МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ

ЦЕНТР ЗАОЧНОЇ, ДИСТАНЦІЙНОЇ ТА ВЕЧІРНЬОЇ ФОРМ НАВЧАННЯ

КАФЕДРА КОМП'ЮТЕРНИХ НАУК

СЕКЦІЯ ІКТ

ВИПУСКНА РОБОТА

на тему:

«Рішення задачі комівояжера алгоритмом мурашиної колонії та порівняння з методом простого перебору»

Завідувач

випускаючої кафедри **Довбиш А. С.**

Керівник роботи **Шаповалов С. П.**

Студента групи ІНдн – 72С **Коротенко В. В.**

СУМИ 2021

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ

Центр заочної, дистанційної і вечірньої форм навчання
Кафедра комп'ютерних наук

Затверджую_____

Зав. кафедрою Довбиш А.С.

"_____" _____2021 р.

ЗАВДАННЯ

до випускної роботи

Студента п'ятого курсу, групи ІНз-71С спеціальності "Комп'ютерні науки" заочної форми навчання Коротенко Вячеслава Володимировича

Тема:«Рішення задачі комівояжера алгоритмом мурашиної колонії та порівняння з методом простого перебору»

Затверджена наказом по СумДУ

№_____ от _____2021 р.

Зміст пояснювальної записки: 1) аналітичний огляд застосувань структур даних ; 2) постановка завдання й формування завдань дослідження; 3) опис основних положень, математичних моделей і алгоритмів, що використовуються для рішення поставленого завдання; 5) розробка інформаційного й програмного забезпечення; 6) аналіз результатів моделювання.

Дата видачі завдання "_____"_____2021 р.

Керівник випускної роботи _____ Шаповалов С. П.

Завдання прийняв до виконання _____ Коротенко В. В.

РЕФЕРАТ

Записка: 41 стор., 7 рис., 2 табл., 1 додаток, 7 джерел інформації.

Об’єкт дослідження — задача комівояжера.

Мета роботи — комп’ютерний порівняльний аналіз алгоритму мурашиної колонії та метода простого перебору для рішення задачі комівояжера.

Методи дослідження — математичне моделювання, комп’ютерна реалізація алгоритмів на ЕОМ.

Результати — розроблено інформаційне та програмне забезпечення комп’ютерного порівняльного аналізу рішення задачі комівояжера алгоритмом мурашиної колонії та методом простого перебору. Проведено огляд алгоритмів, виконано комп’ютерну реалізацію за допомогою алгоритмічної мови програмування С++.

ЗАДАЧА КОМІВОЯЖЕРА, АЛГОРИТМ МУРАШИНОЇ КОЛОНІЇ, МЕТОД ПРОСТОГО ПЕРЕБОРУ, ПОРІВНЯЛЬНИЙ КОМП’ЮТЕРНИЙ АНАЛІЗ,МОВА ПРОГРАМУВАННЯ С++

ЗМІСТ

ВСТУП..... 5

1 АНАЛІЗ ПРОБЛЕМИ. ПОСТАНОВКА ЗАДАЧІ ДОСЛІДЖЕННЯ..... 6

1.1Аналітичній огляд основних алгоритмів..... 6

1.2 Постановка задачі..... 9

2 МАТЕМАТИЧНА ПОСТАНОВКА ЗАВДАННЯ ТА ВИБІР МЕТОДУ ЇЇ РІШЕННЯ. 10

2.1 Алгоритм мурашиної колонії (ACO) та його застосовність..... 11

2.2 Алгоритм простого перебору..... 15

3 РОЗРОБКА ІНФОРМАЦІЙНОГО ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА ТЕСТОВІ РОЗРАХУНКИ……17

3.1 Формування вхідних даних 17

3.2 Комп’ютерна реалізація алгоритмів та основні її складові..... 17

3.3 Порівняльний аналіз алгоритмів на тестових прикладах 19

ВИСНОВКИ..... 25

СПИСОК ЛІТЕРАТУРИ..... 26

ДОДАТОК..... 27

ВСТУП

Задача комівояжера (в англомовних дослідженнях, TSP – travelling salesman problem) представляє собою одну з добре відомих й важливих в інформаційно-комунікаційних технологіях проблем [1-7].

Типове її представлення: маємо заданий список міст(пунктів) для відвідування з заданими відстанями між ними. Комівояжеру потрібно знайти найкоротший маршрут, при якому він відвідає кожне місто й повернеться до початкового. В такій постановці задачу можна віднести до класу комбінаторно оптимізаційних проблем. Незважаючи на простоту постановки задачі, доведено, що TSP належить до класу NP-повних задач й тому її рішення при достатньо великих вхідних даних потребує багато зусиль.

Важливість рішення TSP крім теоретичної привабливості додається різноманітністю його застосувань. Типові програми для досліджень операцій включають маршрутизацію транспортних засобів, проводку комп’ютера, вирізання шпалер та послідовність робіт. Основним застосуванням у статистиці є комбінаторний аналіз даних, наприклад, упорядкування рядків і стовпців матриць даних або визначення кластерів.

TSP має кілька застосувань навіть у найчистішому формулюванні, таких як планування, логістика та виготовлення мікрочіпів. Трохи модифікований, він з’являється як підпроблема у багатьох сферах, таких як секвенування ДНК. У цих додатках місто-концепція представляє, наприклад, клієнтів, місця пайки або фрагменти ДНК, а відстань концепції - час подорожі або вартість, або міру схожості між фрагментами ДНК. TSP також з'являється в астрономії, оскільки астрономи, спостерігаючи за багатьма джерелами, хочуть мінімізувати час, витрачений на переміщення телескопа між джерелами; у таких проблемах

Еквівалентним формулюванням даної проблеми є з точки зору теорії графів: Дано повний зважений графік (де вершини представляли б міста, ребра представляли би дороги, а ваги - вартість або відстань цієї дороги), знайти гамільтоновий цикл з найменшою вагою.

1 АНАЛІЗ ПРОБЛЕМИ. ПОСТАНОВКА ЗАДАЧІ ДОСЛІДЖЕННЯ

1.1Аналітичній огляд основних алгоритмів

Завдання комівояжера (або TSP від англ. Travelling salesman problem) - одна з найвідоміших задач комбінаторної оптимізації, що полягає в пошуку самого вигідного маршруту, що проходить через зазначені міста хоча б по одному разу з наступним поверненням в початковий місто.

Вперше проблема TSP була сформульована в 1930 році і собою в сьогоденні в інформаційно-комуніувційних технологіях однією з найбільш вивчених проблем оптимізації [1-7]. Незважаючи на те, що проблема обчислювально складна (відноситься до класу NP-складних), відомо багато евристичних і точних алгоритмів, за допомогою яких розв’язувались практичні завдання.

Дослідження по розв’язанню задачі комівояжера можна розділити умовно на два напрями [1-7]. 1. Розробка точних алгоритмів, які працюють досить швидко лише для невеликих розмірів задач; 2. Розробка "неоптимальних" або евристичних алгоритмів, які забезпечують апроксимовані рішення за розумний час.

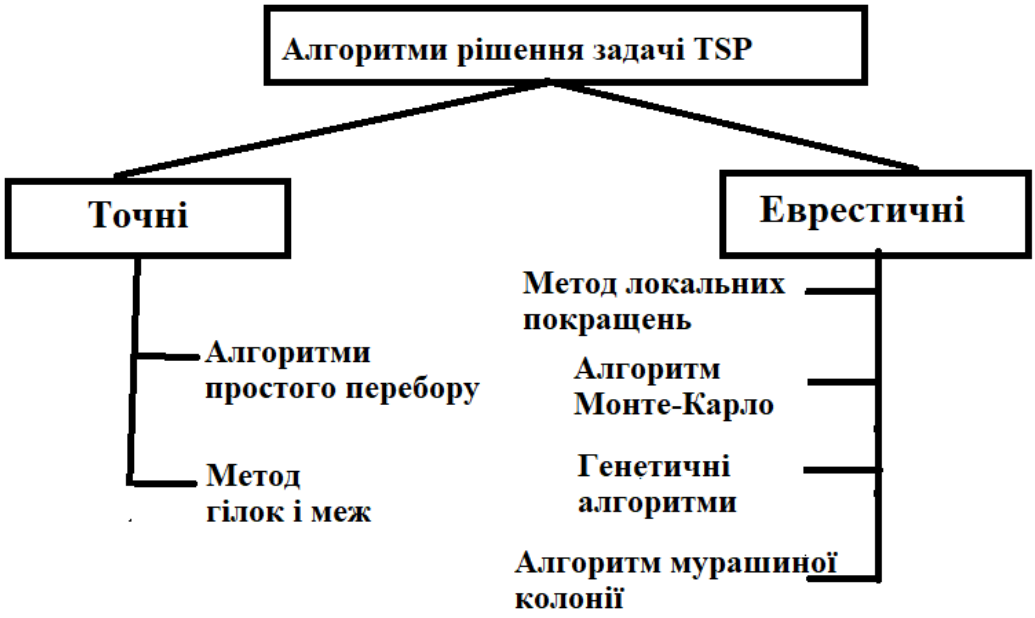


Рисунок 1.1 – Алгоритми рішення задачі TSP

Можна виділити ще пошук особливих випадків для проблеми TSP ("підзадачі"), для яких можливі або краща, або точна евристика.

Важливо також розрізняти задачі TSP симетричні та асиметричні. Для симетричного випадку (зазвичай називається просто TSP), для всіх відстаней у D

рівність $d_{ij} = d_{ji}$ виконується, тобто не має значення, рухаємось від і до j або навпаки,відстань однакова. У асиметричному випадку (званому ATSP) відстані не рівні для всіх пар міст. Проблеми такого роду виникають, коли ми не маємо справи з просторовими відстанями між містами, але, наприклад, з вартістю або необхідним часом, пов'язаним з поїздкою між місця, де ціна на квиток на літак між двома містами може бути різною. В даному разі розглянемо симетричний варіант задачі TSP.

Цілком очевидно, що завдання може бути вирішена перебором всіх варіантів пересувань комівояжера і вибором оптимального. Вся річ в тому, що кількість можливих маршрутів N дуже швидко зростає з ростом кількості для відвідування міст n

$$N = n! \qquad (1.1)$$

Наприклад, для n = 100 кількість варіантів буде представлятися 158-значним числом. Сучасна ЕОМ, здатна перебирати мільйон операцій в секунду, буде битися з завданням протягом приблизно 144 років, що нам «смертним» непідвласне дочекатись рішень.

Вважається доведеним, що не існує точного алгоритму рішення TSP, що має поліноміальну складність (тобто має асимптотичну оцінку часу виконання алгоритму $T(n) = O(n^{\alpha})$) трудомісткості виконання. Отже, задача TSP для скільки-небудь великих n стає майже нерозв'язною для точних алгоритмів.

TSP - проблема, яка важка для NP, і її так легко описати, а так складно вирішити. В такому випадку слід відмовитися від спроб відшукати точне рішення задачі комівояжера і зосередитися на пошуку наближеного - нехай не оптимальні, але хоча б близького до нього. З причини великої практичної важливості завдання корисними будуть і наближені рішення [1-7].

Еволюційний алгоритм (EA) - це алгоритм, вперше поданий Чарльзом Дарвіном у 1859 році. Він забезпечує рішення для різних проблеми оптимізації. Він копіює процес еволюції, що відбувається в природі, тобто мутація, рекомбінація та Природний відбір. GA (Генетичний алгоритм) - це тип еволюційного алгоритму. GA забезпечує вирішення проблеми в форму рядків чисел і застосовує такі

оператори, як мутація та рекомбінація. Це починається з початкового та за допомогою цих операторів знаходить найбільш підходяще покоління населення.

Алгоритм ACO (Оптимізація колонії мурашок) - це також евристичний алгоритм, вперше знайдений Марко Доріго, який забезпечує оптимальне рішення, імітуючи спосіб, яким мурахи знаходять їжу. ACO - це тип CI (інтелект роїв) техніка.

Алгоритм PSO (Рой частинок), вперше знайдений Кеннеді та Еберхартом, також забезпечує оптимальне рішення для проблеми і надихається зграєю птахів, риб та стадами тварин. Це імітує те, як вони знаходять собі їжу середовища і дотримується підходу до обміну інформацією.

Окрім представлення TSP, як задачі комбінаторної оптимізації, TSP також може бути сформульована як теоретична задача теорії графів.

Представимо TSP в формулюванні задачі теорії графів. Теорія графів визначає проблему як знаходження гамільтонового циклу з найменшою вагою для заданого повного зваженого графа. Такого роду постановка задачі широко поширена в інженерних програмах та деяких промислових проблемах, таких як машинне планування, стільникове виробництво та проблеми призначення частоти можуть бути сформульовані як TSP.

Нехай задано зважений граф G = (V, E), в якому міста відповідають множині вершмн V = {1, 2,. . . , n} і кожне ребро $e_i \in E$ має відповідну вагу w_i, що представляє відстань між містами, які вона з'єднує. Якщо граф не є повним, відсутні ребра можна замінити ребрами з дуже великими відстанями.

Метою рішення задачі TSP є пошук гамільтонового циклу, тобто циклу, який відвідує кожен вузол на графі рівно один раз, з найменшою можливою вагою для заданого графа. Це формулювання природно призводить до процедур, що передбачають пошук мінімальних каркасів дерев для заданого графа.

TSP також можуть бути представлені як цілочисельні та лінійні програми програмування. Формулювання цілочисельного програмування (IP) базується на задачі присвоєння з додаткове обмеження відсутності під-турів.

1.2 Постановка задачі

Для досягнення мети дослідження виконаємо наступне завдання.

Провести порівняльний комп'ютерний аналіз алгоритмів рішення задачі комівояжера. В якості алгоритмів порівняння обрати – алгоритм мурашиної колонії та алгоритм простого перебору.

В рамках поставленого завдання потрібно знайти рішення підзадач:

1. Створити комп'ютерну реалізацію алгоритмів.
2. Провести порівняльний аналіз цих алгоритмів на тестових завданнях.
3. На основі проведених досліджень зробити висновки застосовності алгоритмів.

2 МАТЕМАТИЧНА ПОСТАНОВКА ЗАВДАННЯ ТА ВИБІР МЕТОДУ ЇЇ РІШЕННЯ

Перефразуємо задачу комівояжера, використовуючи модель графа. Вершини графа в такій постановці будуть відповідати заданим містам, які потрібно відвідати комівояжеру, а ребра (дуги), що зв'язують вершини між собою будуть відповідати дорогам між містами, по яким може пересуватися комівояжер. Вартості дуг будуть задавати відстані між містами. Тоді рішенням задачі є пошук у зваженому графі такого замкнутого маршруту (потрібно пройти граф по всім вершинам та повернутися в початкову), щоб сума вартостей всіх дуг по яким пересувався комівояжер була мінімально можливої для даного графа.

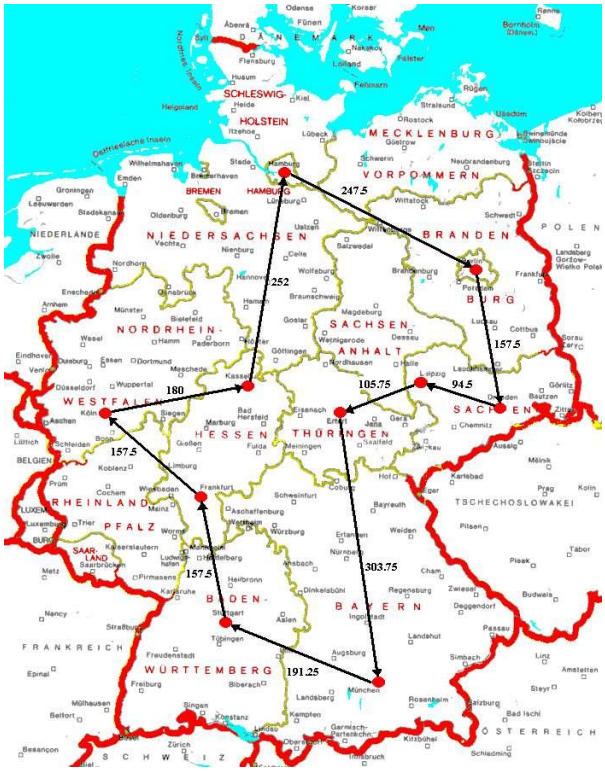


Рисунок 2.1 – Ілюстрація рішення задачі комівояжера

Перейдем до математичної постановки задачі для обраній математичній моделі графа.

Нехай задано граф G (V, E), де V – множина вершин, E – множина ребер. Задамо на ребрах вагову функцію C:E×E →R, яка відображує відстані між містами, що сполучені конкретним ребром. Тобто c_{ij} відповідає ребру $e_{ij} = \{V_i, V_j\}$ і вказує відстань від міста V_i та V_j відповідно.

Розв’язання задачі комівояжера вимагає пошуку такого замкненого маршруту в даному графі, що проходить по всім вершинам графа (по кожній тільки одноразово) с закінченням в тій вершині з якої цей маршрут розпочався.

Введемо нову змінну x_{ij} , яка буде індикатором чи входить ребро графа в шуканий маршрут комівояжера, вона дорівнює 1 якщо ребро належить маршруту та 0 в протилежному випадку.

Тоді, математична постановка має рівняння

$$min\left\{\sum_{i\in V}\sum_{j\in\frac{V}{[i]}} c_{ij}\, x_{ij} \,\middle|\, x\ valid\ (1)(2), xij\ \in\{0,1\}\right\} \qquad (2.\ 1)$$

2.1 Алгоритм мурашиної колонії (ACO) та його застосовність

ACO винайшов та описав у 1992 р. дослідник штучного інтелекту Марко Доріго [2-4]. Пізніше, в багатьох дослідженнях почали вбачати аналогію між системами колоній мурашок та нейронними мережами з урахуванням поведінкової ефективності навчання [2] й почали використовувати ACO для рішення інших задач. Наприклад, відзначимо проведення інтерполяції зображень на основі алгоритму колонії мурашок [4]. На рис. 2.2 надані деякі застосування ACO.

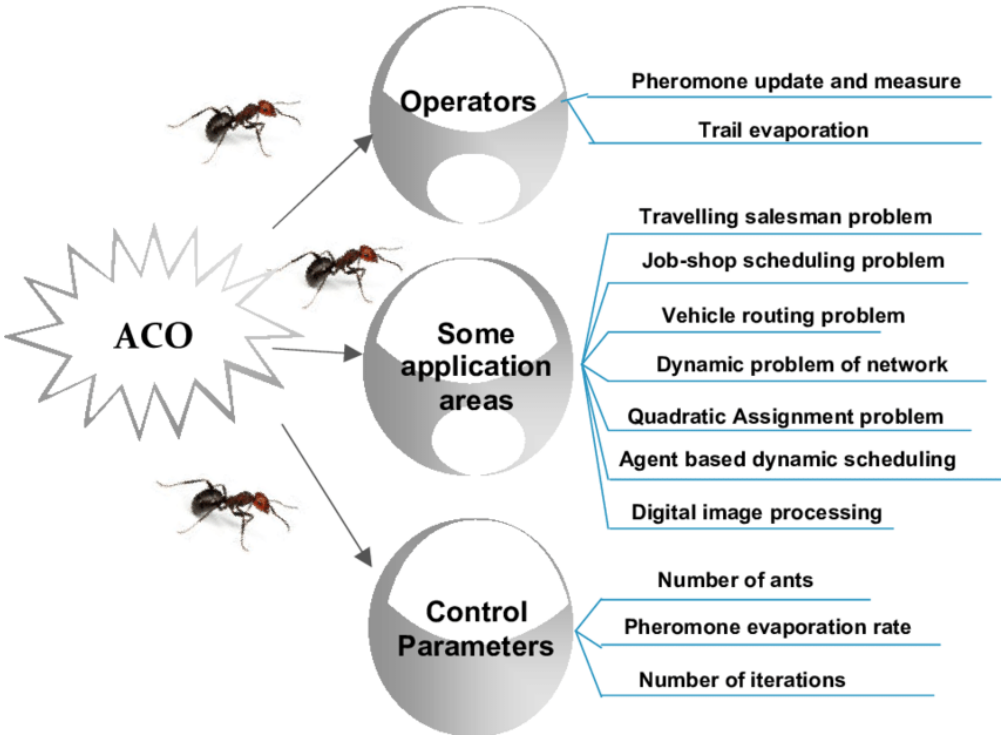


Рисунок 2.2 – Застосовність алгоритму ACO

Сам алгоритм ACO винайдений завдяки натхненної поведінки мурах у реальному світі [3]. Мурахи дотримується принципу виживання колоній, а не виживання особини, оскільки вони завжди залишаються у своїх колоніях.

Мурахи відомі тим, що виконують дуже складні завдання дуже просто. Вони шукають найкоротший шлях між джерелом їжі та їхнім гніздом. Спочатку вони хаотично блукають у своєму оточенні. Досліджуючи шлях, вони відкладають хімічну речовину, тобто феромон, на своєму шляху, який вони відвідують. Якщо мураха знаходить їжу, вона знову повертається до свого гнізда шляхом повторного осадження цього феромонового сліду на землю. Інші мурахи відчувають запах цих феромонових стежок. Шанси мурахи вибір шляху залежить від кількості феромону, що осідає на цьому шляху. Вища концентрація феромону на шлях, ймовірність обрати цей шлях іншим мурахом також зростає. Отже, ймовірність вибору шляху та щільність феромони прямо пропорційні. Цей факт був математично описаний співвідношенням

$$\begin{cases} P_{ij,k}(t) = \dfrac{\left[\tau_{ij}(t)\right]^{\alpha}\left[\mu_{ij}(t)\right]^{\beta}}{\sum_{i\in J_{i,k}}\left[\tau_{ij}(t)\right]^{\alpha}\left[\mu_{ij}(t)\right]^{\beta}} < Rand(1), if\ j \in J_{i,k}\ , \\ P_{ij,k}(t) = 0, if\ j \notin Ji,k \end{cases} \qquad (2.\ 2)$$

По цій формулі визначається $P_{ij,k}(t)$ – ймовірність пересування мурашки з номером k від міста з індексом і до міста з індексом j. На її визначення впливають:

$\tau_{ij}(t)$ – кількість феромону, що накопичилась на ребрі $\{V_i, V_j\}$, що з'єднує між собою міста V_i та V_j :

$\mu_{ij}(t)$ – параметр, що враховує відстань між вершинами V_i та V_j . На самому разі він приймається $\mu_{ij}(t) = 1/dij$, де d_{ij} - відстань між вершинами V_i та V_j;

α – параметр для контролю впливу на вибір феромонів, що задає ваги слідів феромону на ребрах. При $\alpha = 0$ мураха обирає найближчу вершину до тої, в якій він знаходиться, тобто даний алгоритм переходить до жадібної стратегії вибору;

β – параметр для контролю впливу $\mu_{ij}(t)$, що враховує ваги самих ребер (відстаней між містами). Якщо β = 0, тоді спрацьовує лише вплив феромону (сліпі мурахи), що приведе до швидкого виродження маршрутів до одного субоптимального рішення;

$Rand(1)$ – генерація випадкового числа з інтервала (0; 1).

Деякі зауваження щодо використання формули (2. 2).

1. Сама формула визначає тільки ймовірності пересування мурах між містами, а потім в яке місто мураха відправиться вирішується за правилом рулетки. Всі ці ймовірності утворюють сектори на рулетці, а потім запускається «шарик» і на якому секторі він зупиниться – те й визначить шлях комахи.
2. Ймовірності $P_{i,j,k}(t)$ в одній і тій же вершині для різних комах можуть різнитися, бо $P_{i,j,k}(t)$ – функція від $J_{i,k}$ - списку ще не відвіданих вершин мурахою k.
3. Опісля проходження маршруту кожної мурахи підраховується кількість феромону, що змінює існуючий розподіл на графі.

   Якщо, $T_k(t)$ – маршрут, пройдений мурахою k на ітерації t; $L_k(t)$ — довжина цього маршруту, то кількість феромону яка додається на кожному пройденому ребру, що належить маршруту, обчислюється формулою

$$\Delta\tau_{ij,k}\ \ (t) = \frac{Q}{L_k(t)} = \begin{cases} \frac{Q}{L_k(t)}, \text{якщо } (ij) \in T_k(t) \\ 0, \text{якщо } (ij) \notin T_k(t) \end{cases}, \qquad (2.3)$$

де Q - параметр, значення якого обирають одного порядку з довжиною оптимального маршруту.

За формулою (2.4) ми зиінюємо кількості феромону на ребрах

$$\tau_{ij,k}(t+1) = \tau_{ij}(t) + \mathrm{c} \cdot \sum_{k=1}^{N_{ij}} \Delta\tau_{ij}(t), \qquad (2.4)$$

й знову запускаємо колоніє мурах.

В формулі (2 .4) 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 { V_i, V_j }.

Отже, в застосуваннях ACO при рішенні задачі TSP штучний мураха - це простий обчислювальний агент, який шукає хороші рішення даної задачі оптимізації. Щоб застосувати алгоритм колонії мурашок, задачу оптимізації потрібно перетворити на задачу пошуку найкоротшого шляху на зваженому графі. На першому кроці кожної ітерації кожен мураха стохастично будує рішення, тобто порядок, у якому слід дотримуватися переміщення по вершинам та ребрам графа. На другому кроці порівнюються шляхи, знайдені різними мурахами. Останній крок складається з оновлення рівня феромонів для кожного ребра графа й переходу для наступного кроку ітерацій.

В термінах псевдокоду запишемо

```
procedure ACO_MetaHeuristic is
        while not terminated do
generateSolutions()
daemonActions()
pheromoneUpdate()
        repeat
                end procedure
```

Взагалі кажучи, ACO розглядаються як метаевристика, причому кожне рішення представлене мурашкою, що рухається в просторі пошуку. Мурахи можуть розглядатися як імовірнісні багатоагентні алгоритми, що використовують розподіл ймовірностей для здійснення переходу між кожною ітерацією.

Широке розмаїття алгоритмів (для оптимізації чи ні), що прагнуть самоорганізації в біологічних системах, призвело до концепції "ройового інтелекту" [3], що є дуже загальною структурою, в яку вкладаються алгоритми колонії мурашок.

2.2 Алгоритм простого перебору

В computer science алгоритм простого перебору відносять до інструментарію грубого пошуку або вичерпного пошуку, який можна подати як генерування та перевірка. Вочевидь, це загальна алгоритмічна парадигма (загальна техніка вирішення будь-яких проблем), яка полягає у систематичному перерахуванні всіх можливих кандидатів на рішення та перевірці, чи задовольняє кожен кандидат твердження проблеми. В англомовних дослідженнях ця парадигма має назву - Brute Force Algorithms.

Такого роду алгоритми є прямолінійними методами вирішення проблеми, які покладаються на чисту обчислювальну потужність та випробування будь-якої можливості, а не вдосконалених методів підвищення ефективності. Як правило їх не хвилює вдосконалення алгоритмічної асимптотичної оцінки трудомісткості, головна мета – перебрати всі можливі варіанти розв’язку та обрати той, що задовольняє постановці завдання. Brute Force Algorithms перевіряє всі можливі сценарії, а не використовує якийсь прийом, щоб пришвидшити процес. Складність повного перебору залежить від кількості всіх можливих рішень задачі (див. рис. 2.3, який показує дію алгоритму). Якщо простір рішень дуже великий, то повний перебір може не дати результатів протягом декількох років або навіть століть.

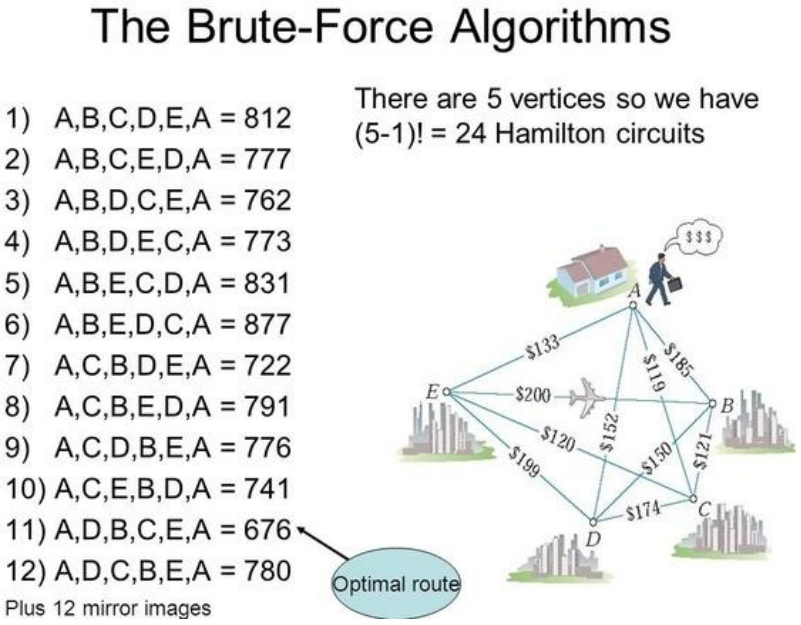


Рисунок 2.3 – Дія алгоритму простого перебору для рішення задачі

Повернемось до постановки завдання – рішення задачі комівояжера в представленні її моделлю графа.

Метод простого перебору в застосуванні для її рішення буде полягати у переборі всіх індексів вершин окрім початкової і обрахування довжини знайдених маршрутів. Вибір серед цих маршрутів того, що має мінімальну довжину і буде шуканим рішенням нашої задачі.

Доведено, що будь-яке завдання з класу NP може бути вирішена повним перебором. При цьому, навіть якщо обчислення цільової функції від кожного конкретного можливого рішення задачі може бути здійснено за поліноміальний час, в залежності від кількості всіх можливих рішень повний перебір може зажадати експоненціального часу роботи.

Підсумуємо сказане, та відмітимо переваги та недоліки представленого алгоритму.

Переваги:

- доступність та зрозумілість в виконанні;
- досить не складна комп’ютерна реалізація;
- не потрібно доводити його застосовність та підходить до виконання будь-яких завдань.

Недоліки:

- неоптимізований алгоритм, просто перебирає сценарії рішення;
- має погану асимптотичну оцінку трудомісткості;
- час виконання алгоритму T(n) конкретно для задачі комівояжера $T(n) = O(n!)$, що встановлює оцінку зверху для кількості вершин в графі (кількості міст n). Наприклад, для $n>20$ вважається, що цей алгоритм не ефективний (читай, не застосовний).

3 РОЗРОБКА ІНФОРМАЦІЙНОГО ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА ТЕСТОВІ РОЗРАХУНКИ

3.1 Формування вхідних даних

За математичною постановкою задачі (див. розділ 2) складено комп'ютерну реалізацію. Вхідними даними, що описують задачу TSP послуговують матриця суміжності зваженого графа та початкові дані колонії мурах (див. рішення тестових прикладів).

3.2 Комп'ютерна реалізація алгоритмів та основні її складові

В якості мови програмування оберемо C++., яка дозволяє писати як в процедурному стилі, так і в об'єктно-орієнтованому і функціональному. Компілятори C ++ є на кожній операційній системі, більшість програм легко переноситься з платформи на платформу, з середовищем розробки і бібліотеками у нас точно не виникне проблем.

Основні причини для вибору мови програмування C++: 1. Реалізація в мові ОПП; 2. STL; 3. Поєднання високорівневих і низькорівневих засобів програмування. Ця мова є однією з затребуваних й в неї не тільки постійно оновлюються стандарти, а й постійно впроваджуються поліпшення.

C ++ стандартизовано Міжнародною організацією зі стандартизації (ISO), остання стандартна версія ратифікована та опублікована ISO у грудні 2020 року як ISO / IEC 14882: 2020 (неофіційно відома як C ++ 20).

Комп'ютерна реалізація розпочинається з підключення стандартних бібліотек та простору імен

```
/* Підключення бібліотек */
 #include <locale>
#include <stdlib.h>
#include <iostream>
#include <malloc.h>
#include <conio.h>
#include <time.h>
#include <fstream>
#include <string.h>
```

```
#include <Windows.h>
/* Підключення простору імен */
using namespace std;
#define ALPHA    1                    // вага ферменту
#define BETA     3                    // коефіцієнт евристики
#define T_MAX    200                  // час життя колонії
#define M    20                  // кількість мурах в колонії
#define p    0.5                 // коефіцієнт випаровування феромону
#define SIZE 1000000                 // максимальна довжина назви файлу
```

Далі описуються функції та процедури, що використовуються при рішенні проблеми. Наведемо основні функції додатку (див. додаток):

Функція ReadSize – читання вхідних даних з файлу й запис його розміру.

ReadMatrix – читаннння матриці суміжності графа з файлу й запис її в масив.

Probability – підраховує ймовірності переходу мурахи до конеретного міста.

InitializationLength - початкова ініціалізація довжини маршруту.

GenerationPheromon – генерує значення феромону.

InitializationAnts -початкова ініціалізація розміщення мурах.

AntColony – знаходження рішення проблеми алгоритмом мурашиної колонії.

PrintResult – вивід результатів на екран.

Хід виконання алгоритму розв’язання задачі на ЕОМ.

Програма при вході запрошує ім’я файлу, де зберігається матриця суміжності графу. Дані записуються в двовимірний масив.

Далі користувачеві пропонується ввести початкову вершину. Після цього розпочинає пошук мінімального маршруту за мурашиним алгоритмом і виводить на екран результати.

Далі програма розраховує мінімальний маршрут з такими ж параметрами тільки методом простого перебору і також виводить результати обрахунків на екран.

Впевнившись в правильності роботи програми, спробуємо порівняти алгоритми з різними налаштуваннями на графах різної розмірності.

Представимо алгоритм АСО, реалізований на ЕОМ псевдокодом

Step 1: [Initialization]

$t := 0$; NC := 0;

For each edge (i,j), initialize trail intensity to $\tau_{ij}(0) := \tau_0$

Step 2: [Starting node]

For each ant k:

Place ant k on a randomly chosen city and store this information in $Tabu_k$

Step 3: [Build a tour for each ant]

For i from 1 to n:

For k from 1 to m:

Choose the next city j, $j \notin Tabu_k$, among the cl candidate cities according to:

$$j = \begin{cases} \arg\max_{\ell \notin Tabu_k} \left\{ [\tau_{i\ell}(t)]^{\alpha} \cdot [\eta_{i\ell}]^{\beta} \right\} & \text{if } q \le q_0 \\ J & \text{if } q > q_0 \end{cases} \quad (3)$$

where J is chosen according to the probability:

$$p_{ij}^{k}(t) = \frac{[\tau_{ij}(t)]^{\alpha} \cdot [\eta_{ij}]^{\beta}}{\sum_{\ell \notin Tabu_k} [\tau_{i\ell}(t)]^{\alpha} \cdot [\eta_{i\ell}]^{\beta}} \quad (1)$$

Store the chosen city in $Tabu_k$

Local update of trail for chosen edge (i,j) :

$$\tau_{ij}(t) := \rho_{\ell} \cdot \tau_{ij}(t) + (1-\rho_{\ell}) \cdot \Delta\tau_{ij} \quad \text{where } \Delta\tau_{ij} = \tau_0 \quad (5)$$

Step 4: [Global update of trail]

Compute length of tour, L_k, for each ant k

Apply local improvement method for the tours of all ants k and recompute L_k

For each edge $(i,j) \in Cycle^*$, update the trail according to:

$$\tau_{ij}(t+1) := \rho_g \cdot \tau_{ij}(t) + (1-\rho_g) \cdot \Delta\tau_{ij} \quad \text{where} \quad \Delta\tau_{ij} = 1/L_{cycle^*} \quad (4)$$

$t := t + 1$; NC := NC + 1

Step 5: [Termination conditions]

Memorize the shortest tour found to this point

IF (NC < NC_{MAX}) and (Not stagnation behavior)

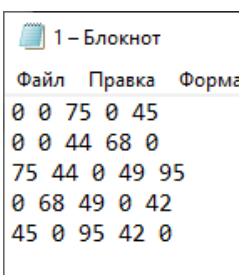
THEN empty all $Tabu_k$ and go to step #2

ELSE Stop

3.3 Порівняльний аналіз алгоритмів на тестових прикладах

Граф з 5 вершин

Вхідний файл – 1.txt.



Параметри алгоритму.

```
ALPHA   1           // вага ферменту
BETA    2           // коефіцієнт евристики
T_MAX   100         // час життя колонії
M       5           // кількість мурах в колонії
p       0.6         // коефіцієнт випаровування феромону
```

Результат роботи програми.

```
Введите путь файла: 1.txt
Введите начальную вершину от 1 до 5: 2
Муравьиный алгоритм
Путь: 2 -> 3 -> 1 -> 5 -> 4 -> 2
Длинна пути: 274
Время работы алгоритма: 3.8892e-06

Простой перебор
Путь: 2 -> 3 -> 1 -> 5 -> 4 -> 2
Длинна пути: 274
Время работы алгоритма: 5.52003e-09
Для продолжения нажмите любую клавишу . . .
```

Рисунок 3.1 – Тестовий приклад на 5 міст

Граф з 9 вершин

Вхідний файл 5.txt.

5 – Блокнот

Файл Правка Формат Вид Справка

```
0 34 69 28 0 0 0 0 0
34 0 7 0 16 0 0 0 0
69 7 0 3 75 15 0 0 0
28 0 3 0 0 0 91 0 0
0 16 75 0 0 0 0 83 39
0 0 15 0 0 0 49 0 54
0 0 0 91 0 49 0 0 54
0 0 0 0 83 0 0 0 6
0 0 0 0 39 54 54 6 0
```

Оберемо початкову вершину – 3.

```
Введите путь файла: 5.txt
Введите начальную вершину от 1 до 9: 3
Муравьиный алгоритм
Путь: 3 -> 4 -> 1 -> 2 -> 5 -> 8 -> 9 -> 7 -> 6 -> 3
Длинна пути: 288
Время работы алгоритма: 0.000179442

Простой перебор
Путь: 3 -> 4 -> 1 -> 2 -> 5 -> 8 -> 9 -> 7 -> 6 -> 3
Длинна пути: 288
Время работы алгоритма: 9.51359e-06
Для продолжения нажмите любую клавишу . . .
```

Рисунок 3.2 – Тестовий приклад на 5 міст

Граф з 13 вершин

Вхідний файл – 4.txt

4 – Блокнот

Файл Правка Формат Вид Справка

```
0 69 71 113 167 107 113 76 149 29 122 122 133 189
69 0 33 102 101 39 72 13 93 68 96 55 73 136
71 33 0 68 100 52 45 45 78 57 64 57 101 118
113 102 68 0 128 111 48 114 89 85 23 103 164 108
167 101 100 128 0 64 81 100 42 156 107 45 92 64
107 39 52 111 64 0 68 35 68 103 98 22 54 110
113 72 45 48 81 68 0 81 45 92 30 56 122 77
76 13 45 114 100 35 81 0 97 78 107 56 61 140
149 93 78 89 42 68 45 97 0 133 66 46 114 43
29 68 57 85 156 103 92 78 133 0 96 113 139 169
122 96 64 23 107 98 30 107 66 96 0 86 152 85
133 73 101 164 92 54 122 61 114 139 152 70 0 152
189 136 118 108 64 110 77 140 43 169 85 88 152 0
```

Параметри роботи алгоритму

```
#define ALPHA   1           // вага ферменту
#define BETA    2           // коефіцієнт евристики
#define T_MAX   800         // максимальний час життя колонії
#define M       13          // кількість мурах в колонії
#define p       0.6         // коефіцієнт випаровування феромону
#define SIZE 1000000        // максимальна довжина назви файлу
```

Початкова вершина – 10

Результат роботи програми.

```
Введите путь файла: 4.txt
Введите начальную вершину от 1 до 13: 10
Муравьиный алгоритм
Путь: 10 -> 2 -> 9 -> 13 -> 4 -> 6 -> 7 -> 5 -> 1 -> 3 -> 12 -> 11 -> 8 -> 10
Длинна пути: 709
Время работы алгоритма: 0.00180399

Простой перебор
Путь: 10 -> 2 -> 7 -> 5 -> 1 -> 8 -> 9 -> 11 -> 13 -> 6 -> 3 -> 12 -> 4 -> 10
Длинна пути: 633
Время работы алгоритма: 0.109485
Для продолжения нажмите любую клавишу . . .
```

Рисунок 3. 3 – Тестовий приклад на 13 міст

З тестових прикладів можна побачити те, що при малій кількості вершин, метод простого перебору має гарний час роботи і при цьому дає завжди точний результат, на відміну від мурашиного алгоритму. Але мурашиний алгоритм має перевагу в тому, що може працювати з графами, кількість вершин в яких заве-
лика Давайте розглянемо залежність результатів мурашиного алгоритму від його параметрів, заодно порівняємо їх з результатами методу простого перебору. За вхідні дані візьмемо граф з 13 вершин та початкову вершину 5. Повторюватимемо дії з кожним набором параметрів по 5 разів.

Вхідний файл 4.txt

4 – Блокнот

Файл Правка Формат Вид Справка

```
0 69 71 113 167 107 113 76 149 29 122 122 133 189
69 0 33 102 101 39 72 13 93 68 96 55 73 136
71 33 0 68 100 52 45 45 78 57 64 57 101 118
113 102 68 0 128 111 48 114 89 85 23 103 164 108
167 101 100 128 0 64 81 100 42 156 107 45 92 64
107 39 52 111 64 0 68 35 68 103 98 22 54 110
113 72 45 48 81 68 0 81 45 92 30 56 122 77
76 13 45 114 100 35 81 0 97 78 107 56 61 140
149 93 78 89 42 68 45 97 0 133 66 46 114 43
29 68 57 85 156 103 92 78 133 0 96 113 139 169
122 96 64 23 107 98 30 107 66 96 0 86 152 85
133 73 101 164 92 54 122 61 114 139 152 70 0 152
189 136 118 108 64 110 77 140 43 169 85 88 152 0
```

Таблиця 3.1 Результати рішення алгоритмом простого перебору

Простий перебір	
Час виконання	Результат
0,103171	633

Таблиця 3.2 Результати роботи алгоритму ACO з часом виконання та різними значеннями α, β.

Параметри	Краще рішення	Гірше рішення	Середній результат	Середній час виконання
α = 1, β = 1				
100 ітерацій	770	839	803	0,00022
1000 ітерацій	724	874	794	0,0024
α = 2, β = 1				
100 ітерацій	838	884	864	0,00016
1000 ітерацій	781	908	857	0,00165
α = 5, β = 1				
100 ітерацій	703	1032	915	0,000215
1000 ітерацій	880	1118	1003	0,00245
α = 0, β = 1				
100 ітерацій	742	742	742	0,00018
1000 ітерацій	742	742	742	0,0019
α = 1, β = 2				

100 ітерацій	760	882	791	0,000152
1000 ітерацій	734	815	776	0,002
$\alpha = 1, \beta = 5$				
100 ітерацій	724	778	755	0,0002
1000 ітерацій	696	744	723	0,0023
$\alpha = 1, \beta = 0$				
100 ітерацій	889	1224	1108	0,0002
1000 ітерацій	1062	1285	1155	0.0022

З результатів досліджень бачимо, що жоден з наборів параметрів мурашиного алгоритму не показав точну відповідь, але найкращим виявився набір $\alpha = 1, \beta = 5$ з середньою довжиною маршруту 723. Час роботи мурашиного алгоритму прямопропорційний часу життя колонії мурах(кількості ітерацій). Так як мурашиний алгоритм зав'язаний на випадкових значення, неможливо визначити потрібну кількість ітерацій. Тому порівняння алгоритмів за часом роботи не має особливого сенсу. Метод перебору має стовідсоткову точність, але в 45 разів повільніший від мурашиного алгоритму (результат 696, 1000 ітерацій). Отже, для вибору потрібного алгоритму треба враховувати розмір графу і точність розрахунків.

Асимптотичні оцінки алгоритмів

Час роботи мурашиного алгоритму є поліноміальним $O(t * m * n^2)$, де t – кількість ітерацій, n – кількість вершин в графі, m – кількість мурах.

Час роботи методу простого перебору O(n!), де n – кількість вершин.

ВИСНОВКИ

В випускній роботі проведені дослідження по застосуванні алгоритму мурашиної колонії (ACO) для рішення задачі комівояжера (TSP).

В якості порівняльного аналізу проведений аналіз одержаних рішень та рішень методом простого перебору.

В результаті проведених досліджень одержані наступні висновки:

1. Алгоритм ACO застосовний для рішення задач TSP. При малих вхідних даних він приводить до точних рішень. При достатньо великих вхідних даних рішення одержуємо з заданою точністю.
2. Алгоритм простого перебору ефективно працює для малої кількості вхідних даних. При кількості міст більш ніж 20 він не є ефективним в застосуванні. Тоді потрібно застосовувати ACO.
3. Комп’ютерне тестування проводилось на різних вхідних даних і має практичне застосування для аналізу алгоритмів.

СПИСОК ЛІТЕРАТУРИ

1. Математичні методи дослідження операцій: підручник/ Є. А. Лавров, Л. П. Перхун, В. В. Шендрик та ін. – Суми: Сумський державний університет, 2017. – 212 с.

2. Hassan M. H. Mustafa, Ayoub Al-Hamadi, Mohamed Abdulrahman, Shahinaz Mahmoud, Mohammed O Sarhan On Comparative Analogy between Ant Colony Systems and Neural Networks Considering Behavioral Learning Performance// Journal of Computer Sciences and Applications. 2015, Vol. 3 No. 3, 79-89.

3. Біологічні основи мурашиних колоній – [Електронний ресурс]. URL: http://posibniki.com.ua/post-prikladni-sistemi-kolektivnogo-intelektu-swarm-intelligence (Дата звернення 15.10.2020).

4. Rukundo, O., Cao, H. Advances on image interpolation based on ant colony algorithm. SpringerPlus 5, 403 (2016) – [Електронний ресурс]. URL: https://springerplus.springeropen.com/articles/10.1186/s40064-016-2040-9

5. Korte B., Vygen J. Combinatorial Optimization: Theory and Algorithms (Algorithms and Combinatorics) 6th ed., New York, 2018,455 p.

6. Divya M. A Comparison of Ant Colony Optimization Algorithms Applied to Distribution Network Reconfiguration// International Journal of Engineering Research & Technology, Volume 3, Issue 01, 2016. – pp. 1-4.

7. Hahsler M., Hornik K. TSP – Infrastructure for the Traveling Salesperson Problem// Journal of Statistical Software, December 2007, Vol. 23, Issue 2, 2007. – pp. 1-21.

ДОДАТОК

```
/* Підключення бібліотек */

#include <locale>
#include <stdlib.h>
#include <iostream>
#include <malloc.h>
#include <conio.h>
#include <time.h>
#include <fstream>
#include <string.h>
#include <Windows.h>

/* Підключення простору імен */
using namespace std;
#define ALPHA    1                  // вага ферменту
#define BETA     3                  // коефіцієнт евристики
#define T_MAX    200                // час життя колонії
#define M    20                  // кількість мурах в колонії
#define p    0.5                 // коефіцієнт випаровування феромону
#define SIZE 1000000                 // максимальна довжина назви файлу

/* Структура МУРАХА */
struct ANT_TYPE {
        int number; // кількість вершин
        double length; // довжина
        int* array; // масив вершин
};

/* Функція ReadSize                                                    */
/* Призначення:                                                        */
/*     Читанння графа з файлу и запис його розміру                     */
/* Вхідні дані:                                                        */
/*     D - посилання на масив, що вміщує довжини всіх ребер в графі    */
/*     name - назва файлу, що вміщує граф                              */
/*     n    - змінна, що вміщує кількість вершин в графі               */
/* Выходные данные:                                                    */
/*     D - посилання на масив, що містить довжину всіх ребер в графі   */
/*     name - назва файлу, що вміщує граф                              */
/*     n - змінна, що вміщує кількість вершин в графі                  */
```

```
/* Значення повернене функцією:                                    */
/*      size - розмір матриці сумісності                          */

int ReadSize(double** D, char name[SIZE], int* n)
{
        FILE* input = NULL;
        int size = 0, j = 0;
        char ch = 0;

        if (fopen_s(&input, name, "rb") == NULL)
        {
                while (!feof(input))
                {
                        ch = fgetc(input);
                        if (ch == '\n') {
                                size++;
                                j = 0;
                        }
                        else if (ch == ' ') {
                                j++;
                        }
                        else if (ch == '-') {
                                printf("Граф має відємні ваги.\nНеможливо застосувани
алгоитм мурашиної колонії.\nДля продовження натисніть будь-яку кла-
вішу...\n");
                                _getch();
                                exit(0);
                        }
                }
                fclose(input);
                *n = size;
        }
        else {
                cout << "Помилка відкриття файлу.\nДля продовження натисніть
будь-яку клавішу...\n";
                _getch();
                exit(0);
        }
        return size;
```

```
}

/* Функція ReadMatrix                                                    */
/* Призначення:                                                          */
/*      Читанння графа з файлу и запис його в масив                      */
/* Вхідні дані:                                                          */
/*      D - посилання на масив, що вміщує довжини всіх ребер в графі     */
/*      name - назва файлу, що вміщує граф                               */
/*      n - змінна, що вміщує кількість вершин в графі                   */
/* Вихідні дані:                                                         */
/*      D - посилання на масив, що містить довжину всіх ребер в графі    */
/*      name - назва файлу, що вміщує граф                               */
/*      n - змінна, що вміщує кількість вершин в графі                   */
/* Значення повернене функцією:                                          */
/*      size - розмір матриці сумісності                                 */

int ReadMatrix(double** D, char name[SIZE], int* n) {

        FILE* input = NULL;
        int size = ReadSize(D, name, n);
        if (fopen_s(&input, name, "rb") == NULL)
        {
                for (int t = 0; t < size; ++t) {
                        for (int k = 0; k < size; ++k) {
                                int num;
                                fscanf_s(input, "%d", &num);
                                D[t][k] = num;
                                /* Обробляємо випадок, якщо не існує марш-
рут з t в k (якщо це не петля) */
                                if (t != k && D[t][k] == 0) D[t][k] = INFINITY; // Te-
пер тут знаходиться нескінченність
                        }
                }
                fclose(input);
        }
        return size;
}
```

```
/* Функція Probability                                                    */
/* Призначення:                                                           */
/*      Підрахунок ймовірності переходу мурахи в місто                    */
/* Вхідні дані:                                                           */
/*      to - місто, куди хоче перейти мураха                              */
/*      ant - мураха                                                      */
/*      tao - посилання на масив з феромоном                              */
/*      epsilon - посилання на масив с видимістю                          */
/*       size - розмір матриці сумісності                                 */
/* Вихідні дані:                                                          */
/*      to - місто, куди хоче перейти мураха                              */
/*      ant - мураха                                                      */
/*      tao - посилання на масив з феромоном                              */
/*      epsilon - посилання на масив с видимістю                          */
/*       size - розмір матриці сумісності                                 */
/* Значення що функція повертає:                                          */
/*      res - значение ймовірності переходу                               */

double Probability(int to, ANT_TYPE ant, double** tao, double** epsilon, int size) {
        double res = 0.0, sum = 0.0;
        if (D[from][to] == INFINITY) return 0.0;
        int from = ant.array[ant.number - 1];
        /* Якщо вершина уже відвідана, повертаємо 0 */
        for (int i = 0; i < ant.number; ++i) {
                if (to == ant.array[i]) {
                        return 0.0;
                }
        }
        /* Підрахунок суми в знаменнику */
        for (int j = 0; j < size; ++j) {
                int flag = 1;
                /* Перевірка: чи відвідувала мураха j вершину? */
                for (int i = 0; i < ant.number; ++i) {
                        if (j == ant.array[i]) {
                                flag = 0;
                        }
                }
                /* Якщо ні, то додаємо до загальної суми */
```

```
            if (flag && from >= 0) {

        sum += pow(tao[from][j], ALPHA) * pow(epsilon[from][j], BETA);
            }
        }
        if (from >= 0) {
            /* Повертаємо значення ймовірності */

        res = pow(tao[from][to], ALPHA) * pow(epsilon[from][to], BETA) / sum;
        }
        return res;
}

/* Функція InitializationLength                                    */
/* Призначення:                                                    */
/*     Початкова ініціалізація довжини маршруту                    */
/* Вхідні дані:                                                    */
/*     epsilon - посилання на масив з видимістю                    */
/*     D - посилання на масив, що містить довжину всіх ребер       */
/*     size - розмір матриці сумісності
                    */
/* Вихідні дані:                                                   */
/*     epsilon - посилання на масив з видимістю                    */
/*     D - посилання на масив, що містить довжину всіх ребер       */
/*     size - розмір матриці сумісності
                                        */
/* Значення що функція повертає:                                   */
/*     відсутнє                                                    */

void InitializationLength(double** epsilon, double** D, int size) {
        for (int i = 0; i < size; ++i) {
            /* Створення двовимірного динамічного масиву */
            epsilon[i] = (double*)malloc(sizeof(double) * size);
            for (int j = 0; j < size; ++j) {
                /* Якщо поточне ребро не є петлею */
                if (i != j) {
                    epsilon[i][j] = 1.0 / D[i][j];
                }
            }
```

```
        }
}

/* Функція GenerationPheromon                                   */
/* Призначення:                                                 */
/*      Початкова генерація феромону                            */
/* Вхідні дані:                                                 */
/*      tao - посилання на масив з початковим феромоном         */
/*      size - розмір матриці сумісності                        */
/* Вихідні дані:                                                */
/*      tao - посилання на масив з початковим феромоном         */
/*      size - розмір матриці сумісності                        */
/* Значення що функція повертає:                                */
/*      відсутнє                                                */
                                                                */
void GenerationPheromon(double** tao, int size) {
        /* Генерація феромону */
        srand((unsigned)time(NULL));
        for (int i = 0; i < size; ++i) {
                /* Створення двовимірного динамічного масиву */
                tao[i] = (double*)malloc(sizeof(double) * size);
                for (int j = 0; j < size; ++j) {
                        /* Випадкове заповнення масиву */
                        tao[i][j] = 1 + (size * rand() / RAND_MAX);
                }
        }
}

/* Функція InitializationAnts(                                  */
/* Призначення:                                                 */
/*      Початкове розміщення мурах                              */
/* Вхідні дані:                                                 */
/*      ants - масив з колонією мурах                           */
/*      size - розмір матриці сумісності                        */
/*      city_1 - початкове місто                                */
/* Вихідні дані:                                                */
/*      ants - масив з колонією мурах                           */
/*      size - розмір матриці сумісності                        */
/*      city_1 - початкове місто                                */
```

```
/* Значення що функція повертає:                                    */
/*      відсутнє
                                                                    */

void InitializationAnts(ANT_TYPE ants[M], int size, int city_1) {
        /* Цикл по всіх мурахах */
        for (int k = 0; k < M; ++k) {
                ants[k].number = 0;
                ants[k].length = 0.0;
                ants[k].array = (int*)malloc(sizeof(int) * (size));
                /* Розміщення колонії в початковій вершині */
                ants[k].array[ants[k].number++] = city_1;
        }
}

/* Функція Pheromon                                                 */
/* Призначення:                                                     */
/*      Оновлення феромону                                          */
/* Вхідні дані:
                                                                    */
/*      ants - масив з колонією мурах                               */
/*      Q - початковий параметр алгоритму                           */
/*       size - розмір матриці сумісності                           */
/*      tao - посилання на масив з феромоном                        */
/*      k - номер мурахи в колонії                                  */
/* Вихідні дані:                                                    */
/*      ants - масив з колонією мурах                               */
/*      Q - початковий параметр алгоритму                           */
/*       size - розмір матриці сумісності                           */
/*      tao - посилання на масив з феромоном                        */
/*      k - номер мурахи в колонії                                  */
/* Значення що функція повертає:                                    */
/*      відсутнє
                                                                    */

void Pheromon(ANT_TYPE ants[M], double Q, double** tao, int k) {
        for (int i = 0; i < ants[k].number - 1; ++i) {
                int from = ants[k].array[i % ants[k].number];
                int to = ants[k].array[(i + 1) % ants[k].number];
```

```
                tao[from][to] += Q / ants[k].length;
                tao[to][from] = tao[from][to];
        }
}

/* Функція AntColony                                                    */
/* Призначення:                                                         */
/*      Знаходження маршруту за допомогою мурашиного алгоритму         */
/* Вхідні дані:                                                         */
/*      D - посилання на масив, що містить довжину всіх ребер в графі   */
/*       size   - розмір матриці сумісності                             */
/*       city_1 - початкове місто                                       */
/*       city_2 - кінцеве місто                                         */
/* Вихідні дані:                                                        */
/*      D - посилання на масив, що містить довжину всіх ребер в графі   */
/*       size   - розмір матриці сумісності                             */
/*       city_1 - початкове місто                                       */
/*       city_2 - кінцеве місто                                         */
/* Значення що повертає функція:                                        */
/*      way - найкоротший маршрут, кількість його вершин та довжина     */

ANT_TYPE AntColony(double** D, int size, int city_1) {
        /* Початкова ініціалізація даних */
        ANT_TYPE way;
        way.number = 0;
        way.length = -1;
        way.array = (int*)malloc(sizeof(int) * (size));
        double Q = 1.0;
        ANT_TYPE ants[M];
        /* Ініціалізація даних про довжину і кількість феромону */
        double** epsilon = NULL, ** tao = NULL;
        /* Створення динамічних масивів */
        epsilon = (double**)malloc(sizeof(double*) * (size));
        tao = (double**)malloc(sizeof(double*) * (size));

        /* Виклик функції для ініціалізації початкового маршруту */
        InitializationLength(epsilon, D, size);
```

```
/* Виклик функції для генерації феромону */
GenerationPheromon(tao, size);

/* Виклик функції для розміщення всіх мурах в початковому місті */
InitializationAnts(ants, size, city_1);

/* Основний цикл мурашиного алгоритму */
for (int t = 0; t < T_MAX; ++t) {
	/* Цикл по мурахам */

	for (int k = 0; k < M; ++k) {
		/* Локальний пошук маршруту для k-ої мурахи */
	   /* Оновлення мурах */
		ants[k].number = 1;
		ants[k].length = 0.0;
		int block;
		do {
			block = 0;
			int J_max = -1;
			double P_max = 0.0;
			for (int j = 0; j < size; ++j) {
				/* Перевірка ймовірності переходу в вершину j */
				if (ants[k].array[ants[k].number - 1] != j) {
					/* Виклик функції для підрахунку ймовірно-
сті */

	double P = Probability(j, ants[k], tao, epsilon, size);
					if (P != 0 && P >= P_max) {
						P_max = P; // запам'ятовуємо бі-
льшу ймовірність
						J_max = j; // запам'ятовуємо номер мі-
ста, з цією ймовірностю
					}
	f(P==0) block++;

				}
			}

			if (J_max >= 0) {
```

```
                                /* Підрахунок пройденої відстані і прокладення маршруту */

        ants[k].length += D[ants[k].array[ants[k].number - 1]][J_max]; // поточній довжині додається значення з D
                                ants[k].array[ants[k].number++] = J_max;
                        }

                if(block==size-1) break; } while (ants[k].number != size); // поки не досягнута кінцева вершина
        if(block == size-1) continue;

        ants[k].length += D[ants[k].array[ants[k].number - 1]][city_1]; // поточній довжині додається значення з D
                        ants[k].array[ants[k].number++] = city_1;
                        /* Залишаємо слід феромону від колонії */
                        Pheromon(ants, Q, tao, k);

                        /* Перевірка, чи знайшли ми краще рішення? */
                        if (ants[k].length < way.length || way.length < 0) { // довжина побудованого маршруту менша тої, що спочатку задана в графі
                                /* Переприсвоєння номерів вершин, довжин і маршруту */
                                way.number = ants[k].number;
                                way.length = ants[k].length;
                                for (int i = 0; i < way.number; ++i) {
                                        way.array[i] = ants[k].array[i];
                                }
                        }

                }
                /* Цикл по ребрах */
                for (int i = 0; i < size; ++i) {
                        for (int j = 0; j < size; ++j) {
                                /* Оновлення феромона для ребра (i, j) з розрахунком на його випаровування */
                                if (i != j) {
                                        tao[i][j] *= (1 - p);
                                }
```

```
                        }
                }
        }

        /* Звільнення пам'яті */
        free(epsilon);
        free(tao);

        /* Повернення найкоротшого маршруту */
        return way;
}

/* Функція PrintResult                                          */
/* Призначення:                                                 */
/*      Вивід найкоротшого маршруту і його довжини на єкран     */
/* Вхідні дані:                                                 */
/*      way - найкоротший маршрут, кількість його вершин і довжина  */
/* Вихідні дані:                                                */
/*      way - найкоротший маршрут, кількість його вершин і довжина  */
/* Значення що повертає функція:                                */
/*      відсутнє                                                */

void PrintResult(ANT_TYPE way) {
        cout << "Путь: " << ++way.array[0];
        for (int i = 1; i < way.number; ++i) {
                cout << " -> " << ++way.array[i];
        }
        cout << "\nДлинна пути: " << way.length << endl;
}




/* Функція swap                                                 */
/* Призначення:                                                 */
/*      перестановка елементів масиву місцями                   */
/* Вхідні дані:                                                 */
/*      a - масив, індекси елементів                            */
```

```
/* Вихідні дані:                                    */
/*      a - змінений масив                          */
/* Значення що повертає функція:                    */
/*      відсутнє            */

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

/* Функція NextSet                                  */
/* Призначення:                                     */
/*      перестановка елементів масиву               */
/* Вхідні дані:                                     */
/*      a - масив, розмір масиву                    */
/* Вихідні дані:                                    */
/*      a - змінений масив                          */
/* Значення що повертає функція:                    */
/*      false або true                              */

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;
}

/* Функція  L                                       */
```

```
/* Призначення:                                                    */
/*      розрахунок довжини маршруту                                */
/* Вхідні дані:                                                   */
/*      a - масив, розмір масиву, індекс початкової точки, матриця відстаней */
/* Вихідні дані:                                                  */
/*      l - довжина знайденого маршруту                           */
/* Значення що повертає функція:                                  */
/*     l - довжина знайденого маршруту                            */

double L(int* a, int n, int city, double **D) {
        double l = D[city][a[0]];
        for (int i = 0; i+1 < n; i++) {
                l += D[a[i]][a[i + 1]];
        }
        l += D[a[n - 1]][city];
        return l;
}
int main()
{
        LARGE_INTEGER start, end;
        double res_time, t;
        srand(time(NULL));
        setlocale(LC_ALL, ".1251");
        double** D = NULL;
        int size = 0, city_1 = 0;
        char name[SIZE];

        printf("Введіть шлях файлу ");
        scanf_s("%s", name, SIZE);

        size = ReadSize(D, name, &size); //читання розміру матриці
        D = new double* [size];


        for (int t = 0; t < size; ++t) {
                D[t] = new double[size];
        }

        for (int k = 0; k < size; k++) {
```

```
        for (int r = 0; r < size; r++) {
            D[k][r] = 0.0;
        }
    }

    /* Заповнення масиву D */
    ReadMatrix(D, name, &size);

    /* Ініціалізація початкової точки */
    while (city_1 < 1 || city_1 > size) {
        cout << "Введите начальную вершину от 1 до " << size << ": ";
        cin >> city_1;
    }

    /* Використання мурашиного алгоритму */
    QueryPerformanceCounter(&start);
    t = double(start.QuadPart) / CLOCKS_PER_SEC;
    ANT_TYPE way = AntColony(D, size, --city_1);
    QueryPerformanceCounter(&end);
    res_time = double(end.QuadPart - start.QuadPart) / t;
    cout << "Мурашиний алгоритм\n";
    PrintResult(way);
    cout << "Час виконання алгоритму " << res_time << endl<<endl;

    int* way_p = new int[size-1];
    int* minway_p = new int[size];

    for (int i = 0, j = 0; i < size-1; i++, j++) {
        if (i == city_1) j++;
        way_p[i] = j;
       minway_p[i] = j;
    }
  minway_p[size - 1] = city_1;
    /* Використання повного перебору */
    double length = 0, minlength;
    QueryPerformanceCounter(&start);
    t = double(start.QuadPart) / CLOCKS_PER_SEC;
    minlength = length = L(way_p, size - 1, city_1, D);
    while (NextSet(way_p, size - 1)) {
```

```
            length = L(way_p, size - 1, city_1, D);
            if (length < minlength) {
                minlength = length;
                for (int i = 0; i < size - 1; i++) {
                    minway_p[i] = way_p[i];
                }
                minway_p[size - 1] = city_1;
            }
        }
        QueryPerformanceCounter(&end);
        res_time = double(end.QuadPart - start.QuadPart) / t;
        cout << "Простой перебор\n";
        cout << "Путь: " << ++city_1;
        for (int i = 0; i < size; ++i) {
            cout << " -> " << ++minway_p[i];
        }
        cout << "\nДлинна пути: " << minlength << endl;
        cout << "Время работы алгоритма: " << res_time << endl;
        system("pause");

        delete[] D;
        delete[]way_p;
        delete[]minway_p;

        return 0;
}
```