

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ЗАПОРІЗЬКА ПОЛІТЕХНІКА»

Факультет комп'ютерних наук та технологій
Кафедра комп'ютерних систем та мереж

Пояснювальна записка

до дипломного проєкту (роботи)

магістра

(ступінь вищої освіти)

на тему КОМП'ЮТЕРНА СИСТЕМА ДЛЯ ВИРІШЕННЯ І ОПТИМІЗАЦІЇ
ЗАДАЧ ЛОГІСТИКИ З ВИКОРИСТАННЯМ ШТУЧНОГО ІНТЕЛЕКТУ

Виконав: студент 2 курсу, групи КНТ-612м
спеціальності _____

123 Комп'ютерна інженерія

(код і найменування спеціальності)

Освітня програма (спеціалізація)

Спеціалізовані комп'ютерні системи

СОЛОГУБОВ І. Я.

(ПРИЗВИЩЕ та ініціали)

Керівник СКРУПСЬКИЙ С.Ю.

(ПРИЗВИЩЕ та ініціали)

Рецензент КАПЛІЄНКО Т.І.

(ПРИЗВИЩЕ та ініціали)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет Комп'ютерних наук і технологій

Кафедра «Комп'ютерні системи та мережі»

Ступінь вищої освіти магістерський

Спеціальність 123 Комп'ютерна інженерія

(код і найменування)

Освітня програма (спеціалізація) «Спеціалізовані комп'ютерні системи»

(назва освітньої програми (спеціалізації))

ЗАТВЕРДЖУЮ

Зав. кафедри Кудерметов Р.К.



“ 24 ” жовтня 2023 року

З А В Д А Н Н Я
НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА

Сологубова Іллі Ярославовича

(ПРИЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Комп'ютерна система для вирішення і оптимізації

задач логістики з використанням штучного інтелекту

керівник проєкту (роботи) к. т. н., доцент, СКРУПСЬКИЙ Степан Юрійович

(науковий ступінь, вчене звання, ПРИЗВИЩЕ, ім'я, по батькові)

затверджені наказом вищого навчального закладу від “ 24 ” жовтня 2023 року № 400

2. Строк подання студентом проєкту (роботи) 10 грудня 2023 року

3. Вихідні дані до проєкту (роботи) існуючі алгоритми знаходження оптимального

шляху з використанням методів штучного інтелекту та без

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1) Аналіз технічного завдання, дослідження наявних методів пошуку оптимального

маршруту

2) Розробка структури та алгоритму роботи системи;

3) Реалізація та дослідження системи;

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

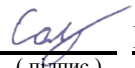
6. Консультанти розділів проєкту (роботи)

Розділ	ПРИЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-3	СКРУПСЬКИЙ С.Ю., доцент		
нормоконтроль	ЩЕРБАК Н.В., ст. викл.		

7. Дата видачі завдання 01.10.2023 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Аналіз наявних методів та алгоритмів пошуку оптимального шляху, огляд основних методів штучного інтелекту для вирішення задачі, огляд методів корекції помилок в нейронних мережах	08.10.2023 р.	
2	Розробка структури системи	16.10.2023 р.	
3	Розробка алгоритму роботи системи	07.11.2023 р.	
4	Реалізація системи	12.11.2023 р.	
5	Дослідження роботи алгоритму системи	14.11.2023 р.	
6	Оформлення отриманих результатів у ПЗ	19.11.2023 р.	
7	Оформлення графічного матеріалу	25.11.2023 р.	
8	Оформлення допоміжного матеріалу	26.11.2023 р.	

Студент  Ілля СОЛОГУБОВ
(підпис) (ім'я, ПРИЗВИЩЕ)

Керівник проєкту (роботи)  Степан СКРУПСЬКИЙ
(підпис) (ім'я, ПРИЗВИЩЕ)

РЕФЕРАТ

ПЗ: 89 с., 25 рис., 2 табл., 51 джерело.

ГЕНЕТИЧНІ АЛГОРИТМИ, ГЕОІНФОРМАЦІЙНА СИСТЕМА, КОМП'ЮТЕРНА СИСТЕМА, ЛОГІСТИКА, МЕТОДИ ШТУЧНОГО ІНТЕЛЕКТУ, ЗАДАЧА КОМІВОЯЖЕРА

Об'єкт дослідження – комп'ютерна система для вирішення логістичних задач, такий як знаходження оптимального маршруту та автоматизація доставки.

Предмет дослідження – використання штучного інтелекту та методів для знаходження оптимального маршруту між точками.

Мета роботи – проектування та аналіз комп'ютерної системи для автоматизації і оптимізації роботи сервісів доставки.

Матеріали, методи та технічні засоби: алгоритми пошуку оптимальних шляхів, методи штучного інтелекту для оптимізації задач. Мови програмування Python, TypeScript. Фреймворки React, NestJS, Tailwind, Flask. Бібліотеки Optuna, NumPy. Зовнішні сервіси FireBase, GoogleMaps, OpenStreet.

Результати – розроблено комп'ютерну систему з використанням методів штучного інтелекту для оптимізації і автоматизації бізнес-процесів пов'язаних з доставкою товарів. Проведено аналіз та тестування використання генетичних алгоритмів в комбінації з алгоритмами на графах для оптимізації задачі пошуку оптимальних шляхів.

Галузь використання – логістика.

ABSTRACT

Explanatory note to the master's work: 89 p., 25 figures, 2 tables, 51 sources.

GENETIC ALGORITHMS, GEOINFORMATION SYSTEM, COMPUTER SYSTEM, LOGISTICS, METHODS OF ARTIFICIAL INTELLIGENCE, TRAVELING SALESMAN PROBLEM

Object of research – computer system for solving logistic tasks, such as finding an optimal route and automating delivery.

Subject of research – the use of artificial intelligence and methods to find the optimal route between points.

Objective of the work – design and analysis of a computer system for automation and optimization of delivery service operations.

Materials, methods, and technical means: algorithms for searching optimal paths, artificial intelligence methods for task optimization. Programming languages Python, TypeScript. Frameworks React, NestJS, Tailwind, Flask. Libraries Optuna, NumPy. External services FireBase, GoogleMaps, OpenStreet.

Results – developed a computer system using artificial intelligence methods to optimize and automate business processes related to goods delivery. Analysis and testing of the use of genetic algorithms in combination with graph algorithms for optimizing the task of finding optimal paths.

Field of application – logistics.

ЗМІСТ

Перелік умовних позначень, скорочень, термінів	8
Вступ.....	9
1 Теоретичні відомості.....	10
1.1 Постановка предметної області	10
1.2 Алгоритми пошуку оптимального шляху і методи їх оптимізації	11
1.2 Метод гілок і меж	12
1.3 Метод динамічного програмування	14
1.4 Метод ближнього сусіда.....	15
1.5 Метод мінімального остовного дерева	17
1.6 Генетичні алгоритми.....	18
1.7 Метод імітації відпалу	21
1.8 Алгоритм Беллмана-Хелда-Карпа	23
1.9 Мурашиний алгоритм	24
1.10 Використання штучного інтелекту для вирішення задачі TSP	26
1.11 Висновки за розділом 1.....	29
2 Розробка системи	29
2.1 Вибір базової архітектури системи	30
2.2 Вибір архітектури для програмного забезпечення системи	35
2.3 Вибір архітектури клієнт-серверної взаємодії	41
2.4 Вибір інструментів для реалізації.....	54
2.4.1 Інструменти розробки сервісної частини	54
2.4.2 Інструменти розробки обчислювального сервісу	56

2.4.3 Інструменти розробки клієнтської частини.....	58
2.4.4 Зовнішні сервіси для розробки системи	61
2.5 Реалізація системи.....	62
2.6 Висновки за розділом 2.....	67
3 Аналіз та дослідження системи	68
3.1 Ідея використання мінімального остовного дерева.....	69
3.1.1 Знаходження меж оптимального рішення.....	69
3.1.2 Створення початкової сильної популяції	71
3.1.3 Мутація популяції	75
3.1.4 Модифікований відбір	76
3.2 Аналіз роботи системи і тестування.....	77
3.2.1 Використання ChatGPT	82
3.4 Висновки за розділом 3.....	83
Висновки	84
Перелік джерел посилання	85

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

- ШІ – Штучний інтелект;
- TSP – Travelling salesman problem (Задача комівояжера);
- MST – Minimum spanning tree (Мінімальне остовне дерево);
- ГА – Генетичний алгоритм;

ВСТУП

Логістика – світова галузь, яка була і залишається актуальною впродовж всієї історії людства. Дана галузь також охоплює частково інші галузі такі як військова справа, економіка, бізнес, муніципальне планування та інше.

Логістика має безлічі задач, такі як транспортування товарів, будування промислових ланцюгів, оптимізація складських запасів, маршрутизація пакетів в мережі, планування оптимального трафіку міст. Однак першою і основною задачею даної галузі залишається саме транспортування товарів. В сучасному світі питання транспортування товарів залишається актуальним, оскільки навантаження на неї стає все більшим кожного року, оскільки з ростом економіки і людських потреб, збільшується обсяг перевезень товарів, доставка сировини, тощо.

Виходячи з цього, однією з найпріоритетніших задач логістики - є пошук оптимального шляху. Оптимальний шлях може визначатися різними критеріями – найдешевше перевезення, найкоротший чи найшвидший шлях доставки необхідних товарів до пунктів призначення. Дана задача досі залишається актуальною, оскільки існує багато способів її вирішення, але жоден з них не є оптимальним алгоритмом її вирішення.

Враховуючи дані обставини і сучасний рівень розвитку штучного інтелекту (ШІ) – даний інструмент може мати суттєвий вплив на вирішення даної проблеми. ШІ здатний на обробку великого обсягу даних та більш глибокий аналіз даних який може будуватися на попередньому досвіді системи.

Проведене дослідження більш зосереджується саме на використанні штучного інтелекту, як одного із способів оптимізації під час вирішення задач пошуку оптимального шляху, а також комбінації з іншими методами для досягнення цієї ж цілі.

1 ТЕОРЕТИЧНІ ВІДОМОСТІ

1.1 Постановка предметної області

Весь світ будується на правильному використанні наявних ресурсів, вони можуть бути різними – товари масового збуту, сировина, їжа, зброя, енергоресурси, інформація та ін. Наявність ресурсів є лише одним із основних питань які ставить перед собою держава чи якесь підприємство, не менш важливим є питання доставки цих ресурсів до місць їх обробки, створення оптимальних ланцюгів виробництва.

Зі зростанням обсягів виробництва, глобалізацією та підвищення транспортних можливостей, особливо актуальним стає питання знаходження оптимальних шляхів постачання ресурсів. Дана задача вирішується людьми ще з часів зародження цивілізації, однак чітко сформулювати її стало можливим після відкриття розділу математики – теорії графів і створення на її основі найвідомішої задачі – задача комівояжера (travelling salesman problem або TSP). Метою даної задачі є знаходження оптимального маршруту для всіх місць призначення з умовою їх досягнення хоча б один раз.

Створювана комп'ютерна система – в основі являє собою алгоритм для вирішення задачі TSP з використання різних методів оптимізації та штучного інтелекту. Дана система може мати будь який вигляд і використовуватися для абсолютно будь яких цілей, як то доставка товарів від складу до всіх клієнтів так і маршрутизації пакетів у мережі.

Дана система створюється в першу чергу для вирішення питань доставки фізичних товарів від місця вивозу товарів до точок призначення. Основними складовими даної системи – є база даних, алгоритм який вирішує поставлену задачу, інтерфейс вводу та виводу даних для користувача та сервіс навігації.

База даних зберігає два типи інформації – користувацьку та внутрішню (системну). Користувацька – це той тип інформації який користувач вводить залежно від своїх потреб – склади підприємства, транспорт, замовлення (точки

призначення) та ін. Системна – використовується для потенційного навчання системи та кешування деяких даних для покращення її працездатності у майбутньому.

Алгоритм – програмний код що реалізує обробку введених даних від користувача чи даних які надійшли з інших джерел та видає необхідний результат.

Інтерфейс дозволяє користувачу надавати дані що необхідні системи для обробки та бачити результат, та отримувати візуальний результат обробки, для подальшого його використання.

Сервіс навігації – це може бути будь який зовнішній сервіс який буде використовуватися працівниками для візуального відображення шляху який їм необхідно пройти та підтримувати курс на всьому шляху.

Основною проблемою даної комп'ютерної системи буде саме знаходження алгоритму чи їх комбінацією разом з навчанням моделі штучного інтелекту, який буде видавати результат за оптимальний час. Це є основною проблемою оскільки дана задача в математиці досі не вирішена і на разі є невирішуємою і всі методи для знаходження точного рішення зводилися до перебору [1].

Наступною проблемою може бути здатність систем до масштабування. У разі виникнення потреби модифікації алгоритму з введенням додаткових параметрів, додаткового функціоналу чи розширенням умови задачі, може виникати проблема неактуальності знань в моделі ШІ, оскільки вона буде навчена на той час для вирішення інших задач і її необхідно буде перенавчати.

1.2 Алгоритми пошуку оптимального шляху і методи їх оптимізації

Задача комівояжера належить до задачі з множини NP-повних, до цієї групи задач належать ті, що не мають оптимального алгоритму для їх вирішення і єдиним відомим способом їх вирішення є повний перебір усіх можливих відповідей. Дана задача не є виключенням, вона також не має алгоритму і базовий алгоритм для її

вирішення має асимптотичну складність $O(n!)$, що означає повний перебір усіх можливих маршрутів і вибір з них найкоротшого. [2].

Дана проблема змусила вчених, математиків, а потім і програмістів винайти декілька способів оптимізації та просунутих алгоритмів перебору. Жоден з них не є оптимальним, але дозволяє працювати з більшими даними, що часто задовольняє реальні проблеми. Найвідоміші з таких алгоритмів та методів пошуку [3,4,5]:

- метод гілок та меж;
- метод динамічного програмування;
- метод ближнього сусіда;
- метод мінімального остового дерева;
- генетичні алгоритми (ГА);
- метод імітації відпалу;
- алгоритм Беллмана-Хелда-Карпа;
- мурашиний алгоритм.

1.2 Метод гілок і меж

Даний алгоритм є оптимізаційним, в його основі залишається перебір, але за допомогою деяких кроків можна зменшити кількість варіантів, відкинувши заздалегідь неоптимальні. Алгоритм надає точний результат, оскільки є одним з алгоритмів перебору.

Даний метод може бути використаний не тільки для вирішення задач теорії графів, але й інших задач, однак сам алгоритм для задачі комівояжера представлений алгоритмом Літтла: В кожній строчці матриці необхідно знайти мінімальний елемент і його відняти від всіх елементів цієї строки, так само зробити для стовпців. Таким чином отримується зведена матриця. Віднімаючи однакові значення від шляхів що ведуть від однієї вершини в інші, результат ніяк не зміниться, однак це дозволяє привести матрицю у більш легкий вигляд для

обробки. Після чого ми маємо просумувати мінімальні елементи щоб отримати константу яка буде являти собою нижню границю всіх множин маршрутів. Далі необхідно для кожного нуля визначити його ступінь, який являє собою суму мінімальних значень строки і стовпця в яких він знаходиться. Після цього необхідно буде взяти ту дугу (нуль) який буде мати максимальний ступінь. Дерево рішень буде розбито на дві складові – одна з гілок та де ми беремо ребро у рішення, друге рішення – ребро ми не беремо. Далі ступінь додається до того піддерева де ребро не було взяте, для взятого ребра вираховується границя з нової таблиці де видаляється строка і стовбець, в якому знаходиться дуга, а симетричне значення змінюється на знак нескінченності. Алгоритм проводиться до тих пір доки не буде знайдено рішення чи не буде остаточно відомо що рішення не існує [6, 7].

На рисунках 1.1 та 1.2 наведені приклади побудованого дерева за методом гілок і меж та приведеної матриці відповідно.

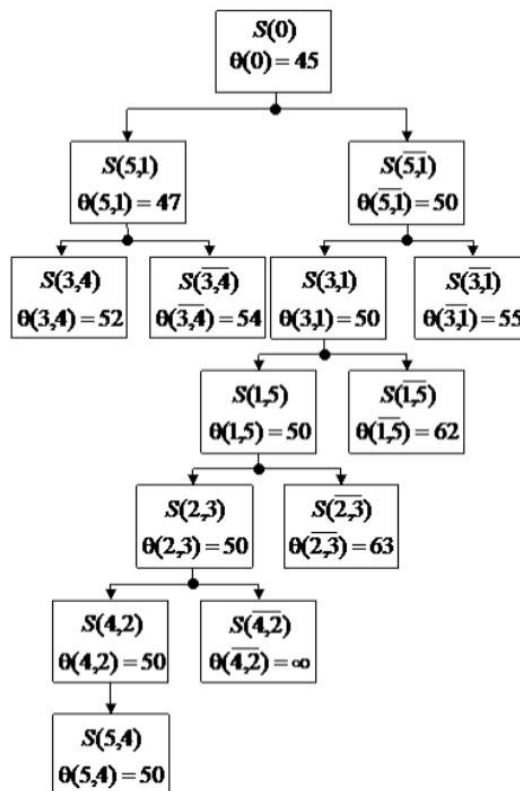


Рисунок 1.1 – Приклад вигляду дерева, за методом гілок і меж

	1	2	3	4	5	α_i		1	2	3	4	5
1	∞	14	9	16	7	7	1	∞	7	2	9	0
2	20	∞	9	19	14	9	2	11	∞	0	10	5
3	18	15	∞	12	12	12	3	6	3	∞	0	0
4	23	10	13	∞	17	10	4	13	0	3	∞	7
5	7	6	6	6	∞	6	5	1	0	0	0	∞

Рисунок 1.2 – Приклад приведення матриці вартості шляхів

Асимптотичну складність алгоритму неможливо порахувати, оскільки виходячи з алгоритму він являє собою такий самий перебір, однак він дозволяє заздалегідь відкинути неоптимальні рішення і достатньо сильно спрощує його. Швидкість так само залежить від кількості вхідних даних, на поганій виборці алгоритм може бути так само повільний.

1.3 Метод динамічного програмування

Метод динамічного програмування має просту основу, його суть полягає у тому що обрати місто і будувати шляхи до всіх інших, поступово записуючи у пам'ять значення мінімальних шляхів маршрутів від міста до інших. Таким чином наприклад при зустрічі 5 точок для яких ми рахували мінімальний шлях, ми можемо взяти цей результат і пропустити розрахунок [8].

Таким чином даний алгоритм пропонує швидкість виконання в обмін на суттєве використання пам'яті. У формулах 1.1 та 1.2 наведені відповідно асимптотична складність алгоритму та використання пам'яті.

Асимптотичну складність представлено формулою 1.1:

$$O(n + \frac{(n-1)}{2} \cdot (n-2) \cdot 2^{(n-2)}) \quad (1.1)$$

Використання пам'яті представлено формулою 1.2:

$$\sim 2^n \sqrt{\frac{2n}{\pi}} \quad (1.2)$$

На рисунку 1.3 продемонстровано графік залежності часу виконання алгоритму від кількості вершин у вхідних даних.

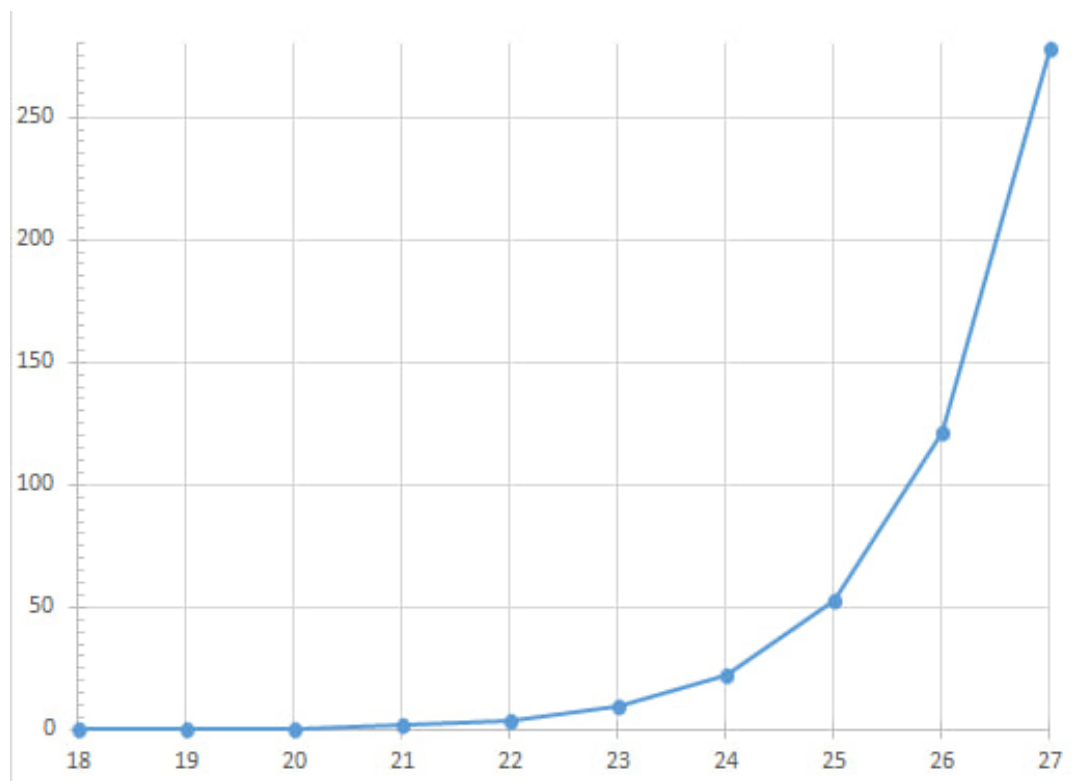


Рисунок 1.3 – Графік швидкості виконання алгоритму у секундах

1.4 Метод ближнього сусіда

Один з найпростіших алгоритмів для пошуку оптимального шляху, його ідея полягає в обході графу включаючи на кожному вершину графа, яка є найближчою відносно останнього обраного пункту призначення серед усіх вершин які ще не були включені в рішення задачі. Оскільки алгоритм на кожному кроці бере найкоротший шлях до інших вершин, він належить до категорії жадібних алгоритмів.

Жадібний алгоритм не надає точного рішення для задач TSP, однак іноді його може бути достатньо і його простота в комбінації з лінійною складністю. Для перевірки оптимальності рішення можна використовувати декілька. Одним зі способів може бути згаданий вище метод пошуку нижньої границі. Також одним із способів є порівняння шляху пройденого на останніх кроках і на початкових, якщо їх можна зіставити то можна вважати що шлях обрано оптимальний [9, 10].

В найпростішому вигляді кроки алгоритму наступні:

- побудувати граф і все вершини позначити білими;
- вибрати початкову вершину від якої алгоритм почне будувати маршрут і позначити її чорною;
- вибрати іншу найближчу білу вершину і позначити її чорною як пройдену;
- повторювати кроки 2 та 3 до тих пір доки в графі всі вершини не будуть помічені як пройдені.

Результатом буде послідовність вершин яка буде представляти маршрут згідно жадібного алгоритму.

Найочевиднішими проблемами даного алгоритму є те що для графу з 4 вершин можна завжди знайти такі вхідні дані за якими жадібний алгоритм буде давати найгірше рішення. Одним з найчастіших прикладів неоптимального розрахунку є наявність петель чи перехресть [11]. Приклад подібної ситуації продемонстровано на рисунку 1.4.

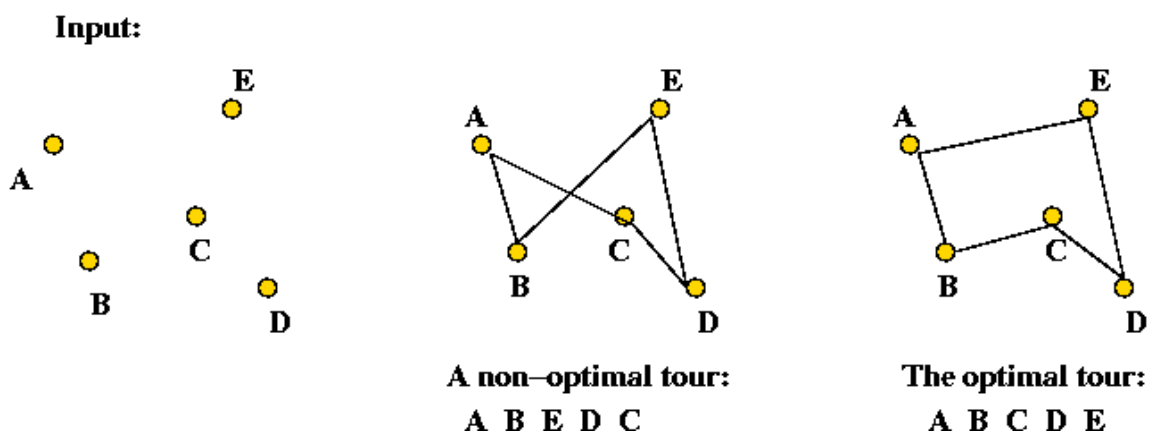


Рисунок 1.4 – Приклад неоптимального рішення жадібним алгоритмом

Для вирішення цього можна використовувати декілька методів визначення перехресть, що можуть представляти собою звичайний геометричний метод де знаючи координати вершини можна знайти перетин двох відрізків якщо він є, або якщо використовується сервіс для навігації на карті, то він може показати дану інформацію. Коли знайдено перехрестя, можна спробувати змінити порядок вершин і спробувати його вирішити. Даний спосіб також може не давати оптимального рішення, однак збільшить вірогідність того що побудований маршрут буде оптимальним.

1.5 Метод мінімального остовного дерева

Даний алгоритм будується на ідеї того що можна розглядати в якості мінімального шляху мінімальне остовне дерево – це такий граф в якому немає розривів і всі N вершин з'єднані між собою $N-1$ ребрами, будь яке видалення ребра розриває граф. Особливістю такого графа і відповідно маршруту є те що він хоч і не є циклічним, але якщо побудувати маршрут його обходу до довжина шляху буде гарантовано не більш ніж в двічі більшою за мінімальний шлях, оскільки при такому обході кожне ребро проходиться максимум 2 рази [12].

Пошук мінімального остовного дерева є простою задачею яка має не таку погану асимптотичну складність як перебір. Базова складність наведена у формулі 1.3, покращена асимптотична складність з використанням структури даних Union Find наведена у формулі 1.4.

$$O(N^2 + M \log(N)) \quad (1.3)$$

$$O(M \log(N)) \quad (1.4)$$

На рисунку 1.5 продемонстровано приклад мінімального остовного дерева на якому можна спробувати обійти цей граф і переконатися що шлях не буде гірший за подвоєний мінімальний. Також цю властивість можна використати як один із способів розрахунку верхньої межі пошуку шляху, тобто такої довжини шляху перевищення якої буде гарантувати що цей шлях точно вже не є оптимальним і за рахунок цього скорочувати кількість ітерацій в інших алгоритмах.

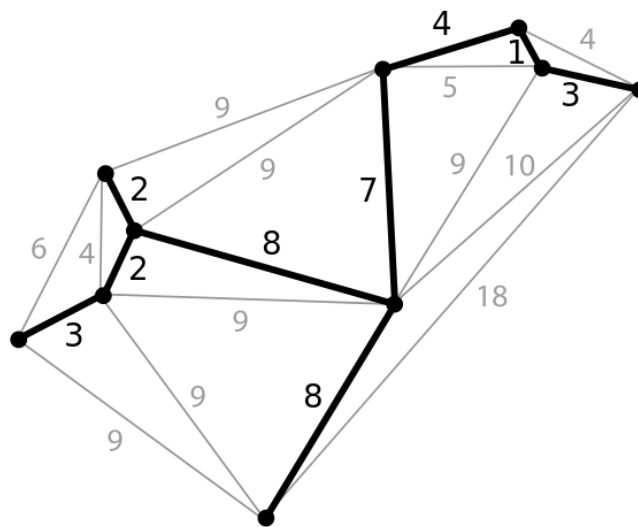


Рисунок 1.5 – Приклад мінімального остовного дерева

Даний алгоритм можна розглядати як допоміжний чи відправну точку для вирішення задач, наприклад для подальшої оптимізації можна спробувати використати алгоритм Ліна-Кернігана ідея якого полягає у тому щоб робити підміни маршрутів і оптимізацію на деяких його сегментах, таким чином покращуючи вже існуючий маршрут

1.6 Генетичні алгоритми

Дані алгоритми використовують в своїй основі еволюційний метод відбору рішень. Ці алгоритми використовуються у випадку якщо кількість можливих

рішень занадто велика для їх тестування, що для задачі комівояжера в даному випадку є також цілком виправданим [14].

Щоб почати роботу за даним алгоритмом, треба визначитися з тим як представити рішення у вигляді списку чисел чи як їх називають у генетичному алгоритмі – набором генів. Деяка кількість рішень буде називатися популяцією які потім повинні будуть між собою конкурувати за допомогою різних методів з метою “вижити” під час еволюції та “еволюціонувати”. Відповіді що залишаться можна буде вважати оптимальними розв’язками задачі чи хоча б наближеними до оптимального рішення (маршруту).

Після цього на наступних етапах необхідно визначитися з функцією пристосованості для задачі комівояжера вона може бути простою, це може бути коефіцієнт який буде визначатися як $1 / (\text{довжина маршруту})$. Чим нижче буде довжина тим краще і тим більше значення буде у функції пристосованості.

Після того як визначено функцію та визначений спосіб побудови геному – можна починати алгоритм. На першому кроці обираються батьки, цей вибір може бути випадковим, але може бути вже на цьому кроці оптимізованим якщо наприклад відома деяка межа, відстань гірше за яку відстань можна не розглядати, наприклад як згадано це було при розгляді методу мінімального остовного дерева. Далі відбувається схрещування яке може відбуватися за допомогою деяких генетичних операторів таких як кросовер чи мутації. Під час них геноми нащадків можуть змінюватися згідно правил схрещування, а також з деякою вірогідністю мутувати.

Процес схрещування двох обраних геномів батьків можна описати наступним алгоритмом [15]:

- обирається точка розриву;
- записуються гени першого батька до точки розриву у новий геном;
- записуються гени другого батька після точку розриву якщо не були успадковані;
- якщо гени нащадка не заповнені, то необхідно взяти неуспадковані гени першого батька після точки розриву;

- друга частина геному заповнюється аналогічно крокам 2-4, за винятком того що у кожному кроці метод вибору геномів батьків протилежний.

Приклад схрещування геномів наведено на рисунку 1.6.

0	3	1	4	2	20
0	2	4	3	1	11
0	3	4	1	2	18
0	2	1	4	3	18

Рисунок 1.6 – Приклад схрещування геномів

Процес мутації може мати будь який вигляд, але його алгоритм полягає у тому що спочатку відбувається перевірка шансу на проведення мутації і якщо вона має бути проведена, то у нащадка змінюються два гени місцями чи один ген змінюється випадково на інший, після чого проводиться перерахунок функції пристосованості.

Після того як процес розмноження закінчено проводиться оцінка всіх функцій пристосованості нащадків і за методами відбору визначається які нащадки залишаться для подальшої еволюції, а які вийдуть з цього процесу. Існує декілька основних методів відбору:

- турнірний (Tournament Selection) – суть його полягає у тому щоб вибрати з популяції деяку групу нащадків і порівняти їх функції пристосованості, найкраще значення функції вказуватиме на те який індивід буде батьком для наступного етапу еволюції;

- рулетка (Roulette Wheel Selection) – схожий на турнірний, але його відмінність полягає у тому що вибір наступних батьків стає ймовірнісним. Чим

краще значення функції пристосованості – тим вище вірогідність того що певний нащадок стане наступним батьком;

- селекція за рангом (Rank-Based Selection) – нащадки сортуються за значеннями своїх функцій і кожному надається свій ранг, після чого вони обираються за рангом. Корисно для випадків коли з’являються рідкісні еволюції зі значеннями що сильно відрізняються і це може завадити еволюції в певних моментах процесу;

- селекція по долям (Stochastic Universal Sampling) – відрізняється від селекції рулеткою лише тим що батьки обираються з деяким заданим інтервалом, щоб зменшити вплив фактору випадковості;

- селекція відсівом (Truncation Selection) – всі індивіди сортуються за значеннями функції пристосованості і лише найкращі обираються для подальшої еволюції, в той час як інші з неї виходять.

Процес еволюції відбувається до тих пір доки нова популяція не буде відрізнятися від іншої. Отримане найкраще значення функції пристосованості у одного з нащадків що залишився і буде рішенням задачі.

1.7 Метод імітації відпалу

Метод який побудовано на фізичних властивостях кристалізації речовин. В металах припускається що атоми виставлені у кристалічну решітку в якій допустимі переходи окремих атомів з однієї комірку в іншу. Цей процес відбувається поступово і ймовірно при пониженні температури, до того ж вірогідність переходу зменшується зі зменшення температури. В такому положенні системи завжди намагається дійти до глобального мінімуму, оскільки стійка решітка буде дорівнювати мінімальній активності атомів, оскільки атом намагається перейти у стан з найменшим рівнем енергії. Тобто метал поступово

нагрівається і охолоджується до тих пір доки не буде створена стійка кристалічна структура. На рисунку 1.7 відображено візуальне представлення методу відпалу.

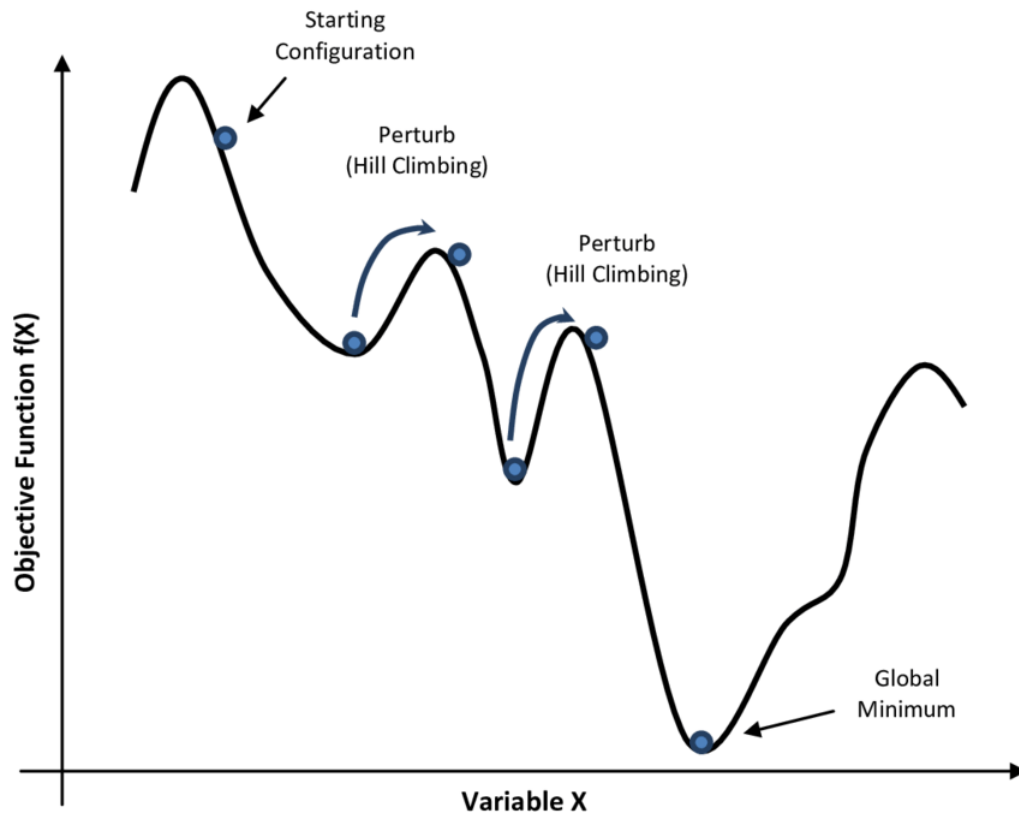


Рисунок 1.7 – Візуальне представлення методу відпалу

Загальний алгоритм в контексті програмування та вирішення задачі комівояжера наступний:

- обирається випадковим чином початковий стан системи, тобто перший маршрут в якому;
- проводиться оцінка вартості маршруту відповідно сумуючи усі ребра які з'єднують вершини на шляху;
- генерується випадковим чином новий алгоритм, вносячи невеликі зміни у поточний маршрут. Логіка генерації може бути будь якою, але в простому вигляді це може бути перестановка пари міст місцями;
- обчислення нової вартості маршруту;
- нова вартість порівнюється з поточною і якщо нова вартість менша то воно приймається як поточне рішення, у протилежному випадку приймається з

ймовірністю що зменшується зі збільшенням різниці в вартості та з підвищенням температури в даній системі. Таким чином ймовірнісним підходом алгоритм намагається позбутися застрягання у локальному мінімумі який може не задовольняти вимогу задачі;

- проводиться зменшення температури, це параметр який контролює ту саму ймовірність переходу атому (рішення);

- повторення кроків 3-6 у циклі доти, доки температура системи не досягне нуля;

- у системи залишається найкраще рішення яке і є потенційною відповіддю до задачі.

Даний метод гарний тим що дозволяє таким процесом охолодження спробувати уникнути локальних мінімумів на яких можуть часто застрягати інші оптимізаційні методи. На жаль метод не є повноцінним алгоритмом для вирішення задачі також і використовується лише як оптимізаційний метод з намаганням наблизитися до оптимального рішення, але іноді може знайти саме правильне рішення.

Дана стохастична модель процесу нагадує генетичний алгоритм і його так само можна налаштовувати змінюючи параметри температури і формули за якою розраховується ймовірність переходу з одного рішення в інше, іноді такі задачі на пошук оптимальних параметрів можуть бути вирішені штучним інтелектом.

1.8 Алгоритм Беллмана-Хелда-Карпа

Алгоритм динамічного програмування що може насправді працювати у доповненні з іншим алгоритмом. Його ідея будується на просто правилі, що локальний мінімум для під графа є частиною цілого рішення яке складається з таких самих локальних мінімумів. Один з алгоритмів що надає реальне рішення задачі.

Його алгоритм у загальному вигляді наступний [16]:

- ініціалізація графу – створюється матриця графу, що описує всі відстані між вершинами в ньому і встановлюється спеціальний коефіцієнт N – кількість вершин для яких починається пошук;
- для кожної підмножини з N вершин знаходиться рішення задачі. Цей крок може бути виконаний будь яким алгоритмом який буде обрано розробником. В базовому вигляді береться звичайний неоптимальний перебір;
- кількість вершин N подвоюється;
- кроки 2 та 3 виконуються до тих пір доки кількість вершин не перевищить кількість вершин у всьому графі;
- обирається мінімальне рішення серед знайдених.

Даний алгоритм лише представляє з себе оптимізаційний метод який дозволяє запам'ятати локальні рішення для певних підграфів і використовувати їх аналогічно звичайному алгоритму динамічного програмування, лише з тією різницею що рішення знаходиться одразу до підграфів і потребує менше пам'яті. Асимптотична складність даного алгоритму залишається експоненційною у його базовому вигляді і не є доцільною до використання на великих масивах даних, однак може бути використаний як основа для оптимізації якогось іншого алгоритму.

1.9 Мурашиний алгоритм

Один з метаевристичних алгоритмів який так само не надає точного рішення, однак може надати наближене до оптимального. Даний алгоритм є універсальним оскільки дозволяє знаходити маршрути для будь яких задач у тому числі і задачі комівояжера і будується він на ідеї того, як мурахи будують власні шляхи до їжі від мурашника.

В природі мурахи коли бачать їжу починають рух без чіткого уявлення щодо його оптимальності, вони роблять це різними шляхами і за собою після кожного проходу вони залишають феромони. Чим коротше шлях який проходить мураха до їжі і до мурашника назад, тим більше разів мураха може пройти по цьому шляху і тим більшу кількість феромону вона залишає. Шлях на якому буде залишено більше всього феромону буде приваблювати запахом інших мурах і вони почнуть рух по ньому до їжі. Таким чином після декількох заходів мурахи залишають для себе лише один найоптимальніший шлях руху серед тих що спробували. Цей шлях може не бути гарантовано рішенням такої задачі, однак такий маршрут буде максимально наближений до реального рішення [17].

Загальний алгоритм для вирішення задачі:

- кожна мураха повинна почати виконувати обхід графу переходячи від своєї початкової вершини до іншої на основі вірогідності, яка залежить від сили запаху феромону на ребрах відповідно. Також мураха починає таким чином балансувати між розвідкою нового шляху (ребро без феромону) та використанням пройдених шляхів (ребра з феромоном);

- після того як мурахи обійшли граф, порівнюється кожне рішення з досягнутих. На основі якості даних рішень наносить феромон на відповідні маршрути і кожне ребро таким чином отримує певну вагу. Ця інформація має бути передана кожній мурасі в системі на основі якій буде виконуватися перший крок алгоритму;

- крок 1 та 2 повторюється до тих пір доки не буде досягнутий певний критерій зупинки алгоритму і це може бути наприклад відсутність якоїсь зміни між циклами за аналогією як це відбувається в ГА;

- серед знайдених маршрутів обирається найкоротший який і приймається за рішення задачі.

Таким чином алгоритм на відміну від інших намагається балансувати між тим аби знаходити нові рішення, експериментувати і експлуатувати вже знайдені рішення, тобто локальні оптимуми як в алгоритмі Беллмана-Хелла-Карпа. Даний алгоритм так само може налаштовуватися оскільки основну роль відіграє саме

формула щодо прийняття ймовірного рішення щодо розвідки та використання старого шляху.

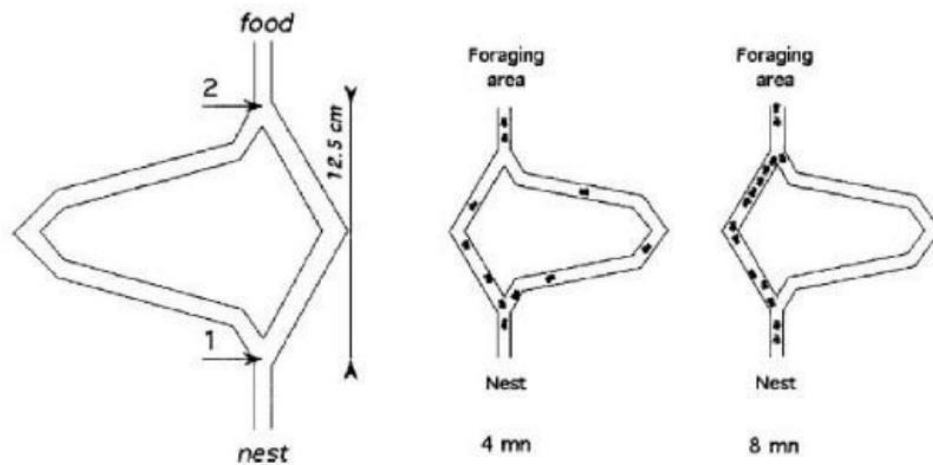


Рисунок 1.8 – Візуальне представлення руху мурах від гнізда до їжі

На рисунку 1.8 продемонстровано як мурахи в колонії в залежності від довжини шляху і частоти обходу певного шляху починають обирати шлях. Мурахи що йшли коротким шляхом залишали за собою більше феромонів і приваблювали інших мурах в колонії тому з часом всі почали йти саме ним.

1.10 Використання штучного інтелекту для вирішення задачі TSP

Серед зазначених алгоритмів 2 алгоритми – генетичний і мурашиний відносяться до алгоритмів штучного інтелекту, однак також до них можна частково віднести ще алгоритм відпалу. Особливість даних алгоритмів полягає у тому що вони всі мають стохастичну модель і побудовані на ідеї прийняття рішень чи адаптації і кожен з них може бути налаштований в залежності від того як саме буде виставлена вага коефіцієнтів.

Однією з особливостей ШІ є здатність навчатися та набувати досвід. Якщо брати до уваги людський досвід і здатність його мозку до вирішення певних задач

то можна одразу зрозуміти, що наш мозок наприклад бачучи безліч кількості точок, при спробі побудувати оптимальний маршрут – обере рішення що попаде в 5-10% найкращих рішень. Таким чином наш людський розумі вже має в собі алгоритм до відсіву 90% відсотків рішень. Дана особливість полягає у тому що ми завчасно бачимо неоптимальні шляхи, наприклад ті що створюють багато перехресть петель чи наприклад починають рух від першого міста до останнього і потім знову до одного з тих що стоїть поруч з першим.

В контексті алгоритмів, наприклад мурашиного та генетично, ШІ може почати набувати досвіду від кількості вирішених задач змінюючи в ту чи іншу сторону коефіцієнтів в залежності від того як змінюється рішення задачі підтягуючи їх до оптимального значення. Для цього стають у нагоді декілька методів машинного навчання.

Існує декілька способів машинного навчання, найпоширеніші – навчання з вчителем та без. Навчання без вчителя гарно підходить для задач кластеризації чи класифікації коли надається група об'єктів і ШІ має згрупувати всі об'єкти за якимись критеріями і визначити самі критерії розподілу. Для задачі пошуку оптимального шляху більш корисним є спосіб навчання з вчителем коли моделі що навчають задають якусь ситуацію, набір даних і також дають рішення до якого вона має максимально наблизитися. Таким чином можна спробувати навчити модель для вирішення задачі комівояжера. Основними цілями даного навчання будуть знаходження коефіцієнтів для таких алгоритмів як – генетичний, мурашиний та відпалу. Також таке навчання може бути використано ще для інших задач, таких як визначення стартової групи популяції [18].

Двома з основних методів машинного навчання з вчителем, для TSP, є [19]:

- метод корекції помилки – ідея цього методу полягає у тому щоб залишати зв'язки між нейронами до тих пір доки їх реакція залишається вірною і приводить до потенційного правильного рішення. У випадку якщо стається помилка то вага зв'язку змінюється на 1 в залежності від знаку помилки в ту чи іншу сторону. Також існує декілька модифікації такого методу - метод корекції помилок без квантування, метод корекції помилок із квантуванням, метод корекції помилок із

випадковим знаком підкріплення, метод корекції помилок із випадковими обуреннями;

- метод зворотнього поширення помилки (back propagation) – більш складний механізм виправлення помилок основною ідеєю якого є поширення помилки у зворотньому напрямку від виходів системи до входів. Даний метод дозволяє більш точно виставити вагу для кожного нейрону.

Загальний алгоритм backpropagation:

- ініціалізується нейронна мережа з випадковими вагами, які будуть в процесі навчання згідно алгоритму модифікуватися. На цьому етапі також треба налаштувати крок збіжності який в майбутньому має бути використаний для зміни ваг після отримання помилки;

- виконання обчислення нейронною мережею у прямому вигляді, тобто від її входів до виходів. Для кожного нейрону буде обчислюватися значення згідно його функції активації;

- розрахунок локального градієнту, який вираховується з помилки. Сама помилка розраховується як різниця між поточним значення фінальної функції активації і очікуваним;

- проводиться перерахунок ваг всіх нейронів, обчислюватися нова вага буде як добуток кроку збіжності, локального градієнту та відповідного вихідного сигналу нейрону. Перерахунок йде до вхідних вершин;

- алгоритм повторюється до тих пір доки помилка в мережі не буде стабілізована і не задовольнить умові зупинки.

Таким чином нейронна мережа може налаштовувати свої ваги до тих пір доки не знайде оптимальні значення що будуть задовольняти умовам задачі.

Основним методом для вирішення задачі комівояжера є метод зворотнього поширення помилки оскільки він надає змогу більш точно збалансувати вагу.

Одним з цікавих способів вирішення задачі за допомогою ШІ може бути використання ChatGPT який не є спеціалізованою моделлю, однак може мати достатньо досвіду хоча б для знаходження локальних оптимумів чи інших елементів.

1.11 Висновки за розділом 1

Було проведено аналіз основної проблематики вирішення задачі комівояжера. Проведено аналіз існуючих методів оптимізації та вирішення даної задачі, в яких було досліджено їх основні особливості, ідеї, переваги та недоліки. В результаті цього було виявлено основні можливості для оптимізації рішення задачі, а також основні ідеї, методики якими користувалися розробники нині для вирішення задач TSP. Дана інформація дає можливість для проведення експериментів і спробу створення алгоритму, який використовуючи різні методи рішення, їх комбінації може знайти оптимальне рішення задачі комівояжера за оптимальний час.

Також було проведено аналіз алгоритмів і методів ШІ для вирішення даної задачі що дозволяє розширити можливості звичайних алгоритмів і покращити вирішення задачі в поєднанні з ними. Були досліджені основні можливості які надає ШІ, методи мишиного навчання та корекції помилок.

2 РОЗРОБКА СИСТЕМИ

Згідно з поставленими вимогами до створюваної системи, необхідно створити інтерфейс, обчислювальний модуль, серверну частину для комунікації інтерфейсу з логікою системи та обрати необхідний інструмент для маніпуляції даними на мапі. Також необхідно визначити архітектуру системи за якою вона буде має бути побудована, а саме логіка взаємодія системи і користувача, а також взаємодії її окремих компонентів. Дана система є розподіленою, тому необхідно обрати архітектуру клієнт серверної взаємодії і інструмент що дозволять реалізувати її. Для окремих компонентів системи також визначається свій інструмент.

Поняття архітектури в залежності від контексту може визначати спосіб взаємодії і організацію коду, модулів окремих частин системи. Це може бути архітектура взаємодії сервера та клієнтської частини, архітектура організації коду щоб розділити логіку системи на шари і підвищити масштабованість додатку, архітектура взаємодії окремих модулів всередині системи та ін.

2.1 Вибір базової архітектури системи

Першим кроком необхідно обрати архітектури взаємодії всіх частин системи. Існує два основні підходи до реалізації системи – монолітна (одне ціле) та мікросервісна (незалежні модулі).

Монолітна архітектура – є традиційною архітектурою якою частіше за все користуються при розробці. Це підхід в якому всі частини системи, такі як бази даних, обчислювальні модулі, інтерфейс, бізнес логіка працюють нероздільно і не можуть існувати окремо [20].

Зазвичай це означає що вся кодова база знаходиться в одному місці, працює зі спільним сховищем даних, яке частіше за все представляє із себе базу даних і файлову систему (сховище). Таким чином вносячи зміну навіть у певну частину такої системи, розробник через нероздільність системи впливає в певних моментах на всю її роботу. Найчастішим таким випадком є процес оновлення в якому доводиться тимчасово переводити всю систему у неробочий стан.

Мікросервісна архітектура – дана архітектура представляє собою розбиття всієї системи на окремі незалежні модулі. Такі модулі є повністю незалежними, кожен з них має свої ресурси, свої дані і кодову базу. За необхідності модулі можуть “спілкуватися” один з одним для обміну даними та виконанням певних функцій, однак вони не мають бути занадто тісно зв’язані між собою, щоб їх можна було вилучити чи легко замінити. Така взаємодія часто реалізується за допомогою якогось API, частіше за все використовуючи HTTP-ресурси [20].

Візуальне представлення цих архітектур наведено на рисунку 2.1. Кожна з цих архітектур має свої певні переваги і недоліки які треба враховувати при розробці.

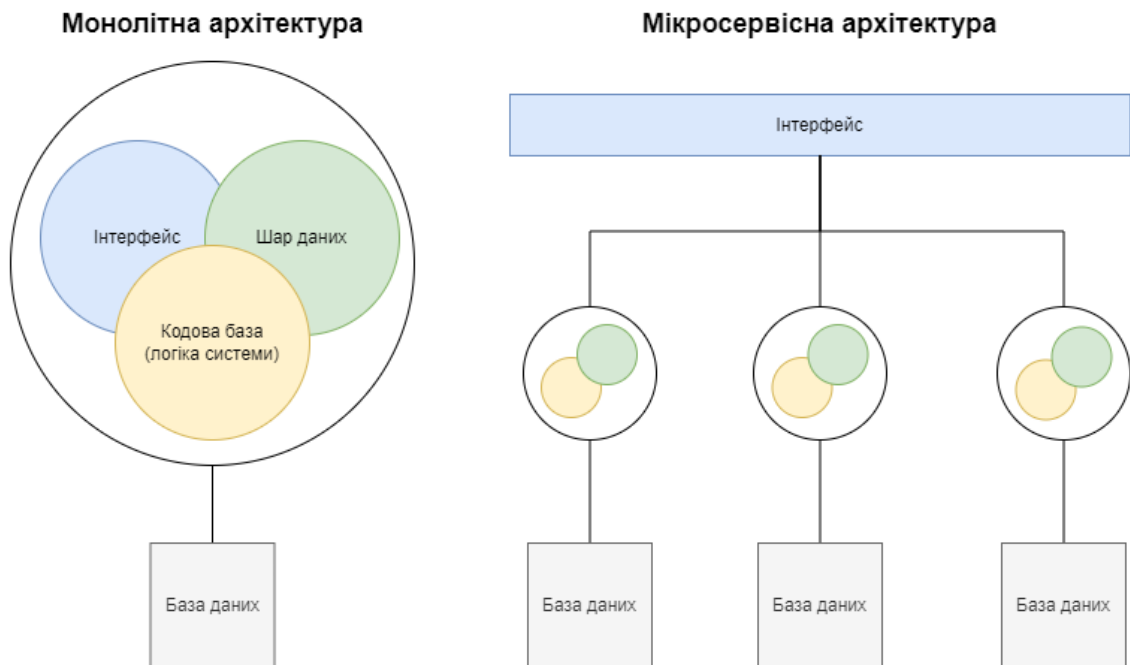


Рисунок 2.1 – Візуальне представлення монолітної і мікросервісної архітектур

Переваги монолітної архітектури [20,21,22]:

- легке керування системою. Оскільки вся система являє собою одне ціле, то й системи моніторингу роботи системи, такі як логування, ведення журналу діяльності роботи системи, викликів функції користувачами, чи такі дії як кешування, загальна обробка помилок та ін., потребують менше сил та ресурсів для її реалізації.

- швидке і просте розгортання. Перший етап розгортання стає більш легких оскільки необхідно розгорнути одну систему в якій знаходиться весь функціонал. Тобто для розгортання необхідно лише налаштувати необхідну групу файлів і запустити її;

- простий процес тестування. Виходячи з того що система нероздільна, отже й тестування стає легшим оскільки достатньо розгорнути її, перевірити весь наявний функціонал, без необхідності додаткового тестування взаємодії окремих

модулів і відповідно їх розгортання. Однак слід відзначити що саме тестування може бути повільнішим, оскільки тестуванню має підлягати вся система;

- проста архітектура. Оскільки абсолютно вся кодова база працює зі спільними ресурсами, то немає необхідності розроблювати архітектуру для взаємодії окремих модулів, та перевірки коректності їх синхронізації та ін.

Недоліки архітектури [20,21,22]:

- складність масштабування. Кодова база з розвитком системи невпинно росте і неможливість розділити систему на окремі незалежні модулі стає проблемою. Наявність спільних ресурсів також вимагає їх нарощування, як то структури бази даних, підвищене навантаження на системи моніторингу, кешування та ін. Структура коду і взаємодія між частинами системи також стає складнішою і в залежності від обраної архітектури для організації коду, можуть виникати ті чи інші проблеми чи навіть конфліктні ситуації в системі;

- низька гнучкість. Відокремлення якихось блоків у системі чи їх заміна може призвести до складнощів, оскільки якщо моноліт був написаний з використанням певного інструменту для керування даними, їх спеціалізованою обробкою, то доведеться переписувати всю систему для використання нових технологій;

- складність оновлення розвинутої системи. Нероздільність додатку не дає можливості оновлювати систему окремо, не зупиняючи роботу інших частин. Таким чином оновлення системи супроводжується тимчасовою зупинкою всієї системи, а також більш довгим оновленням оскільки вся система заново має розгортатися.

Переваги мікросервісної архітектури [20,21,22]:

- вища гнучкість та краще масштабування. Ця перевага вставиться в протипагу проблемі монолітній архітектурі. Завдяки тому що кожен модуль представляє з себе незалежний компонент системи, то його можна легко модифікувати, змінювати в ньому набір інструментів чи взагалі видалити його без впливу на всю систему в цілому. Таким чином, якщо стає необхідність для якоїсь певної задачі використати новий інструмент чи модифікувати алгоритм роботи, будь які зміни вплинуть лише на сам мікросервіс;

- легкість оновлення. Завдяки тому що кожен модуль є незалежним, з'являється можливість модифікації окремого компоненту і оновлення його у системі без зупинки всієї системи. Дана властивість дуже корисна коли необхідно змінити наприклад лише функціонал що становить 5% системи і це можна зробити без розгортання заново усієї іншої частини проекту;

- швидше тестування окремих модулів. У випадку якщо змінюється лише один модуль що не має тісних зв'язків з іншим – тестування може бути швидшим за рахунок того що відомо, що інша частина проекту буде працювати заплановано і вона вже є протестованою;

- ізоляція системи. Помилки що будуть ставатися під час розробки будуть впливати лише на модуль що модифікується. Таким чином будь які нововведення стають більш безпечними.

Недоліки мікросервісної архітектури [20,21,22]:

- складність розгортання. Для розгортання мікросервісної архітектури необхідно розгортати кожен мікросервіс як окрему систему, а також після розгортання необхідно буде впевнитися що кожен сервіс працює стабільно і має доступ до кожного іншого сервісу. Також необхідно впевнитися що вони правильно синхронізуються і ніде не має пауз чи порушень у взаємодії після розгортання;

- складність глобального тестування. Необхідно тестувати кожен модуль окремо, а також у комплексі один з одним. Складність підвищується також через те що кожен модуль має власні окремі ресурси;

- побічні проблеми. Моніторинг системи, логування та інші подібні операції стають більш складнішими, оскільки мають бути налаштовані окремо на всіх модулях, чи має бути створений окремий модуль який буде виконувати усю цю додаткову роботу, що також вимагає більш ретельних і складних архітектурних рішень.

Виходячи з аналізу переваг та недоліків розглянутих архітектур можна зробити висновок що вони взаємовиключні. Здебільшого дана архітектура пріоритетніша для розробки, оскільки момент коли складність масштабування і

підтримки системи стає складнішою за мікросервісну архітектуру знаходиться дуже далеко у часі і частіше за все більшість системи не доходить до цієї точки.

Однак мікросервісна архітектура може бути корисною у випадку якщо систему можна дуже легко розділити на окремі модулі, тобто існує така частина системи яка може використовуватися майже як незалежна частина. В такому випадку вилучення модулю і перетворення його в окремий сервіс не є критичним. В такому випадку модуль представляється як окремий інтегрований сервіс.

Отже для поточної системи здебільшого підходить моноліт, однак за рахунок того що в системі наявний обчислювальний модуль який виконує задачі обчислення маршрутів, тобто має основний алгоритм і він не має прямого тісного зв'язку з іншою частиною – він може бути для простоти відокремлений від усієї системи що в майбутньому може надавати додаткові можливості щодо легкої модифікації цього модулю, а також можливості інтеграції даного модуля у інші системи. Також оскільки даний модуль робить складні обчислення, то його робота потребує максимальної кількості наявних ресурсів для більшої ефективності, тому зважаючи на наведені факти – рішення щодо створення його у вигляді окремого сервісу стає більш доцільним.

У висновку, можна окреслити у базовому вигляді архітектуру системи як монолітну, але з окремим обчислювальним модулем у вигляді зовнішньої інтеграції як приклад інтеграції сервісу Google Maps для роботи з мапою, чи як мікросервісну але лише з двома модулями, один з яких буде основним і мати майже всю кодову базу, а другий робити обчислення. Схема архітектури створюваної системи наведена на рисунку 2.2.

З даної схеми можна побачити що дана система є розподіленою, де є клієнтська частина і серверна. Також існує один додатковий сервіс який створюється під час розробки системи і два зовнішніх сервіси для роботи з мапами і авторизацією. Слід зазначити що і сервер і клієнт мають використовувати їх, але кожен у своїх цілях. Клієнтська для авторизації і демонстрації мапи, серверна – для перевірки даних зареєстрованого через сервіс користувача і отримання додаткових даних з щодо маршруту (час, навантаження трафіку)

відповідно. Зовнішні сервіси є інтегрованими і в даній системі їх спілкування йде за допомогою спеціально наданого API. На схемі додатково можна побачити які компоненти ще присутні у системі і хто ними може керувати.

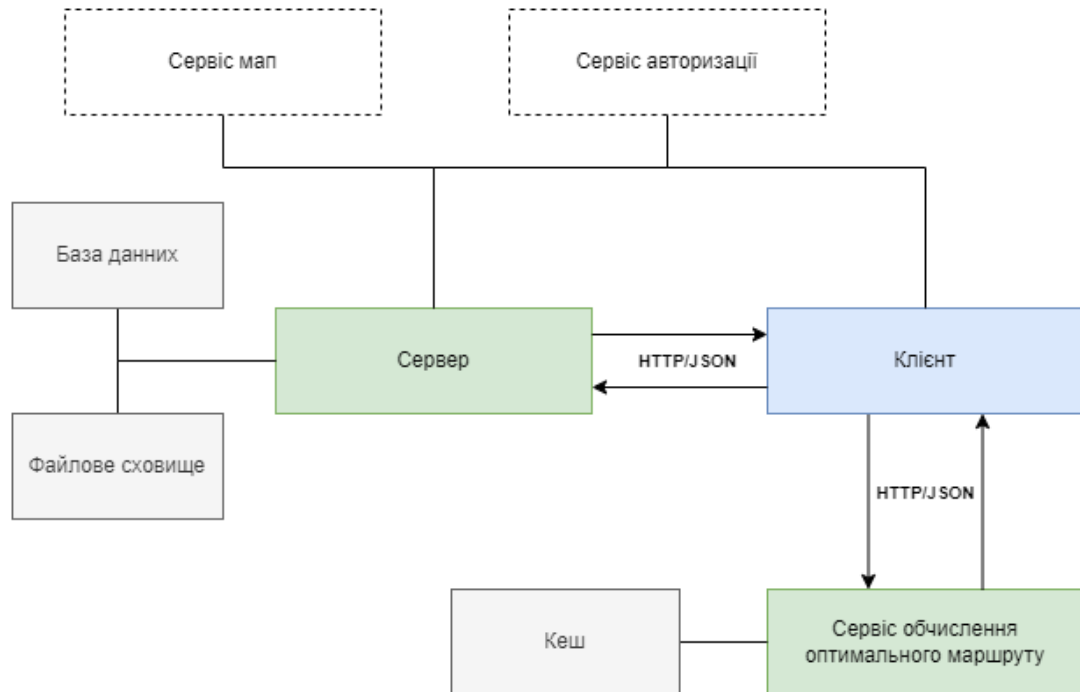


Рисунок 2.2 – Архітектура поточної системи

Дана схема є спрощеною, надалі буде розглянуто які саме технології будуть використані для реалізації кожного компоненту, а також архітектуру клієнт-серверної взаємодії для спілкування між сервером і клієнтом. Однак зі схеми вже видно що спілкування йде за протоколом HTTP використовуючи передачу даних у вигляді JSON об'єктів.

2.2 Вибір архітектури для програмного забезпечення системи

Для організації кодової бази існує декілька архітектур, однак всі вони мають спільну основу і відображають основні принципи чистої архітектури, яка була

заснована Робертом Мартіном [22]. На рисунку 2.3 продемонстрована візуально схема розділення архітектури системи по шарам.

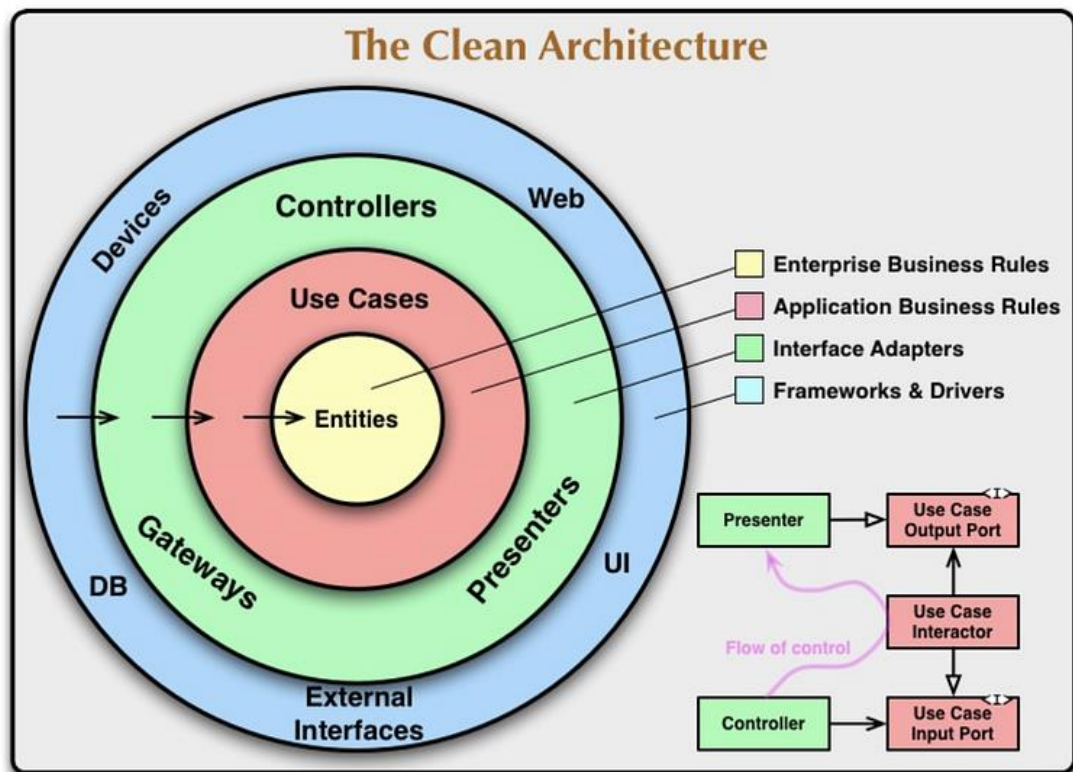


Рисунок 2.3 – Схема розбиття частин системи на шари за чистою архітектурою

Продемонстрований приклад організації компонентів системи вважається еталоном. Основною ідеєю даної архітектури є зниження залежностей одних компонентів від іншої, а саме логіки системи від конкретної реалізації системи. За даними принципами система має бути більш предметно-орієнтованим, ніж інструментально-орієнтованим.

Дана архітектура ставить на меті можливість відокремлення всієї логіки системи від інструментів які вона використовувала для реалізації поточних задач. Таким чином якщо система будується правильно, то задачі видалення чи заміни інструменту не будуть викликати труднощів.

Також чиста архітектура пропонує коректне розділення кодової бази системи за їх функціоналом, щоб була можливість побудувати окремі частини

системи модульно з можливістю їх перевикористання, а також підвищити її масштабованість.

Отже за цією архітектурою існує чотири шари на які поділяється система і правило залежностей, що є правилом їх взаємодії між собою. Система має наступні рівні (від нижнього до вищого) [23]:

- рівень відображення (рівень фреймворків і драйверів);
- рівень логіки системи;
- рівень домену;
- рівень даних.

Рівень даних – це рівень на якому визначаються всі основні об’єкти системи. На ньому знаходяться всі конкретні реалізації класів, описуються їх поля, залежності один від одного та ін.

Рівень домену – визначає всі необхідні інтерфейси для взаємодії з даними. На цьому рівні відбуваються всі необхідні обчислення і маніпуляції з даними, наприклад їх зберігання, модифікація тощо. Лише цей рівень має доступ до даних, але він так само використовує певні інтерфейси для маніпуляції з даними.

Рівень логіки системи – на цьому рівні знаходиться вся основна логіка взаємодії системи з користувачем. саме тут відбувається обробка дій користувача, перевірка даних на коректність, делегування певних дій на рівень домену де відбувається обробка даних.

Рівень відображення – зовнішній рівень на якому мають використовуватися всі інструменти які задіяні для вирішення таких задач як відображення користувацького інтерфейсу, зовнішні інтерфейси взаємодії з якимись сервісами, бази даних та ін.

Ядром такої архітектури є саме рівні – даних, домену та бізнес логіки. Вони не повинні мати зв’язків з конкретними інструментами, а лише використовувати інтерфейс зв’язку між ними. Таким чином рівень відображення являє собою рівень який може змінюватися в залежності від потреб системи [23].

Існує також правило залежностей щодо взаємодії даних шарів. Кожен рівень системи може спілкуватися з рівнем що знаходиться всередині. Таким чином

наприклад рівень відображення не може отримати доступ одразу до даних, а рівень даних не може самостійно передати дані для відображення в інтерфейсі. Таким чином зберігається ізольованість рівнів систем один від інших і підвищується здатність системи до масштабування, тестування і кодова база стає більш передбачуваною [23].

Існує декілька готових шаблонів проектування системи за чистою архітектурою. Відомими з них є MV(C/P/VM) - шаблони. Їх ідея повністю базується на принципах чистої архітектури щодо розділення логіки програмного забезпечення та слідуванню правилу залежності. Кожен шаблон поділяє всю структуру кодової бази на три частини [24]:

- M (model) – визначає собою рівень домену. Кожна модель має доступ до даних через свій інтерфейс і виконує певні операції і необхідні обчислення;

- C/P/VM (controllers) – визначає рівень логіки системи. Саме контролери оброблюються всі дії користувача, проводять валідацію даних і передають ці дані на рівень моделей. Контролери мають декілька реалізацій, але всі вони відрізняються лише принципом роботи з інтерфейсом. C (контролер) – робота з інтерфейсом де контролер має прямий доступ до користувацького інтерфейсу. P (presenter) – робота з користувацьким інтерфейсом де контролер не має прямого доступу, а спілкується так само через інтерфейс (API), що дозволяє перевикористовувати логіку і легше тестувати. VM (viewModel) – підхід в якому будь яка зміна даних веде до автоматичного оновлення інтерфейсу без необхідності зв'язку з інтерфейсом, інтерфейс лише підписується на оновлення даних.

- V (view) – визначає рівень представлення. На цьому рівні будується інтерфейс який отримує дані і відображає їх користувачу. Також цей рівень реагує на всі дії користувача і делегує їх контролерам для подальшої обробки

Дані шаблони гарно демонструють принципи розділення логіки програмного забезпечення. Шаблон MVVM демонстрував також ідею максимального абстрагування інтерфейсу в системі, оскільки система за даним шаблоном працює абсолютно незалежно від інтерфейсу і інтерфейс лише підписується на дані і їх оновлення.

Ідея шаблону MVVM була розвинута і виділена у окрему парадигму – реактивне програмування. Дана парадигма передбачає те що програма має автоматично реагувати і оброблювати потік даних. Таким чином, якщо якийсь компонент системи залежить від певних даних, то він підписується на їх оновлення і прослуховує. Коли дані оновлюються то викликаються події які ініціюються оновлення всіх залежних від них компонентів. Даний підхід став корисним не тільки для інтерфейсів, однак оскільки інтерфейс завжди є тією частиною системи, яка демонструє дані користувачу і частіше за все має реагувати на їх зміну – саме на нього був здійснений основний вплив [24, 25].

Практичним прикладом використання даної парадигми у розроблюваній системі може бути системи побудови маршруту в інтерфейсі системи. Наприклад система відправляє масив точок які необхідно обійти у зовнішній сервіс, який обчислює маршрут і після того як інформація щодо побудованого маршруту надходить назад у систему – система автоматично реагує на них і будує маршрут. Таким чином може відбуватися і зміна, коли з'являється нова точка і змінений маршрут при оновленні автоматично оновлюється в системі.

Дана парадигма виявилася успішною і отримала свій розвиток у вигляді Flux архітектури яка потім була реалізована у майже всіх інструментах для створення інтерфейсу користувача. Дана архітектура базується на однонаправленому потоці даних, де інтерфейс підписується на оновлення даних, які відображають стан системи і ініціює події які за ланцюгом викликають певні дії, такі як обробку даних, виклики зовнішніх сервісів і оновлення стану. Даний підхід витіснив всі інші підходи щодо обробки даних і став основним. У сучасній розробці інші використовуються здебільшого для навчальних цілей, ніж для реальної розробки, тому вибір такого підходу стає очевидним [26].

Виходячи з аналізу сучасного підходу до розробки програмного забезпечення системи, можна побудувати схему архітектуру розроблюваної системи. Дана система є розподіленою, тобто існує дві незалежні частини системи – клієнтська частин та серверна. Обидві спілкуються між собою за допомогою інтерфейсу і жодна не є повноцінною системою. Дані і їх обробка знаходяться віддалено на

сервері, інтерфейс і обробка інтерфейсу знаходиться у користувача. Для таких систем архітектура програмного забезпечення визначається окремо, але її так само можна продемонструвати в цілому, що продемонстровано на рисунку 2.4

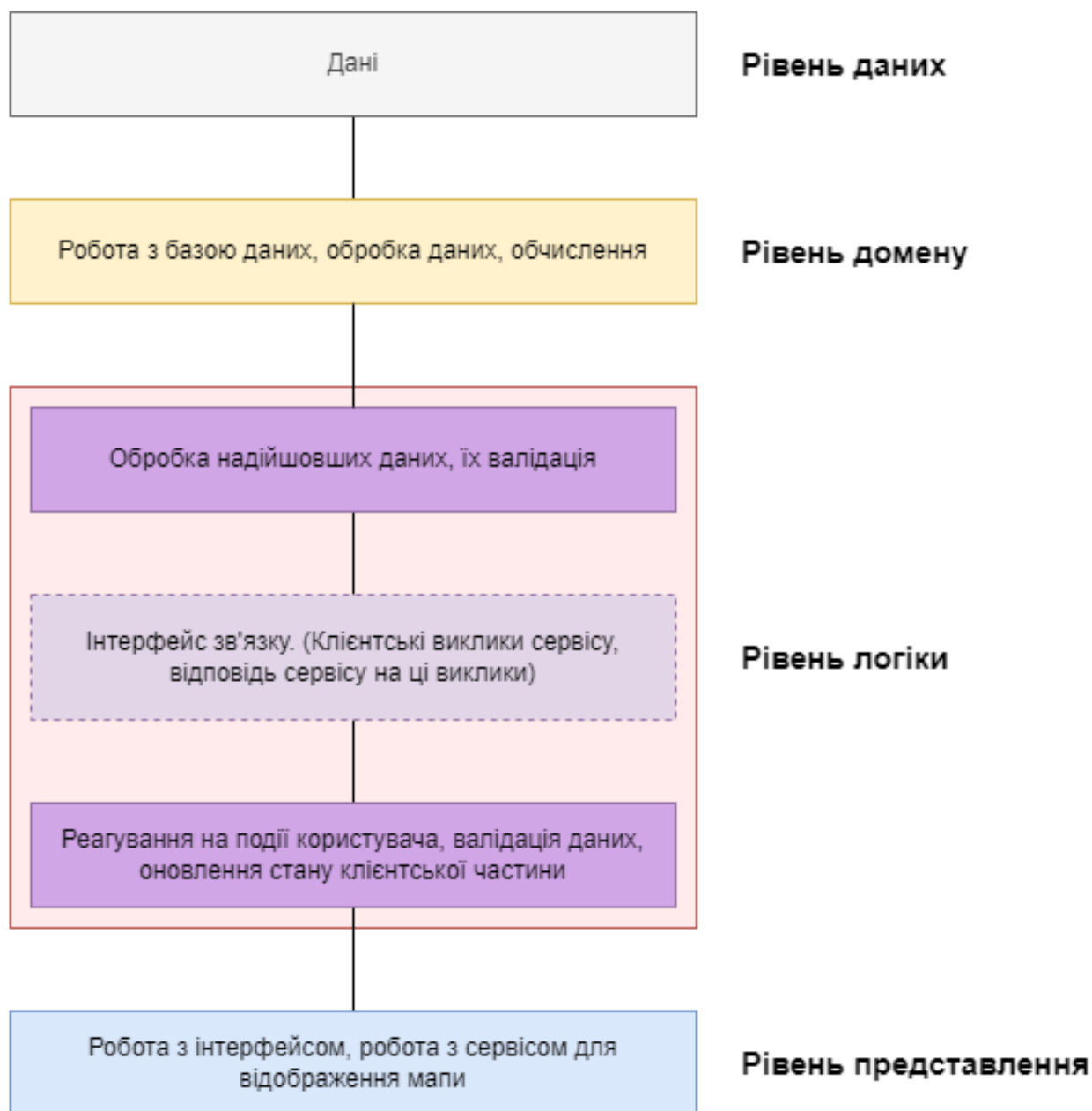


Рисунок 2.4 – Архітектура роботи програмного забезпечення за рівнями

Продемонстрована схема є загальною для всієї системи. Якщо описувати архітектуру окремо для кожної частини, то єдина відмінність буде полягати у тому що рівень домену для клієнтської частини буде представлений у вигляді його

внутрішнього стану і викликом зовнішнього сервісу через спеціальне API. Для серверної частини рівень представлення буде відсутній у тривіальному вигляді і представлений у вигляді даних, що є відповіддю до дій викликаних клієнтською частиною системи.

2.3 Вибір архітектури клієнт-серверної взаємодії

Під час розробки розподілених систем, основним питанням є питання щодо взаємодії частин між собою. Основними моментами під час розробки такої системи – була необхідність створити таку взаємодію, яка буде підтримувати гарну масштабованість і певний стандарт в обміні даними.

Існує чотири основних архітектурних підходів [27, 28]:

- RPC;
- SOAP;
- REST;
- GraphQL;

RPC (віддалений виклик процедур) – це найстаріший підхід до передачі повідомлень між частинами системи. Даний в загальному плані є специфікацію яка дозволяє віддалено викликати функції так ніби вони існують локально. Дана специфікація має декілька конкретних реалізацій серед яких JSON-RPC, XML-RPC який в подальшому був розвинений у окрему архітектуру SOAP [27].

Робота RPC полягає у тому що клієнт, викликаючи віддалену процедуру (функцію), має передати саму функцію, її параметри і можливо ще додаткову інформацію яка може бути необхідна, наприклад токен авторизації для отримання дозволу скористатися даною функцією. Ці дані серіалізуються в певному вигляді і як було наведено вище на разі це JSON або XML. Сервер отримує дані у переданому форматі, десеріалізує їх і виконує виклик функції з переданими параметрами [27]. Конкретна реалізація передбачає існування певного класу,

заглушки яка десереалізує дані на відправку і отримання. В сучасних реалізації з розвитком асинхронно стилю програмування, дані реалізації виглядають як звичайний клас сервісу з асинхронними методами.

Для передачі даних використовується протокол HTTP, з використання двох типів запитів - GET – отримання даних і POST – всі інші дії, але може використовуватися і лише POST. Приклад тіла запиту наведено у лістингу 2.1, в даному прикладі передається метод updateUser з відповідними параметрами і додатково ще токен.

Лістинг 2.1 – Приклад тіла запиту в JSON-RPC

```
{
  "method": "updateUser",
  "params": {
    "email": "test@gmail.com",
    "phone": "+1234567890",
    "username": "username",
    "newPassword": "mY_pass"
  },
  "token": "AFY*&r@#!*&hf!wiofsdj kfas",
}
```

Переваги RPC [27, 28]:

- простий в реалізації. Дана архітектура легка в реалізації, оскільки все що в ній достатньо лише передати функцію і параметри до кінцевої точки, не використовуючи ніяких додаткових механізмів, можливостей протоколу HTTP з їх налаштуванням;

- легкість додавання нового функціоналу. З точки зору клієнтської частини виклик нової функції здійснюється за рахунок виклику іншої функції з новими параметрами, цей механізм можна навіть автоматизувати і згенерувати нову функцію в заглушці. З точки зору серверної частини створюється нова функція, але механізм виклику функції після десереалізації залишається незмінним;

- мінімальна корисне навантаження. Дана особливість корисна у випадку складних обчислень коли в системі йде велике навантаження і частини системи

обмінюються величезною кількістю повідомлень. Вона пояснюється тим, що RPC відправляє лише необхідні дані – метод та параметри без додаткової інформації.

Недоліки RPC [27, 28]:

- розкриття внутрішньої структури. Створюється занадто тісний зв'язок з конкретною реалізацією серверної частини, оскільки архітектура полягає у тому що викликати прямі методи на кінцевій точці з конкретними параметрами. З точки зору безпеки дана особливість стає проблемною. Також виникає неясність з точки зору реалізації функції вибору процедур і щодо вибору кінцевої точки, оскільки немає стандарту реалізації;

- тісний зв'язок з мовою програмування чи платформою. RPC в залежності від його реалізації може відрізнятися в різних фреймворках чи мовах програмування. Таким чином доводиться вчити кожен раз як певний інструмент реалізує цей механізм, також через відсутність єдиного стандарту;

- відсутність вбудованої системи обробки помилок, яку необхідно створювати самостійно і яка так само не має певного стандарту і в готових реалізаціях може відрізнятися.

SOAP – розвиток RPC підходу. Цей протокол є високостандартизованим і використовує і описує всі запити на мові XML. Даний протокол для транспорту повідомлень може використовувати будь-який протокол прикладного рівня – HTTP, SMTP, FTP, одна частіше за все використовується саме HTTP.

Повідомлення SOAP містить наступні складові [29]:

- envelope – головний елемент, в ньому знаходяться всі інші частини повідомлення і саме він вказує на структуру повідомлення;

- header – необов'язковий елемент який містить додаткові дані такі як токен для авторизації, метадані та інші параметри, що не мають відношення до основного повідомлення але зчитуються системою для контролю доступу та інших операцій;

- body – основний блок в якому описується вся корисна інформація, що містить у собі основні дані повідомлення. Такими даними є функція, команда та набір параметрів для неї;

- fault – необов’язковий блок який необхідний у разі виникнення помилок для надання її опису клієнтській.

Зважаючи на чітку структуру повідомлення, а також те що даний протокол був створений на мові веб-сервісів WSDL - це дозволило багатьом мовам налаштувати його і зробити за певним стандартом. Таким чином вирішивши одну з проблем RPC.

Переваги SOAP [27, 28]:

- стандартизація. Висока стандартизованість вирішила проблему з неявною реалізацією підходу обміну повідомленнями і зробила її незалежною від платформи на якій відбувається розробка.

- вбудована обробка помилок, що вирішує вище описану проблему RPC;

- більші можливості щодо гарантування безпеки. SOAP може використовувати перевірені алгоритми XML-шифрування, а також використовувати WS-Security – протокол розширення для даної архітектури, що визначає правилами за якими можна гарантувати конфіденційність і цілісність даних. В ньому описуються різні способи підпису повідомлення, сертифікації, створення додаткових токенів та механізм шифрування;

- може бути використаний з широким спектром протоколів прикладного рівня як було зазначено вище.

Недоліки SOAP [27, 28]:

- великий обсяг даних що передаються. Одна з найкритичніших проблем, основна на тому що протокол може використовувати лише формат XML, який через свою структуру і наявність додаткових метаданих має більшу вагу ніж подібний аналог в JSON форматі, що створює додаткове навантаження на систему;

- складна структура. Також критична проблема що так само пов’язана із використанням XML схемою, що викликає складнощі у швидкому розумінні повідомлення і його модифікації.

Приклад SOAP повідомлення наведено в лістингу 2.2. В цьому лістингу наведено повну структуру SOAP повідомлення за винятком блоку помилки.

Протокол SOAP зміг подолати проблеми стандартизації, однак в свою чергу це створило додаткові проблеми з навантаженням на систему. В якості спроби подолати їх, повідомлення могли бути відправлені у якості звичайних документів у форматі XML, а всі необхідні параметри пересилаються в запитих HTTP протоколу. Однак з іншого боку дана оптимізація пов'язує SOAP з HTTP, що надалі створює проблеми у разі зміни протоколу, таким чином позбавляючи можливості його використання з іншими протоколами.

Лістинг 2.2 – Приклад SOAP-повідомлення

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header>
    <auth>Bearer 23q8hy0f7wean9guewrsdfinvkcxl</auth>
  </soap:Header>
  <soap:Body>
    <createProduct xmlns="http://shop.com/product">
      <name>Product Name</product>
      <description>Product description</description>
      <qty>2</qty>
    </createProduct>
  </soap:Body>
</soap:Envelope>
```

REST – архітектурний підхід який повністю побудований з повноцінним використанням можливостей HTTP протоколу. Даний підхід є стандартизований і пропонує набір правил за якими необхідно будувати архітектуру взаємодії між сервісами. Також існує набір рекомендацій, як правильно використовувати протокол HTTP і його можливості для реалізації даної архітектури. Іноді трапляються ситуації коли рекомендаціям неможливо повноцінно слідувати чи трапляються спеціальні винятки коли їх необхідно порушувати, наприклад з міркувань безпеки.

Набір обов'язкових правил для побудови REST [30]:

- клієнт-серверна архітектура. Має зберігатися чітке розділення системи на клієнтів та серверів. Обидві частини мають працювати лише за принципом запит-відповідь, не маючи інших зв'язків щодо конкретного інтерфейсу, того як

зберігаються дані та ін. Завдяки цьому правилу зберігається незалежність клієнту та серверу один від одного, що дозволяє змінювати їх окремо один від одного не вносячи змін в структуру;

- відсутність стану. Правило пов'язане з вимогою клієнт-серверної архітектури. Сервер не повинен зберігати жодної інформації щодо клієнту, що означає відсутність збереження будь якого його стану. Іноді виникає потреба у збереженні якихось даних, які мають відношення щодо поточних дій користувача у системі. Для таких задач стан системи зберігається на клієнті у вигляді сесії чи інших механізмів який потім заново відновлюється на сервері. За даним правилом досягається також ізольованість частин системи і її стабільність, оскільки помилки у роботі серверу, його заміна, можуть порушити стан чи видалити його відповідно;

- контроль над кешуванням. Кожен запит має бути чітко ідентифікований на те, чи може бути він закешований. Сервер має чітко надавати такі можливості для ідентифікації. Зазвичай це означає що запити які використовуються для читання даних можуть бути закешовані до моменту їх зміни, а запити що вносять зміни – мають помічатися як некешовані, оскільки жодна зміна не є ідентичною минулій;

- єдиний інтерфейс. Для взаємодії з сервером має бути створений єдиний інтерфейс зв'язку з ним, що створює незалежність для його взаємодії з будь-якою платформою;

- розбиття за шарами. Роботу з ресурсами серверу можна розділити на шари, де наприклад для обробки одного ресурсу, сервер може використовувати інший сервіс, а той у свою чергу інші. Однак кожен такий шар може знати і мати уявлення лише про наступний шар;

- код за вимогою. Необов'язкове правило, яка може дозволяти серверу віддавати частину коду клієнту виконання на його стороні. Найпоширенішим прикладом може бути видача у відповіді JavaScript скрипту для обробці в браузері.

Умова єдиного інтерфейсу також має свої певні умови [30]:

- ідентифікація ресурсу. Кожен ресурс має мати унікальний ідентифікатор який його ідентифікує. В REST, оскільки він використовує протокол HTTP, таким ідентифікатором є URI. Наприклад для доступу до певного товари може

використовуватися запит <https://myShop/products/21>, чи товарів загалом для роботи з ними - <https://myShop/products>;

- робота з ресурсами через представлення. При роботі з ресурсами, система отримує доступ до них через спеціальний запит, який надсилає у відповіді дані в певному форматі, частіше за все в одному з наступних – JSON, XML, HTML. Дане представлення має бути відокремленим від їх реального вигляду у якому вони зберігаються у системі;

- самодостатність повідомлень. Кожне повідомлення має у собі містити всі необхідні дані для його обробки. Це досягається тим що система як описано вище не може мати стану і не може спиратися на інші дані які могли бути створені у системі під час його роботи, наприклад кеші;

- HATEOAS. За цією умовою клієнт може впливати на систему лише через дії які визначають динамічно сервером і надаються клієнту. Наприклад це може бути гіперпосилання та гіпертекст. Прикладом такого може бути система оформлення замовлення послуги в якійсь інтернет системі, де після успішно обробленого запиту на оформлення послуги, сервер повертає інші посилання що можуть надавати наступні можливості такі як відмінити замовлення, зв'язатися з замовником чи виправити його. Дана умова часто не виконується, оскільки такий підхід потребує від розробника більше сил на створення правильної реалізації [27].

Завдяки даним вимогам REST допомагає вирішити основні проблеми в побудові взаємодії між сервісами які були виявлені в RPC та SOAP. Однак деякі загальні проблеми досі залишаються невирішеними, такі як проблема налаштування безпеки.

Також існують правила щодо конкретної реалізації REST API. Вони не є жорсткими вимогами і іноді можуть порушуватися. Вони описують як правильно використовувати протокол HTTP і його можливості.

Кожен запит в HTTP представляє собою URL адресу як ідентифікатор ресурсу і метод. Кожен метод має виконувати певну ціль. Основними методами якими користуються при розробці REST API є:

- GET – даний тип запитів підтримує кешування і має завжди використовуватися лише для задач отримання даних, також передає нескриті параметри тому з точки зору безпеки не може використовуватися для передачі чутливих даних;

- POST - запит для створення даних. Не кешується і передає всі дані у тілі запиту, тому він може використовуватися також іноді як заміна GET у випадку чутливості даних, чи нейтралізації кешу. Даний запит не є іденпотентним оскільки кожен запит повертає різну відповідь;

- PUT – використовується для модифікації даних, через це багаторазові його виклики надають однаковий результат;

- DELETE – використовується для видалення даних;

- PATCH – запит на часткове оновлення даних, відрізняється від PUT тим що може бути не іденпотентним. Використовується рідкісно.

Кожен запит HTTP має заголовки в яких зберігається вся додаткова інформація що необхідна для обробки – формат відповіді, аутентифікаційні дані, статус запиту та інше. Дана інформація не має потрапляти у тіло запитів чи їх параметри.

Всі запити HTTP має певний формат відповіді який включає у себе код статусу, тіло відповіді і так самого заголовки. За даним принципом вимагається щоб дана інформація завжди була коректною. Тип відповіді має відповідати формату запиту який був запрошений, кожен статус має відповідати дійсності. Статуси діляться на чотири основні групи, де 2XX відповідають за успішні дії, 3XX – перенаправлення, 4XX – логічні помилки та 5XX – непередбачувані помилки. Наприклад найпоширенішими є статус 200 – успішне виконання, 401 – неавторизований доступ, 404 – ресурс не знайдено чи 500 – помилка на сервері.

Переваги REST [30]:

- максимальна абстракція. REST скриває внутрішню структуру серверу і надає універсальний інтерфейс для взаємодії що може використовуватися незалежно від мови програмування, інструменту чи платформи;

- клієнт-серверна архітектура. REST створює чітке розділення задач за відповідальністю між сервером та клієнтом і не має жодного прямого зв'язку щодо стану клієнту і також власного стану, що підвищує масштабованість системи і її гнучкість з можливістю підміни серверу, введення додаткових сервісів та їх незалежну від клієнта модифікацію;

- стандартизація. Володіючи знанням про певні вимоги і рекомендації щодо побудови REST архітектури, розробник може швидко зрозуміти те як побудована взаємодія між сервером і клієнтом;

- легкість реалізації. Дана архітектура не потребує додаткових розробок для взаємодії з нею. Все відбувається за допомогою універсальних HTTP запитів які реалізовані в кожній системі з самого початку.

Недоліки REST [30]:

- надлишковість. Основна проблема REST архітектури оскільки однією з вимог для серверу є самодостатність повідомлень, отже сервер з нього має отримувати завжди абсолютно всю необхідну інформацію для роботи з даними. Клієнт в свою чергу отримує всі дані які передбачені у відповіді від сервера. Дані можуть бути надлишковими для поточної задачі, а фільтрація даних для кожного сценарію вимагає створення окремого запиту чи спеціального підходу який би дозволив обирати необхідні, що також ускладнює розробку;

- відсутність вимог до конкретної реалізації REST API. Немає остаточних вимог щодо конкретної структури серверу і вирішення таких задач як ідентифікації ресурсів, використання методів для отримання даних, передачі додаткової інформації (ключів, токенів), передачі параметрів та ін.

REST архітектура стала найпоширенішою через свою простоту і очевидність в реалізації, витіснивши майже повністю RPC та SOAP. Однак деякі недоліки щодо надлишковості в один момент розвитку систем ставали занадто суттєвими і для надвеликих проєктів був створений ще один спосіб взаємодії.

GraphQL – нова технологія клієнт-серверної взаємодії. Це мова запитів що в свою чергу стала деяким самостійним архітектурним підходом, особливість якого

полягає в тому що система виконуючи запит на сервер описує те, як саме його потрібно обробляти.

Необхідність створення такого підходу була виникла через наявну надлишковість в REST архітектурі. В REST щоб отримати необхідний набір даних іноді доводиться робити декілька запитів підряд, чи створювати на сервері додаткові кінцеві точки що будуть повертати всі ці дані разом. Іноді даних може бути занадто багато, коли наприклад клієнтському додатку потрібні лише 3 поля із 40 що повертає сервер. GraphQL був створений в першу чергу саме для вирішення таких проблем. Він дозволяє описувати стороні що відправляє запит, які саме поля необхідно повернути. На сервері у свою чергу описується лише схема об'єкту один раз з якої клієнт обирає дані для отримання.

Для використання даної технології, сервер має створити спеціальні схеми, що є описом об'єктів з яких можна отримати дані і функції для обробки запитів. На відміну від REST архітектури, кінцева точка до якої робляться запити одна. Клієнт при створенні запиту має вказати функцію яку треба викликати на сервері і масив даних які він бажає отримати. Оскільки з використанням даного інструменту один і той же запит (з точки зору HTTP) може повертати різні дані – має використовуватися метод POST, аналогічно архітектурі RPC і всі дані записуються у тіло запиту. Приклад створення схеми об'єкта та створення запиту наведено у лістинга 2.3 та 2.4 відповідно.

Лістинг 2.3 – Приклад опису схеми об'єкту User

```
type User {  
  id: ID!  
  username: String!  
  email: String!  
  phone: String!  
}
```

При описі схем, GraphQL використовує власний набір типів – це Int, Float, String, Boolean, ID. Кожна схема також є типом який може бути використаний для побудови інших об'єктів, наприклад список працівників у відділенні доставки буде позначено типом [User!].

Лістинг 2.4 – Приклад запиту до GraphQL

```
query {  
  getAllUser {  
    id  
    name  
    phone  
  }  
}
```

В даному прикладі сервер поверне список користувачів з полями `id`, `name` та `phone` без інформації щодо пошти. Якщо наприклад буде необхідно повернути лише ФІО та пошту, клієнт вказує лише два відповідних поля і отримує список користувачів лише з даними полями. Таким чином якщо у великій системи, є об'єкт який має 100 полів і різні комбінації цих полів використовуються наприклад в 10 різних сценаріях, GraphQL надає можливість клієнту обрати для кожного сценарію необхідні дані і таким чином зменшуючи об'єм отримуваних даних. Сервер у свою чергу не потребує створення додаткових кінцевих точок з метою оптимізації кожного сценарію та інших маніпуляцій.

Переваги GraphQL [31, 32]:

- мінімальна корисне навантаження у відповідях. Основна перевага GraphQL завдяки якій кожна відповідь серверу має мінімальну вагу, оскільки повертає лише необхідний об'єм інформації. В навантажених системах відіграє ключову роль, допомагає оптимізувати трафік і тим самим зменшити навантаження на систему і вплинути позитивно на її пропускну здатність;

- спрощена обробка складних ресурсів. На відміну від інших архітектур, GraphQL дозволяє проводити вибірку полів з вкладеними сутностями, використовуючи при цьому один і той же інтерфейс. В інших архітектурах доводиться створювати додаткові кінцеві точки і методи для обробки даних, що мають містити додаткову інформацію окрім базової. Наприклад може існувати запит за яким користувач може отримати статтю і окремо запит що буде містити до неї коментарі. GraphQL дозволяє отримати подібні дані за один запит і ступінь вкладеності може бути будь-яким;

- самоописування. Схеми в GraphQL чітко описують з якими даними можуть працювати клієнти сервісу і який тип має кожне з його полів, на відміну від інших архітектур, де завчасно не можна точно знати який тип даних буде відправлено у відповіді. Це сильно спрощує роботу з точки зору документації API, оскільки схеми повністю описують структуру відповіді.

Недоліки GraphQL [31, 32]:

- відсутність вбудованого кешування. Як було зазначено вище, дана архітектура не може використовувати протокол HTTP для автоматичного кешування даних, оскільки кожний запит може мати різну схему. Різні вимоги щодо даних також ускладнюють і власну реалізацію кешу оскільки неможливо зберігати всі можливі комбінації полів при отриманні якогось ресурсу;

- складність створення запитів при великій вкладеності об'єктів. Кожен запит має вибірку з полів об'єкту і у випадку коли об'єкт стає занадто складним з великою кількістю вкладених об'єктів, створення самих запитів стає проблематичним;

- додаткові складності у виконанні деяких завдань. Обробка помилок ускладнюється, оскільки сервер може віддати лише код 200 чи 5XX, логічні помилки мають додатково оброблюватися. Робота з файлами не вбудована, необхідно використовувати сторонні інструменти чи власні рішення, окреме API для обробки файлів.

На основі аналізу GraphQL, можна зрозуміти що даний інструмент краще підходить для навантажених систем і мікросервісної архітектури, чи системах що мають швидко змінюватися і адаптуватися.

Існує також що один тим взаємодії між клієнтом та сервером через сокети. В даному випадку сервер і клієнт під'єднуються один до одного і починають прослуховувати команди один одного. Даний підхід необхідний для роботи систем у реальному часі і майже завжди використовується паралельно з іншими підходами.

Отже для розробки поточного системи, було обрано архітектуру REST. GraphQL не надає суттєвих переваг для даної системи, але вносить додаткові

складнощі у реалізацію, що робить його в даному випадку не досить доцільним до використання.

Для поточної системи можна в загальному випадку окреслити наступну схему основної частини API згідно створених ресурсів системи:

- /api/auth - авторизація користувача;
- /api/couriers – доступ до ресурсу кур'єрів;
- /api/orders – доступ до ресурсу замовлень;
- /api/routes – доступ до ресурсу маршрутів;
- /api/(couriers|orders|routes|depots)/:id - доступ до конкретної сутності.

Кожен з запитів має варіант з власним тим (GET, POST, PUT, DELETE) які призначені для виконання CRUD операцій на даними. Сервер для розрахунку оптимального шляху матиме лише дві кінцеві точки /api/routes та /api/routes/:id для розрахунку поточного шляху і оновлення існуючого.

Також для клієнт-серверної взаємодії використовуються сокети, для підтримки постійного зв'язку між клієнтам і сервером, оскільки події в системі такі як нові замовлення і оновлення маршрутів відбуваються динамічно, через що кожен з працівників так само швидко має отримувати актуальну інформацію щодо цього. Також кожен працівник і машина доставки мають відстежуватися на протязі шляху, для проведення додаткових оцінок щодо часу доставки та можливих маніпуляцій з маршрутом.

Сокети обмінюються між собою за допомогою команд які чітко визначені на сервері і клієнті. Для відправлення даних щодо поточного місцезнаходження кур'єра чи машини буде використана команда під назвою track, з якою передаються координати широти та довготи. Для клієнтської частини 'orders-placed', що надає список замовлень і маршрут за яким їх треба доставити.

Таким чином REST надає всі необхідні можливості для створення взаємодії в розподіленому додатку і його недоліки в поточній розробці не сильно впливають на систему.

2.4 Вибір інструментів для реалізації

Для реалізації розподіленою системи необхідно вибрати інструменти які будуть задовольняти вимогам архітектури побудови системи для кожної з її частин. В даній системі окремо розглядаються інструменти реалізації основної серверної частини, модулю обчислення оптимального шляху та клієнтської частини.

2.4.1 Інструменти розробки сервісної частини

Для побудови серверної частини системи був обраний фреймворк NestJS з мовою програмування TypeScript. Даний інструмент був побудований на основі Node Js, з використанням з використанням методу програмування через dependency injection. Основними ключовими перевагами його є [33, 34]:

- використання TypeScript. Фреймворк повністю побудований на даній мові що робить його строго типізований і всі модулі що створюються для нього так само мають бути строго типізованими. Ця особливість часто вирішує проблему звичайних JavaScript інструментів які іноді можуть не мати підтримки мови TypeScript, що в деяких ситуація ускладнює розробку;

- модульність. Вся архітектура сервер будується на незалежних модулях, що в свою чергу розділяються на декілька частин згідно чистої архітектури. Даний підхід підвищує гнучкість оскільки кожен модуль є ізольованим що сприяє масштабуванню системи і її тестуванню. Також даний підхід дозволяє потенційно перевикористовувати готові модулі, чи розділити систему на мікросервіси;

- використання декораторів. Декоратори дозволяють спростити написання коду оскільки вони огортають існуючі методи і дозволяють надати їм необхідну додаткову логіку. В даному фреймворці є всі необхідні базові декоратори що дозволяють вказувати такі речі як тип запиту, команду сокету яку оброблює метод, парсинг чи трансформація вхідних даних, контроль доступу до ресурсу та ін. Це дозволяє писати код більш чисто і в самоописуванім;

- тестування та dependency injection. Фреймворк повністю побудований з використанням мтеодології dependency injection, що дозволяє автоматично

створювати необхідні об'єкти у системі і додавати їх як залежності. Через це код стає більш передбачуваним і полегшує тестування.

Даний інструмент є досить потужним і як можна побачити надає багато готових рішень для побудови серверу. Враховуючи також те що сервер використовує ту саму мову, що буде використовувати й клієнтська частина додатку – це надає можливості щодо перевикористання деякого коду, такого як опис інтерфейсів, об'єктів і усуває потенційні проблеми щодо серіалізації та десеріалізації даних, які можуть трапитися у разі використання різних мов.

Для зберігання даних використовується база даних. Існують два основних типи баз даних – реляційні та не реляційні. Реляційні будуються на використанні таблиці які є відношеннями і взаємозв'язками між ними що побудовані за правилами реляційної алгебри. Нереляційні представляють з себе сховище незв'язаних між собою документів [35, 36].

Для даної системи пріоритетніший саме реляційний тип баз даних, оскільки існує декілька об'єктів в системі які мають зв'язки один з одним. Реляційні системи керування базами даних (СКБД) дозволяють ще на етапі створення бази описати їх структуру та зв'язки, підтримуючи цілісність ще на даному етапі. Нереляційні СКБД підтримують цілісність лише на рівні логіки системи і не гарантують її.

Серед основних представників реляційних баз даних є – MySQL, MariaDB та PostgreSQL. В поточній системі різниця між даними СКБД не є суттєвою, оскільки їх основні відмінності будують на можливостях індексування, способі зберігання даних, використанні внутрішніх ресурсів та ін. Основним критерієм за яким в системах рекомендовано обирати PostgreSQL є його об'єктно-орієнтованість що надає можливість керування складними типами даних і навіть проводити операції з JSON об'єктами [37]. Однак в даній системі немає необхідності в даному функціоналі тому для роботи з базою даних було обрано MySQL.

MySQL та MariaDB мають схожий синтаксис для опрацювання запитів, тому можуть замінятися за необхідності. MySQL також легше розгорнути на віддалених серверах оскільки існує більше можливостей для цього і більше готових рішень.

З точки зору роботи з СКБД на сервері можна використовувати спеціальні бібліотеки – ORM, які дозволяють декларативно у коді описувати структуру бази даних та обробляти запити. Дані бібліотеки дозволяють гнучко керувати базою і автоматично її оновлювати, що допомагає легко підтримувати та масштабувати систему.

Однією з таких бібліотек для керування реляційними СКБД є TypeORM. Ця бібліотека дозволяє працювати з усіма наведеними раніше СКБД, тому це дозволить у майбутньому перейти на будь-яку з них у разі необхідності [38]. Демонстраційний приклад декларативного опису об'єкту користувача наведено у лістингу 2.5.

Лістинг 2.5 – опис об'єкту користувача

```
@Entity('users')
export class User {
  @PrimaryColumn({ type: "varchar", length: 255 })
  @Exclude()
  id: string;

  @Column({ type: "varchar", length: 120, nullable: true })
  @Expose({ groups: [AccessGroups.ALL_USERS] })
  username: string;
}
```

Таким чином можна легко описати всю структуру бази даних та робити запити у вигляді звичайних методів що будуватимуть з описної схеми SQL запити, наприклад `usersRepository.findOne({ where: { id: id } })`. Даний підхід набагато спрощує роботу з СКБД оскільки не вимагає від розробника слідкувати за всіма SQL запитами і змінювати їх у разі зміни схеми.

2.4.2 Інструменти розробки обчислювального сервісу

Даний модуль є окремим сервісом, для обчислення оптимального маршруту між заданими точками. Він виконує лише цю задачу і немає прямого зв'язку з даними основної системи. Взаємодія з цим модулем так само відбувається за допомогою REST.

Для його створення було обрано мову програмування Python та фреймворк Flask. Дана мова програмування була обрана через свій простий синтаксис та величезною кількістю математичних бібліотек таких як NumPy, SciPy, та ін. Також існує декілька спеціалізованих бібліотек для роботи з машинним навчання та обробки графів [39].

Для виконання математичних обчислень використовується бібліотека NumPy, яка була створена для ефективною роботи з багатовимірними масивами даних і має велику кількість корисних функцій що можуть допомогти в обчисленні і роботі з графами.

Фреймворк Flask був обраний для побудови REST API для даного сервісу. Даний фреймворк дозволяє дуже легко створити сервер в найпростішому вигляді сервер може бути піднятий лише за допомогою двох команд [40].

Основні переваги даного фреймворку для поточної системи [41, 42]:

- легкість та мінімалістичність. Даний фреймворк може використовуватися самостійно без додаткових бібліотек. Він дозволяє швидко і легко розгорнути веб-сервіс, що досить сильно допомагає у розробці веб-сервісів оскільки вони за своєю архітектурою виконують не багато задач і можуть використовувати лише частину того функціоналу що зазвичай використовується при розробці монолітної системи;
- незалежність. Повний контроль над створення сервісу і абсолютні можливості щодо контролю за архітектурою. В даному інструменті немає обмежень щодо розробки, що дозволяє експериментувати з різними ідеями щодо роботи сервісу;
- кросплатформеність. Незалежний від операційної системи і може бути розгорнутий будь де.

Дані переваги роблять Flask ідеальним вибором для створення маленького сервісу, не використовує у системі стільки ж інструментів як основний серверний модуль і має проводити лише обчислення.

Однак слід зазначити що дана мова програмування не є швидкою, паралельні обчислення та в цілому використання ресурсів не є максимально ефективними. У випадку критичної необхідності щодо оптимізації модуля і його прискорення

можна використати мову програмування Go (Golang) як заміну Python. Ця мова програмування була створена і спрямована саме на прискорення мікросервісів, тому вона може бути ідеальним вибором для майбутньої потенційної проблеми [43].

Однак дана мова не має стількох математичних бібліотек і досить молода, тому в ній немає багато готових рішень для проведення обчислень, через що перехід на більш швидкі обчислення потребуватиме додаткових зусиль.

2.4.3 Інструменти розробки клієнтської частини

Для створення клієнтської частини існує багато інструментів. Інтерфейс може бути створений для багатьох платформ, але в даній системі в першу чергу клієнтська частина представляє із себе веб-додаток.

Для створення веб-інтерфейсу існує багато фреймворків, але три основних з них – це React, Angular та Vue [44]. Кожен з цих фреймворків використовує в якості мови програмування Javascript чи Typescript. Для розробки поточної системи було обрано фреймворк React з мовою програмування TypeScript.

Використання мови програмування TypeScript дозволяє так само зробити веб-додаток строго типізованим, що в свою чергу полегшує розробку. Як було зазначено вище, через те що сервер використовує ту ж саму мову програмування що й клієнтська частина – об'єкти, інтерфейси, константи і деякі функції/бібліотеки можуть бути перевикористані в обох частинах. Таким чином розробка рівню домену в клієнтській частині полегшується. Основна увага приділяється розробці логіки взаємодії з сервером да безпосередньої розробки інтерфейсу який буде відображатися на екрані користувача.

Переваги React [45, 46]:

- компонентний підхід. Весь інтерфейс будується на компонентнах, які є окремими незалежними його частинами, що в свою чергу можуть перевикористовуватися в інших компонентах. Також кожна компонента може мати певну логіку, що стосується саме інтерфейсу. Такий підхід дозволяє розбивати інтерфейс на блоки, які можуть знаходитися у правильній ієрархії, що дозволяє більш гнучко їх модифікувати і легше масштабувати інтерфейс;

- велика кількість готових рішень. React має велику кількість бібліотек і готових рішень, Компонетний підхід без обмежень в архітектурі дозволив легко створювати розробникам власні бібліотеки, через що була зібрана їх велика база і одна задача може мати десятки готових варіантів реалізації;

- віртуальний DOM. Використання віртуального DOM надає можливість оновлювати інтерфейс одразу з усіма необхідними змінами і лише тих частин які безпосередньо були змінені, що позитивно впливає на продуктивність;

- відсутність проблем із сумісністю. На відміну від інших фреймворків, дана бібліотека ніколи не має проблем із сумісністю так як її архітектура максимально спрощена і нові версії завжди сумісні з минулими. В інших фреймворках доволі часто можна стикнутися із ситуацією коли корисні бібліотеки не мають аналогу на новій версії чи в поточний момент часу не готові для використання.

Недоліки React [45, 46]:

- відсутність будь якої стандартизації чи вбудованої архітектури. Це вимагає від розробника чіткого розуміння і досвіду побудови додатку та організації коду. Недолік стає критичним у випадку зміни розробників оскільки кожен з них має своє бачення архітектури і у разі використання сторонніх бібліотек, особливо у разі керування станом додатку;

- недостатньо функціоналу без бібліотек. Функціонал який надається з самого початку недостатній для побудови повноцінного додатку, тому необхідно знати і використовувати багато сторонніх бібліотек які виконують навіть такі тривіальні задачі як – маршрутизація, керування формою, валідація та ін;

- важкість керування станом. Немає жодних правил щодо керування станом додатку. Кожна компонента оновлюється і оновлює свої дочірні компоненти у разі зміни її стану. Через це необхідно дотримуватися кращих практик щодо керування станом, щоб чітко планувати оновлення компонентів і робити це оптимально. В сторонніх компонентах завчасно не відомо, як саме відбувається оновлення і іноді це може стати проблемою.

Оцінюючи переваги і недоліки можна побачити, що наявні недоліки не є критичні. Їх можна подолати, завдяки слідуванню правильним підходам до

розробки і використанню правильних бібліотек. Даний фреймворк є набагато гнічкішим і легшим за раніше описані Angular та Vue. Angular має весь необхідний функціонал без бібліотек і чітко сплановану архітектуру, однак він корисний в складних і дуже великих системах і в системах навіть середньої складності не є доцільним. Vue більш легкий, але менш гнучкий і має меншу кількість бібліотек.

Для обробки стану в React існує декілька інструментів – redux, redux-toolkit, mobx та вбудований. Серед всіх варіантів була обрана бібліотека MobX [47].

Вбудований інструмент керування стану і redux бібліотеки будуються на архітектурі Flux. Вони дозволяють керувати інтерфейсом згідно методології реактивного програмування. React компонентни підписуються на оновлення даних і генерує події на які система реагує і оновлює дані в стані додатку, після чого оновлюються компоненти. Однак даний підхід вимагає явного визначення подій, оброблювача події, підписки на оновлення вказуючи конкретні дані які відстежуються і коректного оновлення стану [48].

Бібліотека MobX також втілює методологію реактивного програмування, однак на відміну від інших інструментів дозволяє автоматизувати всі рутинні процеси. З даною бібліотекою створюється клас дані якого є станом, а методи обробниками подій. Компоненти автоматично оновлюються і підписуються на оновлення даних таких класів, до того ж оновлення відбувається лише за необхідності [47]. Дана бібліотека представляє з себе найлаконічніший спосіб керування станом додатку і більш очевидний з точки зору коду. Зважаючи на таку особливість даний інструмент є ідеальним вибором для розробки.

Для спрощеного написання візуального відображення інтерфейсу був обраний фреймворк Tailwind CSS. Даний інструмент – це CSS фреймворк, що має багато готових класів для створення інтерфейсу та спрощені налаштування для створення палітр чи інших констант. Для створення простих інтерфейсів які не використовують складних візуальних ефектів, таких як анімації даного інструменту цілком достатньо [49].

2.4.4 Зовнішні сервіси для розробки системи

Для реалізації деяких задач може бути зручнішим використання зовнішніх сервісів які можуть спростити їх реалізацію. В поточній системі є зовнішні сервіси які виконують відповідно дві задачі – авторизацію та роботу з мапами.

Для спрощення авторизації був обраний сервіс Firebase. Даний сервіс дозволяє налаштувати будь-який тип авторизації, такі як пошта та пароль, телефон та авоторизація через інші сервіси. Firebase зберігає всю необхідну інформацію про користувача, а також створює токени доступу згідно всім вимогам безпеки.

В конкретній реалізація використання сервісу полягає в наступному – на клієнтській частині користувач авторизується через сервіс і отримує токен, далі цей токен відправляється на сервер де він розшифровується також через FireBase і з нього отримуються всі необхідні дані. Сервер має спеціальний файл з конфігурацією для цього сервісу в якому знаходяться ключі необхідні для роботи з ним. Це дозволяє не пересилати і не збирати конфіденційну інформацію з користувача напряду і не пересилати через HTTP запити. В свою чергу токен який пересилається не може бути розшифрований кимось хто не має коректного кофігураційного файлу [50].

Зважаючи на дані можливості досягається вимога безпеки керування даними, а також відкриваються можливості для масштабування авторизаційного модуля в системі.

Для відображення мапи було обрано сервіс Google Maps. Даний сервіс надає всі необхідні можливості, він дозволяє працювати з мапою за допомогою власного апі і кастомізувати її будь-яким чином додаючи власні позначки, маршрути та ін.

Для отримання розрахункових параметрів такі як відстань між пунктами призначення чи час який необхідно витратити для досягнення цілі, було обрано сервіс OpenrouteService. Даний сервіс є опенсорс проектом, який збирає базу від своїх користувачів і в свою чергу є безкоштовним. Це робить його більш пріоритетнішим за використання Google Maps, оскільки його функціонал є доволі вартісним, через наявність додаткового функціоналу який не потрібен системі чи може бути замінений дешевою альтернативою [51].

Для поточної системи даний сервіс надає всю необхідну інформацію яку можна використовувати для обробки. Обчислювальному модулю необхідна інформація щодо параметрів які відповідають самому маршруту між двома точками, а користувач має бачити побудований маршрут і мати можливість відстежувати його.

2.5 Реалізація системи

Остаточну схему реалізації можна представити у вигляді як це показано на рисунку 2.5. Дана схема представляє собою завершену діаграму, зі всіма зв'язками між компонентами системи і інструментами що в них використовуються.

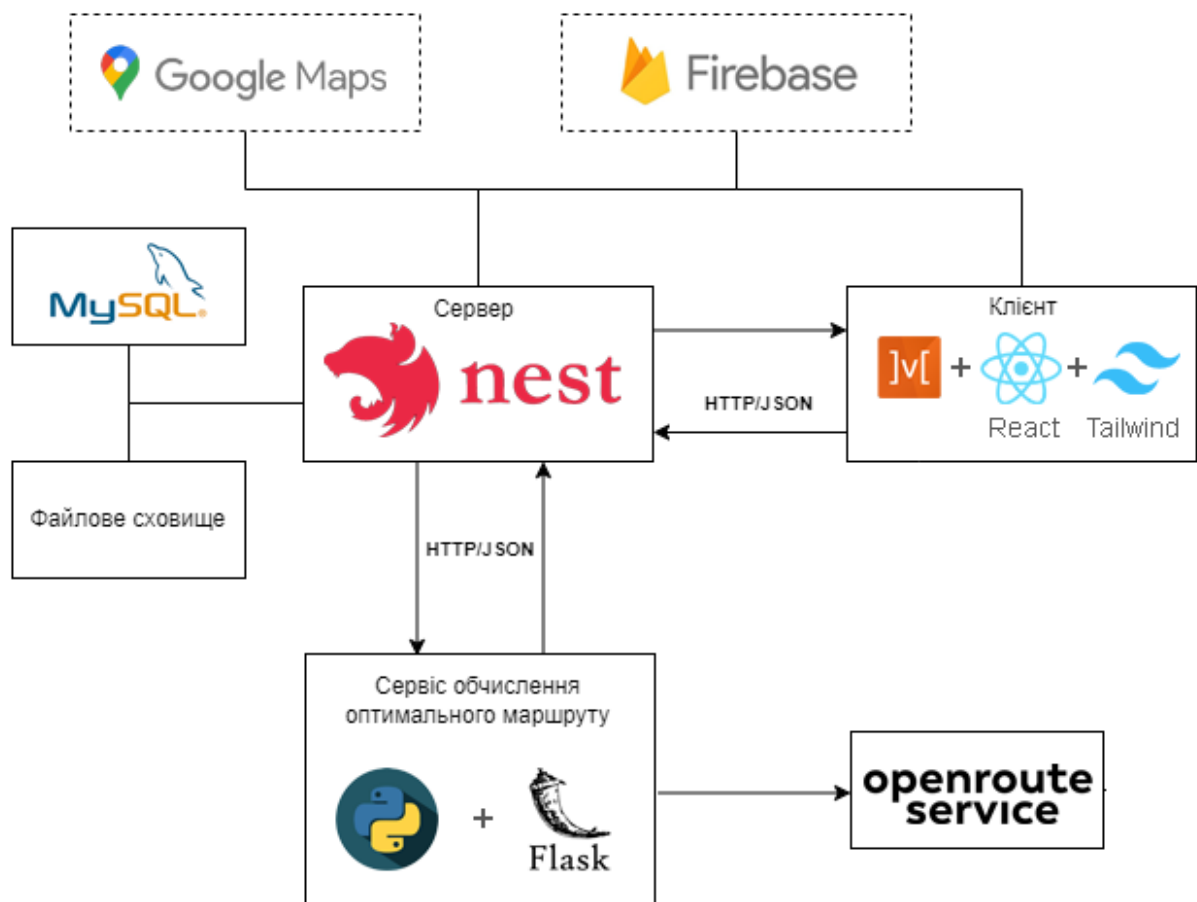


Рисунок 2.5 – Отримана повна діаграма поточної системи

Першим етапом роботи з системою є реєстрація в ній. Вона відбувається без особливих додаткових налаштувань, все що необхідно зробити – це налаштувати Firebase сервіс. Необхідно створити в спеціальній Google консолі проект, в якому задаються методи авторизації і далі необхідно буде лише забрати ключі для серверної частини і клієнтської.

Далі система працює наступним чином. Коли користувач заходить у систему він бачить вікно яке зображено на рисунку 2.6 і може зайти у додаток за одним з методів. Якщо все зроблено вірно то користувач отримує спеціальний idToken який потім передається на сервер і у випадку якщо той налаштований правильно то можна за допомогою цього токена отримати такі дані як email, ім'я користувача, фото та інші дані.

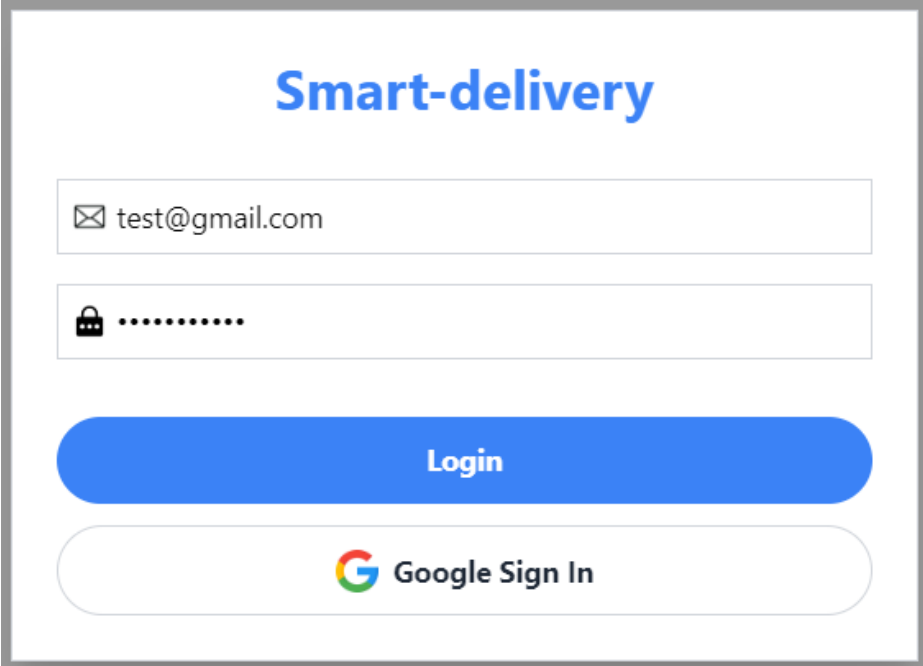
The image shows a login interface for an application named "Smart-delivery". At the top, the title "Smart-delivery" is displayed in a blue, sans-serif font. Below the title, there are two input fields. The first field contains an email address "test@gmail.com" and has an envelope icon on the left. The second field contains a password represented by a series of dots and has a lock icon on the left. Below these fields are two buttons. The first button is blue with rounded corners and contains the text "Login" in white. The second button is white with rounded corners and contains the Google logo followed by the text "Google Sign In".

Рисунок 2.6 – Вікно реєстрації

Після авторизації користувач входить до головного меню де є декілька основних панелі. Користувач має змогу зайти до основної панелі керування, де він в ролі адміністратора має можливості керування усіма необхідними компонентами системи.

можливостей. Через це доцільно замінити функціонал на будь-який інший сервіс що допоможе намалювати маршрут за переданими координатами за більш меншу ціну. Самі кур'єри можуть використовувати свої клієнтські мапи і їх GPS функціонал який є повністю безкоштовним.

Якщо натиснути на будь-який маркер замовлення чи саме замовлення які знаходяться на боковій панелі – можна відкрити вікно для редагування, яке продемонстровано на рисунку 2.8. Дані форми присутні для будь-якої сутності і мають схожий вигляд. За допомогою них можна керувати даними і вносити дані в них за необхідністю.

Список замовлень

Пошук...

Замовлення #1236187123
Відправник: Наталя Іванівна
Телефон: 123-456-7890
Адреса: вулиця Миколи Киценка, 64, Запоріжжя, Запорізька область, Україна, 69000

Замовлення #1236187124
Відправник: Андрій Леченко
Телефон: 987-111-5579
Адреса: вулиця Гончарова, 31А, Запоріжжя, Запорізька область, Україна, 69000

Замовлення #1236187125
Відправник: Петро Павлик
Телефон: 117-769-3850
Адреса: вулиця Іванова, 31, Запоріжжя, Запорізька область, Україна, 69000

Замовлення #1236187126
Відправник: Михайло
Телефон: 235-996-0954
Адреса: вулиця Троїцька, 27, Запоріжжя, Запорізька область, Україна, 69000

Замовлення #1236187123

Статус: Processing

Відправник →

Отримувач ↓

Ім'я: Андрій Прізвище: Ліщенко

Телефон: 087-223-3224 Код: 324897

Опис ↓

Режим доставки: Стандарт	Оплачує: Отримувач	Вага: 500 г
Габарити: 100x20x60	Ціна товару: 30 000	Ціна доставки: 150

Адреса: вулиця Миколи Киценка, 64, Запоріжжя, Запорізька область, Україна, 69000

Рисунок 2.8 – Демонстрація форми замовлення

Також для моніторингу і додаткової логіки в системі відстежується поточне перебування кожного з працівників, щоб визначити потенційні можливості до виконання того чи іншого замовлення. Необхідно для того щоб врахувати наявності машин, чи наявності працівників взагалі які мають змогу довести той чи інший товар.

Однією з основ системи є так само її алгоритм аналізу замовлення і прийняття рішень щодо розбиття замовлень по групах, які можуть бути виконані за одну

поїздки, та розбиття цих груп за відділеннями які можуть виконати замовлення, беручи лише наближені точки. На рисунку 2.9 продемонстровано умовно як система розбиває замовлення. Кожне відділення за можливості оброблює найближчі замовлення.

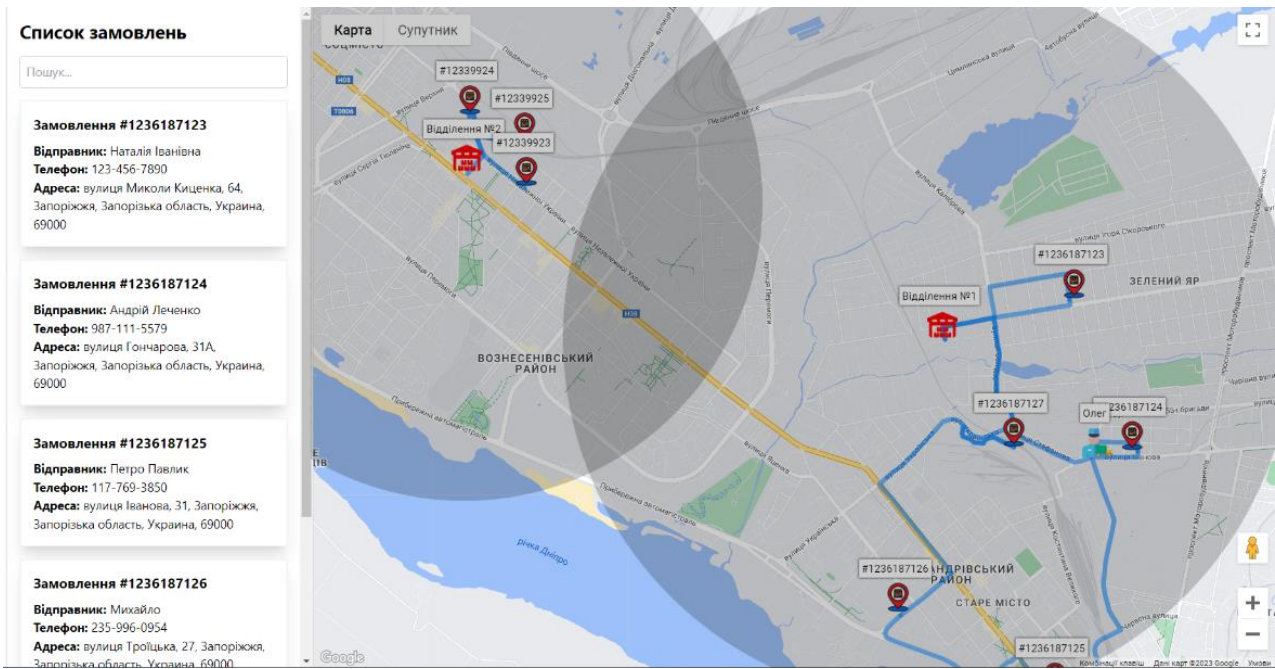


Рисунок 2.9 – Розбиття замовлень за відділеннями

Для отримання матриць дистанцій та часу для кожної групи замовлень використовується OpenRouteService. Приклад матриці приведено на рисунку 2.10, для кращого відображення, матриця була взята з офіційної документації API.

Durations	9.700817, 48.476406	9.207773, 49.153882	37.572963, 55.801279	115.665017, 38.100717
9.700817, 48.476406	0 sec	5697.92 sec	92554.48 sec	395657.06 sec
9.207773, 49.153882	5544.28 sec	0 sec	91234.76 sec	394337.31 sec
37.572963, 55.801279	92220.66 sec	90592.18 sec	0 sec	304513.56 sec
115.665017, 38.100717	394988.88 sec	393360.41 sec	303900.22 sec	0 sec

Рисунок 2.10 – Матриця дистанцій із servісу OpenRouteService

Основний алгоритм системи можна описати за схемою наведеною на рисунку 2.11.

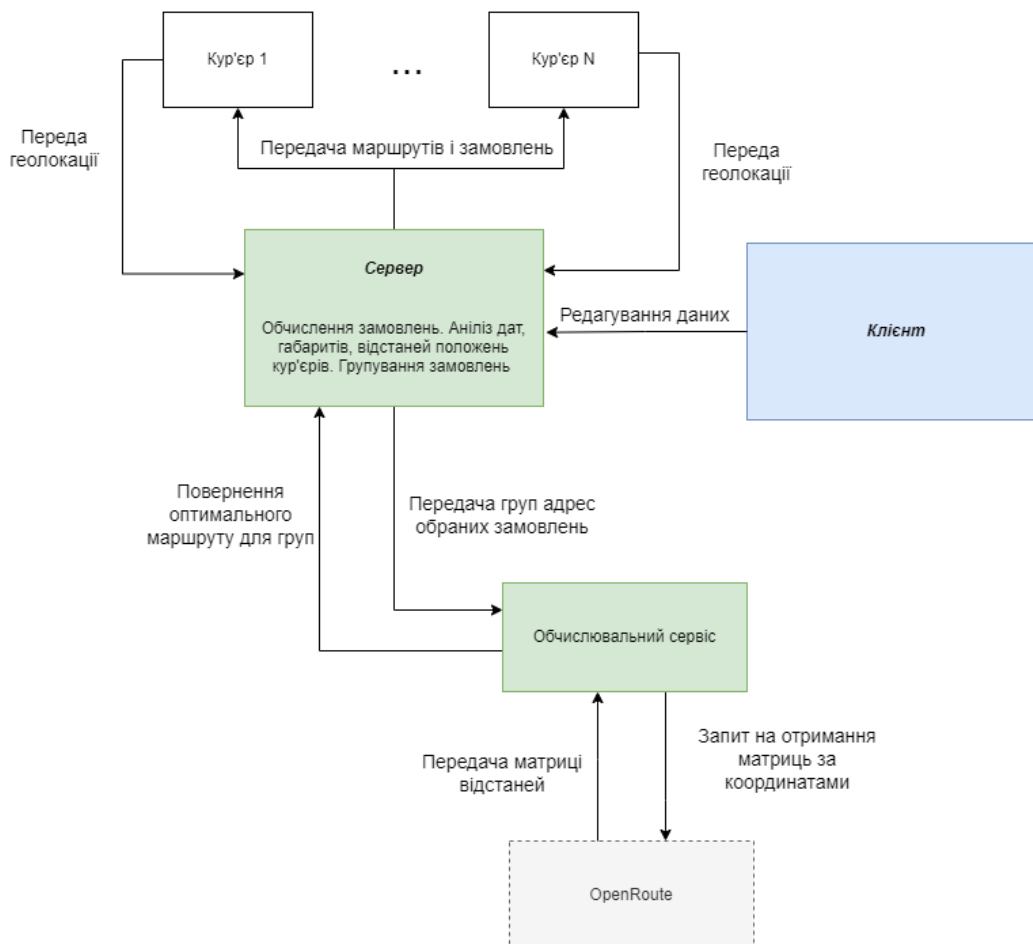


Рисунок 2.11 – Схема алгоритму обробки замовлень в системі

2.6 Висновки за розділом 2

Проаналізовано основні етапи і сучасні підходи до розробки системи. Проаналізовані такі аспекти як побудова архітектури системи, було розглянуто монолітну і мікросервісну архітектури. Проаналізовано сучасні підходи до створення клієнт-серверної взаємодії, сучасні підходи до обміну даними між різними частинами системи. Досліджено архітектурний підхід для побудови

кодової бази системи і наявні інструменти, що були використані для реалізації системи.

Реалізовано системи для автоматизації сервісу доставки яка може сама аналізувати замовлення, визначати оптимальний шлях виконання замовлення – розбиття їх за можливістю виконання в одну поїздку, знаходження оптимального шляху доставки, моніторинг, аналіз поточних ресурсів на складах та ін.

Створена система є універсальною за своєю логікою і може використовуватися з будь-якими суміжними підобластями. Тобто дана система може бути використана у більших масштабах враховуючи всю країну чи використана у іншій сфері, як наприклад військовій.

Дана система підходить добре саме для задач коли необхідно доставити якийсь з товарів чи ресурсів без прив'язки до конкретного місця чи сильної прив'язки до часу. Наприклад у сфері розумних магазинів де важливо лише доставити товар з будь-якої можливої точки покупцеві чи військова промисловість де снаряди чи інше спорядження має бути доставлено якомога швидше підрозділам без прив'язки до чіткого складу.

3 АНАЛІЗ ТА ДОСЛІДЖЕННЯ СИСТЕМИ

Основою даної системи є її алгоритм пошуку оптимального шляху. Система після аналізу наявних замовлення і їх розбиття на групи які можуть бути виконані за один раз, має провести розрахунок оптимального шляху, для того аби швидше виконати їх.

В якості основи для алгоритму був обраний один із методів штучного інтелекту – генетичний алгоритм. Генетичний алгоритм не має чіткої реалізації, а представляє собою базовий алгоритм з певними кроками – мутація, кросовер, відбірка та ін. Кожен з цих етапів може сам представляти собою певний алгоритм.

Їх правильний підбір, та підбір впливаючих на систему коефіцієнтів – є основним фактором, що визначає ефективність системи.

В поточній системі запропонована система, що використовує генетичний алгоритм з попереднім аналізом графу і відбору параметрів за допомогою пошуку мінімального остовного дерева (MST).

3.1 Ідея використання мінімального остовного дерева

Користь використання мінімального остовного дерева для вирішення задачі комівояжера генетичним алгоритмом полягає у тому, що дана структуру може надати гарні початкові умови і межі, з якими можна спробувати оптимізувати алгоритм.

3.1.1 Знаходження меж оптимального рішення

Першим і очевидним параметром який можна отримати після побудови дерева – нижня та верхня межа довжини шляху оптимальних рішень. Мінімальне остовне дерево представляє собою сукупність всіх мінімальних ребер наявних у графі що об'єднують всі вершини. Якщо спробувати обійти дерево то у кращому випадку сума всіх довжин ребер виявиться найкращим рішенням, в гіршому кожне ребро доведеться максимум обійти двічі, через що рішення буде максимум у двічі довше за оптимальне.

Виходячи з цього, можна визначити що оптимальні рішення необхідно шукати серед варіантів, чия довжина не перевищує $2N$, де N це сума всіх довжин ребер остовного дерева.

В контексті генетичного алгоритму, можна модифікувати за допомогою даного критерію такі кроки як – мутацію та вибірку наступного покоління. В поточній системі вони обидва були модифіковані.

Ідея вибірки кращого покоління для його розмноження полягає у наступному видаляючи з колонії заздалегідь відоме неоптимальне рішення, з поступовим з

наближенням верхньої межі до нижньої. Таким чином алгоритм фільтрації полягає у тому що на кожному кроці видаляються неоптимальні індивіди, з додатковою перевіркою кількості тих що “вижили”. Якщо кількість решти індивідів достатньо висока, алгоритм зсуває межу і проводить відсіювання знову доки не пройде межу кількості популяції. У випадку якщо не вдалося зсунути межу, спроба збільшує спеціальний лічильник, який у разі свого переповнення надає сигнал що колонія більше не здатна отримати кращу величину і потенційно можна вважати що її еволюція дійшла до піку, що є кінцем виконання алгоритму.

Мутація в свою чергу опирається на лічильник. Чим ближче наближається момент переповнення лічильника і закінчення виконання алгоритму – тим частіше починають відбуватися мутації. Ідея полягає у тому, що у випадку коли колонія не здатна дати нове рішення за допомогою механізму кросоверу, це може потенційно означати що в неї не вистачає генетичного різноманіття і необхідно збільшуватися вірогідність мутації аби його спробувати створити.

Таким чином можна вважати, що таке використання меж імітує алгоритм природнього відбору в дикій природі. Система постійно погіршує умови які змушують колонію швидше змінюватися, аби підвищити вірогідність до виживання. На рисунку 3.1 зображено візуалізацію у вигляді графіку залежності кількості індивідів від рішення, яке вони собою представляють.

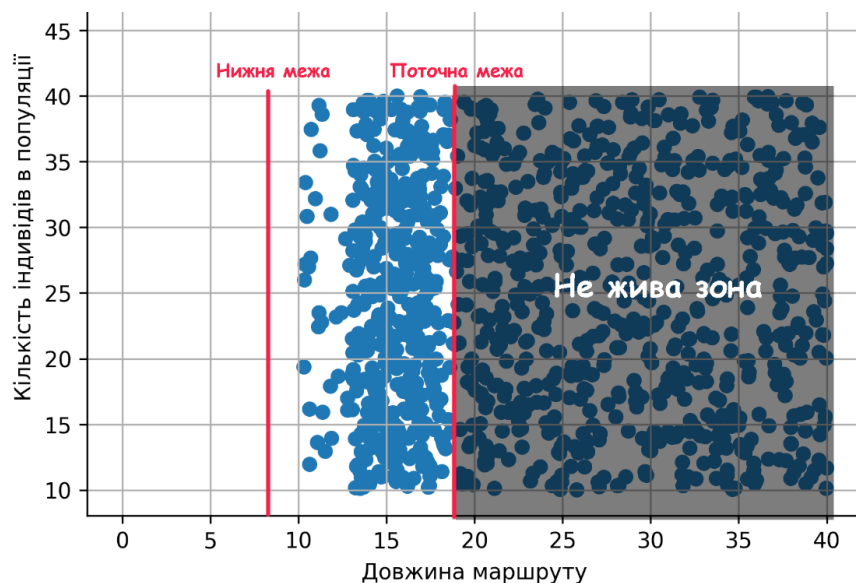


Рисунок 3.1 – Візуалізація поведінки колонії при відборі з межею

З рисунку 3.1 можна побачити, що більшість рішень збирається біля межі, яка поступово їх відсіює, нижня межа майже ніколи не досяжна.

3.1.2 Створення початкової сильної популяції

Мінімальне остовне дерево дозволяє за допомогою простих алгоритмів обходу графу побудувати шлях, що буде попадати у діапазон рішень між раніше згаданими межами. Дане твердження доводиться тим, що найгірший шлях між будь-якими двома вершинами – це шлях по ребрам мінімального остовного дерева, у кращому випадку буде існувати інший більш короткий шлях, який автоматично буде враховано.

На рисунку 3.2 проілюстрована побудова оптимального шляху використовуючи мінімальне остовне дерево та використовуючи обхід у глибину, можна побачити, що шлях буде кращим за найгірший випадок, однак не буде остаточним рішенням.

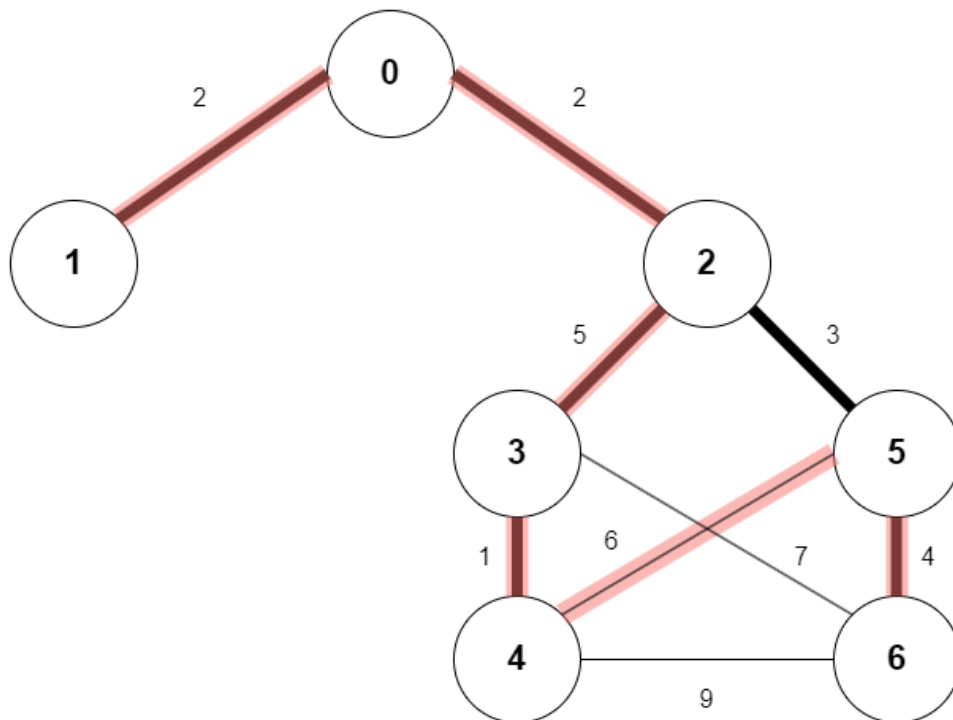


Рисунок 3.2 – Здобуття шляху обходу графу із мінімального остовного дерева

Як можна побачити чорним кольором виділене мінімальне остовне дерево, а червоним напівпрозорим кольором виділено шлях побудований пошуком у

глибину. Якщо пошукати оптимальний шлях, можна побачити що більш коротким є шлях 1-0-2-5-6-3-4 оскільки його сума надає відстань 19, в той час як наведений шлях має відстань 20. Однак такий шлях все одно знаходиться в оптимальному діапазоні.

Слід зазначити, що для простоти спеціально був наведений простий граф з малою кількістю ребер. В реальності такі прості графи є гіршими випадками для використання таких оптимізаційних алгоритмів як генетичний, оскільки кількість їх рішень мала і перебір спрацює швидше ніж буде проведено хоча б десятків генерацій популяцій. Найчастіше ж трапляються задачі з пошуком шляху у графів що наближаються до повних, тобто більшість вершин зв'язана з кожною іншою через ребро. На рисунку 3.3 представлено шлях з 20 точок, кожні з яких не мають обмежень і вважаються зв'язаними між собою кожна з кожною.

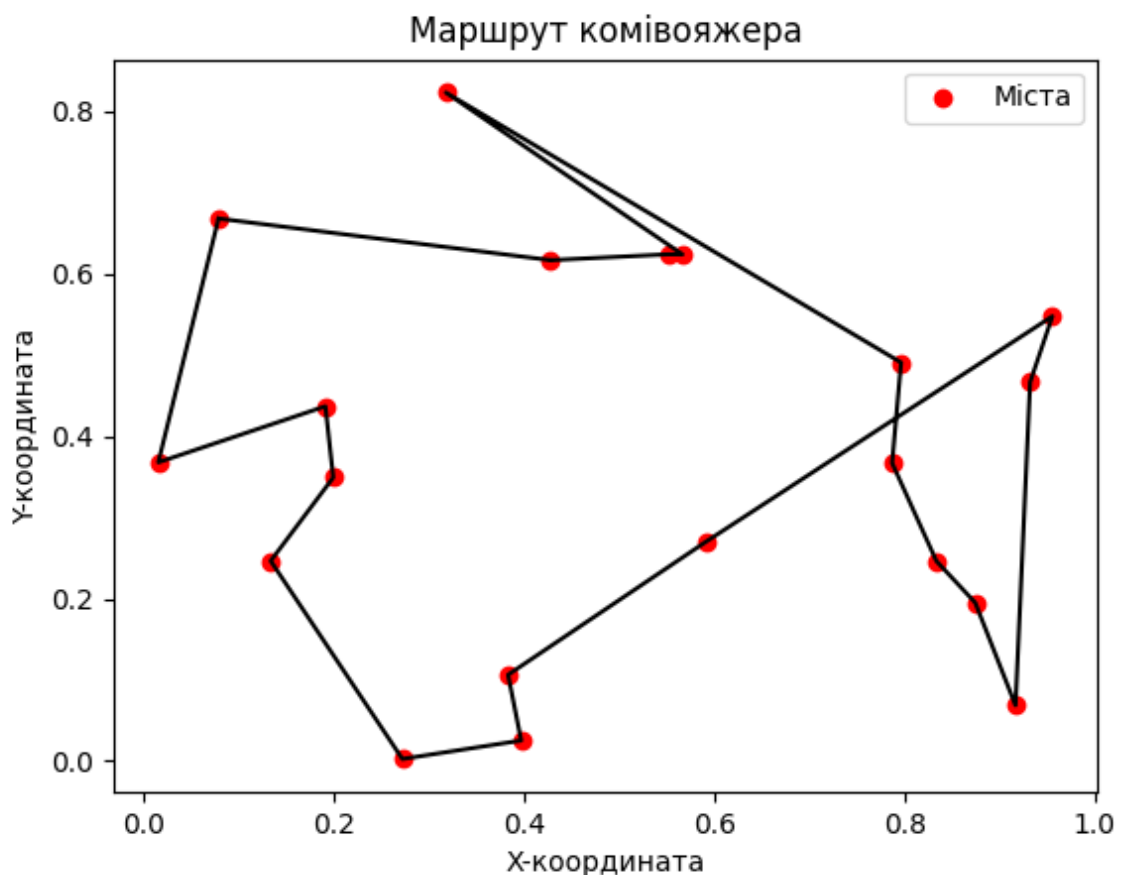


Рисунок 3.3 – Здобуття шляху обходу графу із 20 містечок використовуючи MST

В даному випадку граф стає більш складним, однак видно, що в деякій мірі отриманий шлях є більш менш оптимальний лише з однією явною петлею і не доцільним розворотом.

Знайдений шлях можна одразу додавати як одне з найкращих рішень в популяцію, а його довжину можна спробувати взяти як нову верхню межу і зсунути її ближче. Найцікавішою особливістю такого шляху є те, що його підграфи можуть становити самі по собі в деякій мірі оптимальне рішення. Зробивши дане припущення можна зробити висновок, що якщо спробувати використати ці локальні оптимуми і правильно побудувати між ними мости – можна з певною вірогідністю отримати оптимальне рішення. На рисунку 3.4 візуально продемонстрована ідея взяття підграфів і будування між ними мостів виходячи з попереднього побудованого шляху на рисунку 3.3.

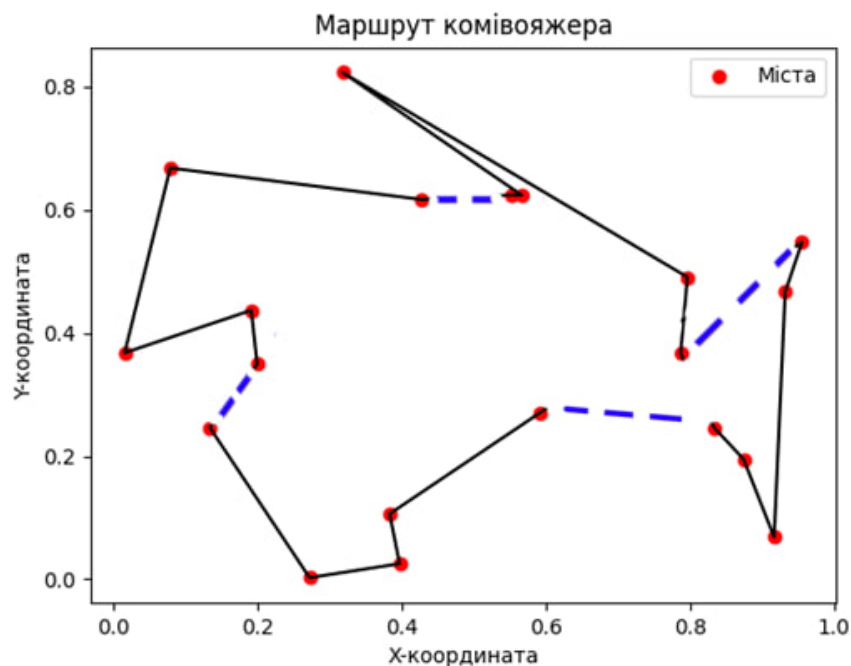


Рисунок 3.4 – Можливий індивід після побудови мостів в підграфах по 5 вершин

З даного прикладу можна побачити що наприклад побудова мостів між підграфами може вирішити проблему виникнення петлі і усунути її ще на моменті генерації першої колонії.

Базовий алгоритм генерації представляє з себе просту генерацію послідовних чисел від 0 до $N-1$, де N це кількість міст, а далі випадкове їх перемішування. В поточній системі пропонується алгоритм який буде брати за основу початковий знайдений маршрут з діапазону потенційно оптимальних, і на його основі буде створювати першу колонію.

Ідея алгоритму наступна – береться масив точок що визначає шлях, створюється другий масив який відповідає за діапазон точок у підграфах який буде використовуватися для тасування точок у шляху. На кожному кроці доки колонія не досягла потрібного розміру, по чергово береться діапазон із масиву діапазонів і випадково ці шматки перетасовуються утворюючи новий масив. Для того, щоб зберегти генетичне різноманіття і підграфи не повторювалися, після того як масив діапазонів був повністю пройденим, проводиться циклічний зсув графу, аби зсунути таким чином послідовності у наступних шматках шляху. Отримана колонія потенційно матиме найкращу пристосованість і більше швидко має досягти найкращого рішення. Додатково з ціллю поліпшення генетичного різноманіття вводиться невелика кількість індивідів, що згенерована випадковим чином

Алгоритм випадкової генерації першої колонії з використанням базового шляху наведено у лістингу 3.1.

Лістинг 3.1 – Реалізація алгоритму генерації початкової популяції

```
def chunk_shuffle(data, chunk_size):
    chunks = [data[i:i + chunk_size] for i in range(0, len(data),
chunk_size)]
    random.shuffle(chunks)
    shuffled_data = [item for sublist in chunks for item in sublist]
    return shuffled_data

def generatePopulation(original_array, chunk_sizes, total_arrays):
    shuffled_arrays = []
    size = total_arrays / len(chunk_sizes)
    for _ in range(int(size)):
        original_array = cyclic_shift_right(original_array, 1)
        for chunk_size in chunk_sizes:
            shuffled_arrays.append(chunk_shuffle(original_array,
chunk_size))

    return shuffled_arrays
```

3.1.3 Мутація популяції

Одну з основних ролей в підході із генерацією популяцій через найкращий індивід відіграє мутація. В біології існує поняття обмеженого генетичного різноманіття – це ситуація в якій популяція не має достатнього різноманіття генів, для отримання нових індивідів і колонія від колонії майже нічим не відрізняється і ніяк не адаптується. Єдиним можливим варіантом вирішення такої проблеми стають мутації, які випадковим чином починають змінювати генетичний код і утворювати нові комбінації.

В поточній системі алгоритм мутації побудований на ідеї того, що у випадку коли індивіди перестають надавати довгий час краще рішення – вони починають швидше мутувати. Алгоритм мутації наведено в лістингу 3.2.

Лістинг 3.2 – Реалізація алгоритм мутації

```
def mutate(self, order, mutation_rate):
    if random.random() < mutation_rate:
        idx1, idx2 = random.sample(range(self.num_cities), 2)
        order[idx1], order[idx2] = order[idx2], order[idx1]

    return order

def adapt_mutation_probability(self, base_mutation_rate):
    return base_mutation_rate + (self.doom_skipped /
self.doom_stop) * base_mutation_rate
```

В даному алгоритмі використовуються два параметри `doom_skipped` та `doom_stop` – так звані лічильник кінця існування популяції без змін та максимальна кількість циклів без змін. Таким чином чим ближче підходить час до завершення алгоритму, тим вірогідніші стають мутації. В даному випадку шанс збільшується вдвічі, коли таймер доходить до кінця.

Через значний вплив на систему, даний параметр має бути ретельно протестований, чи навчений на хорошій моделі, оскільки занадто високий параметр псує популяцію. При великих значеннях може втратити в один момент найкращі рішення і деградувати, занадто низький показник призводить до нестачі генетичного різноманіття і нове рішення не знаходиться взагалі.

3.1.4 Модифікований відбір

Для подолання проблем обмеженості генетичного різноманіття через використання мінімального остовного дерева, було проведено декілька експериментів з алгоритмом відбору. Фінальна версія алгоритму наведена у лістингу 3.3.

Лістинг 3.3 – Реалізація алгоритму вибірки

```
def select_best_individuals(self, new_population):
    new_population = sorted(new_population, key=lambda x:
self.calculate_distance(x))[:self.population_size]
    pure_population = list(filter(lambda x:
self.calculate_distance(x) < self.max_length, new_population))
    while len(pure_population) >= self.population_size:
        self.max_length -= self.doom_tick
        self.doom_skipped = 0
        pure_population = list(filter(lambda x:
self.calculate_distance(x) < self.max_length, pure_population))

    self.doom_skipped += 1
    if len(pure_population) > self.population_size * 0.5:
        return sorted(pure_population)[:self.population_size]
    elif is_best:
        pure_population = sorted(pure_population)
        if len(pure_population) == 0:
            pure_population =
new_population[:int(self.population_size * 0.25)]
        if len(pure_population) < self.population_size * 0.5:
            chunk_sizes = [1, 2, 4, 4, 7, 7]
            population = get_shuffled_arrays(pure_population[0],
chunk_sizes, self.population_size * 0.5 - len(pure_population))
            pure_population.extend(population)
        return sorted(pure_population)
```

В запропонованому алгоритмі селекція йде за ідеєю очищення по межі. Пропонується очищення популяції до кількості не нижче ніж на половину від заданого значення алгоритму на етапі ініціалізації. Якщо дана подія відбулася – береться чверть найкращих рішень з минулої популяції, а друга чверть генерується за алгоритмом генерації з найкращого поточного рішення.

Дана модифікації селекції, допомагає частково подолати проблему нестачі генів і суттєво пришвидшити швидкість знаходження оптимального рішення.

3.2 Аналіз роботи системи і тестування

На рисунку 3.5 наведено приклад роботи отриманого алгоритму. Зліва наведено початковий маршрут, з якого створюється початкова популяція. Праворуч розташований знайдений маршрут – вирішення задачі комівояжера. Як можна побачити з такого прикладу, початковий маршрут вже має необхідні оптимальні шматки, тому є велика вірогідність що оптимальний маршрут може бути знайдений ще на етапі ініціалізації першої популяції.

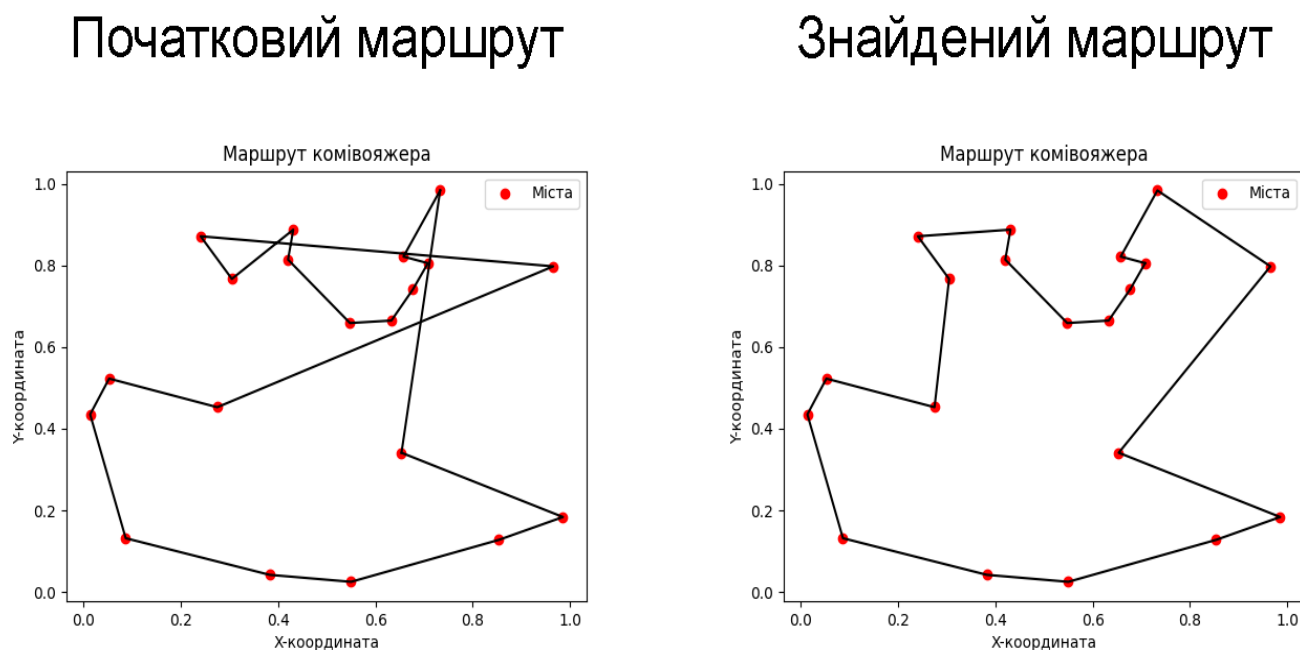


Рисунок 3.5 – Знаходження оптимального шляху

В таблиці 3.1 результати у часі і довжині для алгоритму з використанням генерації популяції через використання основи у якості знайденого рішення через мінімальне остовне дерево та без. Також наведено спеціальний параметр дельта який визначає на скільки сильно відрізняється рішення двох рішень. Позитивне значення вказує на те що алгоритм з використанням отримав кращий результат, від'ємне – гірший.

Таблиця 3.1 – Результати тестування алгоритму з основою та без

№	Алгоритм з основою		Алгоритм без основи		Δ (відстань)	Δ (час)
	Відстань	Час (сек)	Відстань	Час (сек)		
1	3.985	8.592	4.041	10.595	0.014	0.189
2	3.592	6.129	3.592	9.666	0	0.366
3	4.169	8.636	4.169	11.103	0	0.222
4	3.753	7.627	3.769	10.849	0.004	0.297
5	4.546	7.291	4.429	11.486	-0.026	0.365
6	3.617	10.345	3.798	13.586	0.048	0.239
7	4.330	9.531	4.236	5.301	-0.022	-0.798
8	3.316	9.337	3.305	11.837	-0.003	0.211
9	3.259	9.884	3.146	9.342	-0.036	-0.058
10	4.099	7.290	4.188	11.954	0.021	0.390
11	3.697	7.061	3.682	9.527	-0.004	0.259
12	4.074	6.811	4.101	8.235	0.007	0.173
13	3.925	5.367	3.925	9.502	0	0.435
14	3.557	9.505	3.486	9.563	-0.021	0.006
15	3.486	9.563	4.587	9.969	0.240	0.041
16	3.592	9.969	3.591	10.268	0	0.029
17	4.327	9.573	4.411	5.246	0.019	-0.825

За даними з таблиці можна побачити що алгоритм без основи працює в середньому повільніше, а в деяких випадках стільки ж часу скільки й алгоритм з основою. Обидва алгоритми видають значення що відрізняють один від одного не більше ніж на 5%, при чому алгоритм з основою частіше знаходить більш оптимальне значення. Однак слід зазначити що іноді відіграє роль фактор рандомізації значень і алгоритм без використання основи отримує випадковим чином краще рішення ще на етапі генерації значень. Таким чином іноді стаються випадки коли він спрацьовує навпаки швидше чи отримує більш оптимальне значення. За експериментальною оцінкою таких випадків не більше 10%.

Виходячи з цього можна зробити висновок, що використання оптимальної початкової вибірки і її корекції під час відбору у середньому може надати більш швидше оптимальну відповідь. Однак також слід зазначити, що іноді можна побачити вплив фактору нестачі генетичного різноманіття, оскільки випадкова генерація іноді може надати оптимальне рішення з більшою вірогідністю, ніж випадкове тасування підграфів.

Доцільним також є перевірити ідею використання основи з іншими прийнятими реалізаціями генетичних алгоритмів. На рисунку 3.6 наведено графік залежності часу роботи генетичних алгоритмів від кількості вершин. Для тестування був обраний стандартний алгоритм використовуючий фітнес функцію як критерію для проведення мутацій та відбору батьків з критерієм остановки через перевірку наявності зміни популяції. Даний алгоритми є стандартним і універсальним для будь якої задачі. В якості іншого алгоритму були переглянуті найкращі реалізації з готовими коефіцієнтами.

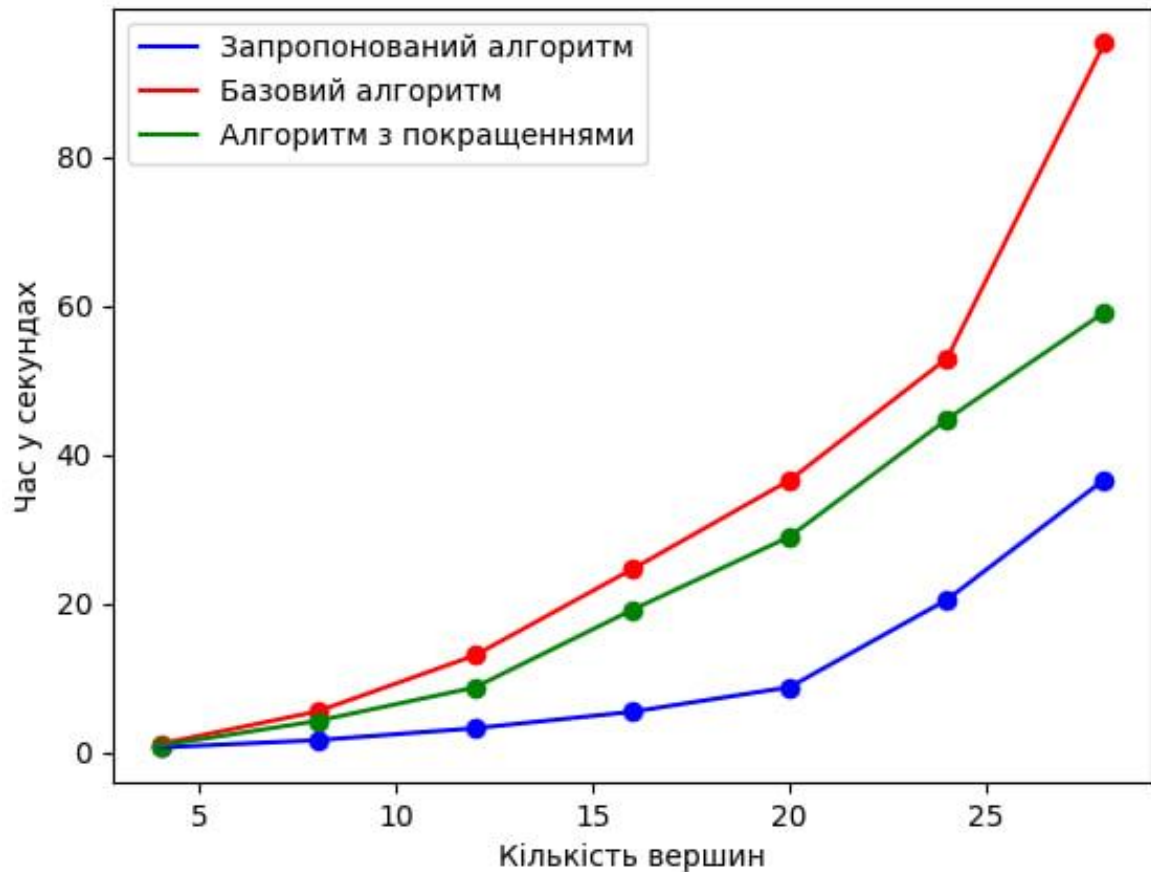


Рисунок 3.6 – Залежність часу роботи від кількості вершин в різних алгоритмах

Графіки зображені зверху вниз – базовий алгоритм, з покращеннями та запропонований відповідно. З графіку можна побачити що алгоритм з основою для вирішення задачі комівояжера працює так само швидше. Однак слід зазначити, що зі зростанням кількості вершин – зростає і вірогідність помилки в розрахунку алгоритмом з основою. Алгоритм з покращеннями видає більш часто оптимальніший варіант маршруту на великій кількості вершин. В гірших випадках алгоритм з основою починає мати похибку приблизно у 10-12%.

Також у таблиці 3.2 наведено ці результати тестування за часом та точністю відповіді. Точність визначається параметром дельта, який визначає у скільки разів рішення гірше відносно реального рішення. В якості дослідження параметру дельта було взято одні з найгірших показників, для визначення потенційної помилки.

Таблиця 3.2 – Результати роботи трьох алгоритмів

Кількість вершин	Час виконання (t, сек)			Похибка рішення (Δ , разів)		
	$t_{з\text{ осн.}}$	$t_{\text{базовий}}$	$t_{з\text{ покращ.}}$	$\Delta_{з\text{ осн.}}$	$\Delta_{\text{базовий}}$	$\Delta_{з\text{ покращ.}}$
4	0.672	1.154	0.924	0	0	0
8	1.655	5.510	4.243	0	0	0
12	3.257	13.037	8.735	0	0	0
16	5.468	24.615	22.153	0	0.002	0
20	7.767	36.551	29.972	0.002	0.008	0
24	17.511	52.904	40.736	0.01	0.012	0
28	30.645	95.227	52.096	0.019	0.016	0.01

З таблиці можна побачити що при збільшенні кількості міст, може почати зростати також і похибка рішення, однак на 28 вершина вона сягає не більше 3%. Навіть більш, менш навчений алгоритм може почати надавати невелику помилку у рішенні. Основним фактором в цьому є саме підбір оптимальних параметрів, зміна яких безпосередньо впливає на точність та час виконання. Неправильне налаштування з основою причиною, за якою точність двох алгоритмів – базового

і з основою, стає значно гіршою з ростом вершин ніж у навченого алгоритму, а також причиною тривалого виконання алгоритму, при досягненні тих самих результатів.

Даний підхід до створення генетичного алгоритму з використання допоміжних методів і ідеї пов'язаних з конкретною задачею, як генерація оптимальної початкової колонії, можуть пришвидшити виконання алгоритму. Однак ціною використання даного підходу – є втрата універсальності генетичного алгоритму. Розглядаючи дану реалізацію, стає зрозуміло що алгоритм став конкретизованим под задачу комівояжера і більше не може бути використаний у інших задачах.

Втрата універсальності може стати критичним чинником і ускладнити налаштування алгоритму. Алгоритм використовує конкретизований підхід може мати додаткові, унікальні для нього параметри, чи унікальний підхід для будь-якого з етапів його роботи, що в свою чергу вимагає додаткових керегувань, складнішого тестування та навчання алгоритму.

Досліджуючи поточний алгоритм можна визначити декілька унікальних для генетичного алгоритму параметрів – величина підграфів які піддаються випадковому перемішуванню, частота для групи підграфів, швидкість зсуву межі та вплив нестачі генетичного різноманіття.

Для швидкості сходження популяцій, дуже сильний вплив має правильність перемішування підграфів. Даний параметр не є простим коефіцієнтом, а визначає собою масив чисел який відповідає шматкам які перемішуються, що ускладнює підбір правильного показника, зважаючи на те що для різною кількості вершин він має динамічно змінюватися. Експериментально було визначено, що одним з найкращих способів розбиття є розбиття за ступенем двійки, від 1 до $N / 4$, де N кількість вершин, з додаванням у кінець числа $N / 4$, якщо воно відсутнє. До того ж коротші шматки в першій половині масиву мають перемішуватися вдвічі частіше.

Вплив нестачі генетичного різноманіття, також сильно впливає на остаточне вирішення. Однією з основних проблем алгоритму, є помилкове визначення моменту сходження популяції через те, що один з наборів генів швидше витісняє

інші і більшість індивідів стають ідентичним, доходячи до точки коли перестають еволюціонувати. Пошук правильного коефіцієнту і алгоритму мутації може сприяти вирішенню цієї проблеми. Для конкретної реалізації експериментально був визначений коефіцієнт шансу 0.1 з збільшенням його удвічі з приближенням часу таймера.

Одним з найвпливовіших вирішень проблеми нестачі генетичного різноманіття - є саме алгоритм селекції – дана реалізації суттєво пришвидшує алгоритм і допомагає знайти швидше оптимальне рішення з мінімальною похибкою.

Додатковими параметрами даного алгоритму можна визначити швидкість зсуву межі, та алгоритм з якого отримуються оптимальні рішення. Зсув межі не є унікальним варіантом для даного алгоритму, а є аналогом певного критерію для динамічної корекції показників таких як мутації, селекції та ін. Підбираючи алгоритм для пошуку початково рішення можна пробувати розв'язувати петлі за оптимальний час, знаходити оптимальні підграфи та ін. Дані параметри також можуть вплинути на роботу алгоритму.

Виходячи з цих даних можна стверджувати що в даному алгоритму існує один новий параметр що налаштовується безпосередньо – масив шматків та один з параметрів що має значно більшу вагу ніж у тривіальній версії алгоритму – коефіцієнт мутації.

Таким чином можна зробити висновок, що конкретизувавши алгоритм можна пришвидшити його для виконання певних задач, однак за рахунок ускладнення його навчання через ведення додаткових критерії та зміну ваги існуючих.

3.2.1 Використання ChatGPT

Одним з експериментів для дослідження було використання ChatGPT в якості інструменту для підрахунку оптимального маршруту. ChatGPT є новим інструментом який може виконувати будь-які поставлені перед ним задачі. Він навчений більш ніж на 100 ГБ даних і потенційно може надавати оптимальні рішення для подібних задач.

Згідно експерименту на вхід чату подавалися 15 координат, які необхідно було з'єднати у замкнутий шлях який обходить їх всі. Дослідження показали, що дана модель здатна іноді знаходити оптимальні рішення що відрізняються від реального не більш ніж на 10%, що насправді не є хорошим результатом, але може дати поштовх для знаходження більш оптимального.

Однак провівши велику кількість тестувань було встановлено, що здебільшого маршрути які він надає відрізняють від оптимального більш ніж на 30% і попадають у зону між двома межами, які були описані вище. Цей інструмент на жаль не є доцільним для використання у подібних задачах, однак може бути використаний потенційно для генерації початкових колоній чи визначення інших параметрів.

3.4 Висновки за розділом 3

В даному розділі було розглянуто основний алгоритм системи щодо знаходження оптимального шляху між обраними вершинам. Було протестовано ідею використання мінімального остовного дерева в якості основи для здобуття певних параметрів і оптимізації генетичного алгоритму.

Дана ідея демонструє, що генетичні алгоритми хоч і є універсальними, однак в деяких випадках можуть бути використані в поєднанні з іншими алгоритмами, які є більш конкретизованими для вирішення певної сфери задач. З отриманих даних при аналізі було визначено що даний підхід дозволяє оптимізувати стандартні генетичні алгоритми і зробити їх більш швидкими.

Таким чином можна зробити висновок, що в деяких сферах задач можна використати наближені до неї алгоритми, для отримання певних параметрів. Які в свою чергу можуть бути позитивно використані для оптимізації генетичних алгоритмів і зробити його ще ефективнішим, позбувшись його універсальності.

ВИСНОВКИ

Був проведений аналіз предметної області та вимог яких необхідно дотримуватися під час розробки. Були визначені цілі, необхідний функціонал та досліджені існуючі методи реалізації схожих систем.

При розробці системи було проведено аналіз сучасних методів і архітектур для побудови системи, проведено аналіз існуючих бібліотек та фреймворків. Дані інструменти були використані для реалізації необхідного функціоналу та виконання вимог які були наведені у предметній області.

Після детального аналізу було розроблено систему для автоматизації основних бізнес-процесів сервісів доставки. Вона задовольняє всім вимогам і має необхідний функціонал. Створена система є певною мірою універсальною і може бути перевикористана з іншими предметними областями, з невеликими виправленнями.

Дана система має потенціал для майбутньої автоматизації і переведення бізнесу майже до автоматичного рівня. Це потенційно досягається більшим впровадженням прав щодо прийняття рішень системою і усуненням людини від прийняття рішень. Впровадженням автоматичного моніторингу чи безпілотних засобів доставки.

Наукова новизна - запропоновано підхід пошуку оптимального шляху, який використовує комбінацію генетичного алгоритму та алгоритмів пошуку мінімального основного дерева і обходів на графах, що надає можливості для оптимізації і калібрування алгоритму.

Практична цінність - розроблена комп'ютерна система, яка реалізує запропонований підхід та дозволяє пришвидшити виконання пошуку оптимального шляху майже на 30% з відносно мінімальною похибкою.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1 Solving the Travelling Salesman Problem URL: <https://blog.routific.com/blog/travelling-salesman-problem> (дата звернення: 20.09.2023).

2 NP-complete problem URL: <https://www.britannica.com/science/NP-complete-problem> (дата звернення: 20.09.2023).

3 11 Animated Algorithms for the Traveling Salesman Problem URL: <https://stemlounge.com/animated-algorithms-for-the-traveling-salesman-problem/> (дата звернення: 01.10.2023).

4 Three different methods to solve the travelling salesman problem URL: <https://baobabsoluciones.es/en/blog/2020/10/01/travelling-salesman-problem-methods/> (дата звернення: 01.10.2023).

5 Explained: What is Traveling Salesman Problem (TSP) URL: <https://www.upperinc.com/guides/travelling-salesman-problem/> (дата звернення: 01.10.2023).

6 Understanding Branch and Bound in Optimization Problems URL: <https://medium.com/walmartglobaltech/understanding-branch-and-bound-in-optimization-problems-d8117da0e2c5> (дата звернення: 04.10.2023).

7 Traveling Salesperson problem using branch and bound URL: <https://www.javatpoint.com/traveling-salesperson-problem-using-branch-and-bound> (дата звернення: 07.10.2023).

8 Задача комівояжера (TSP) точне рішення - метод динамічного програмування URL: <https://habr.com/ru/articles/701458/> (дата звернення: 10.10.2023).

9 Solving TSP by using combinatorial Bees algorithm with nearest neighbor method URL: <https://link.springer.com/article/10.1007/s00521-022-07816-y> (дата звернення: 11.10.2023).

10 Traveling Salesman Problem: Nearest Neighbor Algorithm Solution URL: <https://blog.devgenius.io/traveling-salesman-problem-nearest-neighbor-algorithm-solution-e78399d0ab0c> (дата звернення: 12.10.2023).

11 Розв'язання задачі комівояжера методом найближчого сусіда на Python URL: <https://habr.com/ru/articles/329604/> (дата звернення: 12.10.2023).

12 Фахівці з інформатики йдуть непройденними дорогами URL: <https://habr.com/ru/articles/399873/> (дата звернення: 14.10.2023).

13 Minimum spanning tree - Kruskal's algorithm URL: https://cp-algorithms.com/graph/mst_kruskal.html (дата звернення: 14.10.2023).

14 Katoch S., Chauhan S. S., Kumar V. A review on genetic algorithm: past, present, and future. Multimedia Tools and Applications. 2020. URL: <https://doi.org/10.1007/s11042-020-10139-6> (дата звернення: 14.10.2023).

15 Evolution of a salesman: A complete genetic algorithm tutorial for Python URL: <https://towardsdatascience.com/evolution-of-a-salesman-a-complete-genetic-algorithm-tutorial-for-python-6fe5d2b3ca35> (дата звернення: 15.10.2023).

16 Held–Karp algorithm URL: https://www.wikiwand.com/en/Held-Karp_algorithm (дата звернення: 20.10.2023).

17 Dorigo M., Caro G. D., Gambardella L. M. Ant Algorithms for Discrete Optimization. Artificial Life. 1999. Vol. 5, no. 2. P. 137–172. URL: <https://doi.org/10.1162/106454699568728> (дата звернення: 20.10.2023).

18 4 types of learning in machine learning explained URL: <https://www.techtarget.com/searchenterpriseai/tip/Types-of-learning-in-machine-learning-explained> (дата звернення: 25.10.2023).

19 Understanding Backpropagation Algorithm URL: <https://towardsdatascience.com/understanding-backpropagation-algorithm-7bb3aa2f95fd> (дата звернення: 29.10.2023).

20 Microservices vs monolith: Which architecture is the best choice for your business? URL: <https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-business/> (дата звернення: 31.10.2023).

21 Architecture Comparison: Monolithic vs Microservices. URL: <https://medium.com/design-microservices-architecture-with-patterns/architecture-comparison-monolithic-vs-microservices-4109265c4806> (дата звернення: 30.10.2023).

22 Microservices vs. monolithic architecture. URL: <https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith> (дата звернення: 30.10.2023).

23 The Clean Architecture — Beginner's Guide. URL: <https://betterprogramming.pub/the-clean-architecture-beginners-guide-e4b7058c1165> (дата звернення: 31.10.2023).

24 MVC, MVP, MVVM: Which One to Choose? URL: <https://www.makeuseof.com/mvc-mvp-mvvm-which-choose/> (дата звернення: 31.10.2023).

25 What Is Reactive Programming? URL: <https://www.baeldung.com/cs/reactive-programming> (дата звернення: 31.10.2023).

26 An introduction to the Flux architectural pattern. URL: <https://www.freecodecamp.org/news/an-introduction-to-the-flux-architectural-pattern-674ea74775c9/> (дата звернення: 31.10.2023).

27 Comparing API Architectural Styles: SOAP vs REST vs GraphQL vs RPC. URL: <https://www.altexsoft.com/blog/soap-vs-rest-vs-graphql-vs-rpc/> (дата звернення: 02.11.2023).

28 Different Types of APIs – SOAP vs REST vs GraphQL. URL: <https://www.freecodecamp.org/news/rest-vs-graphql-apis/> (дата звернення: 02.11.2023).

29 XML Soap. URL: https://www.w3schools.com/xml/xml_soap.asp (дата звернення: 03.11.2023).

30 REST API: Key Concepts, Best Practices, and Benefits. URL: <https://www.altexsoft.com/blog/rest-api-design/> (дата звернення: 04.11.2023).

31 What is the difference between GraphQL and REST? URL: <https://aws.amazon.com/ru/compare/the-difference-between-graphql-and-rest> (дата звернення: 04.11.2023).

32 GraphQL Vs. REST APIs. URL: <https://hygraph.com/blog/graphql-vs-rest-apis> (дата звернення: 04.11.2023).

33 Six Reasons why you should use NestJS. URL: <https://dev.to/kalashin1/six-reasons-why-you-should-use-nestjs-3k8b> (дата звернення: 06.11.2023).

34 Advantages and disadvantages of NestJS. URL: <https://themobilereality.com/blog/advantages-and-disadvantages-of-nestjs> (дата звернення: 06.11.2023).

35 Relational vs. Non-Relational Database: Pros & Cons. URL: <https://aloha.co/blog/relational-vs-non-relational-database-pros-cons> (дата звернення: 06.11.2023).

36 What is the difference between relational and non-relational databases? URL: <https://aws.amazon.com/ru/compare/the-difference-between-relational-and-non-relational-databases/> (дата звернення: 06.11.2023).

37 Why Use PostgreSQL For Your Next Project? URL: <https://cleancommit.io/blog/why-use-postgresql-for-your-next-project/> (дата звернення: 07.11.2023).

38 SQL (TypeORM). URL: <https://docs.nestjs.com/recipes/sql-typeorm> (дата звернення: 07.11.2023).

39 Python libraries math, scipy, numpy, matplotlib. URL: <https://svitla.com/blog/python-libraries-math-scipy-numpy-matplotlib> (дата звернення: 07.11.2023).

40 Flask 3.0.0. URL: <https://pypi.org/project/Flask/> (дата звернення: 08.11.2023).

41 What Is Flask and How Do Developers Use It? A Quick Guide. URL: <https://careerfoundry.com/en/blog/web-development/what-is-flask/> (дата звернення: 08.11.2023).

42 Python Flask: pros and cons. URL: <https://dev.to/detimo/python-flask-pros-and-cons-1mlo> (дата звернення: 08.11.2023).

43 Why Golang Is So Fast: A Performance Analysis. URL: <https://www.bairesdev.com/blog/why-golang-is-so-fast-performance-analysis/> (дата звернення: 08.11.2023).

44 Vue vs. React vs. Angular – Pros, Cons, Use Cases and Comparison. URL: <https://getdevdone.com/blog/vue-vs-react-vs-angular-pros-cons-use-cases-and-comparison.html> (дата звернення: 11.11.2023).

45 Vue vs. React vs. Angular – Pros, Cons, Use Cases and Comparison. URL: <https://www.linkedin.com/pulse/react-jsnode-jspros-cons-shagufta-naz/> (дата звернення: 11.11.2023).

46 Pros and Cons of Using React in Web Development. URL: <https://javascript.plainenglish.io/pros-and-cons-of-using-react-in-web-development-ffb81a6235b9> (дата звернення: 11.11.2023).

47 What to choose: Mobx or Redux? Mobx vs Redux. URL: <https://medium.com/@InspireTech/what-to-choose-mobx-or-redux-mobx-vs-redux-4c624b156ac3> (дата звернення: 12.11.2023).

48 Building Reactive Apps with Redux, RxJS, and Redux-Observable in React Native. URL: <https://www.toptal.com/react-native/react-redux-rxjs-tutorial> (дата звернення: 11.11.2023).

49 Rapidly build modern websites without ever leaving your HTML. URL: <https://tailwindcss.com/> (дата звернення: 11.11.2023).

50 Firebase Authentication. URL: <https://firebase.google.com/docs/auth?hl=ru> (дата звернення: 14.11.2023).

51 Services | Openrouteservice. URL: <https://openrouteservice.org/services/> (дата звернення: 14.11.2023).