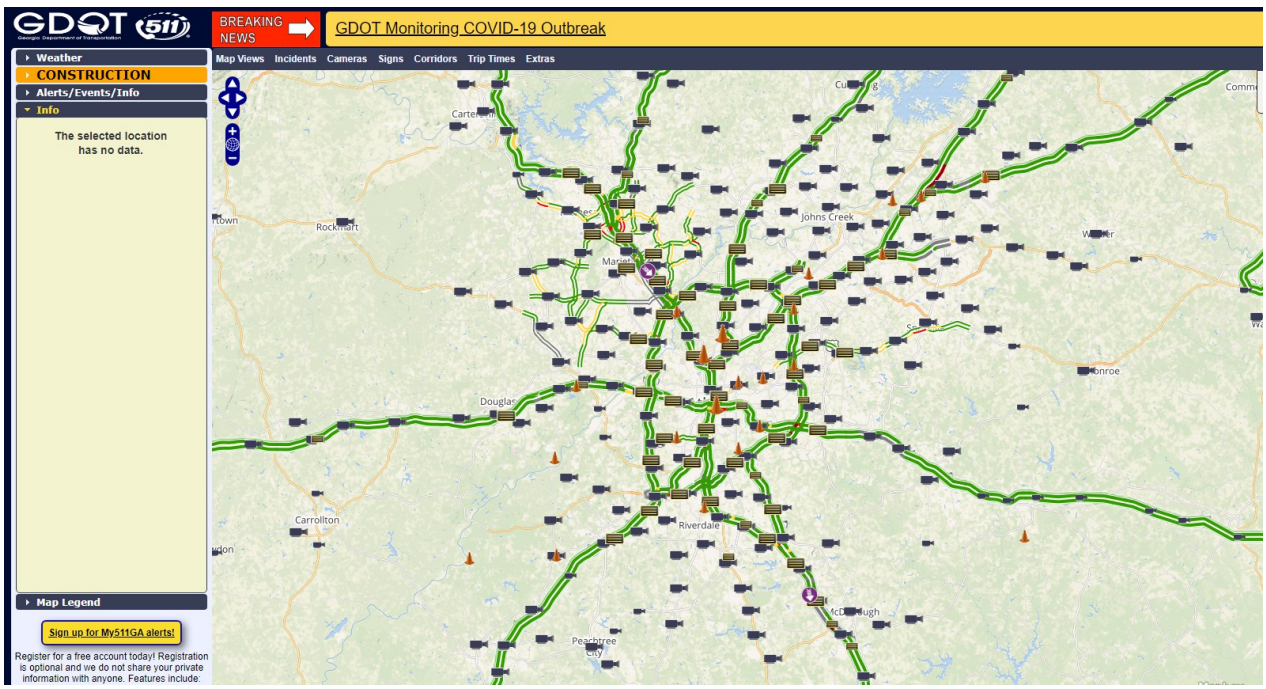


Рисунок 1.2 – візуалізація даних за допомогою GDOT

Головним недоліком додатку є те, що програма підтримує цілий спектр віджетів, які потрібно завжди доставляти на вашу робочу машину (замість використання єдиної системи), а також відсутність можливості задати свій розклад завантаження нових даних.

HAL24k Solutions — це набір рішень для розумного управління трафіком та завантаженням доріг у місті. HAL24K вирішує проблеми, які впливають на мобільність у містах та дозволяє збирати дані для їх вирішення. Також даний набір рішень дозволяє проводити аналіз поточної ситуації та дозволяє робити прогноз, як саме зміниться ситуація при впровадженні запропонованих дій.

Використовуючи декілька потоків даних як необроблений вхід, наша платформа AI та система для збору поточних даних Dimension дозволяє створювати, масштабувати та впроваджувати ефективні рішення розумної мобільності.

Місцеві органи влади, дорожні оператори та будівельні компанії можуть використовувати дані в режимі реального часу, щоб створити діючі відомості для поліпшення контролю руху та експлуатації.

HAL24k надає користувачам наступний перелік можливостей:

- надає можливість імпорту даних з бази даних, текстових файлів у форматі CSV, а також попереднє опрацювання цих даних за допомогою різноманітних алгоритмів (фільтрів). Ці фільтри використовуються для трансформування даних, групування даних за відповідними характеристиками, а також для видалення певних атрибутів;
- надає можливість застосувати різноманітні фільтри та аплети для того щоб оцінити результати, відобразити графіки тощо;
- надає доступ до методів, які дозволяють оцінити взаємозв'язки між факторами створення трафіку;
- дозволяє ідентифікувати фактори, які найбільш впливають на оптимальність руху;
- відображає точкові діаграми.

Приклад програмного інтерфейсу HAL24k наведено на рисунку 1.3.

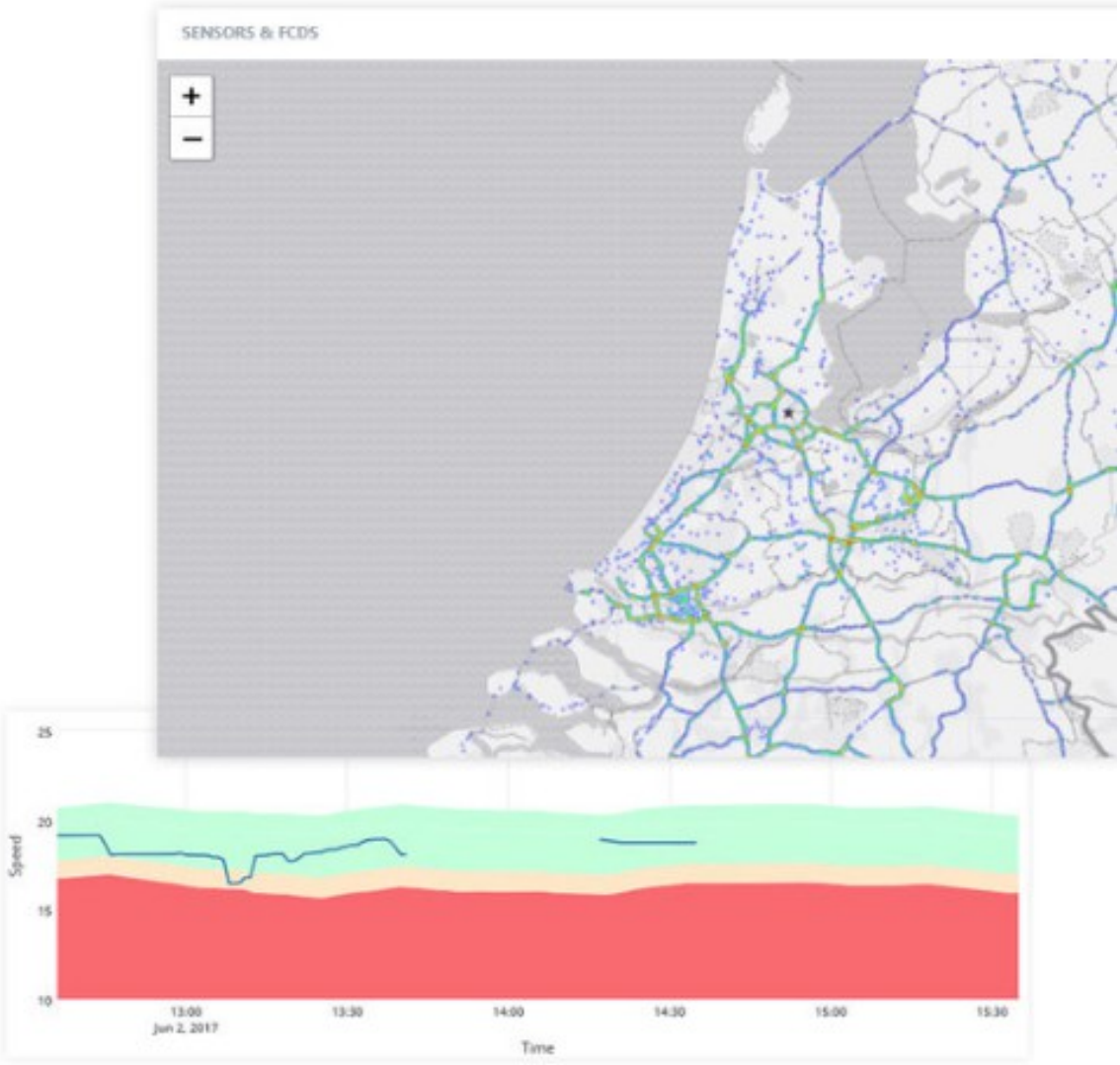


Рисунок 1.3 – Програмний інтерфейс HAL24k

З головних мінусів – досить складний візуальний інтерфейс, неможливість кастомізації інтефейсу щодо потреб користувача.

Жодна із сучасних програмних реалізацій та систем не є вирішує задачу зменшення шкідливих викидів, як пріоритетну ціль [4]. В наведені системи лише оптимізують трафік для легшого пересування людей. Сьогодні не існує рішення, яке дозволяє будувати шляхи з найменшими викидами до атмосфери. Також кожна з існуючих систем намагається дослідити викиди трафіку та його контроль базуючись лише на власне використання, що створює проблему відсутності відкритих даних для аналізу та будування систем, які могли би використовувати існуючи дані.

1.4 Постановка задачі

Метою роботи є дослідження основних методів, технологій та алгоритмів для будування маршрутів перевезень з мінімальною кількістю шкідливих викидів. Результатом дослідження повинна стати розробка веб-додатку на основі проведених досліджень.

Для забезпечення роботи веб-додатку необхідна наявність у користувача пристрою з доступом до мережі Інтернет та щонайменше наступними системними вимогами: Microsoft Edge, Google Chrome, Mozilla Firefox.

Для максимально ефективного планування та вібображення найбільш екологічного маршруту в системі необхідно реалізувати:

- будування графу переміщення відповідно маршруту;
- систему із можливістю анализу і візуалізації маршруту за вказаними додатковими параметрами;
- методи усунення та оптимізації критичних дорожніх ділянок.

Функціонал додатку повинен бути наступний:

- авторизація користувача в системі;
- можливість налаштування параметрів даних мап;
- отримання візуального відображення маршрутів на мапі;
- можливість відображення критичних ділянок забруднення;
- можливість задавати точки переміщення за маршрутом.

Відповідно до аналізу предметної галузі, задоволення потреб кінцевих користувачів системи будування маршрутів перевезень з мінімальною кількістю шкідливих викидів є пріоритетним напрямом діяльності, а системи обробки трафіку набирають все більшу популярність, отже аналіз та розробка системи будування маршрутів перевезень з мінімальною кількістю шкідливих викидів та є актуальною.

2 ОПИС ДОСЛІДЖЕНЬ ТА ПРИЙНЯТИХ ПРОГРАМНИХ РІШЕНЬ

2.1 Аналіз підходів з знаходження найкоротшого шляху у графі

Теорія графів - це вивчення графів, які є математичними структурами, що використовуються для моделювання парних відносин між об'єктами (рис 2.1). Граф у цьому контексті складається з вершин (їх також називають вузлами або точками), які з'єднані ребрами (також називаються посиланнями або лініями). Розрізняють неорієнтовані графи, де краї симетрично пов'язують дві вершини, і орієнтовані графи, де ребра пов'язують дві вершини несиметрично; Графи є одним із головних об'єктів вивчення дискретної математики [5].

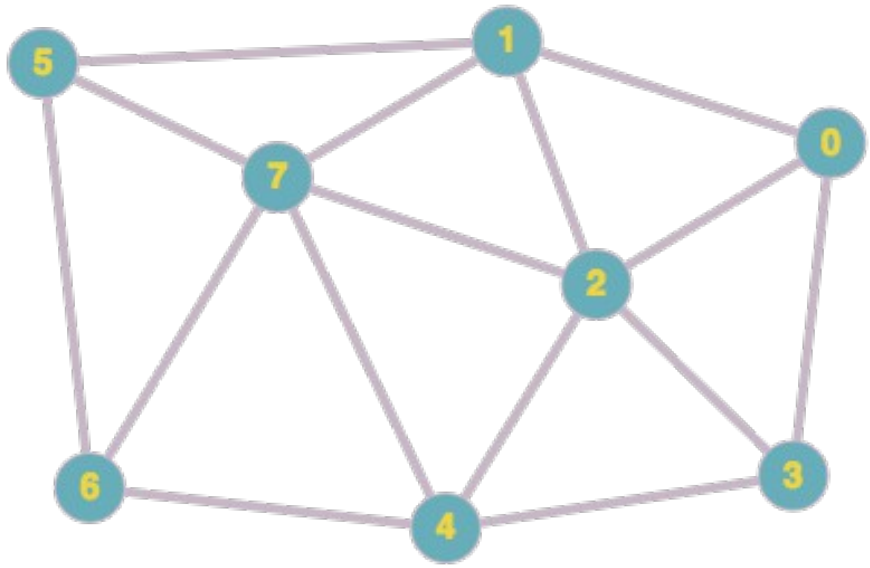


Рисунок 2.1 – Зображення графу

Графи використовуються у багатьох галузях науки та техніки. Вони можуть бути використані для моделювання великої кількості відносин та зв’язку у фізичних, біологічних та хімічних процесах. Багато задач, які мають практичне відображення можуть бути представленні за допомогою графів.

Для роботи з знаходженням найкоротшого шляху у графі існує багато алгоритмів. Та усі вони базуються на декількох основних алгоритмах:

- DFS(Depth-first search);
- BFS(Breadth-First Search);
- Двонаправлений пошук;
- Алгоритм Дейкстри;
- Алгоритм Беллмана-Форда.

Кожен з алгоритмів має свої властивості. Тому потрібно проаналізувати кожен з них та обрати який з алгоритмів можна застосувати для рішення поставленої задачі.

Треба зауважити, що існує велика класифікація графів. Графи можуть мати ваги на ребрах, на вершинах, бути орієнтованими або ні, мати цикли, бути деревом та інше.

З точки зору предметної галузі найголовнішою характеристикою графа у даному випадку буде граф з вагою на ребрах. В цілому граф може бути, як орієнтованим, так і неорієнтованим. Це залежить від характеристик доріг. Але задля наочності у цьому розділі будуть розглянуті неорієнтовані графи.

2.2 DFS(Depth-first search)

Алгоритм пошучу в глубину – це один з найвідоміших та найпростіших алгоритмів для пошуку найкоротшого шляху у графі (рис. 2.2). Однак він має певні недоліки та обмеження. Перш за все граф повинен бути ациклічним, тобто не мати циклів, і його ребра не повинні мати вагу. Якщо ці умови виконані, ви можете скористатися дещо зміненим DFS, щоб знайти свій найкоротший шлях [6].

Якщо шлях між стартовою та кінцевою вершинами не існує, найкоротший шлях матиме довжину -1. Ми ініціалізуємо найкоротший шлях з цим значенням і запускаємо рекурсивну DFS. Ця рекурсивна DFS дещо змінена в тому сенсі, що вона буде відслідковувати глибину пошуку і зупинятися, як тільки вона досягне

stopNode. Поточна глибина, коли вона досягає stopNode - це найкоротша довжина шляху.

```
const shortestPathDfs = (startNode, stopNode) => {
  const previous = new Map();
  let shortestDistance = -1;
  const dfs = (currentNode, depth) => {
    if (currentNode === stopNode) {
      shortestDistance = depth;
    } else {
      for (let neighbour of adjacencyList.get(currentNode)) {
        previous.set(neighbour, currentNode);
        dfs(neighbour, depth + 1);
      }
    }
  };
  dfs(startNode, 0);
  return { shortestDistance, previous };
};
```

Рисунок 2.2 – Реалізація DFS

Причина, чому ми не можемо використовувати його для циклічних графів, полягає в тому, що коли ми знаходимо шлях, ми не можемо бути впевнені, що це найкоротший шлях. DFS не дає такої гарантії. Тобто для його роботи потрібно мати граф який не має циклів (такий граф є деревом) або усунути їх.

Розглянемо продуктивність алгоритму. Нехай n - кількість вузлів і e - кількість ребер нашого графа. Цей алгоритм має складність у часі O (n). Через свою рекурсивну природу він використовує стек викликів і тому має споживання асимптотичної пам'яті O(n). Також, через те що граф повинен бути ациклічним уникається можлива складність у O(n+e). Оскільки це означає, що e ≤ n-1 і тому O (n*e) = O(n).

2.3 BFS(Breadth-First Search)

Трохи модифікований BFS – дуже корисний алгоритм для пошуку найкоротшого шляху (рис. 2.3). Він простий і застосовний до всіх графіків без ваги ребер:

```
const shortestPathBfs = (startNode, stopNode) => {
  const previous = new Map();
  const visited = new Set();
  const queue = [];
  queue.push({ node: startNode, dist: 0 });
  visited.add(startNode);

  while (queue.length > 0) {
    const { node, dist } = queue.shift();
    if (node === stopNode) return { shortestDistande: dist, previous };

    for (let neighbour of adjacencyList.get(node)) {
      if (!visited.has(neighbour)) {
        previous.set(neighbour, node);
        queue.push({ node: neighbour, dist: dist + 1 });
        visited.add(neighbour);
      }
    }
  }
  return { shortestDistance: -1, previous };
};
```

Рисунок 2.3 – Реалізація BFS

Це просто реалізація BFS, яка відрізняється лише кількома деталями. З кожним вузлом, який зберігається у черзі, ми додатково зберігаємо відстань до startNode. Коли ми досягнемо stopNode, ми просто повернемо відстань, яку було збережено разом з нею [7].

Це працює через природу BFS: сусід сусіда не відвідується до того, як були відвідані всі прямі сусіди. Як наслідок, усі вузли, що мають відстань x від startNode, відвідуються після відвідування всіх вузлів з відстані менше x. BFS

спочатку відвідає вузли з відстані 0, а потім усі вузли з відстані 1 тощо. Ця властивість є причиною, чому ми можемо використовувати BFS для пошуку найкоротшого шляху навіть у циклічних графіках.

Продуктивність: нехай g описує найбільшу кількість сусідніх вузлів для будь-якого вузла нашого графіка. Більше того, нехай d - довжина найкоротшого шляху між startNode та stopNode. Тоді цей алгоритм має часову складність O (g^d).

BFS здійснює пошук графіка в так званих рівнях. Кожен вузол на рівні має однакову відстань до стартового вузла. Для досягнення рівня 1, кроків O (g^2) потрібно досягти рівня 2 і так далі. Тому для досягнення рівня d потрібні кроки O (g^d). Використовуючи змінні n і e знову, час виконання все ще O (n + e). Однак O (g^d) - більш точне твердження, якщо шукати найкоротший шлях.

У деяких графіках черга може містити всі її вузли. Тому він також має просторову складність O(n).

Невелике зауваження: фактичний час виконання вищевказаної реалізації гірший, ніж O(n+e). Причина в тому, що масив JavaScript використовується в якості черги. Операція зсуву займає час O(s), де s - розмір черги. Однак можливо реалізувати чергу в JavaScript, яка дозволяє операції enqueue та dequeue в O(1), як описано в моїй попередній частині.

2.4 Двонаправлений пошук

Третій спосіб отримання найкоротшого шляху – це двонаправлений пошук. Як і BFS, він застосовується у неорієнтованих графах без ваги ребер. Для здійснення двонаправленого пошуку ми в запускаємо один BFS з node1 і один з node2 одночасно. Коли обидва BFS зустрічаються, ми знайшли найкоротший шлях.

Це не найкоротший алгоритм, але його реалізація є достатньо простою оскільки в його основі лежить BFS (рис 2.4). Такий алгоритм має значну перевагу

перед класичним BFS, тому що він є значно продуктивнішим. Якщо BFS дозволяє нам знайти шлях довжиною l за певну кількість часу, двосторонній пошук дозволить нам знайти шлях довжиною 2l [8].

```
const bidirectionalSearch = (startNode, stopNode) => {
  const previous = new Map();
  const visited1 = new Map();
  const visited2 = new Map();
  const queue1 = [];
  const queue2 = [];
  queue1.push({ node: startNode, dist: 0 });
  queue2.push({ node: stopNode, dist: 0 });
  visited1.set(startNode, 0);
  visited2.set(stopNode, 0);

  while (queue1.length > 0 || queue2.length > 0) {
    if (queue1.length > 0) {
      const { node, dist } = queue1.shift();
      if (visited2.has(node)) {
        return {
          shortestDistance: dist + visited2.get(node),
          previous,
        };
      }

      for (let neighbour of adjacencyList.get(node)) {
        if (!visited1.has(neighbour)) {
          previous.set(neighbour, node);
          queue1.push({ node: neighbour, dist: dist + 1 });
          visited1.set(neighbour, dist + 1);
        }
      }
    }
```

Рисунок 2.4 – Реалізація Двонаправленого пошуку

Розглянемо ефективність алгоритму більш детально. Двонаправлений пошук закінчується після рівнів d/2, оскільки це центр шляху. Обидва одночасно BFS відвідують вузлів, що становить в загальній складності. Це призводить до O(), а тому робить двонаправлений пошук швидшим, ніж BFS, на коефіцієнт .

2.5 Алгоритм Дейкстри

Цей алгоритм э одним з найвідомішим для пошуку найкоротшого шляху у графі. Його перевага перед DFS, BFS та двонаправленим пошуком полягає в тому, що ви можете використовувати його у всіх графіках із позитивною вагою ребра (рис 2.5). Однак цей алгоритм не працює з негативними значеннями, оскільки припинення в цьому випадку не гарантується [9].

Щоб зрозуміти алгоритм Дейкстри, важливо зрозуміти пріоритетні черги. Черга з пріоритетом - це абстрактна структура даних, яка дозволяє виконувати наступні операції:

- isEmpty: перевіряє, чи в черзі пріоритетів містяться якісь елементи;
- insert: вставляє елемент разом із значенням пріоритету;
- ExtraHighestPriority: повертає елемент з найвищим пріоритетом і видаляє його з черги пріоритетів

З’ясуємо що значить пріоритет. Математично пріоритет повинен дозволяти визначати частковий порядок на елементах черги пріоритетів. У нашому випадку нам потрібна черга пріоритету для зберігання всіх вузлів у графіку разом із їхніми відстань до нашого стартового вузла. Отже наша операція extraGighestPriority буде називатися extraMin, що є більш описовим ім'ям для отримання вузла з мінімальною відстані до початкового вузла.

Перейдемо до принципу роботи алгоритму Дейкстри. Обхід графу починається з ініціалізації найкоротшого шляху від стартового вузла до кожного іншого вузла в існуючому графі. Спочатку це буде нескінченність для кожного вузла, крім самого стартового вузла. Пусковий вузол буде ініціалізований з 0, оскільки це відстань до себе. Ми вставляємо всі вузли до нашої черги пріоритету разом з їх попередньо ініціалізованою дистанцією до обраного початкового вузла як пріоритетного [10].

Тепер переходимо до проходу графу. Поки черга пріоритетів не буде порожньою, ми дістаємо вузол з поточною найкоротшою відомою відстані до нашого стартового вузла. Назвемо це currentNode. Тоді ми перебираємо петлі над усіма сусідами currentNode, і для кожного з них перевіряємо, чи досягає його

через currentNode коротше, ніж відомий в даний час найкоротший шлях до цього сусіда. Якщо так, ми оновлюємо найкоротшу відстань до сусіда і продовжуємо. Але подивіться самі:

```
const dijkstra = (startNode, stopNode) => {
  const distances = new Map();
  const previous = new Map();
  const remaining = createPriorityQueue(n => distances.get(n));
  for (let node of adjacencyList.keys()) {
    distances.set(node, Number.MAX_VALUE);
    remaining.insert(node);
  }
  distances.set(startNode, 0);

  while (!remaining.isEmpty()) {
    const n = remaining.extractMin();
    for (let neighbour of adjacencyList.get(n)) {
      const newPathLength = distances.get(n) + edgeWeights.get(n).get(neighbour);
      const oldPathLength = distances.get(neighbour);
      if (newPathLength < oldPathLength) {
        distances.set(neighbour, newPathLength);
        previous.set(neighbour, n);
      }
    }
  }

  return { distance: distances.get(stopNode), path: previous };
};
```

Рисунок 2.5 – Реалізація алгоритму Дейкстри

Ми надали функцію зворотного виклику черзі пріоритетів, яка має доступ до нашої карти відстаней. Це використовується в реалізації черги пріоритетів для отримання мінімальної відстані. Однак реалізація черги з пріоритетом не повинна обговорюватися в цьому творі [11].

Часова складність цього алгоритму сильно залежить від реалізації черги пріоритетів. Нехай n - кількість вузлів і e - кількість ребер у графіку. Якщо він

реалізований простим масивом, алгоритм Діккстри запуститься в O (n^2) [12]. Однак, більш поширена реалізація використовує купу Фібоначчі в якості черги пріоритетів. У цьому випадку час виконання знаходиться в межах O (e + n log (n)).

2.6 Алгоритм Беллмана-Форда

Останній алгоритм, який треба розглянути, - це алгоритм Беллмана-Форда. Цей алгоритм був названий в честь двох вчених: Річарда Беллмана та Лістера Форду. Вони винайшли цей алгоритм у 1956 році при вивчені задачі пошуку найкоротшого шляху в графі (рис 2.6). На відміну від алгоритму Дейкстри, він може мати справу з негативними вагами. Є лише одне обмеження: графік не повинен містити негативних циклів. Негативний цикл - це цикл, ребра якого дорівнюють негативному значенню. Однак алгоритм здатний виявляти негативні цикли і тому припиняється - хоча і без найкоротшого шляху [13].

Алгоритм дуже схожий на алгоритм Дейкстри, але він не використовує черги пріоритетів. Натомість він неодноразово перебирає петлі по всіх краях, оновлюючи відстані до початкового вузла аналогічно алгоритму Дейкстри. Нехай n знову - кількість вузлів у нашому графіку. Алгоритм Беллмана-Форда перетинає рівно n-1 раз по всіх краях, оскільки без циклу шлях у графі ніколи не може містити більше ребер, ніж n-1.

Після багаторазового циклічного перегляду всіх ребер алгоритм ще раз перетинає всі ребра. Якщо одна з відстаней все ще не є оптимальною, це означає, що в графіку повинен бути негативний цикл [14].

Розглянемо продуктивність алгоритму. Можливість боротьби з негативними вагами в крайній частині виходить ціною. Складність виконання алгоритму Беллмана-Форда становить O (n * e).

```
const bellmanFord = (startNode, stopNode) => {
  const distances = new Map();
  const previous = new Map();
  for (let node of adjacencyList.keys()) {
    distances.set(node, Number.MAX_VALUE);
  }
  distances.set(startNode, 0);

  for (let i = 0; i < adjacencyList.size - 1; i++) {
    for (let n of adjacencyList.keys()) {
      for (let neighbour of adjacencyList.get(n)) {
        const newPathLength = distances.get(n) + edgeWeights.get(n).get(neighbour);
        const oldPathLength = distances.get(neighbour);
        if (newPathLength < oldPathLength) {
          distances.set(neighbour, newPathLength);
          previous.set(neighbour, n);
        }
      }
    }
  }
  for (let n of adjacencyList.keys()) {
    for (let neighbour of adjacencyList.get(n)) {
      if (distances.get(n) + edgeWeights.get(n).get(neighbour) < distances.get(neighbour)) {
        // there is a cycle with negative weight
        return null;
      }
    }
  }

  return { distance: distances.get(stopNode), path: previous };
};
```

Рисунок 2.6 – Реалізація алгоритму Беллмана-Форда

Через додаткову складність цього алгоритму, його слід використовувати лише в тому випадку, якщо у вас дійсно є негативні ваги та ми не маємо змоги усунути їх.

2.7 Дослідження графу

Згідно з поставленою задачею, алгоритм повинен працювати з неорієнтованим графом, який має вагу на ребрах. Однак, для вирішення проблеми знаходження оптимального за викидами маршруту нам потрібно мати дві характеристики для кожного ребра - це відстань та викид CO2 на обраній ділянці (рис 2.7).

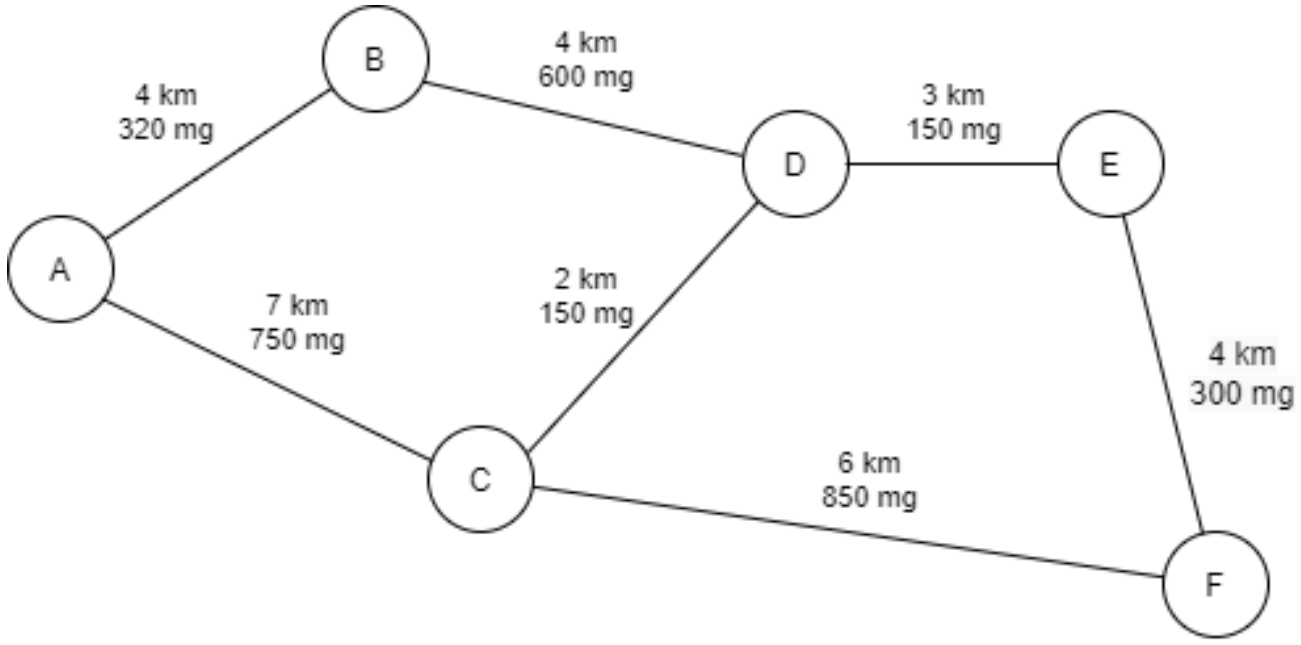


Рисунок 2.7 – Граф відстаней та викидів

Візьмемо для прикладу наступний граф (рис 2.7). Розглядаючи класичне знаходження найкоротшого маршруту за відстанню. Знайдемо найменшу відстань між вершиною A та F. Всього можливо побудувати 4 маршрути:

- A-B-D-E-F(15 км);
- A-B-D-C-F(16 км);
- A-C-D-E-F(16 км);
- A-C-F(13 км).

Найкоротшим шляхом буде маршрут A-C-F. Що відображено на рисунку 2.8 нижче.

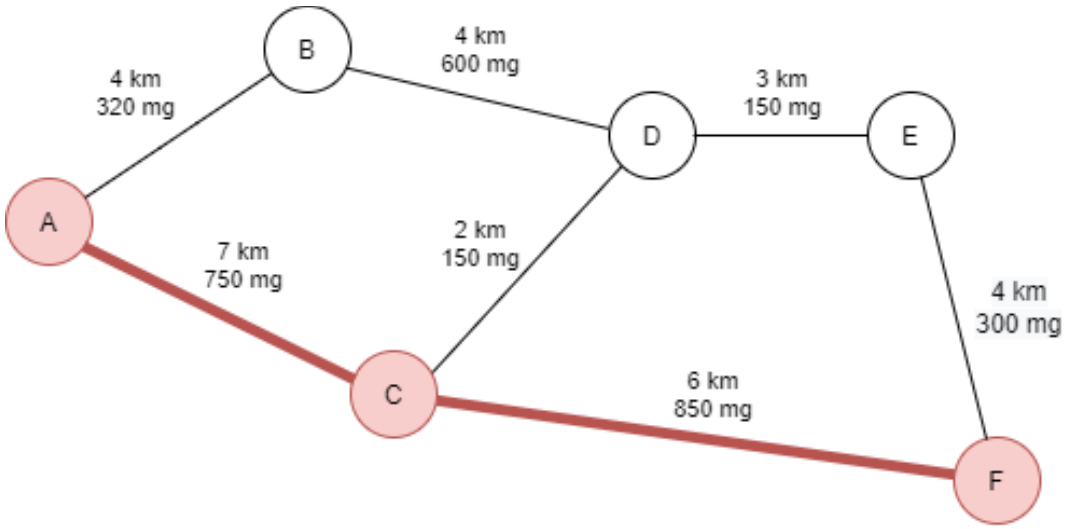


Рисунок 2.8 – Найкоротший шлях

Тепер розглянемо той самий граф, але для підрахунку маршруту використаємо значення викидів.

- A-B-D-E-F(1370 мг);
- A-B-D-C-F(1920 мг);
- A-C-D-E-F(1350 мг);
- A-C-F(1600 мг).

Найефективнішим шляхом буде маршрут A-C-D-E-F. Що відображено на рисунку 2.9 нижче.

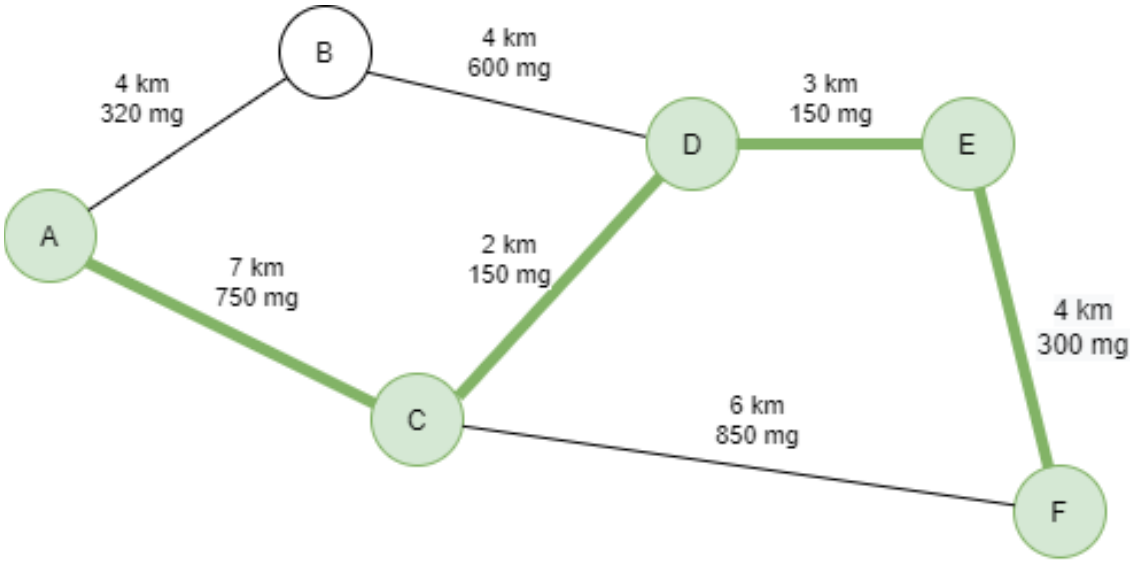


Рисунок 2.9 – Шлях з найменшою кількістю викидів

На розглянутому прикладі можна побачити, що для різних характеристик ми отримаємо не співпадіння маршрутів, що є реальною ситуацією через різні умови на ділянках доріг, які залежать від трафіку на ділянці, якості дорожнього покриття, характеру місцевості та іншого [15].

Якщо брати приклад з реального життя, то можна привести наступний приклад.

Якщо нам потрібно дістатися автомобілем метро Перемога з метро Холодна Гора у місті Харкові, то в нас є 2 основних шляху. Поїхати через місто або використати кільцеву дорогу (рис 2.10).

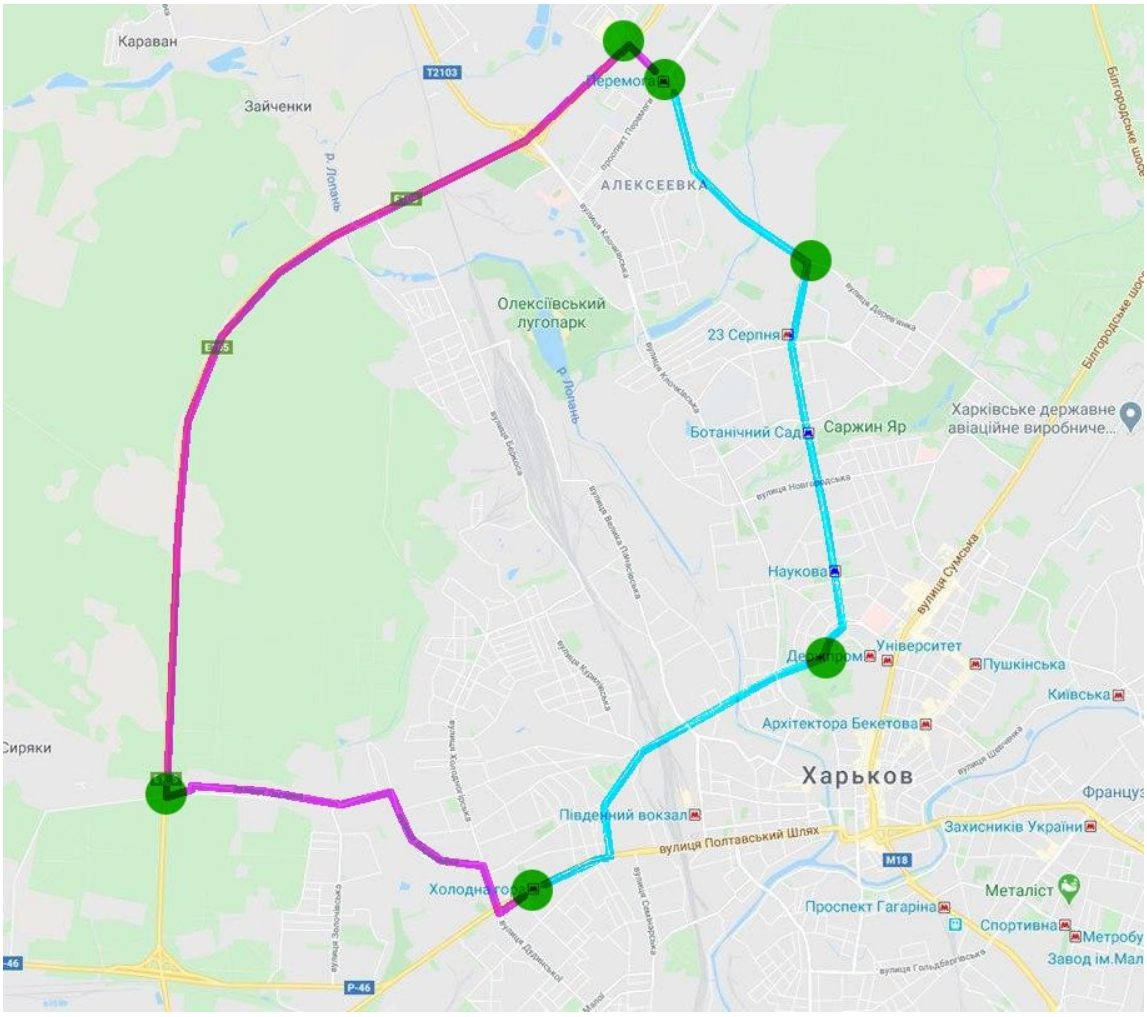


Рисунок 2.10 – Можливі шляхи

Шлях через місто є коротшим – 10,8 км. А шлях через кільцеву дорогу – 15,5 км. В ідеальних умовах ми зможемо швидко та без зайвих зупинок дістатися

точки призначення через місто. Але в реальних умовах в місті є багато світлофорів та існує певний трафік [16].

Тому якщо ми візьмемо дані з відкритих джерел щодо того, скільки викидів CO2 отримується з 1 літру бензину, то це значення буде дорівнювати 2,3 кілограмів CO2.

Враховуючи, що витрати пального у місті будуть сягати десь 10 літрів на 100 км, а за городом витрати будуть 6 літрів на 100 км можна зробити наступний апроксиматичний підрахунок:

((15 * 6)/100)*2,3=2,07 кг CO2 якщо їхати через кільцеву дорогу

((11 * 10)/100)*2,3=2,53 кг CO2 якщо їхати через місто

Ми бачимо, що при такому відносному підрахунку використання кільцевої дороги призводить до значно менших викидів ніж використання дорог у місті.

Збудуємо граф для підрахунку викидів на всіх ділянках маршруту (рис 2.11).

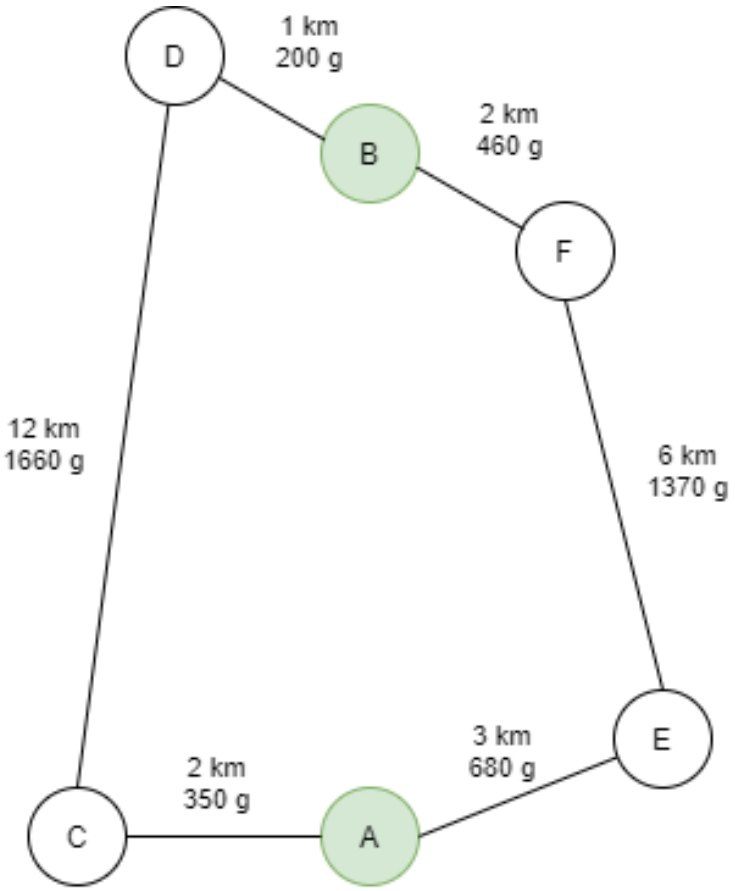


Рисунок 2.11 – Шляхи перенесені на граф

На графі було виділено наступні ділянки:

- A – C – метро Холодна Гора – Кільцева дорога;
- C – D – Кільцева дорога;
- C – B – Кільцева дорога – метро Перемога;
- A – E – метро Холодна Гора – проспект Науки;
- E – F – проспект Науки – вулиця Ахсарова;
- F – B – вулиця Ахсарова – метро Перемога.

Враховуючи характер усіх ділянок результуючий підрахунок буде мати наступний вигляд.

- Шлях через місто – 2,51 кг CO2;
- Шлях через кільцеву дорогу – 2,21 кг CO2.

Як результат ми отримали різницю в понад 13% між викидами на різних шляхах.

2.8 Реалізація алгоритму

Враховуючи попередній аналіз існуючих алгоритмів (рис 2.12) та розгляду побудови графу, було прийняте рішення взяти за основу алгоритм Дейкстри для побудови результуючого алгоритму, тому що він здатен працювати з графами ребра яких мають вагу [17].

Algorithm	Negative Edge Weights	Positive Edge Weights > 1	Cyclic	Runtime
DFS	✗	✗	✗	O(n + e)
BFS	✗	✗	✓	O(n + e) or $O(g^d)$
Bidirectional Search	✗	✗	✓	O(n + e) or $O(g^{(d/2)})$
Dijkstra	✗	✓	✓	O(e + n log(n))
Bellman-Ford	✓	✓	✓	O(n * e)

Рисунок 2.12 – Порівняння алгоритмів

Також цей алгоритм буде працювати якщо граф має цикли. Алгоритм Белмана-Форда теж може буди застосований, але він має додаткові можливості, які не є важливими у контексті даної задачі (рис 2.13).

```
1 reference
public void dijkstra(int[,] graph, int src)
{
    V = graph.GetLength(0);

    int[] dist = new int[V];
    bool[] sptSet = new bool[V];

    for (int i = 0; i < V; i++)
    {
        dist[i] = int.MaxValue;
        sptSet[i] = false;
    }

    dist[src] = 0;

    for (int count = 0; count < V - 1; count++)
    {
        int u = minDistance(dist, sptSet);

        sptSet[u] = true;

        for (int v = 0; v < V; v++)
            if (!sptSet[v] && graph[u, v] != 0 && dist[u] != int.MaxValue && dist[u] + graph[u, v] < dist[v])
                dist[v] = dist[u] + graph[u, v];
    }

    printSolution(dist);
}
```

Рисунок 2.13 – Код алгоритму Дейкстри

На наступних рисунках відображена реалізація алгоритму Дейкрсти, на допоміжного алгоритму знаходження мінімальної відстані (рис 2.14).

```
int minDistance(int[] dist,
                bool[] sptSet)
{
    int min = int.MaxValue, min_index = -1;

    for (int v = 0; v < V; v++)
        if (sptSet[v] == false && dist[v] <= min)
        {
            min = dist[v];
            min_index = v;
        }

    return min_index;
}
```

Рисунок 2.14 – Код алгоритму знаходження мінімальної відстані

Для того щоб використати цей алгоритм для нашого графу потрібно використати не 1 масив з відношенням та вагами між вершинами, а 2 Один масив

буде використаний для знаходження мінімальної відстані, інший для знаходження мінімальних викидів. Для цього використаємо алгоритм для кожного з масивів. Тому ініціалізація та виклик методів буде виглядати наступним чином (рис 2.15).

```
int[,] graphDistance =  new int[,] { {   0,    4,    0,    0,    0 },
                                     {   4,    0,    8,    0,    0 },
                                     {   0,    8,    0,    7,    0 },
                                     {   0,    0,    7,    0,    9 },
                                     {   0,    0,    0,    9,    0 } };
int[,] graphPollution = new int[,] { {   0,  800,    0,    0,    0 },
                                     { 800,    0, 1850,    0,    0 },
                                     {   0, 1850,    0, 1200,    0 },
                                     {   0,    0, 1200,    0, 1620 },
                                     {   0,    0,    0, 1620,    0 } };
var t = new DijkstrasAlgorithm();
t.dijkstra(graphDistance, 0);
t.dijkstra(graphPollution, 0);
```

Рисунок 2.15 – Ініціалізація масивів та виклик методів

Для відображення був написаний метод для відображення результатів роботи алгоритму (рис 2.16).

```
Vertex          Distance from Source
0               0
1               4
2               12
3               19
4               28
Vertex          Distance from Source
0               0
1               800
2               2650
3               3850
4               5470
```

Рисунок 2.16 – Вивід консолі

У виводі консолі можна побачити мінімальні значення, які потрібні для досягання кожної з вершин відправляючись з першої.

Таким чином, можна зробити висновок, що найефективнішим підходом для рішення поставленої задачі, буде використання комбінації двох алгоритмів Дейкстри для кожного з типів даних (кількість шкідливих викидів та відстань). Як було зазначено раніше, саме цей алгоритм дозволяє зробити підрахунок

мінімальної відстані від точки старту, до точки фінішу. А використання двох алгоритмів дає змогу побачити різницю між розрахунком за різними характеристиками. Додаючи сюди результати аналізу графу та прикладу задачі на місцевості, можна зробити висновок у потребі використання отриманого алгоритму для рішення задач даного типу.

3 ПРОЕКТУВАННЯ ПРОГРАМНОЇ СИСТЕМИ

3.1 UML проектування програмної системи

UML (Unified Modeling Language) – стандартизована мова моделювання. UML діаграми є надзвичайно ефективним засобом проектування систем та відображення необхідних властивостей у графічному уявлені. UML був розроблений у 1994-1996 роках трьома програмними інженерами, які працювали у Rational Software. Пізніше UML був прийнятий в якості стандарту в 1997 році і до цього часу він отримав лише кілька оновлень [18].

У сукупності діаграми UML описують межу, структуру та поведінку системи та об'єктів всередині неї. Також треба зауважити, що UML описує роботу як програмного забезпечення, так і апаратних систем.

UML не є мовою програмування, але існують інструменти, які можна використовувати для генерації коду на різних мовах за допомогою діаграм UML. UML має пряме відношення до об'єктно-орієнтованого аналізу та проектування.

При проектуванні програмного рішення були зроблені UML діаграми, які повністю покривають усі аспекти системи. Отримані діаграми розглянуто далі.

Створенні UML діаграми було реалізовано за допомогою веб-застосунку draw.io.

Перша з діаграм, яку потрібно розглянути – це Use case діаграма. За допомогою цієї діаграми, ми можемо почати, як користувач взаємодіє з результуючою системою. У системі існує користувач, який отримує доступ до веб-системи, яка відображає оптимальний маршрут для користувача.

На Use Case діаграмі можна побачити основний функціонал веб додатку. Користувач звертається до системи, щоб отримати відповідну відповідь за одним з можливих запитів. Після цього система звертається до API сервісу, який виконує відповідну обробку запитів та повертає результат користувачеві.

Діаграма зображена на рисунку 3.1.

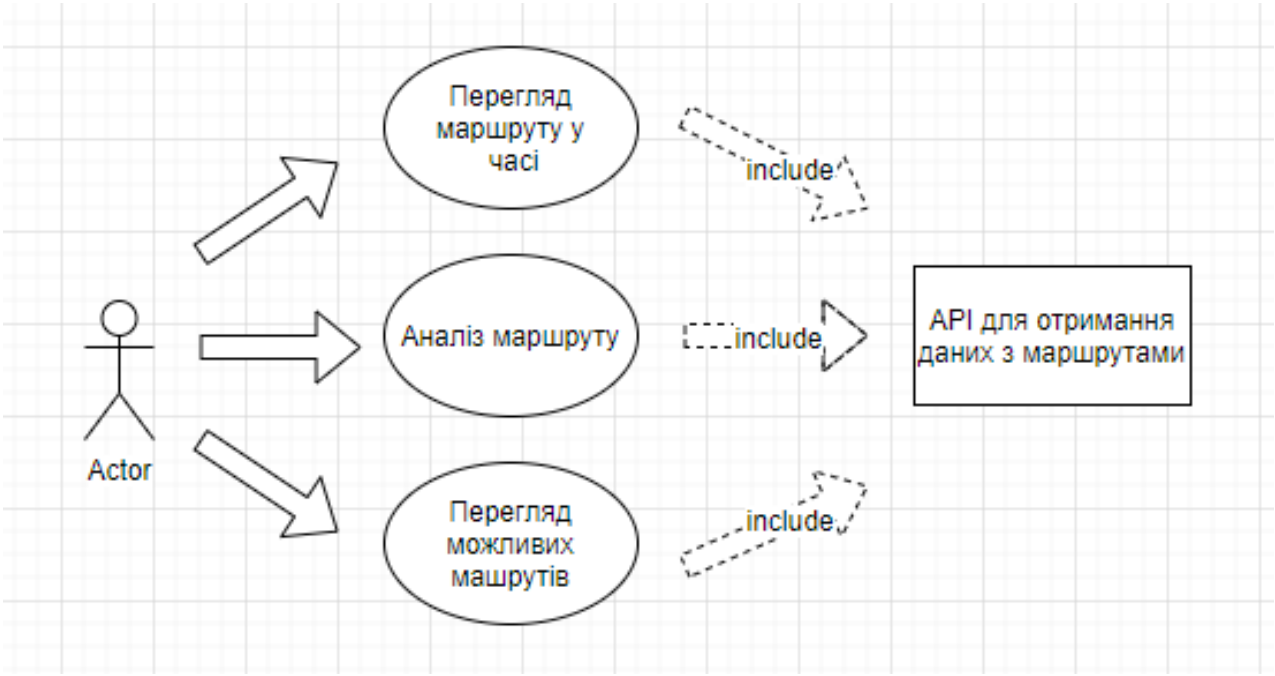


Рисунок 3.1 – Use case діаграма

Якщо нам потрібно більш детально розглянути життєвий цикл запиту, то це можна зробити за допомогою діаграми послідовності. На цій діаграмі відображується основний потік взаємодії користувача та системи з подальшою обробкою запиту системою.

На діаграмі можна побачити що спочатку користувач проходить авторизацію в системі. Клієнт приймає запит та відправляє його до системи, частиною якої є сервіс відповідний за авторизацію. Якщо системі вдається вдало обробити запит, то отримана відповідь повертається на клієнт. Після чого клієнт відображує результат запиту користувача. Отже користувач бачить чи зміг він авторизуватися у системі.

Після цього користувач вже здатен обрати маршрут. Після чого процес відправки запиту та отримання відповіді повторюються. Подальша робота з системою будується аналогічним чином.

Те що ми бачимо на діаграмі послідовності дає змогу дізнатися через які рівні проходить запит та за які дії відповідає кожен з них.

Діаграму наведено на рисунку 3.2.

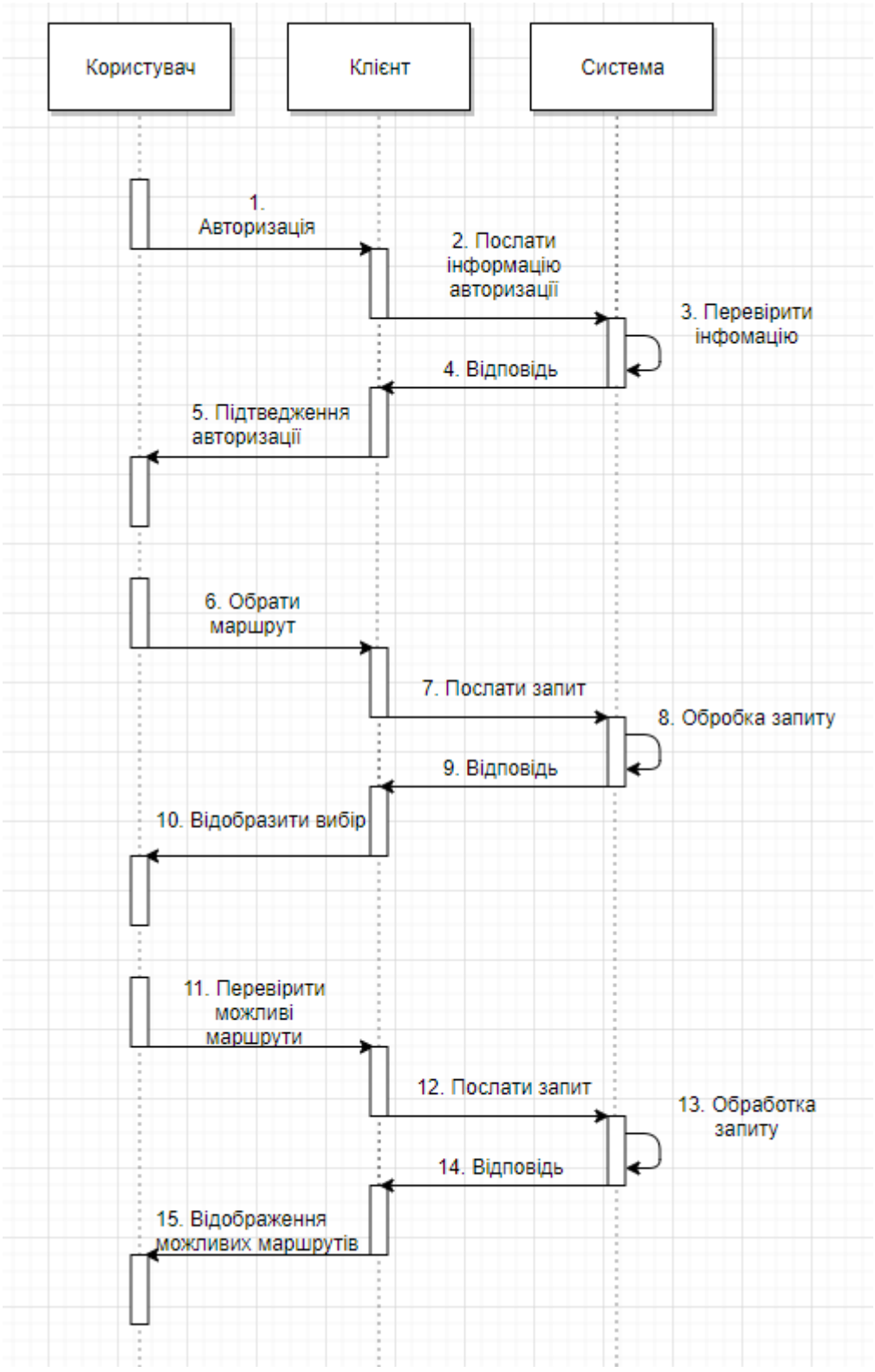


Рисунок 3.2 – Діаграма послідовності

Наступна діаграма, це діаграма комунікації. Ця діаграма використовується для моделювання взаємодії між об’єктами та частинами системи у вигляді впорядкованих повідомлень. Комунікаційна діаграма представляє комбінацію

інформації, взятої з діаграми класів, послідовності і варіантів використання та описує водночас і статичну структуру і динамічну поведінку системи.

Представлена діаграма комунікації демонструє перехід дій між системними компонентами та її рівнями. Коли користувач звертається до системи, то його дії переходять з клієнту до серверу з послідовним запитом до API маршрутів (рис 3.3).

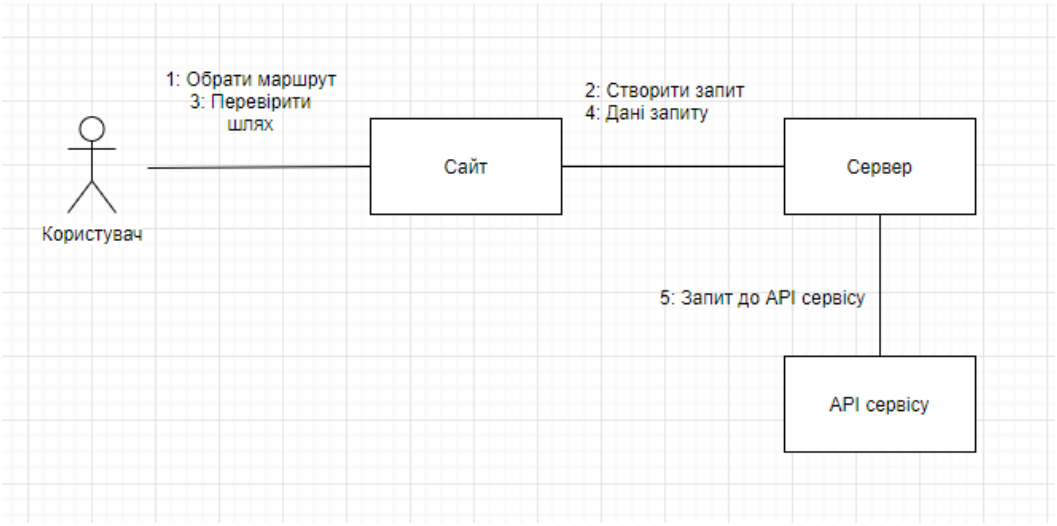


Рисунок 3.3 – Діаграма комунікацій

Діаграма активності допомагає відобразити, як саме користувач виконує взаємодію з системою та демонструє які шаги проходить користувач при використанні веб-додатку. На діаграмі видно, що користувач першим кроком виконує авторизацію, та у разі успішної авторизації переходить до використання основних функціональних можливостей додатку (рис 3.4).

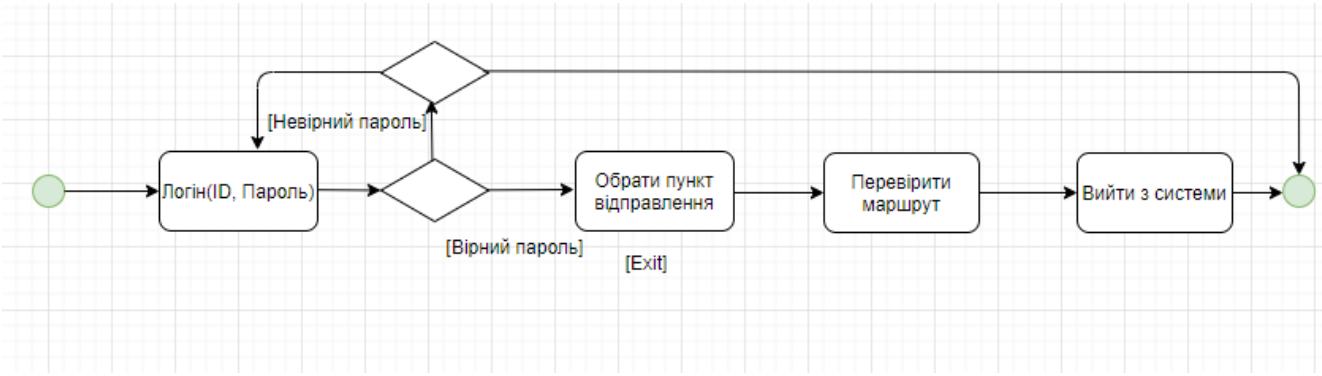


Рисунок 3.4 – Діаграма активності

3.2 Архітектура веб-додатку

Під час розробки було прийняте рішення, що до застосування клієнт-серверної архітектури. Сьогодні майже кожен веб-застосунок використовує таких архітектурний підхід. Данна архітектура дозволяє у повній мірі реалізувати систему розраховану на багату кількість користувачів. Також саме відношення клієнт-сервер є однією з необхідних вимог для будування RESTful системи [19].

Рисунок 3.5 демонструє устрій архітектури програмного рішення. Для кожного рівня вказані основні технології, які були застосовані для реалізації відповідних задач.

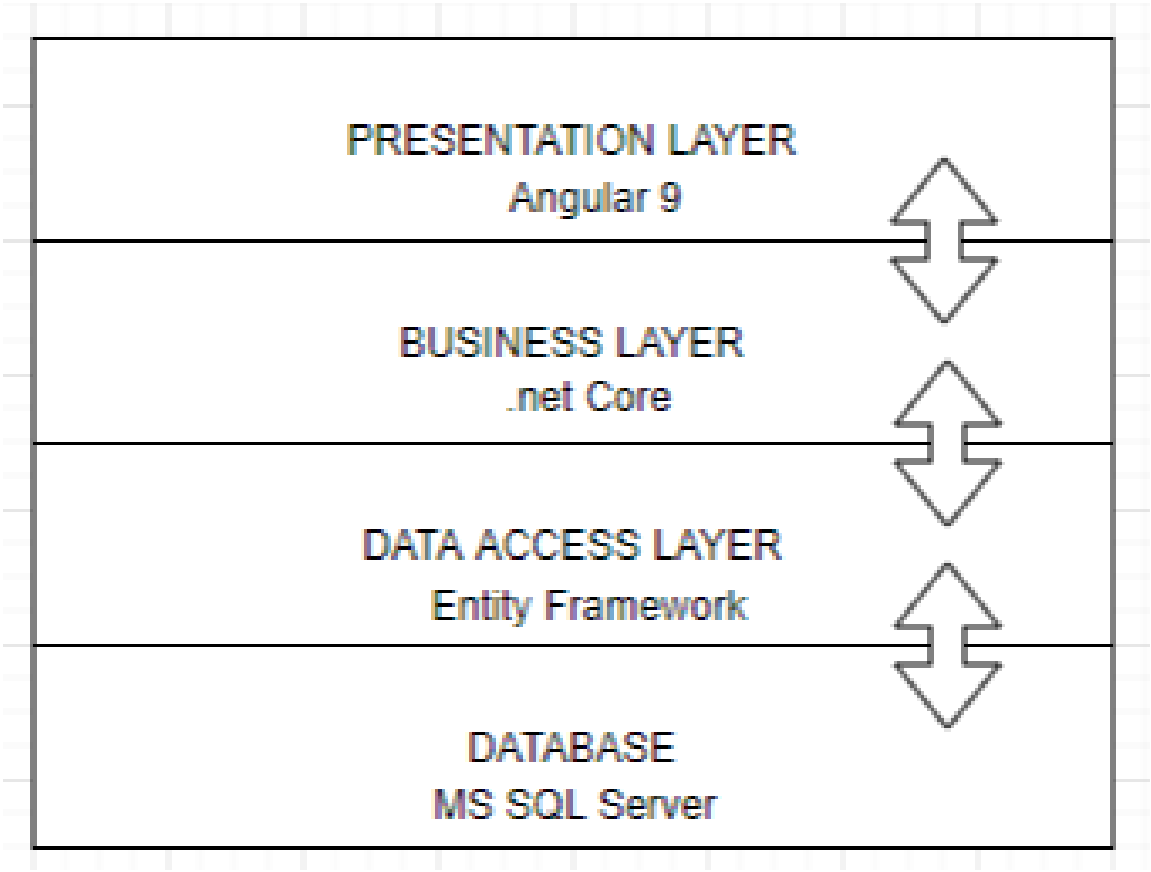


Рисунок 3.5 – Архітектура системи

Presentation Layer – на цьому рівні користувач взаємодіє з системою. На цьому рівні знаходяться усі елементи інтерфейсу з якими взаємодіє користувач. Цей рівень включає компоненти для користувача інтерфейсу, механізм отримання даних від користувача. Фізично цей рівень складається зі html файлів, css-файлів, js-файли, зображення та інше. Також контролери, які використовуються для обробки запитів також знаходяться на цьому рівні. Треба зауважити що валідація даних присутня на даному рівня. Для реалізації цього рівня застосовано окремий проект створений за допомогою фреймворку Angular 9 та стилей Bootstrap 4.

Business Layer – цей рівень складається з компонентів, які відповідають за обробку запитів, які надходять з попереднього рівня. Вся логіка обробки запиту відбувається на цьому рівні. Валідація запитів теж відбувається на цьому рівні. Та у комбінації з валідацією на Presentation Layer ми отримуємо високий рівень безпеки роботи системи.

Data Access Layer – на цьюму рівні зберігаються моделі, які описують сутності, які використовуються для створення схеми бази даних при підході Code First. Коли ми будуємо базу даних за класами, на не через створення схеми безпосередньо через використання СУБД. Також на цьому рівні знаходяться класи для здійснення запитів до бази даних. Тобто на цьому рівні ми маємо класи, які реалізують контекст бази даних(у даному випадку використовується Entity Framework для управління контекстом) та класи, які реалізують патерн репозиторій та Unit of Work.

Database – це рівень, який застосовується для того, щоб зберігати дані, які використовуються нашою системою. У даному випадку використовується реляційна база даних, для роботи з якою використовується мова SQL. Безумовно, при необхідності можна зробити часткову або повну міграцію на нереляційні рішення.

4 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

4.1 Вибір засобів розробки

Першим кроком було вирішено яке середовище більш за все підійде для розробки веб-системи. Для написання серверної частини було обрано Visual Studio 2019. Дана IDE є найпопулярнішим середовищем розробки різноманітних проектів у інфраструктурі .NET. Треба відмітити що дуже практичним є застосування Visual Studio 2019 разом з плагіном ReSharper від компанії JetBrains, який значно розширює функціонал програмного середовища.

Для написання клієнтської частини було обрано VS Code. Це дуже потужний редактор, який дозволяє зручно працювати з усіма типами файлів клієнтського додатку.VS Code має вбудовану консоль що дозволяє керувати сервером та підключенням бібліотек відразу з редактору.

Системою контролю версій було обрано GIT. Дана система контролю версій вбудовується у Visual Studio 2019, що дозволяє робити всю необхідну роботу з занесення нового коду до репозиторію не покидаючи середовище розробки. Це дозволяє не запускати окремих додатків, але без повністю построїти роботу без сторонніх додатків не вдасться, тому що деякий функціонал для роботи з GIT реалізовано не кращим образом.

Середовищем для роботи з базою даних була СУБД MS SQL Management Studio 17. Дана СУБД є доволі зручною та дозволяє у повній мірі працювати зі створенням бази даних та її редагуванням. Крім того використовуючи план запиту можна робити значну оптимізацію наявних запитів.

Для того щоб швидко відображати утворені маршрути на карті було вирішено використовувати Google Maps API та Directions API. Це дозволило побудувати швидку та ефективну систему відображення маршрутів.

4.2 Опис програмної системи

Говорячи про програмну систему можна виділити 2 основних компоненти: клієнтський додаток та серверний додаток.

На стороні серверу реалізований RESTful-сервіс, який виступає ендпоінтом для взаємодії клієнта з сервером. Кожен з методів уніфікований та має однаковий інтерфейс роботи, тому всі методи застосовують JSON для комунікації.

Головною ціллю розробки було отримати зручний додаток з яким буде легко працювати усім групам користувачів. Тому весь інтерфейс роботи построєний на принципі створення найпростішого інтерфейсу для ефективної роботи з веб-додатком

Веб-клієнт у повній мірі застосовує та імплементує основні принципи юзабіліті з погляду UI та UX. Коли користувач звертається до системи в перший раз, то він попадає на сторінку авторизації. (рис. 4.1). Для того щоб мати змогу скористуватися основним функціоналом додатку, користувач повинен ввести імейл та пароль.

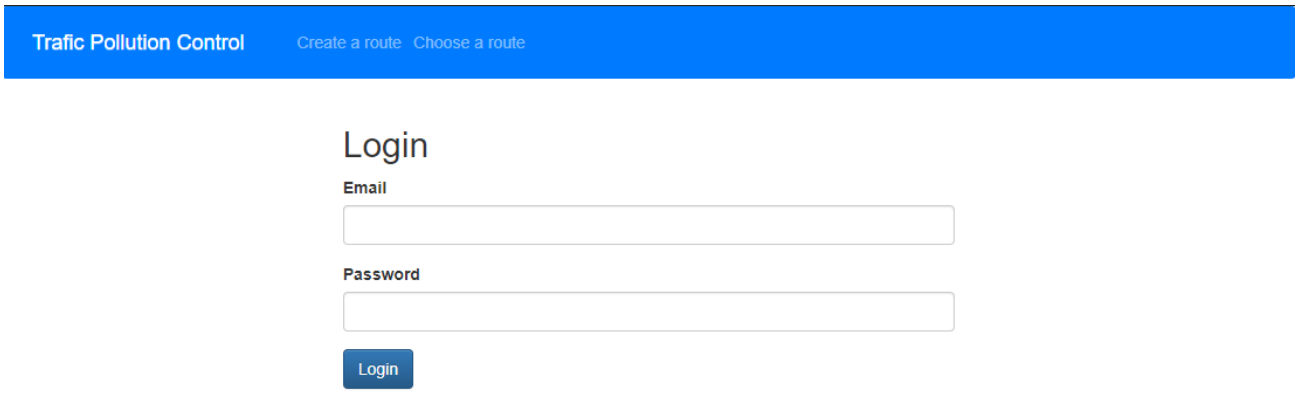


Рисунок 4.1 – Сторінка авторизації

Якщо у користувача немає облікового запису, то він має змогу зареєструватись, вказавши своє ім’я, імейл та пароль (рис 4.2). Треба зазначити що валідація даних реалізована, як на клієнті, так і на сервері. Валідація на клієнті дає змогу не тільки сповістити користувача у разі введення їм недопустимих даних або про помилку в них та й значно підвищує безпечність системи, а валідація на сервері не дасть змоги відправити штучні дані, які можуть бути зроблені з наміром злому тистеми [20].

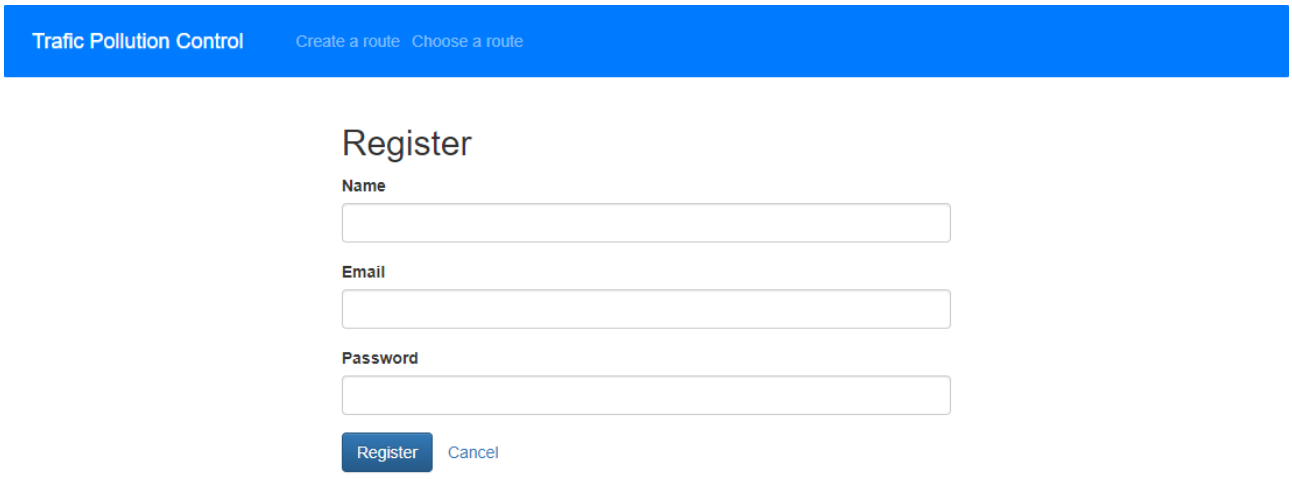
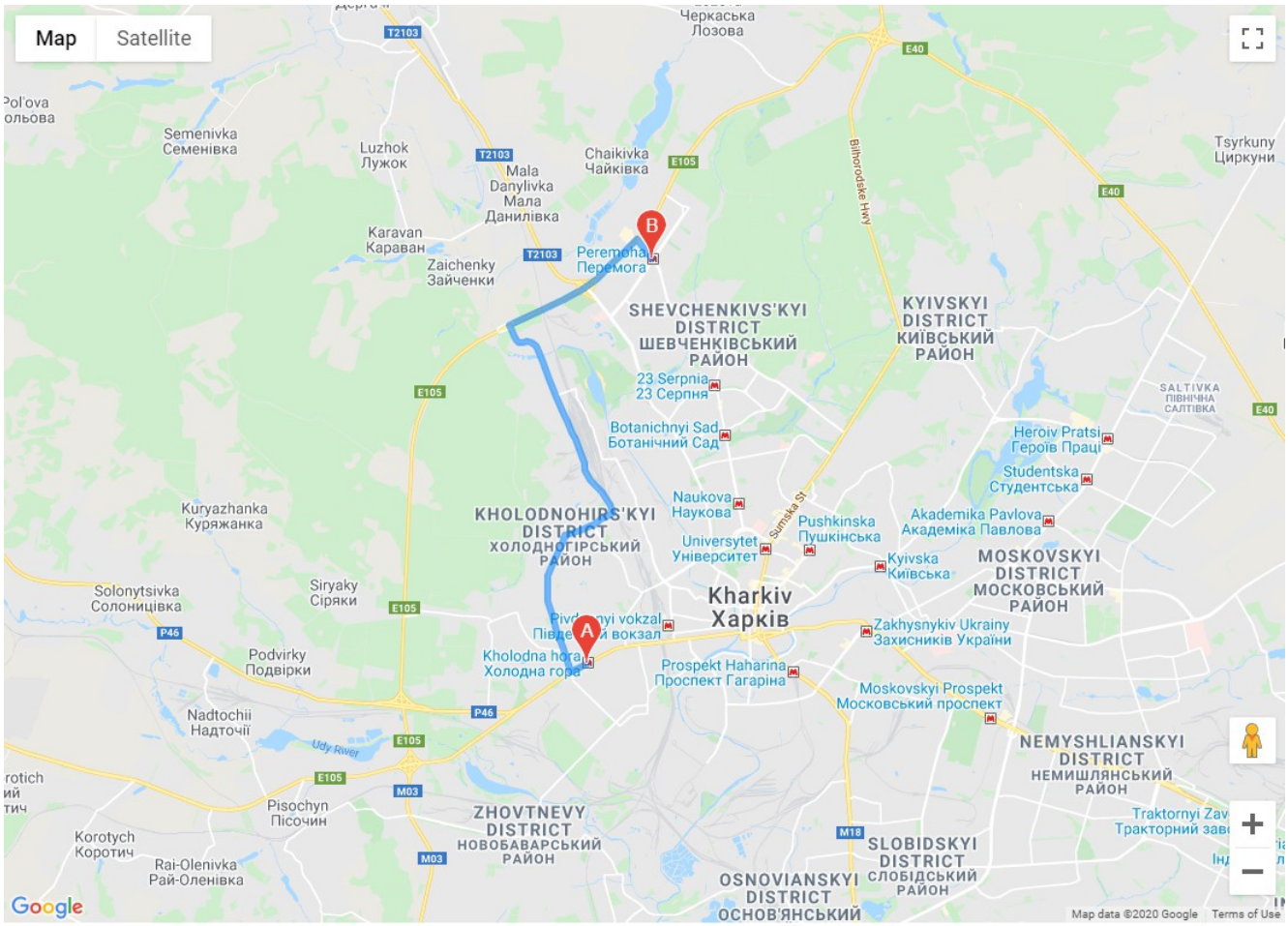


Рисунок 4.2 – Сторінка реєстрації

Після того, як користувач зміг пройти авторизацію він потрапляє на головну сторінку де відображується останній обчислений маршрут, який було розраховано до цього (рис 4.3).

Можна побачити, що користувач бачить маршрут оптимальний за двома показниками: за шляхом та кількістю викидів. Інформацію про дистанцію та викиди користувач може побачити у нижній часті екрану. Ці дані є динамічними, що означає що вони обчислюються кожен раз після зміни маршруту.

Одразу на цій сторінці користувач може послідовно клікнути на пункти з якого та до якого він хоче доїхати. Після цього на карті буде відображено новий маршрут.



Distance: 13.1 km

Pollution: 2.413712 g

Рисунок 4.3 – Головна сторінка

Оскільки, для відображення карти використовується карта від Google, то користувач має змогу змінювати масштаб, пересуватися по карті. Також присутня кнопка зміни режиму карти. За замовчуванням користувач бачить відображення у виді мапи, але можна перейти до режиму знімків зі супутнику.

ВИСНОВКИ

В ході науково-дослідницької роботи був проведений аналіз та досліджено існуючі програмні системи контролювання автомобільного трафіку у місті а також застосування цих систем у сучасному житті. З’ясовано як само такі системи використовуються у містах та у міжміських сполученнях, які дані та характеристики отримують такі системи. Визначено як вони застосовують отримані дані для розрахунку завантаження доріг, рівня забруднення та які рішення можна отримати базуючись на отриманих даних. Було досліджено як саме такі рішення застосовують теорію графів для роботи з маршрутами. Отримано які основні алгоритми роботи з ними використовують ці системи.

Окремо було досліджено як саме автомобілі забруднюють довколишнє середовище. Визначено, яка з речовин переважає у викидах та як можна робити підрахунок таких викидів. Додатково було визначено які с можливості розрахунку викидів автомобілів залежно від типу дороги, характеру місцевості, наявності трафіку та інших.

Було проведене дослідження існуючих алгоритмів пошуку у графах. Кожен з взятих алгоритмів було реалізовано у вигляді коду та досліджено за допомогою набору тестових даних. Після порівняльного аналізу та опису складності кожного з алгоритмів Було обрано найефективніший алгоритм роботи відповідно до предметної галузі.

Сукупність проведених досліджень та аналізу дозволила зробити відповідні висновки, які були використанні при проектуванні системи.

Як результат науково-дослідницької роботи було розроблено програмну систему, яка дозволяє обчислювати та будувати та зберігати найефективніший шлях пересування автомобілів відповідно до найменшої кількості шкідливих викидів.

Методи розробки базуються на інструментах розробки веб-застосувань на платформі .NET, протоколу передачі даних HTTP, фреймворків ASP.NET Core, та

ADO.NET для створення веб-застосунків на платформі .NET,технології AJAX з використанням фреймовку jQuery, текстового формату обміну даних JSON, базі даних SQL, середовищі розробки Microsoft Visual Studio 2019 та з використанням системи управління базою даних Microsoft SQL Server 2017. Також для відбраження карт було досліджено різні способи відображення карт та обрнано для застосування Google Maps API та Directions API для эфективного будування маршрутів.

Для зберігання графів був проведений аналіз наявних баз даних, для вибору тих, які можуть зберігати графи з відповідними характеристиками. Відповідно була спроектована та розроблена база даних, яка дозволяє зручно та ефективно працювати з графами.

Розроблена інформаційна система є багатофункціональною у використанні. Інтерфейс отриманого додатку є дуже зручним у використанні, що допомагає користувачу швидко зрозуміти як користуватися даною системою. Система має середні вимоги до апаратного та програмного забезпечення, забезпечує досить високу швидкість роботи, безпечна у використанні, що, безумовно, є явними плюсами цього додатка, полегшуючими його роботу на усіх етапах експлуатації.

Отримана система знаходиться у дуже важливій та популярній екологічній ніші. Актуальність отриманого додатку підтверджується великою зацікавленістю, як корпорацій так і урядів багатьох розвинутих країн, які зараз звертають багато уваги на проблеми великої кількості шкідливих викидів. Тому дана система є дуже корисною для вирішення екологічних проблем. Однак треба зазначити, що впровадження такої системи потребує значної підтримки, тому що для її роботи потрібно володіти великою кількістю інформації як про загальні характеристики доріг так і про завантаження доріг у реальному часі.

Система отримана у ході розробки задовольняє вимогам винесеним до неї у постановці задачі.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Medium [Електронний ресурс].2019. Дата оновлення: 07.04.2020. URL: https://medium.com/better-programming/5-ways-to-find-the-shortest-path-in-a-graph-88cfefd0030f (дата звернення: 03.05.2020).
2. Ріхтер, Джефрі CLR via C#. Программування на платформі Microsoft .NET Framework 2.0 на мові C#. Питер, 2007. - 656 с.
3. Робинсон, С.; Корнес, О.; Глинн, Д. и др. C# для профессионалов. Москва, 2005. - 396 с.
4. Adam Freeman Pro ASP.NET MVC 4. Apress, 2013. - 756 с.
5. Грабер М. SQL. СПб, 2003. – 644 с.
6. Чедвик Джесс , Снайдер Тодд , Панда Хришикеш ASP.NET MVC 4. Розробка реальних веб-додатків з використанням ASP.NET MVC. Вильямс - Москва, 2013. - 432 с.
7. Дейт, К.Дж. Введення в системи баз даних. Москва, 2005.- 1328 с.
8. Кренке, Д. Теорія і практика побудови баз даних. СПб, 2003. – 800 с.
9. Jonathan Mccracken Test–Drive ASP.NET MVC. Москва, 2010. - 250 с.
10. Молінаро, Э. SQL. Збірник рецептів. СПб, 2009. - 672 с.
11. Kirill, S., Pribylnov, D., Martovytskyi, V., Chupryna, A. Investigation of network infrastructure control parameters for effective intellectual analysis// 14th International Conference on Advanced Trends in Radioelectronics, Telecommunications and Computer Engineering, TCSET 2018 – Proceedings. 2018. P 983.

12. Arsenov A., Ruban I., Smelyakov K., Chupryna A Evolution of Convolutional Neural Network Architecture in Image Classification Problems// Selected Papers of the XVIII International Scientific and Practical Conference on IT and Security (ITS 2018).–CEUR Workshop Processing. 2018. P. 35.

13. Фаулер, М. UML. Основы [Текст] : пер. с англ. А.: Петухов/ М. Фаулер, К. Скотт. - СПб.: Символ, 2017. - 184 с.
14. W3Schools [Електронний ресурс].2019. Дата оновлення: 22.04.2020. URL: https://www.w3schools.com/angular/ (дата звернення: 25.04.2020).

15. Мюлер Джордж Р. Проектирование баз данных и UML – Москва: Лори, 2013. – 432 с.
16. Пугачев, С. Разработка приложений для Windows на языке C# [Текст] / С.В. Пугачов, А.М. Шеріев, К.А. Кичинський. – СПб.: В-Петербург, 2013. – 416 с.
17. Рихтер, Дж. CLR via C# Программирование на платформе Microsoft .NET Framework 4.0 на языке C# [Текст] / Дж. Рихтер, пер. с англ. Радченко И., Рузмайкина И. – СПб. Питер, 2013. – 428 с..
18. Шилдт, Г. C#4.0: Полное руководство [Текст] / Г. Шилдт. ; пер. с англ. И. Берштейн. – М.: ООО «И.Д. Вильямс», 2011. – 356 с.
19. Этапы разработки пользовательского интерфейса [Електронний ресурс] – Режим доступу http://www.4stud.info/user-interfaces/stages-of-development-user-interface.html
20. AllMath. Теорія графів та комбінаторика [Електронний ресурс]. 2016. Дата оновлення:07.04.2020.URL: http://www.allmath.ru/highermath/algebra/graph/graph6.htm (дата звернення: 03.05.2020).