Київський національний університет імені Тараса Шевченка
Факультет комп'ютерних наук та кібернетики
Кафедра дослідження операцій

ВИПУСКНА КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА
за спеціальністю 113 «Прикладна математика»
на тему:

Задача Комівояжера

Студентки 4 курсу
Йосипчук Анастасії Ігорівни

Науковий керівник:
Доцент, доктор фізико-математичних наук
Самойленко Ігор Валерійович

Робота заслухана на засіданні кафедри дослідження операцій та рекомендована до захисту в ЕК, протокол №9 від 23 травня 2023 р.

Завідувач кафедри ДО проф. Іксанов О.М.

Київ – 2023

ЗМІСТ

ВСТУП 3

Розділ 1. 4

1. Історія виникнення задачі та розвитку рішень 4

2.Постановка задачі 6

3. NP P класи задач 7

4.Задачі, що зводяться до Комівояжера 8

4.1 Задача Гамільтона 9

4.2 Задача про переналадку станків 10

5. Методи розв’язування задачі Комівояжера 11

5.1. Алгоритм повного перебору 11

5.2. Алгоритм гілок та меж 11

5.3. Алгоритм пошуку найближчого сусіда 13

5.4. Генетичний алгоритм 14

5.5 k-opt 15

5.6. Мурашиний алгоритм 15

6. Критерії оцінки якості розв’язку 17

Розділ 2. 19

1. Формулювання задачі 19

2. Характеристика технічних інструментів 19

3. Реалізація програми 19

3.1 Реалізація Алгоритмів 20

3.2 Візуалізація 26

4. Результати роботи програми 27

ВИСНОВОК 34

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 35

ДОДАТОК А. Координати місць 36

ВСТУП

Задача комівояжера є однією з найважливіших оптимізаційних задач, , розв'язання якої дозволяє знайти найкоротший шлях, який проходить через певну кількість міст або точок, з урахуванням визначених обмежень.

Актуальність дослідження задачі комівояжера полягає у тому, що вона є ключовою для розв'язання проблем оптимізації маршрутів і ресурсів у різних галузях, а в сучасному світі, де швидкість та ефективність стали визначальними факторами успіху, оптимізація є критичним завданням. Задача виникла з практичної потреби знайти оптимальний маршрут для торгового представника, який повинен відвідати кілька міст, повернувшись у вихідне місце та зараз має широкий спектр застосувань у різних сферах, зокрема телекомунікаціях, проектуванні мікросхем, комп’ютерній графіці, виробничому плануванні, проте основною галуззю, в якій задача комівояжера є її невід’ємною частиною, лишається логістика . У цій сфері задача комівояжера використовується для планування маршрутів, що допомагає знизити витрати на паливо, скоротити час доставки, підвищити ефективність використання транспортних ресурсів та покращити досвід клієнтів. Наведені приклади є лише частиною переліку, де задача комівояжера має практичне застосування, тому вона продовжує викликати інтерес дослідників і практиків, які працюють над оптимізацією маршрутів та послідовностей в різних контекстах.

Метою дипломної роботи є дослідження і розв'язання задачі комівояжера з метою знайти оптимальний маршрут та реалізувати для цього ефективний алгоритм з мінімальними обчислювальними витратами, використовуючи стандартний технічний пристрій.

Для досягнення цієї мети поставлені наступні задачі:

1. Теоретичне вивчення задачі, огляд і аналіз існуючих методів розв'язання задачі комівояжера.

3. Формулювання задачі для практичного дослідження.

4. Реалізація методів розв’язання, їх аналіз та порівняння в застосуванні, побудова оптимального маршруту для обраної задачі.

РОЗДІЛ 1.ТЕОРЕТИЧНА ЧАСТИНА

1 Історія виникнення задачі та розвитку рішень

Задача комівояжера має свою значну історію розвитку. Її походження можна відслідкувати ще з XVIII століття, коли Леонард Ейлер досліджував задачу про хід коня, а XIX столітті Вільям Гамільтон поставив задачу про ікосіан - головоломку, де треба відвідати кожну вершину додекаедра рівно один раз (яку ми розглянемо в наступному розділі)

Вже у 1832 році була видана книга "Комівояжер - як йому поводитися та що він має робити, щоб доставляти товари та мати успіх у своїх справах" авторства невідомого комівояжера, звідки і пішла назва. У цій книзі була описана задача, але для її вирішення не застосовувалися обчислення, у книзі були лише наведені приклади маршрутів для окремих регіонів Німеччини та Швейцарії.

Упродовж XX століття було проведено багато досліджень з задачі комівояжера з теоретичної та практичної точок зору, у 1972 році Річард Карп довів NP-повноту задачі пошуку гамільтонових шляхів, що випливає з поліноміального зв'язування, чим підтвердив NP-важкість задачі комівояжера, і на основі цих властивостей було надано теоретичне обґрунтування складності пошуку рішень задачі на практиці. Також розроблені нові алгоритми оптимізації, такі як метод відсічення, гілок та меж, і розроблено стандартизовані набори прикладів задачі комівояжера для порівняння рішень.

У 1990-х роках була розроблена програма під назвою Concorde, яка була створена Девідом Аплгейтом, Робертом Биксбі, Вашеком Хваталом та Уільямом Куком, що дозволяє знаходити оптимальні маршрути для великих наборів міст і має високу точність розв'язання, і лишається важливим інструментом дотепер. У той же час, Герхард Райнельт створив TSPLIB - набір тестових прикладів для задачі комівояжера різної складності,

яку Concorde використовує для оцінки та порівняння своїх результатів,одним з її досягнень стало знаходження оптимального маршруту для набору міст з 85 900 вершинами в 2006 році.

Сучасні вчені працюють над вдосконаленням алгоритмів розв'язання задачі комівояжера і виявленням нових способів його використання для вирішення реальних проблем у різних галузях.

2 Постановка задачі Комівояжера

Дано скінчену множину місць, які мають бути відвідані комівояжером. Кожне місце представлено точкою на площині або в просторі, і кожній парі місць присвоєна відстань або метрика, що визначає відстань між ними. Задача полягає в знаходженні найкоротшого шляху, який проходить через кожне місце рівно один раз і повертається в початкову точку.

Якщо змінити постановку задачі і не вимагати повернення до початкового пункту, то отримаємо *незамкнуту* задачу комівояжера.

Також, задача комівояжера поділяється на симетричну та асиметричну. Симетричною ми її називаємо, якщо відстані між парами місць однакові, тобто те з якої в яку точку ми рухаємося не має значення, асиметричною - якщо відстані між парами місць не є рівними.

Позначаємо міста числами від 1 до n, а вартість (відстань) від міста i до міста j як c_{ij} , x_{ij} приймають значення 1 або 0. Якщо x_{ij} = 1, це означає, що маршрут проходить від міста i до міста j, в іншому випадку - x_{ij}=0. Цільовою функцією цієї задачі є мінімізація всього маршруту комівояжера:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} \to min\,,$$

з умовами:

$$\sum_{j=1}^{n} x_{ij} = 1 \ (i = 1, n; i \neq j),$$

обмеження одноразовим в`їздом та

$$\sum_{i=1}^{n} x_{ij} = 1 \ (i = 1, n; i \neq j),$$

обмеження одноразовим виїздом.

Хоч і постановка здається простою і нам спадає на думку, що ми можемо просто перебрати всі варіанти маршрутів і таким чином знайти оптимальний розв’язок, такий підхід може працювати лише при дуже невеликій кількості даних. Розглянемо детальніше чому.

Якщо комівояжер вирушає з першого міста, він може обрати одне з (n-1) міст, потім одне з (n-2) залишених міст, потім одне з (n-3) і так далі , аж до моменту поки не залишиться одне місто. Отже, загальна кількість можливих маршрутів становить (n-1)!

Тобто для повного перебору можливих варіантів, потрібто проаналізувати

$$(n-1)(n-2)\ldots2\cdot1=(n-1)!$$

кількість комбінацій, що при значній кількості точок стає неможливим навіть для дуже потужних обчислювальних машин.

3 NP P задачі

NP (Non-deterministic Polynomial time) та P (Polynomial time) є двома різними класами задач складності в теорії обчислень, та їх поняття виникли при розв'язуванні великої кількості оптимізаційних задач у математиці, коли вчені часто стикалися з тим, що ці задачі були складними для розв'язання, тобто не були відомі ефективні алгоритми з поліноміальною складністю для їх рішення, в результаті почався пошук альтернативних методів розв'язання, такі як наближені алгоритми, які можуть знайти прийнятні розв'язки, хоча не

гарантують знаходження оптимального рішення. Отже:

1. P - це клас задач, для яких існують ефективні детерміновані алгоритми, що здатні знайти розв'язок за поліноміальний час відносно розміру вхідних даних. Іншими словами, задачі класу P можна розв'язати швидко, використовуючи алгоритми з обмеженим часом виконання, що збільшується поліноміально з розміром вхідних даних.

2. NP - клас задач, для яких можна ефективно перевірити правильність розв'язку, але знаходження самого розв'язку може вимагати значно більше часу. Недетерміновані алгоритми, які використовують неоднозначність та можливість одночасної обробки кількох варіантів, можуть знайти розв'язок NP-проблеми за поліноміальний час, але немає гарантії, що цей розв'язок є правильним.

Задача Комівояжера відноситься саме до NP класу задач, тобто неї не існує відомого поліноміальних алгоритму вирішення.

4 Задачі математичне формулювання яких співпадає з задачею Комівояжера

Різновидів задач, математичне формулювання яких співпадає з задачею комівояжера є безліч, зокрема задача Гамільтона, Ейлера (з яких і почалася історія заді комівояжера), задача збору по тривозі (Alarm Collection Problem), проводки і енергопостачання (Wiring and Power Supply Problem), оптимізації програм (Program Optimization Problem), а також ми можемо самостійно сформулювати безліч інших.

Розглянемо детальніше дві з них: задачу Гамільтона, історично пов’язаної з задачею комівояжера, а також задачу про переналадку станків, аби показати, як рішення на перший погляд не схожої задачі зводиться до комівояжера.

4.1 Задача Гамільтона

У 1857 році ірландський математик Гамільтон запропонував гру під назвою "подорож по додекаедру". Вона викликала інтерес серед математиків, і в результаті задача відіграла значну роль у вивченні графів та пошуку оптимальних шляхів. Гра полягала в обході всіх вершин правильного додекаедра з певними умовами.

Додекаедр - це особливий тип многогранника, який має 12 правильних п'ятикутників як грані. У нього загалом 20 вершин і 30 ребер. Кожне ребро з'єднує дві вершини, а грані складаються з п'ятикутників

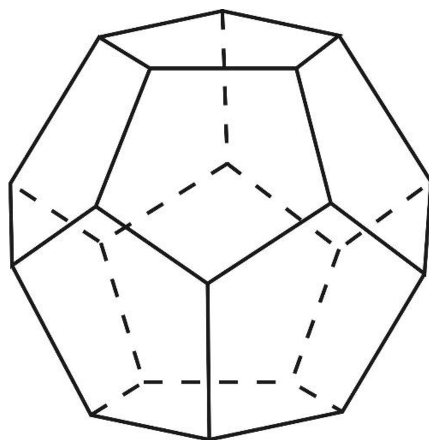

Рис.1.4.1.1

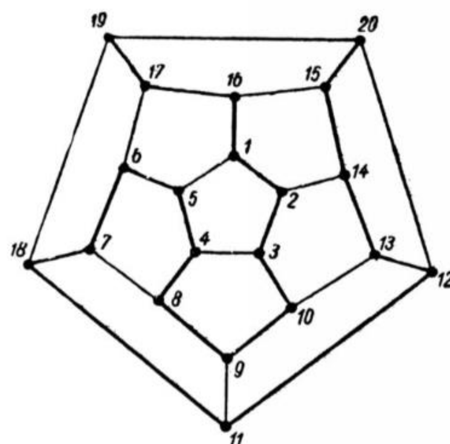

Рис.1.4.1.2

В кожній вершині додекаедра збігаються три ребра. Уявимо, що наш додекаедр зроблений з дроту і його можна розтягнути без розривів. Отримаємо зображений на малюнку граф. Необхідно побувати у всіх вершинах рівно один раз та повернутись у вихідну, при цьому рухатись можна тільки виключно ребрами додекаедра. Кожен такий шлях, якщо він існує і задовольняє описані умови, називається циклом, але не просто звичайними, а гамільтоновим циклом, або також гамільтоновим контуром. Для того щоб звести цю задачу до задачі комівояжера, уявимо, що кожна

вершина означає столицю, відстань між якими дорівнює 1, якщо вони знаходяться на одному ребрі, та нескінченності, якщо вони знаходяться на різних. Отже, отримаємо матрицю 20×20 взаємних відстаней між нашими 20 містами. Якщо гамільтоновий шлях існує, то його довжина дорівнює 20, і якщо такого шляху немає, то довжина маршруту буде рівне нескінченності. Розв’язавши задачу комівояжера з використанням заданої матриці у якості вхідних даних, ми отримаємо відповідь чи варіант обходу існує, а також один з варіантів маршруту.

4.2 Задача про переналадку станків

Припустимо, що є набір деталей, які потрібно обробити на автоматичній лінії, а час налаштування лінії для обробки i-ї деталі є значно великим і залежить від того, яка деталь оброблялась перед цим.

Питання полягає у тому, в якому порядку слід обробляти деталі, щоб зменшити витрати часу на налаштування, і мета задачі полягає в мінімізації загального часу переналадки станків і максимізації продуктивності виробничого процесу.У цьому випадку нас не цікавить час обробки деталей, оскільки цей час є постійною величиною у загальному часі виконання замовлення незалежно від порядку обробки деталей.

Бачимо, що сформульована задача є чистим випадком задачі про комівояжера для n міст. Нам потрібно, щоб лінія відвідувала n станів, причому сума часів переходу з одного стану в інший була мінімальною.

У цій задачі деталі відповідають містам, а часи налаштування відповідають відстаням між містами. Метою є знайти оптимальну послідовність міст (деталей), щоб мінімізувати загальну пройдену відстань (час налаштування) на лінії.

Розв'язок задачі про комівояжера можна застосувати для визначення оптимального порядку обробки деталей, тим самим мінімізуючи час

налаштування, покращуючи загальну ефективність автоматичної лінії та знижуючи затрати.

5. Алгоритми вирішення задачі Комівояжера

Алгоритми вирішення задачі комівояжера поділяються ті які дають точний та наближений результат. Розглянемо детальніше деякі з них, а саме найбільш розповсюдженні у використанні методи.

5.1 Алгоритм повного перебору

Найпростішим способом розв'язання задачі комівояжера є перебір всіх перестановок, тобто упорядкованих комбінацій і вибір найбільш вигідної з них.

Метод є загальним методом вирішення проблем, який полягає у генерації списку всіх можливих кандидатів для розв'язку і перевірці кожного з них. У випадку вирішення задачі Комівояжера він передбачає систематичне перерахування всіх можливих варіантів маршруту для та перевірку, чи вони відповідають вимозі, тобто значенню найоптимальнішого маршруту. Метод займає багато часу при великій кількості варіантів та є тоді неефективним, час виконання для цього підходу залежить поліноміального множника O(n!), тобто факторіалу кількості міст. Тому це рішення стає непрактичним навіть для всього 20 міст. З очевидних плюсів методу (як і будь якого точного) - якщо всі можливі рішення перераховані, то можна гарантувати, що підхід знайде правильний варіант.

5.2. Алгоритм гілок та меж

Алгоритм гілок та меж (Branch and Bound) є популярним методом для

розв'язування комбінаторних оптимізаційних задач. Основна ідея полягає в тому, щоб систематично перебрати всі можливі варіанти розв'язку задачі, зберігаючи при цьому найкращі поточні результати та використовуючи верхні та нижні межі для відсіювання неоптимальних гілок пошуку.

Якщо вважати міста вершинами графа, а шляхи (i, j) - його дугами, то знаходження мінімального шляху, при тому що комівояжер проходить один раз через кожне місто, і повертається назад назад, можна розглядати як знаходження на графі гамільтонового контуру мінімальної довжини. У даному алгоритмі ключовими елементами є гілкування дерева та встановлення меж. Під час обчислення розв'язку для підзадачі, встановлюються верхні і нижні межі. Для поточного вузла в дереві ми обчислюємо межу для найкращого можливого розв'язку, який ми можемо отримати, спускаючись по цьому вузлу. Якщо межа для найкращого можливого розв'язку виявляється гіршою, ніж поточний найкращий розв'язок (найкращий, знайдений до цього моменту), то ми ігноруємо піддерево, що має корінь в цьому вузлі.

Алгоритм:

1. Проводимо операцію редукції матриці відстаней по рядках і стовпцях, знаходимо нижню межу як $H = \sum_i d_i + \sum_j d_j$

2. У кожній клітині, в яких 0, знаходимо коефіцієнт додаючи найменший елемент стрічки та найменшого елемента j стовпчика (за виключенням значення елемента для якого шукаємо коефіцієнт)

3. Вибираємо ребро розгалуження *(i, j)* за максимальною величиною суми коефіцієнта . Потім виключаємо його з безлічі шляхом заміни елемента матриці на ∞.

4. Проводимо операцію редукції матриці відстаней по рядках і визначаємо нижню межу цієї підмножини маршрутів *H (i, j).*

5. Включаємо дугу в маршрут шляхом виключення i-го рядка і *j-то* стовпчика з матриці і заміни симетричного елемента матриці *на* ∞ для

запобігання утворенню негамільтонового циклу.

6. Визначаємо нижню границю підмножини

7.Порівнюємо нижні межі ,та що має менше значення нижньої межі йде на розгалуження.

Алгоритм повторюємо до того моменту, поки порядок матриці більший за 2, та визначаємо гамільтоновий цикл при отриманні матриці 2x2.

До плюсів даного підходу зараховується те, що він дає гарантований результат. Проте, хоч він працює швидше, ніж метод повного перебору, зарахунок застосування верхніх та нижніх меж дозволяє відсіяти гілки пошуку, які не можуть призвести до оптимального розв'язку, залежно від розміру задачі, кількості обмежень та їх складності, алгоритм теж вимагати значних обчислювальних ресурсів

5.3 Алгоритм пошуку найближчого сусіда.

Алгоритм пошуку найближчого сусіда є наближеним методом для розв'язання задачі комівояжера. Він відноситься до категорії жадібних алгоритмів. Загальний підхід жадібного алгоритму полягає в послідовному виборі найкращого доступного варіанту на кожному кроці. На кожному кроці алгоритму ми додаємо нове ребро до вже знайденої частини маршруту. Алгоритм завершує свою роботу, якщо знайдений розв'язок і не намагається його покращити. Іншими словами, на кожному кроці алгоритм обирає найближче доступне місто і додає його до маршруту, продовжуючи цей процес до отримання повного маршруту. Метод працює швидко та простий в реалізації, проте він приймає локальні рішення на кожному кроці, не оцінюючи загальну структуру маршруту, що веде до відповідних результатів при отриманні побудованого маршруту. Враховуючи плюси та мінуси методу найближчого сусіда, він є привабливим варіантом для простих задач комівояжера, особливо при обмежених ресурсах обчислювальної потужності та часу та коли нам не принципово знайти максимально наближений до

оптимального результат.

5.4 Генетичний алгоритм

Генетичний алгоритм є еволюційним алгоритмом, який моделює процес природного відбору для пошуку оптимального розв'язку задачі та використовує ідеї генетики та природної еволюції для генерації та вдосконалення розв'язків.

Опис кроків генетичного алгоритму для задачі комівояжера:

1.Створення початкової популяції, де кожна "хромосома" представляє можливий маршрут . Початкова популяція генерується випадковим чином або за допомогою інших евристичних методів.

2. Обчислення пристосованості кожного маршруту в популяції. У випадку задачі комівояжера, пристосованість вимірюється як загальна відстань між містами у маршруті.

3. Вибір кращих особин (маршрутів) для формування наступного покоління. Чим краще пристосованість, тим вищі шанси на відбір.

4. Створення нащадків шляхом комбінування генетичного матеріалу (маршрутів) вибраних батьків. Це може включати випадковий вибір точки розриву і обмін генетичним матеріалом.

5. Введення малої ймовірності мутації в генетичний матеріал нащадків. Це може включати зміну порядку міст або інші дрібні зміни в маршруті.

6. Формування наступного покоління, яке замінює попереднє і складається з вибраних батьків і їх нащадків.

7. Кроки 2-6 повторюються протягом кількох поколінь, або до досягнення критерію зупинки.

8. Вибір найкращого маршруту з останньої популяції як наближеного оптимального розв'язку задачі комівояжера.

Генетичний алгоритм має велику обчислювальну складність для

великих просторів розв'язків, він вимагає налаштування декількох параметрів, таких як розмір популяції, ймовірність схрещування та мутації, кількість ітерацій, неправильне налаштування яких може привести до погіршення результату. Проте він може працювати з задачами, які мають велику просторову складність, використовувати паралельну обробку, а також забезпечує глобальний огляд простору розв'язків.

5.5 k-opt

Алгоритм k-Opt вибирає k ребра в маршруті та розглядає всі можливі способи їх обміну. Для кожного можливого обміну перевіряється, чи призводить він до скорочення відстані. Якщо так, то вони здійснюють обмін. Процес продовжується до тих пір, поки не буде досягнуто найкращого можливого маршруту.

Якщо виконується критерій зупинки (наприклад, певна кількість ітерацій без покращень), алгоритм завершується, і найкращий знайдений маршрут повертається як результат.

Цей алгоритм є простим та ефективними способом покращення розв'язків задачі комівояжера шляхом локальних змін маршруту. Він може бути використаний як самостійний метод, а також застосовуватися в поєднанні з іншими для покращення вже побудованого маршруту.

5.6 Мурашиний алгоритм

Мурашиний алгоритм, також відомий як алгоритм оптимізації мурашиної колонії , є ефективним поліноміальним алгоритмом, який використовується для розв'язання задачі комівояжера та подібних завдань пошуку оптимальних маршрутів на графах. Цей алгоритм імітує поведінку мурашиних колоній, які в природі використовують феромони для комунікації та визначення найкоротшого шляху до їжі.

Мурашиний алгоритм має перевагу у паралельному виконанні, це дозволяє розв'язувати задачу комівояжера паралельно, використовуючи кілька мурашиних колоній або процесорів, що призводить до збільшення швидкості вирішення задачі та глобальному пошуку оптимального рішення. Однак, він також залежить від правильної настройки параметрів та може бути чутливим до початкового стану мурашиних колоній.

Опис алгоритму:

1. Встановлюємо параметри алгоритму, такі як кількість мурах, розмір матриці феромонів, початкові рівні феромонів , швидкість випаровування феромонів і т.д.

2.Випадково розміщуємо мурах на початкових позиціях.Кожна мураха рухається по маршруту, вибираючи наступне місто на основі ймовірностей, які залежать від феромонів і відстаней.

3.Ймовірність вибору кожного міста обчислюється за допомогою функції вибору, яка поєднує значення феромонів $\tau_{ij}(t)$ та інформацію про відстань (η_{ij}):

$$P_{ij,k}(t) = \frac{\left[\tau_{ij}(t)\right]^{\alpha} \cdot \left[\eta_{ij}\right]^{\beta}}{\sum_{l \in i,k}[\tau_{il}(t)]^{\alpha} \cdot [\eta_{il}]^{\beta}},$$

α – параметр для контролю впливу на вибір феромонів, що задає ваги слідів феромону на ребрах, α = 0 мураха обирає найближчу вершину до тої, в якій він знаходиться, β – параметр для контролю впливу, що враховує ваги самих ребер (відстаней між містами),якщо β = 0, тоді спрацьовує лише вплив феромону ,що приводить до швидкого виродження маршрутів до одного субоптимального рішення, Σ - сума по всім доступним містам. Мурахи продовжують переміщуватись до тих пір, поки не відвідають усі міста.

4.Після проходження маршруту кожної мурахи підраховується кількість феромону, що змінює існуючий розподіл на графі. Кількість феромону, яка додається на кожному пройденому ребру, що належить

маршруту, обчислюється формулою:

$$\Delta\tau_{ij}(t) = \frac{Q}{L_k(t)}$$

де $\Delta\tau_{ij}(t)$- збільшення феромону на ребрі між містами , Q - параметр, значення якого обирають одного порядку з довжиною оптимального маршруту, L - довжина маршруту мурахи.

Далі за формулою змінюємо кількість феромону на ребрах:

$$\tau_{ij}\ (t\ +\ 1)\ =\ \tau_{ij}\ +\ \text{c}\ \cdot \sum_{k=1}^{N_{ij}} \Delta\tau_{ij}(t)$$

c – коефіцієнт випаровування феромону, $c \in [0, 1]$; i і j - вершини, що з'єднуються ребрами, які відвідала k-та мураха, N_{ij} – загальна кількість мурах, що відвідали ребро.

5.Знову запускаємо колонію мурах.

На кожній ітерації мурахи здатні покращувати свій маршрут шляхом вибору оптимальних міст і оновлення феромонів. Після закінчення алгоритму, найкращий знайдений маршрут повертається як результат роботи мурашиного алгоритму для задачі комівояжера.

6 Критерії оцінки якості розв'язку

Для оцінки якості роботи алгоритму при вирішенні задачі комівояжера використовуються наступні критерії:

Загальна довжина шляху: Цей критерій вимірює загальну довжину шляху, який пройшов комівояжер. Метою є знайти найкоротший можливий шлях, щоб відвідати всі міста. Чим менша загальна довжина шляху, тим краще розв'язок.

Час виконання: Цей критерій оцінює час, необхідний для знаходження розв'язку задачі комівояжера. Швидкість виконання алгоритму є важливим фактором, особливо при роботі з великими наборами даних. Кращі

алгоритми повинні забезпечувати ефективну роботу в обмежені терміни.

Рівень оптимальності: Цей критерій визначає, наскільки близько розв'язок до оптимального. Для невеликих наборів даних, можливо, відомий точний оптимальний шлях, і розв'язок порівнюють з ним. Для великих наборів даних, де точний розв'язок неможливо знайти, використовуються нижні межі або статистичні методи для оцінки оптимальності розв'язку.

Стійкість розв'язку: Цей критерій оцінює стійкість розв'язку до змін. Якщо малі зміни вхідних даних призводять до суттєвих змін у розв'язку, то він вважається менш стійким. Стійкий розв'язок має меншу схильність до впливу шуму або дискретних змін у вхідних даних.

Ресурсоємність: Ресурсоємність відноситься до використання ресурсів, таких як пам'ять, обчислювальна потужність і час виконання, для знаходження розв'язку.

РОЗДІЛ 2. ПРАКТИЧНА ЧАСТИНА

1 Формулювання задачі

Студент приїхав в Львів, щоб доповнити свою роботу по культурології фотографіями визначних місць міста. Він обрав взяти в аренду велосипед для пересування містом , тому що вирішив що хоче бути мобільним в побудові машрутів. Його ціль – сфотографувати якомога більше цікавих йому місць, які він наперед зазначив на карті, та витратити на це якомога менше коштів (Вартість проїзду на велосипеді пропорційна до відстані між точками). Він починає свій маршрут від Оперного театру, біля якого знаходиться місце, де він взяв велосипед, та закінчує його там же.
Координати місць на карті знаходяться в додатку А. (60 точок)

2 Характеристика технічних інструментів

Для реалізації програмного коду використовується Python 3.11, середовище – PyCharm, Community Edition 2022.2.3. Процесор - AMD Ryzen 7 3700U with Radeon Vega Mobile Gfx , 2.30 GHz, тип системи - 64-розрядна операційна система, процесор на базі архітектури x64.

3 Реалізація

Для розв’язання задач мною було виділено 3 алгоритми, а саме:

1) Алгоритм повного перебору

2) Алгоритм пошуку найближчого сусіда

3)Генетичний алгоритм

4) k-opt. Використовуємо для покращення кінцевого результату.

Тестові дані задані у координатній формі, напишемо програму, яка буде обробляти координати точок на карті та перетворювати їх в матрицю відстаней. Для цього використаємо бібліотеку Geopy, яка надає засоби для

роботи з географічними даними, зокрема для обчислення відстаней між точками на земній поверхні.

```
from geopy.distance import geodesic

# Координати точок
coordinates = [...]

# Функція для обчислення відстані між двома точками з
округленням до метрів
def calculate_distance(coord1, coord2):
    distance = geodesic(coord1, coord2).meters
    return round(distance)


# Створення матриці відстаней
distance_matrix = []
for i in range(len(coordinates)):
    distances = []
    for j in range(len(coordinates)):
        distance = calculate_distance(coordinates[i],
coordinates[j])
        distances.append(distance)
    distance_matrix.append(distances)

# Виведення матриці відстаней
print(distance_matrix)
```

3.1 Реалізація алгоритмів

1) Агоритм повного перебору

```
import itertools


def swaping(k, i, j):
    k[i], k[j] = k[j], k[i]


def distance_checking(k, distance):
    dist = distance[0][k[0]]
    for i in range(1, len(k)):
        dist += distance[k[i - 1]][k[i]]
    dist += distance[k[len(k) - 1]][0]
```

```
    return dist
def tsp_brute_force(distance, places):
    n = len(places)
    k = list(range(n))
    arr = [0] * n

    minimal_distance = distance_checking(k, distance)

    all_routes = []  # Зберігатиме всі можливі маршрути
    all_distances = []  # Зберігатиме відстані для кожного
маршруту

    permutations_count = 0
    for p in itertools.permutations(range(1, n)):
        k[1:] = p
        new_distance = distance_checking(k, distance)
        all_routes.append(k.copy())
        all_distances.append(new_distance)
        if new_distance < minimal_distance:
            minimal_distance = new_distance
        permutations_count += 1

    return  all_routes, all_distances,permutations_count

def find_shortest_route(all_routes, all_distances):
    shortest_index = min(range(len(all_distances)), key=lambda
i: all_distances[i])
    shortest_route = all_routes[shortest_index]
    shortest_distance = all_distances[shortest_index]
    return shortest_route, shortest_distance

distance_m = [] # матриця відстаней

places = [] # список місць

all_routes, all_distances, permutations_count =
tsp_brute_force(distance_m, places)


shortest_route, shortest_distance =
find_shortest_route(all_routes, all_distances)
print("\nShortest Route:", [places[i] for i in shortest_route])
print("Shortest Distance:", shortest_distance)
```

2) Алгоритм пошуку найближчого сусіда

```
distance_m = [] # матриця відстаней

places = [] # список місць
```

```
n = len(distance_m)
visited = [False] * n  # Відстежуємо відвідані міста
route = []  # Шлях
current_place = 0  # Починаємо з 1 точки

for _ in range(n):
    # Додаємо місце
    route.append(current_place)
    visited[current_place] = True

    # Знаходимо
    next_place = None
    min_distance = float('inf')
    for i, distance in enumerate(distance_m[current_place]):
        if not visited[i] and distance < min_distance:
            min_distance = distance
            next_place = i

    # Переходимо до наступного
    current_place = next_place

# Adding the starting city at the end to make a full round trip
route.append(0)

optimal_route = [places[route[i]] for i in range(len(route) -
1)]
optimal_route.append(places[route[-1]])

end_time = time.time()
execution_time = end_time - start_time

print("Optimal route:", optimal_route)
print(route)

total_distance = sum(distance_m[route[i]][route[i + 1]] for i in
range(len(route) - 1))
print("Length of the optimal route:", total_distance)
```

3) Генетичний алгоритм

```
import random
import math


# Вхідні дані
distance_m = [] # матриця відстаней

places = [] # список місць
```

```
# Кількість місць
num_places = len(places)

# Генерація початкової популяції
def generate_initial_population(population_size):
    population = []
    for _ in range(population_size):
        chromosome = random.sample(range(1, num_places),
num_places - 1)
        chromosome.insert(0, 0)  # Початкове значення завжди 0
        population.append(chromosome)
    return population

# Обчислення відстані для маршруту
def calculate_distance(chromosome):
    total_distance = 0
    for i in range(num_places - 1):
        placeA = chromosome[i]
        placeB = chromosome[i + 1]
        total_distance += distance_m[placeA][placeB]
    # Додати відстань від останнього місця до першого
    total_distance += distance_m[chromosome[-1]][chromosome[0]]
    return total_distance

# Відбір найкращих особин за фітнес-функцією
def selection(population, num_parents):
    sorted_population = sorted(population, key=lambda x:
calculate_distance(x))
    parents = sorted_population[:num_parents]
    return parents

# Схрещування двох батьків для створення нащадка
def crossover(parent1, parent2):
    # Вибір випадкової точки розриву
    crossover_point = random.randint(1, num_places - 1)  #
Перший елемент не змінюємо

    # Створення нащадка шляхом комбінування частин батьків
    child = parent1[:crossover_point]
    for place in parent2:
        if place not in child:
            child.append(place)
    return child

# Мутація нащадка
def mutate(chromosome, mutation_rate):
    if random.random() < mutation_rate:
        # Вибір двох випадкових позицій для обміну місцями
        mutation_points = random.sample(range(1, num_places), 2)
# Перший елемент не змінюємо
        chromosome[mutation_points[0]],
chromosome[mutation_points[1]] = chromosome[mutation_points[1]],
```

```
chromosome[mutation_points[0]]
    return chromosome

def genetic_algorithm(population_size, num_generations,
num_parents, mutation_rate):
    # Генерація початкової популяції
    population = generate_initial_population(population_size)
    # Змінні для збереження кількості повторних поколінь і
попереднього найкращого відстані
    repeated_generations = 0
    previous_best_distance = math.inf

    for generation in range(num_generations):
        # Відбір найкращих особин
        parents = selection(population, num_parents)

        # Створення нової популяції
        new_population = parents.copy()

        while len(new_population) < population_size:
            # Вибір двох батьків для схрещування
            parent1, parent2 = random.sample(parents, 2)

            # Схрещування
            child = crossover(parent1, parent2)

            # Мутація
            child = mutate(child, mutation_rate)

            # Додавання нащадка до нової популяції
            new_population.append(child)

        # Оновлення популяції
        population = new_population

        # Вивід найкращого результату в поточному поколінні
        best_chromosome = min(population, key=lambda x:
calculate_distance(x))
        best_distance = calculate_distance(best_chromosome)

        # Перевірка на повторне покоління з однаковим найкращим
значенням
        if best_distance == previous_best_distance:
            repeated_generations += 1
        else:
            repeated_generations = 0

        # Перевірка на зупинку алгоритму
        if repeated_generations > 30:
            print("Algorithm stopped due to repeated generations
with the same best distance.")
            break
```

```
        # Оновлення попереднього найкращого значення
        previous_best_distance = best_distance
    # Знаходження найкращого розв'язку
    best_chromosome = min(population, key=lambda x:
calculate_distance(x))
    best_distance = calculate_distance(best_chromosome)

    return best_chromosome, best_distance

# підбір параметрів ,виклик генетичного алгоритму
population_size = 2000
num_generations = 400
num_parents = 2
mutation_rate = 0.1


best_chromosome, best_distance =
genetic_algorithm(population_size, num_generations, num_parents,
mutation_rate)


# Виведення результатів
print("Best Distance:", best_distance)
optimal_route = "[" + ", ".join([places[city_index] for
city_index in best_chromosome]) + "]"
print("Best Route:")
print(optimal_route)
```

4) Алгоритм k-opt

```
def four_opt(route, distance_matrix):
    improved = True
    best_distance = calculate_distance(route, distance_matrix)

    while improved:
        improved = False
        for i in range(1, len(route) - 3):
            for j in range(i + 1, len(route) - 2):
                for k in range(j + 1, len(route) - 1):
                    for l in range(k + 1, len(route)):
                        new_route = perform_four_opt_swap(route,
i, j, k, l)
                        new_distance =
calculate_distance(new_route, distance_matrix)

                        if new_distance < best_distance:
                            route = new_route
                            best_distance = new_distance
                            improved = True
```

```
    return route, best_distance

def perform_four_opt_swap(route, i, j, k, l):
    new_route = route[:i] + route[j:k+1][::-1] + route[i:j+1] +
route[l:] + route[k+1:l][::-1]

    return new_route

def calculate_distance(route, distance_matrix):
    total_distance = 0
    for i in range(len(route) - 1):
        cityA = route[i]
        cityB = route[i + 1]
        total_distance += distance_matrix[cityA][cityB]
    total_distance += distance_matrix[route[-1]][route[0]]
    return total_distance
```

3.3 Візуалізація результатів

Для візуалізації отриманих результатів скористаємося бібліотекою folium. Це бібліотека для мови програмування Python, яка дозволяє створювати інтерактивні картографічні візуалізації. Вона побудована на основі бібліотеки Leaflet.js, що робить її потужним інструментом для створення карт із використанням географічних даних.

```
import folium
from folium.plugins import MarkerCluster


# Створення базової карти Львова
lviv_map = folium.Map(location=[49.841952, 24.032236],
zoom_start=15)

# Додавання маркерів для точок маршруту
marker_cluster = MarkerCluster().add_to(lviv_map)

coordinates = calculations.coordinates

points = Data_points.points

for i, coord in enumerate(coordinates):
    point_name = f"Point {i+1}"
    point_location = [coord[0], coord[1]]
    points.append({"name": point_name, "location":
point_location})
```

```
    folium.Marker(location=point_location,
popup=point_name).add_to(marker_cluster)
    print(point_name,point_location)
1# Додавання ліній між точками маршруту
shortest_route_indices = best_chromosome +
[best_chromosome[0]]

line_locations = [points[i]["location"] for i in
shortest_route_indices]
folium.PolyLine(locations=line_locations,
color='blue').add_to(lviv_map)


# Відображення карти
lviv_map.save("C:/ route_map.html")
```

4 Результати роботи програми

Почнемо аналіз роботи алгоритмів з малої кількості точок.

а) 5 точок:

1)Алгоритм повного перебору.

Довжина шляху: 3276.

Кількість переборів: 24.

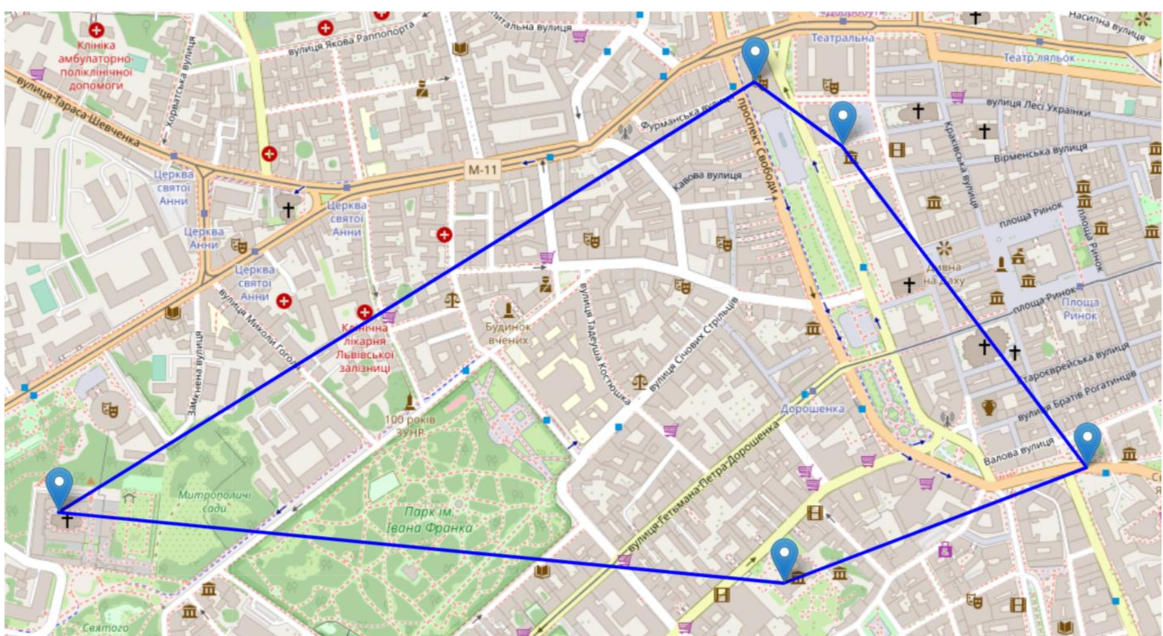

Рис.3.4.1

2)Алгоритм пошуку найближчого сусіда.

Довжина шляху: 3276.

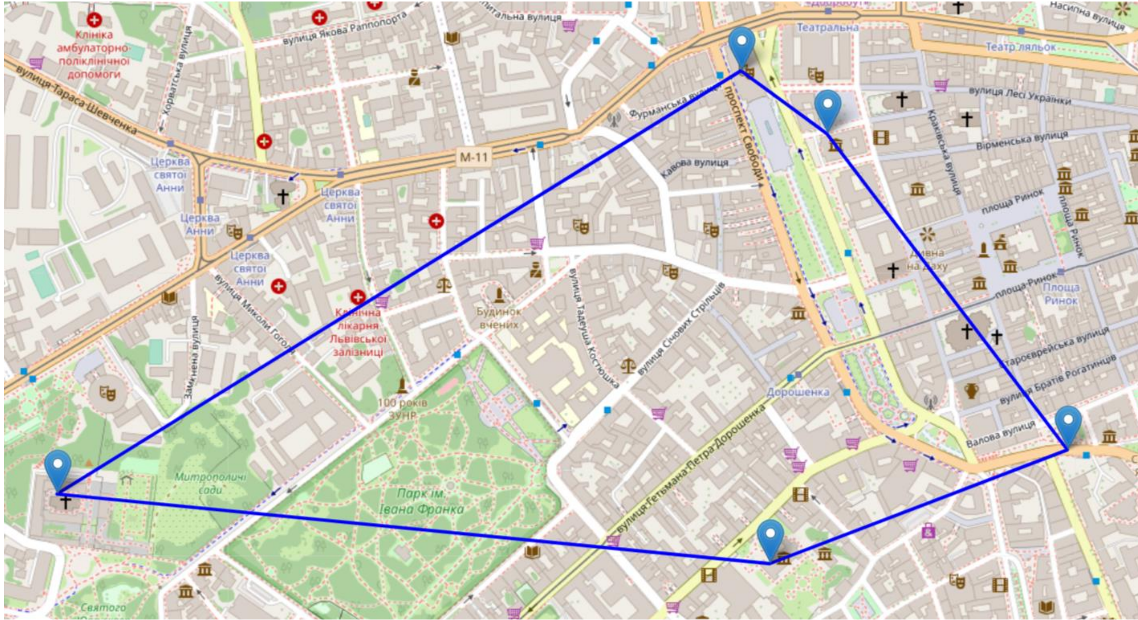

Рис.3.4.2

3)Генетичний алгоритм.

Довжина шляху: 3276.

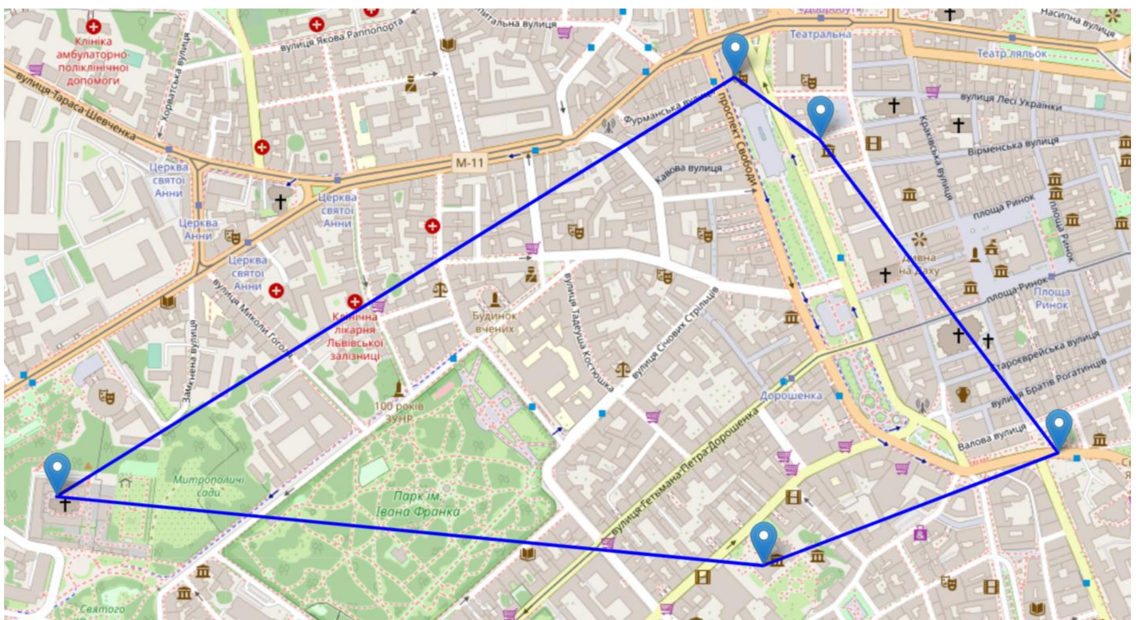

Рис.3.4.3

Висновки:

Кожен з алгоритмів знаходить оптимальний маршрут, а також, при даній кількості точок, кожен з них працює швидко. Додамо в маршрут більше точок та порівняємо результати знову.

б) 11 точок:

1) Алгоритм повного перебору.

Довжина маршруту: 6935 м.

Кількість переборів: 3628800.

Час виконання: 13.403428554534912 с.

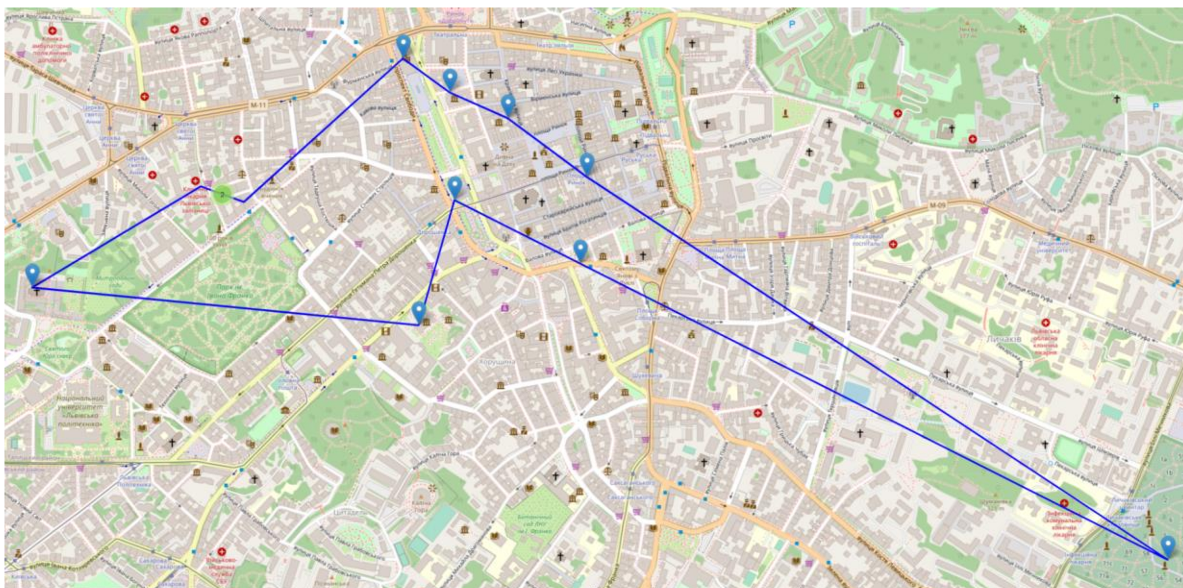

Рис.3.4.4

2) Алгоритм найближчого сусіда. (Рис.3.4.5)

Довжина маршруту: 8451 м.

3) Генетичний алгоритм. (Рис.3.4.5)

Довжина шляху: 6935 м.

Час виконання: 7.177375 с.

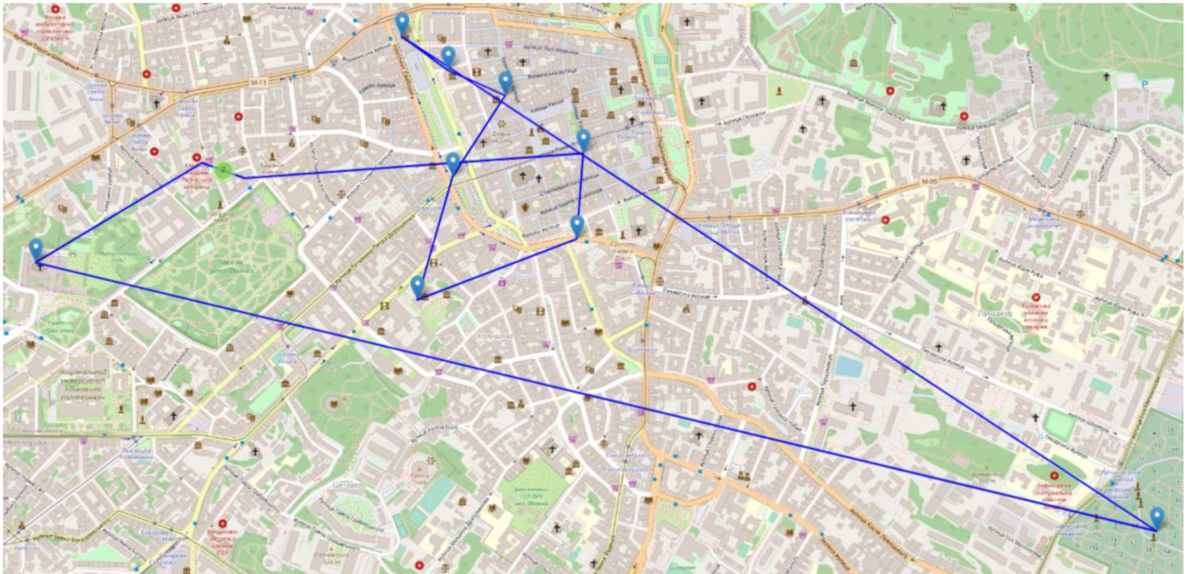

Рис.3.4.5

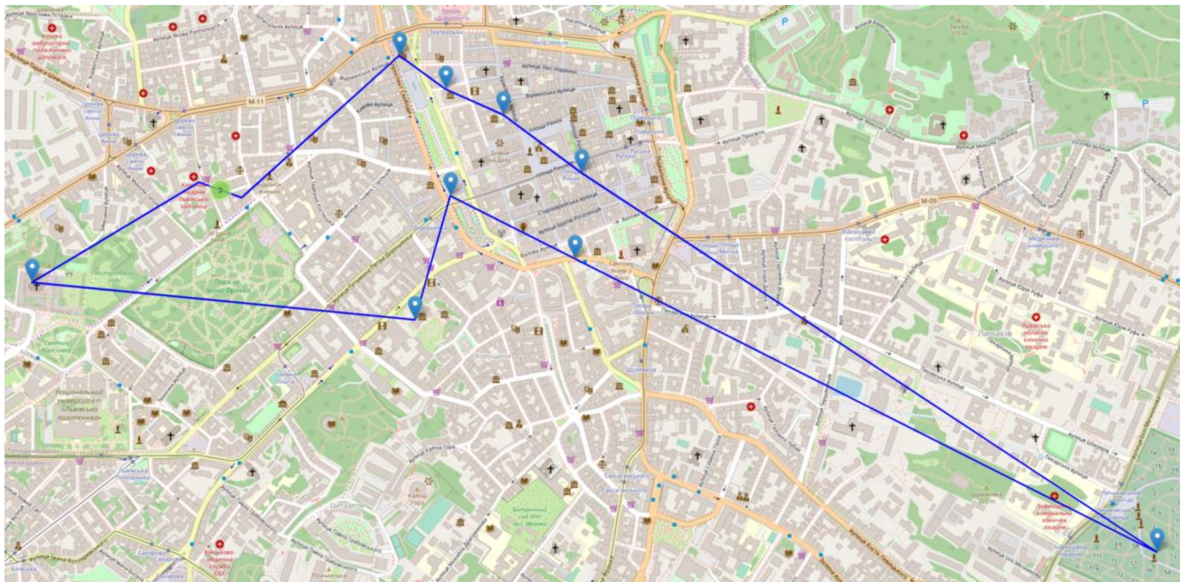

Рис.3.4.6

Висновки:

Алгоритм пошуку найближчого сусіда – найшвидший, але зі зростанням кількості точок на карті ми отримуємо результат побудови маршруту, що відрізняється від оптимального на 1516 м .

Алгоритм повного перебору є точним методом, тому результат його роботи – найкоротший з можливих шляхів, проте разом з такою точністю ми стикаємося з недоліками даного методу, а саме обчислювальна складність алгоритму. При побудові маршруту між 11 точками виконується 3628800 переборів різних комбінацій, на що комп'ютер витрачає 13.403428 секунди.

При 12 - 39916800 переборів, час виконання - 165.4999780 с. При намаганні обчислити оптимальний маршрут для 13 точок програма довго не завершувала роботу, а також використовувала великий обсяг пам’яті, тому було прийняте рішення не чекати завершення її виконання. Отже, емпірично досліджено, що максимальна кількість точок для побудови маршруту за допомогою даного алгоритму (використовуючи спроможності комп’ютера, характеристики якого описані в роботі) – 12.

Генетичний метод при даній кількості точок дає оптимальний результат.

в) Побудова маршруту для 60 точок:

Перейдемо до побудови оптимального маршруту для всієї кількості точок. Ми встановили раніше, що алгоритм повного перебору не можна використати для обробки такої кількості даних, тому зупинимося на використанні двох алгоритмів, а саме пошуку найближчого сусіда та генетичному алгоритмі.

1) Алгоритм пошуку найближчого сусіда. (Рис. 3.4.7)

Довжина шляху : 24264 м.

Час виконання : 0.000999 с.

2) Генетичний алгоритм. (Рис. 3.4.8)

Довжина аршруту: 20966 м.

Час виконання: 220.698038 с.

Висновки:

Генетичний алгоритм виконувався довше,але побудував маршрут коротший на 3298 м., ніж маршрут побудований алгоритмом пошуку найближчого сусіда, тож дав кращий результат наближеного до оптимального маршруту.

Саме до нього застосуємо k-opt метод. Емпіричним підходом було обрано k=4.

Рис.3.4.7

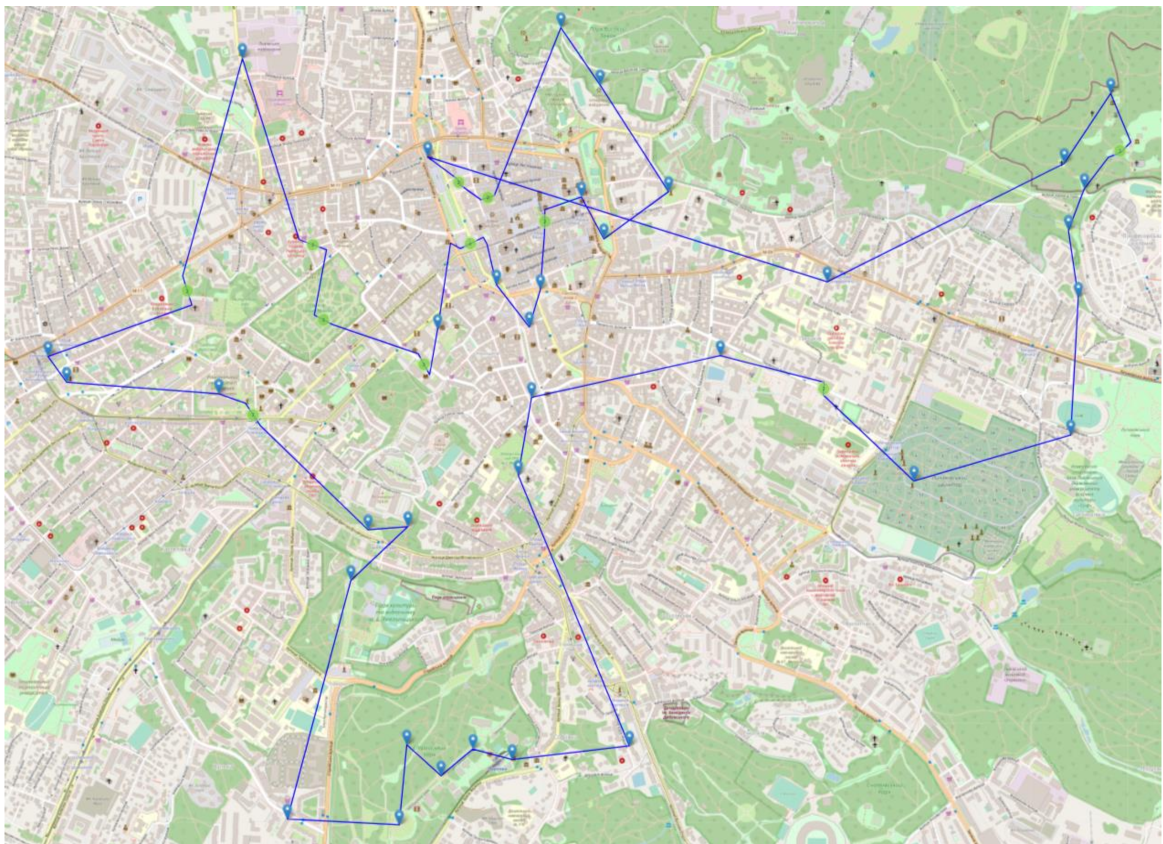

Рис.3.4.8

4) Покращення результату 4-opt методом.

Довжина маршруту: 19990 м.

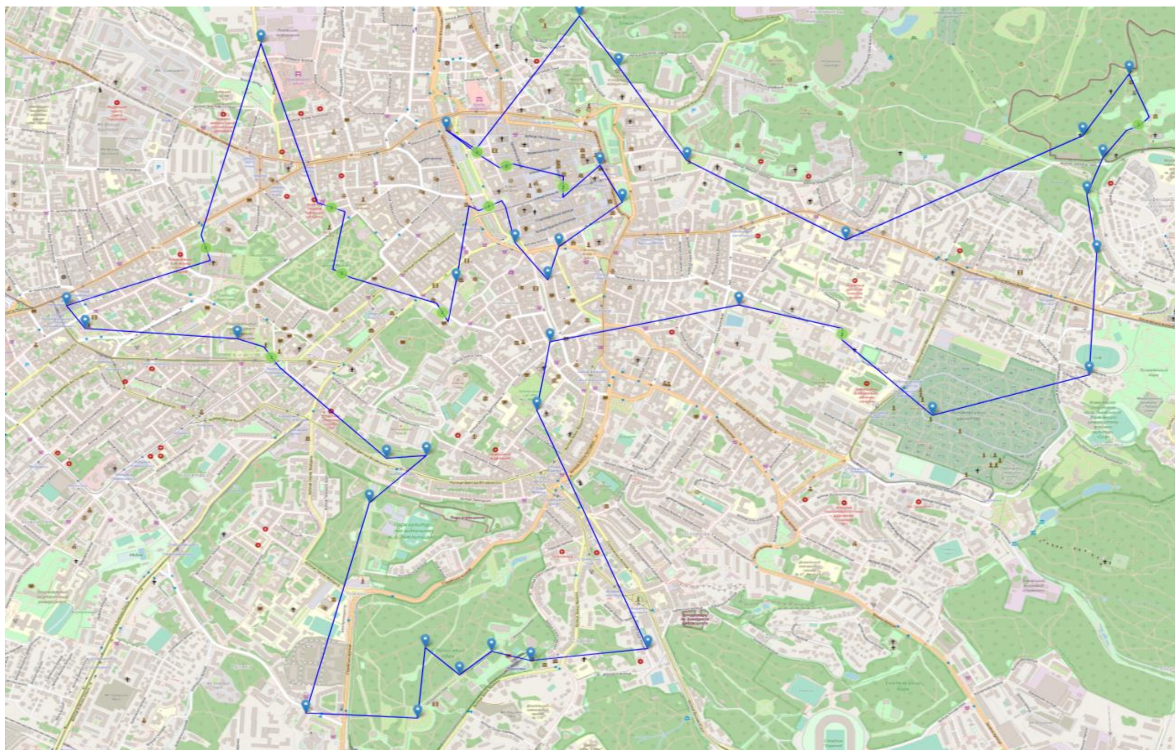

Рис.3.4.9

ВИСНОВОК

В роботі було розглянуто історію виникнення задачі Комівояжера, її теоретичну постановку, задачі, що зводяться до неї (Гамільтона, Переналадка станків) а також точні та наближені алгоритми розв'язання (алгоритм повного перебору, гілок та меж, алгоритм найближчого сусіда, алгоритм випадкових чисел, генетичний алгоритм, мурашиний алгоритм, k-opt алгоритм) та оцінку якості їх виконання.

В практичній частині було реалізовано вибірково 4 з них: алгоритм повного перебору, алгоритм найближчого сусіда, генетичний алгоритм, k-opt, та візуалізовані результати за допомогою бібліотеки Python folium. В результаті ми побудували наближений до оптимального маршрут та проаналізували роботу алгоритмів на прикладі використаної задачі.

Ми дослідили, що алгоритм повного перебору при потужності використаного технічного приладу ефективно працює при малій кількості точок (12), і зі збільшенням їх кількості час роботи та використання пам'яті зростає так, що використання даного методу стає не оптимальним.

Метод найближчого сусіда є найшвидшим, проте зростанням кількості точок алгоритм вже після збільшення пунктів до 12 почав будувати маршрут, довжина шляху якого відрізняється від оптимального на 22%. Проблеми з даним методом найбільш яскраво видно, якщо в даних є віддалені від основної сукупності точки.

Результати роботи генетичного алгоритму для всіх вхідних даних (60 точок) були найкращими, проте відрізнялися при різних вхідних параметрах. Його роботу можна покращити, змінивши налаштування, але зі зростанням точності зростає і час роботи програми. Швидшим підходом для отримання більш оптимального результату є використання k-opt методу до вже побудованого за допомогою генетичного алгоритму маршрута.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1)Sipser M. Introduction to the Theory of Computation / New York: Cengage Learning, 2012. – 256-276 с.

2)Davendra D. Traveling Salesman Problem, Theory and Applications / Croatia: InTech, 2010. - ISBN 978-953-307-426-9

3)Мудров В.И. Задача о коммивояжере / М.: Знание, 1969

4)Levitin, Anany Introduction to the design & analysis of algorithms / Anany Levitin. — 3rd ed., 2011. – 97 с.

5)D. Applegate The Traveling Salesman Problem: A Computational Study / Applegate D., Bixby R, Vašek Chvatál , Cook W - Princeton University Press, 2006. – 94 с.

6)M. Affenzeller, Genetic Algorithms and Genetic Programming Modern Concepts and Practical Applications / Affenzeller M., Wagner S, Winkler S., Beham A. – 2009.

7)C´esar Rego and Fred Glover / The Traveling Salesman Problem and Its Variations – 2007.

8)L . Bianchi An ant colony optimization approach to the probabilistic traveling salesman problem / Bianchi L. Gambardella L.M., Dorigo M. – Berlin.: Springer, 2002.

ДОДАТОК А. Координати місць

```
coordinates =[(49.844063,24.0260955),  #Львівська опера
            (49.8393106,24.0324403), #Пам'ятник Данилу Галицькому
            (49.8378833,24.0266434),  ##Палац Потоцьких
            (49.8387599,24.0127819),  #Собор Святого Юра
            (49.8432619,24.0277694),  #Вірменський собор
            (49.8410923,24.0188351),  # Будинок вчених
            (49.8324603,24.0535161),  # Музей-заповідник «Личаківський цвинтар»
            (49.8407785,24.0279352),  # Латинський катедральний собор
            (49.8426794, 24.0298398),  # Аптека-музей
            (49.8413164,24.0326871), #Музей-Арсенал
            (49.8407341, 24.0203864),  # Львівська майстерня шоколаду
            (49.8409674, 24.0273737) ,  # пам'ятник Тарасу Шевченку
           (49.8354456,24.0319384), # Пам'ятник Грушевському
           (49.8326696,24.0312027), #Ботанічний сад Львівського Університету
(49.8430549,24.027907), #Національний музей імені А.Шептицького
(49.8370203,24.0255571), #Львывська Галерея мистецтв
(49.8308247,24.0249719), #Цитадель руїни
(49.8359919,24.0484572), #Львівський національний медичний університет імені Данила
Галицького
(49.8384044,24.0196877), #Парк Франка
(49.8289055,24.0217634),  #Пам'ятник Богдану Хмельницькому
(49.8355731,24.0143173), #Національний університет "Львівська Політехніка"
(49.8409353,24.0294333), #Гарнізонний храм святих апостолів Петра і Павла УГКЦ
(49.8466322,24.0358067), #Нижній оглядовий майданчик Високого замку
(49.8486836,24.0335908), #Парк-пам'ятка "Високий Замок"
(49.8475715,24.0156367), #Музейно-культурний комплекс пивної історії - Львіварня
(49.8380579,24.0207415), #Головна алея парк Франка
(49.8359873,24.00572), #Пам'ятник Степану Бандері
(49.8369109,24.0046857), #Храм Свв. Ольги і Єлизавети
(49.8230239,24.0249164), #Стрийський ПАрк
(49.822978,24.0374674), #Український Католицький Університет
(49.8224938,24.0308593), #Державний меморіальний музей Михайла Грушевського
(49.8228676,24.0286381), #Сенсотека
(49.8219119,24.0268446),  #Фонтан Івасик Телесик
(49.8201552,24.0244817),  #Пам'ятник 110 Річчя Українському Футболу
(49.8203584,24.0181737),  #Будинок-кросворд
(49.8369449,24.0425843),  #Львівський національний університет ветеринарної медицини та
біотехнологій імені Степана Ґжицького
(49.8390368,24.0627844),  #Мозаїка КС
(49.8445792,24.0657403),  #Музей народної архітектури і побуту у Львові
(49.8379637,24.031817),  #Пам'ятник Володимиру Івасюку
(49.8440489,24.0644742),  #Водяний млин кін. XIX ст.
(49.8429212,24.0631295),  #Шевченківський Гай
(49.8442096,24.0651758),  #Козацький човен – Дуб
```

(49.8414322,24.0622233), #Оглядовий майданчик
(49.8438184,24.0619824), #Церква святої Параскеви П'ятниці зі Стоянова
(49.8395848,24.0486274), #Музей історії медицини Галичини ім. Мар'яна Панчишина
(49.8355461,24.0483882), #Музей хвороб людини
(49.8421776,24.0326923), #Львівський історичний музей
(49.8425265,24.029102), #Державний природознавчий музей НАН України
(49.8428134,24.0396745), #Пам'ятник на честь 125-річчя товариства "Просвіта"
(49.8362647,24.0261965), #Пам’ятник Василеві Стефанику
(49.8344204,24.0166786), #Пам'ятник Михайлу Вербицькому
(49.8411259,24.0360309), #Пам'ятник В'ячеславу Чорноволу
(49.841177,24.0292143), #Пам'ятник Івану Підкові
(49.8426191,24.0347617), #Пам'ятник Іванові Федорову
(49.8397917,24.0122782), #Пам'ятник Богдану-Ігорю Антоничу
(49.8307164,24.022729), #Пам'ятник Борцям за волю України
(49.8394731,24.0299748), #Пам'ятник Адамові Міцкевичу
(49.8352425,24.0158226), #Пам’ятник Львівським політехнікам, полеглим у роки Другої світової війни
(49.8463293,24.0646103),#Церква св. Трійці 1774 р. із с. Клокучка
(49.8341104,24.0623589) #Пластиліновий будиночок