Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Факультет комп'ютерної інженерії та управління
(повна назва)

Кафедра електронних обчислювальних машин
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

Рівень вищої освіти другий (магістерський)

Методи рішення задачі комівояжера на основі обчислювального інтелекту
(тема)

Виконав:
студент II курсу, групи СПм-22-3
Онищенко О.І.
(прізвище, ініціали)

Спеціальність
123 «Комп'ютерна інженерія»
(код і повна назва спеціальності)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма
Системне програмування
(повна назва освітньої програми)

Керівник: доц. Іващенко Г.С.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри ЕОМ ______________ Коваленко А.А.
(підпис) (прізвище, ініціали)

2024 р.

Харківський національний університет радіоелектроніки

Факультет комп’ютерної інженерії та управління

Кафедра електронних обчислювальних машин

Рівень вищої освіти другий (магістерський)

Спеціальність 123 «Комп’ютерна інженерія»
(код і повна назва)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма Системне програмування
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри ________________
(підпис)
“____”________________ 20__ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студенту Онищенку Олександру Івановичу
(прізвище, ім’я, по батькові)

1. Тема роботи Методи рішення задачі комівояжера на основі обчислювального інтелекту

затверджена наказом по університету від “ 01 ” квітня 2024 р. № 257 Ст

2. Термін подання студентом роботи до екзаменаційної комісії 15 червня 2024 р.

3. Вхідні дані до роботи 1) алгоритми рішення ЗК на основі обчислювального інтелекту: генетичні, мурашині алгоритми, самоорганізуюча карта Кохонена; 2) бібліотека TSPLIB з готовими графами ЗК та відомими найкоротшими маршрутами.

4. Перелік питань, що потрібно опрацювати у роботі

1) огляд існуючих підходів до вирішення ЗК;
2) вибір та обґрунтування методики та засобів дослідження;
3) програмна реалізація алгоритмів;
4) проведення експериментальних досліджень;
5) висновки.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (слайдів)

Слайд-презентація – 17 слайдів

6. Консультанти розділів роботи (заповнюється за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Огляд існуючих досліджень	02.04.24-08.04.24	
2	Вибір та обґрунтування методики дослідження	09.04.24-16.04.24	
3	Вибір інструментальних засобів	17.04.24-22.04.24	
4	Розробка програмного застосунку	23.04.24-06.05.24	
5	Проведення експериментів	07.05.24-23.05.24	
6	Оформлення матеріалів кваліфікаційної роботи	24.05.24-03.06.24	
7	Подання кваліфікаційної роботи керівникові та	04.06.24-07.06.24	
	її попередній захист		
8	Подання кваліфікаційної роботи на рецензу-	08.06.24-12.06.24	
	вання		

Дата видачі завдання 01 квітня 2024 р.

Студент ____________________
(підпис)

Керівник роботи ____________________ доц. Іващенко Г.С.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи: 87 с., 13 рис., 3 табл., 2 дод., 37 джерел.

ЗАДАЧА КОМІВОЯЖЕРА, ОБЧИСЛЮВАЛЬНИЙ ІНТЕЛЕКТ, ЕВОЛЮЦІЙНІ АЛГОРИТМИ, МАРШРУТ, ГРАФ, ГЕНЕТИЧНИЙ АЛГОРИТМ, КРОСОВЕР, МУТАЦІЯ, СЕЛЕКЦІЯ, МУРАШИНИЙ АЛГОРИТМ, ШТУЧНІ НЕЙРОННІ МЕРЕЖІ, КАРТА КОХОНЕНА.

Метою кваліфікаційної роботи є експериментальне дослідження ефективності використання при вирішенні задачі комівояжера методів обчислювального інтелекту, зокрема, генетичних алгоритмів, мурашиних алгоритмів та самоорганізуючих карт Кохонена, та аналіз їх швидкодії при пошуці маршрутів обходу графів різних розмірностей.

У ході виконання кваліфікаційної роботи розроблено програмний застосунок із реалізаціями досліджуваних підходів та засобами автоматизації проведення експериментів. Виконано серії експериментів для різних варіантів генетичних та мурашиних алгоритмів та налаштувань їх вхідних параметрів з метою визначення підходу, що надає найкоротші маршрути за якомога менший час.

ABSTRACT

Master's thesis: 87 pages, 13 figures, 3 tables, 2 appendices, 37 sources.

TRAVELLING SALESMAN PROBLEM, COMPUTATIONAL INTELLIGENCE, EVOLUTIONARY ALGORITHMS, ROUTE, GRAPH, GENETIC ALGORITHM, CROSSOVER, MUTATION, SELECTION, ANT COLONY OPTIMIZATION, ARTIFICIAL NEURAL NETWORKS, KOHONEN'S SELF-ORGANIZING MAP.

The major goal of this thesis is an experimental research of the efficiency of using computational intelligence methods for solving the traveling salesman problem, in particular, using genetic algorithms, ant colony optimization algorithms, and Kohonen's self-organizing maps, and an analysis of their performance for searching of routes with a various number of graph vertices.

In the course of the qualification work, a software application with implementations of the researched approaches and instruments of automating experiments was developed. A series of experiments were performed for different variants of genetic and ant algorithms and settings of their input parameters in order to determine the approach that provides the shortest routes in the shortest possible time.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ 8
ВСТУП 9
1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ 10
1.1 Опис задачі комівояжера 10
1.1.1 Актуальність вирішення ЗК 11
1.1.2 Математична модель ЗК 12
1.2 Аналіз існуючих рішень 14
1.3 Постановка задачі 17
2 ВИКОРИСТОВУВАНІ МЕТОДИ ВИРІШЕННЯ ЗК 18
2.1 Генетичні алгоритми 18
2.1.1 Оператор кросоверу 19
2.1.2 Оператор мутації 21
2.1.3 Оператор селекції 22
2.1.4 Ініціалізація параметрів генетичних операторів 25
2.1.5 Аналіз збіжності ГА 25
2.2 Мурашиний алгоритм 26
2.3 Самоорганізуюча карта Кохонена 28
3 ПРОГРАМНА РЕАЛІЗАЦІЯ ЗАСТОСУНКУ 32
3.1 Загальний огляд структури програмного рішення 32
3.2 Програмне представлення сутностей задачі комівояжера 34
3.3 Реалізація генетичного алгоритму 36
3.3.1 Особливості реалізації генетичних операторів 36
3.3.2 Опис загальної структури алгоритму 38
3.4 Реалізація мурашиного алгоритму 40
3.4.1 Опис загальної структури програмного коду 40
3.4.2 Реалізація обходу графу агентами 43

3.5 Реалізація самоорганізуючої карти Кохонена 45
3.5.1 Послідовність дій алгоритму 45
3.5.2 Опис базового типу SOM 45
3.5.3 Адаптація SOM для вирішення задачі комівояжера 46
3.5.4 Вирішення проблеми відгалужень на маршруті 49
3.6 Структура засобів автоматизації експериментів 50
3.6.1 Налаштування та запуск досліду 50
3.6.2 Збереження результатів експерименту 52
4 РЕЗУЛЬТАТИ ЕКСПЕРИМЕНТАЛЬНИХ ДОСЛІДЖЕНЬ 54
4.1 Налаштування параметрів для проведення досліджень 54
4.2 Дослідження збіжності генетичних алгоритмів 54
4.3 Дослідження використання генетичних алгоритмів для вирішення задачі комівояжера 56
4.4 Дослідження мурашиного алгоритму 59
4.5 Дослідження використання самоорганізуючої карти Кохонена 60
4.6 Аналіз результатів досліджень 62
4.7 Можливі напрямки подальших досліджень 63
ВИСНОВКИ 64
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 65
ДОДАТОК А Графічний матеріал кваліфікаційної роботи 69
ДОДАТОК Б Вихідний код програмного забезпечення 79

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ГА – генетичний алгоритм

ЗК – задача комівояжера

ООП – об’єктно-орієнтоване програмування

МА – мурашиний алгоритм

ШНМ – штучні нейронні мережі

BMU – best matching unit

CTSP – constrained traveling salesman problem

CX – cycle crossover

GA – genetic algorithm

IOX – inver-over crossover

OX – order crossover

OBX – order-based crossover

PMX – partially-mapped crossover

SOM – self-organizing map

SpOX – single point ordered crossover

TSP – traveling salesman problem

ВСТУП

Зростання кількості підприємств, які спеціалізуються на логістиці, такі, як поштові служби, таксі, торгівельні компанії, логістичні відділи промислових підприємств, призводить до зростання обсягу транспорту на дорогах і появи заторів. Через це виникає необхідність планування великої кількості складних маршрутів, і зростає потреба в ефективних методах вирішення проблем, пов'язаних з маршрутизацією. Серед таких проблем однією з найважливіших є знаходження найкоротших транспортних маршрутів з метою принесення максимального доходу при мінімальних витратах, що в комбінаторній оптимізації відомо як задача комівояжера (ЗК).

У зв'язку з високою обчислювальною складністю ЗК та її приналежністю до класу NP-повних задач, точні методи розв'язання є незастосовними для задач високої розмірності через неможливість отримання рішення за прийнятний час. За цієї причини, для вирішення ЗК використовуються евристичні алгоритми, які дають змогу знаходити рішення задачі комівояжера за задовільний час, але при цьому не гарантують знаходження найкоротшого маршруту, надаючи рішення, які відповідають шляхам обходу, довжина яких лише наближена до найменшого можливого значення. В умовах зростання потреби в збереженні часу, матеріальних та інтелектуальних ресурсів, актуальною є потреба у розробці більш ефективних та швидких алгоритмів для пошуку кращих рішень ЗК. Перспективним в цьому напрямку є використання методів обчислювального інтелекту, що базуються на природних явищах (нейронні мережі, еволюційні процеси, ройовий інтелект тощо) [1].

У роботі досліджені підходи до рішення ЗК на основі обчислювального інтелекту. Розглянуто перспективні напрямки для вирішення задачі комівояжера, такі як генетичні алгоритми, мурашині алгоритми та штучні нейронні мережі, зокрема самоорганізуюча карта Кохонена.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Опис задачі комівояжера

Задача комівояжера є однією з найвідоміших та найбільш вивчених задач комбінаторної оптимізації. Класична постановка цієї задачі полягає в пошуці найкоротшого маршруту на множині точок (міст), який включає в себе всі задані міста лише один раз і повертається в початкову точку.

Зростання актуальності проблеми пошуку найкращих маршрутів спостерігалося з давніх часів із розвитком торгівлі між віддаленими населеними пунктами. Серед наукових праць, першою згадкою про проблему, подібну до ЗК, є задача про хід шахового коня в роботі Леонарда Ейлера, яка датується 1759 роком [2]. В 1832 році була видана книга «Комівояжер – як він повинен поводитися і що повинен робити для того, щоб доставляти товар і мати успіх у своїх справах – поради старого кур'єра», в якій була фактично описана ЗК, але математичний апарат для її вирішення на той час ще не застосовувався [3]. Перша згадка ЗК як оптимізаційної задачі належить математику Карлу Менгеру, який на математичному колоквіумі 1932 року назвав її «проблемою посильного», що полягає в пошуці найкоротшого шляху, який об'єднує скінченну множину точок, відстані між якими є відомими [4]. Найбільш раннє відоме використання вже сучасної назви «задача комівояжера» в контексті математичної оптимізації зустрічається в роботі Дж. Б. Робінсон «Про Гамільтонову гру» (1949), де розглядалася задача пошуку найкоротшого маршруту між столицями штатів США, починаючи з Вашингтону [5]. В 1972 році, Річард Карп довів NP-повноту задачі пошуку гамільтонових шляхів, звідки випливає приналежність ЗК до класу NP-повних задач [6]. Важливим етапом у розвитку методів вирішення задач великих розмірностей задокументований в роботі Кровдера і Падберга, які в 1980 році запропонували вирішення

симетричної ЗК з 318 містами комбінованим підходом із методами січних площин і гілок та меж [7], що залишалося найбільшою вирішеною задачею до 1987 року, коли було знайдено рішення для задачі вже з 532 вершинами [8], а у 1991 рекорд досяг 1000 вершин [9]. У 2006 році було знайдене рішення задачі комівояжера для графу, який налічує 85900 вершин, що досі залишається найбільшим вирішеним екземпляром задачі комівояжера [10].

1.1.1 Актуальність вирішення ЗК

Задача комівояжера залишається актуальною в сучасному світі через постійне зростання складності процесів та обсягів даних у різних сферах діяльності, де ЗК має практичні застосування [11], таких як логістика, телекомунікації, обчислювальні системи тощо.

У задачах логістики та транспорту задача комівояжера використовується для визначення кращих маршрутів транспортних засобів, доставки вантажів та обслуговування клієнтів.

В телекомунікаційній галузі планування маршрутів мережі для забезпечення максимальної ефективності та мінімізації затримок є однією з важливих задач. Застосування методів рішення ЗК дозволяє підвищити якість обслуговування клієнтів та заощадити використання ресурсів.

Вирішення задачі пошуку найкоротшого маршруту допомагає при організації виробничих процесів, зокрема, вигідного розміщення устаткування з метою зменшення часових затримок та полегшення обслуговування обладнання.

Розв'язання задачі комівояжера є одним з розповсюджених тестових завдань для перевірки ефективності та швидкодії різних алгоритмів та обчислювальних методів. Вирішення цієї задачі допомагає вдосконалювати алгоритми оптимізації та машинного навчання.

Таким чином, актуальність задачі комівояжера та розробки більш точних та швидких методів її вирішення полягає у великому практичному

значенні задачі в різних сферах діяльності, зростанні обсягів даних та необхідності заощадження ресурсів.

1.1.2 Математична модель ЗК

Математично ЗК визначається наступним чином: нехай задано повний орієнтований граф *G = (V, E)*, який має *N* вершин. *V* – множина вершин (міст), *E* – множина дуг (зв’язків між містами):

$$V = \{v_0, v_1, ..., v_{n-1}\}, \quad (1.1)$$

$$E = \{v_i, v_j \in V \mid i \neq j\}, \quad (1.2)$$

де матриця $C = |c_{ij}|$ розмірністю $N * N$, c_{ij} відповідає вартості (відстані) переходу з між вершинами *i* та *j*. Варіантом рішення ЗК вважається шлях обходу графу, який визначається як матриця $X = |x_{ij}|$, $x_{ij} \in \{0, 1\}$, де значення 1 свідчить про наявність переходу з вершини *i* до вершини *j* у знайденому циклі, 0 задається в протилежному випадку. Рішення ЗК полягає в знаходженні маршруту, який має найменшу можливу сумарну вартість переходів:

$$\sum_{i=i}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} \to \min, \quad (1.3)$$

де кожна вершина включена до маршруту тільки один раз, тобто має тільки одну вхідну, та тільки одну вихідну дуги:

$$\sum_{j=1, j \neq i}^{n} x_{ij} = 1, (i, j = \overline{1, n}), \quad (1.4)$$

$$\sum_{i=1, i \neq j}^{n} x_{ij} = 1, (i, j = \overline{1, n}). \tag{1.5}$$

При цьому, має існувати лише один шлях, що включає одразу всі вершини графа, що визначається умовою:

$$u_i - u_j + nx_{ij} \leq n - 1, i, j = \overline{1, n}, i \neq j. \tag{1.6}$$

1.1.3 Класифікація ЗК

За довгий час досліджень проблеми пошуку найкоротших маршрутів, з’явився ряд варіантів задачі комівояжера із певними відмінностями в постановці, що наближає модель ЗК до умов тієї чи іншої прикладної проблеми та забезпечує передумови для розробки максимально ефективних методів її вирішення.

Одним із поширених різновидів задачі є симетрична ЗК [12], в якій вартість переходу між двома вершинами не залежить від напрямку. Спираючись на розглянуту математичну постановку, даний вид задачі можна описати додатковою умовою

$$c_{ij} = c_{ji}, i, j = \overline{1, n}, i \neq j, \tag{1.7}$$

що робить симетричну ЗК зручною моделлю для вирішення проблем в декартовій системі координат в Евклідовому просторі: планування виробничих ліній на підприємствах, логістика пошуку і розміщення товарів на складах тощо.

Іншою різновидністю є асиметрична ЗК [13], яка допускає різну вартість дуги між двома вершинами графу в залежності від напрямку здійснення переходу. Такий підхід є актуальним для проблеми прокладання

кабелів в електронних обчислювальних пристроях [11], в транспортних задачах, де існує потреба в ресурсах для пересування (палива) і відвідуванні місць їх поповнення, урахування розкладу рейсів різних видів транспорту при плануванні туристичних поїздок та інші.

Поняття задачі комівояжера з обмеженнями (CTSP – constrained traveling salesman problem) [14] об’єднує в собі цілий клас різновидів ЗК, які накладають різні типи обмежень до початкової постановки задачі. Це дозволяє враховувати додаткові умови досліджуваної проблеми, серед яких є як особливості процесу переміщення знайденим маршрутом, так і специфіка області застосування. До таких обмежень можуть бути віднесені: часові вікна або кінцеві терміни відвідування міст, обмеження місткості для пасажирських перевезень, або ж об’єму чи маси у випадку вантажоперевезень, урахування динамічних перешкод на маршрутах, викликаних змінами у трафіку (пробки, ремонтні роботи, аварії) або погодними умовами тощо.

1.2 Аналіз існуючих рішень

На сьогоднішній день, відомі дві основних категорії алгоритмів вирішення ЗК: точні та евристичні.

Точні алгоритми, такі, як повний перебір і метод «гілок та меж» [15], надають чітку послідовність дій для пошуку гарантовано найкоротшого шляху, але з огляду на NP-повноту та факторіальну складність задачі, вони є недоцільними для пошуку на графах розмірністю вище за декілька десятків вершин.

Евристичні алгоритми в свою чергу не гарантують знаходження найкращого маршруту, але прикладні методи в їх основі дозволяють знайти наближене рішення за прийнятний час, що робить можливим їх використання на графах розмірністю в сотні та тисячі вершин. До таких можна віднести імітацію відпалу [16], метод найближчого сусіда [17], та методи

обчислювального інтелекту, які включають еволюційні алгоритми, імітацію ройового інтелекту, штучні нейронні мережі тощо.

Використання точних методів вирішення ЗК є недоцільним в умовах постійного зростання вимог до розмірності графів та швидкості пошуку шляху. За цієї причини, основним вектором досліджень з пошуку нових та вдосконалення вже існуючих методів пошуку рішень ЗК наразі є евристичні алгоритми, зокрема, на основі методів обчислювального інтелекту.

Одним з широко використовуваних підходів до вирішення ЗК є генетичні алгоритми (ГА), які дозволяють пришвидшити пошук за рахунок розгляду обмеженої множини можливих рішень замість виконання перебору великої кількості маршрутів, мають здатність ітеративно покращувати знайдені рішення шляхом модифікації вже наявних та відбору найкращих варіантів маршрутів. Широкі можливості модернізації та проведення експериментів зумовлює вибір ГА предметом досліджень багатьох актуальних наукових робіт присвячених вирішенню ЗК. Встановлено, що підхід до реалізацій операторів генетичного алгоритму (кросоверу, селекції, мутації), а також конфігурування вхідних параметрів (ймовірність мутації, розмір популяції тощо) мають значний вплив на швидкість роботи алгоритму та довжини отримуваних шляхів [18]. Крім того, залишається актуальною проблема передчасної збіжності алгоритму [19], яка виражається в зниженні різноманіття генотипів популяції, і таким чином може негативно позначається на точності та швидкості алгоритму. Враховуючи вказані проблеми, існує потреба в реалізації різних генетичних операторів, ретельній експериментальній перевірці їх комбінацій та розробці засобів відстеження збіжності алгоритму та підтримання необхідного рівня різноманіття генотипів, зокрема за допомогою генетичного оператора мутації.

Іншим напрямком вирішення ЗК є використання мурашиного алгоритму, який являє собою модель багатоагентної системи, де результат роботи кожного агенту впливає на діяльність інших, і таким чином поступово надається перевага найкращим маршрутам. Можливість визначати та

налаштовувати гетерогенну поведінку для агентів [20] дозволяє знаходити кращі рішення ЗК. Іншим шляхом до покращення результативності алгоритму є зміни в порядку оновлення феромону на маршрутах, а також підбір початкових умов [21]. Зокрема, високу чутливість до змін мають такі параметри, як інтенсивність феромону агента та коефіцієнт випаровування феромону. Також при формуванні шляхів важливим є встановлення балансу між прагненням агента до вибору найкоротших переходів та спиранням на концентрацію феромону. Результатом невірно налаштованої поведінки агентів може стати перетворення імітації мурашиної колонії на звичайний жадібний алгоритм в першому випадку, або швидка збіжність у локальному мінімумі в другому випадку, що обмежує можливості алгоритму в пошуці найкоротших маршрутів.

Перспективним методом для вирішення ЗК також є використання штучних нейронних мереж (ШНМ), як загалом ефективного та широко застосовуваного методу аналізу та інтерпретації даних. Здатність ШНМ адаптуватися до складних структур даних зумовлює результативність такої архітектури, як самоорганізуюча карта Кохонена, у вирішенні ЗК. Використання топології замкнутого кільця [22] дає змогу застосовувати мережу при пошуку маршрутів для ЗК, але проблема появи окремих відгалужень на маршруті в процесі виконання алгоритму перешкоджає знаходженню найкоротших шляхів. Окрім невеликих відгалужень на пізніх етапах навчання мережі [23], можлива поява й більш масштабних подовжень маршруту ще на початку роботи алгоритму, що робить важливим аспектом вибір функції сусідства та швидкості переходу алгоритму від глобальних до локальних змін.

В результаті проведеного аналізу існуючих досліджень, зроблено висновок, що результативність описаних підходів наразі є недостатньою, та існують ще не задіяні напрямки вдосконалення, що потребують більш детального експериментального дослідження.

1.3 Постановка задачі

Метою роботи є дослідження особливостей використання підходів обчислювального інтелекту, таких як генетичні алгоритми, мурашині алгоритми та самоорганізаційна карта Кохонена для вирішення задачі комівояжера. Особлива увага має бути приділена дослідженню впливу параметрів налаштувань зазначених методів на швидкість пошуку та довжини знайдених шляхів обходу графів великого розміру. Для порівняння точності алгоритмів, вирішено використовувати готові екземпляри симетричної ЗК із заздалегідь визначеними найкоротшими маршрутами, що надаються відкритою бібліотекою TSPLIB [24].

Для проведення експериментів у ході дослідження необхідно створити програмний застосунок, який включатиме наступні компоненти та буде підтримувати такі можливості:

- засоби для роботи з графами ЗК, що включають можливість їх завантаження з бібліотеки TSPLIB, або формування випадкових графів із заданої кількістю вершин;
- пошук рішення на завантаженому чи згенерованому графі;
- запуск експериментів з метою дослідження впливу окремих параметрів налаштувань на ефективність роботи алгоритмів, із подальшим збереженням результатів у файл. Інформація про проведений експеримент має містити його повну конфігурацію: різновид алгоритму, що запускався, опис окремих його компонентів (якщо є), значення вхідних параметрів.

Проведення досліджень включає отримання застосунком даних, які дають змогу оцінити ефективність різних методів рішення ЗК (час роботи, вартість знайдених маршрутів, швидкість збіжності алгоритму тощо) для різних розмірностей графів, провести аналіз і отримати порівняльну характеристику результатів експериментів. Використання готових графів із заздалегідь відомим найкоротшим маршрутом дозволить об’єктивно порівнювати точність реалізованих алгоритмів пошуку шляхів обходу.

2 ВИКОРИСТОВУВАНІ МЕТОДИ ВИРІШЕННЯ ЗК

2.1 Генетичні алгоритми

Генетичний алгоритм є імітацією природних еволюційних процесів та явищ генетики [25], що розглядає варіанти рішень оптимізаційних задач як особини, які мають генотип – множину характеристик, і відрізняються один від одного їх значеннями (станами). Ключову роль у алгоритмі має поняття функції пристосованості (fitness function), яка приймає в якості аргументу генотип особини, і повертає певне кількісне значення, яке дозволяє оцінити пристосованість особини та визначити ймовірність її потрапляння до наступного покоління – ітерації алгоритму.

У загальному випадку, ГА працює з обмеженим набором особин – поколінням, початкові генотипи яких задаються випадковим чином. Етапи ГА включають використання генетичних операторів – алгоритмів, що базуються на реальних явищах генетики. До таких операторів належить кросовер (схрещування), який відповідає за формування нових особин на основі генотипів батьківських особин, імітуючи процес розмноження популяції. Тип кросовера визначає, якою буде послідовність дій формування нових генотипів. Іншим важливим оператором є мутація, метою якої є підтримання різноманіття генотипу популяції та запобігання передчасної збіжності алгоритму, що досягається шляхом внесення випадкових змін до фрагментів генотипу чи окремих генів. Зазвичай мутація відбувається з певною ймовірністю, яка задається окремим вхідним параметром. Третім стандартним оператором ГА є селекція (відбір) – обмеження популяції до заданого розміру в кінці ітерації, після утворення нових дочірніх особин. Як і в природних еволюційних процесах, алгоритм селекції спирається на рівень пристосованості особини, забезпечуючи перехід в нове покоління тільки найкращих чи найбільш перспективних рішень. Як окремий етап ГА іноді

також виділяють процес формування пар із відсіяних в ході селекції осіб, на основі яких утворюватимуться нові особини. Генетичний алгоритм може бути успішно застосований для пошуку наближених рішень ЗК за виконання ряду умов. Особинами слід вважати маршрути, які відносяться до множини рішень ЗК, тобто такі, що відповідають умовам (1.4-1.6). Для кодування генів обрано шляхове представлення – генотипом особини вважатиметься перелік вершин маршруту в порядку їх обходу. За показник пристосованості приймається довжина шляху. Відповідно до (1.3), пристосованість особини є обернено пропорційною до довжини її маршруту.

2.1.1 Оператор кросоверу

Оператор кросоверу є моделлю генетичного процесу схрещування особин, який в процесі вирішення ЗК виступає засобом формування нових рішень, серед яких має бути якомога більше маршрутів коротших за отримані в попередньому поколінні чи в початковій популяції. Згідно (1.4-1.5), для особин, гени яких кодуються шляховим представленням, важливо використовувати такі алгоритми кросоверу, що враховують необхідність містити в генотипі всі наявні вершини ЗК і не допускати їх повторень. За цієї причини, формування нащадків найчастіше включає в себе два етапи: одна частина генотипу береться з певним чином обраних позицій з генотипу першої батьківської особини, після чого відбувається дозаповнення решти тими генами з другої особини, що не були задіяні в першому етапі. Здійснення цих дій двічі зі зміною порядком розгляду батьківських особин дозволяє формувати з кожної обраної пари по два нових варіанти рішення.

Одним з таких алгоритмів є реорганізуючий кросовер (OBX – order-based crossover) [26], принцип роботи якого полягає у випадковому виборі кількості позицій та їх номерів, в які записуються гени першої батьківської особини, а решта заповнюється бракуючими генами з другої особини пари у порядку їх слідування в генотипі (рисунок 2.1).

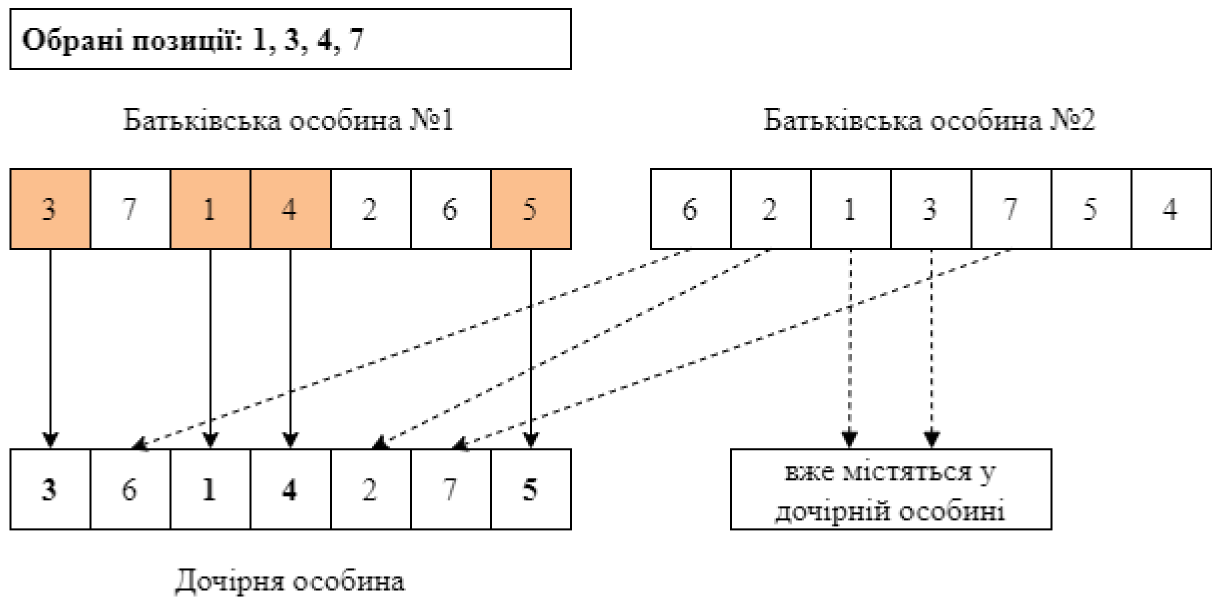

Рисунок 2.1 – Приклад роботи кросоверу OBX

Іншим прикладом є кросовер з інверсією за опорним елементом (inver-over crossover, IOX) [27], який утворює нові особини на основі генотипу тільки однієї батьківської особини і змінює його шляхом інверсії фрагменту. Від генотипу другої особини-предка залежить довжина та розташування фрагменту, для якого виконуватиметься інверсія. Кроки виконання алгоритму описуються наступним чином (рисунок 2.2):

- у першій особині P_1 випадково обирається одна вершина, номер позиції якої позначається як s_1;
- отримується значення обраної вершини $P_1(s_1) = g$;
- у другій особині, визначається номер позиції k, такий, що $P_2(k) = g;$
- визначається номер позиції s_2 такий, що $P_1(s_2) = P_2(k + 1)$;
- утворюється новий нащадок: генотип батьківської особини P_1 копіюється в нову особину, після чого здійснюється інверсія для фрагменту між позиціями з номерами s_1 та s_2.

З метою більш детального дослідження ефективності генетичного алгоритму у вирішенні ЗК, окрім описаних кросоверів в роботі також розглядалися такі, як: упорядкований (Order crossover, OX) [28],

одноточковий порядковий (Single Point Ordered crossover, SpOX) [29], кросовер із частковим відображенням (Partially-Mapped Crossover, PMX) [30] та циклічний кросовер (Cycle crossover, CX) [31].

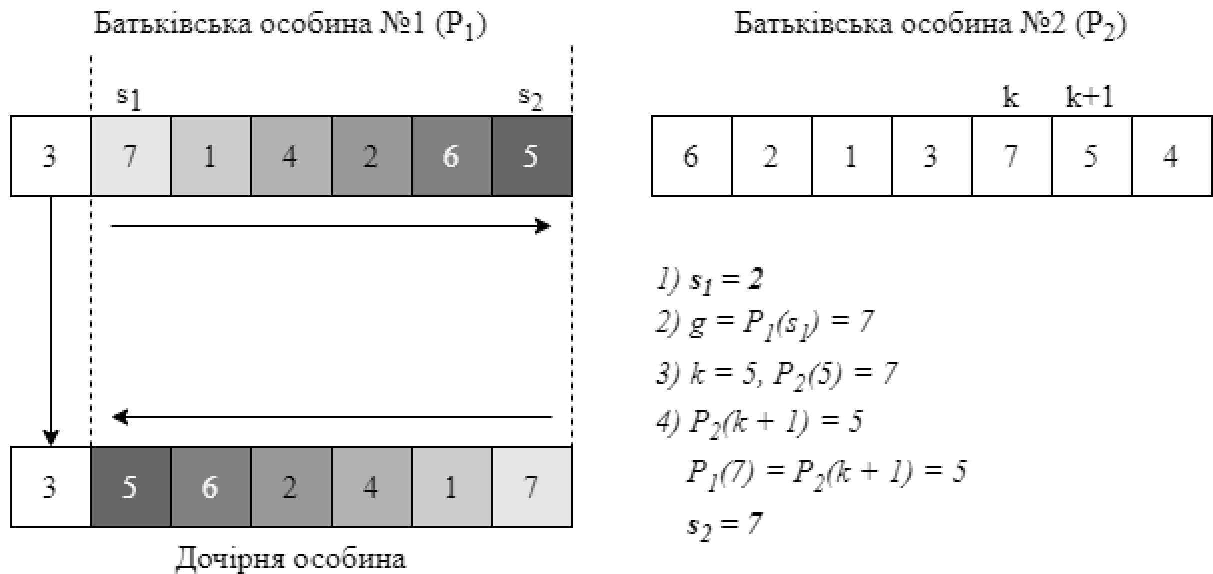

Рисунок 2.2 – Приклад роботи кросоверу Inver-over

2.1.2 Оператор мутації

Мутації є ще одним етапом алгоритму, що бере за основу реальне явище – випадкові зміни в генотипі, які можуть призводити до появи більш пристосованих особин (кращих рішень) і збільшують різноманіття генотипів популяції, затримуючи збіжність алгоритму. Ця властивість є особливо актуальною у випадку ГА, оскільки множина можливих станів генотипу та розміри популяції значно поступаються тим, що можна спостерігати в природних генетичних процесах, що призводить до швидких темпів збіжності.

Для оцінки впливу різних підходів до зменшення різноманіття рішень, в роботі реалізовано декілька варіантів мутацій, такі як: обмін двох фрагментів генотипу (swap), зсув фрагменту (shift) та інверсія фрагменту (inversion) маршрутів (рисунок 2.3).

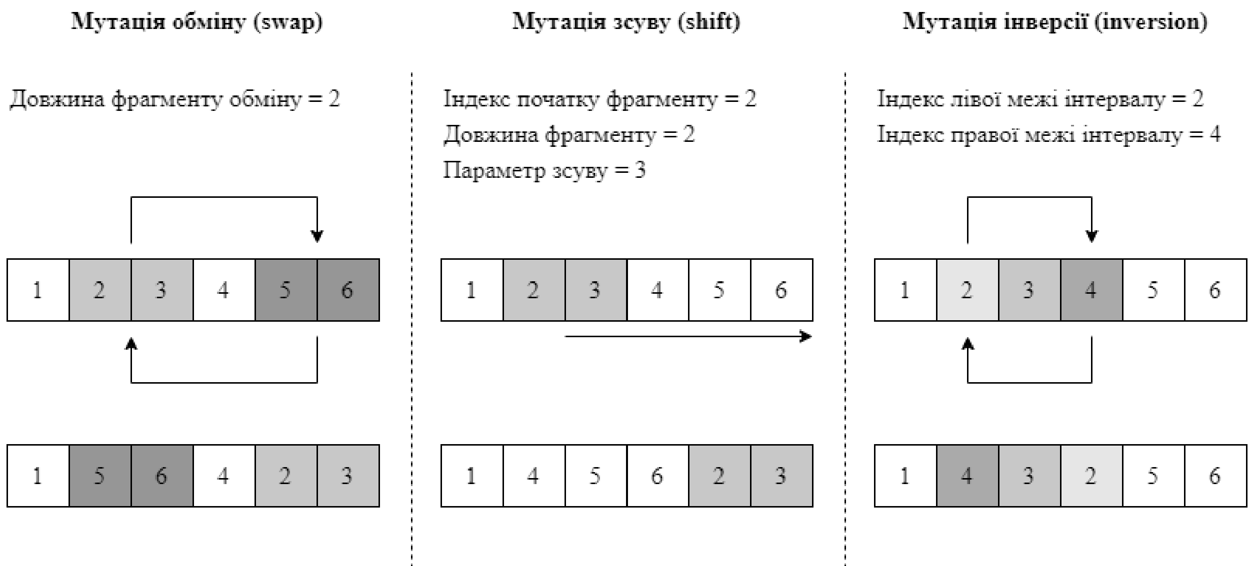

Рисунок 2.3 – Опис роботи використовуваних мутацій

Мутація обміну являє собою простий обмін місцями двох випадкових фрагментів генотипу, довжина яких може бути налаштована відповідним параметром, і знаходитися в інтервалі від 1 вершини (точковий обмін), до половини довжини маршруту. Мутація зсуву обирає один фрагмент генотипу і встановлює його на нове місце, переміщуючи повз інші елементи маршруту. Робота даної мутації налаштовується за допомогою параметрів, що задають фрагмент (індекс початку фрагменту та його довжина), а також параметр опису процесу зсуву, модуль якого позначає кількість одиниць для переміщення, а знак – напрямок зсуву (додатне число – праворуч, від’ємне – ліворуч). Мутація інверсії оперує двома параметрами – індексами лівої і правої межі, які визначають фрагмент, вершини змінять свій порядок у маршруті.

2.1.3 Оператор селекції

Операція селекції у ГА імітує процес зміни поколінь, що включає відбір найбільш пристосованих особин і відсіювання всіх інших, поступово покращуючи пристосованість популяції та підтримуючи її постійний розмір.

В даній роботі вирішено модифікувати класичний процес селекції, розділивши його на два етапи: підготовка, що включає процес формування пар особин, для подальшого утворення нових рішень в ході виконання кросоверу, та відбір – скорочення популяції до початкової кількості, в результаті якого залишаються тільки особини з найвищим показником пристосованості. Додатково передбачена можливість вибору стратегії формування нового покоління – відбір N найкращих із об’єднаної множини особин поточного покоління та їх нащадків, або формування нового покоління виключно з нових особин.

Перша реалізація етапу підготовки використовує метод рулетки, що полягає в випадковому виборі особини із деякою ймовірністю, що залежить від її показника пристосованості. Назва методу зумовлена тим, що цей процес можна порівняти із умовною рулеткою, яка має сектори різного розміру та стрілку в центрі. Вибір рішення може бути описаний як процес розкручування стрілки, що після зупинки вкаже на один із секторів. Згідно постановки ЗК (1.2), слід враховувати, що ймовірність вибору рішення має бути обернено пропорційною до довжини маршруту (рисунок 2.4).

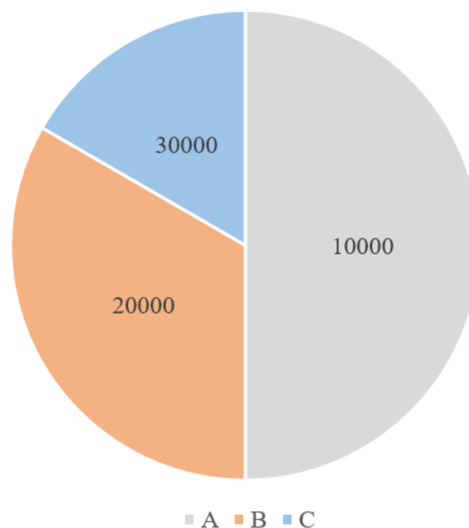

Рисунок 2.4 – Розподілення ймовірностей вибору в рулетці для особин «А», «В», «С», з довжинами маршрутів 10000, 20000 та 30000 відповідно

Першим кроком алгоритму формування пар методом рулетки є опис множини ймовірностей вибору рішень $R = \{D_i^{-1}\}$, де D_i – довжина і-го маршруту. Далі формуються сектори рулетки, які описуються їх лівими (2.1) та правими (2.2) межами

$$B_1 = \sum_{k=1}^{i-1} R_k , \tag{2.1}$$

$$B_2 = B_1 + R_i , \tag{2.2}$$

де i – особина, для якого будується сектор. Вибір першої особини пари здійснюється наступним чином: генерується випадкове число на проміжку $[0; \sum_{k=1}^{N} R_k]$, де N – розмір популяції, після чого по вирахуваним межам секторів, встановлюється конкретна особина із номером m. Для недопущення повторного вибору цієї особини до пари, вона виключається з рулетки шляхом коригування меж секторів для всіх рішень, що йдуть за першим обраним (2.3)

$$\begin{aligned} &\forall i \geq m, \\ &B_1' = B_1 - R_m, \\ &B_2' = B_2 - R_m. \end{aligned} \tag{2.3}$$

Вибір другої батьківської особини відбувається аналогічним чином, але з урахуванням виключеного першого обраного рішення, наступне випадкове число має лежати на проміжку $[0; \sum_{k=1}^{N} R_k - R_m]$.

Інший розглянутим підходом до формування пар є турнірний метод, що полягає у випадковому виборі довільного числа кандидатів (найчастіше двох), після чого з них обирається один з найкращим значенням фітнес-функції. Даний метод застосовується для вибору кожного рішення до нової

батьківської пари з урахуванням додаткових умов по недопущенню повторень при виборі другої особини.

2.1.4 Ініціалізація параметрів генетичних операторів

З метою дослідження впливу порядку задання вхідних параметрів різних операторів ГА на отримувані результати, вирішено застосовувати такі способи ініціалізації цих параметрів, як:

- «manual» (ручне внесення) – значення параметра задається користувачем один раз і більше не змінюється протягом роботи алгоритму;
- «one time» – параметр також встановлюється тільки один раз, але задається випадковим значенням;
- «every generation» – нове випадкове значення задається для кожного нового покоління;
- «every individual» – нове випадкове значення задається для кожної особини (або пари особин у випадку кросовера) протягом кожного покоління роботи алгоритму.

2.1.5 Аналіз збіжності ГА

Однією з головних проблем при розгляді ГА є поступове зменшення різноманіття генотипу популяції в процесі роботи алгоритму, що виражається в появі особин з однаковим генотипом. Цей процес суттєво гальмує чи зовсім зупиняє пошук нових, кращих рішень. Швидкість розвитку цього явища може залежати від кількості вершин ЗК, розміру популяції, числа поколінь, ймовірності мутації особливостей конкретних реалізацій операторів ГА тощо. З метою аналізу та пошуку методів запобігання раннього виникнення передчасної збіжності, в роботі запропоновано критерій оцінки – показник виродження популяції D, і два варіанти його визначення – лінійна (2.4) та квадратична функції (2.5), де A – кількість

унікальних генотипів у популяції, N – загальний розмір популяції. Виродження складає 0, якщо $A = N$ (повторення генотипів в популяції відсутні), і збільшується до 1 зі зростанням кількості повторюваних рішень (рисунок 2.5).

$$D = \frac{A-1}{N-1}, A = \overline{1,N}, D \in [0;1], \tag{2.4}$$

$$D = \frac{A^2-1}{N^2-1}, A = \overline{1,N}, D \in [0;1]. \tag{2.5}$$

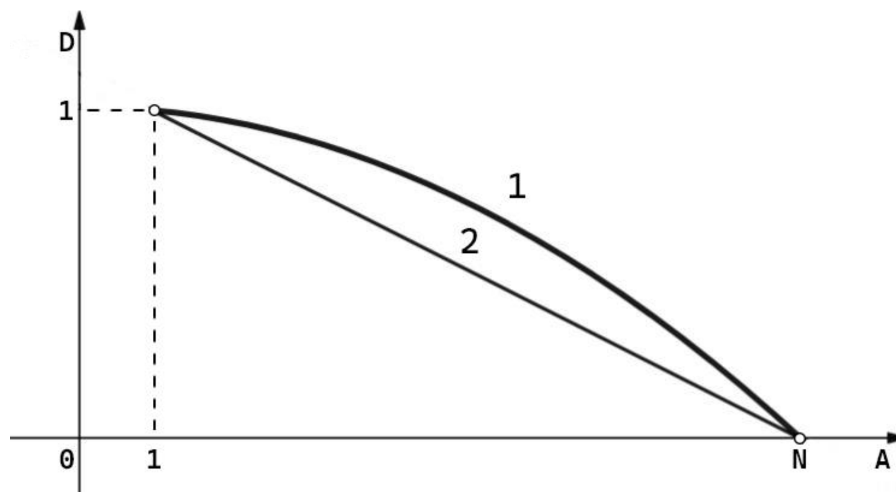

Рисунок 2.5 – Графіки квадратичної (1) та лінійної (2) функцій оцінки виродження популяції ГА

2.2 Мурашиний алгоритм

Другим розглянутим в роботі підходом до вирішення ЗК є мурашиний алгоритм (МА), в основі якого поведінка однойменних комах, які здатні ефективно знаходити найкоротші шляхи до джерел їжі при колективній взаємодії. Важливу роль в цьому процесі має позначення найкоротших маршрутів спеціальною речовиною – феромоном, концентрація якої впливає на вибір шляху агентом (мурахою). Феромон має здатність до випаровування, тому при проходженні різними маршрутами однаковою кількістю агентів, коротші шляхи зберігатимуть вищу концентрацію речовини, що

зумовлюватиме більш ймовірний їх вибір у наступних ітераціях (рисунок 2.6) Рівень феромону на довших маршрутах навпаки наближатиметься до повного випаровування, і таким чином, переважна більшість особин колонії поступово виділятиме найкоротші шляхи [32].

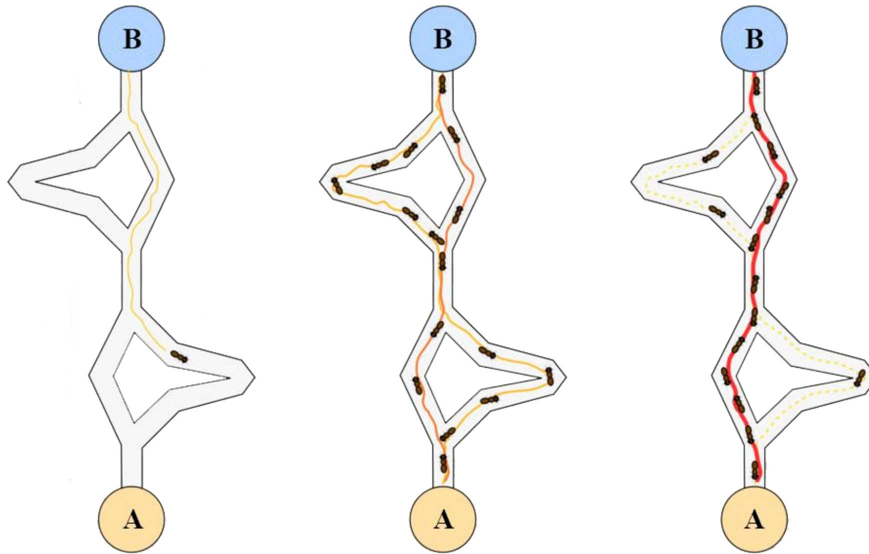

Рисунок 2.6 – Пошук найкоротшого шляху мурашиною колонією із позначенням інтенсивності феромону

В основі процесу формування нового рішення лежить прокладання шляху агентом по переходам між вершинами, з поступовим додаванням нових вершин, ймовірність вибору яких визначається як

$$p_{ij} = \frac{t_{ij}^{\alpha} + \frac{1}{w_{ij}^{\beta}}}{\sum_{l \in S_i} (t_{il}^{\alpha} + \frac{1}{w_{il}^{\beta}})}, \tag{2.6}$$

де i – поточна вершина, в якій розташований агент, j – вершина, що розглядається для вибору, S_i – множина ще не відвіданих вершин, t – кількість феромону на переході, w – довжина переходу, α та β – регульовані

параметри, що при виборі переходу визначають пріоритетність феромону чи вартості переходу відповідно.

В кінці кожної ітерації, після побудови маршрутів усіма агентами, кількість феромону на переходах оновлюється згідно правила:

$$\tau_{ij}^{'} = (1-\rho)\tau_{ij} + \frac{k}{L}, \qquad (2.7)$$

де τ_{ij} – попередня кількість феромону на переході між вершинами і та j, $\rho \in [0;1]$ – коефіцієнт випаровування феромону, k – кількість феромону, що залишає агент, L – довжина переходу.

Використовувана в дослідженні реалізація МА включає такі підходи, як алгоритм Ant System [33], Ant Colony System, головною особливістю якого є виділення класу елітарних агентів для закріплення найкращих шляхів [34] та методу Min-Max, орієнтованого на спадання загального рівня феромону на всьому просторі маршрутів, залишаючи тільки найкоротші шляхи [35].

Наявність пам'яті в кожного агента, яка зберігає інформацію про вершини графу, пройдені протягом однієї ітерації, дозволяє визначати ще не відвідані вершини, чим задовольняє умови (1.4-1.6), що дозволяє використовувати розроблену реалізацію мурашиного алгоритму для вирішення ЗК.

2.3 Самоорганізуюча карта Кохонена

Широкого розповсюдження при вирішенні задач, які оперують складними структурами даних, набули такі методи машинного навчання, як штучні нейронні мережі. ШНМ можуть бути застосовані для вирішення широкого спектру задач, таких, як розпізнавання образів, кластеризація, прогнозування явищ та вирішення оптимізаційних задач, зокрема пошуку найкоротших маршрутів [36]. В залежності від поставленого завдання,

мережі можуть відрізнятися архітектурою (розташування нейронів, розподілення та напрям зв'язків між ними), підходом до навчання тощо.

Для вирішення ЗК найбільш доцільним є використання самоорганізуючої карти Кохонена (Self-Organizing Map – SOM), що використовує підхід «навчання без вчителя» – налаштування мережі без використання тренувальних даних, направлене на виявлення неочевидних взаємозв'язків та закономірностей даних. Особливістю самоорганізуючих карт є організація мережі у вигляді двовимірної сітки, де значення ваг кожного нейрону позначають його координати на заданому просторі характеристик вхідних даних [23, 37].

На кожній ітерації навчання мережі вхідний сигнал послідовно надходить від кожного елементу даних на всі нейрони одночасно, а вихідний сигнал породжує лише один нейрон, який носить назву BMU (Best Matching Unit) і є найближчим до обраної вершини за значенням вагів. В кінці ітерації BMU коригує свої ваги в бік наближення до вершини, яка розглядалася.

Початковим призначенням мереж Кохонена є задачі кластеризації та спрощення візуалізації даних, проте є можливим застосування ряду заходів, які дозволять адаптувати ШНМ до вирішення ЗК. Однією з таких дій є вибір архітектури у вигляді замкнутого кільця [22], що дасть змогу інтерпретувати навчену мережу як готовий маршрут обходу графу і задовольнити умови (1.4-1.6). Початкове розміщення мережі обрано в вигляді кола, яке розміщується в центрі графа ЗК, чий радіус задається параметром R.

Друга умова, яка визначає завершеність процесу навчання мережі, є введенням поняття асоційованих нейронів. Асоційованим вирішено називати такий нейрон, для якого відстань до вершини даних досягла заданого порогового значення ε. Коли це відбувається, така вершина ЗК виключається з перебору в наступних ітераціях, а асоційований нейрон більше не потрапляє в перелік кандидатів для вибору BMU. При цьому, кожен нейрон може бути асоційований не більш ніж з одною вершиною ЗК. Введене поняття також визначає головну умову зупинки навчання мережі – для

кожної вершини ЗК має існувати асоційований нейрон. Важливе значення в процесі навчання мережі має функція сусідства $h(n, n_{BMU})$, яка визначає порядок зсуву нейронів мережі залежно від їх розташування відносно BMU. Якщо певним чином виконувати зсув не тільки BMU, але й деякої кількості близьких до нього нейронів, це дозволить більш рівномірно розподілити вузли мережі між вершинами ЗК і уникнути появи великих відгалужень. В якості такої функції вирішено обрати функцію Гауса наступного вигляду:

$$h_{i,BMU} = e^{\frac{-D(n_i, n_{BMU})^2 * L(n_i, n_{BMU})}{\sigma}}, \quad (2.8)$$

де $L(n_i, n_{BMU})$ – евклідова відстань від нейрону n_i до BMU, $D(n_i, n_{BMU})$ – відстань від нейрону n_i до BMU, що виражається в кількості переходів між ними по мережі, σ – параметр «еластичності» мережі.

Використання функції D дозволяє враховувати віддаленність нейронів згідно топології мережі та віддавати більшу перевагу тим вузлам, які знаходяться ближче до BMU по порядку слідування в мережі, а не лише за лінійною відстанню. Параметр σ визначає «еластичність» мережі – його значення прямо пропорційно впливає на кількість сусідніх з BMU нейронів, які змінять своє положення, і на відстань їх зсуву. В ході побудови карти σ зменшується із темпами, які задаються окремим параметром $\Delta\sigma$, поступово зменшуючи значення функції сусідства, і роблячи кожну наступну зміну в мережі більш локальною. Після обчислення функції Гауса, відбувається зсув нейронів в бік поточної вершини ітерації, що описується як

$$n_i' = n_i + \mu h_{i,BMU}(d - n_i), \quad (2.9)$$

де n_i – нейрон, що переміщується, d – вершина ЗК, в бік якої відбувається зміщення нейрону, $\mu \in (0;1]$ – коефіцієнт навчання, що вповільнює збіжність алгоритму, забезпечуючи точніші рішення.

На останніх ітераціях роботи ШНМ, коли локальність змін є максимальною (відбувається пересування переважно тільки BMU), може виникнути явище, яке має негативний вплив на остаточний результат. Якщо один і той же нейрон обирається як BMU для двох чи більше вершин ЗК, його наближення до однієї вершини частково компенсується переміщенням в бік іншої, що призводить до уповільнення процесу асоціації. Для запобігання цьому явищу, вибір найближчого до вершини нейрона визначено як

$$n_j : L(n_j, d_i) * p_j \to \min, \tag{2.10}$$

де p_j є штрафним коефіцієнтом, що має на меті завадити занадто частому вибору нейрону як найближчого та прискорити пересування в бік однієї конкретної вершини. Після кожного вибору елементу як BMU його штрафний коефіцієнт збільшується на значення, що задається окремим параметром Δp:

$$p_j^{'} = p_j + \Delta p. \tag{2.11}$$

Початкове значення коефіцієнту для всіх нейронів дорівнює 1, що відповідає повній відсутності штрафів, так як в такому випадку оцінка нейрона при виборі BMU буде точно відповідати відстані до вершини, що розглядається.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ЗАСТОСУНКУ

3.1 Загальний огляд структури програмного рішення

Програмна реалізація досліджуваних методів вирішення ЗК представляє собою сукупність окремих проектів, котрі відповідають конкретним алгоритмам або наборам програмних інструментів, що можуть застосовуватися одним або декількома алгоритмами та слугувати для подальшого розвитку та розширення застосунку.

Загальна структура програмного рішення включає 16 проектів, які можна розділити на 5 груп згідно алгоритму чи моделі, яку вони представляють (рисунок 3.1). Серед цих груп:

- TSP – засоби для програмного моделювання задачі комівояжера, що включають структури даних для зберігання графу ЗК, підготовлені приклади з TSPLIB та функції для їх завантаження готових графів або генерації випадкових ЗК, розрахунок довжини знайденого маршруту для обраного графу тощо;
- GA – частина рішення, що стосується генетичного алгоритму, операторів кросоверів, мутації, селекції;
- Ant Colony – набір проектів, що містить розроблені варіанти мурашиних алгоритмів;
- SOM – розділ, що включає реалізацію самоорганізуючої карти Кохонена;
- Algorithms – умовний розділ, що містить допоміжні засоби і алгоритми, які застосовуються в процесі роботи методів рішення ЗК, та поділяється на два проекти: Algorithms.Utility – спільні елементи для роботи ГА, МА та автоматизації експериментів, та Algorithms.SphereTools – реалізація розподілення точок по колу для початкового розташування мережі SOM.

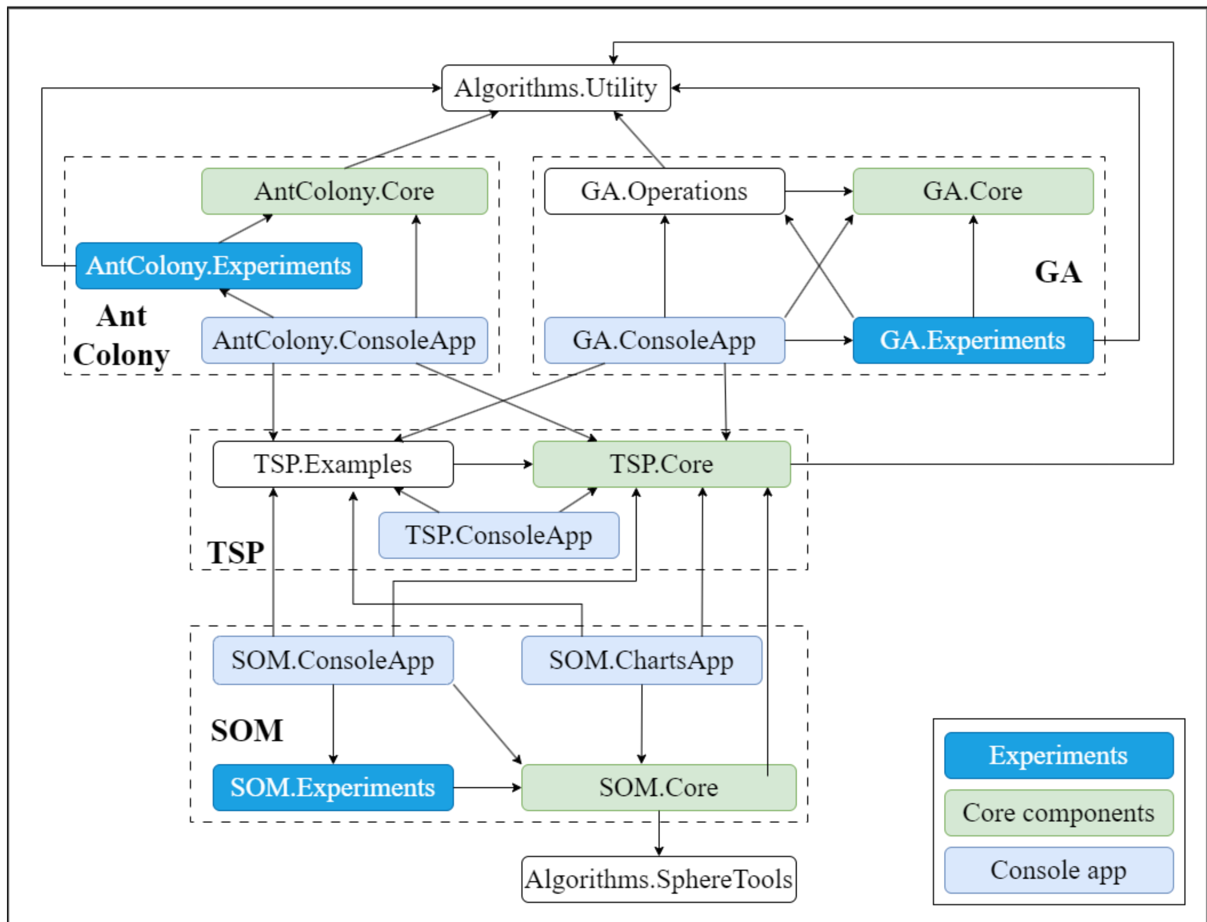

Рисунок 3.1 – Структура програмного рішення

Кожен компонент групи може мати власне призначення, серед яких: core components – набір типів, які включають необхідний мінімум засобів для побудови алгоритмів та/або набір готових реалізацій, і можуть бути наслідувані та модернізовані в користувацьких бібліотеках з метою побудови власних реалізацій алгоритмів; experiments – засоби автоматизації проведення експериментів для конкретних алгоритмів; console app – консольний застосунок для налагодження алгоритмів в процесі розробки. Виключеннями з даної класифікації є проекти GA.Operations, що є окремим винесеним розділом з реалізаціями генетичних операторів, та TSP.Examples, що містить приклади з бібліотеки TSPLIB та функціональність їх завантаження до необхідних структур даних. Також окреме значення має проект SOM.ChartsApp, який слугує для налагодження самоорганізуючих

карт із можливістю поетапного графічного відображення процесу навчання штучної нейронної мережі.

3.2 Програмне представлення сутностей задачі комівояжера

Кореневим компонентом реалізованого застосунку є проект TSP.Core, що включає структури даних та логіку, необхідні для програмного представлення ЗК, яке зможуть використовувати розроблені алгоритми з інших частин рішення. Серед типів, що входять до проекту:

- TSPNode – клас-шаблон вершини ЗК;
- TSPModel – модель задачі;
- TSPModelGenerator – засоби для генерації випадкових графів ГА.

Базовим елементом моделі ЗК є вершина графу, що в застосунку представлена типом TSPNode (лістинг 3.1), та містить мінімальні необхідні дані – двовимірні координати вершини та її назву, а також методи доступу до даних, що визначаються інтерфейсом IVector.

Лістинг 3.1 – Програмна реалізація вершини ЗК

```
public class TSPNode : IVector<double> {
     public string Name { get; set; }
     public double X { get; set; }
     public double Y { get; set; }
     double IVector<double>.this[string axis] {
        get { switch (axis) {
                case "X": case "x": return X;
                case "Y": case "y": return Y;
                default: return 0D; } }
        set {switch (axis) {
                case "X": case "x": X = value; break;
                case "Y": case "y": Y = value; break; } } }
    double IVector<double>.this[int index] {
        get { return index switch { 0 => X, 1 => Y }; }
        set {switch (index) {
                case 0: X = value; break;
                case 1: Y = value; break;};} }
    int IVector<double>.Count => 2;}
```

Наступним елементом, який втілює власне модель ЗК, є клас TSPModel. Даний тип приймає як вхідні дані набір вершин ЗК, отриманих в ході випадкової генерації або завантаження з файлів TSPLIB. Окрім зберігання списку вершин в колекції nodes, в процес ініціалізації моделі також включене обчислення довжин переходів між всіма парами вершин у графі (лістинг 3.2). Вирахувані відстані зберігаються у дворівневій колекції пар «ключ-значення» nodesDistancesMap, що дозволяє отримувати довжину необхідного переходу з послідовним вказанням двох вершин. Такий підхід дозволяє пришвидшити виконання алгоритмів, послідовність дій яких потребує періодичного розрахунку довжин маршрутів, знайдених в ході роботи, представляючи їх як суму довжин переходів, отриманих із константною алгоритмічною складністю. Отримання цієї суми здійснюється функцією GetDistance, яка додатково виконує попередню перевірку на належність переданого маршруту до завантаженої моделі ЗК, головною умовою якою є рівність множин точок маршруту та точок TSPModel.

Лістинг 3.2 – Отримання довжини рішення ЗК

```
public class TSPModel {
     private readonly IEnumerable<TSPNode> nodes;
     private Dictionary<TSPNode, Dictionary<TSPNode, double>>
nodesDistancesMap;
     public TSPModel(IList<TSPNode> nodes) { ...
          for (var i = 0; i < nodes.Count; i++) {
               nodesDistancesMap.Add(nodes[i], new
Dictionary<TSPNode, double>());
               for (var j = 0; j < nodes.Count; j++) {
                    if (i == j) continue;
                    nodesDistancesMap[nodes[i]].Add(nodes[j],
GetSectionDistance(nodes[i], nodes[j])); } } }
     public double GetDistance(IList<TSPNode> route) {
          var distance = 0D;
          if (nodes.Except(route).Any()) throw new
ArgumentException(...);
          for (var i = 0; i < route.Count - 1; i++)
          distance += nodesDistancesMap[route[i]][route[i + 1]];
          distance +=
nodesDistancesMap[route.Last()][route.First()];
          return distance; }...}
```

3.3 Реалізація генетичного алгоритму

3.3.1 Особливості реалізації генетичних операторів

Програмна реалізація ГА, як і всі інші розглянуті алгоритми, використовує підходи ООП, що дає змогу використовувати розроблені базові типи для подальшого створення нових варіантів алгоритмів та гнучкого налаштування їх поведінки. Так, всі реалізації генетичних операторів наслідують базовий клас BaseGAOperation, який передбачає задання налаштувань, які зберігаються в полі з користувацьким типом GAOperationSettings (лістинг 3.3).

Лістинг 3.3 – Базовий тип операції ГА

```
public abstract class BaseGAOperation {
     protected GAOperationSettings operationSettings;
     protected BaseGAOperation(GAOperationSettings
operationSettings){
          this.operationSettings = operationSettings ?? throw
new ArgumentNullException(nameof(operationSettings));
          if (operationSettings.InitType ==
GAOperationInitType.OneTime)InitSettings();}
     protected abstract void InitSettings();}
public class GAOperationSettings {
     public GAOperationInitType InitType { get; set; } =
GAOperationInitType.Manual;
     public int NodesCount { get; set; }}
```

Використання базового класу дозволяє визначати спільні налаштування для будь-яких операцій ГА, одним з яких є поле InitType, що може приймати одне зі значень Manual, OneTime, EveryGeneration, EveryIndividual, які відповідають розглянутим в 2.1.4 методам ініціалізації параметрів.

Для кожного з типів генетичних операторів передбачений окремий інтерфейс, що дає змогу уніфікувати сигнатури їх функцій і тим самим підхід до створення нових операторів. Так для кросоверів передбачений інтерфейс ICrossover, основною функцією якого є GetNextGeneration, що приймає

множину пар особин, застосовує необхідний алгоритм схрещування та повертає список нових рішень. Оператори мутацій базуються на інтерфейсі IMutation, головною функцією якого є ProcessMutation, аргументами якої є список особин поточного покоління, та ймовірність мутації. Інтерфейс ISelection представляє основу для реалізації першого етапу модифікованого оператору селекції, в якому метод GetParentPairs слугує для утворення пар особин і в якості вхідних даних приймає набір пар «ключ-значення» з інформацією про особини та їх значень функцій пристосованості.

Розглянуті інтерфейси реалізуються окремими абстрактними класами для кожного типу операторів: BaseCrossover, BaseMutation, BaseSelection, які також наслідують клас BaseGAOperation, таким чином надаючи розробнику гнучкий підхід, що об’єднує в собі можливості параметризації із модульністю ГА, що виражається в простій заміні в алгоритмі одних реалізацій операторів на інші. На прикладі реалізації кросоверу Inver-over (лістинг 3.4) можна зазначити, що тип є нащадком BaseCrossover, і має власну реалізацію функції GetNextGeneration, що надається інтерфейсом та виконує специфічну для Inver-over послідовність дій, та функції InitSettings, що походить з базового класу операції, та визначає порядок задання унікальних для цього алгоритму кроссоверу параметрів. В залежності від використовуваного методу ініціалізації параметрів, місце виклику InitSettings відрізнятиметься. Так, для одноразового задання (OneTime), відповідна умова для значення параметра InitType перевірятиметься в конструкторі BaseGAOperation, що автоматично впроваджує це налаштування для всіх операторів. Якщо параметри мають отримувати своє значення на кожній ітерації, аналогічна умова для значення GAOperationInitType.EveryGeneration розміщується в головній функції оператора (GetNextGeneration у випадку кросовера). Виконання головної функції передбачає обхід переданих особин або пар особин у циклічному операторі, в тілі якого можна розмістити виклик методу InitSettings на випадок встановлення параметрів при обробці кожної особини (every individual).

Лістинг 3.4 – Реалізація кросоверу Inver-over

```
public class InverOverCrossover : BaseCrossover
{
     public int sPosition { get; set; }
     public int kPosition { get; set; }
     public InverOverCrossover(GAOperationSettings
operationSettings) : base(operationSettings) {}
     public override IList<Individual<TGene>>
GetNextGeneration<TGene>(IList<(Individual<TGene>,
Individual<TGene>)> parents)
     {
     IList<Individual<TGene>> children=new
List<Individual<TGene>>(parents.Count * 2);
          if (operationSettings.InitType ==
GAOperationInitType.EveryGeneration) InitSettings();
          foreach (var pair in parents) {
          var firstChildGenome = new List<TGene>(pair.Item1);
          var secondChildGenome = new List<TGene>(pair.Item2);
               if (operationSettings.InitType ==
GAOperationInitType.EveryIndividual)
               InitSettings(); ... }
          return children;
     }
     protected override void InitSettings() {
          sPosition = Random.Shared.Next(0,
operationSettings.NodesCount - 1);
          kPosition = Random.Shared.Next(0,
operationSettings.NodesCount - 1); }
}
```

3.3.2 Опис загальної структури алгоритму

Основним класом, через який генетичний алгоритм запускається на виконання, є GeneticAlgorithm (лістинг 3.5), який для початку роботи потребує передачі ряду параметрів для налаштування необхідної конфігурації. Серед параметрів слід зазначити оператори селекції, кросоверу, мутації, список особин, що виступають як початкова популяція, об’єкт класу GASettings (далі – «налаштування ГА»), який представляє набір додаткових налаштувань алгоритму, та посилання на функцію розрахунку пристосованості.

Лістинг 3.5 – Опис класу GeneticAlgorithm

```
public class GeneticAlgorithm<TGene>{...
     public GeneticAlgorithm(ISelection selection, ICrossover
crossover, IMutation mutation, IList<Individual<TGene>>
population, GASettings settings, Func<Individual<TGene>, double>
fitnessGetter) {...}
     public virtual (Individual<TGene> Individual, double
Fitness) Run() {...}
     public virtual IList<Individual<TGene>> GetNextGeneration()
{...}}
```

Для керування ходом виконання алгоритму передбачено дві функції: GetNextGeneration та Run. Призначенням GetNextGeneration (лістинг 3.6) є моделювання життя одного покоління, що включає виконання головних функцій трьох генетичних операторів та оновлення змінної stopCondition, яка приймає значення true в одному з двох випадків: досягнута гранична кількість поколінь, що виконувалися підряд без знаходження кращого рішення (задається змінною StagnatingGenerationsLimit налаштувань ГА) або перевищення показником виродження популяції (що описано у відповідному підпункті 2.1.5) встановленого порогового значення (змінна DegenerationMaxPercent налаштувань ГА). Також у даній функції передбачена можливість задавати порядок формування нового покоління за одним з двох правил: покоління включає тільки нові особини, отримані в результаті схрещування, або відбір найкращих особин з множини батьківських та дочірніх особин за їх пристосованістю, що відповідає другому етапу модифікованої селекції і регулюється змінною OnlyChildrenInNewGeneration налаштувань ГА.

Лістинг 3.6 – Моделювання покоління ГА GetNextGeneration

```
public virtual IList<Individual<TGene>> GetNextGeneration() {...
     var parentPairs = selection.GetParentPairs(fitnesses,
sort);
     var children = crossover.GetNextGeneration(parentPairs);
     mutation.ProcessMutation<Individual<TGene>,
```

```
TGene>(children, settings.MutationProbability);
     if (settings.OnlyChildrenInNewGeneration) {...
/*leaving only children*/
     } else {...
/*forming a new generation from the most adapted individuals */}
     var currentIterationBestResult = fitnesses.OrderBy(x =>
x.Value).FirstOrDefault(); var resetStagnation = false;
     if (currentIterationBestResult.Value <
currentBestResult.Fitness || currentBestResult.Fitness == 0D)) {
          currentBestResult = (currentIterationBestResult.Key,
currentIterationBestResult.Value);
          resetStagnation = true;}
     if (settings.StagnatingGenerationsLimit.HasValue &&
settings.StagnatingGenerationsLimit.Value > 0) {
/*checking count of generations without improving the result*/
     }
     if (settings.DegenerationMaxPercent.HasValue &&
settings.DegenerationMaxPercent.Value > 0D){
          /*checking the degeneration coefficient*/}
     return population;}
```

Функція Run реалізує повний процес виконання ГА, що включає циклічне виконання методу GetNextGeneration, та перевірку значення змінної stopCondition, з метою припинення виконання алгоритму при досягнені однієї з двох умов зупину. Якщо ні одна з умов не виконується, цикл завершується після проходження числа ітерацій, заданого у змінній GenerationsMaxCount налаштувань ГА.

Незважаючи на те, що функція GetNextGeneration відповідає лише одному кроку виконання алгоритму, як і Run, вона має публічний модифікатор доступу, що дає змогу розробнику оброблювати окремі покоління з метою прокрокового аналізу роботи ГА або графічного відображення стану популяції.

3.4 Реалізація мурашиного алгоритму

3.4.1 Опис загальної структури програмного коду

В основі реалізації МА лежить абстрактний клас BaseAlgorithm, який надає певні готові кроки, спільні для всіх наявних варіантів алгоритмів, а

також деякі абстрактні методи, реалізація яких повинна бути надана у класах-нащадках (лістинг 3.7). Так, для різних версій МА відрізняється порядок випаровування феромону (EvaporatePheromones), принцип побудови маршруту агентом (TravelPath) і код функції для запуску алгоритму Run.

Лістинг 3.7 – Опис класу BaseAlgorithm

```
public abstract class BaseAlgorithm<TNode> where TNode : class
{...
     protected BaseAlgorithm(IList<TNode> nodes, Func<TNode,
TNode, double> edgeDistanceGetter, AntColonySettings settings)
{...
/*data initialization, including the pheromone amount */
}
     public abstract IList<IList<TNode>>
Run(AntPopulationSettings antSettings);
     private protected abstract void TravelPath(Ant<TNode> ant);
     protected virtual double GetTravelProbability(TNode
currentNode, TNode unvisitedNode) {
     return Math.Pow(pheromoneMap[currentNode][unvisitedNode],
settings.PheromoneWeight) + Math.Pow(1 /
edgeDistanceGetter(currentNode, unvisitedNode),
settings.DistanceWeight);}
     protected abstract void EvaporatePheromones();
     protected virtual List<Ant<TNode>> TravelAllPaths(int
antCount) {...}}
```

Спільну базову реалізацію для всіх варіантів має логіка ініціалізації рівнів феромону на переходах між містами, дані про які зберігаються у дворівневій колекції пар «ключ-значення», схожим чином до відстаней переходів на графі ЗК, функція часткового розрахунку ймовірності переходу агенту між двома вершинами GetTravelProbability, та метод запуску обходу шляхів всіма агентами TravelAllPaths.

Метод TravelAllPaths також включає попередню ініціалізацію агентів в екземплярах класу Ant, основне призначення яких – зберігати пройдені у процесі пошуку шляху вершини графу у пам’яті, яка представлена списком TravelledPathMemory (лістинг 3.8).

Лістинг 3.8 – Опис класу BaseAlgorithm

```
public class Ant<TNode>{
     public IList<TNode> TravelledPathMemory;
     public double PersonalPheromoneAmount;
     public Ant() { TravelledPathMemory = new List<TNode>(); }}
```

Як і абстрактні методи, базова функціональність також наділена відповідними модифікаторами, які дають змогу застосовувати принцип поліморфізму ООП і за необхідності розширяти чи перевизначати порядок дій у таких методах. Початковими даними для ініціалізації класу алгоритму мають бути список міст, функція отримання вартості переходу між двома вершинами, та об’єкт класу AntColonySettings (далі – «налаштування колонії»).

До налаштувань колонії входять такі параметри, як: вагові коефіцієнти кількості феромону α та довжини переходу β (PheromoneWeight, DistanceWeight), коефіцієнт випаровування ρ (EvaporationCoefficient), початкова кількість феромону на переходах (InitialPheromoneAmount), кількість феромону, який залишають на маршруті звичайні та елітарні агенти (CommonAntPheromoneAmount, CommonEliteAntPheromoneAmount) та інші.

Окрім налаштувань колонії, при запуску алгоритму за допомогою функції Run також необхідно передати екземпляр класу AntPopulationSettings – налаштування агентів, до яких відносяться кількість задіяних у МА звичайних та елітних мурах (AntCount, EliteAntCount).

Також передбачена можливість задавати кількість залишеного на шляху феромону індивідуально для кожного агенту за допомогою масивів AntPersonalPheromoneAmounts та EliteAntPersonalPheromoneAmounts. Ця функціональність діє за умови значення false для параметрів налаштувань мурашиної колонії UseCommonAntPheromoneAmount та UseCommonEliteAntPheromoneAmount, які визначають використання загальної чи індивідуальної кількості.

3.4.2 Реалізація обходу графу агентами

У реалізації класичного підходу Ant System тіло функції Run включає 3 основні дії, що описують ітерацію алгоритму: обхід шляхів (TravelAllPaths), випаровування частини феромонів (EvaporatePheromones) та оновлення феромону на переходах згідно до прокладених маршрутів (2.7), після чого повертається список шляхів, знайдених на поточній ітерації (лістинг 3.9). Оскільки кількість феромону на переходах графа зберігається у внутрішньому стані класу, виконання МА можливо продовжити, поступово покращуючи знайдені маршрути з кожною ітерацією.

Лістинг 3.9 – Реалізація функції Run для алгоритму Ant System

```
public override IList<IList<TNode>> Run(AntPopulationSettings antSettings){
     var ants = TravelAllPaths(antSettings.AntCount);
     EvaporatePheromones();
     foreach (var ant in ants)
     for (var i = 0; i < ant.TravelledPathMemory.Count - 1; i++)
     {var currentPheromoneAmount = pheromoneMap
[ant.TravelledPathMemory[i]][ant.TravelledPathMemory[i + 1]];
          var pheromoneDelta =
settings.UseCommonAntPheromoneAmount ?
settings.CommonAntPheromoneAmount : ant.PersonalPheromoneAmount;
               pheromoneDelta /=
edgeDistanceGetter(ant.TravelledPathMemory[i],
ant.TravelledPathMemory[i + 1]);
     pheromoneMap[ant.TravelledPathMemory[i]][ant.TravelledPathM
emory[i + 1]] = currentPheromoneAmount + pheromoneDelta;}
     return ants.Select(x => x.TravelledPathMemory).ToList();}
```

Метод TravelPath, який відповідає за побудову маршруту для одного агенту, обирає по черзі невідвідані вершини з загального переліку nodes, допоки такі присутні, і зберігає їх до пам'яті агенту TravelledPathMemory, що дозволяє задовольнити вимогам до маршруту ЗК (лістинг 3.10). Всім можливим переходам із поточної вершини, на якій знаходиться агент, надається оцінка, яка залежить від вартості переходу, кількості феромону на

переході, та значення вагових коефіцієнтів α і β, і розраховується в методі GetTravelProbability. Після чого, формується ряд чисел probabilityIntervals, де першим числом є оцінка переходу до першого вузла, а кожне наступне є сумою попереднього числа та оцінки вірогідності переходу до вершини, що розглядається. Таким чином останнє число N цього ряду завжди буде сумою всіх оцінок переходів, і пари сусідніх чисел утворюватимуть діапазони, довжина яких пропорційна до необхідної ймовірності вибору наступного переходу. Далі генерується випадкове число randomValue в діапазоні від 0 до N, і визначається номер діапазону intervalIndex, якому це число належить. Оскільки перелік інтервалів формувався в порядку слідування вершин списку nodes, за значенням intervalIndex стає можливим визначити номер вершини, яку потім додається в пам’ять агенту та видаляється із nodes. За номером діапазону отримується номер вершини, який додається в пам’ять агенту та видаляється із загального переліку.

Лістинг 3.10 – Реалізація функції TravelPath

```
private protected override void TravelPath(Ant<TNode> ant) {
     var nodes = this.nodes.ToList();
     ant.TravelledPathMemory.Add(nodes[0]);nodes.RemoveAt(0);
     while (nodes.Any()){
          var probabilityIntervals = new List<double>();
          for (var i = 0; i < nodes.Count; i++) {
               var travelProbabilityPart =
GetTravelProbability(ant.TravelledPathMemory.Last(), nodes[i]);
if (i == 0) probabilityIntervals.Add(travelProbabilityPart);
else probabilityIntervals.Add(probabilityIntervals.Last() +
travelProbabilityPart);}
          var randomValue =
Random.Shared.NextDouble() * probabilityIntervals.Last();
          probabilityIntervals.Add(randomValue);
          probabilityIntervals.Sort();
          var randomValueIndex =
SearchHelper.BinarySearch(probabilityIntervals, randomValue);
if (randomValueIndex != -1){ var intervalIndex =
(randomValueIndex==probabilityIntervals.Count - 1) ?
randomValueIndex - 1 : randomValueIndex;
probabilityIntervals.RemoveAt(randomValueIndex);
     ant.TravelledPathMemory.Add(nodes[intervalIndex]);
     nodes.RemoveAt(intervalIndex);}}}
```

3.5 Реалізація самоорганізуючої карти Кохонена

3.5.1 Послідовність дій алгоритму

З урахуванням розглянутих елементів та особливостей роботи алгоритму роботи SOM, загальну послідовність дій для побудови самоорганізуючої карти пошуку рішення ЗК можна описати наступним чином. Виконується ініціалізація ШНМ $N = |n_i|$, де n_i є окремий нейрон. Для кожної вершини d ЗК, що не має асоційованих нейронів, з підмножини неасоційованих нейронів обирається BMU (2.10), розраховується функція сусідства h (2.8), потім для кожного нейрону n_i розраховується нове положення n'_i (2.9), і якщо $L(d, n_{BMU}) < \varepsilon$, то нейрон n_{BMU} стає асоційованим з вершиною d. Після цього оновлюється штрафний коефіцієнт p для нейрона n_{BMU} (2.11). Описана процедура повторюється доки залишаються неасоційовані з мережею вершини ЗК, після обробки яких утворюється маршрут із асоційованих нейронів у порядку їх з’єднання у мережі.

3.5.2 Опис базового типу SOM

Побудова мережі Кохонена в програмному рішенні спирається на абстрактний клас BaseSOM, який надає готовий шаблон для роботи з різними структурами даних та топологіями мереж. BaseSOM надає поля для зберігання інформації про вузли мережі, елементи вхідних даних, стан та налаштування навчання мережі, списки асоційованих нейронів тощо (лістинг 3.11). Вхідні дані та нейрони мережі задаються типами, що реалізують інтерфейс IVector, який надає універсальний метод доступу до координат векторів. Такий підхід робить можливим використання векторів довільної розмірності, в тому числі й вершини ЗК, які представлені класом TSPNode. Подібно до реалізації ГА, архітектура BaseSOM також надає можливість або запустити повне виконання алгоритму функцією BuildMap,

або здійснювати навчання мережі покроково з метою налагодження або графічного відображення: метод ProcessIteration здійснює один обхід неасоційованих векторів вхідних даних, а значення булевого поля FinishCondition встановлюється в true за відсутності неасоційованих вхідних векторів, що може бути використано в умові зупинки алгоритму.

Лістинг 3.11 – Опис класу BaseSOM

```
public abstract class BaseSOM<TVector> where TVector :
IVector<double>{...
     protected IList<TVector> dataVectors;
     protected int networkSize; public SOMSettings settings;
     protected IList<TVector> networkVectors;
     protected Dictionary<TVector, Dictionary<TVector, double>>
networkTopologyDistances;
     protected Dictionary<TVector, double>
networkDistancePenalties;
     protected Dictionary<TVector, bool> networkReadiness;
     protected Dictionary<TVector, bool> dataReadiness;
     public IList<TVector> Network => networkVectors.ToList();
     public bool FinishCondition => !this.dataReadiness.Any(x =>
x.Value == false);
protected abstract void InitNetwork(Topology topology);
     protected virtual void InitSphereNetwork(int networkSize)
{this.networkSize = networkSize;}
     protected abstract double GetNeighbourFunction(TVector
chosenVector, TVector otherVector);
     public abstract void ProcessIteration();
     public abstract IEnumerable<TVector> BuildMap()}
```

3.5.3 Адаптація SOM для вирішення задачі комівояжера

Для пошуку маршрутів ЗК розроблено клас TwoDimensionalSOM, який є нащадком BaseSOM і розрахований на операції з двохкоординатними векторами. В методі InitNetwork, який здійснює ініціалізацію мережі, вирішено розглядати початкове розташування нейронів у вигляді кільця (InitSphereNetwork), але наявність аргументу методу із типом переліку Topology дозволяє за необхідності додавати та використовувати інші топології (лістинг 3.12). Першою дією InitSphereNetwork є розрахунок

початкових координат нейронів, із задіянням проекту Algorithm.SphereTools, функціональність якого дозволяє знайти координати заданого числа точок на колі за відомими координатами центру та радіусом. Після задання початкового положення нейронів, словник networkTopologyDistances заповнюється інформацією про кількість переходів між всіма парами нейронів, що є необхідною складовою при обчисленні функції сусідства. Останньою дією метода є ініціалізація штрафних коефіцієнтів при виборі BMU, якщо встановлений параметр UseDistancePenalties класу SOMSettings (далі – «налаштування SOM»), екземпляр якого передається в конструктор класу TwoDimensionalSOM.

Лістинг 3.12 – Опис процесу ініціалізації мережі

```
protected override void InitNetwork(Topology topology){
     switch (topology){case Topology.Sphere: default:
     InitSphereNetwork(this.dataVectors.Count);break;}}
protected override void InitSphereNetwork(int networkSize){
     .../*calculating the center and radius of the circle*/
     var circlePoints = SphereTools.GetCirclePoints(networkSize,
radius, (centerX, centerY));
     for (int i = 0; i < networkSize; i++){
this.networkVectors.Add(Activator.CreateInstance<TPoint2D>());
     this.networkVectors.Last()["x"] = circlePoints[i].X;
     this.networkVectors.Last()["y"] = circlePoints[i].Y;
     this.networkReadiness.Add(this.networkVectors[i], false);
     this.networkTopologyDistances.Add(this.networkVectors[i],
new Dictionary<TPoint2D, double>());
     for (int j = 0; j < i; j++) {
          .../*filling amount of edges between neurons*/}}
     if (settings.UseDistancePenalties){
this.networkDistancePenalties=new Dictionary<TPoint2D,double>();
          foreach (var vector in this.networkVectors)
               this.networkDistancePenalties.Add(vector,
networkDistancePenaltiesDefaultValue);}...}
```

З метою підвищення швидкодії алгоритму, у програмній реалізації описаного порядку навчання мережі був передбачений ряд додаткових дій. По-перше, при обході неасоційованих вершин, функція сусідства h не обчислюється для BMU, а її значення приймається за одиницю, оскільки h

спирається на відстань від вказаного нейрона до BMU (2.8), яка в випадку самого BMU завжди відома і дорівнює нулю. Також встановлено, що в ході роботи алгоритму, параметр «еластичності» мережі σ (CooperationCoefficient у налаштуваннях SOM) поступово зменшується до низьких значень, при яких розраховане значення функції сусідства більше не має суттєвого впливу на пересування нейронів, але сам розрахунок h продовжує здійснювати обчислювальне навантаження, вповільнюючи процес навчання мережі. За цієї причини, до налаштувань SOM введено параметр CooperationThreshold, який задає порогове значення для CooperationCoefficient, після досягнення якого, функція сусідства далі не розраховуватиметься, а подальші переміщення мережі будуть виключно локальними (лістинг 3.13).

Лістинг 3.13 – Опис умов переміщення нейронів мережі

```
public override void ProcessIteration() {...
     foreach (var dataVector in dataVectors.Where(x =>
this.dataReadiness[x] == false)){.../*retrieving the BMU*/
          if (distance > settings.RoundPrecision) {
               /*if distance between BMU and TSP city is bigger
than RoundPrecision (ε) make it closer to city */
          } else {/*otherwise, make BMU associated to city*/}
          if (params.UseElasticity &&
params.CooperationCoefficient > params.CooperationThreshold){
/*if elasticity feature is used and σ is big enough, calculate
the neighbor function for the rest of neurons*/
var networkVectorsToAdjust = workingNetworkVectors.ToList();
networkVectorsToAdjust.Remove(closestNetworkVector);
          var elasticityCoefs = networkVectorsToAdjust.Select(x
=> GetNeighbourFunction(x, closestNetworkVector)).ToArray();
          for (int i = 0;i < networkVectorsToAdjust.Count;i++){
          if (elasticityCoefs[i] > 0D)
               for (var j = 0;j<closestNetworkVector.Count;j++)
networkVectorsToAdjust[i][j] = networkVectorsToAdjust[i][j] +
settings.LearningCoefficient * (dataVector[j] -
networkVectorsToAdjust[i][j]) * elasticityCoefs[i];}
settings.CooperationCoefficient *= 1D -
settings.CooperationFading / 100D;
     }
.../*updating penalty coefficients*/}
}
```

3.5.4 Вирішення проблеми відгалужень на маршруті

Оскільки розподілення вершин ЗК у графі не є рівномірним, є актуальною проблема появи відгалужень на маршруті, які ускладнюють пошук найкоротшого шляху. Вибір функції сусідства та параметра еластичності мережі допомагає запобігти цьому на початку роботи алгоритму, але на пізніх етапах навчання мережі, в ній можуть залишатися нейрони, чиї сусіди вже асоційовані з вершинами, але які не мають поблизу доступних вершин для асоціації з ними. Алгоритм роботи карти Кохонена буде рухати такі нейрони до найближчих доступних вершин, які можуть знаходитися на віддаленій ділянці простору графу ЗК, що породжує відгалуження на фрагментах маршруту (рисунок 3.2), які роблять кінцевий шлях обходу графа значно довшим.

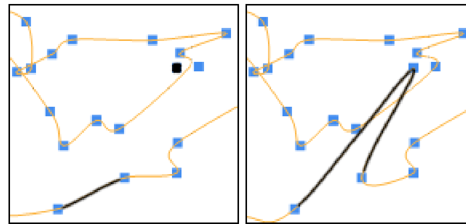

Рисунок 3.2 – Поява відгалужень при нерівномірному розподіленні точок мережі

З метою уникнення появи відгалужень на пізніх етапах навчання, вирішено формувати мережу з кількістю нейронів, що перевищує кількість вершин ЗК. Після виконання умови зупину, при отриманні остаточного результату, нейрони, які не мають асоційованих вершин ЗК, не потраплятимуть до маршруту при його формуванні. Доцільне значення коефіцієнта збільшення кількості нейронів визначається експериментально для конкретного графа і задається параметром NetworkSizeMultiplier налаштувань SOM.

3.6 Структура засобів автоматизації експериментів

3.6.1 Налаштування та запуск досліду

Для кожного виду реалізованого алгоритму, розроблений застосунок має функціональність автоматизації проведення експериментів, яка описана в відповідних проектах: GA.Experiments, MA.Experiments та SOM.Experiments.

Проекти мають подібну структуру, головним компонентом є ExperimentEngine – клас, який відповідає за запуск серії експериментів у окремій функції Run, до якої передаються всі необхідні дані: граф ЗК, конфігурація алгоритму, значення вхідних параметрів, налаштування експерименту, метод збереження результатів тощо (лістинг 3.14). Результати, отримані в ході експерименту, зберігаються в структурі даних типу ExperimentResult, куди потрапляє така інформація, як: опис та довжини знайдених маршрутів, загальний час виконання алгоритму та показники властиві конкретним алгоритмам (середній час ітерації, коефіцієнт виродження тощо).

Лістинг 3.14 – Приклад функції запуску серії експериментів для ГА

```
public static IList<GAExperimentResult<TNode>> Run<TNode, TResearch>(
     IList<TNode> nodes,GASettings settings,
     GAExperimentSettings<TResearch> experimentSettings,...)
     where TResearch : struct, IComparable<TResearch>{
     //initialize writers, get parameter reflection info
     var researchedProperty = settings.GetType()
.GetProperty(experimentSettings.ResearchedParameterName);
     var researchedParam =
experimentSettings.ResearchedParameterRange.Min;...
     while(researchedParam.Value.CompareTo(experimentSettings.Re
searchedParameterRange.Max.Value) <= 0){...
               //set researched parameter
researchedProperty.SetValue(settings, researchedParam.Value);...
               for (int i = 0; i < repeatingCount; i++){
                    .../*setting up the algorithm and run it*/
var timer = Stopwatch.StartNew();
var bestResult = algo.Run(); timer.Stop();
```

```
var experimentResult = new GAExperimentResult<TNode>(){
/*filling experiment results data*/
ResearchedParameterName =
experimentSettings.ResearchedParameterName,
MinResult = resultGetter(bestResult.Individual),
ResearchedParameterValue =
researchedProperty.GetValue(settings),
Time = timer.Elapsed     };
          resultsList.Add(experimentResult);
          /*save results with writers*/...
          /*add summary record if needed*/
          if (repeatingCount > 1){
     var group = resultsList.Where(x=>!x.IsGroupResult &&
x.GroupGuid==groupGuid);
var groupExperimentResult = new GAExperimentResult<TNode>(){
IsGroupResult = true, GroupGuid = groupGuid,... };
          resultsList.Add(groupExperimentResult);
     /*save summary record with writers*/...}
researchedParam.AddStore(experimentSettings.ResearchedParameterS
tep);}... return resultsList;}
```

Налаштування експериментів зберігаються в типі ExperimentSettings, куди окрім специфічних для алгоритмів налаштувань, входять декілька параметрів, які задають мету експерименту та керують його ходом. Перший, ResearchedParameterName, приймає рядок із назвою поля з класу налаштувань алгоритму, після чого, доступ до необхідного поля здійснюється за допомогою функціональності рефлексії у .NET. Обрана таким чином властивість приймається за досліджуваний параметр, наприклад, «MutationProbability» у випадку ГА, якщо потрібно з’ясувати вплив ймовірності мутації на результат роботи алгоритму.

ResearchedParameterRange – вектор з двома координатами, що позначають мінімальне та максимальне значення досліджуваного параметра, якими можуть бути цілі чи дійсні числа. Також допустимими граничними значеннями є змінні з переліку (enum), що виявляється корисним при дослідженні різних компонентів алгоритму, наприклад, генетичних операторів. Значення ResearchedParameterStep задає крок, з яким досліджуваний параметр буде змінюватися для кожного нового експерименту в одній серії. Також існує можливість задавати кількість експериментів, які мають виконуватися за однакових умов без змін значення

досліджуваного параметру, що задається змінною ControlRepeatingCount. Якщо згідно даного налаштування виконується більше одного експерименту, до результатів також додається запис зі зведеними даними (середній час виконання, найкращий отриманий результат тощо). Така можливість може знадобитися для алгоритмів, чутливих до генерації випадкових значень.

3.6.2 Збереження результатів експерименту

Важливим засобом автоматизації проведення серій експериментів є збереження отриманих результатів експериментів у зручній формі для детального ознайомлення та подальшої обробки. З цією метою розроблено перелік типів, які реалізують інтерфейс IExperimentResultWriter із функцією Write (лістинг 3.15). Функція приймає як вхідний параметр екземпляр класу ExperimentResult і виконує його збереження згідно наданої у класі-нащадку реалізації. Класи, які реалізують інтерфейс IExperimentResultWriter:

- ConsoleWriter – виведення інформації в програмну консоль, для спостереження за ходом досліду в процесі його виконання;
- FileWriter – абстрактний клас, що містить готову основу для запису даних у файл на диску;
- JsonWriter – нащадок FileWriter, який дозволяє зберігає результати дослідів в зручному для програмної обробки форматі JSON;
- CSVWriter – нащадок FileWriter, який надає можливість збереження в табличному форматі CSV, зручному для аналізу та виявлення закономірностей, зокрема, за рахунок можливості побудови додаткових зведених таблиць, графіків, діаграм тощо.

Лістинг 3.15 – Інтерфейс IExperimentResultWriter

```
public interface IExperimentResultWriter<TResearch> where
TResearch : struct, IComparable<TResearch> {
     void Write<TNode>(ExperimentResult<TNode> result);}
```

Використання спільного інтерфейсу робить можливим використання декількох засобів збереження одночасно. В методі Run, це можливо здійснити шляхом передачі до вхідного аргументу writerTypes декількох значень переліку WritersEnum, назви яких відповідають класам сутностей Writer. Перед ініціалізацією алгоритму пошуку рішення ЗК та встановлення початкового значення досліджуваного параметру викликається метод CreateWriters, який створює реалізації IExperimentResultWriter згідно до значення writerTypes (лістинг 3.16).

Лістинг 3.16 – Реалізація збереження результатів експерименту

```
public static IList<GAExperimentResult<TNode>> Run<TNode,
TResearch>(...
WritersEnum writerTypes = WritersEnum.None, string path = "")
     where TResearch : struct, IComparable<TResearch>{
     IList<IExperimentResultWriter<TResearch>> writers =
CreateWriters(writerTypes, path, settings, experimentSettings);
while (/*researched parameter boundaries check condition*/){...
     for (int i = 0; i < repeatingCount; i++){
//run algorithm, form the record
var experimentResult = new GAExperimentResult<TNode>() {...}
if (writers!=null && writers.Any())
//save the record with writers
foreach (var writer in writers)writer.Write(experimentResult);}
               if (repeatingCount>1){ //same for summary records
var groupExperimentResult=new GAExperimentResult<TNode>(){...};
                    resultsList.Add(groupExperimentResult);
               if (writers != null && writers.Any())
                    foreach (var writer in writers)
                    writer.Write(groupExperimentResult);}...}
if (writers != null && writers.Any()) //deinitializing writers
     foreach (var writer in writers)
          if (writer is IDisposable disposableWriter)
               disposableWriter.Dispose();...}
```

Функція Write для кожної з наявних реалізацій класів збереження викликається одразу після формування кожного нового запису серії дослідів або запису зі зведеними даними, що дозволяє уникнути втрати частково отриманих результатів у випадку виникнення програмних виключень, технічних несправностей чи інших непередбачуваних обставин.

4 РЕЗУЛЬТАТИ ЕКСПЕРИМЕНТАЛЬНИХ ДОСЛІДЖЕНЬ

4.1 Налаштування параметрів для проведення досліджень

З метою аналізу роботи методів вирішення ЗК, вирішено проводити досліди на графах з бібліотеки TSPLIB, оскільки в них надані вже відомі довжини найкращих можливих маршрутів обходу графу. Враховуючи можливу залежність швидкості та точності виконання алгоритму від розмірності графа, обрано задачі з різною кількістю вершин: tsp225, pcb442 та pr1002, кількість вершин у яких вказана у їх назвах.

Процес проведення експерименту включає вивчення впливу конкретного вхідного параметру чи замінного компоненту алгоритму, що здійснюється шляхом виконання ряду дослідів, умови яких мають відрізнятися тільки значенням досліджуваної змінної. Мінімальна, максимальна межі, а також крок, з яким змінюється значення параметра, задаються безпосередньо перед початком експерименту.

4.2 Дослідження збіжності генетичних алгоритмів

Дослідження використання ГА для рішення ЗК починаються з перевірки темпів збіжності алгоритму для різних комбінацій генетичних операторів та розглянутих функцій оцінки виродження популяції – лінійної та квадратичної (2.4-2.5). В процесі дослідження також має бути врахований вплив обраного підходу до ініціалізації параметрів, серед яких вирішено розглядати такі, що включають генерацію випадкових значень: «one time» (OT), «every generation» (EG), «every individual» (EI). Оскільки основним призначенням оператора мутації є підвищення різноманіття генотипу, що призводить до уповільнення темпів збіжності, вирішено не використовувати його в даній серії експериментів. Умови досліду обрані наступні:

- ймовірність мутації – 0%;
- розмір популяції – 250 особин;
- основна умова зупину – досягнення коефіцієнтом виродження *D* значення 0,8 або вище;
- другорядна умова зупину – досягнення максимально можливої кількості поколінь, яка складає 5000.

Серед змінних величин: 2 варіанти розрахунку показника виродження популяції, 6 можливих реалізацій кросоверу, 2 види оператора селекції та 3 метода ініціалізації параметрів, що за умови зміни значення тільки одного параметра зумовлює проведення 72 дослідів для одного графу ЗК.

В ході серії експериментів, проведених для задачі pr1002 (таблиця 4.1), виявлено, що найменшу схильність до передчасної збіжності забезпечує кросовер IOX із методом ініціалізації початкових параметрів «every individual», який має близькі показники виродження для обох досліджуваних методів відбору – турнірної селекції (Т) та рулеткою (Р) і для обох варіантів розрахунку показника виродження *D*, які знаходяться в межах 0,012-0,034.

Таблиця 4.1 – Збіжність ГА на графі pr1002 для лінійної функції D

№ п/п	Кросовер	Селекція	Ініціалізація	Кільк. ітерацій	Виродження
1	IO	T	EI	5000	0,0120
2	IO	P	EI	5000	0,0201
3	IO	T	EG	5000	0,0375
4	IO	P	EG	5000	0,0924
5	OBX	T	EI	2536	0,8393
6	OBX	T	EG	2306	0,8755
7	IO	T	OT	434	0,8568

У ході обробки результатів експериментів, були використані як лінійна функція розрахунку виродження (таблиця 4.1), так і квадратична (таблиця 4.2).

Таблиця 4.2 – Збіжність ГА на графі pr1002 для квадратичної функції D

№ п/п	Кросовер	Селекція	Ініціалізація	Кільк. ітерацій	Виродження
1	IO	P	EI	5000	0,0186
2	IO	T	EI	5000	0,0343
3	IO	P	EG	5000	0,0396
4	IO	T	EG	5000	0,1505
5	OBX	P	EI	2548	0,9050
6	OBX	T	EG	2181	0,9048
7	IO	P	OT	597	0,8318

4.3 Дослідження використання генетичних алгоритмів для вирішення задачі комівояжера

На наступному етапі досліджується швидкість роботи різновидів генетичного алгоритму, знайдені ними довжини маршрутів. З цією метою, проводиться серія дослідів, аналогічна до попередньої за винятком умови зупину – досягнення 500 поколінь, що забезпечує виконання однакової кількості структурних етапів алгоритму всіма комбінаціями операторів.

У результаті експериментів з селекцією методом рулетки (рисунок 4.1) встановлено, що шлях з найменшою довжиною знаходиться при використанні кросоверу OX з ініціалізацією методом EI (3880514 на графі pr1002). Довжина знайденого шляху при конфігурації з кросовером IOX: для ініціалізації EI – 4028028, для EG – 4063712.

При розгляді результатів роботи при різних конфігураціях з селекцією турніром виявлено, що найкоротший шлях в цій частині досліду знаходить варіант з кросовером OX при налаштуванні EI – 3626139, що є найкращим результатом серед усіх експериментів даного етапу досліджень. Довжина маршруту, знайденого варіантом із кросовером PMX та порядком задання параметрів EI – 4220263. Конфігурація з OX і налаштуванням EG надала шлях довжиною 4393589, з кросовером IOX та ініціалізацією EI – 4475845.

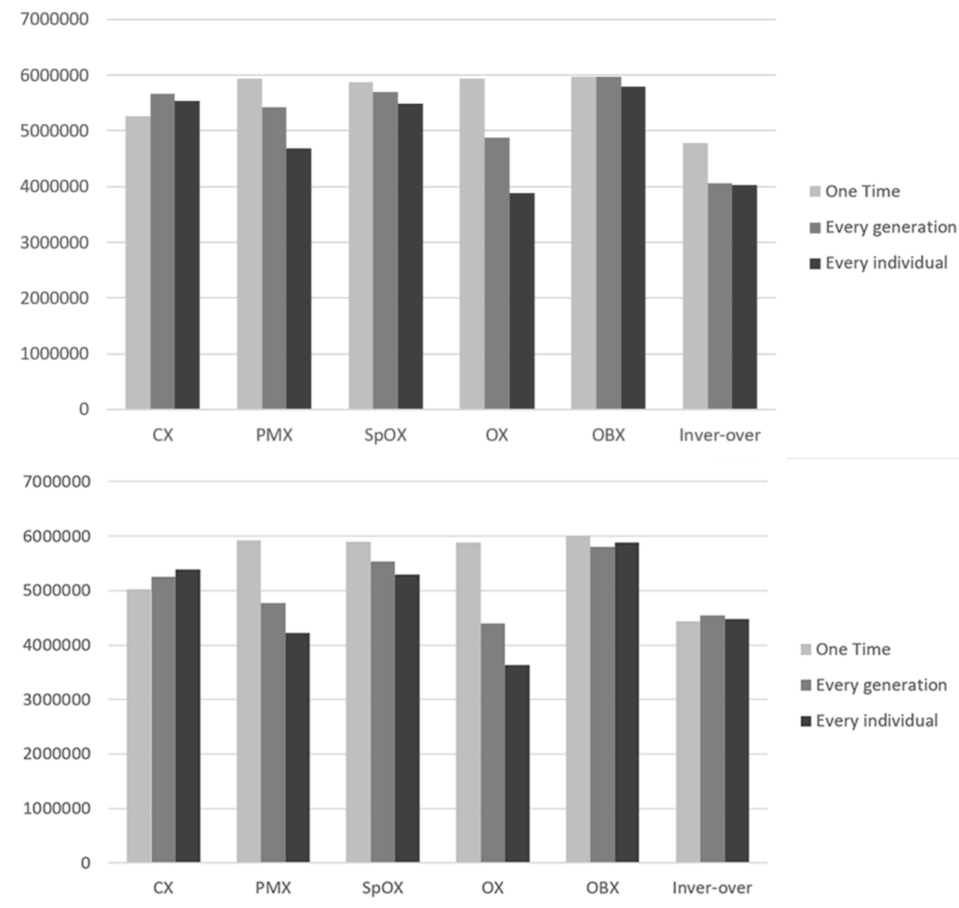

Рисунок 4.1 – Довжини маршрутів, знайдених ГА, при використанні різних операторів селекції, кросоверів та методів ініціалізації параметрів

З огляду на час виконання алгоритму (рисунок 4.2), конфігурація з кросовером IOX показала найвищу швидкість при використанні обох видів відбору. Так, для методу рулетки алгоритми з IOX виконувалися в 11-12 разів швидше за варіанти з OX (28 та 326 секунд відповідно для EG, 26 та 321 для EI), при турнірній селекції – в 10-12 разів (29 та 313 секунд в режимі EG, 25 та 314 для EI). У порівнянні з конфігурацією, яка застосовує PMX, конфігурація з кросовером IOX призведе до здійснення генерації того ж числа поколінь в 4 рази швидше (109 та 25 секунд для селекції турніром та налаштування EI). Найкраще з отриманих співвідношень довжин знайдених шляхів та часу виконання зумовлює вибір конфігурації, що передбачає використання кросоверу IOX, метод рулетки для відбору та варіантом ініціалізації EI для ГА для проведення подальших експериментів.

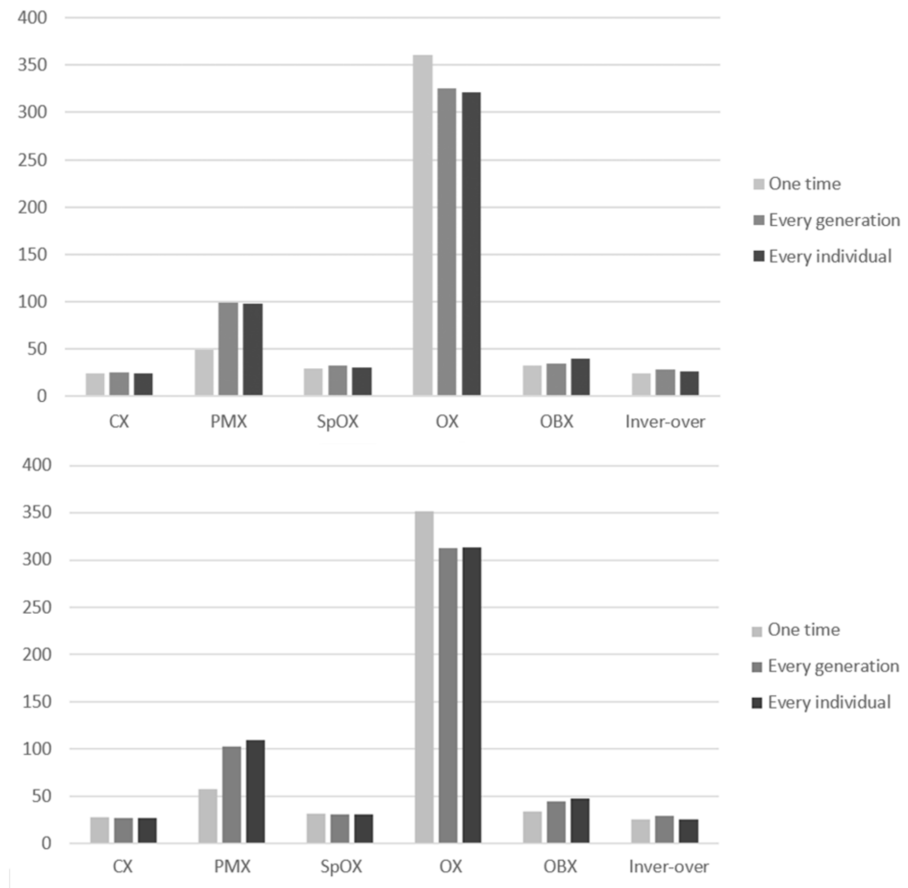

Рисунок 4.2 – Час виконання 500 поколінь ГА, при використанні різних операторів селекції, кросоверів та методів ініціалізації параметрів

Проведені експерименти також показали, що обрана конфігурація (кросовер IOX, ініціалізація параметрів EI, селекція методом рулетки) відрізняється як здатністю до пошуку кращих результатів за одне й те ж число поколінь порівняно з іншими варіантами, так і стійкістю до передчасної збіжності, яка дає змогу не використовувати оператор мутації для підвищення різноманітності генотипу і дозволяє тим самим зменшити час роботи алгоритму.

З використанням зазначеної найкращої конфігурації проведена серія експериментів, в ході якої визначався необхідний розмір популяції та число поколінь, для яких генетичний алгоритм надає найкоротші маршрутів за прийнятний час (4.6).

4.4 Дослідження мурашиного алгоритму

Дослідження використання МА вирішено почати зі встановлення найкращого співвідношення параметрів α та β для загального підходу Ant System. Початкові умови експериментів обрані наступні:

- кількість агентів – 25;
- кількість одиниць феромону агенту – 100;
- кількість ітерацій – 25;
- коефіцієнт випаровування – 50%;
- α змінюється від 0,5 до 5 з кроком 0,5, β змінюється від 1 до 5 з кроком 1 – усього 50 експериментів.

Результати досліду (рисунок 4.3), показують, що кращі результати досягаються при α від 1,5 та вище, та β від 2 та вище. Найкоротший маршрут на графі pr1002 (365244) отриманий при α=2,5 та β=5. Згодом, цей результат для алгоритму Ant System було покращено (361885) при збільшенні кількості агентів та ітерацій до 35 (4.6).

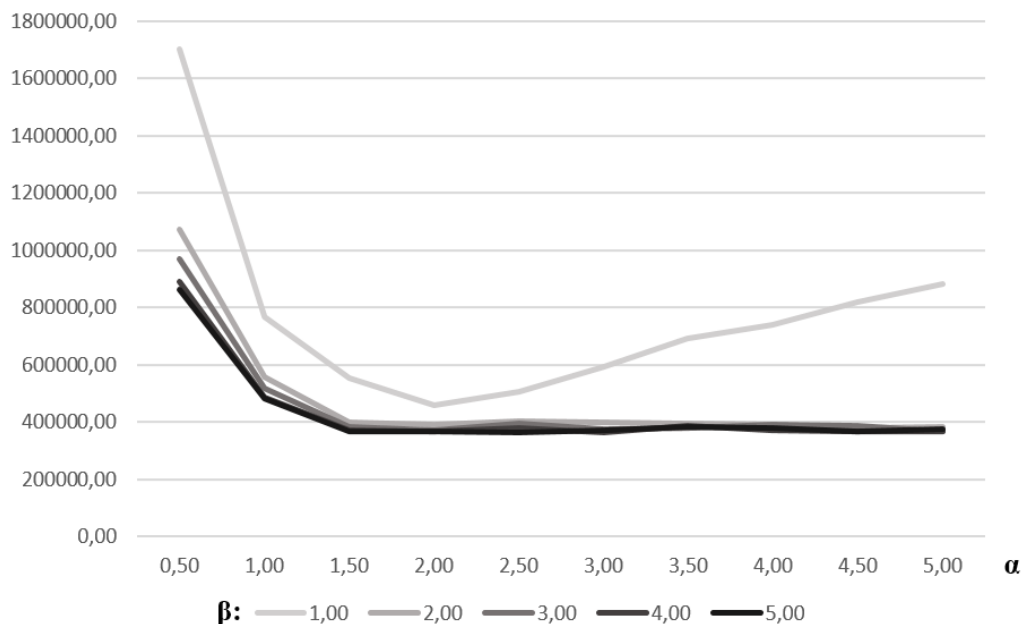

Рисунок 4.3 – Вплив параметрів α та β на довжину маршруту, знайденого мурашиним алгоритмом

Проведено дослідження доцільності застосування концепції елітних агентів. П’яти агентам (з загальної популяції 25) наданий відповідний клас, кількість феромону, що додається ними змінюється від 125 до 350 одиниць кроком в 25 одиниць. Найкращий результат отримано при кількості феромону 225 одиниць (рисунок 4.4). Надалі це значення використовувалося для отримання шляхів з мінімальною довжиною (4.6).

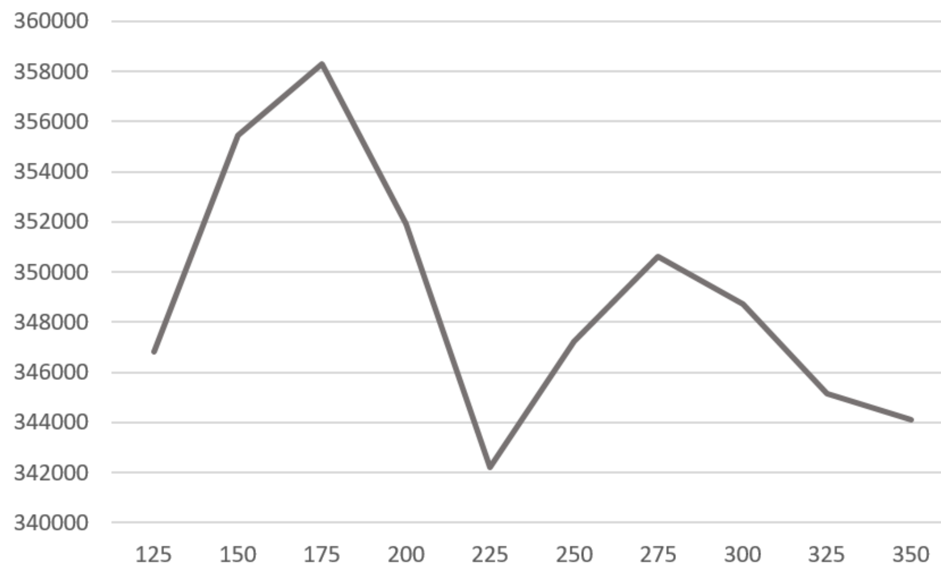

Рисунок 4.4 – Залежність довжини мінімального маршруту від кількості феромону, що залишають елітні агенти

4.5 Дослідження використання самоорганізуючої карти Кохонена

При дослідженні використання SOM для вирішення ЗК, важливим аспектом є уникнення відгалужень в процесі пошуку маршруту, що досягається лінійним збільшенням кількості нейронів в мережі (розділ 3.5.4).

Для пошуку кращого коефіцієнта збільшення, проведений дослід, в якому вказаний параметр змінювався від 1 (стандартний розмір мережі, кількість нейронів відповідає кількості вершин графу) до 10 із кроком в одиницю. Відповідно результатам експерименту (рисунок 4.5) зроблено висновок, що тенденція до зменшення довжини знайденого шляху

проявляється при збільшенні кількості нейронів у 2 та 3 рази. Оскільки вибір більших множників не має позитивного впливу на результат і збільшує час виконання алгоритму лінійно, доцільним є вибір коефіцієнту збільшення рівного 3.

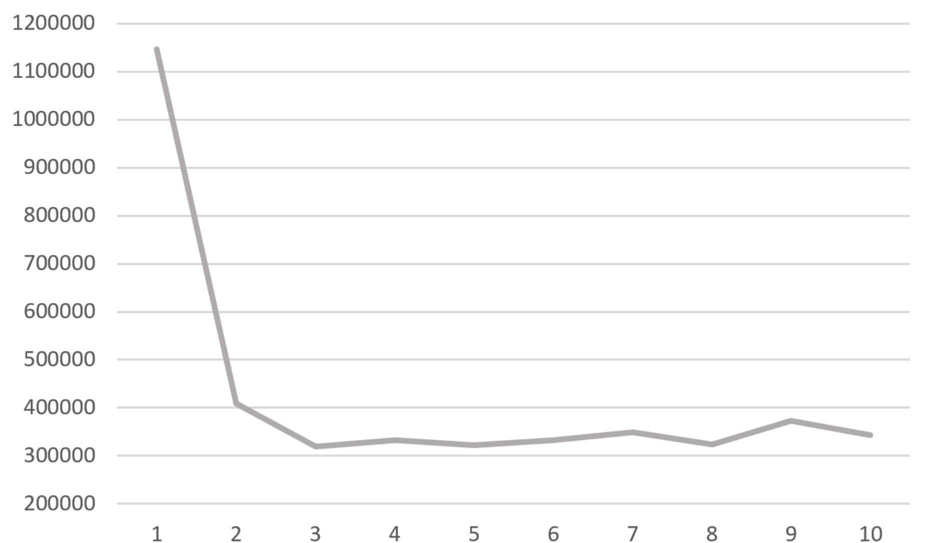

Рисунок 4.5 – Залежність довжини маршруту від значення множника кількості нейронів для графу pr1002

В процесі дослідження, для графів різної розмірності не виявлено закономірності між отримуваним результатом та значеннями таких параметрів мережі, як темп зростання штрафних коефіцієнтів Δp та коефіцієнту швидкості навчання μ. Відповідно, налаштування мережі для пошуку кращих значень цих параметрів проводилося окремо для кожного з представлених графів. В результаті виконання серій експериментів для встановлення значень Δp та μ, які забезпечують знаходження найкоротших маршрутів для кожного з трьох графів, були отримані наступні значення: для tsp225 – 18% та 35% відповідно, для pcb442 – 7,5% та 35%, для pr1002 – 20,5% та 60%. Значення інших параметрів роботи штучної нейронної мережі обрані наступні: еластичність мережі – $\sigma = N^2$, де N – кількість вершин на графі, темп зменшення еластичності мережі $\Delta\sigma = 0{,}1\%$.

4.6 Аналіз результатів досліджень

В результаті виконання експериментів, отримані зведені дані роботи для всіх розглянутих алгоритмів та трьох графів ЗК. Результати включають довжини отриманих маршрутів, час виконання алгоритмів та відсоток, на який відрізняються знайдені рішення від довжини шляхів відомих найкращих маршрутів (таблиця 4.3). L_{best} – довжина найкоротшого маршруту з бібліотеки TSPLIB, L – знайдена довжина маршруту, t – час роботи алгоритму.

Таблиця 4.3 – Зведені результати пошуку найкоротших шляхів досліджуваними алгоритмами

Граф			tsp225	pcb442	pr1002
L_{best}			3919	50778	259045
ГА	L		4252	58076	342513
	Error, %		8,5	14,37	32,22
	t, c		9	68	2944
МА	AS	L	4968	66595	361885
		Error, %	26,77	31,15	39,7
		t, c	8	32	144
	EA	L	4754	66646	349750
		Error, %	21,31	31,15	35,01
		t, c	6	23	127
	MM	L	4385	62581	320163
		Error, %	11,89	23,24	23,59
		t, c	8	29	137
SOM	L		5514	60681	309255
	Error, %		40,7	19,5	19,38
	t, c		3	8	24

Проведені дослідження показали, що найкращі результати для графів розміром у 225 та 442 вершини забезпечує генетичний алгоритм (4252 та 58076, при найкращих маршрутах довжиною 3919 та 50778 відповідно), але для графу на 1002 вершини довжина знайденого шляху поступається результатам роботи Ant System, елітарної модифікації та карті Кохонена. Час, необхідний для пошуку маршрутів, у випадку ГА виявився найбільшим з усіх розглянутих підходів. Зі зростанням розміру графу до 1002 вершин, процес пошуку прийнятного результату за допомогою ГА може тривати до години, що ускладнює застосування алгоритму при вирішенні прикладних задач. Найкращий результат для найбільшого графу має нейронна мережа – 309255 при відомій найменшій довжині в 259045.

Виявлено, що карта Кохонена дозволяє знайти наближені рішення значно швидше за інші розглянуті алгоритми, при чому ця різниця зростає із збільшенням кількості вершин графу – SOM швидший за інші алгоритми у 2-3 рази для задачі на 225 вершин та 3-8 разів для 442 вершин. На графі з 1002 вершинами, SOM показує час у 5-6 разів швидший за мурашині алгоритми. Подібне спостереження дозволяє зробити висновок про доцільність використання даного підходу для вирішення ЗК високих розмірностей.

4.7 Можливі напрямки подальших досліджень

Для подальших досліджень вирішення ЗК на основі методів обчислювального інтелекту є доцільним розгляд наступних питань:

- пошук швидких автоматизованих методів підбору початкових значень параметрів для самоорганізуючих карт Кохонена;
- можливість використання інших архітектур ШНМ для вирішення ЗК;
- аналіз більшої кількості існуючих та розробка нових модифікацій мурашиного алгоритму;
- дослідження нових реалізацій генетичних операторів для ГА.

ВИСНОВКИ

В даній роботі розглянуті підходи до вирішення задачі комівояжера з використанням методів обчислювального інтелекту, таких як генетичні алгоритми, мурашині алгоритми та самоорганізуюча карта Кохонена. Проведено аналіз існуючих досліджень щодо ефективності застосування зазначених підходів. Виявлені недоліки, такі, як передчасна збіжність ГА та проблема появи відгалужень на маршрутах SOM.

Реалізовано програмний застосунок, що дозволяє запускати розглянуті алгоритми на графах ЗК згенерованих випадковим чином або завантажених з бібліотеки TSPLIB. У застосунку передбачена можливість автоматизації проведення дослідів із можливостями визначення впливу окремих вхідних параметрів та збереження отриманих результатів у файл, що дозволило пришвидшити та спростити процес експериментального дослідження та аналізу отриманих результатів.

Експериментальні дослідження на трьох графах ЗК різного розміру показали, що найбільшу точність рішень для задач розмірністю у 225 та 442 вершини має генетичний алгоритм, а для задачі на 1002 міста найкоротший шлях був знайдений за допомогою ШНМ. Також мережа Кохонена виявилася найшвидшим методом пошуку наближених рішень, що має важливе значення при практичному використанні ЗК.

Проведений аналіз збіжності ГА при використанні різних реалізацій генетичних операторів та способів їх налаштувань, найкращі результати отримано для конфігурації із кросовером типу Inver-over, та методом ініціалізації його параметрів, який встановлює нові випадкові значення для кожної особини популяції на кожному поколінні.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Іващенко, Г. С. Методи рішення задачі комівояжера на основі обчислювального інтелекту [Текст] / Г. С. Іващенко, О. І. Онищенко, М. Е. Бондаренко, Н. В. Здорик // Системи управління, навігації та зв'язку. – 2024. – №2. – С. 99–105.

2. Euler, L. Solution d'une question curieuse que ne paroit soumise à aucune analyse [Текст] / L. Euler // Mémoires de l'académie des sciences de Berlin. – 1766. – Vol. 15. – P. 310–337.

3. Voigt, B. F. Der Handlungsreisende wie er sein soll und was er zu thun hat, um Aufträge zu erhalten und eines glücklichen Erfolgs in seinen Geschäften gewiß zu sein: Mit einem Titelkupfer [Текст] / B. F. Voigt. – 1832. – 203 p.

4. Menger, K. Das Botenproblem [Текст] / K. Menger // Ergebnisse eines Mathematischen Kolloquiums, Springer – New York. – 1932. – Vol. 2. – P. 11–12.

5. Robinson, J. B. On the Hamiltonian game (a traveling-salesman problem) [Текст] / J. B. Robinson // RAND Corporation. – 1949. – 10 p.

6. Karp, R. M. Reducibility among Combinatorial Problems [Текст] / R. M. Karp. // Complexity of Computer Computations. The IBM Research Symposia Series. Springer. – Boston, MA. – 1932. – P. 85–103.

7. Crowder, H. Solving Large-Scale Symmetric Travelling Salesman Problems to Optimality [Текст] / H. Crowder, M. W. Padberg // Management Science. – 1980. – Vol. 26. – No. 5. – P. 495–509.

8. Padberg, M. Optimization of a 532-city symmetric traveling salesman problem by branch and cut [Текст] / M. Padberg, G. Rinaldi // Operations Research Letters. – 1987. – Vol. 6. – No. 1. – P. 1–7.

9. Grötschel, M. Solution of large-scale symmetric travelling salesman problems [Текст] / M. Grötschel, O. Holland // – 1991. – Vol. 51. – P. 141–202.

10. Applegate, D. L. The Traveling Salesman Problem: A Computational Study [Текст] / D. L. Applegate // Princeton University Press. – 2006. – 606 p.

11. Davendra, D. Travelling Salesman Problem, Theory and Applications [Текст] / D. Davendra // InTech. – 2010. – 338 p.

12. Deng, Y. An Improved Genetic Algorithm with Initial Population Strategy for Symmetric TSP [Текст] / Y. Deng, Y. Liu, D. Zhou // Advanced Techniques for Computational and Information Sciences. – 2015. – P. 1–6.

13. Roberti, R. Models and algorithms for the Asymmetric Traveling Salesman Problem: an experimental comparison [Текст] / R. Roberti, P. Toth // Journal on Transportation and Logistics. – 2012. – Vol. 1. – No. 1. – P. 113–133.

14. Lahyani, R. A unified matheuristic for solving multi-constrained traveling salesman problems with profits [Текст] / R. Lahyani, M. Khemakhem // Journal on Computational Optimization. – 2017. – №5(3). – P. 393–422.

15. Dhanasekar, S. A Branch and Bound Algorithm to Solve Travelling Salesman Problem (TSP) with Uncertain Parameters [Текст] / S. Dhanasekar, S. K. Dash // Mathematics and Statistics. – 2022. – Vol. 10. – No. 4. – P. 358–365.

16. Geng, X. Solving the traveling salesman problem based on an adaptive simulated annealing algorithm with greedy search [Текст] / X. Geng, Z. Chen, W. Yang // Applied Soft Computing. – 2011. – Vol. 11. – No. 4. – P. 3680–3689.

17. Kizilateş, G.. On the Nearest Neighbor Algorithms for the Traveling Salesman Problem [Текст] / G. Kizilateş, F. Nuriyeva // Advances in Computational Science, Springer – Heidelberg. – 2013. – P. 111–118.

18. Eskandari, S. Two new selection methods and their effects on the performance of genetic algorithm in solving supply chain and travelling salesman problems [Текст] / S. Eskandari, M. K. Rafsanjani // International Journal of Bio-Inspired Computation (IJBIC). – 2023. – Vol. 22. – No. 3. – P. 176–184.

19. Ramadan, S. Reducing Premature Convergence Problem in Genetic Algorithm: Application on Travel Salesman Problem [Текст] / S. Z. Ramadan // Computer and Information Science. – 2021. – Vol. 6. – No. 1. – P. 47–57.

20. Wang, Y. Ant colony optimization for traveling salesman problem based on parameters optimization [Текст] / Y. Wang, Z. Han. // Applied Soft Computing. – 2021. – Vol. 107. – P. 1–11.

21. Yue, Y. An Improved Ant Colony Optimization Algorithm for Solving TSP [Текст] / Y. Yimeng, X. Wang // International Journal of Multimedia and Ubiquitous Engineering. – 2015. – Vol. 10. – No. 12 – P. 153–164.

22. Zhang, J. An overall-regional competitive self-organizing map neural network for the Euclidean traveling salesman problem [Текст] / J. Zhang, X. Feng, B. Zhou, D. Ren // Neurocomputing. – 2012. – Vol. 89. – P. 1–11.

23. Sasamura, H. A simple learning algorithm for growing ring SOM and its application to TSP [Текст] / H. Sasamura, R. Ohta, T. Saito // Proceedings of the 9th International Conference on Neural Information Processing. – 2002. – Vol. 3. – P. 1287–1290.

24. TSPLIB [Електронний ресурс] – Режим доступу : www/ URL: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/ – 29.03.2024 р. – Загол. з екрану.

25. Lambora, A. Genetic Algorithm – A Literature Review [Текст] / A. Lambora, K. Gupta, K. Chopra // 2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing. – 2019. – P. 380–384.

26. Singh, D. R. A Hybrid Algorithm with Modified Inver-Over Operator and Genetic Algorithm Search for Traveling Salesman Problem [Текст] / D. R. Singh, M. K. Singh, T. Singh // Advanced Computing and Communication Technologies. – 2016. – P. 141–150.

27. Kumar, R. Analyzing The Performance Of Crossover Operators (OX, OBX, PBX, MPX) To Solve Combinatorial Problems [Текст] / R. Kumar, M. Memoria, M. Thapliyal, M. Kirola, I. Ahmad, A. Gupta, S. Tyagi, N. Ansari // 2022 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COM-IT-CON). – 2022. – P. 817–821.

28. Shafie, M. F. Optimization of Saleman Travelling Problem Using Genetic Algorithm with Combination of Order and Random Crossover [Текст] / M. F. Shafie, F. Ahmad, M. K Osman, A. P. Ismail, K. A. Ahmad, S. Z. Yahaya // 2023 IEEE 13th International Conference on Control System, Computing and Engineering (ICCSCE). – 2023. – P. 255–258.

29. Przewozniczek, M. Towards Finding an Effective Uniform and Single Point Crossover Balance for Optimization of Elastic Optical Networks [Текст] / M. Przewozniczek // 2015 Second European Network Intelligence Conference. – 2015. – P. 40–46.

30. Deep, K. Variant of partially mapped crossover for the Travelling Salesman problems [Текст] / K. Deep, H. Mebrahtu // International Journal of Combinatorial Optimization Problems and Informatics. – 2012. – Vol. 3. – No. 1. – P. 47–69.

31. OuYang, Q. The study of comparisons of three crossover operators in genetic algorithm for solving single machine scheduling problem [Текст] / Q. OuYang, H. Xu // International Conference on Manufacturing Science and Engineering (ICMSE 2015), Atlantis Press. – 2015. – P. 293–297.

32. Chopard, B. The Ant Colony Method [Текст] / B. Chopard, M. Tomassini // An Introduction to Metaheuristics. – 2018. – P. 81–96.

33. Colorni, A. Distributed Optimization by Ant Colonies [Текст] / A. Colorni, M. Dorigo, V. Maniezzo // Proceedings of ECAL91 – European Conference on Artificial Life, Elsevier Publishing. – 1991. – P. 134–142.

34. Dorigo, M. The Ant System: Optimization by a colony of cooperating agents [Текст] / M. Dorigo, V. Maniezzo, A. Colorni // IEEE Transactions on Systems, Man, and Cybernetics. – 1996. – Vol. 26. – No. 1. – P. 29–41.

35. Stützle, T. MAX-MIN Ant System [Текст] / T. Stützle, H. H. Hoos // Future Generation Computer Systems. – 2000. – No. 1. – P. 889–914.

36. Sarikyriakidis, S. Using Self-organizing Maps to Solve the Travelling Salesman Problem: A Review [Текст] / S. Sarikyriakidis, K. Goulianas, A. I. Margaris // WSEAS transactions on systems. – 2023. – Vol. 22. – P. 131–159.

37. Faigl, J. Autonomous Data Collection Using a Self-Organizing Map [Текст] / J. Faigl, G. A. Hollinger // IEEE Transactions on Neural Networks and Learning Systems. – 2018. – Vol. 29. – No. 5. – P. 1703–1715.