КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп'ютерних наук та кібернетики
Кафедра дослідження операцій

ВИПУСКНА КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

зі спеціальності 113 «Прикладна математика»
на тему:

ПОРІВНЯЛЬНА ХАРАКТЕРИСТИКА ТА РЕАЛІЗАЦІЯ АЛГОРИТМІВ РОЗВ'ЯЗАННЯ ЗАДАЧІ КОМІВОЯЖЕРА

Студентки 4 курсу
Бандерс Марії Миколаївни

Науковий керівник:
доцент, кандидат фіз.-мат. наук
Якимів Роман Ярославович

Робота заслухана на засіданні кафедри дослідження операцій та рекомендована до захисту в ЕК, протокол №……. від.....2021 р.

Завідувач кафедри ДО проф. Іксанов О.М.

Київ – 2021

РЕФЕРАТ

Обсяг роботи 53 сторінки, 11 ілюстрацій, 6 таблиць, 13 використаних джерел, 2 додатки.

ЗАДАЧА КОМІВОЯЖЕРА, АЛГОРИТМИ І МЕТОДИ РОЗВ'ЯЗАННЯ ЗАДАЧІ КОМІВОЯЖЕРА, ТОЧНІ АЛГОРИТМИ, МЕТОД НАЙБЛИЖЧОГО СУСІДА, МЕТОДИ ВИПАДКОВОГО ПОШУКУ, АЛГОРИТМ КОЛОНІЇ МУРАХ.

Об'єктом дослідження є варіації задачі комівояжера та реалізація точних та наближених методів знаходження найвигіднішого маршруту для практичних задачах малого бізнесу.

Метою роботи є дослідження теоретичних аспектів розв'язання задачі комівояжера, а також написати написання та перевірка програмного забезпечення для пошуку оптимального розв'язання задач із візуалізацією.

Інструменти розробки: середовище програмування Microsoft Visual Studio 2016, мова програмування C++, а також середовище VS Code, мова програмування Python 3.9.4.

У роботі виконане теоретичне дослідження та огляд алгоритмів та методів розв'язання задачі комівояжера, варіації постановки даної задачі, розроблено програмне забезпечення для реалізації обраних алгоритмів для пошуку оптимальних маршрутів для поставлених задач, зроблено висновки щодо доцільності використання різних алгоритмів для практичного застосування.

ЗМІСТ

ВСТУП

Транспортна логістика – це система по організації доставки, під чим розуміється переміщення яких-небудь матеріальних предметів чи об'єктів, речовин тощо з одного місця в іншу точку по оптимальному маршруту. Оптимальним маршрутом вважається такий маршрут, що логістичний об'єкт можна доставити у найкоротші терміни (або заздалегідь визначені терміни) з мінімальними витратами та мінімальною шкодою для об'єкта доставки (шкодою вважається негативний вплив зі сторони зовнішніх факторів, наприклад, умов транспортування, часових факторів при доставці об'єктів, на які впливає час доставки – як приклад, перевезення об'єктів, що мають невеликий термін придатності).

Транспортну логістику можна представити у вигляді шести послідовних задач, при чому до кожної за виключенням першої можна переходити лише після розв'язання попередньої. Цими задачами є:

1. Врахування наявності товарів на складі, реєстрація товару на складі з інформацією про залишок.
2. Врахування та обробка заявок на отримання товарів – збір заявок з усіх пунктів та наступне формування спільної заявки.
3. Розрахунок завантаження транспорту, обраховування необхідної кількості грузових робочих одиниць (автомобілів) для того, щоб транспортувати усі товари з складеної заявки з урахуванням параметрів грузу, такі як форма, об'єм, маса та можливостей автомобілей (вантажопідйомність, довжина, ширина, висота відсіку для багажу або контейнера).
4. Розбиття оброблюваного участку на геозони (кластери) з урахуванням наявних одиниць транспорту (задача кластеризації).

5. Побудова найбільш вигідного маршруту для доставки товару (задача комівояжера).
6. Комплектування контейнера або багажного відсіку автомобілей, що використовуються (щоб за прибуття до відповідного пункту призначення об’єкт для нього знаходився відносно інших ближче до «початку» контейнера/відсіку), вона ж – задача про рюкзак.

Таким чином можемо запевнитись, що задача комівояжера є одним з необхідних елементів транспортної логістики, якій передують деякі інші задачі, а також після якої стоять інші завдання, які неможливо виконати без отримання відповіді на питання «який маршрут є оптимальним».

На практиці, у реальному житті підприємці дуже часто зустрічаються з задачею комівояжера у певному вигляді, при чому необхідно побудувати для транспортування не лише оптимальний маршрут з мінімальними витратами на транспортування, а й знайти цей маршрут за мінімальний час для того, щоб мати перевагу над конкурентами.

Саме тому у даній кваліфікаційній роботі будуть розглянуті як теоретичні аспекти дослідження та розв’язання задачі комівояжера, так і буде проведено власне дослідження роботи алгоритмів на тестових наборах даних. Крім теоретичного обґрунтування того, який алгоритм є оптимальним для вирішення практичних задач комівояжера, ми практично розглянемо, який буде показувати оптимальний результат із використанням підручних засобів техніки (а не суперкомп’ютерів).

Доповідь структурована наступним чином. Розділ 1 – теоретичний і включає у себе розгляд основних понять, необхідних для постановки та розробки алгоритмів для розв’язання задачі комівояжера, постановку задачі комівояжера, методи та алгоритми розв’язання задачі комівояжера, що включать як точні, так і наближені. Розділ 2 – практичний. У ньому сформулювана задача, до якої буде

написана програма обрахунку, надано інформацію про тестувальний набір даних, а також продемонстровано результати роботи програми.

У висновках буде наведено результат порівняльного аналізу теоретичного та практичного застосування алгоритмів та методів для розв’язання задачі комівояжера, а також зроблено висновок, які алгоритми є оптимальними для розв’язання практичних задач з використанням класичної техніки.

Мета роботи:

Дослідити теоретичні аспекти розв’язання задачі комівояжера, а також написати програму та перевірити її на класичних обчислювальних приладів, що є доступними для малого та класичного бізнесу.

Завдання роботи:

1. Провести теоретичний огляд умови задачі комівояжера.
2. Розглянути відомі способи для розв’язання задачі комівояжера, побудувати теоретично алгоритми для розв’язання.
3. Зробити теоретичну оцінку алгоритмів розв’язання задачі комівояжера.
4. Сформулювати задачу комівояжера, для якої буде написана реалізація обраних методів розв’язання.
5. Оцінити швидкість та ефективність алгоритмів на прикладі програми.
6. Зробити порівняння виконаної теоретичної та практичної оцінки ефективності роботи алгоритмів.

РОЗДІЛ 1. ТЕОРЕТИЧНА ЧАСТИНА

1 Розгляд основних понять

1.1 Класи задач P, NP

При розв’язуванні великої кількості задач оптимізації математики багаторазово зіштовхувались з тим, що ці задачі були певною мірою важкі для розв’язання – важкі у тому сенсі, що для них на той момент не були знайдені ефективні алгоритми, що мають поліноміальну складність, а також є вагомі аргументи, щоб припустити, що такі алгоритми взагалі не існують.

Результатом досліджень таких задач стали поняття класів P, NP.

Клас P (від англійського «polynomial») – це такий клас задач, які можна розв’язати детермінованими алгоритмами за поліноміальний час роботи.

Клас NP (від англійського «non-deterministic polynomial») – це клас задач, які можна розв’язати за поліноміальний час роботи недетермінованими алгоритмами.

Пояснимо різницю між детермінованими та недетермінованими алгоритмами. Припустимо, що під час виконання алгоритму дійшли до такого місця, де має бути зроблений вибір з декількох варіантів. Детермінований алгоритм дослідив би одну альтернативу повністю та потім повернувся б до дослідження інших, а недетермінований може досліджувати усі можливі варіанти одночасно, по суті копіюючи сам себе для кожного варіанту, що розглядається. Усі копії такого алгоритму незалежні та можуть створювати подальші копії. Якщо одна з копій отримує результат, що зробила на певному кроці неправильний вибір, то вона зупиняє своє виконання, а якщо знаходить розв’язання, то сигналізує про успіх та всі інші копії припиняють виконання алгоритму.

Задача класу NP є складною, якщо кожна кожна задача з NP зводиться до неї; задача називається NP-повною, якщо вона одночасно є NP-складною та входить у NP. NP-повну задачу неможливо розв'язати жодним відомим поліноміальним алгоритмом.

Звичайно, поняття NP-повної задачі має практичне значення – такі задачі є складнорозв'язними з обчислювальної сторони, тобто для будь-якого алгоритму, що розв'язує NP-повну задачу у найгіршому випадку знадобиться експоненційна кількість часу, що робить його неможливим для використання на практиці за виключенням задач з дуже малою кількістю даних.

Одним з можливих підходів до таких задач заключається підхід, що базується на побудові алгоритмів поліноміальної складності, які знаходять близький до оптимального результат (який часто збігається з оптимальним), так звані наближені алгоритми.

1.2 Необхідні визначення з теорії графів

Графом $G = (X, U)$ називається деяка множина точок X, що з'єднані між собою відрізками (множина U). Графи будуються без врахування масштабних співвідношень.

Елементи множини X називаються вершинами (або вузлами) графу. Відрізки з множини U називають ребрами. Якщо граф є орієнтованим, тобто відрізки з множини U мають напрямок, то їх називають дугами.

Маршрутом (або орієнтованим маршрутом) орієнтованого графа називається послідовність дуг, у якій кінцева вершина усякої дуги, крім останньої, є початковою вершиною наступної дуги, що входить у маршрут.

Маршрут називається ланцюгом, якщо усі дуги різні (тобто зустрічаються рівно один раз).

Ланцюг є простим, якщо усі його вершини різні (окрім, можливо, першої та останньої, якщо маршрут є замкненим).

Цикл – це маршрут, що починається та закінчується в одній і тій самій вершині.

Гамільтонів цикл у орієнтованому графі – це орієнтований цикл (контур), що проходить рівно один раз через кожну вершину графу (тобто простий орієнтований ланцюг).

Іноді дугам графу G співставляються числа – дузі (x_i, x_j) ставиться у відповідність число c_{ij}, що називають вагою дуги або її довжиною. Тоді граф G називається графом зі зваженими дугами.

1.3 Лінійне програмування

Практичні задачі щодо різних областей людської діяльності часто характеризуються наступними властивостями:

1) На змінні величини накладається деяка, часто велика кількість обмежень.
2) Задача має велику варіантивність розв'язків, з яких потрібно знайти найкращий або оптимальний.

Наявність великої кількості обмежень та можливих розв'язків створює труднощі при розв'язанні таких задач. Розділ математики, який включає методи, що дозволяють шляхом певних розрахунків знаходити шукане розв'зання з усіх можливих, називається математичним програмуванням.

Математична модель – це точний опис задачі за допомогою математичного апарату (для цього можуть використовуватись функції, рівняння та/або системи рівнянь, нерівності та/або системи нерівностей тощо). У моделі мають бути враховані всі істотні фактори, задані умовою задачі.

Загальною задачею лінійного програмування, що представляється у формі запису, називається задача, у якій потрібно визначити оптимум (мінімум або максимум) цільової функції

$$F = c_1x_1 + c_2x_2 + \cdots + c_nx_n \to opt$$

при обмеженнях

$$\sum_{j=1}^{n} a_{ij}x_j \leq (\geq) b_i;\ \ (i = \overline{1,s}),$$

$$\sum_{j=1}^{n} a_{ij}x_j = b_i;\ \ (i = \overline{s+1,m}),$$

$$x_j \geq 0, j = \overline{1,n},$$

де a_{ij}, b_i, c_j – деякі коефіцієнти.

2 Постановка задачі комівояжера

Комбінаторика є розділом математичної науки, що вивчає комбінації та їх кількості, зіставлені з урахуванням певних конкретизованих умов, з заданих об’єктів. Задача комівояжера є однією з найвідоміших задач комбінаторної оптимізації, вона ключова для організації транспортної логістики. Вона полягає у знаходженні найвигіднішого маршруту, що проходить через вказані міста (пункти, точки) хоча б по одному разу. У якості умов вказується критерій вигідності маршруту (найкоротший, найдешевший, найвигідніший за сукупністю цих характеристик тощо) та відповідні матриці відстаней, вартостей тощо.

Наведемо класичне формулювання задачі комівояжера.

Комівояжер має проїхати n міст. Для того, щоб зменшити витрати, він повинен побудувати маршрут таким чином, щоб побувати у кожному місті по одному разу та повернутись у початкове.

Розглянемо математичну модель даної задачі.

Маємо n пунктів, вартість проїзду між якими задана матрицею C. Математично маршрут, який він має пройти за умовою мінімізації вартості, можна представити формулою:

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

де n – кількість пунктів, c_{ij} – відповідний елемент матриці вартостей C між пунктами, x_{ij} – елемент матриці переходу між компонентами (дорівнює 1, якщо маршрут включає у себе переїзд з точки i безпосередньо у точку j і рівний 0 у протилежному випадку).

Якщо зробити невелику видозміну умови постановки задачі та не потребувати повернення у початкове місто, отримаємо незамкнену задачу комівояжера.

Простота формулювання умови йде поруч з надзвичайною складністю її розв’язання, при чому складність полягає в обчислювальній частині, адже один з варіантів розв’язання цілком очевидний – перебрати усі можливі маршрути та обрати найменший, найкоротший, оптимальний. Але на практиці навіть найшвидші обчислювальні машини та суперкомп’ютери не можуть виконати такий перебір при значній кількості міст.

Виходячи з першого міста, комівояжер може відправитись у будь-яке з $(n-1)$ з міст, що ще лишились, з якого може відправитись у $(n-2)$ з міст, що лишились і так далі, поки не залишиться одне місто, у якому він ще не побував.

Отже, для повного перебору можливих варіантів, які необхідно буде розглянути у повному переборі, буде розібрано

$$(n-1)(n-2)\dots 2\cdot 1=(n-1)!$$

варіантів, а зі збільшенням кількості міст n це число буде надзвичайно швидко зростати і вже при 15-20 містах досягає астрономічних чисел. Час для виконання повного перебору також пропорційно зростає, а отже від такого методу розв'язання необхідно відмовлятись у першу чергу через неможливість вкластись у будь-яке обмеження у часі, що може виникнути при розв'язанні практичних задач.

У теперішній час існує чимало алгоритмів та методів для розв'язання задачі комівояжера, але їх пошуки були надзвичайно складними. Проте перед переходом до огляду самих варіацій розв'язання задачі, розглянемо деякі варіації формулювання проблеми, що розглядається, а також задачі Ейлера та Гамільтона, математична модель та формулювання яких співпадає з задачею комівояжера та саме з них вона починає свою історію.

2.1 Варіації задачі комівояжера

1) Задача Гамільтона

Англійський математик, механік-теоретик та фізик-теоретик сер Уільям Роуен Гамільтон вигадав гру, що отримала у свій час широку популярність. Іграшка представляє собою додекаедр, тобто правильний багатогранник з 12 п'ятикутними гранями та 20 вершинами. Вважаємо, що цей додекаедр приблизно зображає земну кулю. Кожна вершина зображає столицю однієї з держав. Необхідно побувати у всіх столицях рівно один раз та повернутись у вихідний пункт, при цьому рухатись можна тільки виключно ребрами додекаедра.

Кожен такий шлях, якщо він існує і задовольняє описані умови, називається гамільтоновим контуром або гамільтоновим циклом.

Зведемо дану задачу до вже відомої у формулюванні задачі комівояжера. Для цього будемо вважати, що відстані між кожними двома містами-столицями рівні 1, якщо вони знаходяться на одному ребрі та нескінченності, якщо на різних. За цим можемо створити матрицю взаємних відстаней між 20 містами (таблиця 1.2.1, де ми нескінченні відстані залишимо порожніми клітинками для зручності). Неважко переконатись, що якщо гамільтонів шлях існує, то його довжина дорівнюватиме 20, якщо ж не існує, то довжина буде рівна нескінченності. Тому якщо розв’язати задачу комівояжера з використанням заданої матриці у якості вхідних даних, то ми одразу отримаємо один з варіантів такого маршруту (якщо він існує).

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	x	1			1											1				
2	1	x	1											1						
3		1	x	1							1									
4			1	x	1			1												
5	1			1	x	1														
6					1	x	1										1			
7						1	x	1										1		
8				1			1	x	1											
9								1	x	1	1									
10			1						1	x			1							
11									1		x	1						1		
12											1	x	1							1
13										1		1	x	1						
14		1											1	x	1					
15														1	x	1				1
16	1														1	x	1			
17						1										1	x		1	
18							1				1							x	1	
19																	1	1	x	1
20												1			1				1	x

Таблиця 1.2.1

Практично установлено, що для вказаного додекаедра існує декілька рівноцінних гамільтонових циклів, один з яких зображено на розгортці (рисунок 1.2.1), де маршрут зображено товстим контуром.

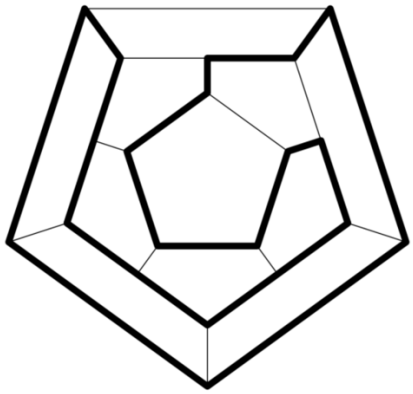


Рисунок 1.2.1

Дану задачу можна дещо ускладнити, якщо кожному ребру поставити у відповідність деякі нерівні між собою числа (замість одиниць), які можна практично назвати відстанями між сусідніми столицями. Тоді розв’язання задачі комівояжера приведе до, як правило, єдиного можливого оптимального розв’язання, а не певної кількості рівнозначних маршрутів, які виникають за рахунок однакових відстаней між сусідніми столицями.

2) Задача про обхід шахової дошки

Дуже схоже виглядає задача про обхід конем шахової дошки, яку розглянув ще Леонард Ейлер, відомий математик та механік.

Формулювання даної задачі виглядає наступним чином: потрібно обійти конем усі поля шахової дошки, побувавши у кожному полі по одному разу.

Перенумеруємо усі клітинки шахової дошки і будемо вважати, що «відстані» між двома клітинками дорівнюють одиниці, якщо кінь може за один хід з однієї потрапити в іншу, і що дана «відстань» буде рівна нескінченності у протилежному випадку. Аналогічно до таблиці 1.2.1 для задачі Гамільтона можемо побудувати матрицю відстаней для данох задачі. Таким чином

отримаємо незамкнену задачу комівояжера і у розв'язанні отримаємо шлях довжиною 63 (для шахової дошки 8×8) або нескінченність. Якщо отримати шлях із довжиною 63, то такий обхід можливий і розв'язання вкаже один з таких можливих обходів, якщо ж нескінченність, то задача нерозв'язна.

3) Задача екстреного збору

Розглянемо ще одну задачу, більш прикладну та дещо ближчу до реального життя.

Студенти одного університету живуть у різних частинах міста та проходять спільний квест. Час, який потрібен для того, щоб перейти від місця проживання i-ого студента до місця проживання j-ого студента позначимо як t_{ij}. Один з них чує сигнал збору та дізнається кінцеве місце зустрічі. Після цього він добирається до місця проживання другого студента, розказує йому про кінцеве місце зустрічі та направляється у місце зустрічі сам. Після цього другий студент направляється до третього і так далі. Останній студент після отримування такого сигналу відправляється одразу у місце зустрічі, заходити до інших студентів йому не потрібно. Запитання задачі полягає у тому, щоб мінімізувати час, за який буде виконано такий ланцюжок дій студентів.

Сформульована таким чином задача є задачею про найкоротший маршрут з n ланок. Відмінність даної задачі від класичного формулювання незамкненої задачі комівояжера у тому, що останній, кінцевий пункт маршруту не має значення, а у незамкненій задачі комівояжера вказаний як пункт початку, так і кінцевий.

4) Інші варіації

Можна сформулювати безліч прикладних задач, які будуть зводитись до задачі комівояжера, пов'язаних з об'їздом міст чи пунктів з певною метою та поверненням у вихідну точку, наприклад, задачі доставки товарів, пошта, постав продуктів харчування у торгових точках тощо. Так само існують варіації

формулювання з прокладанням кільцевих ліній енергопередач або прокладанням газопроводу, проводження електрики до робочих місць тощо. Їх усі буде поєднувати можливість розв'язання та зведення до задачі комівояжера.

3 Методи та алгоритми розв'язання задачі комівояжера

По складності класична задача комівояжера відноситься до складу NP-повних задач, тобто оптимальний ефективний поліноміальний алгоритм розв'язання для пошуку точної відповіді ще не розробили. Точною відповіддю називаємо маршрут мінімальної вартості, довжини, тривалості (в залежності від заданих умов) серед усіх можливих маршрутів для даних міст. Наближеною відповіддю вважаємо маршрут мінімальної вартості, довжини та тривалості (знову ж у залежності від заданих умов) серед усіх знайдених алгоритмом маршрутів.

Усі методи та алгоритми для пошуку відповіді на задачу комівояжера можна поділити на алгоритми точного та наближеного розв'язання.

3.1 Точні методи розв'язання класичної задачі комівояжера

1) Алгоритм повного перебору

Цей алгоритм може використовуватись для довільної класичної задачі комівояжера, але його використання обмежується дуже високою обчислювальною складністю. Для розв'язання симетричної задачі розглядається $\frac{(n-1)!}{2}$ варіантів маршруту, а для несиметричної – $(n-1)!$. Через те, що факторіал зростає дуже швидко зі збільшенням параметру, що відповідає за кількість міст (таблиця 1.3.1), даний алгоритм використовується для розв'язання задач, де за умовою дається маленька кількість міст (або пунктів, що треба відвідати).

Кількість міст n	Кількість маршрутів $(n-1)!$
5	24
10	$\sim 10^5$
15	$\sim 10^{11}$
20	$\sim 10^{17}$
25	$\sim 10^{24}$
30	$\sim 10^{31}$
35	$\sim 10^{38}$
40	$\sim 10^{46}$
45	$\sim 10^{54}$

Таблиця 1.3.1

Для проведення перебору потрібно генерувати перестановки без повторень з n елементів. Для реалізації цього є багато методів, наприклад, можна зробити це лексикографічним перебором. Для цього задаємо алфавіт та, оскільки букви у ньому впорядковані, то можемо впорядкувати і слова (спершу порівнювати першу літеру, далі другу і так далі до того моменту, поки одне зі слів не закінчиться або не знайдеться різна літера). Наприклад, якщо пронумерувати чотири міста від 1 до 4, то перша лексикографічна розстановка буде 1-2-3-4, друга – 1-2-4-3, остання – 4-3-2-1.

2) Метод динамічного програмування

Реалізація даного методу потребує дуже великих витрат оперативної пам'яті, що, так само як і для алгоритму повного перебору, значно зменшує кількість міст, для яких даний алгоритм можна практично застосувати.

Нехай v_1 – початкове місто, а $v_2, v_3, \dots, v_n$ – інші міста, перестановки з k міст будемо позначати як $\pi^k = (\pi_1^k, \dots, \pi_k^k)$, при чому $\pi_i^k \neq \pi_j^k$ при $i \neq j$, а $i, j = 1,2, \dots, n$, $\pi_i^k, \pi_j^k \in \{v_1, \dots, v_n\}$, $\pi_1^k = v_1$, $f_k(\pi_1^k, \dots, \pi_{m-1}^k, \pi_{m+1}^k, \dots, \pi_k^k, \pi_m^k)$ є функцією пошуку найкоротшого шляху від міста v_1 до v_m, $m = 2,3, \dots, k-1$ яка

проходить через усі інші міста. Тоді функція для обчислення довжини найкоротшого шляху з k міста можна задати рекурентним співвідношенням

$$f_k\left(\pi_1^k, \dots, \pi_k^k\right) = \min_{m=2,k-1}[f_{k-1}\left(\pi_1^k, \dots, \pi_{m-1}^k, \pi_{m+1}^k, \pi_{k-1}^k, \pi_m^k\right) + d\left(\pi_m^k \pi_k^k\right)].$$

У даній формулі $d\left(\pi_m^k, \pi_k^k\right)$ – довжина дороги між відповідними містами.

Алгоритмом розв'язання класичної задачі комівояжера може бути послідовне обрахування записаної рекурентної формули.

3) Метод гілок та меж

Даний метод використовується для задач цілочисельного лінійного програмування. Алгоритм Літтла та інших використовує оцінку нижньої границі довжини маршруту, яку можна отримати через властивості зведеної матриці та процесу зведення.

Ключовими у даному алгоритмі є гіллястість дерева та знаходження меж. Основною ідеєю алгоритму є послідовно розбити множину допустимих розв'язків класичної задачі комівояжера завдяки гіллястості на множину підмножин, що не перетинаються, а на основі оцінок, що отримуються з процедури знаходження меж, обирати наступні перспективні підмножини для пошуку найкоротшого маршруту. Після цього виконання алгоритму повторюється.

Використання методу гілок та меж на сучасних обчислювальних машинах дозволяють розв'язувати класичні задачі комівояжера для 100 міст, коли метод повного перебору не застосовується для кількості міст більшої за 15, що є досить непоганим результатом у порівнянні. Подібні алгоритми є евристичними та дозволяють передбачити та виділити підмножини, у яких найбільш ймовірно знайдеться шуканий найкоротший шлях.

Оскільки поняття евристичного алгоритму ще буде зустрічатись у даній роботі, проведемо розбір вищезгаданого алгоритму Літтла. Даний алгоритм є

частковим випадком методу гілок та меж, його швидкість у найгіршому випадку рівна швидкості алгоритму повного перебору. Він надає відповідь на задачу комівояжера у вигляді гамільтонова контура. Опишемо роботу алгоритму:

1) У кожній стрічці матриці відстаней знайдемо мінімальний елемент та віднімемо його від усіх елементів стрічки. Зробимо те ж саме для стовпців, що не містять нулів. Отримаємо матрицю вартостей, кожна стрічка та кожен стовпчик якої містять принаймні один нульовий елемент.
2) Для кожного елемента матриці c_{ij} розрахуємо коефіцієнт $\Gamma_{i,j}$, який буде рівний сумі найменшого елемента та стрічки (за виключенням елементу $c_{ij} = 0$) та найменшого елемента j стовпчика. З усіх коефіцієнтів $\Gamma_{i,j}$ оберемо такий, що є максимальним. Позначимо його як $\Gamma_{k,l} = \max\{\Gamma_{i,j}\}$. Відповідну дугу (k, l) вносимо у гамільтонів контур.
3) Видаляємо k-ту стрічку та стовпчик I, заміняємо на нескінченність значення елементу c_{lk}, оскільки дуга (k, l) включена у контур та зворотній шлях з l в k неприпустимий.
4) Алгоритм кроку 1 повторюємо до того моменту, поки порядок матриці більший за 2.
5) Далі у поточний орієнтований граф вносимо дві дуги, яких не вистачає та які однозначно визначаються матрицею з порядком 2. Отримуємо гамільтонів контур.

У ході розв’язання у циклі постійно рахується значення нижньої границі, що рівне сумі усіх викреслених елементів у стрічках та стовпчиках. Результуюче значення нижньої границі має співпасти з довжиною результуючого гамільтонового контуру.

3.2 Наближені методи розв’язання задачі комівояжера

1) Метод найближчого сусіда

Даний метод базується на так званому жадібному алгоритмі. Жадібний алгоритм – це такий алгоритм, що заключається у прийнятті локально оптимальних рішень на кожному кроці, припускаючи, що кінцеве розв'язання також вийде оптимальним.

Міста послідовно включаються у маршрут, при чому кожне наступне місто має бути найближчим до попереднього пункту серед усіх тих міст, які ще не включені у маршрут.

2) Метод включення найближчого сусіда

На кожному кроці робиться обхід по усіх містах, що вже включені у маршрут. У маршрут додається місто, для якого знайдеться найближчий сусід з числа вже відвіданих пунктів маршруту. Пункт вноситься у маршрут біля сусіда з найкоротшим шляхом, а сусідні вершини модифікуються, щоб розширення маршруту було припустимим.

3) Метод найдешевшого включення

На кожному кроці алгоритм будує маршрут по вже існуючій підмножині пунктів обходу, що вже включені у маршрут, додаючи у нього таке місто у певні ланки маршруту, щоб мінімізувати збільшення вартості усього маршруту.

Усі три розглянуті наближені алгоритми жадібними. Насправді іноді це може приводити до дуже великих похибок і давати чи не найгіршу відповідь для кінцевого маршруту, якщо на одному з кроків буде обрано найкоротший маршрут, а надалі усі відстані будуть максимальними.

Це спричинено тим, що завжди розгляд йде лише на один крок вперед, а не на декілька. Можна зробити невелике покращення, якщо одразу розглядати не одне найближче місто, а, наприклад, 3, проте тоді відповідно значно зросте і складність обрахунків.

4) Метод найкоротшого острову

Даний метод складається з трьох послідовних кроків. Наші міста інтерпретуємо як вершини плаского графу, а довжини між містами будуть рівні довжинам ребер (рисунок 1.3.1, а). Перший крок – це за алгоритмом Пріма побудувати найкоротше остівне дерево, тобто таке дерево, що містить усі вершини графу (рисунок 1.3.1, б).Кожне ребро ми подвоїмо та задамо напрям (тобто кожне неорієнтоване ребро перетворемо на два орієнтованих, що мають протилежний напрям) та отримаємо мультиграф (рисунок 1.3.1, в). Граф є Ейлеровим, бо усі вершини мають парний степінь.

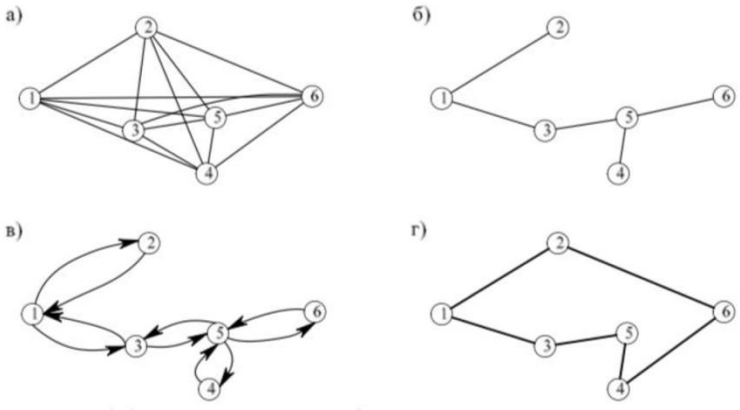


Рисунок 1.3.1

На другому кроці у даному мультиграфі виділяється маршрут найменшої довжини, який проходить через кожен пункт, при цьому кожна пара сусідніх вершин у ньому зустрічається рівно один раз. Маємо ейлерів цикл. Після видалення повторних входжень пунктів на трьому кроці кожен пункт зустрічається принаймні один раз. Ейлерів цикл переходить у гамільтонів та ми отримуємо наближене розв’язання задачі комівояжера (рисунок 1.3.1, г).

5) Методи випадкового пошуку

Методи з використанням елементів випадковості з’явились як наслідок очевидності недоліків методів, заснованих на переборі та математичних моделях, що вже були розглянуті раніше у даній роботі.

Введення випадково побудованого маршруту дозволяє побудувати алгоритми випадкового пошуку, які в умовах великої кількості параметрів, розмірності та початкової невизначеності задачі можуть працювати швидше за інші алгоритми пошуку та оптимізації. Найпростіший вид – грубий випадковий пошук. Розв’язання класичної задачі комівояжера тоді виглядає наступним чином:

1) випадково будуємо початковий маршрут - послідовно кожен пункт, крім першого, обираємо випадковим чином;
2) маршрут-кандидат будуємо так само випадковим генеруванням наступної вершини;
3) порівнюємо маршрут-кандидат та початковий. Якщо кандидат виявляється коротшим, то він стає новим оптимальним маршрутом;
4) алгоритм закінчується за деякого критерію зупинки (наприклад, коли кількість перебраних варіантів перевищила задане число або через ліміт заданого часу виконання програми).

Даний підхід дає швидкий результат та може бути покращений за використання евристичних методів та методів локального пошуку, щоб врахувати перспективність обирання маршруту.

6) Генетичні алгоритми

Такі алгоритми входять до групи адаптивних методів. В основі генетичних алгоритмів лежить принцип природнього відбору. У процесі пошуку розв’язання аналізується декілька гілок еволюції. Користуючись «функцією адаптації», яка оцінює якість розв’язання задачі та яка виконує функцію навколишнього

середовища при моделюванні еволюційного процесу, генетичні алгоритми «вирощують» популяції об’єктів з більш адаптованою до реальної ситуації «генною» структурою. Блок-схема класичного генетичного алгоритму наведена на рисунку 1.3.2.

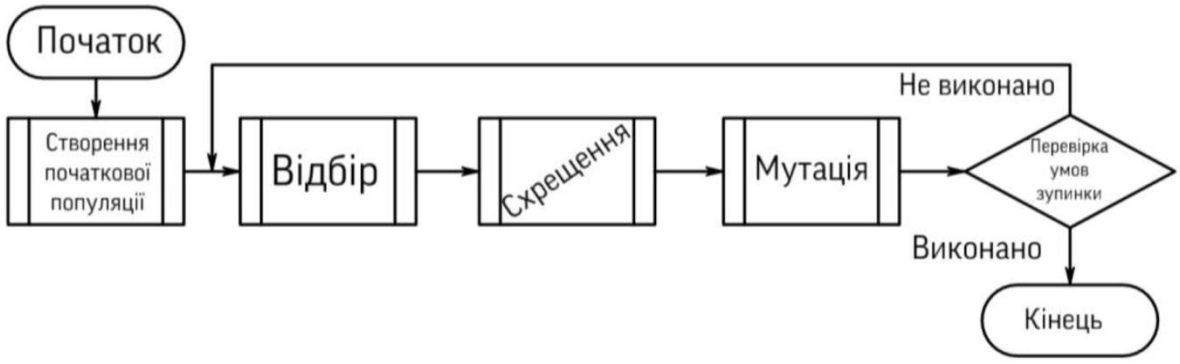


Рисунок 1.3.2

На першому кроці генерується, як правило – випадковим чином, початкова популяція (набір розв’язань). Далі моделюється розмноження. Для цього відбирається пара (розв’язань), схрещується, а отримані нащадки-особи стають новою популяцією. Ці дії повторюються доки не виконається критерій зупинки, наприклад, кількість поколінь схрещень, час виконання, обмеження на необхідну довжину шляху тощо.

7) Алгоритм колонії мурах

Мурашині алгоритми представляють собою ймовірнісну жадібну евристику, де ймовірності встановлюються, враховуючи інформацію про якість рішень, що були зроблені на попередніх кроках.

Даний алгоритм представляє собою реалізацію біонічного підходу до розв’язання задачі за результатами досліджень живих колоній мурах. Мурахи знаходять найкоротший шлях між домом та їжею, використовуючи феромон, речовину із запахом. У процесі пересування мурахи залишають його на землі та рухаються з певною ймовірністю у напрямі, що більше збагачений феромоном.

Загальний мурашиний алгоритм можна подати у наступному вигляді.

Доки не виконані умови виходу, повторювати:

1) створюємо мурах;
2) шукаємо розв’язання;
3) оновлюємо феромони;
4) додаткові дії (опціонально).

Розглянемо детальніше кожен крок у циклі, що розглядається.

На старті, за умови, що нам дано точку початку відкладання маршруту, генеруються усі мурахи, що будуть брати участь в імітації. На цьому етапі також задається початковий рівень феромона. Він ініціалізується деяким додатнім невеликим числом та рівнозначний для усіх можливих варіантів, куди можуть перейти мурахи, що потрібно, аби ймовірності переходу до другої точки не були нульовими.

На другому кроці нам потрібно обрахувати ймовірність переходу з вершини i у вершину j, яка визначається формулою

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

де $\tau_{ij}(t)$ – рівень феромону, η_{ij} – еврістична відстань, α, β – константні параметри. Параметри впливають на жадібність алгоритму, тому їх потрібно підбирати експериментально.

На третьому кроці потрібно робити оновлення феромону. Це робиться відповідно до формули

$$\tau_{ij}(t+1) = (1-\rho)\tau_{ij}(t) + \sum_{k \in \{used(i,j\}} \frac{Q}{L_k(t)},$$

у даній формулі інтенсивність випаровування феромону позначається як $\tau_{ij}(t)$, $L_k(t)$ – ціна поточного розв’язання для k-ої мурахи, Q – параметр, що має

значення порядку ціни оптимального розв'язання, тобто сам вираз $\frac{Q}{L_k(t)}$ є феромоном, що відкладається k-ою мурахою, що використовує ребро (i, j).

8) Методи розв'язання з використанням нейронних мереж

Нагадаємо, що у формулюванні задачі комівояжера ми використовуємо означення

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

при чому

$$x_{ij} \in \{0,1\},$$

$$\sum_{i=1}^{n} x_{ij} = 1, \qquad j = 1,2,\dots,n,$$

$$\sum_{j=1}^{n} x_{ij} = 1, \qquad i = 1,2,\dots,n.$$

Для розв'язання задачі комівояжера використовується матриця нейронів розміром $n \times n$. Задача буде розв'язуватись рекурентною нейронною мережею Хопфілда-Вана, яка описується диференційним рівнянням

$$\frac{\partial u_{ij}(t)}{\partial t} = -\eta \left(\sum_{k=1}^{n} x_{ij}\,(t) + \sum_{l=1}^{n} x_{ij}\,(t) - 2 \right) - \lambda c_{ij} \exp\left(-\frac{t}{\tau}\right),$$

де $x_{ij} = f\left(u_{ij}(t)\right), f(u) = \frac{1}{1+\exp(-\beta u)}$, $\beta, \eta, \lambda, \tau$ – параметри мережі.

Для зменшення ймовірності передчасного повернення до початкової стрічки i_{start} , в останній формулі замінимо вартість переходу між сусідніми пунктами (i, j) на величину

$$c_{ij}^{*} = \begin{cases} c_{ij}, j \neq i_{start}, \\ pc_{ij}, j = i_{start,} \end{cases}$$

де $p \gg 1$ – коефіцієнт штрафу.

З урахуванням заміни маємо диференційне рівняння вигляду

$$u_{ij}^{T+1} - u_{ij}^{T} = \Delta t \left[-\eta \left(\sum_{k=1}^{n} x_{ik}^{T} + \sum_{l=1}^{n} x_{il}^{T} - 2 \right) - \lambda c_{ij}^{*} \exp\left(-\frac{T}{\tau} \right) \right],$$

яке розв'язується ітераційним методом до виконання умови

$$\sum_{k=1}^{n} x_{ik}^{T} + \sum_{l=1}^{n} x_{il}^{T} - 2 < \varepsilon,$$

тут T є номером ітерації, що обраховується, а ε відповідає за задану точність виконання умов обмежень, що задані у формулюванні задачі комівояжера.

Далі виконується перетворення отриманої матриці розв'язань $\|x_{ij}\|$ із використанням принципу «Переможець забирає все»:

1) $i = i_{start}$;
2) для i-тої стрічки матриці знаходиться максимальний елемент $x_{ij_{max}}$, тут j_{max} – номер стовпчика з максимальним елементом; виконується перетворення $x_{ij_{max}} = 1$, усі інші елементи стрічки i та стовпчика j_{max} обертаються у нулі;
3) далі відбувається перехід до стрічки j_{max} та описані дії повторюються, доки не відбудеться повернення до стрічки i_{start}, яке буде означати завершення побудови циклу.

Якщо повернення до стрічки i_{start} стається раніше, ніж у матриці розв'язань $\|x_{ij}\|$ значення 1 отримали рівно n елементів, то це означає, що довжина побудованого циклу менша за n і тому повторюються всі описані дії. Отриманий у результаті цикл оптимізується з використанням локального пошуку 2-opt.

Зрозуміло, що у даному розділі ми розглянули далеко не всі існуючі алгоритми. Існує чимало оптимізацій наведених методів, серед яких метод градієнтного спуску, метод імітації віджигу, метод пошук оптимальних траекторій бджолами (він подібний до алгоритму колонії мурах; спершу на пошуки нектару вилітає деяка кількість бджіл на розслідування, після чого повертаються та кажуть про результати. На місця нектару відправляють бджіл-збірників, а далі бджоли-розслідники знову відправляються на пошук). Існує також чимало варіацій розв’язання задачі із використанням нейронних мереж (ми розглянули варіацію використання мережі Хопфілда, а є також, наприклад, самоорганізаційна мережа Кохонена).

РОЗДІЛ 2. ПРАКТИЧНА ЧАСТИНА

1 Постановка задачі

Розглядатимемо дві замкнені задачі комівояжера. Вони будуть відрізнятись між собою порядком значень, які будуть присвоюватись кожному ребру, а також кількості величин, що будуть розглядатись. До обох задач будемо застосовувати різні методи для різної кількості даних для оцінки різниці між швидкістю та точністю роботи алгоритмів.

Задача 1. Мандрівник столицями Європи

Мандрівник вирішив відвідати більшість столиць Європейських країн. У нього необмежений бюджет на подорож, але він хоче якомога менше часу проводити у перельотах та переїздах, щоб більше часу проводити у містах та присвячувати ознайомленню з місцевою культурою.

Потрібно допомогти мандрівнику побудувати найкоротший маршрут між столицями, які хоче відвідати мандрівник.

Задача 2. Пам’ятки Києва

Студент факультету кібернетики спланував проїхати по усіх цікавих будинках, вулицях та пам’ятках міста Київ, про які він прочитав у соцмережах та дізнався від родичів та друзів. Він також врахував, що пройти їх пішки за день буде неможливо, а їздити громадським транспортом – незручно та важко. У якості транспортного засобу він обрав електричний самокат, щоправда, взяв його в оренду. Вартість проїзду пропорційна відстані між точками на маршруті.

Студент завжди починає свій маршрут біля корпусу факультету комп’ютерних наук та кібернетики Київського національного університету імені

Тараса Шевченка і там же планує закінчити свій маршрут. Він хоче відвідати максимальну кількість пам’яток та мінімізувати вартість оренди самокату.

1.1 Тестові дані

Для задач будемо використовувати дані у різних форматах.

Дані для першої задачі будуть подаватись у вигляді двомірної матриці, що містить попарні відстані між обраними столицями. До максимального маршруту будуть входити 32 міста, кожен починається та закінчується у столиці Нідерландів – Амстердамі.

До переліку міст, які входять до бажаного мандрівником маршруту, входять Анкара (Туреччина), Афіни (Греція), Братислава (Словаччина), Белград (Сербія), Берлін (Німеччина), Валетта (Мальта), Брюссель (Бельгія), Будапешт (Угорщина), Бухарест (Румунія), Дублін (Ірландія), Люксембург (Люксембург), Таллінн (Естонія), Гельсінкі (Фінляндія), Вадуц (Ліхтенштейн), Київ (Україна), Копенгаген (Данія), Лісабон (Португалія), Лондон (Велика Британія), Мадрид (Іспанія), Монако (Монако), Москва (Російська Федерація), Осло (Норвегія), Кишинів (Молдова), Париж (Франція), Прага (Чехія), Рим (Італія), Софія (Болгарія), Стокгольм (Швеція), Відень (Австрія) та Берн (Швейцарія), при чому при зменшенні кількості міст у маршруті залишаються ті міста, що раніше задані у даному списку.

Повна таблиця із даними для задачі про мандрівника подана у додатку А – відстані між містами.

Дані для другої задачі подаються у вигляді координат точок, які мають бути відвідані у маршруті. Подальше їхнє переведення у формат двомірної матриці буде виконуватись за допомогою програмного забезпечення. Даний формат введення простіший для використання на практиці, але застосовний лише

у тих випадках, коли вартість переїзду між двома пунктами пропорційна лінійній відстані між точками, яку можна виразити через їхні координати.

На рисунку 2.1.1 зображені точки, які входять до маршруту.

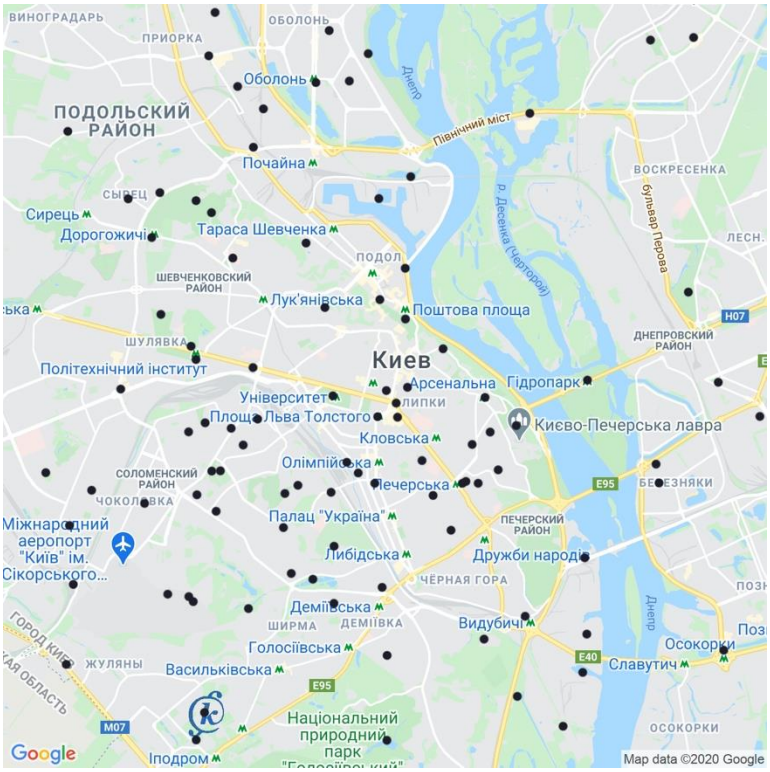


Рисунок 2.1.1

У додатку Б до диплому міститься перелік усіх координат, що мають увійти до маршруту (всього – 100 точок на мапі міста Київ).

1.2 Огляд програмного забезпечення

Для тестування швидкодії роботи алгоритмів використовується персональний комп’ютер з процесором Intel(R) Core(TM) i7-4500U CPU @ 1.80GHz, 2401 МГц.

Алгоритми розроблені у середовищі програмування Visual Studio 2016, мова розробки – C++. Використовуються такі стандартні бібліотеки, як cmath, fstream, iostream, ctime, random, vector, algorithm.

Для візуалізації отриманих результатів використовується персональний комп’ютер з процесором 1,2 GHz Quad-Core Intel Core i7, середовище розробки Visual Studio Code. Мова програмування – Python 3.9.4. Використовуються бібліотеки PIL (для завантаження ілюстрацій для візуалізації та виведення результуючих маршрутів), mathplotlib.pyplot для побудови безпосередньо маршруту між заданими точками.

2 Алгоритми, що використовуються

Для розв’язання задач під час роботи над теоретичною частиною було виділено 4 алгоритми, серед яких один точний та три наближених, швидкодію та ефективність яких буде перевірено на досліджуваних масивах даних.

Розглянемо реалізацію наступних способів розв’язання задачі комівояжера:

1. Метод повного перебору. Точний метод, будемо досліджувати, коли він насправді може дати актуальний вдалий результат.
2. Класичний жадібний метод розв’язання – пошук найближчого сусіда. Найшвидший з досліджуваних методів.
3. Метод випадкового пошуку– наближений метод розв’язання, який може давати різні результати роботи за однакових вхідних даних.
4. Алгоритм колонії мурах – наближений алгоритм із моделюванням поведінки мурах при пошуку найшвидшого шляху до джерела їжі.

   Дані алгоритми будуть реалізовані мовою програмування C++, при виконанні результатом роботи програми буде порядок відвідування точок у маршруті з мінімальною вартістю, власне вартість (довжина) маршруту, а також час, який був витрачений на обрахунок даного маршруту у певному методі.

Також розглянемо алгоритм візуалізації отриманих результатів з використанням мови Python 3.9.4 для покрокової побудови маршруту.

2.1 Реалізація алгоритмів

1) Алгоритм повного перебору

Для даного алгоритму будемо застосовувати допоміжні функції – swap, що замінює значення двох змінних, та NextSet, який буде генерувати послідовність обходу міст.

```
void swap(int *a, int i, int j)
{
    int s = a[i];
    a[i] = a[j];
    a[j] = s;
}

bool NextSet(int *a, int n)
{
    int j = n - 2;
    while (j != -1 && a[j] >= a[j + 1]) j--;
    if (j == -1)
        return false;
    int k = n - 1;
    while (a[j] >= a[k]) k--;
    swap(a, j, k);
    int l = j + 1, r = n - 1;
    while (l < r)
        swap(a, l++, r--);
    return true;
}
```

Відповідно, запишемо основну функцію для алгоритму повного перебору. Ми будемо генерувати відповідні маршруту до знайдених згенерованих послідовностей та для кожної обраховувати дистанцію між точками по порядку, щоб знайти мінімальну, для цього виділимо окремо функцію перевірки відстані.

```
int DistanceCheck(int *a, int n)
{
    int dist = Task1[0][a[0]];
    for (int i = 1; i < n; ++i)
    {
```

```
        dist+=Task1[a[i-1]][a[i]];
    }
    dist += Task1[a[n - 1]][0];
    return dist;
}
```

Відповідно перебір буде реалізовуватись через допоміжні функції простим циклом поєднання.

```
    int mindist = DistanceCheck(a, n);

    while (NextSet(a, n))
    {
        int thisdist = DistanceCheck(a, n);
        if (thisdist < mindist)
        {
            mindist = thisdist;
            for (int i = 0; i < n; ++i)
            {
                arr[i]=a[i];
            }
        }
    }
```

2) Метод найближчого сусіда

Для реалізації даного алгоритму будемо будувати маршрут у додаткову матрицю такого ж розміру, як вихідна. Будемо записувати 0, якщо між точками немає переходу на відповідному кроці, -1, якщо є (у кожній стрічці матриці буде рівно одна величина, що рівна -1). У кожній стрічці вихідної матриці шукаємо найменший елемент, але лише серед тих координат, які ще не були відвідані попередньо.

```
int arr[n][n];
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            arr[i][j] = Task1[i][j];

    int i = 0;
    int t = 0;
    while (t < n-1)
    {
        int min = 999999;
        int new_min_i = 0;
```

```
    for (int j = 0; j < n; ++j)
    {
        int k = minnot0(arr[i][j], min);
        if (k == 1)
        {
            arr[i][new_min_i] = 0;
            new_min_i = j;
            min = arr[i][j];
        }
        else
        {
            arr[i][j] = 0;
        }
    }

    for (int j = 0; j < n; ++j)
    {
        if (arr[i][j] > 0)
        {
            arr[i][j] = -1;
            towns[t] = j;
        }
        if (arr[j][i]!=-1)
            arr[j][i] = 0;
    }

    arr[new_min_i][i] = 0;
    i = new_min_i;

    ++t;
}

cout << i << endl;
arr[i][0] = -1;
```

3) Метод випадкового пошуку

У даному методі, відповідно до розглянутої у першому розділі теорії, будемо використовувати у якості індикатора часу зупинки роботи алгоритму кількість ітерацій k, яка задається при зверненні до функції, відповідальної за реалізацію методу випадкового пошуку.

Ми будемо генерувати випадкову послідовність міст, що будуть відвідані. Якщо замкнений маршрут, що проходить через відповідні точки буде меншим за

контрольний, будемо оновлювати значення найменшого маршруту. Повторюємо, доки виконується умова продовження (не досягнута задана наперед кількість ітерацій). Кількість ітерацій буде визначатись верхнім обмеженням на припустимий час для роботи програми.

```
for (int j = 1; j < k; ++j)
  {
     std::vector<int> array(n);

     for (int i = 0; i < n; ++i)
        array[i] = i+1;

     std::random_shuffle(array.begin(), array.end());

     int distance = Task1[0][array[0]]+Task[array[0]][array[n-1]];

     for (int i = 0; i < n - 1; ++i)
     {
        distance += Task[array[i]][array[i + 1]];
     }

     if (distance < mindist)
     {
        mindist = distance;
        for (int i = 0; i < n; ++i)
        {
           aresult[i] = array[i];
        }
     }
  }
```

4) Алгоритм колонії мурах

Алгоритм та теорія моделювання поведінки у колонії мурах було теоретично розглянуто у першому розділі дипломної роботи, а для реалізації програмного забезпечення будемо користуватись принципами об’єктно-орієнтовного програмування – створимо класи, які відповідають за імітацію поведінки для розв’язання задачі комівояжера.

Перший клас – клас безпосередньо мурахи UAnt (Unique – унікальна особина), який відповідає за поведінку при виборі маршруту для кожної особини:

```
class UAnt {
public:
   vector<int> onetrail;
   set<int> avail;
   double alpha, beta;       // рівні феромону, видимості
   UAnt(double alphaC, double betaC) {
      alpha = alphaC;
      beta = betaC;
      onetrail.push_back(1);
      for(int k=2;k<=d.N; ++k)
      {
         avail.insert(k);
      }
   }
   void reset() {
      vector<int> Len;
      Len.push_back(1);
      onetrail = Len;
      for(int v=2; v<=d.N; ++v)
      {
         avail.insert(v);
      }
   }
   void deposit_A()

      double tour_Cost = d.tour_Cost(onetrail);
      int Q = 100;
      double deposit_Amount = Q / tour_Cost;
      int l = onetrail.size();
      l--;
      for (int i=0; i < l; ++i)
      {
         d.T[onetrail[i]][onetrail[i+1]] += deposit_Amount;
      }
      d.T[onetrail[l]][onetrail[0]] += deposit_Amount;
   }

   vector<int> stop() {
      deposit_A();
      vector<int> temp_a = onetrail;
      reset();
      return temp_a;
   }

   void onestep()
   {
      int current_City = onetrail[onetrail.size()-1];
      double norm_C = probabilityNorm_C(current_City);
```

```
        double p;
        double gp;
        bool moving = false;
        double highest_Prob = 0;
        double city_Highest = 0;
        for (set<int>::iterator l=avail.begin(); l != avail.end() ; ++l) {
            p = movingProbability(current_City,*i,norm_C);
            if (p > highest_Prob) {
                city_Highest = *l;
                highest_Prob = p;
            }
            gp = get_Rand();
            if (gp <= p) {
                moving = true;
                onetrail.push_back(*l);
                avail.erase(l);
                break;
            }
        }
        if(!moving)
        {
            onetrail.push_back(city_Highest);
            avail.erase(city_Highest);
        }
    }

    double get_Rand() {
            double p = (1.0 / (RAND_MAX + 1.0)*(rand()));
        return p;
    }
    double movingProbability( int i, int j, double n)
    {
        double prob = (pow(d.T[i][j], alpha))*(pow(d.visibility[i][j],beta));
        prob /= n;
        return prob;
    }

    double probabilityNorm_C(int current_City) {
        int size = avail.size();
        double n_c = 0*1.0;
        for(set<int>::iterator l=avail.begin(); l != avail.end() ; ++l)
        {
            n_c += (pow(d.T[current_City][*l],alpha))*(pow(d.visibility[current_City][*l],beta));
        }
        return n_c;
    }
};
```

Тепер наведемо ще один клас, який буде відповідати вже за поведінку колонії, а не кожної особини окремо, саме з ним буде відбуватись моделювання алгоритму для розв’язання задач.

```
class AntColony {
public:
   int N;                // кількість точок у маршруті
   int M;                // кількість мурах у моделюванні
   vector<UAnt> ant;          // мурахи – звернення до нашого класу
   double evaporation, alpha, beta;    // рівень випаровування, рівень феромону, рівень видимості

   AntColony(double alphaC, double betaC, double evapropationC)
   {
      alpha = alphaC;
      beta = betaC;
      evaporation = evapropationC;
      M = 51;     // мурахи
      N = d.N;
      for (int i_c=0; i_c < M; ++i_c)
      {
         UAnt a(alpha,beta);
         ant.push_back(a);
      }
   }

   void go() {
      vector<int> ROUTE;
      double min__Tour;
      double tour_C;
      for (int n = 0; n < M ; ++n)
      {
         for (int z = 0 ; z < (N-1) ; ++z)
         {
            for (int l = 0 ; l < M ; ++l)
            {
               ant[l].onestep();
            }
         }
         for (int r = 0 ; r < M ; ++r)
         {
            vector<int> pC = Uant[r].stop();
            if(!ROUTE.size())
            {
               ROUTE = pC;
               min__Tour = d.tour_Cost(p);
               continue;
```

```
            }
            tourC = d.tour_Cost(p);
            if(tourC < min__Tour)
            {
                min__Tour = tour_C;
                ROUTE = pC;
            }
        }
        for (int i_c=1; i_c<=N; ++i)
        {
            for (int j_c=1; j_c<=N; ++j)
            {
                d.T[i_c][j_c] *= evaporation;
            }
        }
    }
  }
};
```

Безпосередньо рівень параметрів інтенсивності феромону, швидкості випаровування, кількості мурах можна змінювати та задавати в основній частині програми або функції, що проводить моделювання із використанням властивостей класу.

5) Допоміжний алгоритм для форматування даних (переведення у таблицю) для другої задачі

Оскільки дані у другій задачі подаються у вигляді координат для 100 точок у маршруті, а алгоритми працюють із матрицями відстаней, напишемо дві нескладні функції, які будуть математично обраховувати дані величини для кожної пари точок і записувати у двомірний масив Task2. KyivPlaces – статична константа, яка відповідає за кількість міст, що входять до маршруту (за умовою – 100).

Спершу задаємо допоміжну функцію для обрахунку відстані між двома точками.

```
double distance(double x1, double y1, double x2, double y2)
{
    return(sqrt((x1 - x2)*(x1 - x2) + (y1 - y2)*(y1 - y2)));
}
```

Тепер наведемо безпосередньо функцію, що створює необхідний масив даних для подальшого застосування алгоритмів пошуку найкоротшого маршруту.

```
void task2_data()
{
    for (int i = 0; i < KyivPlaces; i++)
    {
        for (int j = 0; j < KyivPlaces; j++)
        {
            if (i == j)
                Task[i][j] = 0;
            else
            {
                Task[i][j] = distance(Task_coord[i][0], Task_coord[i][1], Task_coord[j][0], Task_coord[j][1]);
            }
        }
    }
}
```

6) Обрахунок часу роботи алгоритму/методу

Оскільки ключовою метрикою порівняння роботи алгоритмів для наших задач крім довжини маршруту є швидкість роботи алгоритмів, надамо код, що буде обраховувати швидкість роботи функції, що відповідає за певну частину програми (один з методів/алгоритмів).

```
unsigned int start_time = (clock());

*** ОСНОВНИЙ КОД ФУНКЦІЇ ***

unsigned int end_time = (clock());
    unsigned int search_time = end_time - start_time;
```

Відповідно, час роботи програми буде видаватись у мілісекундах, а рахується через точний поточний час на приладі.

7) Візуалізація маршрутів-результатів

Результатом роботи програми буде зображення формату .png, маршрут, що побудований у відповідному порядку на заданих координатах на фоні, що задається картинкою формату .jpg.

Першим кроком імпортуємо необхідні бібліотеки:

```
from PIL import Image, ImageDraw
import matplotlib.pyplot as plt
import numpy as np
```

Тепер наведемо код роботи основної функції drawing, що безпосередньо будує переходи між точками маршруту:

```
def drawing(img, coordinates, path):
    draw = ImageDraw.Draw(img)
    for i in range(len(path) - 1):
        draw.line([coordinates[path[i] - 1], coordinates[path[i + 1] - 1]], width=3, fill=128)
        plt.imshow(np.asarray(img))
        plt.axis('off')
        plt.show()
    img.save("res.png")
```

Для виклику функції задаються координати точок, а також безпосередньо маршрут, де перша і кінцева точки – співпадають. Програма не зобов'язує проходити усі задані точки. Наведемо результати роботи порядкового виведення точок із заданих даних для задачі 1 про мандрівника (рисунок 2.2.А) та задачі 2 про маршрут Києвом (рисунок 2.2.Б). Звернемо також увагу, що побудовані маршрути є замкненими.

Для подальшого порівняння також обрахуємо та наведемо контрольні вартості маршрутів, що побудовані для заданого наперед порядку відвідування міст або точок на мапі Києва.

Рисунок 3.2.1

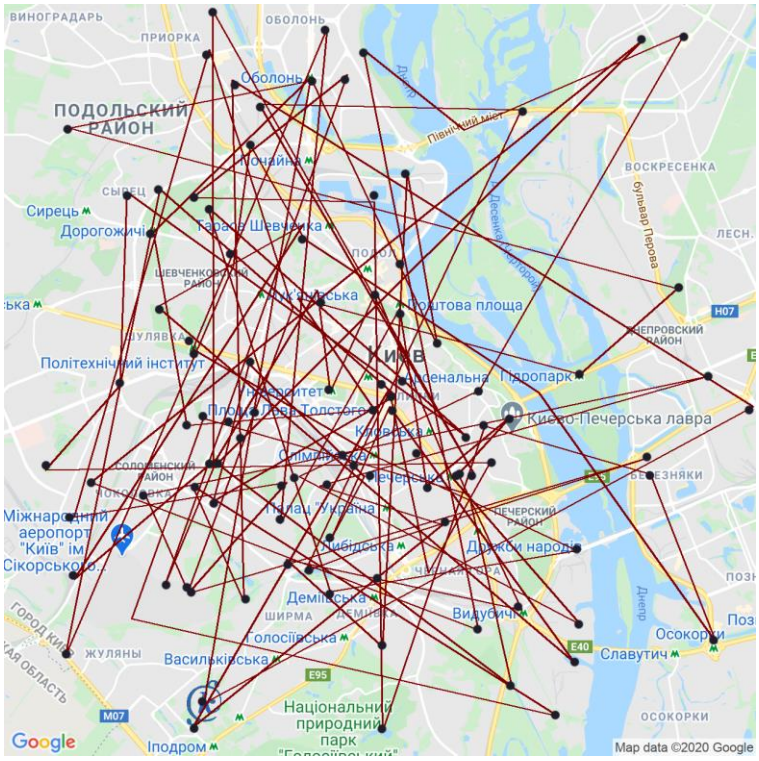


Рисунок 2.2.2

Для задачі 1 контрольна довжина маршруту (рисунок 2.2.1) становить 50793 км.

Для задачі 2 контрольна довжина маршруту (рисунок 2.2.2) становить: 314 км.

3 Результати роботи програми

Для задачі 1 знайдемо результати роботи алгоритмів для різної кількості столиць у маршруті, щоб знайти максимально можливу кількість міст, що будуть задовольняти умову прийнятного часу для точного алгоритму повного перебору. Алгоритми випадкового пошуку та колонії мурах будемо запускати п’ять разів, адже результати можуть змінюватись в залежності від кожного запуску та будемо брати найкращий результат із отриманих п’яти.

Експериментально було визначено, що найбільша розмірність, що розв’язує задачу комівояжера з верхнім обмеженням у 10 хвилин методом повного перебору для комп’ютера з процесором Intel(R) Core(TM) i7-4500U CPU @ 1.80GHz, 2401 МГц становить 13 міст, для 14 час виконання перевищує 10 хвилин. Наведемо також результати роботи інших алгоритмів для даної кількості міст у таблиці 2.3.1.

Повний перебір	**Жадібний**	**Випадкові числа**	**Колонія мурах**
8112 мс	2 мс	7012 мс	5612 мс
9216 км	12917 км	9216 км	9216 км
1, 11, 8, 12, 4, 9, 5, 7, 3, 2, 10, 13, 6, 1 – рис. 2.3.1 (а)	1, 8, 12, 6, 4, 9, 5, 10, 2, 7, 3, 13, 11, 1 – рис. 2.3.1 (б)	1, 11, 8, 12, 4, 9, 5, 7, 3, 2, 10, 13, 6, 1 – рис. 2.3.1 (в)	1, 11, 8, 12, 4, 9, 5, 7, 3, 2, 10, 13, 6, 1 – рис. 2.3.1 (г)

Таблиця 2.3.1

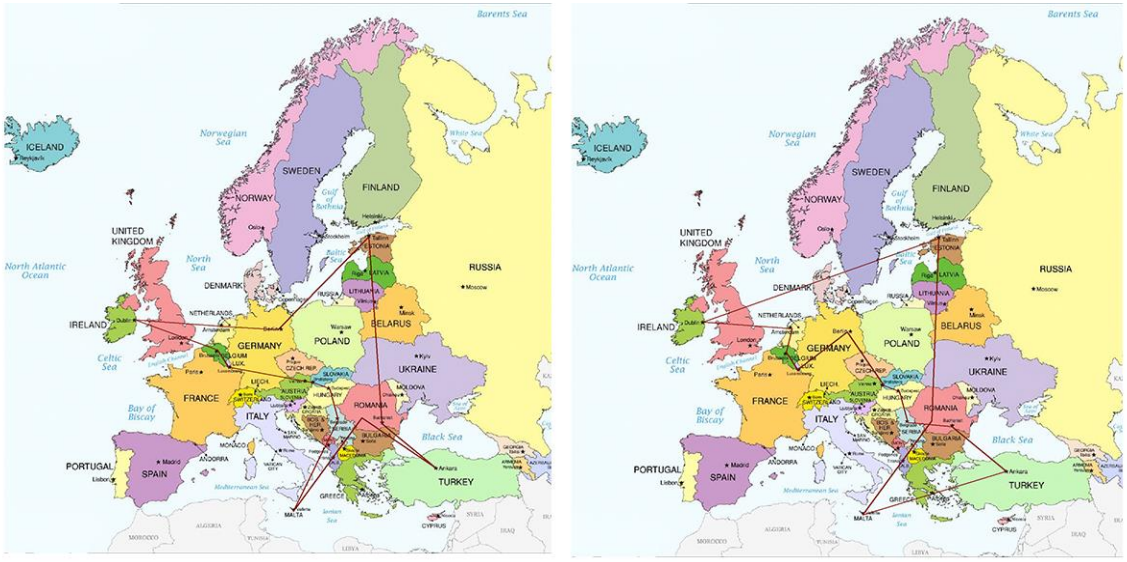
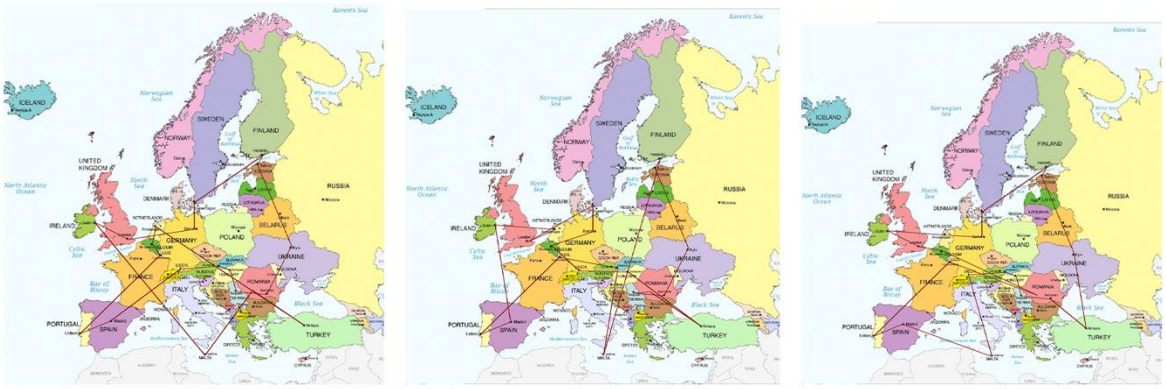


Рисунок 2.3.1

Бачимо, що для даної кількості найкраще спрацювали алгоритми повного перебору та колонії мурах, для жадібного та випадкових чисел отримали неоптимальні маршрути, що відрізняються від оптимальних на 40%. Наведемо результати роботи алгоритмів для 20 столиць у маршруті у таблиці 2.3.2.

Жадібний	Випадкові числа	Колонія мурах
4 мс	13818 мс	5612 мс
17656 км	17835 км	16344 км
8, 12, 18, 20, 15, 4, 9, 5, 10, 16, 13, 14, 17, 6, 19, 11, 7, 3, 2, 1 – рис. 2.3.2 (а)	1, 11, 20, 18, 12, 4, 7, 14, 13, 16, 10, 5, 3, 2, 9, 15, 8, 6, 18, 19, 1– рис. 2.3.2 (б)	1, 11, 19, 8, 3, 12, 18, 20, 7, 15, 4, 9, 5, 10, 2, 16, 13, 14, 17, 6, 1 – рис. 2.3.2 (в)

Таблиця 2.3.2



а) Жадібний **б) Випадкові числа** **в) Колонія мурах**

Рисунок 2.3.2

Наведемо результати роботи алгоритмів для 32 міст (усі, що фігурують в умові задачі) у таблиці 2.3.3.

Жадібний	Випадкові числа	Колонія мурах
6 мс	40016 мс	7612 мс
23012 км	20056 км	20056 км
1, 8, 12, 18, 20, 25, 19, 11, 15, 32, 21, 27, 7, 3, 2, 24, 10, 28, 5, 9, 4, 31, 26, 6, 17, 23, 29, 14, 13, 30, 16, 22, 1 – рис. 2.3.3 (а)	1, 11, 19, 8, 12, 25, 18, 21, 32, 27, 7, 3, 14, 18, 24, 28, 2, 16, 24, 19, 5, 29, 9, 4, 31, 6, 30, 22, 14, 23, 17, 1 – рис. 2.3.3 (б)	1, 11, 19, 8, 12, 25, 18, 21, 32, 27, 7, 3, 14, 18, 24, 28, 2, 16, 24, 19, 5, 29, 9, 4, 31, 6, 30, 22, 14, 23, 17, 1 – рис. 2.3.3 (в)

Таблиця 2.3.3

а) Жадібний **б) Випадкові числа** **в) Колонія мурах**

Рисунок 2.3.3

Помітимо, що найкращі результати дає алгоритм колонії мурах, близькі до оптимальних. Алгоритм із використанням випадкових чисел іноді дає гірший результат, ніж жадібний, але в кінцевому результаті надав оптимальний маршрут, проте це зайняло більш як у 5 разів більше часу, ніж для алгоритму колонії мурах.

Тепер перейдемо до результатів роботи алгоритмів для другої задачі. Оскільки у ній до кінцевого маршруту входить рівно 100 точок, а границю припустимого значення кількості відвідуваних точок ми знайшли при дослідженні першої задачі, будемо порівнювати результати роботи методу пошуку найближчого сусіда, методу випадкового пошуку, алгоритму колонії мурах. Результат обрахунків – довжина маршруту та час, витрачений на обрахунок, надані у таблиці 2.3.4.

Жадібний	**Випадкові числа**	**Колонія мурах**
24 мс	419200 мс (~7 хвилин)	178000 мс (~3 хвилин)
208 км	156 км	156 км

Рисунок 2.3.4

Візуалізацію маршруту для жадібного алгоритму зобразимо на рисунку 2.3.4. Звернемо увагу, що сталась найчастіша проблема, пов’язана з

використанням даного алгоритму – оскільки завжди обирається найближча точка, кінцева – дуже віддалена від початкової, маршрут міг би бути значно оптимальнішим.

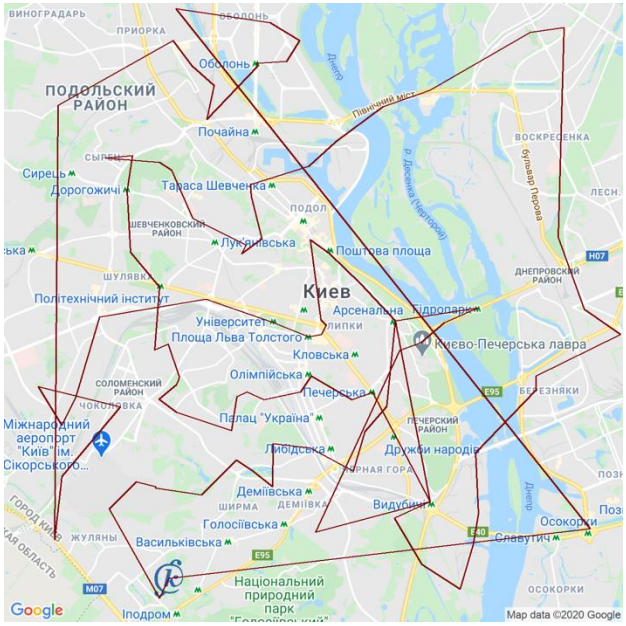


Рисунок 2.3.4

Алгоритми генерування випадкових маршрутів та алгоритм колонії мурах дали аналогічний результат за різний час, покажемо маршрут на рисунку 2.3.5.

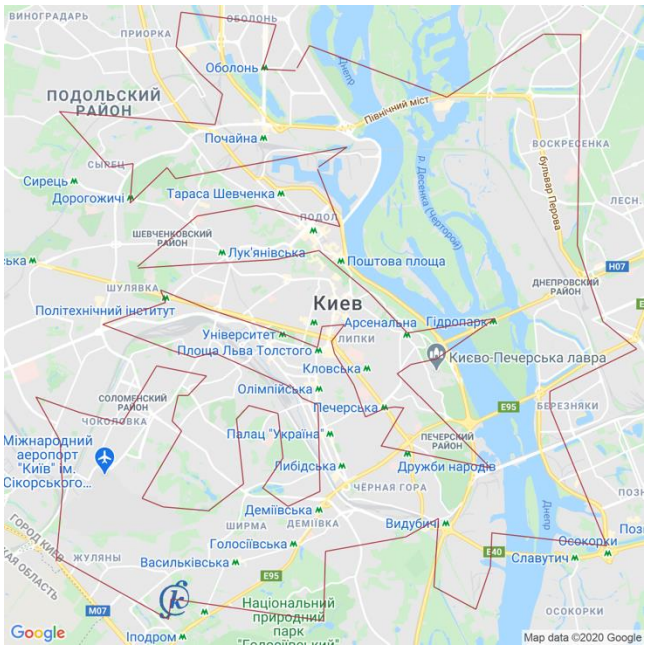


Рисунок 2.3.5

ВИСНОВКИ

У роботі була досліджена актуальність пошуку розв’язань для задачі комівояжера, розглянуто історичні аспекти формулювання проблеми транспортування та побудови найдешевшого маршруту та варіації даного завдання, серед яких, наприклад, задача Гамільтона та задача про обхід шахової дошки.

Було теоретично розглянуто та описано точні (алгоритм повного перебору, метод динамічного програмування, метод гілок та меж) та наближені (метод найближчого сусіда, метод включення найближчого сусіда, метод найдешевшого включення, метод найкоротшого острову, метод випадкового пошуку, генетичні алгоритми, алгоритм колонії мурах, розв’язання за допомогою нейронних мереж) способи знаходження оптимального за певним критерієм маршруту. Такі алгоритми, як повний перебір, метод найближчого сусіда, метод випадкового пошуку та моделювання поведінки колонії мурах були реалізовані із застосуванням мови програмування C++ та візуалізовані із використанням мови Python.

Результати роботи програми для двох задач із різними вхідними дозволяють зробити наступні висновки:

1. Для невеликої кількості точок маршруту $n < 10$ можна використовувати алгоритм повного перебору, оскільки він дає точну відповідь та за такої довжини маршруту знаходить результат досить швидко.
2. Жадібні алгоритми, як-от метод найближчого сусіда, дають найшвидшу відповідь, проте їх варто застосовувати лише у випадках, коли вартість помилки прямує до нуля, а всі точки знаходяться безпосередньо близько одна до одної. Може дати кращий результат у порівнянні з іншими

наближеними методами при невеликих відстанях між координатами та значному обмеженні на максимальний можливий час обрахунку.

3. Алгоритми з використанням генерації випадкових послідовностей і порівняння поточного згенерованого маршруту та попереднього мінімального є простішими у реалізації за генетичні алгоритми та алгоритми, що використовують нейронні мережі, проте працюють повільніше та за сильних обмежень на час обрахунку можуть давати поганий, дорогий результат, іноді навіть гірший за знайдений жадібними алгоритмами.
4. Для задач, що розглядались, найкраще показав себе алгоритм колонії мурах. З ним було отримано оптимальні серед знайдених маршрутів для кожного з контрольних обмежень для обох сформульованих задач.

Виконання візуалізації отриманих результатів та порівняння отриманих маршрутів для кожного методу дозволяє наочно запевнитись у тому, що знайдені шляхи є оптимальними за своїми критеріями.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Мудров В.И. Задача о коммивояжере. – М.: Знание, 1969.
2. Goldschmidt O., Laugier A., Olinick E.V. SONET/SDH ring assignment with capacity constraints // Discrete Applied Mathematics, 2003. Vol. 129. P. 99–128.
3. Ахо, А. Структуры данных и алгоритмы / А. Ахо, Дж. Хопкрофт, Д. Ульман. – М.: Издательский дом «Вильямс», 2001. – 384 с.
4. Гэри, М. Вычислительные машины и труднорешаемые задачи / М. Гэри, Д. Джонсон. – М.: Мир, 1982. – 419 с.
5. A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems, ст. 497-520.
6. The traveling salesman problem and its variations / G. Gutin, A. Punnen, (eds.) // Combinatorial optimization. – Nowell: Kluwer, 2002.
7. Bianchi L., Gambardella L.M., Dorigo M. An ant colony optimization approach to the probabilistic traveling salesman problem // Proceedings of PPSN-VII, Sev- enth international conference on parallel problem solving from nature. – Berlin.: Springer, 2002.
8. Тарков М.С. Решение задачи коммивояжера с использованием рекуррентной ней- ронной сети // Сиб. журн. вычисл. математики / РАН. Сиб. отд-ние. — Новосибирск, 2015. — Т. 18, No 3. — С. 337–347.
9. S. Lin & B. W. Kernighan, “An Effective Heuristic Algorithm for the Traveling-Salesman Problem”, Oper. Res. 21, 498-516 (1973).
10. Riadi, I.C.J & Nefti-Meziani, 2011, Cognitive Ant Colony Optimization: A New Framework In Swarm Intelligence, Proceeding of the 2nd Computing, Science and Engineering Postgraduate Research Doctoral School Conference, 2011
11. Штовба С.Д. Муравьиные алгоритмы // Exponenta Pro. Математика в приложениях. – 2003. – №4. – С. 70–75

12. Ежов, А. А. Нейрокомпьютинг и его применения в экономике и бизнесе/ Ежов А. А. Шумский С. А. - Москва : Национальный Открытый Университет "ИНТУИТ", 2016. - Текст : электронный // ЭБС "Консультант студента"
13. Колесников А.В., Кириков И.А., Листопад С.В., Румовская С.Б., Доманицкий А.А. Решение сложных задач коммивояжера методами функциональных гибридных интеллектуальных систем / Под ред. А.В. Колесникова. — М: ИПИ РАН, 2011. — 295 с., ил. – ISBN 978-5-902030-88-1

ДОДАТОК А. Відстані між містами

	Amste	Ankar	Athen	Bratis	Belgr	Berlin	Valett	Brusse	Budap	Bucha	Dublir	Luxerr	Tallin	Helsin	Vaduz	Kyiv, U	Copen	Lisbor	Londo	Madri	Mona	Mosc	Oslo,	Chisin	Paris,	Prahu	Rome	Sofia,	Stockl	Warsz	Vienn	Berne
Amsterdam, Netherlanc	0	3210	3070	1129	1831	685	2400	198	1442	2328	878	369	1601	1860	780	2070	803	2267	344	1735	1194	2541	1397	1991	475	956	1766	2213	1427	1275	1177	880
Ankara, Tyrkey	3210	0	1644	1902	1379	2696	1515	3165	1774	1168	4073	2719	2552	3444	2401	2270	3157	4813	3539	4169	2600	3086	3736	941	3198	2348	2416	997	3766	2448	2039	2669
Athens, Greece	3070	644	0	1540	1239	2556	718	3025	1634	1299	3933	2395	2870	3416	1799	2401	3017	4673	3399	4029	1843	3217	3596	1326	3058	2308	1417	872	3626	2308	1899	2529
Bratislava, Slovakia	1129	1902	1540	0	1153	626	1589	1090	184	944	1959	930	1518	1600	673	1164	1047	2713	1445	2200	1101	1810	1573	997	1221	326	972	919	1413	606	59	845
Belgrad, Serbia	1831	1379	1239	1153	0	1317	1456	1786	395	713	2694	1411	1880	2177	1010	1331	1778	3434	2160	2790	1312	2147	2357	897	1819	969	1321	382	2387	1069	660	1290
Berlin, Germany	685	2696	2556	626	1317	0	2330	789	922	1771	1530	688	1248	1430	742	1385	461	2892	996	2360	1288	1856	1040	1438	1100	348	1529	1699	1070	590	657	845
Valetta, Malta	2400	1515	718	1589	1456	2330	0	2272	1724	1819	3035	2080	3288	3375	1658	2260	2711	2799	2512	2210	1201	3420	3201	2246	2131	2052	912	1265	3102	2368	1598	1735
Brussels, Belgium	198	3165	3025	1090	1786	789	2272	0	1397	2283	908	217	1673	1952	657	2174	968	2088	374	1556	1031	2646	1562	2101	296	911	1545	2168	1592	1379	1132	659
Budapest, Hungary	1442	1774	1634	184	395	922	1724	1397	0	886	2305	1111	1585	1782	849	1137	1383	3241	1771	2709	1195	1953	1962	909	1561	574	1349	777	1992	674	265	1013
Bucharest, Romania	2328	1168	1299	944	713	1771	1819	2283	886	0	3191	1603	1918	2276	1606	1102	2134	4147	2657	3503	1782	1918	2734	425	2447	1460	2034	427	2764	1181	1151	2003
Dublin, Ireland	878	4073	3933	1959	2694	1530	3035	908	2305	3191	0	1034	2747	2665	1488	2915	1681	2791	534	2259	1651	3386	2275	2922	909	1819	2436	3076	7305	2120	2040	1567
Luxembourg, Luxembou	369	2719	2395	930	1411	688	2080	217	1111	1603	1034	0	1837	2014	440	1905	904	139	537	1454	837	2454	1338	2000	320	670	1221	1833	1526	1209	874	377
Tallin, Estonia	1601	2552	2870	1518	1880	1248	3288	1673	1585	1918	2747	1837	0	86	1965	1216	1058	3877	2259	3377	2510	1002	876	1647	2241	1428	2540	2216	430	934	1533	2088
Helsinki, Finland	1860	3444	3416	1600	2177	1430	3375	1952	1782	2276	2665	2014	86	0	2050	1620	984	4055	2131	3523	2397	1141	914	1733	2263	1739	3003	2559	360	1108	1787	2223
Vaduz, Liechtenstein	780	2401	1799	673	1010	742	1658	657	849	1606	1488	440	1965	2050	0	1787	1009	2080	961	1507	546	2395	1594	1651	643	556	854	1522	1704	1135	611	192
Kyiv, Ukraine	2070	2270	2401	1164	1331	1385	2260	2174	1137	1102	2915	1905	1216	1620	1787	0	1748	4227	2381	3745	2250	816	2348	463	2485	1401	2486	1529	2378	795	1307	2055
Copenhagen, Denmark	803	3157	3017	1047	1778	461	2711	968	1383	2134	1681	904	1058	984	1009	1748	0	3071	1147	2539	1648	2219	1766	1279	809	2019	2160	624	953	1118	1239
Lisbon, Portugal	2267	4813	4673	2713	3434	2892	2799	2088	3241	4147	2791	139	3877	4055	2080	4227	3071	0	2257	644	852	4748	3665	3791	1792	2816	2730	3816	3695	3482	2976	2228
London, UK	344	3539	3399	1445	2160	996	2512	374	1771	2657	534	537	2259	2131	961	2381	1147	2257	0	1725	1232	2852	1741	2436	465	1285	1902	2542	1771	1586	1506	1033
Madrid, Spain	1735	4169	4029	2200	2790	2360	2210	1556	2709	3503	2259	1454	3377	3523	1507	3745	2539	644	1725	0	1281	4216	3133	3263	1260	2284	2086	3172	3163	2950	2444	1696
Monaco, Monaco	1194	2600	1843	1101	1312	1288	1201	1031	1195	1782	1651	837	2510	2397	546	2250	1648	852	1232	1281	0	2934	2142	2076	823	1104	533	1135	2251	1688	1071	501
Moscow, Russia	2541	3086	3217	1810	2147	1856	3420	2645	1953	1918	3386	2454	1002	1141	2395	816	2219	4748	2852	4216	2934	0	2055	1197	2956	1897	3187	2345	1501	1266	2100	2564
Oslo, Norway	1397	3736	3596	1573	2357	1040	3201	1562	1962	2734	2275	1338	876	914	1594	2348	594	3665	1741	3133	2142	2055	0	2208	1873	1388	2613	2739	554	1553	1697	1833
Chisinau, Moldova	1991	941	1326	997	897	1438	2246	2101	909	425	2922	2000	1647	1733	1651	463	1766	3791	2436	3263	2076	1197	2208	0	2299	1319	1832	780	1794	932	1141	1923
Paris, France	475	3198	3058	1221	1819	1100	2131	296	1561	2447	999	320	2241	2263	643	2485	1279	1792	465	1260	823	2956	1873	2299	0	1075	1437	2201	1903	1690	1296	581
Prahue, Czech Republic	956	2348	2208	326	969	348	2052	911	574	1460	1819	670	1428	1739	556	1401	809	2816	1285	2284	1104	1897	1388	1319	1075	0	1290	1351	1418	631	309	6671
Rome, Italy	1766	2416	1417	972	1321	1529	912	1545	1149	2034	2436	1221	2540	3003	854	2486	2019	2730	1902	2086	533	3187	2613	1832	1437	1290	0	1703	2643	1921	1251	888
Sofia, Bulgaria	2213	997	872	919	382	1699	1265	2168	777	427	3076	1833	2216	2559	1522	1529	2160	3816	2542	3172	1135	2345	2739	780	2201	1351	1703	0	2769	1451	1042	1672
Stockholm, Sweden	1427	3766	3626	1413	2387	1070	3102	1592	1992	2764	2305	1526	430	360	1704	2378	624	3695	1771	3163	2251	1501	554	1794	1903	1418	2643	2769	0	1583	1727	1863
Warsaw, Poland	1275	2448	2308	606	1069	590	2368	1379	674	1181	2120	1209	934	1108	1135	795	953	3482	1586	2950	1688	1266	1553	932	1690	631	1921	1451	1583	0	679	1298
Vienna, Austia	1177	2039	1899	59	660	657	1598	1132	265	1151	2040	874	1533	1787	611	1307	1118	2976	1506	2444	1071	2100	1697	1141	1296	309	1251	1042	1727	679	0	748
Berne, Switzerland	880	2669	2529	845	1290	845	1735	659	1013	2003	1567	377	2088	2223	192	2055	1239	2228	33	1696	501	2564	1833	1923	581	6671	888	1672	1863	1298	748	0

ДОДАТОК Б. Координати точок на мапі Києва

(340, 1190) (522, 124) (254, 388) (311, 717) (630, 690) (500, 500)
(980, 630) (1146, 484) (111, 211) (524, 130) (737, 580) (686, 286)
(644, 1094) (480, 945) (881, 1031) (773, 800) (862, 702) (719, 820)
(673, 518) (470, 884) (550, 40) (321, 587) (635, 771) (975, 930)
(555, 1000) (346, 345) (786, 735) (209, 320) (197, 649) (70, 793)
(827, 780) (722, 968) (655, 655) (321, 1231) (746, 886) (537, 820)
(1200, 633) (240, 835) (313, 1000) (491, 518) (349, 77) (119, 873)
(481, 814) (313, 1003) (1100, 771) (633, 920) (512, 966) (880, 180)
(434, 170) (800, 800) (355, 11) (356, 789) (1088, 60) (148, 814)
(807, 1065) (611, 77) (783, 214) (1158, 50) (809, 657) (502, 804)
(981, 1056) (762, 805) (633, 489) (555, 660) (533, 487) (362, 773)
(119, 979) (509, 375) (261, 515) (970, 1120) (656, 685) (556, 911)
(613, 804) (417, 612) (356, 846) (666, 666) (265, 315) (104, 1110)
(390, 445) (419, 240) (511, 400) (910, 661) (1212, 1087) (1100, 800)
(818, 718) (644, 1238) (630, 337) (321, 329) (584, 120) (273, 990)
(360, 780) (864, 1161) (791, 834) (339, 705) (217, 1046) (943, 1212)
(320, 819) (408, 1017) (388, 129) (1275, 685)