Національний лісотехнічний університет України
(повне найменування вищого навчального закладу)

Навчально-науковий інститут комп'ютерних наук
та інформаційних технологій
(повне найменування інституту, назва факультету (відділення))

Кафедра комп'ютерних наук
(повна назва кафедри (предметної, циклової комісії))

Магістерська кваліфікаційна робота

другий (магістерський)
(рівень вищої освіти)

на тему:
«Розроблення рекомендаційної системи закупівель засобами Angular»

Виконав: студент VI курсу групи КН-61м
спеціальності 122 "Комп'ютерні науки"
(шифр і назва напряму підготовки, спеціальності)
Вольський Б. Ю.
(прізвище та ініціали)

Керівник Сторожук О. Л.
(прізвище та ініціали)

Рецензент Крошний І.М.
(прізвище та ініціали)

Львів – 2024

Національний лісотехнічний університет України
(повне найменування вищого навчального закладу)

ННІ комп`ютерних наук та інформаційних технологій
Кафедра комп`ютерних наук
Рівень вищої освіти другий (магістерський)
Спеціальність 122 "Комп`ютерні науки"
(шифр і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_________ Борецька І. Б.
"05" січня 2024 року

З А В Д А Н Н Я
НА ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Вольському Богдану Юрійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи *Розроблення рекомендаційної системи закупівель засобами Angular*

керівник роботи *Сторожук Олександр Леонідович, канд. техн. наук, доцент*,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом вищого навчального закладу від "13" лютого 2023 року № С-49
2. Термін подання студентом роботи "05" січня 2024 року
3. Вихідні дані до роботи
1. вивчення та дослідження предметної області
2.огляд літературних джерел для кращого теоретичного розуміння
3.проектування та розробка архітектурних рішень системи
4.побудова діаграм, що описують функції системи
5.розробка програмного рішення, відповідно до діаграм та функції системи
4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)
Розділ 1. Стан проблемної області
Розділ 2. Інформаційне забезпечення
Розділ 3. Математичне забезпечення
Розділ 4. Програмне забезпечення
Розділ 5. Розроблення стартап проекту
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

6. Дата видачі завдання "15" лютого 2023 року

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломної роботи	Строк виконання етапів роботи	Примітка
1	Огляд літературних даних та інших джерел згідно досліджуваної теми	15.02-31.03.2023	виконано
2	Аналіз досліджуваної теми та вибір відповідних варіантів її розробки	01.04-30.04. 2023	виконано
3	Постановка задачі та її формалізація	01.05-31.05. 2023	виконано
4	Вибір та обгрунтування методів і засобів проведення дослідження	01.06-30.06. 2023	виконано
5	Розроблення концептуальної схеми реалізації завдання	01.07-31.07. 2023	виконано
6	Програмна реалізація завдання	01.08-30.09. 2023	виконано
7	Тестування програмного продукту та отриманих результатів	01.10-30.10. 2023	виконано
8	Розробка пояснювальної записки магістерської роботи	01.11-30.11. 2023	виконано
9	Коригування пояснювальної записки згідно вимог, розроблення презентації	01.12-04.01. 2024	виконано

Студент (підпис) Вольський Б. Ю. (прізвище та ініціали)

Керівник роботи (підпис) Сторожук О. Л. (прізвище та ініціали)

АНОТАЦІЯ

Магістерська кваліфікаційна робота виконана студентом групи КН-61м Вольським Богданом Юрійовичем. Робота направлена на здобуття ступеня магістра за спеціальністю 122 «Комп’ютерні науки».

У результаті роботи було створено й адаптовано оригінальний генетичний алгоритм для розв'язання задачі мандрівного покупця для обраного користувачем товару. Також за допомогою GA було оптимізовано шлях від користувача до супермаркету та створено веб-сайт для візуалізації результатів.

Загальний обсяг роботи 74 сторінки, 28 зображень, 14 посилань.

Ключові слова: генетичний алгоритм, оптимізація, проблема комівояжера, проблема мандрівного покупця, моделювання відпалу.

ABSTRACT

The master's qualification thesis was completed by the student of the KN-61m group, Bohdan Yuriyovych Volsky. The work is aimed at obtaining a master's degree in the specialty 122 "Computer Science".

As a result of the work, an original genetic algorithm was created and adapted to solve the problem of the traveling buyer for the product selected by the user. GA also optimized the path from the user to the supermarket and created a website to visualize the results.

The total volume of work is 74 pages, 28 images, 14 links.

Keywords: genetic algorithm, optimization, traveling salesman problem, traveling buyer problem, annealing simulation.

ТЕХНІЧНЕ ЗАВДАННЯ

Необхідно створити й адаптувати генетичний алгоритм для розв'язання задачі мандрівного покупця для обраного користувачем товару. Також за допомогою генетичного алгоритму потрібно оптимізувати шлях від користувача до супермаркету та створити веб-сайт для візуалізації результатів. Для досягнення поставленої мети потрібно реалізувати наступні завдання:

- Проаналізувати алгоритми розв'язання задач.
- Створити унікальний алгоритм на основі еволюційного алгоритму для розв'язання задачі мандрівника.
- Спроектувати трирівневу систему.
- Створити користувацький інтерфейсу для візуалізації результатів.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАЧЕНЬ 6
ВСТУП 7
РОЗДІЛ 1. СТАН ПРОБЛЕМНОЇ ОБЛАСТІ 9
1.1. Опис предметної області 9
1.2. Огляд наявних аналогів 12
1.3. Постановка задачі 20
Висновки до розділу: 22
РОЗДІЛ 2. ІНФОРМАЦІЙНЕ ЗАБЕЗПЕЧЕННЯ 23
2.1. Аналіз предметної області 23
2.2. Проектування системи 26
Висновки до розділу: 31
РОЗДІЛ 3. МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ 32
3.1. Математичне та алгоритмічне забезпечення 32
Висновки до розділу: 41
РОЗДІЛ 4. ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ 42
4.1. Засоби розробки 42
4.2. Вимоги до технічного та програмного забезпечення 45
4.3. Опис програмної реалізації 45
4.4. Керівництво користувача 62
Висновки до розділу: 63
РОЗДІЛ 5. РОЗРОБЛЕННЯ СТАРТАП-ПРОЕКТУ 64
5.1. Опис ідеї проекту 64
5.2. Відмінність від конкурентів: 64
5.3. Аналіз ринкових можливостей: 65
Висновки до розділу: 66
ВИСНОВКИ 67
СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ 68
ДОДАТКИ 70
ДОДАТОК А. ЛІСТИНГ КОДУ ПРОГРАМИ 70

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАЧЕНЬ

FE – Front-end

TSP – Travelling salesman problem

TPP – Travelling purchaser problem

ABC – Artificial Bee Colony

AC – Ant Colony

SA – Simulated Annealing

SASim – Simulated Annealing with Similarity measure

DSMO - Discrete Spider Monkey Optimization

BSO – Brain Strom Optimization

GA – Genetic Algorithm

AG-BSO – Agglomerative Greedy Brain Strom Optimization

SCX – Sequential Constructive crossover

TCO – Triple Crossover Operator

DFACO – Dynamic Flying Ant Colony

WOA – Whale Optimization Algorithm

VNS – Variable Neighborhood Search

VDWOA – Discrete Whale Optimization Algorithm with VNS

API – Application programming interface

CSS – Cascading Style Sheets

JS – JavaScript

HTML – HyperText Markup Language

DOM – Document Object Model

OOP – Object Oriented Programming

MVC – Model-View-Controller

MVVM - Model-View-ViewModel

ВСТУП

Світ модернізується з кожним днем. Нові ідеї перетворюються на стартапи, а нові продукти з IT-індустрії все більше приваблюють користувачів. У смартфонах є все, що нам потрібно. Банківські картки, якими ми щодня розраховуємося за необхідні нам речі, документи, що засвідчують нашу особу, тощо.

Все поступово стає віртуальним і бажання людей зробити все віртуальним зростає з кожним днем. Люди мають різні потреби, не тільки віртуальні. Щодня людям потрібно їсти, займатися спортом, мати здоровий сон. І в першу чергу, щоб задовольнити цю потребу, люди найчастіше йдуть до супермаркету.

З розвитком технологій покупки тепер здійснюються онлайн. Наразі онлайн-шопінг не є дуже популярною темою, але з появою пандемії COVID-19 все більше супермаркетів розробляють веб-додатки. Це полегшує нам пошук наших улюблених продуктів онлайн у різних супермаркетах. Зазвичай супермаркети продають свою продукцію за певною ціною і намагаються усереднювати її з іншими франчайзинговими мережами. Однак, оскільки кожній компанії потрібно, щоб клієнти обирали супермаркет для покупок, маркетологи запроваджують різноманітні акції та намагаються знизити ціни, щоб залучити якомога більше клієнтів. Тому ми стикаємося з проблемою правильного розподілу покупок, тобто де купувати ті чи інші товари і чи вигідно це для нашого гаманця.

Вирішенням цієї проблеми є веб-додатки, які полегшують розподіл продуктів і показують нам, де саме і в яких супермаркетах ми повинні їх купувати.

Для цього можна використовувати оптимізацію. Оптимізація використовується сьогодні в усіх сферах людської діяльності, допомагаючи

підвищити ефективність і заощадити дорогоцінний час, усуваючи необґрунтовані альтернативи.

Оптимізація особливо корисна для вирішення складних завдань. Наприклад, робота туристичного агента полягає в тому, щоб проїхати через кожне місто лише один раз і повернутися у вихідну точку. Коли ви робите покупки в Інтернеті, кошик накопичується, і коли ви нарешті переходите до етапу вибору потрібного супермаркету, у вас є багато варіантів, що робити. Ця проблема дуже схожа на проблему продавця, який повинен вибрати найкращий шлях вперед. Навіть якщо він обирає найкращий маршрут, він стикається з іншою проблемою. Якщо у нього є можливість користуватися автомобілем, він може робити покупки в декількох супермаркетах і тому повинен правильно розподілити продукти, які йому потрібно купити. Це так звана проблема мандрівного покупця.

Тому пропонується використовувати еволюційні алгоритми для оптимізації задачі вибору правильного супермаркету та задачі розподілу правильних покупок.

Предметом дослідження є процес оптимізації покупок та розробка клієнтської частини додатку.

Об'єктом дослідження є еволюційний алгоритм для вирішення задач TSP та TPP.

Метою даної магістерської роботи є створення рекомендаційної системи, яка допомагає вибрати найкращий спосіб дістатися до супермаркету та оптимізувати покупки в ньому.

РОЗДІЛ 1. СТАН ПРОБЛЕМНОЇ ОБЛАСТІ

1.1. Опис предметної області

1.1.1. Опис процесу діяльності

З давніх часів наші предки намагалися розв'язувати задачі таким чином, щоб процес пошуку рішення займав відносно короткий час, а розв'язок задачі був оптимальним. Пошук оптимального рішення, тобто рішення, яке має більше переваг, ніж недоліків за певний проміжок часу, називається оптимізацією.

На щастя, сьогодні немає необхідності розробляти різні методи для оптимізації конкретної задачі, а лише вибрати відповідний підхід (метод), виходячи з умов задачі та бажаного рішення. Загалом, методи оптимізації поділяються на такі категорії, як лінійне програмування, стохастичне програмування, евристичні та метаевристичні методи.

За останні кілька десятиліть остання категорія методів оптимізації, згадана вище, стала дуже популярною. Це пов'язано з тим, що вони призначені для дослідження набору потенційних рішень проблеми та пошуку найоптимальнішого з них. Ці методи зазвичай базуються на процесах, які існують у природі, наприклад, алгоритм оптимізації мурашника, який базується на поведінці мурах.

Іншим прикладом є генетичний алгоритм, розроблений Джоном Генрі Голландом [1] та іншими. Він заснований на теорії еволюції Чарльза Дарвіна [2] і використовує систему природного відбору особин з популяції для пошуку найкращого рішення (особин). Генетичні алгоритми є ітеративними, де кожна наступна ітерація є набором рішень (популяцій) проблеми. Ітерація складається з декількох етапів: відбір, кросинговер і мутація.

Цей алгоритм оптимізації допомагає знаходити достатні результати для вирішення різних проблем, таких як задача комівояжера (ЗК) [3]. У цій задачі

людині (комівояжеру) потрібно пройти через кожне місто (кількість яких залежить від умови задачі) лише один раз і повернутися до міста, з якого він розпочав свою подорож. Задача ускладнюється, якщо міста замінити на магазини або супермаркети, а також додати умову купівлі в різних супермаркетах і повернення в початкову точку. Дійсно, такі проблеми мають свою термінологію. А саме, the travelling buyer problem (TPP) [4], або проблема мандрівного покупця: основною умовою TPP є наявність одного товару в кожному магазині, але оскільки в супермаркетах зазвичай є кілька товарів, то наш випадок не підходить під загальні критерії оптимізації покупок.

Отже, визначивши цільову область та її складові, можна визначити конкретні кроки процесу діяльності, які допоможуть сформулювати конкретні вимоги, що мають бути виконані для досягнення бажаних результатів.

Загалом, процес діяльності можна розділити на такі етапи:

Аналіз конкурентів, присутніх на ринку.

Аналіз різних методів оптимізації для вирішення проблем ТСП і ТЕС.

Вибір методу оптимізації для виконання завдання.

Провести експерименти та оцінити рішення задачі.

Створити користувацький веб-інтерфейс для відображення результатів.

По-перше, необхідно визначити подібні приклади, які вже існують і були реалізовані, щоб краще зрозуміти результати, яких ви хочете досягти. Потім аналізується найбільш підходящий метод оптимізації для розв'язання задачі комівояжера та її похідних. Після того, як метод обрано, його необхідно застосувати на практиці та проаналізувати результати. Якщо результати незадовільні, спробуйте застосувати метод, який допоможе уникнути небажаних пасток, таких як локальна оптимізація. Нарешті, щоб користувачі могли використовувати запропоновану оптимізацію, необхідно створити користувацький інтерфейс, тобто веб-додаток, який дозволяє користувачам вибирати і додатково оптимізувати бажані товари в кошику.

Щоб краще проілюструвати процес роботи, було створено UML-діаграму діяльності. (рис. 1.1).

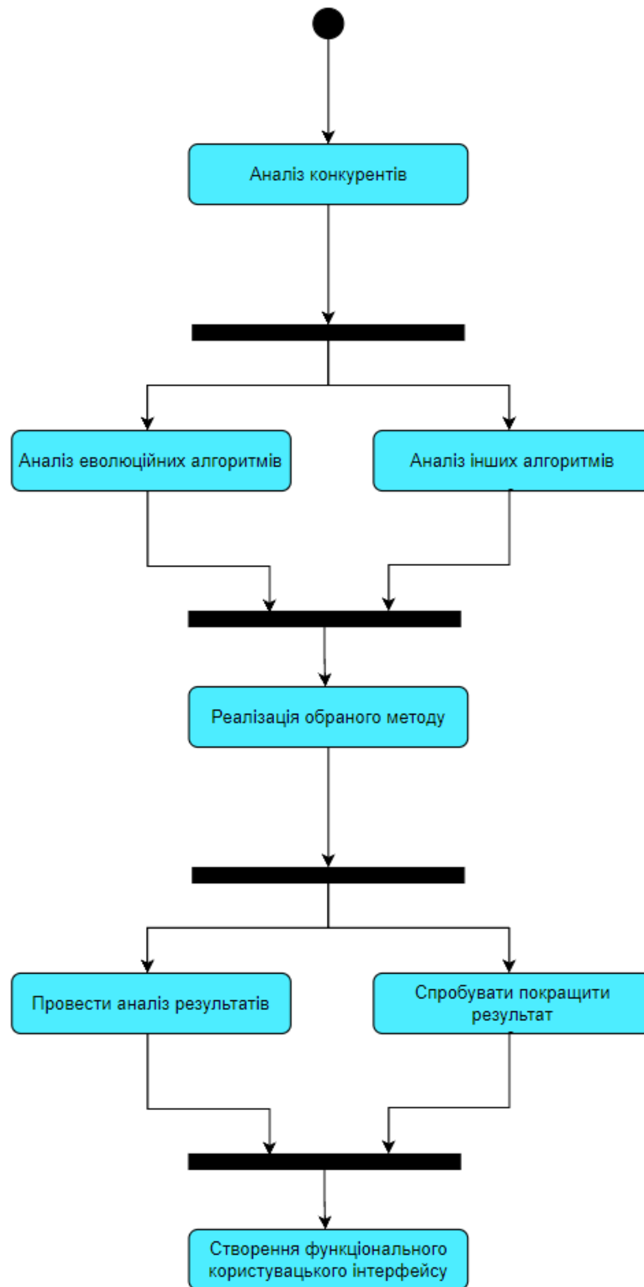


Рис. 1.1 Діаграма діяльності

1.1.2. Опис структури системи

Загалом, система складається з алгоритму оптимізації та інтерфейсу, що представляє результати:

1. Системи можна розділити на наступні компоненти

2. Алгоритм пошуку найкоротшого шляху для всієї вибірки супермаркетів
3. Алгоритм для пошуку оптимального рішення для покупок у супермаркеті та його маршруту.
4. Інтерфейс користувача для об'єднання та відображення результатів попередніх алгоритмів оптимізації користувачеві

1.2. Огляд наявних аналогів

Найкращим рішенням для детальної оцінки наукових джерел є використання схеми PRISMA. Схема показує процес застосування критеріїв включення та виключення для формування остаточної кількості статей.

Пошуковий запит був розроблений і реалізований з використанням критеріїв відбору, розроблених на основі тематики статей та їх похідних, в результаті чого було знайдено 63 наукові роботи. Дві статті були також знайдені в інших наукометричних базах даних та на сторонніх веб-сайтах. Кожне джерело було проаналізовано, і публікації, що не мають відношення до цілей ПНП або ППП, були виключені. Джерела з сумнівними результатами або недоведеною ефективністю також не розглядалися. За попередньою процедурою було отримано 25 публікацій, з яких було відібрано 10. На основі проведеної роботи підготовлено діаграму PRISMA. (рис. 1.2).

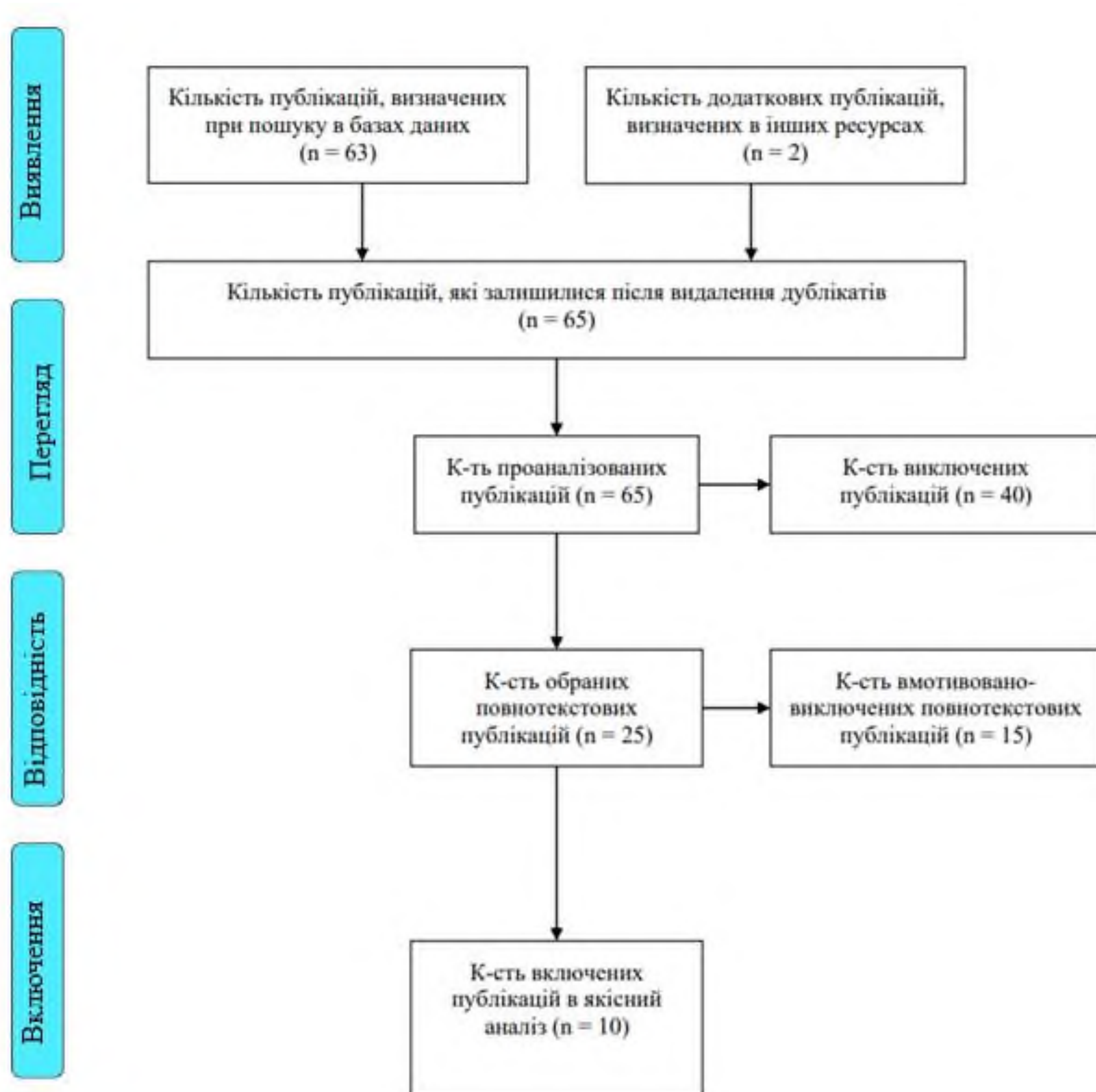


Рис. 1.2 Схема оцінки публікацій PRISMA

Тепер зробимо детальний огляд вибраних джерел.

У статті [5] розглядається гібридний генетичний алгоритм для розв'язання задачі комівояжера. Автор поєднує традиційний розв'язок, отриманий за допомогою звичайного генетичного алгоритму, з так званим локальним розв'язком, який має на меті оптимізувати результати, запропоновані генетичним алгоритмом. Роль локального рішення полягає в тому, щоб об'єднати найкраще індивідуальне рішення задачі комівояжера, доступне на даний момент, з наступним рішенням (тобто рішенням генетичного алгоритму). Кожна хромосома вчиться у попередньої хромосоми за допомогою

алгоритму локального пошуку і вступає в еволюційний процес. Це гарантує, що еволюційний алгоритм не створює абсолютно випадкову популяцію. Експериментальні результати показують, що запропонований автором алгоритм демонструє кращі часові характеристики, ніж звичайні генетичні алгоритми для задач з різними межами. Однак, як зазначає автор, еволюційні алгоритми досить обмежені і тому не завжди знаходять оптимальний розв'язок.

У статті [6] автор пропонує вдосконалити генетичні алгоритми, використовуючи штучні бджолині колонії (ШБК). Штучні бджолині колонії - це евристичні методи, які використовують поведінку медоносних бджіл для покращення популяційних алгоритмів, таких як генетичні алгоритми. Колонія складається з трьох груп бджіл: спостерігачів, розвідників і робітників. Загалом, метод передбачає, що кожен вихід популяційного алгоритму є приманкою, яку кожна група бджіл переробляє певним чином. Автор застосував цей метод для вдосконалення генетичного алгоритму для вирішення проблеми комівояжера. Для цього ABC генерує початкову популяцію. Спочатку генеруються нові маршрути, потім робітники обробляють кожен маршрут. Якщо маршрут кращий за попередній, працівник переключається на нього, а після обробки маршруту знаходить новий маршрут і повторює його, поки не буде знайдено найкращий, а якщо досягається певна межа, то весь процес повторюється. Таким чином формується початкова популяція, яка набагато краща за випадково згенеровану. Автори підкреслюють, що це рішення було порівняно зі звичайними генетичними алгоритмами і що ABC показав кращі результати.

У статті [7] описано генетичний алгоритм гібридних клітин для розв'язання задачі комівояжера. Автор поєднує алгоритм імітації відпалу з клітинним генетичним алгоритмом. Гібридний алгоритм складається з трьох частин: генетичні маніпуляції, елітна стратегія та еволюція клітинного простору. Кожна частина допомагає покращити недоліки звичайних генетичних алгоритмів, а саме: генерація популяції та краще формування функції

пристосованості. Оскільки в цій роботі розглядалися лише симетричні задачі комівояжера, результати для симетричних задач показують кращі результати, ніж традиційний популяційний алгоритм, але автори зазначають, що застосування цього гібридного алгоритму до асиметричних задач та інших задач комівояжера не проводилося. Тому необхідні подальші дослідження ефективності цього алгоритму.

У роботі [8] автори пропонують розв'язати задачу комівояжера, оптимізувавши її за допомогою модифікованого алгоритму мурашиних колоній (АК). Оскільки традиційний алгоритм АК має недоліки, такі як вибір одного і того ж шляху з найбільшою матрицею шляхів та погіршення оптимізації зі збільшенням кількості міст, автор пропонує додавати кращих мурах у нових поколіннях (досить схоже на те, що обговорювалося в [1] щодо генетичних алгоритмів та генетичних алгоритмів та подібного методу). Автор також модифікує традиційний алгоритм АС шляхом введення ваг для кожного шляху міграції та пропонує використовувати імітаційний відпал (SA) та імітаційний відпал з мірами подібності (SASim). В результаті модифікацій алгоритм показав хороші результати, підвищив ефективність і знайшов найкоротший шлях за менший час. Рішення про включення найкращого результату до наступної ітерації виявляється дуже ефективним, що є важливим порівняно зі звичайними АК, які використовують псевдовипадкові генерації.

У роботі [9] для розв'язання задачі комівояжера пропонується використовувати дискретну оптимізацію павукової мавпи (DSMO). Основна відмінність між цими алгоритмами полягає в тому, що основний процес відбувається у два етапи. Кожна особина (павукоподібна мавпа, один з варіантів розв'язання задачі комівояжера) щоразу оновлює свої дані. Тобто, вона об'єднується з іншими особами в локальну групу, порівнюється з локальним лідером, а потім з глобальним лідером. Крім того, параметри (або мутації), які використовуються як вхідні дані для алгоритму, також спочатку порівнюються локально для конкретної вибірки, а потім глобально. Автор

застосував цю оптимізацію до великих і малих вибірок і показав, що задачу комівояжера можна розв'язати за меншу кількість ітерацій, ніж традиційний алгоритм CI, як для малих, так і для великих вибірок.

У статті [10] запропоновано покращений алгоритм BSO (Brainstorm Optimization) для розв'язання задачі комівояжера. Оскільки кластеризація в цьому алгоритмі відіграє важливу роль у підвищенні швидкості збіжності алгоритму, автори спочатку пропонують замінити звичайну кластеризацію за методом k-середніх, що використовується в традиційному BSO, на більш ефективну ієрархічну кластеризацію. Наступним кроком є введення жадібного алгоритму, який намагається згенерувати найкращу популяцію, розглядаючи лише найближчі міста. Нарешті, автор створює оператор кросинговеру, який використовується для відбору лише найкращих генів серед особин. Після застосування алгоритму до тестових даних і порівняння результатів з алгоритмами оптимізації GA і традиційним BSO, автор підкреслює, що запропонований алгоритм AG-BSO перевершує традиційні алгоритми з точки зору ефективності та швидкості збіжності. Ефективність цього алгоритму для задачі комівояжера з великою кількістю міст є неоднозначною.

В роботі [11] представлено розв'язок задачі комівояжера з використанням покращеного ГА (генетичного алгоритму). Удосконалення, запропоноване автором, полягає в заміні звичайної функції кросовера на модифіковану. Після вибору двох особин для схрещування алгоритм ділить кожну особу на три частини. З першої особини вибирається початок і середина, які порівнюються з серединою другої особини і вибирається найкраща. З другої особини вибирається кінець. Ці операції дають два результати, і дублікати відсіюються. Далі алгоритм працює як зазвичай, тобто переходить до мутації. Автор цієї статті порівнює запропонований ним оператор (функцію) кросинговеру з іншими поширеними операторами, такими як: послідовний конструктивний кросинговер (SCX) та оператор потрійного кросинговеру (TCO) за середнім

часом. Результати показують, що авторський оператор кросинговеру на 15% кращий за часом виконання алгоритму ГА.

В роботі [12] автор розпаралелював генетичний алгоритм. Автор пропонує використовувати турніри на етапі селекції ГА. Концепція турніру полягає в тому, що проводиться декілька етапів відбору найкращих особин, з меншою кількістю особин на кожному етапі, і, нарешті, найкращі особини відбираються і потрапляють на етап кросовера. У звичайних ГА всі основні етапи - обчислення фітнес-функції, відбір, кросинговер і мутації - виконуються послідовно. Автори цієї статті розпаралелили кожен з цих етапів, щоб прискорити роботу алгоритму. Однак, оскільки ГА повинен виконувати операції послідовно, автор використовує три різних ядра для розподілу алгоритму. Потік у першому ядрі виконує фітнес-функцію, потік у другому ядрі виконує майже всі операції, а потік у третьому ядрі генерує нову популяцію. Загалом, розпаралелювання показує хорошу ефективність, але час виконання алгоритму збільшується, коли ядра переключаються між собою. Тому автори вказують на те, що використання розпаралелювання ефективне лише для великої кількості міст у задачі комівояжера.

Автори в [13] пропонують покращений алгоритм мурашиної оптимізації. Оскільки більшість алгоритмів, заснованих на мурашиному алгоритмі, знаходять рішення через тривалий час, автори модифікували етапи мурашиного алгоритму. А саме, кількість сусідів, що залежить від найкращого рішення, знайденого алгоритмом, була динамічно змінена, а популяція мурах була розділена на звичайних і літаючих мурах. Кількість феромону також змінювалася, так що в першій ітерації феромон отримували далекі сусіди, а в наступних ітераціях - ближні сусіди Для вимірювання ефективності DFACO його порівнювали зі звичайним алгоритмом ACO, і результати показали, що DFACO був середньою і великою вибіркою в яких DFACO виявився більш ефективним, ніж звичайні алгоритми.

Робота [14] є застосуванням алгоритму китової оптимізації. Цей метод належить до описаних раніше алгоритмів CI і базується на поведінці горбатих китів. Особливістю цих алгоритмів є те, що вони припиняють роботу, коли знайдено локальний найкращий результат, тобто найкращий результат між невеликою кількістю сусідніх результатів. Тому пропонується використання VDWOA (Discrete whale optimization algorithm with variable neighbourhood search). Основні відмінності від звичайного алгоритму WOA полягають у тому, що автори ввели гаусівський розподіл для покращення розкиду китів та VNS для розкиду сусідів. Після модифікації алгоритму були проведені експерименти з порівняння WOA, DWOA та VDWOA, які показали, що VDWOA показав найкращу продуктивність, а DWOA - кращу за VDWOA.

Для аналізу аналога було обрано додатки, що оптимізують маршрути з використанням геолокації та веб-додатки. Ще одним критерієм відбору була можливість порівнювати продукти з різних супермаркетів.

Grocery King - ця програма допомагає користувачам порівнювати ціни в різних супермаркетах. Це один з небагатьох додатків, який допомагає оптимізувати маршрути. Користувачі можуть вибрати кілька супермаркетів і за допомогою вбудованої карти Google подивитися, як дістатися до кожного з них. Він також може перевіряти інформацію про супермаркети, наприклад, час їхньої роботи та наявність знижок.

Основні переваги цього додатку:

- Дозволяє користувачам створювати та зберігати списки продуктів, які вони хочуть купити.
- Можна завантажувати купони та коди знижок.
- У багатьох супермаркетах є можливість сканувати штрих-коди та об'єднувати товари.
- Недоліки додатку:
- Основним недоліком цього додатку є те, що він доступний лише для користувачів IOS.

- Користувачам з невеликим досвідом використання подібних додатків буде складно розібратися в інтерфейсі.

Flipp - це додаток, який дозволяє користувачам обмінюватися знижками на різні товари в супермаркеті з друзями та родиною. Серед корисних функцій - можливість об'єднувати кілька смартфонів і створювати спільні списки покупок. У додатку є функція пошуку, яка допомагає швидко знаходити продукти.

Переваги:

- Зручний інтерфейс як для перегляду товарів, так і для створення списків покупок.
- Після купівлі товару зі списку, список зберігається, а не видаляється. Певні товари зі збереженого списку рекомендуються при створенні нового списку.
- Додаток шукає супермаркети на певній відстані від користувача.
- Недоліки додатку.
- Додаток порівнює товари лише зі знижками.
- Немає можливості перевірити наявність купонів у супермаркеті.

SuperMarkets - корисний український веб-додаток для порівняння цін у популярних супермаркетах України. Додаток дозволяє користувачам перевіряти наявність продуктів поблизу них. Також є функція пошуку за геолокацією - просто виберіть продукти, які ви хочете купити, та увімкніть зчитування геолокації. Сайт шукає товари в найближчих супермаркетах.

Переваги сайту:

- Великий вибір та функція порівняння
- Сканер QR-коду дозволяє швидше знайти потрібний товар.
- На сайті є категорії товарів, що дозволяє сортувати та фільтрувати товари.
- При додаванні товару розробник відображає рейтинг товару, наскільки він корисний і звідки він взятий.

Недоліки сайту, через технічні можливості API, наданого супермаркетом, іноді не працює статус товару на складі.

Проаналізувавши аналогічні товари, було зроблено висновок, що можливість відстежувати географічне розташування користувача необхідна для кращого аналізу сусідніх супермаркетів.

1.3. Постановка задачі

Оскільки оптимізація покупок є досить складним процесом і часто не дозволяє знайти тривіальних рішень, необхідно побудувати інфраструктуру, яка впорається зі складністю завдання.

По-перше, необхідно вибрати метод оптимізації, який буде застосований до завдання. Проаналізувавши роботи, було зроблено наступні висновки: найкращими або більш точними методами оптимізації для задач ТСП та ТЕС є евристичні та метаевристичні методи, які, відповідно, використовуються в цій бакалаврській дисертації.

Генетичні алгоритми (ГА) та оптимізація мурашиних колоній (ОМК) показали найкращі результати, і більшість авторів робіт, що базуються на цих алгоритмах або так чи інакше посилаються на них, рекомендують використовувати їх базові версії, якщо кількість міст на маршруті досить невелика. І це найбільш підходить для нашої задачі. Це пов'язано з тим, що в нашій оптимізації покупок користувачів кількість супермаркетів майже завжди не перевищує 10-15. Ця оптимальна кількість була обрана тому, що маршрути до супермаркетів, які знаходяться далеко від користувача, не є оптимальними.

При виборі між алгоритмами GA та ACO, мурашиний алгоритм показує досить хороші результати для задачі простого продавця, але в результатах алгоритму для задачі мандрівного покупця (TTP), GA показує дещо кращі результати, а крок ACO не зовсім підходить і потребує доопрацювання Слід

зазначити, що крок ACO не зовсім підходить і потребує модифікації. Тому для оптимізації покупок у супермаркеті було обрано алгоритм GA.

Іншими словами, оскільки основною задачею бакалаврської роботи є оптимізація покупок, то її можна розділити на конкретні підзадачі:

По-перше, необхідно створити та оптимізувати обрані користувачем маршрути до супермаркетів. Іншими словами, алгоритм GA застосовується до задачі TSP. Після отримання результатів робляться спроби їх покращити. У більшості проаналізованих робіт було застосовано кілька алгоритмів для покращення результатів ГА. Один з найкращих результатів показав алгоритм SA, який використовується для покращення найкращого шляху, обраного ГА.

Після отримання найкращого шляху на попередньому кроці, ГА потрібно застосувати до задачі про мандрівного покупця; на жаль, ми не змогли знайти жодної статті, яка б конкретно описувала реалізацію алгоритму, оскільки задача про ТПП не є дуже популярною. Тому ми вирішили розробити власний алгоритм розв'язання цієї задачі на основі ГА.

Після розробки та реалізації наведеного вище алгоритму необхідне середовище користувача для візуалізації результатів, отриманих на попередніх кроках. Для цього пропонується створити веб-сайт з певними функціями: кошиком, куди користувач може додавати покупки, та картою, на якій можна вибрати потрібний супермаркет. Цей функціонал відображатиме алгоритмічні результати для продуктів та супермаркетів, вказаних користувачем.

Іншими словами, в загальному вигляді завдання бакалаврської роботи можна сформулювати наступним чином:

Оптимізація покупок користувача шляхом застосування генетичного алгоритму до задачі продавця для обраного користувачем супермаркету; створення генетичного алгоритму для задачі мандрівного покупця для покупок користувача; та візуалізація реалізації на веб-сайті.

Висновки до розділу:

У цьому розділі було проаналізовано предметну область, тобто встановлено процес діяльності та описано структуру системи. Також було проаналізовано переваги та недоліки аналогічних продуктів та зроблено висновки щодо їх функціональності.

Проаналізовано десять робіт, відібраних за схемою PRISMA. Методи оптимізації були обрані для застосування до задач TSP і TPP.

Найголовніше, було сформульовано та охарактеризовано проблему цих робіт. Зроблено висновок, що евристичні та метаевристичні методи оптимізації підходять для вирішення проблеми, і генетичні алгоритми були обрані в якості основи.

РОЗДІЛ 2. ІНФОРМАЦІЙНЕ ЗАБЕЗПЕЧЕННЯ

2.1. Аналіз предметної області

2.1.1. Об'єкти дослідження

У цьому дослідженні генетичні алгоритми повинні бути застосовані до проблем TSP і TPP.

Генетичні алгоритми є метаевристичними алгоритмами і зазвичай відносяться до одного з гібридних класів, тобто еволюційних алгоритмів. Цей метод оптимізації застосовує еволюційний відбір для пошуку оптимального рішення поставленої задачі. Основний процес роботи алгоритму складається з декількох основних фаз. Це означає, що кожна наступна фаза залежить від попередньої, і якщо хоча б одна з них буде виконана неправильно, алгоритм буде працювати неправильно і результати будуть катастрофічними.

Перша фаза - селекція (відбір). Його основна мета - вибрати особу або особини, які найкраще підходять для передачі генів (специфічних характеристик особини, які залежать від поставленого завдання) наступному поколінню. Другий етап - кросинговер, який виділяють як ключовий етап алгоритму, оскільки він вносить в особину найбільш кардинальні зміни, здатні поліпшити наступне покоління. Гени з попереднього етапу змішуються, причому половина генів походить від обраної особини, а інша половина - від іншої особини. Останній етап - мутація, яка викликає певні зміни в особинах, створених на попередньому етапі. Етапи мутації варіюються від випуску до випуску, але концепція є загальною. Відбирається невелика кількість особин і порівнюється з іншим випадково згенерованим коефіцієнтом, і якщо умови виконуються, відбувається етап мутації. Мутація - це часто конкретна зміна в гені.

Оскільки більшість задач, що розв'язуються генетичними алгоритмами, використовують жадібний підхід, зазвичай ГА шукає локальний оптимум. Для

цього інші алгоритми оптимізації використовуються як своєрідний каталізатор для покращення результатів роботи ГА. Оскільки будь-який алгоритм може виступати в ролі каталізатора, ми вирішили використати алгоритм Simulated Annealing (SA) для покращення результатів ГА.

SA - це досить простий алгоритм, заснований на процесі кристалізації різних матеріалів. Алгоритм базується на температурних коливаннях, де встановлюється певна температура, при якій алгоритм запускається, і на кожній ітерації виконується операція, яка залежить від поставленого завдання. Потім перевіряється, чи є результат попередньої операції кращим за попередній, і якщо так, то цей результат вважається найкращим. Однак, якщо робота алгоритму закінчується на цьому етапі, хороші результати оптимізації досягаються рідко. Тому алгоритм має певні особливості. Якщо випадково згенероване число менше, ніж ймовірність, визначена за стандартною формулою (аналог етапу мутації в генетичних алгоритмах), вибирається рішення, згенероване на цьому етапі. Звичайно, це рішення не обов'язково є найкращим, але така операція дозволяє алгоритму перестрибнути через локальний оптимум і знайти кращий результат, ніж перший.

2.1.2. Вхідні дані

Продукти харчування є одним з ключових факторів у цьому питанні. Кожен супермаркет встановлює власні ціни на певні продукти, але, звичайно, ціни коливаються в межах певного середнього цінового діапазону. Для початку користувач обирає продукт, який він бажає придбати, і після завершення процесу вибору користувач потрапляє до інтерфейсу вибору супермаркету, де він може здійснити покупку. На цьому етапі надходять основні вхідні дані для розв'язання задачі TSP, а саме відстань. Точки на земній поверхні мають широту та довготу. Перше "місто" (в концепції TSP) задачі - це довгота і широта користувача, а інші "міста" - це супермаркети та їхні координати.

Отриманий маршрут разом з температурним коефіцієнтом та кількістю ітерацій передається алгоритму SA для перевірки на локальний оптимум.

Після отримання рішення у вигляді оптимізованого маршруту від користувача до супермаркету, цей маршрут і товари, придбані користувачем, використовуються як вхідні дані для другого алгоритму - генетичного алгоритму для розв'язання задачі TEC. Ціни в супермаркеті також слугують вхідними даними для переміщення покупців.

Вхідні дані також включають параметри двох алгоритмів:

- Розмір популяції
- Кількість поколінь
- Кількість турнірів
- Швидкість мутацій

Обмежень на вхідні дані немає, але не можна вибирати або додавати продукти, яких не існує в супермаркеті. Що стосується вибору супермаркету, то навіть якщо користувач обирає супермаркет на іншому кінці країни, генетичний алгоритм ігнорує його при розв'язанні задачі мандрівного покупця. Вхідні параметри алгоритму обираються на основі даних, наданих користувачем.

2.1.3. Вихідні дані

Після отримання вхідних даних кожен етап алгоритму повинен мати вихідні дані. Після обробки завдання продавця в супермаркеті та початкової позиції користувача, вказаної користувачем, повинен бути отриманий результат роботи алгоритму, тобто шлях від супермаркету до користувача. Цей шлях надсилається алгоритму симуляції відпалу і після обробки отримується найкращий шлях або, якщо алгоритм не може знайти локальний оптимум, повертається отриманий шлях.

Потім шлях обробляється генетичним алгоритмом для задачі про мандрівного клієнта. Результат роботи алгоритму наступний:

Конкретний шлях від користувача до конкретного супермаркету

Вартість шляху в грошових одиницях

вартість придбаних товарів та конкретних товарів, придбаних у конкретному супермаркеті.

Обмеження вихідних даних в основному пов'язані з обмеженнями системи користувача та мережевого з'єднання. Без певної форми підключення до Інтернету користувачі не можуть користуватися веб-сайтом. Крім того, якщо з'єднання погане, користувачеві доводиться чекати, поки завантажаться певні компоненти інтерфейсу.

Крім того, якщо користувач не оновлював компоненти на своєму комп'ютері протягом тривалого часу, це може спричинити проблеми з навантаженням на комп'ютер користувача під час процесу оптимізації генетичного алгоритму.

2.2. Проектування системи

Як було коротко описано в попередньому розділі, система складається з трьох рівнів. Генетичний алгоритм відіграє головну роль у системі.

Перший рівень - це початковий рівень, який описується як рівень користувача. Цей рівень містить всі функції, доступні в системі. Як правило, цей рівень розділений на дві сторінки, де зібрані основні функції. Після завантаження системи користувач потрапляє на головну сторінку (сторінку вибору товару). На цій сторінці користувач має можливість вибрати товар. Кожен товар, який користувач вподобав і бажає придбати, додається до кошика. Кошик можна переглянути більш детально і видалити з нього товари. Коли товар додано до кошика, відображається кнопка "оптимізувати", яка при натисканні перенаправляє користувача на іншу сторінку.

На цій сторінці користувач за допомогою певних функцій може переглянути своє місцезнаходження та вибрати супермаркети, доступні для покупки. Загалом, систему можна візуалізувати наступним чином. (рис. 2.2.1).

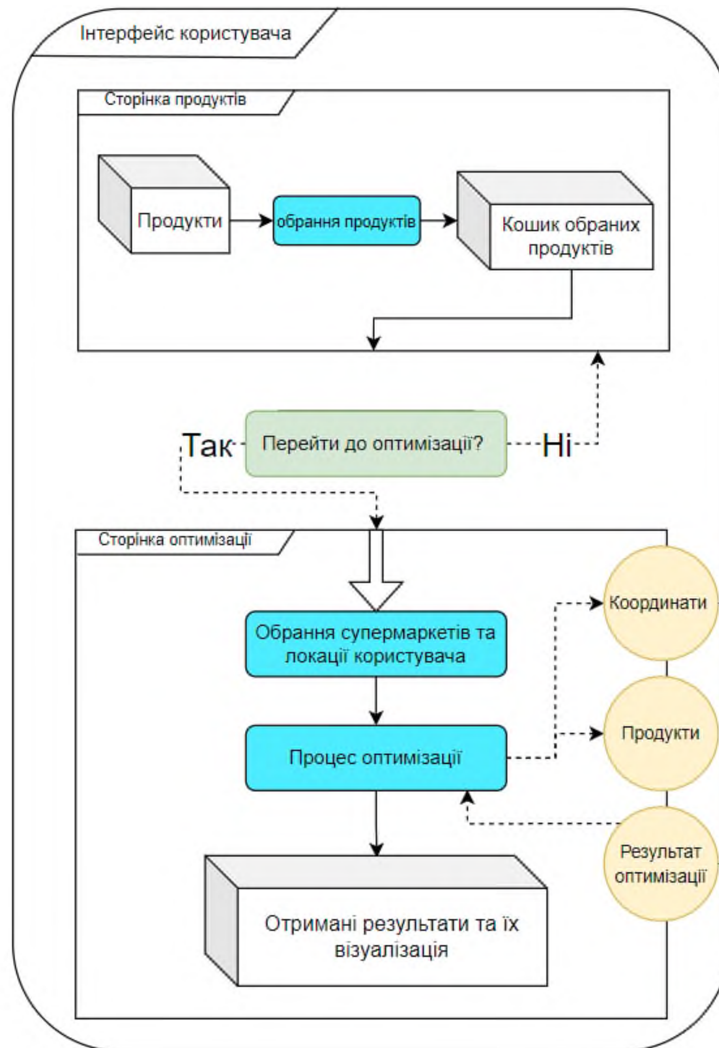


Рис. 2.2.1 Схема шару інтерфейсу користувача

Вхідні дані подаються в процес оптимізації, що включає рівні 1 і 2 системи.

Рівні 2 і 3 - це алгоритми, які розв'язують задачу продавця і задачу клієнта, що подорожує. Кожен генетичний алгоритм повинен включати три основні фази:

- Селекція;
- Кросинговер;
- Мутація;

Кожен алгоритм також має підфазу, яка відіграє важливу роль в оптимізації ГА. Підфази включають генерацію початкової популяції та фітнес-функції.

Другий рівень системи - це алгоритм, який оптимізує шлях від початкової точки (користувача) до супермаркету, що означає, що перший рівень повинен вирішувати задачу продавця. Вхідними даними є координати, задані користувачем, і на першому етапі формується початкова популяція. З цього етапу в гру вступає генетичний алгоритм, який задає початкові параметри та початкову популяцію. На кожному етапі, де відбувається відбір особин, застосовується функція пристосованості. Генетичний процес починається з етапу елітизму (відбору еліти), який полягає у відборі половини найкращих особин з поточної популяції. Наступним етапом є відбір, на якому шляхом турнірного відбору відбираються дві найкращі особини, від яких відбираються гени нащадків. Останній етап - спарювання, під час якого відбувається обмін генами з попереднього етапу. Після отримання нащадків формується нова популяція - на 50% з попередньої популяції (етап елітизму) і на 50% з нащадків, отриманих на етапі кросинговеру. Останній етап алгоритму - мутація, на етапі мутації змішуються гени нащадків, отриманих з генів, відібраних шляхом відбору та кросинговеру, особини зі стадії елітарності на цьому етапі не обробляються. Гени спеціально змішуються або замінюються, щоб уникнути відбору на один і той самий результат у кожній ітерації, додаючи до ефекту відбору, який не є очевидним. На цьому етапі також існує певна ймовірність виникнення сильних мутацій.

Після завершення всіх ітерацій, зазначених в алгоритмі, результатом є оптимізований шлях від початкової позиції до заданої точки. Для покращення результатів отриманий шлях подається в алгоритм симуляції відпалу разом з початковими параметрами.

Алгоритм SA використовує фазу мутації як важливу модифікацію шляху. Це запозичено з попередніх генетичних алгоритмів. На кожній ітерації шлях

мутується і використовується функція пристосованості (також з попередніх алгоритмів) для перевірки того, чи є цей шлях більш оптимальним, ніж попередній. Цей процес триває доти, доки кількість ітерацій не буде вичерпана або доки температура не впаде до нуля. У деяких випадках вибирається шлях, який є гіршим за попередню ітерацію, щоб знайти кращий результат. Характеристики створеної схеми шарів такі. (рис. 2.2.2).

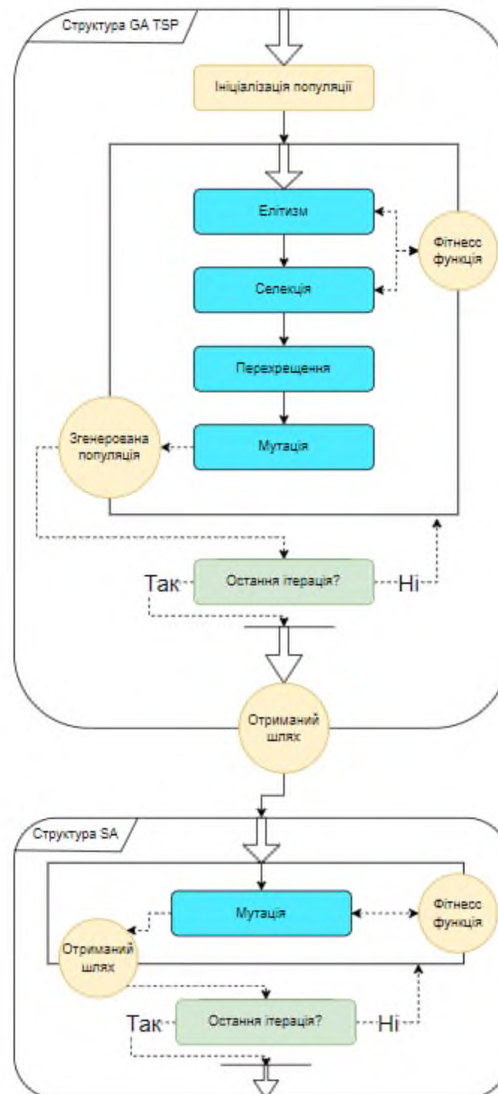


Рис. 2.2.2 Схема шару GA TSP

Покращені шляхи та шляхи, отримані після генетичного алгоритму, включаються до третього шару. На початку шару формується діапазон цін для кожного супермаркету, а також формується початкова популяція. Це подається в алгоритм як вхідні дані. Загалом, принципи роботи генетичного алгоритму

ТЕС такі ж, як і в попередніх шарах, а основні відмінності в алгоритмах пояснюються в наступному розділі. Нижче наведено схему шарів. (рис. 2.2.3).

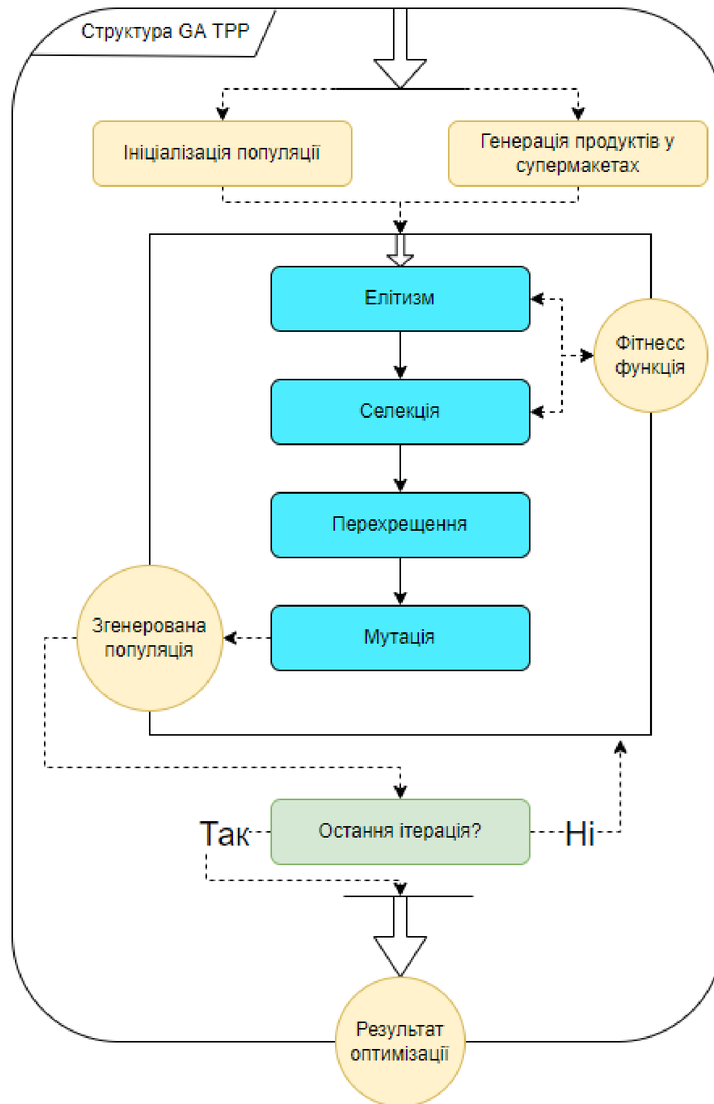


Рис. 2.2.3 Схема шару GA TPP

Після всіх ітерацій отримано вихідні дані у вигляді маршруту користувача до супермаркету, його вартості та переліку продуктів, придбаних у цьому супермаркеті.

Ці дані вводяться в перший рівень - інтерфейс користувача. Дані візуалізуються і представляються у вигляді своєрідного аналізу. Потім користувач може повернутися на початкову сторінку вибору продуктів. Загальна схема проектування системи показана нижче. (рис. 2.2.4).

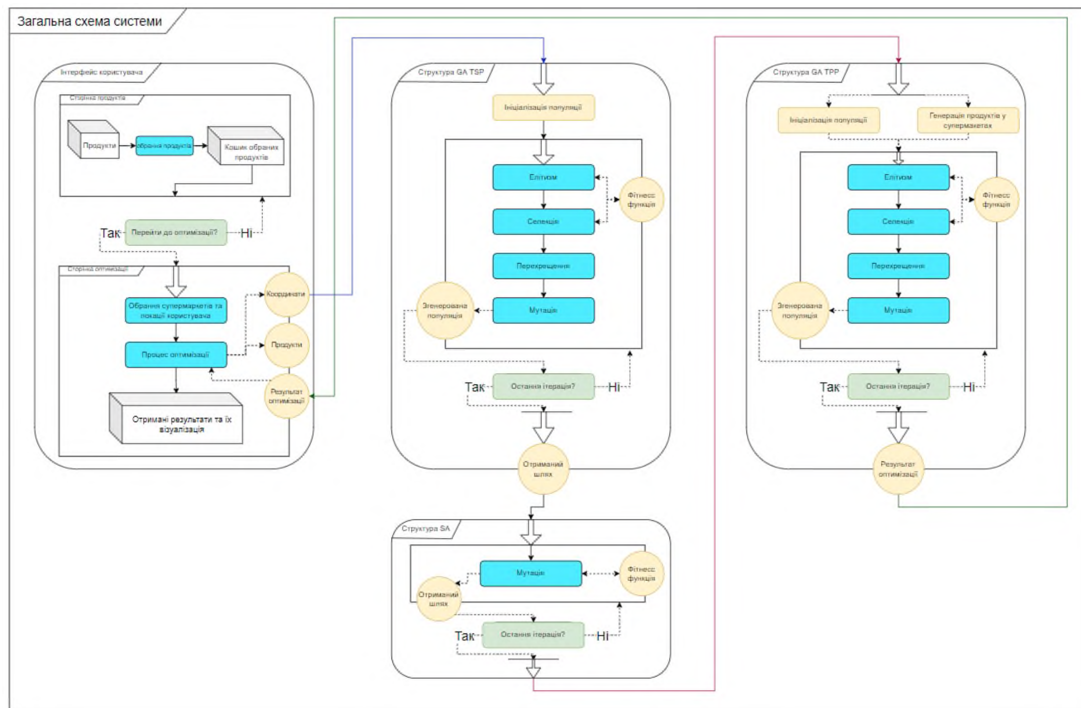


Рис. 2.2.4 Загальна схема системи

Висновки до розділу

На початку цього розділу було проаналізовано предмет дослідження, сформовано та розкрито метод, який використовується для вирішення поставленої задачі. Далі було проаналізовано вхідні дані для коректної роботи алгоритму та вихідні дані, які повинні бути отримані з кожного шару при успішному виконанні завдання.

РОЗДІЛ 3. МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ

3.1. Математичне та алгоритмічне забезпечення

Тепер, коли ми отримали загальне уявлення про систему, необхідно більш детально описати її компоненти.

Як зазначалося вище, інтерфейс користувача включає в себе продукти, список покупок і функцію вибору координат. Кожен товар у супермаркеті має основні параметри: ціна та назва, де назва є постійною, а ціна може змінюватися в межах певного середнього радіусу. Для вибору координат рекомендується використовувати онлайн-карту. Кожна локація, обрана користувачем, має дві характеристики: довготу та широту. Кожна координата є унікальною і не може бути однаковою, оскільки їх комбінація утворює лише одну точку на землі. Основна логіка цього шару - візуалізація та наочність роботи для користувача, оскільки всі дані для наступного шару побудованої системи вже взяті, тому на цьому етапі перший шар не потребує подальшої конкретизації і математичної моделі не існує.

Другий шар системи отримує координати системи у вигляді довготи та широти і передає їх як вхідні параметри генетичному алгоритму TSP. Його першим кроком є ініціалізація популяції. Цей крок має вирішальне значення для роботи алгоритму. Існує два типи ініціалізації: випадкова та евристична. Евристична ініціалізація намагається знайти і застосувати певні шаблони з індивідуумів відповідно до завдання. Основним недоліком є відсутність різноманітності, і така ініціалізація підходить не для всіх завдань. Випадкова ініціалізація, з іншого боку, вирішує перший недолік. У задачі комівояжера люди повинні пройти через кожне місто і повернутися до початкової точки, тому в цьому випадку використовується випадкова генерація.

Геном - це місто з певними координатами. Хромосоми та індивіди - це набір генів. Таким чином, в загальному вигляді індивіда можна представити наступним чином:

6 (lat., lng.)	2 (lat., lng.)	4 (lat., lng.)	3 (lat., lng.)	1 (lat., lng.)	5 (lat., lng.)

Де число – це назва міста, lat. (latitude) – широта міста та lng. (longitude) довгота міста. Кожне місто проходиться з лівої частини у праву, доки не дійде початкової точки. У хромосому не входить початкова точка (точка користувача).

Під час генерації нової популяції, міста перемішуються випадковим чином та утворюють хромосому. Місткість популяції залежить від вказаних вхідних параметрів алгоритму, але зазвичай це число дорівнює наступній формулі (формула 2.1):

$$C_P = n * m, \quad (3.1)$$

де n – це кількість міст, m – коефіцієнт, який відбирається в залежності від потреб алгоритму і вирішується під час аналізу результатів.

Кожна хромосома - це конкретний розв'язок задачі комівояжера. Важливим елементом алгоритму є фітнес-функція. Вона визначає, наскільки одна хромосома краща за інших.

Загалом, фітнес-функцію в задачі комівояжера можна описати як оцінку відстані між містами. Оскільки побудована система працює в реальному середовищі, необхідно визначити відстані шляхом обчислення відстаней, обумовлених шириною та довготою міст. Для цього можна використати формулу Гавеласа, яка обчислює відстань між точками на сфері. (формула 2.2):

$$D = R * arccos[(sin(lat_1) * sin(lat_2)) + cos(lat_1) * cos(lat_2) * cos(long_2 - long_1)], \quad (3.2)$$

де R – це радіус землі та дорівнює 6.379 км.; lat_1, lat_2 – широти першого та другого міста та $long_1$, $long_2$ – довготи. Щоб приблизити фітнес-функцію до реальності, потрібно не тільки обраховувати шлях через формулу гаверсинуса,

а й враховувати витрачені кошти на паливо, щоб дістатися з одного місця до іншого. Для цього скористаємось наступною формулою (формула 2.3):

$$C_{petrol} = A_{cost} * \left(A_{usage} * \frac{D}{100} \right), \tag{3.3}$$

де A_{cost} – середня ціна палива на території України, A_{usage} – середня кількість використання палива у літрах, D – відстань від одного міста до іншого обчисленого за формулою гаверсинуса.

Іншими словами, при обчисленні придатності хромосоми виконується наступний процес: у функцію включаються всі гени і початкові точки (точки користувача) обраної хромосоми. Спочатку обчислюється вартість шляху від початкової точки до першого міста, потім обчислюється вартість шляху до всіх міст в порядку розташування хромосоми. Відстані додаються, нормалізуються і повертаються у вигляді числового значення.

Після формування фітнес-функції відбуваються основні етапи алгоритму. По-перше, у фазі еліти встановлюється елітний потенціал. Елітний потенціал розраховується за наступною формулою. (формула 2.4):

$$C_E \approx \frac{C_P}{2}, \tag{3.4}$$

де C_P – ємність популяції.

Основна мета елітизму - зберегти найкращі особини з попереднього покоління, щоб генетичний алгоритм сходився до деякого оптимального значення швидше, ніж зазвичай. На кожній ітерації елітизму функція пристосованості використовується для пошуку найкращих особин, а знайдені хромосоми додаються до нової популяції і викидаються зі старої. Основна фаза TSP GA починається там, де "елітарність" попереднього покоління ізольована.

Останні популяції, до яких елітарність не була застосована, відбираються. У фазі відбору відбираються найкращі хромосоми для покоління нащадків. Для цього в гру вступає параметр кількості турнірних кандидатів. Будь-який процес відбору кандидатів може бути турніром. У загальному випадку турнір складається з декількох послідовних етапів, на кожному з яких кількість учасників турніру зменшується. Цей процес варіюється в залежності від

конкретної задачі, але в нашому випадку хромосоми тасуються, щоб вибрати певне число. Це число зазвичай обирається відповідно до здібностей популяції.

Етап відбору відбувається двічі, при цьому обирається перший і другий батько. На цьому етапі гени батьків мають певним чином сформувати нові хромосоми. Для цього береться половина генів від першого батька і половина від другого. Спочатку нову хромосому потрібно заповнити генами від першого батька, а для другої половини перевірити, чи не перетинаються міста від другого батька з містами від першого. Іншими словами, неможливо створити шлях, де хромосоми повторюють міста. Це суперечить концепції задачі комівояжера, оскільки призводить до катастрофічних наслідків для алгоритму. Тому, якщо місто не знайдено в першому батькові, до хромосоми додається місто другого батька. У загальному вигляді цей процес можна описати наступним чином:

Припустимо, ви вибрали послідовність генів, яку хочете зберегти від першого батька (виділено жовтим кольором) і перенести до нової особини.

Батько №1

6 (lat., lng.)	2 (lat., lng.)	4 (lat., lng.)	3 (lat., lng.)	1 (lat., lng.)	5 (lat., lng.)

Батько №2

2 (lat., lng.)	3 (lat., lng.)	1 (lat., lng.)	4 (lat., lng.)	5 (lat., lng.)	6 (lat., lng.)

Для цього нам потрібно змінити гени батька №2 так, щоб не пошкодити саму структуру, тобто перенести гени з батька №1 в батька №2 в певній послідовності. Спочатку в нову хромосому переносяться гени батька №1:

Хромосома з генами батька №1

		4 (lat., lng.)	3 (lat., lng.)	1 (lat., lng.)	

Далі проходимось по кожній вільній клітинці сина, заповнюючи її генами батька №2, при цьому перевіряючи чи гени з батька №2 не співпадають з генами першого. У результаті отримуємо нову хромосому:

Хромосома з генами батьків №1 та №2

2 (lat., lng.)	5 (lat., lng.)	4 (lat., lng.)	3 (lat., lng.)	1 (lat., lng.)	6 (lat., lng.)

Після гібридизації формується популяція, яка на 50% складається з еліти попереднього покоління і на 50% з нових хромосом. Потім настає завершальний етап - етап мутацій.

Мутація - найцікавіший етап. Мутації необхідні для того, щоб нові хромосоми відрізнялися одна від одної, тобто для урізноманітнення генів після схрещування. Мутації відбуваються з певною ймовірністю, тому надмірне збільшення ймовірності призведе до поганих результатів, оскільки мутації відбуватимуться завжди.

У задачі про комівояжера мутація - це дуже проста операція заміни міст. У новоствореній хромосомі випадковим чином вибираються два гени, причому перший ген розміщується на місці другого, а другий - на місці першого. Припускаючи, що гени, які підлягають заміні, позначені жовтим кольором, операцію можна представити наступним чином:

Новостворена хромосома

2 (lat., lng.)	5 (lat., lng.)	4 (lat., lng.)	3 (lat., lng.)	1 (lat., lng.)	6 (lat., lng.)

Мутована хромосома

1 (lat., lng.)	5 (lat., lng.)	4 (lat., lng.)	3 (lat., lng.)	2 (lat., lng.)	6 (lat., lng.)

На цьому генетичний цикл завершується. Кількість таких циклів залежить від заданої кількості поколінь; оскільки вхідні параметри GA дуже чутливі, зазвичай їх краще визначати експериментально.

Після всіх поколінь GA отримує оптимізований шлях. Тепер необхідно зрозуміти, чи є знайдені шляхи оптимальними, тобто чи знайшов GA локальний або глобальний оптимум. Для цього використовується алгоритм SA.

Оскільки SA базується на зміні температури, необхідно на кожній ітерації виконувати певні операції над шляхом, щоб перевірити, чи було його оптимізовано. Для цього використовується одна з фаз GA - мутація: SA

починається із встановлення початкової температури, яку можна отримати з наступного рівняння. (формула 2.5):

$$T = \sqrt{n}, \quad (3.5)$$

де n – це кількість генів (міст) у хромосомі (отриманому шляху з GA).

Тому на кожній ітерації гени обмінюються до тих пір, поки температура не впаде до певного рівня. Після обміну поточний результат перевіряється на те, чи є він кращим за попередній, і якщо ні, то попередній результат обирається як найкращий на поточному етапі. Однак така операція не дає глобального оптимуму. Тому в МА використовується певна ймовірність того, що на певній ітерації буде обрано гірший шлях. Ця ймовірність визначається наступним рівнянням. 2.6:

$$P = N_r \leq \ e^{-1 \cdot (\frac{(F_{cur} - F_{next})}{T})}, \quad (3.6)$$

де N_r – це випадково згенероване число, F_{cur} – вартість шляху на минулій ітерації, F_{next} – вартість шляху на теперішній ітерації, T – температура. Якщо температура менша 0, то $T = 1$.

Вибір найгіршого шляху не дуже перспективний, але він допомагає алгоритму уникнути локального оптимуму і наблизитися або знайти глобальний оптимум. Після завершення та охолодження ітерацій GA генерує шляхи, які потрапляють на третій рівень (рівень, де GA використовується для розв'язання задачі мандрівного клієнта).

Як і на початку другого рівня, все починається з ініціалізації початкової популяції. Популяція складається з індивідуумів, гени яких представлені супермаркетами (містами), а конкретні покупки здійснюються в конкретних супермаркетах. На цьому етапі виникає перша проблема. Це проблема різноманітності покупок. Тобто, якщо генерується популяція, в якій всі товари купуються лише в одному супермаркеті, то результат GA TPP не буде оптимальним. Для вирішення цієї проблеми було запропоновано наступне рішення. Для кожного індивіда генерується випадковий порядок супермаркетів, а потім придбані товари додаються до кожного супермаркету по черзі.

Наприклад, нехай є такі товари: банани, помідори, огірки, яблука, груші, лимони та апельсини. Коли товар проходить через кожен супермаркет, до нього додається по одному продукту за раз. Цей процес можна представити наступним чином:

Хромосома TPP

1 (lat., lng.) [банан]	5 (lat., lng.) [помідор]	4 (lat., lng.) [огірок]	3 (lat., lng.) [яблуко]	2 (lat., lng.) [груша]	6 (lat., lng.) [лимон]

Коли було пройдено кожен супермаркет у хромосомі, але не куплено всі товари, у нашому випадку не було куплено апельсин, все починається з знову з першого супермаркету. Фінальний результат згенерованих покупок буде наступний:

Хромосома TPP із заповненими продуктами

1 (lat., lng.) [банан, апельсин]	5 (lat., lng.) [помідор]	4 (lat., lng.) [огірок]	3 (lat., lng.) [яблуко]	2 (lat., lng.) [груша]	6 (lat., lng.) [лимон]

Після генерації популяції перед основним циклом ГА потрібно виконати ще одну операцію: функція пристосованості TPP подібна до TSP, але оскільки потрібно враховувати шляхи, певні функції, такі як визначення відстані та витрат на паливо, копіюються з попереднього шару Функція пристосованості GA TPP базується на наступних кроках для кожного гена Вона починається з багаторазового проходження гена та визначення загальної вартості покупок, зроблених на цьому гені. Щоб визначити це, просто проаналізуйте кожну покупку, здійснену на певному гені, і підсумуйте їх. Наступним кроком є визначення вартості пального від поточного місця (як початкової точки, так і гена, де була здійснена покупка) до наступного місця. Останній крок - повернути суму вартості пального та вартості покупок. Якщо в поточному гені не було зроблено жодної покупки, то ген пропускається і відбувається перехід до наступного гена. Якщо досягнуто останнього гена і там не було зроблено

жодної покупки, маршрут до початкової точки відраховується від попереднього супермаркету, де були зроблені покупки.

Тепер, коли функція пристосованості та початкова популяція готові, виконується основний цикл GA TPP. У цій реалізації також присутній елітизм. Операція елітизму в цьому шарі нічим не відрізняється від операції в GA TSP, тому немає сенсу пояснювати її в цьому шарі. Це стосується і вибору кандидатів на схрещування. Перейдемо до фази кросинговеру, оскільки всі наступні алгоритми відрізняються.

На етапі кросинговеру відбувається обмін генами для створення нових хромосом так само, як і в GA TSP. Перша операція полягає в тому, щоб розділити кошик між двома батьками так, щоб половина кошика належала першому з них, а інша половина - другому. Далі заповнюються гени першого з батьків, щоб з'ясувати, чи були товари в кошику батьків куплені в певному супермаркеті, і цей супермаркет і товари передаються нащадкам:

Жовтим кольором позначено першого батька, зеленим - товар, який розглядав другий батько.

Кошик з продуктами

банан	помідор	огірок	яблуко	груша	лимон	апельсин

Хромосома батька TPP №1

1 (lat., lng.) [банан]	5 (lat., lng.) [помідор]	4 (lat., lng.) [огірок]	3 (lat., lng.) [груша, апельсин]	2 (lat., lng.) [лимон]	6 (lat., lng.) [яблуко]

Хромосома батька TPP №2

1 (lat., lng.) [груша]	5 (lat., lng.) [помідор, банан]	4 (lat., lng.) [апельсин]	3 (lat., lng.) [лимон]	2 (lat., lng.) [лимон]	6 (lat., lng.) [яблуко]

Для початку заповнимо нащадка генами з отриманого з переднього шару, але без покупок:

Хромосома нащадка TPP

1 (lat., lng.) []	5 (lat., lng.) []	4 (lat., lng.) []	3 (lat., lng.) []	2 (lat., lng.) []	6 (lat., lng.) []

Наприклад, розглянемо перший супермаркет батька №1, який придбав банани. Якщо ми знайдемо цей супермаркет у нащадків і додамо продукти, придбані у батька, до першого супермаркету, і зробимо те ж саме для всіх генів батьків №1 і №2, то результат повинен виглядати наступним чином:

Хромосома нащадка TPP після кросоверу

1 (lat., lng.) [банан, груша]	5 (lat., lng.) [помідор]	4 (lat., lng.) [огірок, апельсин]	3 (lat., lng.) [лимон]	2 (lat., lng.) []	6 (lat., lng.) [яблуко]

Після того, як після спарювання отримано потомство, починається фінальна стадія мутації. Для цього на кожній ітерації обробки генів особини генерується випадкова послідовність супермаркетів і відкидаються гени, що мутують. Мутація здійснюється наступним чином: отримуються всі покупки, зроблені в даному супермаркеті, генерується випадкове число від 1 до кількості покупок, і генерується випадковий супермаркет, в який переносяться куплені продукти. Потім, після отримання випадкової покупки, це робиться з іншим геном, і ця ітерація видаляє покупку з гена. Іншими словами, покупки переносяться з цього гена на інший ген:

Хромосома нащадка TPP

1 (lat., lng.) [банан]	5 (lat., lng.) [помідор]	4 (lat., lng.) [огірок]	3 (lat., lng.) [груша, апельсин]	2 (lat., lng.) [лимон]	6 (lat., lng.) [яблуко]

Допустимо обраний продукт це помідор (виділено жовтим), випадковим чином було обрано супермаркет №2, після виконання мутації ми отримуємо наступне:

Хромосома нащадка TPP після мутації

1 (lat., lng.) [банан]	5 (lat., lng.) []	4 (lat., lng.) [огірок]	3 (lat., lng.) [груша, апельсин]	2 (lat., lng.) [лимон, помідор]	6 (lat., lng.) [яблуко]

Слід зазначити, що мутації повинні включати коефіцієнт мутації. Для того, щоб його застосувати, потрібно було прийняти рішення про те, як організувати замовлення на закупівлю. Якщо умова коефіцієнта мутації виконується, тобто випадково згенероване число є меншим за коефіцієнт мутації, то виконується наступна дія: повністю передати закуплений товар іншому випадково обраному гену.

Таким чином, після завершення всіх запланованих поколінь, отримується оптимізований шлях із закупленими товарами. Ця інформація повертається на перший рівень. Залишається лише показати користувачеві всю інформацію про вартість шляху та вартість покупки. Бажано також показати на карті маршрути, які користувачеві необхідно подолати в обох напрямках.

Висновки до розділу

Було спроектовано систему, визначено її основні частини та відображено на схемі. Система була розділена на три рівні:

- Інтерфейс користувача;
- шар GA TSP;
- шар GA TPP.

Кожен рівень системи був детально описаний покроково, висвітлюючи реалізацію GA для кожного рівня та кінцевий результат, який повинен бути отриманий, якщо кожен крок GA був реалізований. Автори надають власну реалізацію і рішення проблеми мандрівного клієнта за допомогою GA.

РОЗДІЛ 4. ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ

4.1. Засоби розробки

Веб-сайт - це, перш за все, візуалізація логіки, яку має на увазі розробник. Для цього розробникам потрібні такі технології, як CSS, JS та HTML. Це мінімальний набір, який вже досить давно використовується для створення частини користувацького інтерфейсу будь-якого сайту. На жаль, зі зростанням популярності сайтів зростає і кількість користувачів, що вимагає значної оптимізації. Вирішенням цієї проблеми є різноманітні фреймворки (бібліотеки), якими зазвичай користуються ФОП-розробники.

У більшості випадків виділяють три основні фреймворки: Vue, Angular і React.

React - це фреймворк, створений компанією Facebook, і є найшвидшим з перерахованих вище. Він використовує технологію Virtual DOM для запису змін. Коли вносяться зміни, вони спочатку записуються у віртуальний DOM. На відміну від традиційного DOM, де сторінка регенерується кожного разу, коли виявляються зміни, концепція віртуального DOM регенерує сторінку лише тоді, коли зміни більше не існують. Коли компілятор розпізнає, що зміни більше не вносяться, віртуальний DOM об'єднується зі звичайним DOM. Це усуває необхідність повторного оновлення інформації на стороні клієнта.

На жаль, React майже не використовує ООП у своїх компонентах, що не підходить для нашої архітектури. React також використовує єдиний потік даних, що є досить застарілим підходом і не підтримується більшістю нових фреймворків.

Vue - відносно новий фреймворк, який швидко набирає обертів. Звичайно, Vue не такий популярний, як React або Angular, але він не поступається; Vue поділяє погляди Angular; Vue підтримує архітектурний патерн MVVM; і Vue є дуже хорошим прикладом архітектури MVVM. По суті,

це означає, що Vue має досить великий HTML-бамп і, на відміну від React, який використовує HTML з JS, Vue використовує JS з HTML. Це дозволяє розробникам швидко створювати нові компоненти і використовувати їх в HTML. Це також покращує якість та читабельність коду. Крім того, ще однією перевагою Vue є те, що перехід з Vue на інші фреймворки не є серйозною проблемою: Vue є високо оптимізованим фреймворком, а вага самого веб-контейнера дозволяє йому швидше генерувати сторінки, коли користувачі відвідують певний сайт.

Оскільки Vue є відносно новим фреймворком, багато інструментів, які вже давно реалізовані в інших фреймворках, все ще перебувають на стадії розробки. Звичайно, Vue підтримується, але підтримка дуже обмежена в порівнянні з іншими фреймворками. Тому, на жаль, Vue не є найкращим рішенням для нашої архітектури.

Angular - це фреймворк, розроблений компанією Google у 2009 році. Спочатку він використовував архітектуру MVC, яка була стандартною на той час. Спочатку Angular зовсім не був популярним через свою оптимізацію; концепція JS всередині HTML здавалася досить гарною ідеєю навіть тоді, тому Google почав активно розвивати її. Після виходу Angular 2 архітектура змінилася з MVC на MVVM, як ми вже згадували раніше. Це зіграло головну роль у популярності та корисності веб-додатків на основі Angular 2. Першою перевагою Angular перед React було використання Typescript на противагу JSX, що дозволяє використовувати сучасну архітектуру, шаблони ООП та функції. Це означає, що Angular має власний інтерфейс командного рядка, який дозволяє створювати та підтримувати компоненти з командного рядка. Документація також є однією з головних переваг Angular, оскільки в ній є абсолютно все, тому немає необхідності шукати багато інформації в Інтернеті.

Це означає, що кожен модуль є контейнером для компонентів, що допомагає відокремити логіку компонентів і директив від решти.

Angular ідеально підходить для великих проектів, які в майбутньому повинні будуть розширюватися, і прийнято орієнтуватися на бібліотеку матеріалів Angular. Це допомагає розробникам використовувати готові компоненти, такі як автозаповнення, діалогові та навігаційні вікна, слайдери та каруселі. Наприклад, у звичайному JS є функція Promise, яка часто використовується для передачі або завантаження даних з сервера. Після виконання, збирач сміття видаляє цю функцію з пам'яті і більше не використовує її. Якщо ви хочете щоразу робити запит до сервера, вам потрібно створювати нову функцію; розробників Angular це не влаштовувало і вони знайшли альтернативу під назвою RxJS. Хоча це не розробка Google, Google підтримує цю бібліотеку і вона допомогла вирішити багато проблем, пов'язаних зі звичайним JS. У RxJS ви можете підписатися за допомогою патерну під назвою Observable. Він підписується на зміни в певному компоненті, і коли він отримує зміни, розробник використовує його для своїх цілей. Після цього Observable не видаляється з пам'яті і продовжує відстежувати зміни в компоненті.

Angular може надати розробникам широкий спектр контенту, але він не є досконалим. Він є найповільнішим з перерахованих вище фреймворків, може бути масивним у створенні і досить складним для клієнтської сторони.

Проаналізувавши та зваживши всі "за" і "проти", Angular стає фаворитом.

Звичайно, при використанні фреймворку Angular, на відміну від Typescript, буде досить складно використовувати інші мови, наприклад, python. Фронтенд потрібно поєднувати з бекендом, що вже виходить за рамки цієї статті. Це пов'язано з тим, що інтерфейсне середовище виконання має бути поєднане з бекендом, що вже виходить за рамки цієї статті. Тому для простоти реалізації для алгоритму використовується Timescript.

Для відображення користувачеві певної інформації, наприклад, маршрутів, використовується Google Maps та його функції. Для кращої

візуалізації результатів роботи алгоритму TSP 2-го рівня експериментальні результати переносяться на мову Python та візуалізуються за допомогою спеціальних бібліотек та середовища Google Colab.

4.2. Вимоги до технічного та програмного забезпечення

Для роботи цієї системи на комп'ютері користувача потрібні: базовий комп'ютер з хорошим доступом до Інтернету та щонайменше 2 ГБ оперативної пам'яті та 1 ГБ відеопам'яті. Інтернет потрібен користувачеві для переходу до домену і для того, щоб сервіс Google Maps міг обробляти дані про географічне розташування користувача і місто проживання.

4.3. Опис програмної реалізації

Для програмної реалізації потрібно встановити та налаштувати Angular. Далі встановлюємо бібліотеку ng-zorro, яка використовується для візуалізації деяких компонентів інтерфейсу.

Після того, як всі програми, необхідні для початку роботи, встановлені, можна переходити до реалізації першого шару. Оскільки цей рівень системи є інтерфейсом користувача, першим кроком є створення продуктів, які будуть доступні в супермаркеті. Зазвичай продукти поділяються на категорії, але тут для простоти було обрано дві категорії: фрукти та овочі. Для перемикання між категоріями створюється меню, в якому користувач може перемикатися між категоріями:

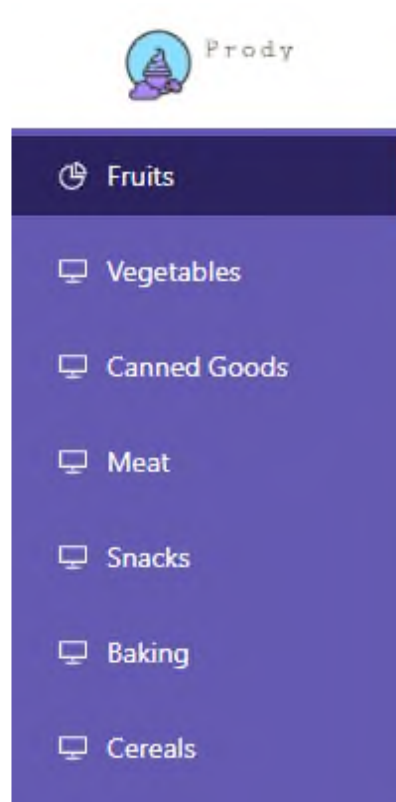


Рис. 4.3.1 Бокове меню з категоріями

У бічному меню є й інші категорії, але вони використовуються для його наповнення, а, як уже згадувалося, функціональних категорій лише дві: овочі та фрукти. Тепер давайте створимо картку товару, щоб користувач міг зробити вибір для покупки. Картка товару повинна містити основні параметри, такі як ціна та назва. Щоб зробити картку товару більш наочною, ми додали деякі додаткові параметри. А саме, вага для певної ціни, рейтинг товару та зображення.

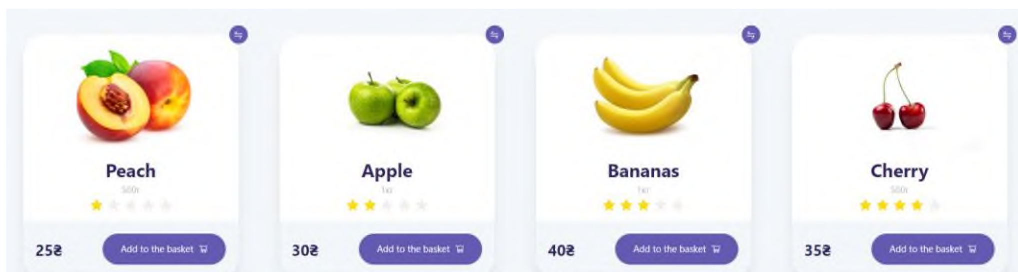


Рис. 4.3.2 Картки продуктів

Ціни на товари усереднюються. Це означає, що ціна конкретного товару, яка відображається на картці, є середньою ціною по всіх супермаркетах (деталі пояснюються на другому рівні реалізації). На картці є додаткова кнопка, яка дозволяє користувачеві додавати товари до кошика. Для цього створюється кошик з товарами:

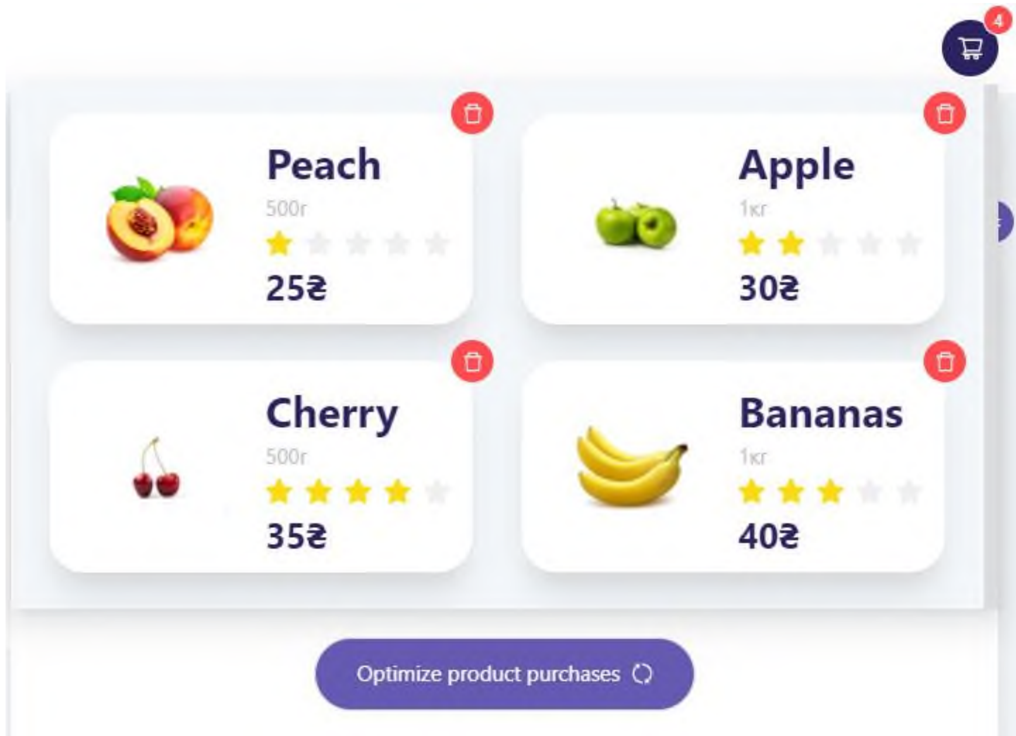


Рис. 4.3.3 Корзина з продуктами

Кошик має наступну логіку: коли користувач додає товар до кошика, у верхньому правому куті додатку з'являється іконка кошика з цифрою - кількістю товарів у кошику. При наведенні курсору на кошик з'являється вікно (Рис. 3.3.3), в якому показані товари, які користувач хоче придбати. Якщо товар було додано помилково, його можна видалити за допомогою іконки кошика біля кожної картки товару. Після завершення додавання товарів і відкриття кошика, користувач може натиснути кнопку "Оптимізувати покупку", щоб перейти на сторінку оптимізації.

Як згадувалося в попередньому розділі, найкращий спосіб візуалізувати маршрут - це відобразити його на карті. Для цього підключіть Google Maps до проекту Angular; оскільки Google Maps є платним сервісом, для коректної

роботи необхідно його придбати. Для цього підключіть свою банківську картку до Google Maps і активуйте необхідні сервіси:

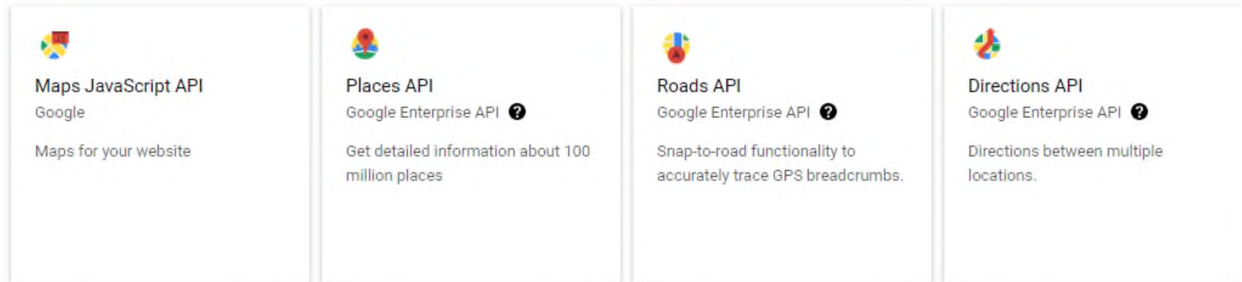


Рис. 4.3.4 Google Maps сервіси

Тепер все що нам залишається зробити це згенерувати API ключ та вставити його у Angular проект:

```
AgmCoreModule.forRoot({
    apiKey: 'BIza4yB6xs6Kczo46L54QEFJTRdrN0YU44sR_2Q'
}),
AgmDirectionModule
```

Рис. 4.3.5 Вставлений API ключ у головний модуль проекту

Після того, як ви виконали всі попередні кроки, ви можете розпочати розробку мапи, встановивши всі бібліотеки, необхідні для роботи з нею. Наступним кроком буде створення карти:

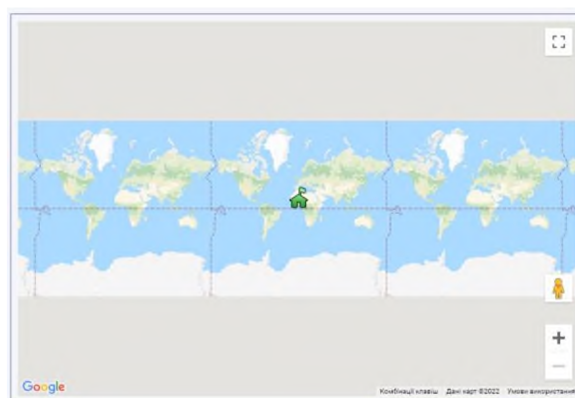


Рис. 4.3.6 Створена карта

Додайте покрокові інструкції, щоб зробити його зручним для користувача. Давайте розділимо оптимізацію на чотири простих кроки:

- Вибір місця розташування користувача
- Вибір супермаркету
- Процес оптимізації
- Візуалізація маршруту на карті

Щоб реалізувати функціонал, додайте до кожного кроку кнопки та інструкції, щоб користувачеві було легко користуватися функціоналом:

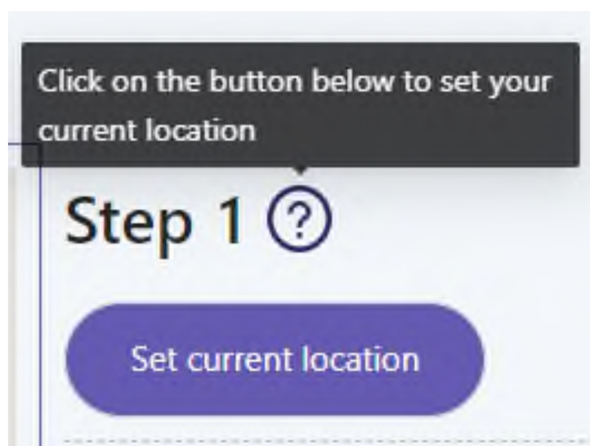
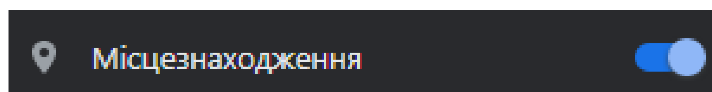


Рис. 4.3.7 Кнопка запуску першого кроку

Це означає, що першим кроком після натискання кнопки є отримання координат місцезнаходження користувача. Для цього користувачеві потрібно перевірити в налаштуваннях свого браузера, чи дозволяє він обробку геолокаційних даних:



Потім встановлена бібліотека використовується для визначення геолокації користувача і прив'язки цього місця до кнопки "Встановити поточне місцезнаходження". Після натискання кнопки користувач отримує своє місцезнаходження на карті, а його позиція позначається зеленим кольором.

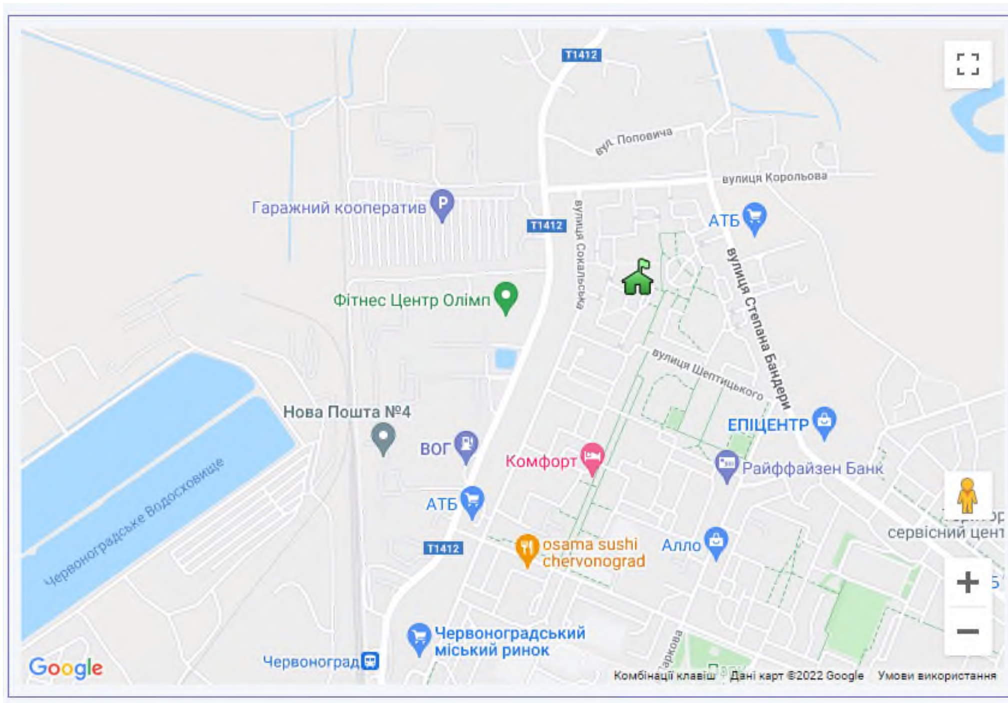
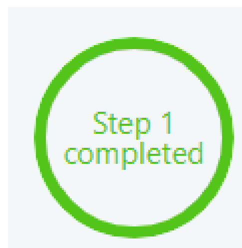


Рис. 4.3.8 Встановлене місцезнаходження користувача

Для розуміння того, що крок відбувся, добавимо в систему своєрідний прогрес, який буде означати закінчення кроку:



Тепер, коли перший крок завершено, переходимо до другого. Результатом другого кроку мають бути дані про супермаркети, а точніше координати. Тому необхідно додати функцію вибору конкретних точок. Це пов'язано з тим, що не всі точки на карті є супермаркетами. Однак на поточному етапі всі точки, обрані користувачем, вважаються супермаркетами, оскільки планується випадковим чином генерувати ціни в супермаркетах. У майбутній реалізації користувачі зможуть вибирати супермаркет у своєму районі, а продукти будуть вибиратися через API цього супермаркету.

Створіть функцію для вибору точок на карті. Позначте кожну точку, обрану користувачем, як "супермаркет" і присвойте їй ціле число, що відповідає кількості обраних супермаркетів:

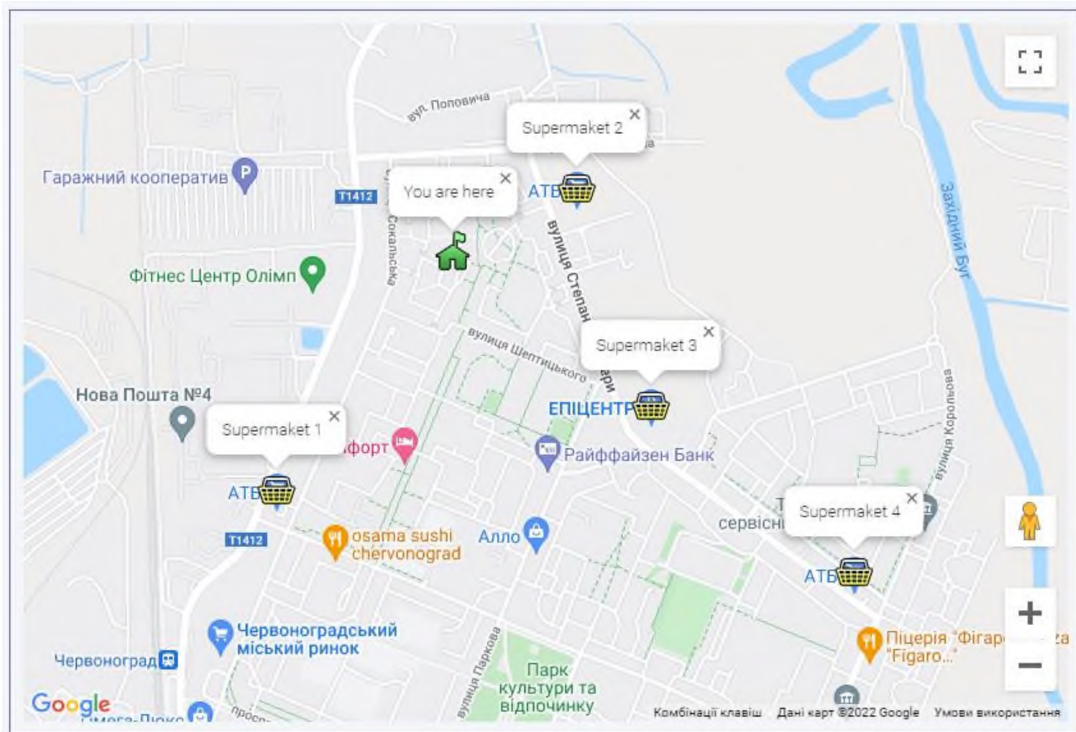


Рис. 4.3.9 Позначені користувачем супермаркети на карті

Тепер, коли всі основні вхідні дані отримано, переходимо до третього етапу реалізації алгоритму - інтерфейсу, тобто шарів 2 і 3 системи. Як пояснювалося в попередньому розділі, шар 2 системи містить координати користувача і координати позначених суперсуперів; оскільки система, створена в Angular, має модульну структуру, то в ньому реалізовані основні класи та інтерфейси загального генетичного алгоритму: особини, гени, популяції, Почніть зі створення інтерфейсу популяції.

```
You, 2 weeks ago | 1 author (You)
import { ProductModel } from "src/app/shared/interfaces/product.interface";

You, 2 weeks ago | 1 author (You)
export interface GeneModel {
    name: string;
    lat: number;
    lng: number;
    geneProducts: GeneProductModel[];

    getProductsCost(): number;
    getDistance(destination: GeneModel): number;
    getPetrolUsage(destination: GeneModel): number;
}

You, 2 weeks ago | 1 author (You)
export interface GeneProductModel extends ProductModel {
    hasBought: boolean;
}
```

Рис. 4.3.10 Інтерфейс загального гена GA

Цей інтерфейс виступає своєрідним описом або схемою для реалізації суті супермаркету. Використовуються методи для вилучення інформації про назву, довготу, широту, перелік товарів і параметрів.

```
You, 2 weeks ago | 1 author (You)
import { GeneModel } from './gene.interface';
import { StartGeneModel } from './start-gene.interface';

You, 2 weeks ago | 1 author (You)
export interface IndividualModel {
    genes: GeneModel[];
    startGene: StartGeneModel;
    _fitness: number;
    _travelCost: number;

    addStartPoint(): void;
    addGene(gene: GeneModel): void;
    copyGenes(): GeneModel[];
    printGenes(): void;
    resetParams(): void;
}
```

Рис. 4.3.11 Інтерфейс загального індивіда GA

Загальна схема індивіда для GA, яка містить список генів (супермаркетів), початковий ген, значення фітнес-функції та загальна вартість шляху.

```
You, 2 weeks ago | 1 author (You)
import { PriceIndividual } from '../classes/price-classes/price-individual.class';
import { TspIndividual } from '../classes/tsp-classes/tsp-individual.class';

You, 2 weeks ago | 1 author (You)
export interface PopulationModel {
    individuals: PriceIndividual[] | TspIndividual[];

    addIndividual(individual: PriceIndividual | TspIndividual): void;
    removeIndividual(individual: PriceIndividual | TspIndividual): void;
    getFittestIndividual(): PriceIndividual | TspIndividual;
}
```

Рис. 4.3.12 Загальний інтерфейс популяції GA

Ця діаграма популяції містить масив особин та методи маніпулювання ними.

Описавши загальний інтерфейс генетичного алгоритму, перейдемо до другого рівня системи, а саме до реалізації GA TSP. Оскільки ми лише порівнюємо відстані (тобто потрібні лише на третьому рівні), то товар на цьому етапі нам не потрібен, але ми все ж генеруємо товар зараз, щоб отримати повноцінний ген, а точніше, щоб використовувати інтерфейс, описаний раніше. Ціни на товари генеруються наступним чином: для кожного товару в конкретному гені генерується певний діапазон випадкових чисел. Оскільки середньостатистичний український товар коштує 25-50 гривень, ми вирішили обрати випадкове число від 1 до 5. Це число додається до кожного товару. Тепер кожен ген матиме різну ціну для кожного товару. Оновлені гени передаються генетичному алгоритму.

Як пояснювалося в розділі 2, кожен GA має власні вхідні параметри, основними з яких є розмір популяції, кількість ітерацій та швидкість мутацій. Розмір популяції задається за формулою, описаною в розділі 2, тобто CP= n*m, де m=100, кількість ітерацій 20 і швидкість мутацій 0.05. Кількість турнірів дорівнює n, де n - кількість міст (супермаркетів).

Далі починається генерація початкової популяції. Вони формуються випадковим чином згідно з алгоритмом. Отримані гени передаються в генеруючу функцію і застосовується перестановка з використанням бібліотеки Лодаша і методу перестановки. Після отримання початкової популяції з різних

шляхів, вона передається на генетичний цикл. Цикл починається з елітизму. За схемою GA TSP, описаною в розділі 2, розмір популяції ділиться навпіл, щоб отримати розмір елітизму. Популяція досліджується і відбирається найкраща половина особин, тобто еліта. Еліта відбирається за допомогою фітнес-функції. Визначається відстань від першого гена до наступного гена в кілометрах за формулою гаверсуса і розраховується використане паливо; станом на 2022.05.12 середня вартість палива становить 42 гривні, тому ця сума використовується в розрахунках. Крім того, в середньому 6 літрів на 100 км використовується при їзді по місту. Тепер підставимо дані у формулу для гривні, витраченої на пальне (формула 3.1):

$$C_{petrol} = 42 * (6 * \frac{D}{100}). \quad (3.1)$$

Вартість розраховується для кожного гена, початкової та кінцевої точки. Нарешті, результат нормалізується шляхом ділення 1 на його вартість. Заповнивши половину нового покоління елітою з початкової популяції, необхідно сформувати іншу половину, тобто нове потомство. Для цього використовується відбір.

Кількість згенерованих популяцій і турнірів передається у функцію відбору. Вона використовується двічі для генерації першого і другого батьків. Батьки передаються на кросинговер, де гени від першого і другого батьків переносяться для створення нових хромосом. Новоутворена хромосома мутує, і два гени замінюються. Таким чином утворюється нова популяція.

Після всіх ітерацій генетичний алгоритм завершує свою роботу, генеруючи певні шляхи. Потім він переходить до алгоритму SA для перевірки локального оптимуму відповідно до схеми шарів; алгоритм SA виконує мутацію генів, тобто заміну генів, на кожній ітерації. Кількість ітерацій становить 500 000, а температура - квадратний корінь з кількості генів.

Після завершення роботи ми отримуємо постійний шлях для її візуалізації, тому можемо успішно використовувати бібліотеки Python, тому

перепишемо код з Typescript на Python і поекспериментуємо в середовищі Google Colab.

Спочатку згенеруємо шлях, що складається з чотирьох міст і початкової точки; параметри GA TSP наступні:

- Розмір популяції: 4 * 100 = 400.
- Кількість турнірів: 4.
- Кількість ітерацій: 16.
- Мутаційний коефіцієнт: 0.05.

Результат:

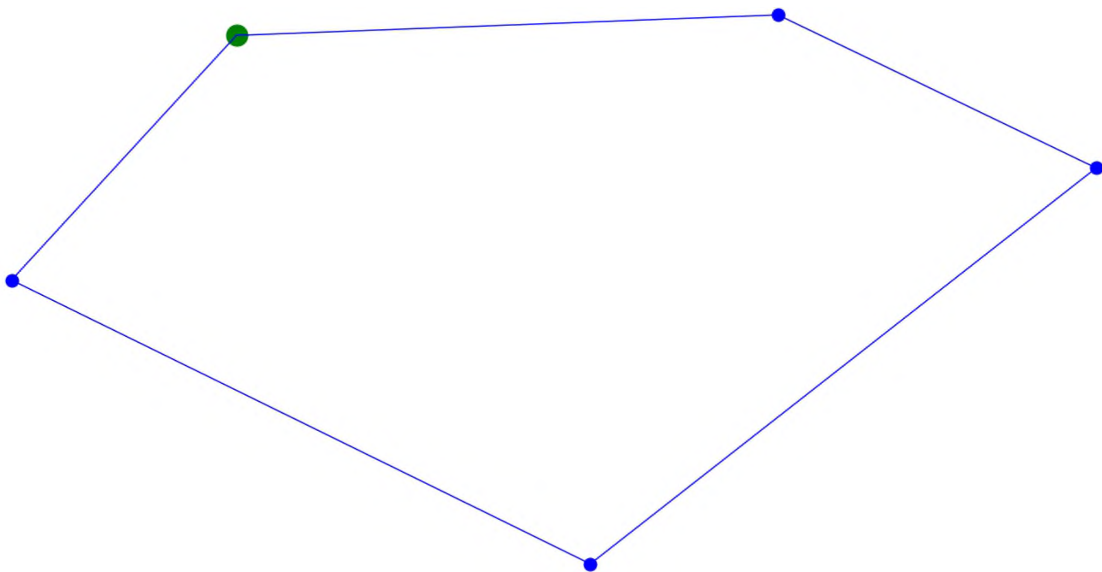


Рис. 4.3.13 Отриманий шлях GA TSP (1)

Зелені точки на зображенні позначають початкові точки, а сині - міста. У загальному випадку точки не перетинаються, тому рішення є оптимальним.

Збільшимо кількість міст до 20 і перевіримо, чи зможе шар, враховуючи параметри ГА, обробити їх для створення правильного шляху:

- Розмір популяції: 20 * 100 = 2000.
- Кількість турнірів: 10.
- Кількість ітерацій: 25.
- Мутаційний коефіцієнт: 0.05.

Результат:

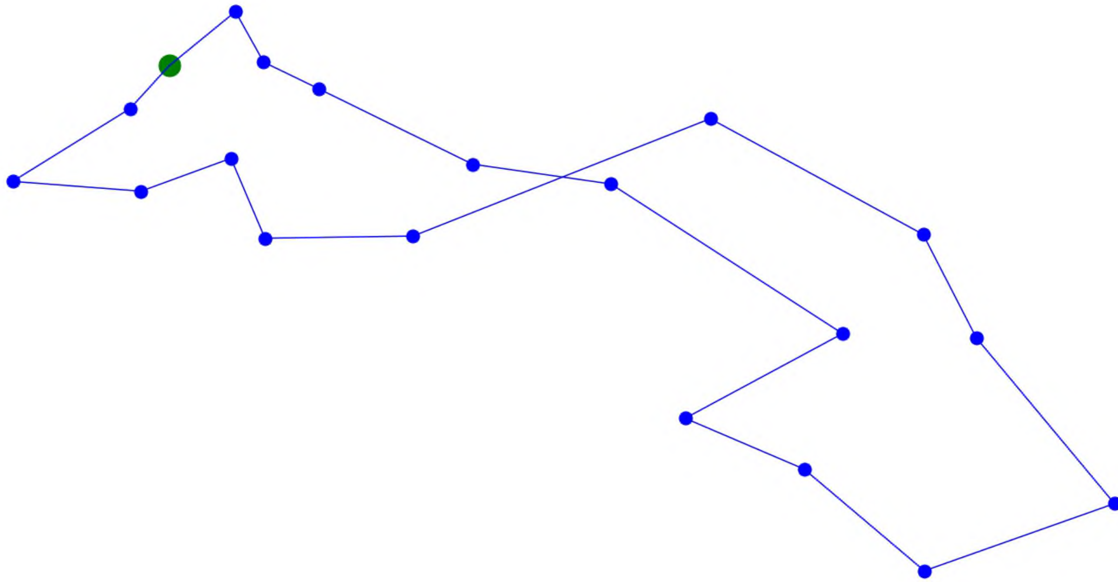


Рис. 4.3.14 Отриманий шлях GA TSP (2)

Як бачимо, немає жодних розривів або перетинів шляху, а враховуючи, що користувачі зазвичай не відвідують більше 10 супермаркетів, можна зробити висновок, що другий рівень системи працює дуже добре.

Оптимізований шлях потрапляє на останній рівень - рівень GA TPP. На цьому етапі існує оптимізований шлях від початкової точки (користувача) до кожного супермаркету та попередньо згенеровані ціни на товари для цього етапу. Потім генерується популяція для GA TPP. Для цього з кошика додається одна покупка, яка проходить через кожен супермаркет, як описано в розділі 2.

Далі відбувається елітарність та відбір, який залишається незмінним для GA TSP та програмної реалізації. Однак на цих етапах використовуються фітнес-функції. Починаючи з операційної фази, фітнес-функція була змінена. По-перше, фітнес-функція обчислює кількість покупок у кожному супермаркеті. Вона ітераційно проходить через кожен товар і перевіряє, чи був він куплений, використовуючи параметр hasBought у класі товару. Після обчислення застосовується наступна логіка

Якщо цей супермаркет вже був куплений, переходимо з поточного супермаркету і розраховуємо витрату палива.

Якщо це останній ген і покупка була здійснена, застосовується попередній крок і розраховується витрата палива до початкової точки.

Якщо це останній ген і не було здійснено жодної покупки, то розраховується лише витрата палива до початкової точки.

Після відбору отримуємо двох батьків і переходимо до схрещування. Розділіть кошик на дві частини і призначте дві частини батькам. Порожні гени без покупок передаються нащадкам, які потім частково заповнюються. Спочатку перша частина кошика заповнюється покупками першого з батьків, потім друга частина - покупками другого з батьків. Після того, як нове потомство отримано, воно відправляється на етап мутації.

Під час мутації перевіряється, чи є коефіцієнт меншим за згенероване число; якщо він більший, то всі покупки переносяться з одного випадкового супермаркету до іншого, а якщо менший, то лише одна покупка.

Оскільки цей етап відіграє важливу роль, його програмну реалізацію доцільно показати у вигляді зображення:

```
mutate(individual: PriceIndividual, mutateRate: number): void {
    individual.genes.forEach((gene) => {
        let indexes = [];

        indexes = _.range(individual.genes.length);
        indexes = indexes.filter((index) => index !== individual.genes.indexOf(gene));

        if (_.random() < mutateRate) {
            let randomCityIndex = _.shuffle(indexes)[0];

            gene.geneProducts.forEach((product) => {
                if (product.hasBought) {
                    product.hasBought = false;

                    let productIndex = gene.geneProducts.indexOf(product);

                    individual.genes[randomCityIndex].geneProducts[productIndex].hasBought = true;
                }
            });
        } else {
            let boughtProductIndexes: number[] = [];

            gene.geneProducts.forEach((product) => {
                if (product.hasBought) {
                    boughtProductIndexes.push(gene.geneProducts.indexOf(product));
                }
            });

            if (boughtProductIndexes.length > 0) {
                let randomBoughtProductIndex = _.shuffle(boughtProductIndexes)[0];

                gene.geneProducts[randomBoughtProductIndex].hasBought = false;
                individual.genes[_.shuffle(indexes)[0]].geneProducts[randomBoughtProductIndex].hasBought = true;
            }
        }
    });

    individual.resetParams()
}
```

Рис. 4.3.15 Реалізація мутаційної функції для GA TPP

Завершення всіх запланованих ітерацій призводить до оптимізації закупівлі. Згідно зі схемою на рис. 2.2.4, результати другого рівня передаються на перший рівень - рівень інтерфейсу користувача Згідно зі схемою на рис. 2.2.4, результати другого рівня необхідно передати на перший рівень - рівень інтерфейсу користувача. Для цього використовується EventEmitter, який передає результати до першого шару за допомогою функції emit() (в реалізації Angular це модуль з позначкою @NGModule decorator).

Процес оптимізації. На веб-сторінці з'являється позначка, яка вказує на те, що оптимізація завершена:

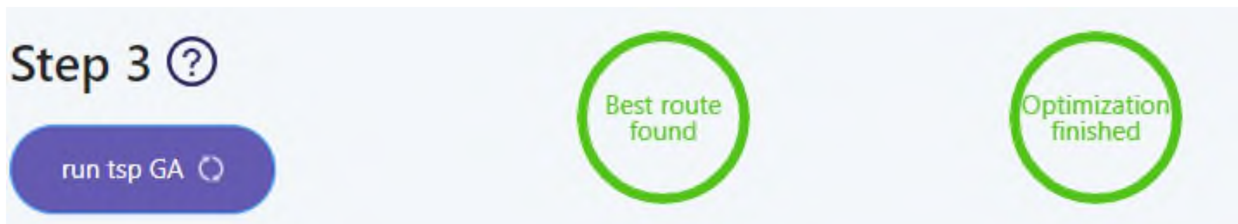


Рис. 4.3.16 Позначки про завершення 3 кроку

Натиснувши на кнопку "Прокласти маршрут до супермаркету" на кроці 4, користувач отримає візуалізацію маршруту та повну інформацію про покупку. Щоб краще уявити роботу системи, розглянемо кілька прикладів. Це пов'язано з тим, що ціна на продукти може коливатися від 25 до 50 гривень, а вартість пального - від 5 до 10 гривень, наприклад, у невеликому містечку. Таким чином, у таких випадках загальна вартість покупок перевищує вартість пального майже в десять разів. Розглянемо спочатку випадок з чотирма супермаркетами в безпосередній близькості.

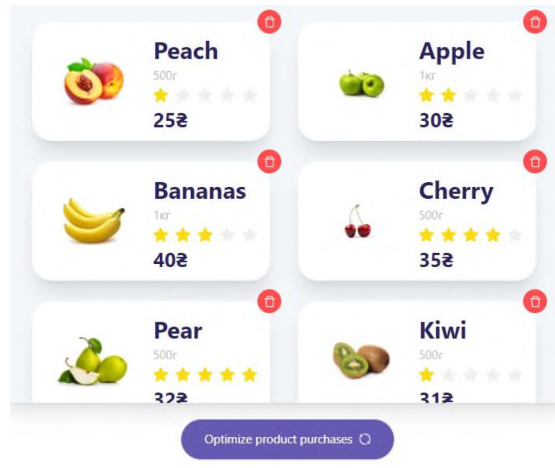


Рис. 4.3.17 Корзина з продуктами для купівлі

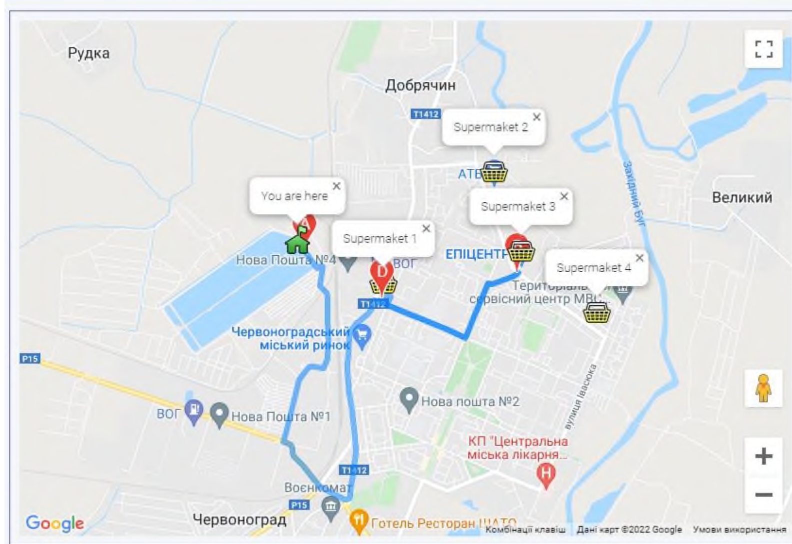


Рис. 4.3.18 Візуалізований шлях з малим розривом супермаркетів

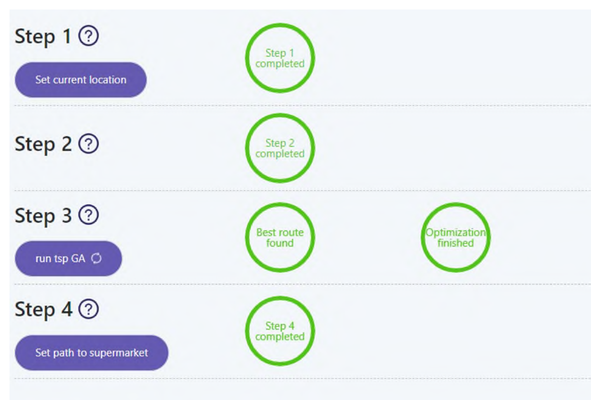


Рис. 4.3.19 Інформація про кожен крок системи

Як видно з рис. 4.3.18, закупівлі здійснюються у двох супермаркетах. Як зазначалося раніше, це пов'язано з різницею між ціною пального та загальною вартістю покупки. Іншими словами, ціна пального, яке користувач витратив, щоб дістатися до супермаркету, набагато нижча, ніж ціна придбаних товарів.

Щоб допомогти користувачам краще зрозуміти, які покупки були зроблені і скільки було витрачено, було розроблено дашборд:

Рис. 4.3.20 Інформаційна панель здійснених покупок (1)

Таким чином, загальні витрати на проїзд та покупки склали 288,482 грн., з яких 281 грн. - на покупки та 7,482 грн. - на пальне. З рисунку 4.3.18 можна зробити висновок, що продукти були придбані в першому та третьому супермаркетах. Для підтвердження цього можна відобразити дашборд (Рисунок 4.3.20). Одразу видно, що в цих двох магазинах є товари з зеленою галочкою (хрестик - не куплено), що означає, що товар було куплено.

З інформаційної панелі можна зробити висновок, що алгоритму було простіше купити товари на 2 грн (яблуко) та 1 грн (ківі) в третьому супермаркеті за нижчою ціною, а в іншому супермаркеті він витратив більше палива.

Тепер розглянемо випадок, коли супермаркети знаходяться далеко один від одного:

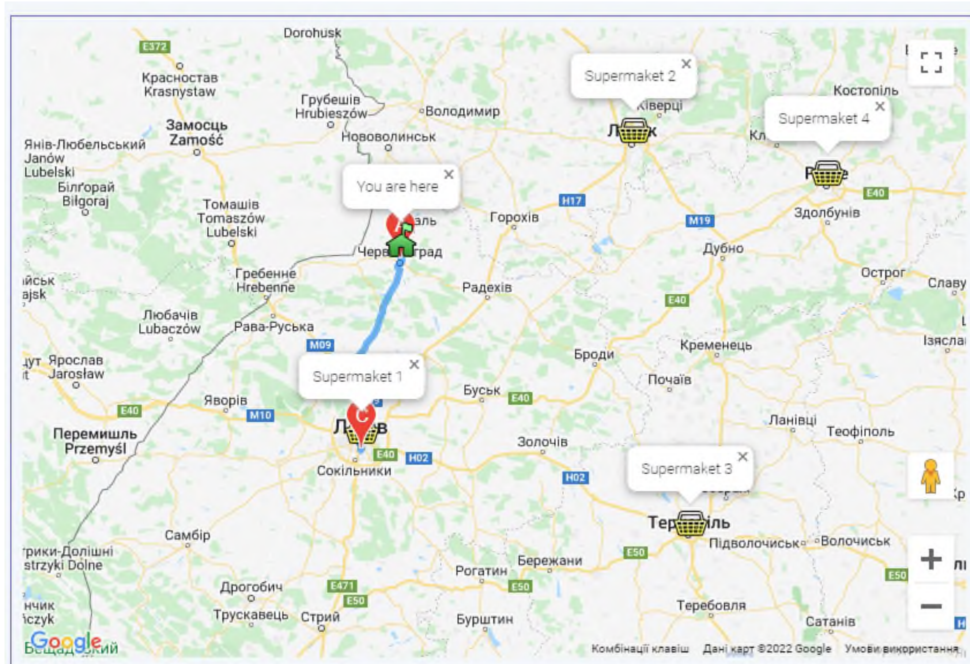


Рис. 4.3.21 Візуалізований шлях з великим розривом супермаркетів

При збільшенні відстані між супермаркетами, алгоритм обрав тільки єдиний супермаркет для покупок, переконаємось у цьому за допомогою інформаційної панелі (рис. 4.3.22).

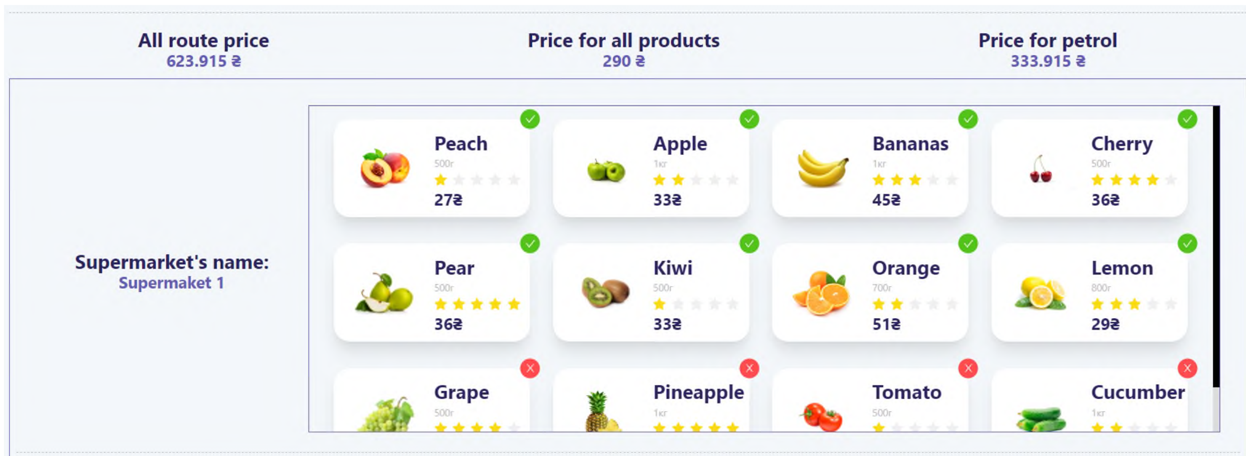


Рис. 4.3.22 Інформаційна панель здійснених покупок (2)

Алгоритм витратив 623,915 гривень на проїзд в один бік, 290 гривень на покупки та 333,915 гривень на пальне. Покупки показують, що алгоритм фактично здійснив вісім покупок в одному супермаркеті.

4.4. Керівництво користувача

Щоб скористатися системою, користувач повинен виконати наступні кроки:

1. Вибрати категорію товару, який він хоче купити.
2. Вибрати товари, які йому подобаються.
3. Переглянути кошик і натиснути на кнопку "Оптимізувати покупку".
4. Коли з'явиться сторінка карти, дотримуйтесь інструкцій з підказками на кожному кроці, позначеними іконкою зі знаком питання.
5. Натисніть кнопку "Встановити місцезнаходження" і перейдіть до вашого поточного місцезнаходження.
6. Виберіть принаймні три супермаркети.
7. Натисніть кнопку "Запустити GA" і зачекайте кілька секунд, поки не з'явиться зелений інформаційний знак.
8. Натисніть кнопку "Встановити шлях до супермаркету".

Висновки до розділу

У цьому розділі були представлені інструменти розробки, які використовувалися для розробки програмної частини проекту. Основним інструментом розробки було обрано фреймворк Angular через його переваги.

Крім того, були встановлені технічні та програмні вимоги до хорошого інтернет-з'єднання та найбільш підходящих компонентів для комп'ютера користувача.

Була розроблена робоча система. Проведено та проаналізовано результати експериментів. Створено посібник користувача для використання реалізованої системи.

РОЗДІЛ 5. РОЗРОБЛЕННЯ СТАРТАП-ПРОЕКТУ

5.1. Опис ідеї проекту

Наш стартап розробляє веб-додаток, який сприяє оптимізації процесу покупок в супермаркеті та допомагає користувачам вибрати найкращий спосіб дістатися до нього. Основна ідея полягає в поєднанні зручного маршруту до супермаркету з ефективним плануванням покупок, що дозволить зекономити час і гроші.

Визначення загальних напрямків використання потенційного товару чи послуги:

1. **Маршрут до супермаркету:** Наш додаток розраховує найшвидший та найефективніший маршрут від поточного місцезнаходження користувача до обраного супермаркету.
2. **Планування покупок:** Користувачі можуть створювати списки покупок, визначати пріоритети, і отримувати рекомендації щодо оптимального порядку витрачення часу в супермаркеті.
3. **Вигода і економія:** За допомогою нашого додатку користувачі зможуть зекономити час, уникнути непотрібних повторних проходжень по відділеннях магазину і зменшити кількість імпульсивних покупок.

5.2. Відмінність від конкурентів:

Наш проект вирізняється інтеграцією двох ключових функціональностей - планування маршруту та покупок, що робить його унікальним на ринку. Крім того, ми надаємо детальні аналітичні дані користувачам про їхні покупки, що дозволяє їм краще розуміти свої витрати та звітувати перед сім'єю чи бюджетом.

Аналіз технологічних можливостей:

- **Геолокація:** Використання GPS технології для визначення місцезнаходження користувача та розрахунку оптимального маршруту.
- **Штучний інтелект:** Використання алгоритмів машинного навчання для аналізу покупок користувачів та надання персоналізованих рекомендацій.
- **Мобільні додатки:** Розробка мобільних додатків для платформ iOS та Android для максимального охоплення аудиторії.

5.3. Аналіз ринкових можливостей:

Зростання інтересу до ефективного планування покупок: З плином часу все більше людей розуміють важливість ефективного витрачання часу в магазинах. Збільшення популярності онлайн-сервісів: Онлайн-покупки і електронні додатки для планування рутинних завдань стають все популярнішими. Зростання популярності сервісів, які допомагають економити гроші та уникати непотрібних витрат.

Розроблення стратегії ринкового впровадження:

- **Бета-тестування:** Залучення обмеженого кола користувачів для тестування та отримання зворотного зв'язку.
- **Партнерства:** Встановлення партнерських відносин з супермаркетами для інтеграції нашого додатку в їхні системи.
- **Маркетинг і реклама:** Проведення ефективних маркетингових кампаній для підвищення уваги користувачів та визначення переваг нашого додатку порівняно з конкурентами.
- **Розширення функціональності:** Постійне оновлення та розширення можливостей додатку відповідно до потреб користувачів і змін на ринку.

Висновки до розділу

Стартап проект обіцяє великий потенціал і вигоди для користувачів. Його унікальність полягає в поєднанні геолокаційних можливостей з інтелектуальним плануванням покупок, що робить процес шопінгу ефективнішим та зручнішим. Основні напрямки використання додатку включають оптимізацію часу, економію грошей і підвищення свідомості споживачів.

Технологічні можливості, такі як використання геолокації та штучного інтелекту, роблять наш продукт конкурентоздатним на ринку. Мобільні додатки для платформ iOS та Android розширюють охоплення аудиторії.

Аналіз ринкових можливостей свідчить про зростання інтересу до подібних сервісів та збільшення популярності онлайн-покупок. Розроблена стратегія ринкового впровадження, включаючи бета-тестування, партнерства та маркетингові заходи, допоможе нашому стартапу зайняти свою нішу в цьому обіцяючому сегменті.

Загалом, дана ініціатива є актуальною і може стати успішним рішенням для сучасних споживачів, що цінують ефективність, зручність і раціональне витрачання часу.

ВИСНОВКИ

В ході магістерської роботи виконано наступні завдання: створено та оптимізовано обраний користувачем шлях до супермаркету за допомогою генетичних алгоритмів та симуляції відпалу.

Після отримання маршрутів генетичний алгоритм було застосовано для розв'язання задачі мандрівного покупця з метою оптимізації покупок користувача. Для цього було самостійно реалізовано GA TPP.

Отримані покупки та шляхи були відображені на веб-сайті, побудованому з використанням фреймворку Angular. Щоб зробити оптимізацію більш наочною, користувачам була представлена інформаційна панель, яка показувала оптимізований маршрут на Картах Google і ціну кожного товару в конкретному супермаркеті, а також суму, витрачену на всю поїздку, вартість палива і всі зроблені покупки.

Користувачам також була надана система допомоги у вигляді підказок та покрокових інструкцій для процесу оптимізації.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

[1] «John Henry Holland», *Wikipedia.* 31, Грудень 2021. https://en.wikipedia.org/wiki/John_Henry_Holland (дата звернення Трав 14 2022)

[2] «Darwinism», *Wikipedia*. 09, Квітень 2021. https://en.wikipedia.org/wiki/Darwinism (дата звернення Трав 14 2022)

[3] «Travelling salesman problem», *Wikipedia*. 05, Травень 2022. https://en.wikipedia.org/wiki/Travelling_salesman_problem (дата звернення Трав 14 2022)

[4] «Traveling purchaser problem», *Wikipedia*. 14, Січень 2019. https://en.wikipedia.org/wiki/Traveling_purchaser_problem (дата звернення Трав 14 2022)

[5] Z. A. Ali, S. A. Rasheed, і N. No’man Ali, «An enhanced hybrid genetic algorithm for solving traveling salesman problem», *Indonesian Journal of Electrical Engineering and Computer Science*, вип. 18, вип. 2, с. 1035–1039, 2020, doi: 10.11591/ijeecs.v18.i2.pp1035-1039.

[6] A. A. K. Taher і S. M. Kadhim, «Improvement of genetic algorithm using artificial bee colony», *Bulletin of Electrical Engineering and Informatics*, вип. 9, вип. 5, с. 2125–2133, 2020, doi: 10.11591/eei.v9i5.2233.

[7] Y. Deng, J. Xiong, і Q. Wang, «A hybrid cellular genetic algorithm for the traveling salesman problem», *Mathematical Problems in Engineering*, вип. 2021, 2021, doi: 10.1155/2021/6697598.

[8] Y. Oonsrikaw і A. Thammano, «Modified ant colony optimization algorithm for multiple-vehicle traveling salesman problems», *International Journal of Networked and Distributed Computing*, вип. 7, вип. 1, с. 29–36, 2018, doi: 10.2991/ijndc.2018.7.1.4.

[9] M. A. H. Akhand, S. I. Ayon, S. A. Shahriyar, N. Siddique, і H. Adeli, «Discrete Spider Monkey Optimization for Travelling Salesman Problem», *Applied Soft Computing*, вип. 86, с. 105887, Січ 2020, doi: 10.1016/j.asoc.2019.105887.

[10] C. Wu і X. Fu, «An agglomerative greedy brain storm optimization algorithm for solving the TSP», *IEEE Access*, вип. 8, с. 201606–201621, 2020, doi: 10.1109/ACCESS.2020.3035899.

[11] S. Akter, N. Nahar, M. Shahadathossain, і K. Andersson, «A New Crossover Technique to Improve Genetic Algorithm and Its Application to TSP», представлена на 2nd International Conference on Electrical, Computer and Communication Engineering, ECCE 2019, 2019. doi: 10.1109/ECACE.2019.8679367.

[12] M. Abbasi, M. Rafiee, M. R. Khosravi, A. Jolfaei, V. G. Menon, і J. M. Koushyar, «An efficient parallel genetic algorithm solution for vehicle routing problem in cloud implementation of the intelligent transportation systems», *Journal of Cloud Computing*, вип. 9, вип. 1, 2020, doi: 10.1186/s13677-020-0157-4.

[13] F. Dahan, K. El Hindi, H. Mathkour, і H. Alsalman, «Dynamic flying ant colony optimization (DFACO) for solving the traveling salesman problem», *Sensors (Switzerland)*, вип. 19, вип. 8, 2019, doi: 10.3390/s19081837.

[14] J. Zhang, L. Hong, і Q. Liu, «An improved whale optimization algorithm for the traveling salesman problem», *Symmetry*, вип. 13, вип. 1, с. 1–13, 2021, doi: 10.3390/sym13010048.

ДОДАТКИ

ДОДАТОК А. ЛІСТИНГ КОДУ ПРОГРАМИ

```
export class TspGeneticAlgorithm implements GeneticAlgorithmModel {
    selection(population: TspPopulation, tournamentSize: number): TspIndividual {
        return new TspPopulation(_.sampleSize(population.individuals,
tournamentSize)).getFittestIndividual();
    }

    crossover(route = undefined, firstParent: TspIndividual, secondParent:
TspIndividual, startGene: StartGeneModel): TspIndividual {
        let genesToFill = firstParent.genes.length;
        let nullGene = new Gene('null', 0, 0, []);

        let genes = [];

        for (let i = 0; i < genesToFill; i++) {
            genes.push(nullGene);
        }

        let child = new TspIndividual(genes, startGene, 0, 0);
        this.fillFirstParentGenes(child, firstParent, Math.floor(genesToFill / 2));
        this.fillSecondParentGenes(child, secondParent);

        return child;
    }

    mutate(individual: TspIndividual, mutateRate: number): void {
        for (let i = 0; i < individual.genes.length; i++) {
            if (_.random() < mutateRate) {
                let sampledGenes = _.sampleSize(individual.genes, 2);
                individual.swap(sampledGenes[0], sampledGenes[1]);
            }
        }
    }

    evolve(
        route = undefined,
        population: TspPopulation,
        tournamentSize: number,
        mutateRate: number,
        startGene: StartGeneModel
    ): TspPopulation {
        let newGeneration = new TspPopulation([]);
        let populationSize = population.individuals.length;
        let eliteSize = Math.floor(populationSize / 2);

        // Elitism
        for (let i = 0; i < eliteSize; i++) {
            let fittest = population.getFittestIndividual();
            newGeneration.addIndividual(fittest);
```

```
            population.removeIndividual(fittest);
        }

        // Crossover
        for (let i = eliteSize; i < populationSize; i++) {
            let firstParent = this.selection(newGeneration, tournamentSize);
            let secondParent = this.selection(newGeneration, tournamentSize);
            let child = this.crossover(undefined, firstParent, secondParent,
startGene);
            newGeneration.addIndividual(child);
        }

        // Mutation
        for (let i = eliteSize; i < populationSize; i++) {
            this.mutate(newGeneration.individuals[i], mutateRate);
        }

        return newGeneration;
    }

    private fillFirstParentGenes(child: TspIndividual, parent: TspIndividual,
genesToFill: number) {
        let startIndex = _.random(0, parent.genes.length - genesToFill - 1);
        let finishIndex = startIndex + genesToFill;

        for (let i = startIndex; i < finishIndex; i++) {
            child.genes[i] = parent.genes[i];
        }
    }

    private fillSecondParentGenes(child: TspIndividual, parent: TspIndividual) {
        let j = 0;

        for (let i = 0; i < parent.genes.length; i++) {
            if (child.genes[i].name === 'null') {
                while (child.genes.includes(parent.genes[j])) {
                    j += 1;
                }

                child.genes[i] = parent.genes[j];
                j += 1;
            }
        }
    }
}
```

```
export class PriceGeneticAlgorithm implements GeneticAlgorithmModel {
    selection(population: PricePopulation, tournamentSize: number): PriceIndividual
{
        return new PricePopulation(_.sampleSize(population.individuals,
tournamentSize)).getFittestIndividual();
    }

    crossover(
        route: TspIndividual,
        firstParent: PriceIndividual,
        secondParent: PriceIndividual,
        startGene: StartGeneModel
    ): PriceIndividual {
        let child = new PriceIndividual(route.copyGenes(), startGene, 0, 0);
        let productCart = _.cloneDeep(startGene.productCart);

        let [firstParentCart, secondParentCart] =
[..._.chunk(_.shuffle(productCart), Math.floor(productCart.length / 2))];

        this.fillWithParentGenes(child, firstParent, firstParentCart);
        this.fillWithParentGenes(child, secondParent, secondParentCart);

        return child;
    }

    mutate(individual: PriceIndividual, mutateRate: number): void {
        individual.genes.forEach((gene) => {
            let indexes = [];

            indexes = _.range(individual.genes.length);
            indexes = indexes.filter((index) => index !==
individual.genes.indexOf(gene));

            if (_.random() < mutateRate) {
                let randomCityIndex = _.shuffle(indexes)[0];

                gene.geneProducts.forEach((product) => {
                    if (product.hasBought) {
                        product.hasBought = false;

                        let productIndex = gene.geneProducts.indexOf(product);

                        individual.genes[randomCityIndex].geneProducts[productIndex
].hasBought = true;
                    }
                });
            } else {
                let boughtProductIndexes: number[] = [];

                gene.geneProducts.forEach((product) => {
                    if (product.hasBought) {
                        boughtProductIndexes.push(gene.geneProducts.indexOf(product
));
```

```
                }
            });

            if (boughtProductIndexes.length > 0) {
                let randomBoughtProductIndex =
_.shuffle(boughtProductIndexes)[0];

                gene.geneProducts[randomBoughtProductIndex].hasBought = false;
                individual.genes[_.shuffle(indexes)[0]].geneProducts[randomBoug
htProductIndex].hasBought = true;
            }
        }
    });

    individual.resetParams()
}

evolve(
    route: TspIndividual,
    population: PricePopulation,
    tournamentSize: number,
    mutateRate: number,
    startGene: StartGeneModel
): PricePopulation {
    let newGeneration = new PricePopulation([]);
    let populationSize = population.individuals.length;
    let eliteSize = Math.floor(populationSize / 2);

    // Elitism
    for (let i = 0; i < eliteSize; i++) {
        let fittest = population.getFittestIndividual();
        newGeneration.addIndividual(fittest);
        population.removeIndividual(fittest);
    }

    // Crossover
    for (let i = eliteSize; i < populationSize; i++) {
        let firstParent = this.selection(newGeneration, tournamentSize);
        let secondParent = this.selection(newGeneration, tournamentSize);
        let child = this.crossover(route, firstParent, secondParent,
startGene);
        newGeneration.addIndividual(child);
    }

    // Mutation
    for (let i = eliteSize; i < populationSize; i++) {
        this.mutate(newGeneration.individuals[i], mutateRate);
    }

    return newGeneration;
}
```

```
    private fillWithParentGenes(child: PriceIndividual, parent: PriceIndividual,
parentCart: string[]) {
        parent.genes.forEach((gene) => {
            gene.geneProducts.forEach((geneProduct) => {
                if (parentCart.includes(geneProduct.productName) &&
geneProduct.hasBought) {
                    let childGene = _.find(child.genes, function (childGene) {
                        return childGene.name === gene.name;
                    });

                    let childProduct = _.find(childGene?.geneProducts, function
(childProduct) {
                        return childProduct.productName ===
geneProduct.productName;
                    });

                    childProduct!.hasBought = true;
                }
            });
        });
    }
}
```