Київський національний університет імені Тараса Шевченка

Факультет комп’ютерних наук та кібернетики

Кафедра теоретичної кібернетики

Кваліфікаційна робота

на здобуття ступеня бакалавра

за спеціальністю 122 «Комп’ютерні науки»

на тему:

Наближене розв’язання геометричної задачі комівояжера з дискретними відстанями

Студента 4-го курсу
Пилипчука Сергія Ярославовича

(підпис)

Науковий керівник:
доцент Ставровський Андрій Борисович

____
(підпис)

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

Завідувач кафедри Крак Ю.В.

Київ – 2021

1. РЕФЕРАТ

Ключові слова: ЗАДАЧА КОМІВОЯЖЕРА.

Об'єктом роботи є розробка програмного засобу для розв'язання геометричної задачі комівояжер з дискретними відстанями за допомогою мурашиного алгоритму та генетичного алгоритму. При розробці програмного засобу буде проведений огляд згаданих алгоритмів з подальшим дослідженням їх швидкодії і якості.

В якості засобу розроблення системи було обрано мову програмування Rust.

Метою роботи є розробка бібліотечного модулю мовою програмування Rust, який буде реалізовувати алгоритм, що розв'язує геометричну задачу комівояжера за допомогою ГА або МА.

Методи розроблення: генерація випадкових даних для дослідження алгоритмів, застосування алгоритмів на цих даних, побудова таблиць, судження щодо якості та швидкодії алгоритмів.

Інструменти розроблення: мова програмування Rust.

Результати роботи: виконано загальний огляд згаданих наближених розв'язків геометричної задачі комівояжера з дискретними відстанями, реалізовано бібліотечний модуль із застосуванням ГА і МА, надано приклади роботи алгоритму із вхідними даними, параметрами системи на якій запущено процес та вихідними даними.

Отримані результати можна буде застосовувати в практичних задачах. Бібліотечний модуль дозволить будь-якому програмісту застосувати розроблене рішення для оптимізаційних задач бізнесу. Наприклад мінімальне огородження території по заданим точкам. Дослідницька частина ж дасть вичерпну інформацію щодо швидкодії та якості алгоритмів за заданих параметрів на випадкових даних.

Новизна полягає в першому у своєму роді бібліотечному модулі мовою програмування Rust для розв'язку геометричної задачі комівояжера. А також в першому дослідженні швидкодії і якості алгоритмів створеного бібліотечного модулю.

ЗМІСТ

2.

РЕФЕРАТ 2

ЗМІСТ 4

ВСТУП 8

РОЗДІЛ 1 10

1.1 Постановка задачі комівояжера 10

1.2 Формулювання Міллера-Цукера-Зелміна 10

1.3 Формулювання Дантціга-Тукера-Зелміна 11

1.4 Геометрична задача комівояжера 12

РОЗДІЛ 2 14

2.1 Мурашиний алгоритм 13

2.2 Генетичний алгоритм 15

РОЗДІЛ 3 19

3.1 Visual Studio Code 18

3.2 Мова програмування Rust 18

РОЗДІЛ 4 21

4.1 Трейти алгоритмів 20

4.1.1 Трейт TSPSolver 20

4.1.2 Трейт Distance 20

4.2 Структура точки на сфері 21

4.2.1 Обчислення відстані між точками на сфері 21

4.3 Генерація тестових даних 22

4.3.1 Бібліотека rand 22

4.3.2 Генерація випадкової точки на сфері 22

4.3.3 Генерація масиву випадкових точок із заданою довжиною 22

4.4 Реалізація алгоритмів наближеного розв’язку задачі комівояжера 22

4.4.1 Мурашиний алгоритм 23

4.4.1.1 Структура AntAlgTSPSolver 23

4.4.1.2 Реалізація трейту TSPSolver для структури AntAlgTSPSolver 23

4.4.2 Генетичний алгоритм 25

4.4.2.1 Функція кросоверу популяції 25

4.4.2.2 Функція «якості» 26

РОЗДІЛ 5 29

5.1 Дослідження ГА 27

5.1.1 Результати дослідження ГА 28

5.1.2 Висновки досліджень ГА 32

5.2 Дослідження МА 32

5.2.1 Результати дослідження МА 33

5.2.2 Висновки досліджень МА 36

ВИСНОВКИ 38

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 39

ДОДАТКИ 40

ДОДАТОК А Код бібліотеки 40

А.1 48

А.2 48

А.3 49

А.4 50

А.5 50

А.6 51

А.7 51

А.8 52

А.9 55

А.10 60

А.11 61

А.12 61

ДОДАТОК Б Посилання на код і дані 56

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

ЗК – задача комівояжера

ГА – генетичний алгоритм

МА – мурашиний алгоритм

3. ВСТУП

Оцінка сучасного стану об’єкта розробки.

Задача комівояжера дуже відома серед математиків і не тільки. Точна дата коли її задали невідома, та перші згадки датуються 1832 році у книжці «Комівояжер — як він має поводитись і що має робити для того, аби доставляти товар та мати успіх в своїх справах — поради старого Кур'єра».

Проблема полягає у знаходженні оптимального за деяким критерієм шляху між скінченною кількістю точок з поверненням у початкову точку, тобто шлях утворює цикл. Це можуть бути точки графу, площини або інші.

Геометрична задача комівояжера полягає у відшукуванні найкращого шляху між точками з заданими координатами. Обмеження на дискретні відстані слугує в якості покращувальної умови, яка не дозволяє розташовувати точки на нескінченності.

За кілька століть існування задачі знайдено чимало наближених рішень геометричної задачі комівояжера. Ми будемо розглядати деякі з них.

Актуальність роботи та підстави для її виконання.

Геометрична задача комівояжера є однією з найважливіших практичних задач у таких сферах як логістика, телекомунікаційні мережі, енергетика, військова сфера.

Для логістичних задач без наперед заданих шляхів, наприклад доставка вантажів авіацією, геометрична задача комівояжера виникає цілком природньо.

Для телекомунікаційних мереж важлива мінімізація витрат на матеріали, тому часто доводиться будувати мінімальний шлях між точками і маршрутизаторами.

В енергетиці потрібно економити на дротах електромережі і мати при цьому замкнутий електричний цикл.

У військовій сфері виникає геометрична задача комівояжера при авіа ударах з кількома цілями. Також алгоритми можуть бути застосовані до

проблеми мінімізації мінного поля навколо деякого об’єкту, побудови огорож, ровів і так далі.

Метою кваліфікаційної роботи є створення програмної системи, яка за оптимальний час розв’язує геометричну задачу комівояжера на даних Open Street Map, використовуючи наближені алгоритми.

Для досягнення цієї мети поставлені наступні **завдання:**

- поглибити знання у мові програмування Rust;
- знайти і дослідити алгоритми наближеного розв’язку геометричної задачі комівояжера, а саме – мурашиний алгоритм та генетичний алгоритм;
- реалізувати алгоритми мовою програмування Rust
- порівняти їхню ефективність на випадкових даних із деякими заданими параметрами;

Об’єкт розроблення – процес створення програми для знаходження найкоротшого шляху між містами за допомогою наближеного алгоритму.

В якості **засобу розроблення** програмного засобу було обрано Visual Studio Code, яка вільно поширюється компанією Microsoft. Реалізовано алгоритм буде мовою Rust з відкритим кодом, що розповсюджується за ліцензією MIT та Apache License і підтримується некомерційною організацією Rust Foundation [1].

Можливі сфери застосування.

Робота може бути використана у якості дослідного матеріалу для обрання оптимального за часом алгоритму геометричного розв’язку задачі комівояжера. Реалізація бібліотеки дозволяє проводити самостійні дослідження. Бібліотека реалізована сучасною і популярною системною мовою програмування, що дозволяє їй бути швидкою і легко підтримуваною.

Взаємозв’язок з іншими роботами.

Роботу було виконано на основі відомих теоретичних знань про задачу комівояжера.

РОЗДІЛ 1
ЗАДАЧА КОМІВОЯЖЕРА ТА ГЕОМЕТРИЧНА ЗАДАЧА КОМІВОЯЖЕРА

1.1 Постановка задачі комівояжера

Задана множина міст і ціна переміщення між ними. Задача комівояжера полягає у знаходженні маршруту що проходить через усі міста і закінчується у початковій точці, тобто утворює цикл. Маршрут це послідовність відвідування міст.

1.2 Формулювання Міллера-Цукера-Зелміна

Позначимо міста номерами 1,…,n і означимо:

$x_{ij} = \{1$ є шлях від міста i до міста j 0 в іншому випадку

Для i = 1,…,n , та u_i – деяка змінна, і також задамо $c_{ij} > 0$ – відстань від міста i до міста j. Тоді ЗК може бути визначена у вигляді задачі лінійного програмування:

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

З такими обмеженнями:

$x_{ij} \in \{0,1\};\ i, j = 1, \dots, n;$

$$u_i \in Z;\ i = 2, \dots, n;$$

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

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

$$u_i - u_j + nx_{ij} \leq n - 1; 2 \leq i \neq j \leq n$$

$$1 \leq u_i \leq n - 1; 2 \leq i \leq n$$

Перша рівність вимагає від розв’язку, щоб з кожного міста був лише один шлях до іншого, друга рівність затверджує неможливість мати більше одного шляху з міста. Останні нерівності гарантують, що існує єдиний маршрут, що обходить усі міста. [2]

1.3 Формулювання Дантціга-Тукера-Зелміна

Позначимо міста номерами 1,…,n і означимо:

$x_{ij} = \{1$ є шлях від міста i до міста j 0 в іншому випадку

Для i = 1,…,n , та u_i – деяка змінна, і також задамо $c_{ij} > 0$ – відстань від міста i до міста j. Тоді ЗК може бути записана такою задачею лінійного програмування:

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

З такими обмеженнями:

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

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

$$\sum_{i \in Q} \sum_{i \neq j, j \in Q} x_{ij} \leq |Q| - 1;\ \forall Q \subseteq \{1, \dots, n\}, |Q| \geq 2$$

Останнє обмеження забезпечує неможливість існування під-шляху у знайденому шляху, що задовольняє усім вище умовам, тобто який є ще одним розв’язком задачі. [3]

1.4 Геометрична задача комівояжера

Даний підвид задачі комівояжера полягає в переозначенні «міст» як точок у просторі. Відповідно у даному просторі повинна існувати «відстань» між точками, в іншому випадку ми не зможемо задати функцію для мінімізації. Функція відстані між двома точками має назву «метрики». Простір може бути як евклідовим, так і не евклідовим. Доведено можливість розв'язку задачі комівояжера за поліноміальний час у евклідовому просторі.

На превеликий жаль карта Землі аж ніяк не евклідовий простір, тому більшість можливих оптимізацій досить умовні і працюють на невеликих площах. Загальні наближені алгоритми розв'язання ЗК вирішують реальні задачі краще.

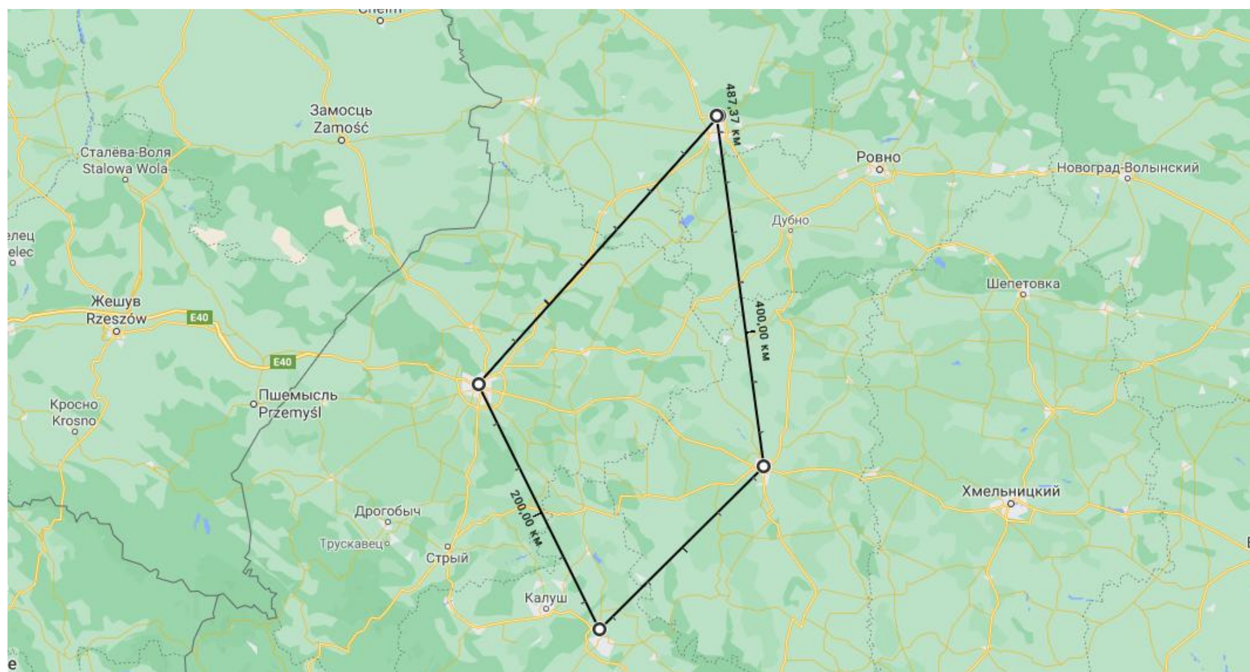


Рисунок 1.1 Приклад використання ЗК для туру по Галичині

1.5 Геометрична задача комівояжера з дискретними відстанями

Базова задача зберігається. Необхідно знайти найменший маршрут, що проходить через усі міста (точки) по одному разу.

Для дослідження роботи алгоритмів розв'язку ЗК з дискретними відстанями нам необхідно мати простір, у якому відстані між точками дискретні.

Найпростіший випадок це простір, у якому відстань від точки до самої себе 0 і у всіх інших випадках 1. Та розв'язувати ЗК у такому просторі немає сенсу, очевидно, що множина розв'язків ЗК співпадатиме з множиною можливих маршрутів з відвідуванням кожної точки один раз.

Тому ми візьмемо трохи складніший простір. Нехай відстань від точки до самої себе дорівнює 0. Кожній точці у відповідність ми надамо деяке натуральне число. Якщо ми маємо точку x, то відповідне їй число це A_x. *Тоді* відстань d(x, y) буде обчислюватись за формулою:

$$d(x,y) = \{0, \qquad x = y\, mod\left(\left(A_x - A_y\right), 2\right) + 1, \;\; x \neq y$$

Тобто можливі значення відстані це 0, 1, 2.

Даний простір є метричним, бо виконуються 3 необхідні умови.

1. d(x,x) = 0.
2. d(x,y) = d(y,x).
3. d(x,z) <= d(x,y) + d(y,z).

Доведення останнього пункту елементарне. Для спростування нам необхідно максимізувати ліву частину і мінімізувати праву. Найбільша можлива між точками відстань – 2, найменша – 1. 2 <= 1 + 1 . 2 <= 2. True. Доведено.

Маючи простір даних точок ми зможемо розв'язувати у ньому ЗК.

РОЗДІЛ 2
НАБЛИЖЕНІ АЛГОРИТМИ РОЗВ'ЯЗКУ ГЕОМЕТРИЧНОЇ ЗАДАЧІ КОМІВОЯЖЕРА

У даному розділі будуть згадані найбільш популярні і відомі алгоритми для розв'язку задачі комівояжера. А також розглянемо алгоритм розв'язку геометричної ЗК.

2.1 Мурашиний алгоритм

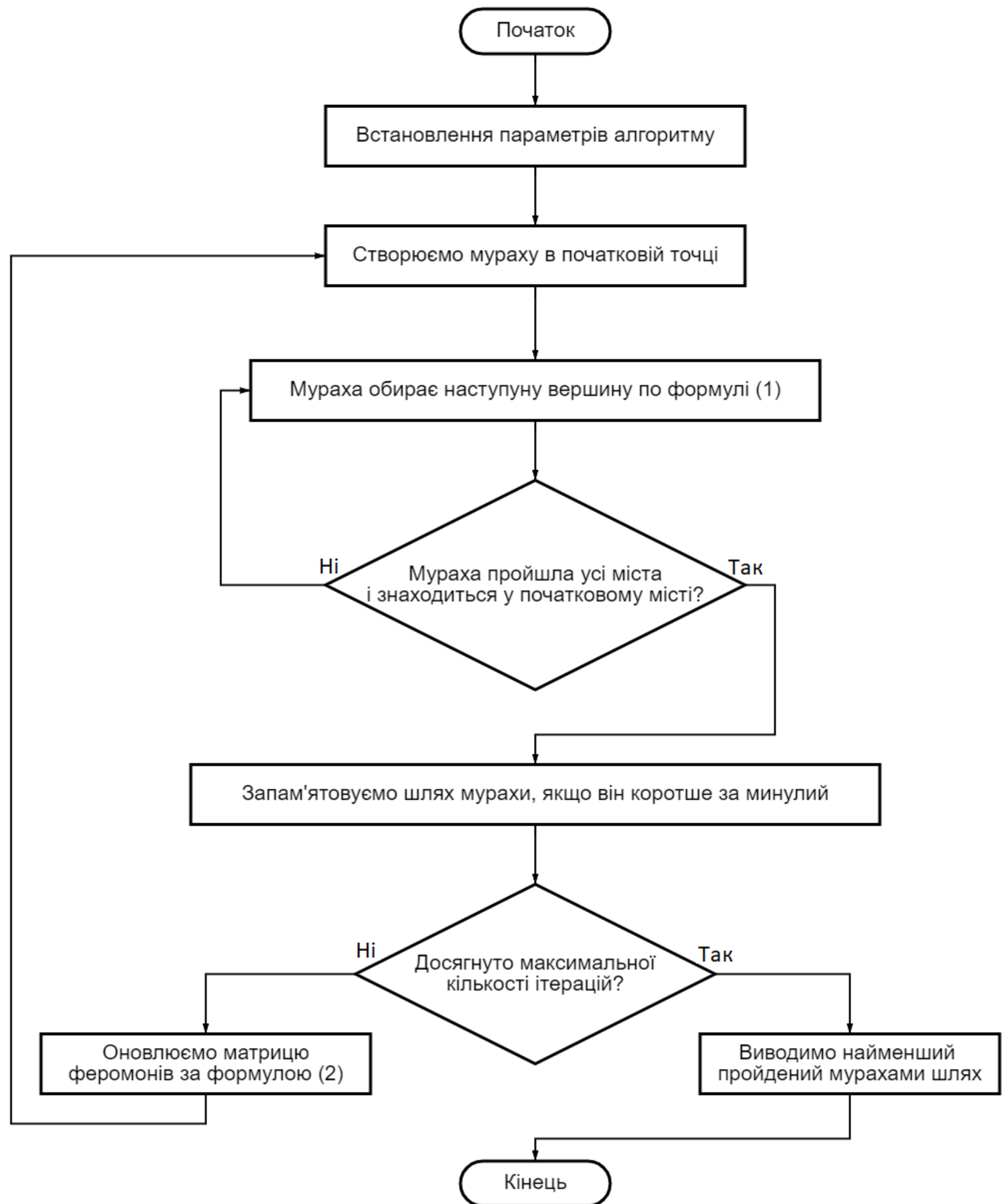
Класичний наближений алгоритм розв'язку задачі комівояжера. Цілком підходить для геометричної задачі комівояжера.

Запропонований у 1991 році трьома видатними науковцями: Альтьєро Колорні, Вітторіо Манієццо й Марко Доріго [4].

Цей алгоритм будується на ідеї життя мурашиної колонії, а саме їх способу добування їжі і організації пересування. Мурахи при русі виділяють спеціальний феромон, який здатні розрізнити інші мурахи. Мурашиній колонії часто необхідно переміщувати вантажі із точок з їжею до мурашників. Завдяки виділенню феромону мурахи рухаються найбільш популярними маршрутами, які з часом стають коротше.

У випадку інтерпретації життя колонії як алгоритму застосовується спрощена версія життя мурах. Мураха зазвичай є деякою структурою даних, яка береже свій пройдений шлях і існує функція вибору наступної точки для переміщення. Також феромони це деяке кількісне значення і їх розповсюдження зберігається у матриці феромонів. Мураха виділяє на одниницю шляху кількість феромонів обернено пропорційну пройденій відстані. Процес виділення відбувається після відвідування усіх «міст».

На наступній сторінці наведено блок-схему алгоритму.



$$p_{xy}^{k} = \frac{(\tau_{xy}^{\alpha})(\eta_{xy}^{\beta})}{\sum_{z \in allowed_x} (\tau_{xz}^{\alpha})(\eta_{xz}^{\beta})}$$	(1)

τ_{xy} – кількість феромону для переміщення із x в y,

α – коефіцієнт в межах [0, 1],

β – коефіцієнт в межах [0, 1],

η_{xy} – «привабливість» шляху з x в y, зазвичай дорівнює оберненій відстані між x і y,

$$\square_{xy} = (1-p)\tau_{xy} + \sum_{k}^{m} \Delta\,\tau_{xy}^{k} \quad (2)$$

p – коефіцієнт випаровування феромону,

τ_{xy}^{k}- кількість феромону що залишає k-та мураха на ділянці від x до y.

2.2 Генетичний алгоритм

Генетичні алгоритми (Genetic algorithms, GAs) це еволюційні алгоритми, що базуються на біологічних еволюційних процесах. В природі, найкращі індивіди мають найбільше шансів вижити і розмножитись, також наступне покоління повинно мати кращі здібності до виживання і бути більш розвиненим. Було зроблено чимало роботи щодо генетичних алгоритмів за кілька останніх десятиліть. ГА базуються на роботі з популяцією «хромосом» які кодують деякі параметри системи.

ЗК одна з найкращих та найвідоміших суттєвих, історичних і дуже складних проблем комбінаторної оптимізації. Минулі три десятиліття були продуктивними у сфері дослідження задачі комівояжера. Одним з наслідків цих досліджень стало використання ГА для наближеного розв’язання ЗК за оптимальний час. Природа ГА дозволяє без будь-яких проблем застосовувати свої підходи у розв’язку ЗК.

Простий і узагальнений ГА можна подати у вигляді таких кроків:

1. Створюємо початкову популяцію з N хромосом.
2. Оцінюємо «якість» кожної хромосоми.
3. Обираємо N/2 батьків з наявної популяції за допомогою пропорційного відбору.

4. Випадково обираємо двох батьків, щоб створити потомство, використовуючи оператор кросоверу (перехрещення двох хромосом).
5. Застосовуємо мутації на потомстві, щоб трішечки змінити хромосоми випадковим чином.
6. Повторюємо кроки 4 і 5 доки існують батьки на яких їх ще не виконано.
7. Заміняємо наявну популяцію батьків на популяцію їх нащадків.
8. Оцінюємо «якість» усіх хромосом у новій популяції.
9. Перевіряємо чи ми досягнули наперед заданої кількості ітерацій генетичного алгоритму, якщо ні то переходимо до кроку 3.

Функція оцінки «якості», кросовер та мутації це основні оператори генетичного алгоритму, та кросовер грає основну роль у ГА. Було запропоновано чимало операторів кросоверу для ЗК. Ми скористаємось найбільш ефективним оператором, дослідження по якому були опубліковані у роботі [5].

Даний оператор описується таким алгоритмом:

1. Обираємо двох батьків для спарювання.
2. Беремо перший елемент другого батька у ролі першого елемента нащадка.
3. Обраний елемент з кроку 2 потрібно знайти в першому батькові і обрати елемент з цією ж позицією у другому батькові і цей елемент повинен бути знайдений знову ж у першому батькові і нарешті останній позиція останнього згаданого елементу буде обрана для взяття елементу з другого батька як першого елементу другого нащадка.
4. Обраний елемент з кроку 3 повинен бути знайдений в першому батькові і обраний на цій же позиції з другого батька як наступний елемент першого нащадка.

5. Повторюємо кроки 3 і 4 доки перший елемент першого батька не з'явиться у другому нащадку і алгоритм зможе бути зупинений.
6. Якщо якісь елементи залишились, тоді однакові елементи з першого батька та другого нащадка і навпаки повинні бути видалені з обох батьків. Для елементів що залишились повторюємо кроки 2, 3 і 4 до завершення алгоритму.

Згідно зі згаданою вище робою, даний оператор кросоверу працює часом удвічі краще популярних PMX і OX операторів. Тому ми будемо його застосовувати у нашій роботі.

РОЗДІЛ 3
ІНСТРУМЕНТАЛЬНІ ЗАСОБИ РЕАЛІЗАЦІЇ

Система була розроблена за допомогою текстового редактора Visual Studio Code. Додаток створено мовою Rust з використанням бібліотек з відкритим кодом, які не розв'язують поставлену задачу. Операційна система комп'ютера Windows 10 Pro x64.

3.1 Visual Studio Code

Microsoft Visual Studio Code — текстовий редактор з відкритим кодом розроблений компанією Microsoft для операційних систем Windows, Linux та MacOS. Включає в себе такі можливості: дебаг, підсвітка синтаксису, інтелектуальні пропозиції завершення коду, скрипти, рефакторинг коду і також може виконувати роль зовнішнього Git-редактору. Користувачі здатні змінювати швидкі клавіші, теми, побажання і встановлювати розширення для збагачення можливого функціоналу програми. Розповсюджується під ліцензією MIT. [6]

3.2 Мова програмування Rust

Rust – мультипарадигмова мова програмування розроблена з оглядкою на швидкодію та безпеку, особливо розробку безпечних багато ниткових (threads) програм. Rust синтаксично подібний до C++, та може гарантувати безпеку по пам'яті використовуючи *borrow checker.* Rust досягає безпеки у роботі з пам'яттю без використання збірників сміття [7].

Rust був започаткований Грайдоном Хоаре з Mozilla Research не без участі Девіда Германа, Брендана Іча та інших у 2010-му році.

Вже 4 роки поспіль, починаючи з 2016-го, Rust обирають як найбільш улюблену мову програмування на відкритому опитуванні програмістів Stack Overflow Developer Survey.

Аналогом загальноприйнятих структур даних із загальною назвою «interface» у мові програмування Rust є «trait». «Трейти» не можуть наслідувати інші трейти, їх може реалізовувати лише структура (Struct). В подальшому ми будемо писати чимало трейтів і структур.

РОЗДІЛ 4
РЕАЛІЗАЦІЯ БІБЛІОТЕКИ

Бібліотека повинна бути придатною для застосування сторонніми програмістами, тому архітектура побудована за принципом мінімально необхідних даних.

Ми будемо тестувати алгоритми для двох видів точок: точки на сфері та точки з дискретними відстанями. Це потрібно нам для того, щоб побачити, або не побачити, різницю в отриманих результатах в залежності від типу точок.

4.1 Трейти алгоритмів

4.1.1 Трейт TSPSolver

Для зручності реалізації алгоритмів та їх використання ми введемо абстракцію над алгоритмами розв'язку задачі комівояжера. Сам власне алгоритм буде структурою, що приймає необхідні для його роботи параметри. А абстракція буде трейтом, з єдиним методом solve_tsp. Даний метод прийматиме на вхід лише два аргументи:

1. points – точки у деякому просторі задані масивом. Ми повинні мати можливість порахувати відстань між ними. Тому структура точки повинна мати реалізацію трейту Distance з підрозділу 4.1.2
2. start_point_idx – індекс початкової точки. Якщо для алгоритму це важливо, то ми беремо цю точку як початкову точку роботи нашого алгоритму.

Реалізацію даного трейту можна знайти у Додатку А.1.

4.1.2 Трейт Distance

Даний трейт слугує абстракцією над відстанню між точками або іншими структурами даних. Це дозволяє нам реалізовувати алгоритм не тільки для точок на сфері, площині, а у загальному випадку, якщо є можливість визначити відстань між точками.

Трейт має лише один метод distance_to, що приймає на вхід один аргумент point, що має такий же тип, що й структура, яка його реалізовує. На вихід ми отримуємо відстань у вигляді числа з плаваючою точкою.

Реалізація цього трейту мовою програмування Rust знаходиться у Додатку А.2.

4.2 Стуктура точки у геометричному просторі з дискретними відстанями

У підрозділі 1.5 ми розглянули точки у дискретному просторі. Кожній точці ми поставили у відповідність деяке натуральне число. Це було зроблено для спрощення моделювання даного простору.

Наша структура міститиме єдине поле identifier, що берегтиме відповідне точці натуральне число.

Також ця структура має реалізацію трейту Distance. Він працює за принципами вказаними у підрозділі 1.5. Код структури можна знайти у Додатку А.13.

4.3 Структура точки на сфері

Дана структура існує для зберігання широти та довготи точки на сфері. Ми будемо її використовувати у наших початкових даних для алгоритмів наближеного розв’язку задачі комівояжера.

Також ця структура має реалізацію трейту Distance, адже це нам потрібно для можливості подати вектор точок на сфері у реалізовані алгоритми. Код знаходиться у Додатку А.3.

4.3.1 Обчислення відстані між точками на сфері

Для обрахунку відстані використовується формула гаверсинуса.

Формула гаверсинуса [9] – визначає найменшу відстань між двома точками на сфері по їх широті і довготі. Неймовірно важлива річ у навігації. Вперше формулу опублікував у 1805-у році Джеймс Ендрю.

Формула має вигляд:

$$d = 2r\ arcsin\ (\sqrt{sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) + cos\ (\varphi_1) cos\ (\varphi_2) sin^2 (\frac{\lambda_2 - \lambda_1}{2})})$$

Де

φ_1, φ_2 – широта точки 1 і широта точки 2;

λ_1, λ_2 – довгота точки 1 і довгота точки 2:

r – радіус сфери.

Реалізація знаходиться у Додатку А.4.

4.4 Генерація тестових даних

Для тестування алгоритмів нам необхідно згенерувати тестові дані. У цьому розділі ми розглянемо застосовані методи та використані бібліотеки для генерації наших даних. Тестові дані будуть у вигляді масиву (вектору) точок, що реалізують трейт Distance.

4.4.1 Бібліотека rand

Бібліотека rand у мові програмування Rust слугує для генерації випадкових даних, і чисел, і рядків, і масивів. Ми будемо користуватись нею для генерації випадкового масиву координат на сфері.

4.4.2 Генерація випадкової точки на сфері

Генеруємо випадкову точку на сфері подаючи довготу як випадкову точку [-π / 2; π / 2] та широту у межах [-π; π]. Точка є екземпляром структури SpherePoint. Код генерації точки наведено у Додатку А.5.

4.4.3 Генерація випадкової точки у просторі з дискретними відстанями

Ми згенеруємо випадкову точку у вигляді структури DiscretePoint згаданої у підрозділі 4.2, згенерувавши випадковий identifier.

Код генерації даної точки можна знайти у додатку А.14.

4.4.4 Генерація масиву випадкових точок на сфері

Для подальшої роботи нам необхідно отримати масив фіксованої довжини з випадковими точками на сфері. У цьому розділі ми розглянемо функцію генерації масиву наперед заданої фіксованої довжини заповненого випадковими точками на сфері, які створені функцією з минулого підрозділу. Для ознайомлення з кодом генерації масиву дивіться у Додаток А.6.

4.4.5 Генерація масиву випадкових точок з дискретними відстанями між ними

Щоб мати можливість досліджувати алгоритми у просторі з дискретними відстанями нам необхідно згенерувати відповідні точки. Користуючись підрозділом 4.4.3 ми створюємо випадкову точку і додаємо її у масив. Реалізація даного підходу знаходиться у Додатку А.15.

4.5 Реалізація алгоритмів наближеного розв'язку задачі комівояжера

Систему було реалізовано мовою Rust за допомогою текстового редактора Visual Studio Code з використанням бібліотек, які не розв'язують поставлену задачу, а тільки допомагають форматувати дані з придатного для людини вигляду у оптимізовані під алгоритми структури та навпаки.

4.5.1 Мурашиний алгоритм

Проект реалізовується мовою програмування Rust. Код реалізовує алгоритм з підрозділу 2.1 у більш детальному вигляді. Також для кращого розуміння того, що відбувається у алгоритмах, весь псевдо-код реалізовано мовою програмування Rust.

4.5.1.1 Структура AntAlgTSPSolver

Згідно із згаданим в підрозділі 4.1.1 трейтом TSPSolver наш алгоритм повинен подаватись у вигляді структури, яка реалізує згаданий трейт.

Введемо структуру AntAlgTSPSolver, вона слугуватиме для збереження параметрів розв'язувача згаданих у підрозділі 2.1, а саме – α, β, p. А також ми будемо берегти iters_count змінну, для того щоб знати кількість ітерацій для виконання. ant_capacity – це кількість феромонів у кожній мурасі. Initial_pheromones_val – кількість початкових феромонів для кожного шляху.

Для забезпечення незмінності параметрів поля структури будуть приватними, а сама структура створюватиметься статичним методом new.

Код реалізації структури знаходиться у Додатку А.7.

4.5.1.2 Реалізація трейту TSPSolver для структури AntAlgTSPSolver

Для початку нам необхідно обчислити відстані між точками і запам'ятати їх. Це буде нашим першим кроком алгоритму.

Після цього ми оголосимо матрицю феромонів, де a[b][c] означатиме тількість феромонів на шляху із b в c.

Так як наш алгоритм повинен видати результат у вигляді вектора, ми оголосимо змінну у яку зберігатимемо найменший знайдений за роботу алгоритму шлях.

Далі ми оголосимо цикл за одну ітерацію якого буде імітуватись рух однією мурахи між точками. Ітерація складатиметься із таких етапів:

1. Зафіксуємо змінну поточної позиції як початкову точку.
2. Оголосимо змінну відвіданих точок у вигляді хеш-таблиці. Це дозволить нам економити час для випадку великої кількості точок. Адже дозволить нам за константний час перевірити, чи імітація мурахи відвідувала точку.
3. На забудьбо оголосити змінну для збереження поточного шляху. Це необхідно нам, щоб оновити дані про найкоротший знайдений шлях у разі необхідності.
4. Далі ми задаємо цикл, який працює доти, доки ми не відвідали усі точки.
   a. Додамо у масив відвіданих точок поточну точку.
   b. Рахуємо знаменник для формули (1) з підрозділу 2.1. Це необхідно для економії обчислювальних ресурсів.
   c. Далі пробуємо перейти у наступну точку генеруючи випадкове число з [0, 1] та порівнюючи його із значенням формули (1). Якщо випадкове число менше, обираємо цю точку як наступну точку мурахи.
5. Оновлюємо матрицю феромонів згідно із формулою (2) з підрозділу 2.1.
6. Якщо отриманий шлях коротший за той, що бережеться у змінній з кроку 3, то запам’ятовуємо цей шлях у змінну із кроку 3.
7. Випаровуємо феромони з матриці феромонів.

Результатом алгоритму є змінна із кроку 3.

Кінець алгоритму.

Згаданий вище алгоритм реалізовано мовою Rust. Код знаходиться у Додатку А.8.

4.5.2 Генетичний алгоритм

Для початку нам необхідно обчислити відстані між точками і запам’ятати їх. Це буде нашим першим кроком алгоритму.

Далі ми ініціалізуємо першу популяцію.

Оголошується цикл, який працює задану кількість ітерацій і складається з таких дій:

4. Спарюємо особ з популяції з наперед заданим шансом.
5. Сортуємо популяцію по «якості».
6. Позбуваємось наперед задану кількість хвосту популяції і заповнюємо його новими елементами, які згенеровані випадковим чином.

В останньому поколінні знаходимо найкращого індивіда і виводимо відповідь.

Реалізація алгоритму знаходиться у Додатку А.9.

4.5.2.1 Функція кросоверу популяції

Етап 1 з циклу підрозділу 4.4.2 вимагає наявності деякої функції спарювання популяції. У даному підрозділі ми її опишемо.

Для початку нам необхідно визначитись з підходом до спарювання популяції. Є два найбільш популярні рішення:

- Спарювати «еліту» з «елітою». Простіше кажучи ми сортуємо популяцію за «якостю» у спадаючому порядку. Далі ділимо масив на N / 2 пар, де N – розмір популяції. І застосовуємо на парах функцію, що утворює нову пару індивидів, назвемо її breed.

- Створювати випадкові пари. Беремо випадкового індивіда, ще одного, так і утворилась пара. На цій парі застосовуємо функцію breed для утворення нової популяції.

У даній роботі використано другий підхід. Підставою для цього була публікація по оператору кросоверу CX2 [5], у якій застосовувався саме цей метод утворення пар.

4.5.2.2 Функція «якості»

Для сортування популяції по «якості» нам необхідно мати деяку функцію «якості» індивіда. Зазвичай функція «якості» у ГА описує деякий метод для знаходження числового значення індивіда. Зазвичай чим більше значення, тим більш «якісним» вважається індивід.

Наприклад. Нехай у нас буде популяція випускників факультету комп'ютерних наук та кібернетики. Тоді можна задати функцію «якості» як функцію, що дає значення росту випускника.

У нашій задачі, задачі реалізації ГА для ЗК, теж необхідно мати функцію «якості». Стандартним підходом є функція, що підраховує розмір шляху, який позначає індивід. Так як нам необхідно, щоб більше значення «якості» відповідало «кращому» індивіду, ми будемо користуватись оберненим до розміру шляху значенням. Тобто «якість» (q) індивіда у генетичному алгоритмі для задачі комівояжера буде обчислюватись за формулою:

$$q = \frac{1}{l}$$

l – довжина шляху, який позначає індивід.

РОЗДІЛ 5
ДОСЛІДЖЕННЯ ШВИДКОДІЇ АЛГОРИТМІВ

Реалізувавши ГА і МА ми відкрили собі шлях у світ досліджень алгоритмів.

Для зменшення розмірності даних ми зафіксуємо деякі значення.

N – кількість точок буде обмежена 3-а значеннями: 10, 100, 500.

S – початкова точка ЗК матиме значення 0.

Також дані матимуть значення *time_elapsed*, що відповідає за час, який був затрачений на виконання алгоритму. Одиниця виміру – мілі секунди.

Наші тестові функції прийматимуть на вході згенеровані масиви масивів точок для тестування. Під час виконання будуть записуватись дані у файли.

Для зображень отриманих даних у вигляді графіків було застосовано мову програмування Python та її бібліотеки pandas, numpy, matplotlib. Код доступний у Додатку А.12.

5.1 Дослідження ГА

Наша реалізація ГА має 5 параметрів:

- population_size – розмір популяції;
- gens_count – кількість поколінь;
- crossover_rate – шанс утворення нової пари із двох індивідів покоління;
- mut_rate – шанс мутації індивіда;
- dying_rate – відсоток «найслабкіших» індивідів, що будуть замінені новими індивідами, згенерованими випадковим чином.

Щоб не досліджувати 5-и вимірний простір зафіксуємо деякі параметри.

1. Фіксуємо population_size на значенні 100.
2. Фіксуємо dying_rate на значенні 0.2.
3. Фіксуємо mut_rate на значенні 0.1.
4. Фіксуємо crossover_rate на значенні 0.8.

Для кожного значення N матимемо окрему таблицю. В результаті у вихідних даних залишиться 3 колонки: *gens_count, path_length*, *time_elapsed.*

gens_count обмежимо значеннями [10, 1000] з кроком 10.

Для кожного N ми виконуємо такі дії:

1. Генеруємо N точок нашого простору.
2. Відкриваємо файл для запису даних.
3. Записуємо у файл заголовки даних.
4. Задаємо цикл, який працює по можливим значенням gens_count.
    a. Ініціалізуємо таймер.
    b. Запускаємо ГА на наших точках і запам’ятовуємо результат.
    c. Зупиняємо таймер.
    d. Записуємо у файл рядок виду (кількість поколінь, довжина отриманого з ГА шляху, затрачений час у мілі секундах)

Алгоритм тестування на даних реалізовано мовою програмування Rust і показано у Додатку А.10. Він записує дані у файли вигляду «ga_test_{N}.csv».

5.1.1 Результати дослідження ГА з точками на сфері

Для N = 10 отримуємо графік зображений на рис 8.2:

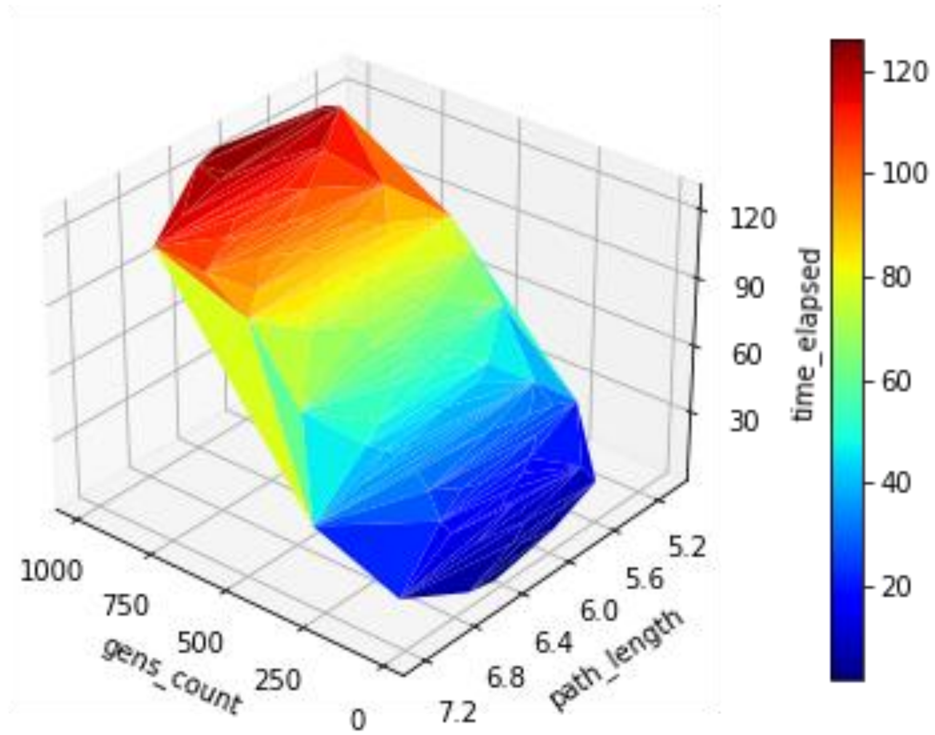


Рисунок 5.1 Результат досліджень ГА для N = 10

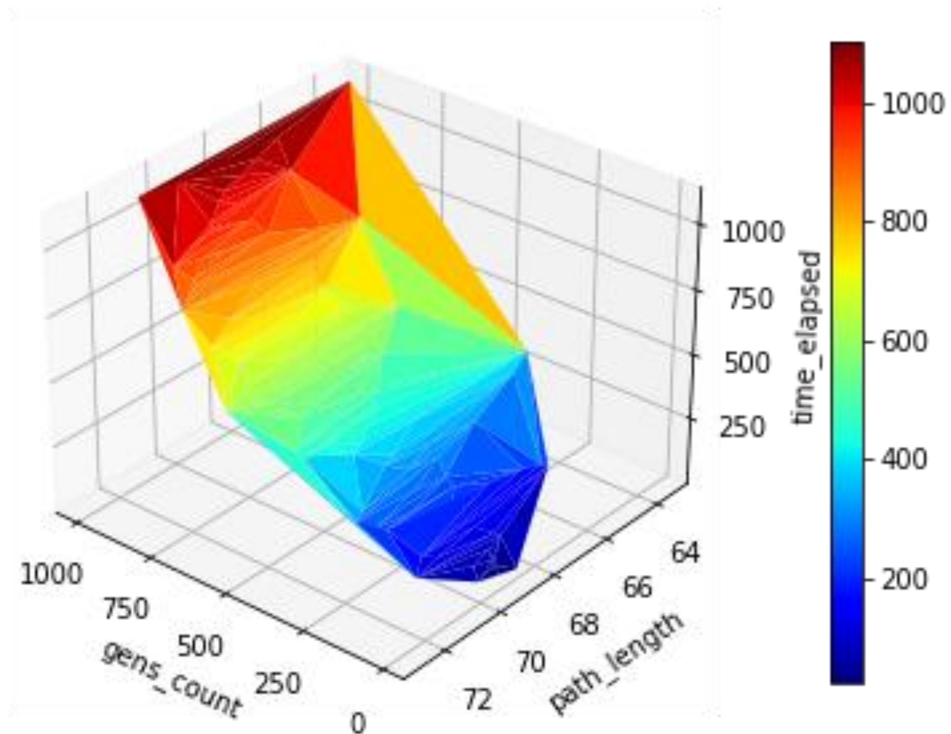


Рисунок 5.2 Результат досліджень ГА для N = 100

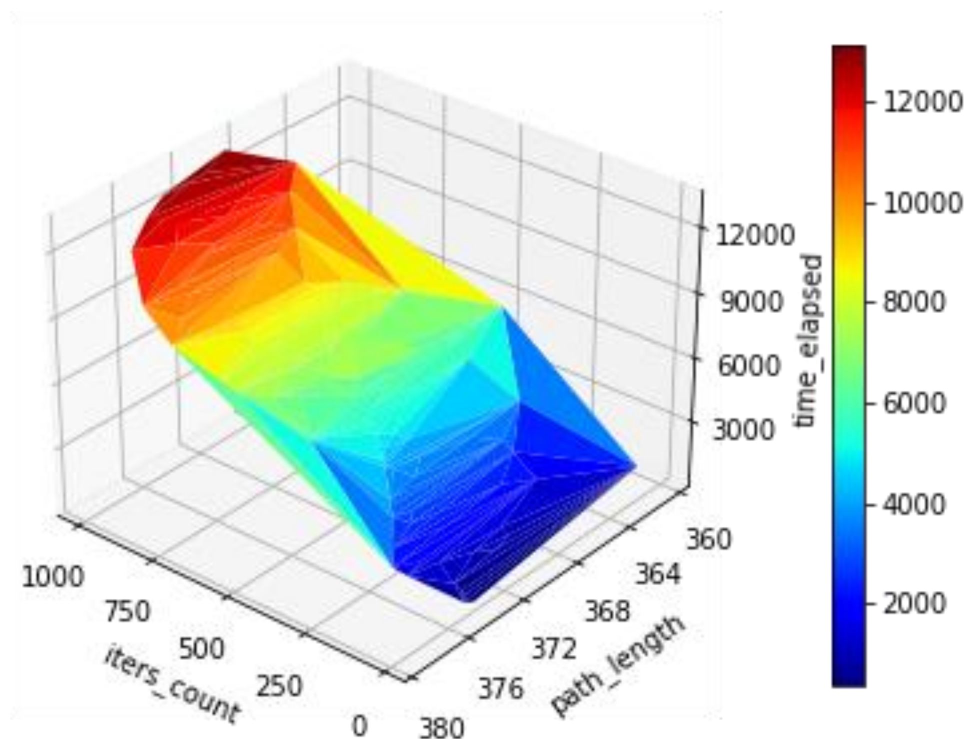


Рисунок 5.3 Результат досліджень ГА для N = 500

На усіх дослідженнях чітко видно залежність між кількістю поколінь та затраченим часом. Тож ми можемо обрати один зріз для подальшого дослідження. Так як в подальшому ми хочемо порівняти ефективність мурашиного алгоритму та генетичного алгоритму, то візьмемо зріз (time_elapsed, path_length). Графіки зображені на рис 5.4, рис 5.5 та рис 5.6:

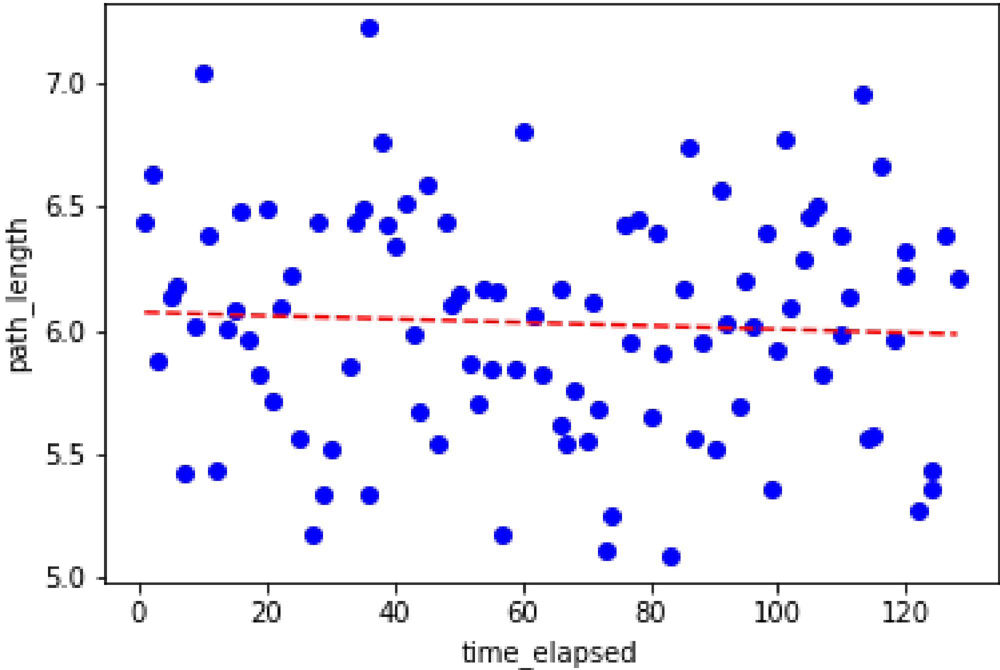


Рисунок 5.4 Результат досліджень ГА для N = 10

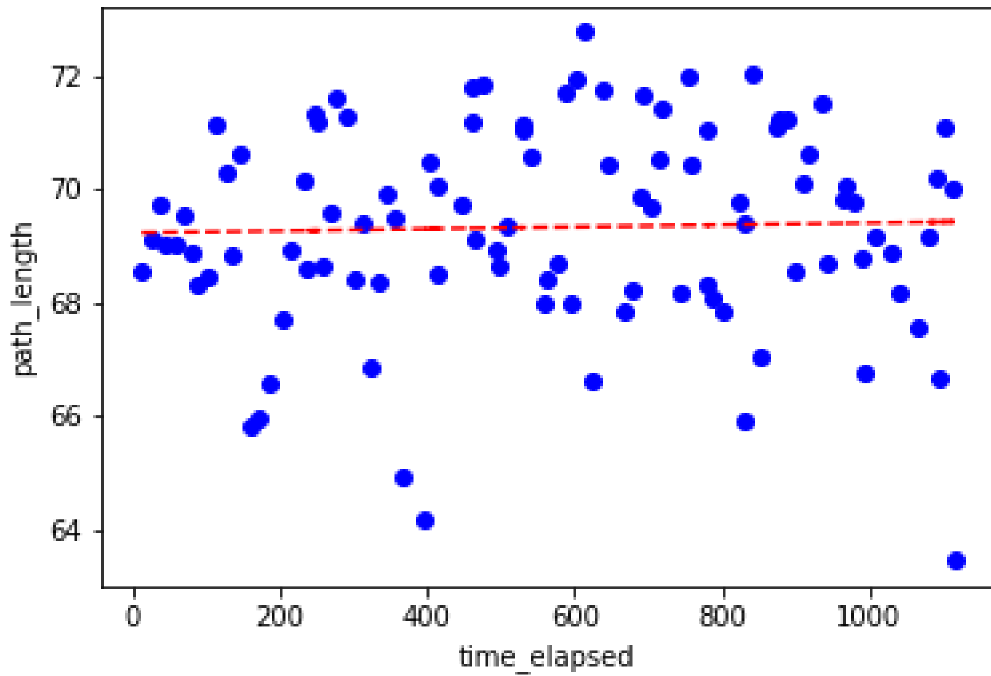


Рисунок 5.5 Результат досліджень ГА для N = 100

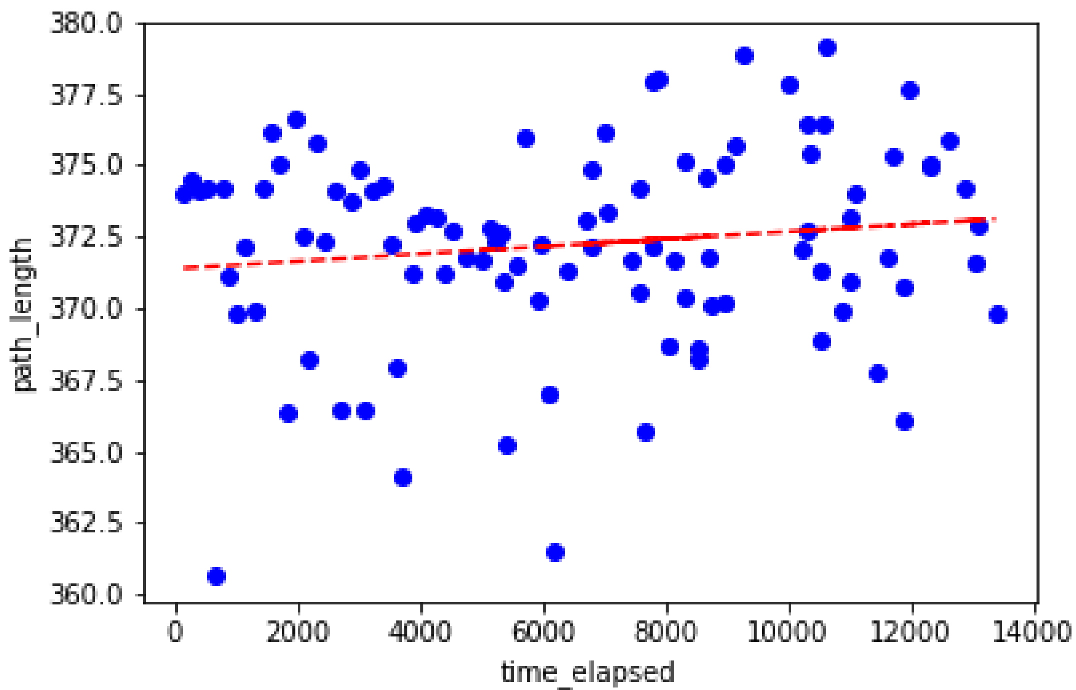


Рисунок 5.6 Результат досліджень ГА для N = 500

5.1.2 Результати досліджень ГА з дискретними відстанями

Також проведені дослідження щодо придатності розробленого генетичного алгоритму для точок з дискретними відстанями. На рисунках від

5.7 до 5.12 можна побачити графіки, візуалізацію даних отриманих під час тестування алгоритму. Самі дані у форматі csv доступні за посиланням у Додатку Б.

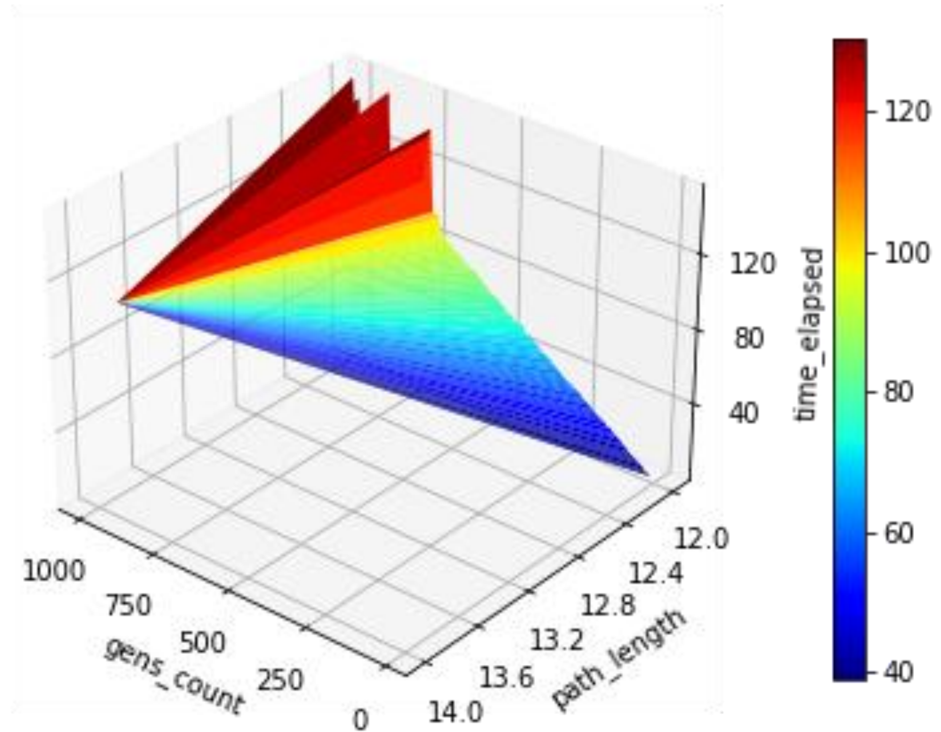


Рисунок 5.7 Результат досліджень ГА для N = 10

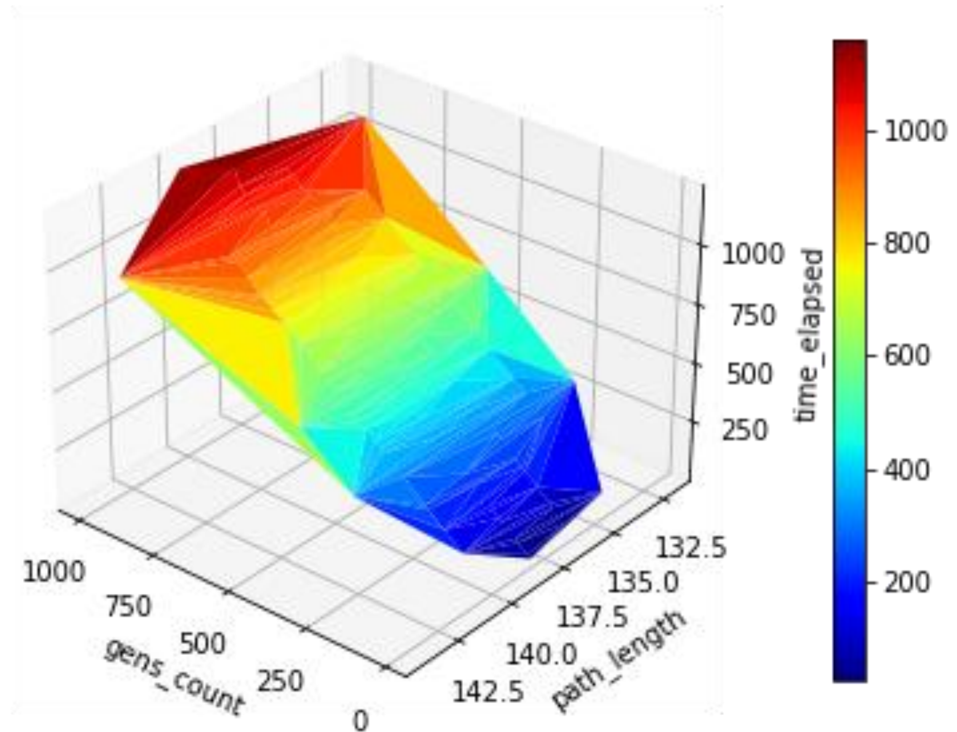


Рисунок 5.8 Результат досліджень ГА для N = 100

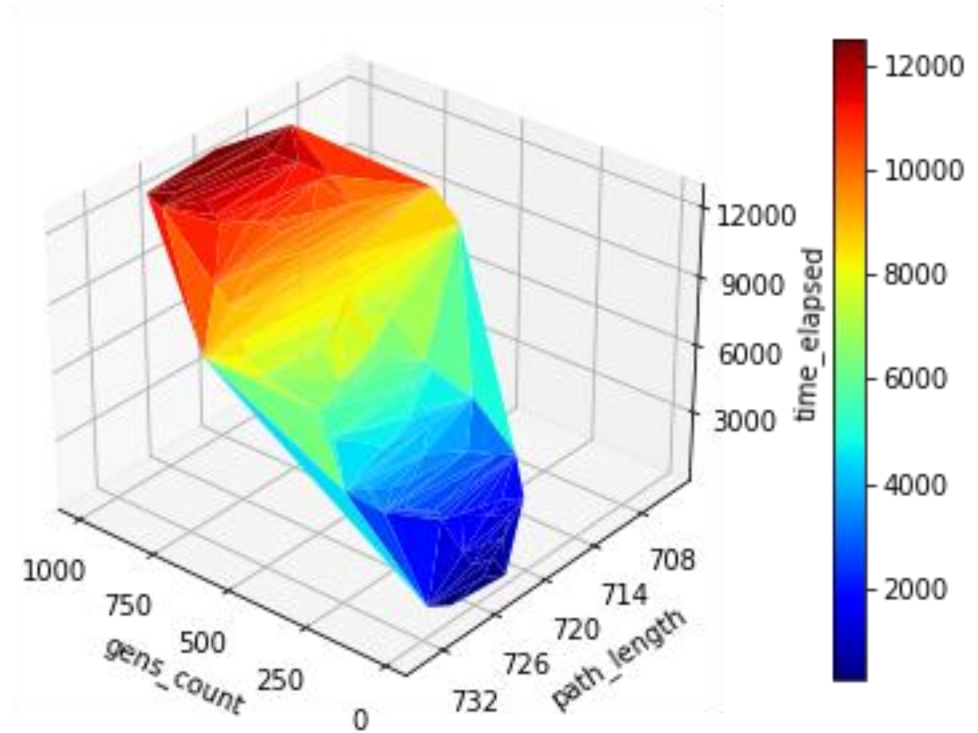


Рисунок 5.9 Результат досліджень ГА для N = 500

Як і у минулому випадку, 3d графіки складні для розуміння, тому далі, по тих же причинах, проекції (time_elapsed, path_length).

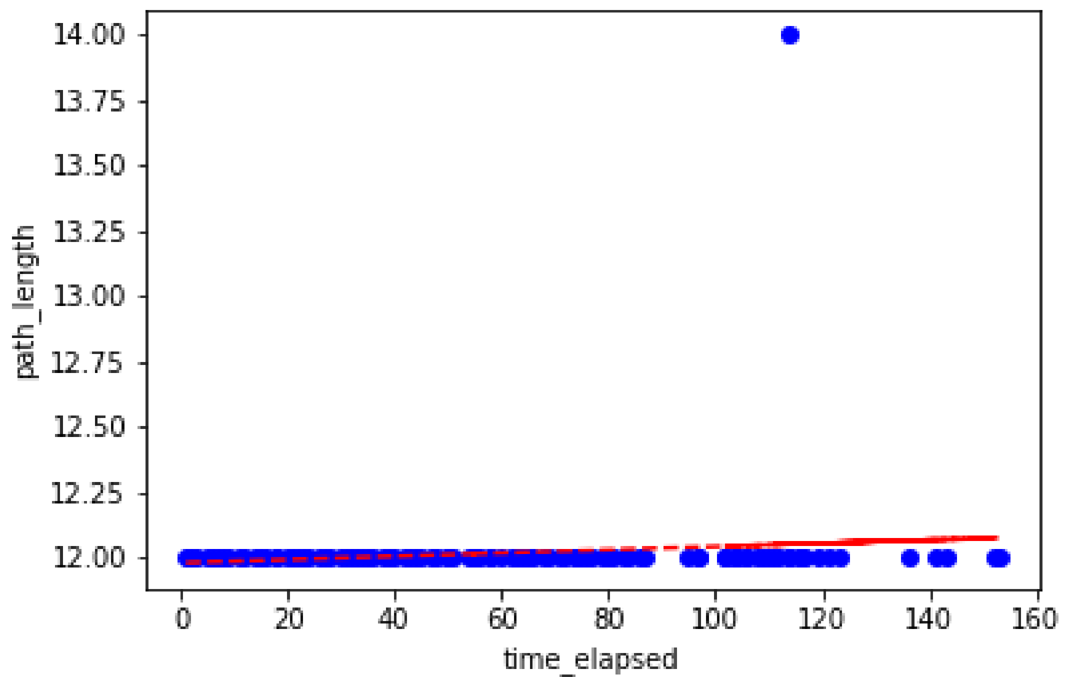


Рисунок 5.10 Результат досліджень ГА для N = 10

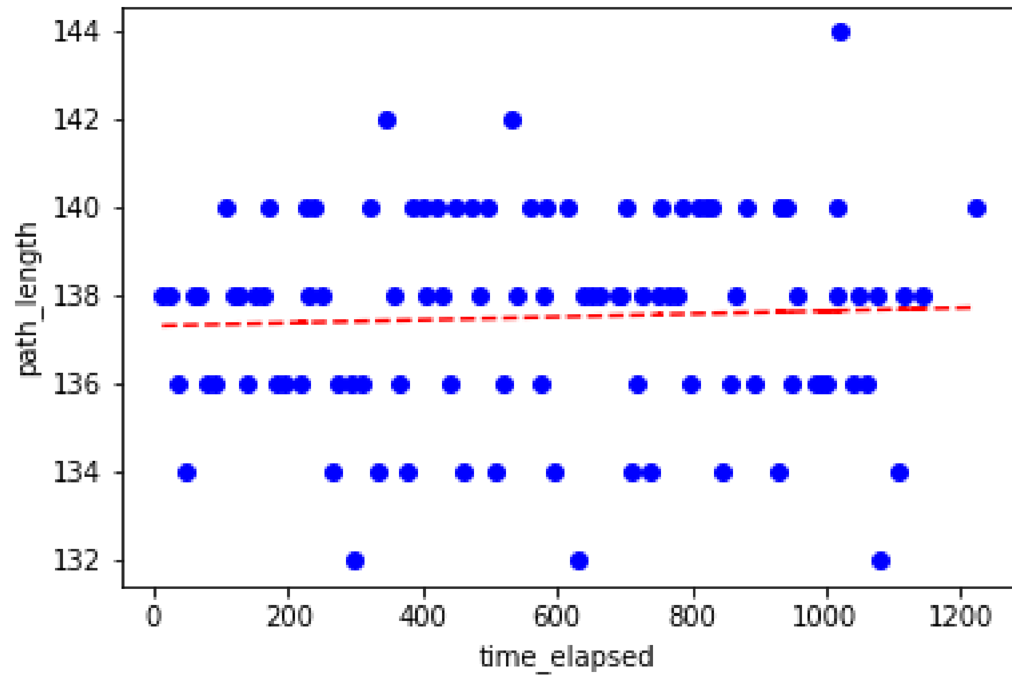


Рисунок 5.11 Результат досліджень ГА для N = 100

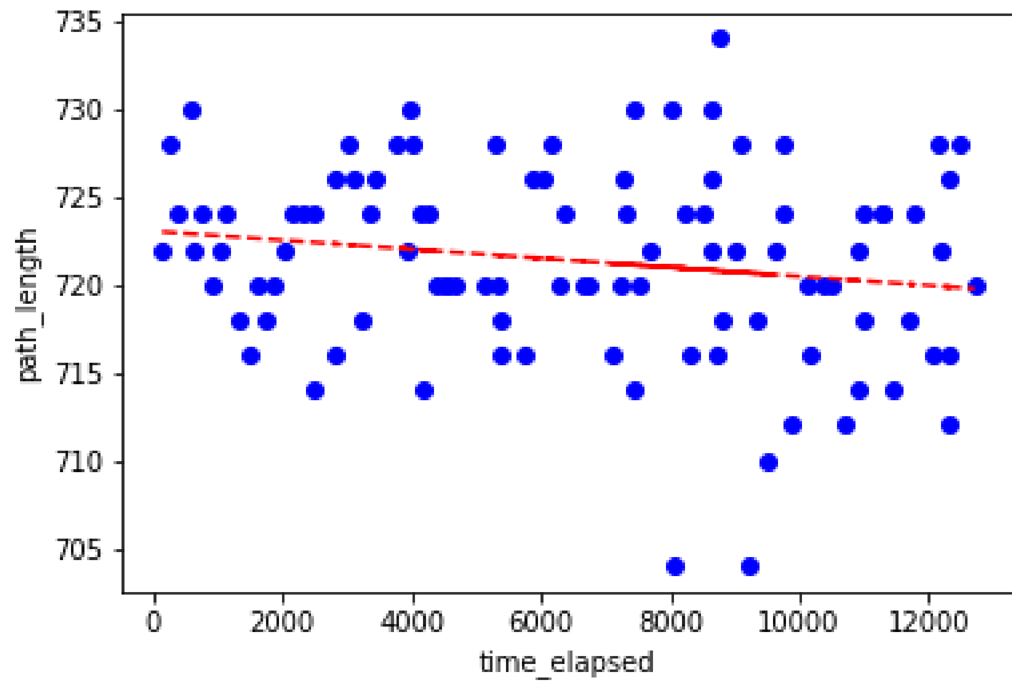


Рисунок 5.12 Результат досліджень ГА для N = 500

5.1.3 Висновки досліджень ГА

Результати виконання ГА з кросовером CX2 не задовольняють задану задачу ні для точок на сфері, ні для точок з дискретними відстанями. По графіках видно, що кращими результати не стають і це скоріше випадкові маршрути, аніж якась послідовність маршрутів яка прямує до оптимального. Можливо CX2 оператор був неправильно реалізований, та це потребує додаткової перевірки. На це вказує непродуктивна робота алгоритму для двох зовсім різних просторів.

5.2 Дослідження МА

Наша реалізація МА має 6 параметрів:

- alfa – коефіцієнт α з підрозділу 2.1;
- beta - коефіцієнт β з підрозділу 2.1;
- ro – коефіцієнт ρ з підрозділу 2.1;
- iters_count – кількість ітерацій алгоритму;
- ant_capacity – «місткість» мурахи;
- initial_pheromones_val – початкова кількість феромонів на ділянці з точки *a* в точку *b*.

Для зменшення розмірності вихідних даних нам просто необхідно зафіксувати деякі параметри.

1. Фіксуємо alfa як 0.5;
2. Фіксуємо beta як 0.5;
3. Фіксуємо ro зі значенням 0.1;
4. Фіксуємо ant_capacity на значенні 10;
5. Фіксуємо initial_pheromones_val як 1;

Для кожного значення N матимемо окрему таблицю. В результаті у вихідних даних залишиться 3 колонки: *iters_count, path_length*, *time_elapsed.*

iters_count обмежимо значеннями [10, 1000] з кроком 10.

Для кожного N ми виконуємо такі дії:

1. Генеруємо N точок нашого простору.
2. Відкриваємо файл для запису даних.
3. Записуємо у файл заголовки даних.
4. Задаємо цикл, який працює по можливим значенням iters_count.
   a. Ініціалізуємо таймер.
   b. Запускаємо МА на наших точках і запам'ятовуємо результат.
   c. Зупиняємо таймер.
   d. Записуємо у файл рядок виду (кількість ітерацій, довжина отриманого з ГА шляху, затрачений час у мілісекундах)

Алгоритм тестування на даних реалізовано мовою програмування Rust і показано у Додатку А.11. Він записує дані у файли вигляду «ma_test_{N}.csv».

Результати дослідження МА з точками на сфері

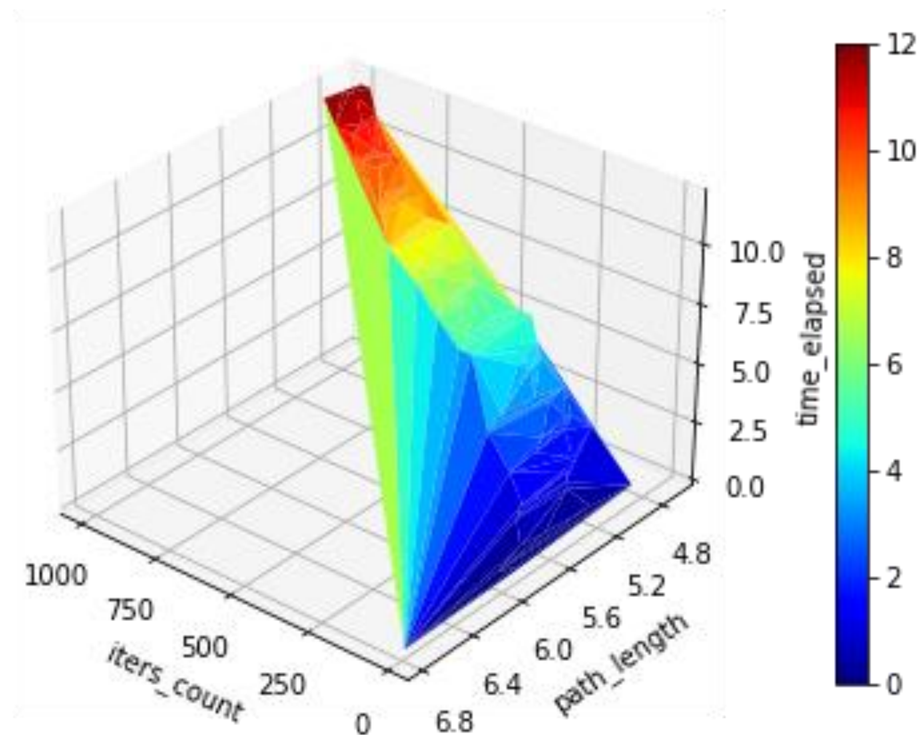


Рисунок 5.13 Результат досліджень МА для N = 10

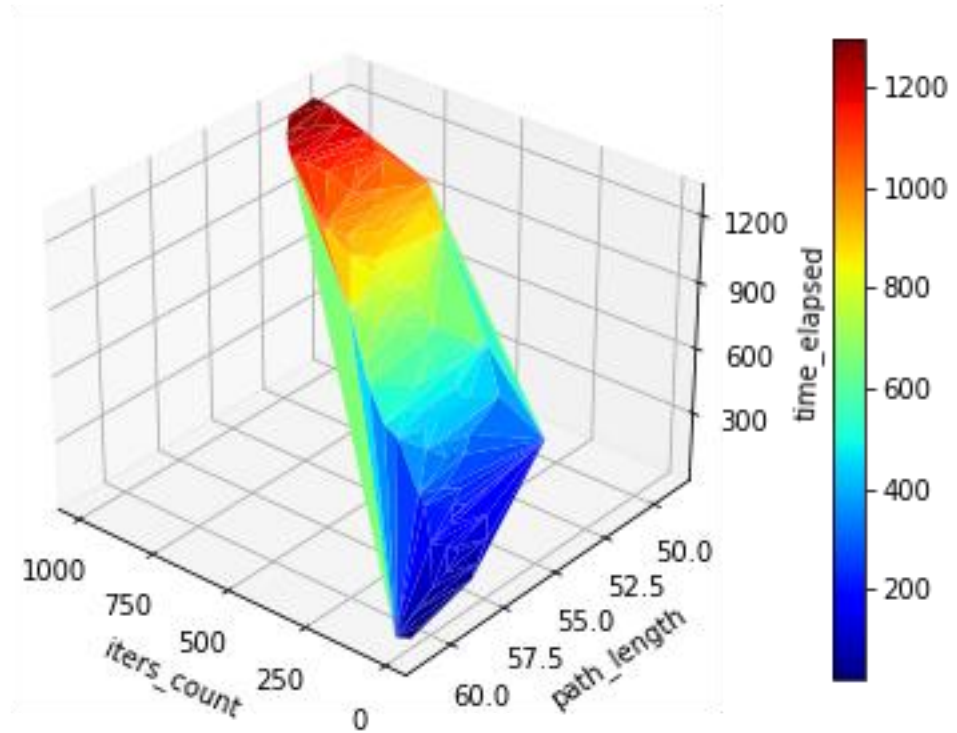


Рисунок 5.14 Результат досліджень МА для N = 100

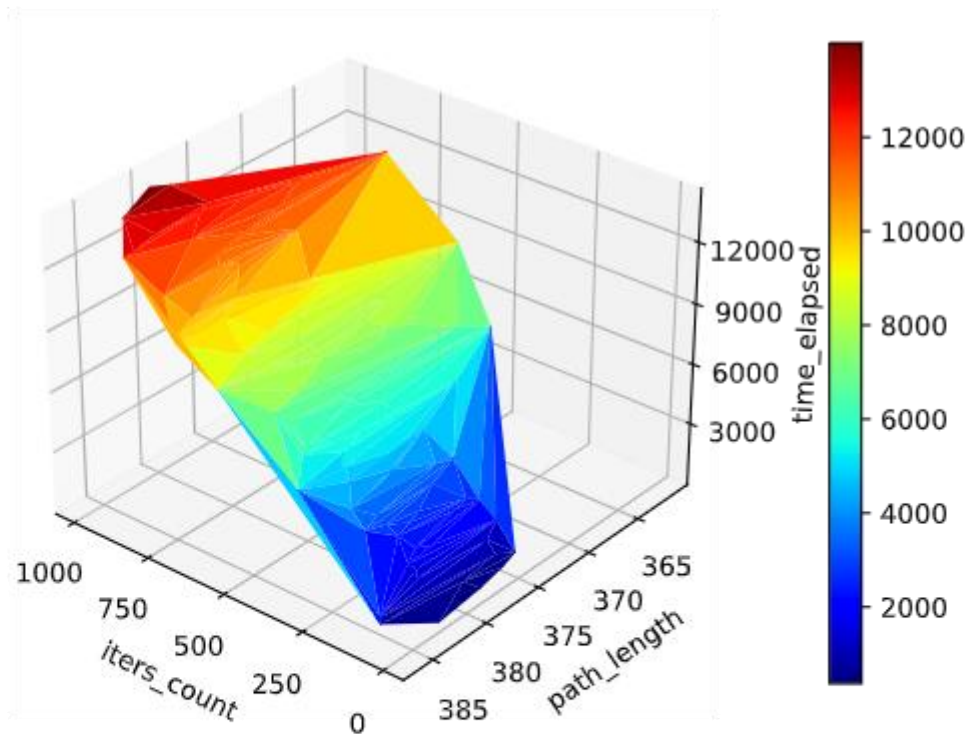


Рисунок 5.15 Результат досліджень МА для N = 500

Як і у минулому підрозділі. На усіх дослідженнях чітко видно залежність між кількістю поколінь та затраченим часом. Тож ми можемо

обрати один зріз для подальшого дослідження. Знову ж таки візьмемо зріз (time_elapsed, path_length). Графіки зображені на рис 8.9, рис 8.10 та рис 8.10:

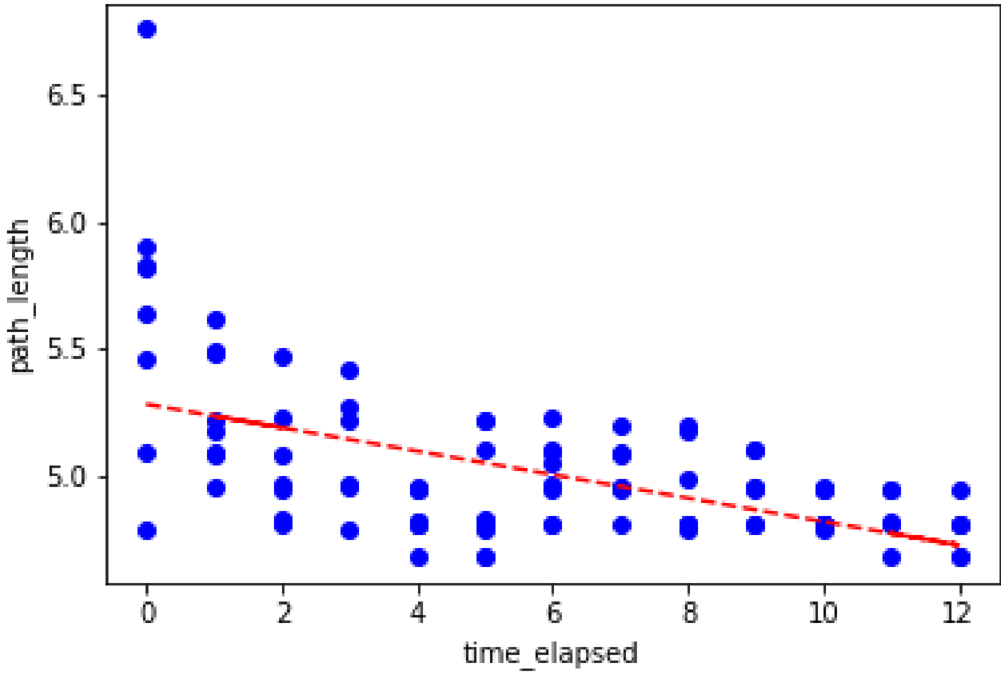


Рисунок 5.16 Результат досліджень МА для N = 10

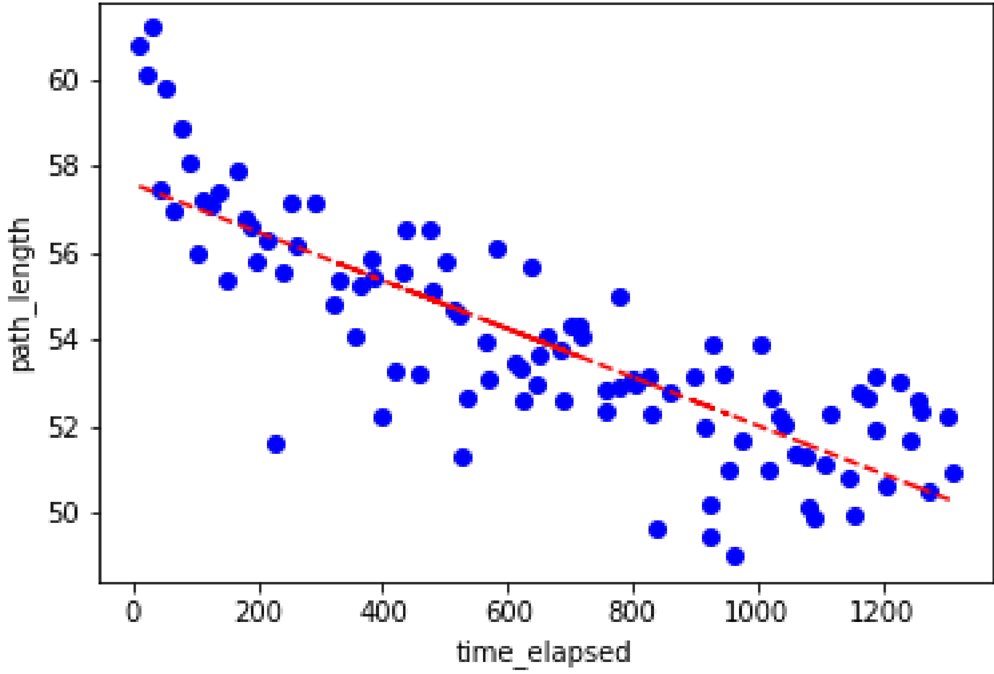


Рисунок 5.17 Результат досліджень МА для N = 100

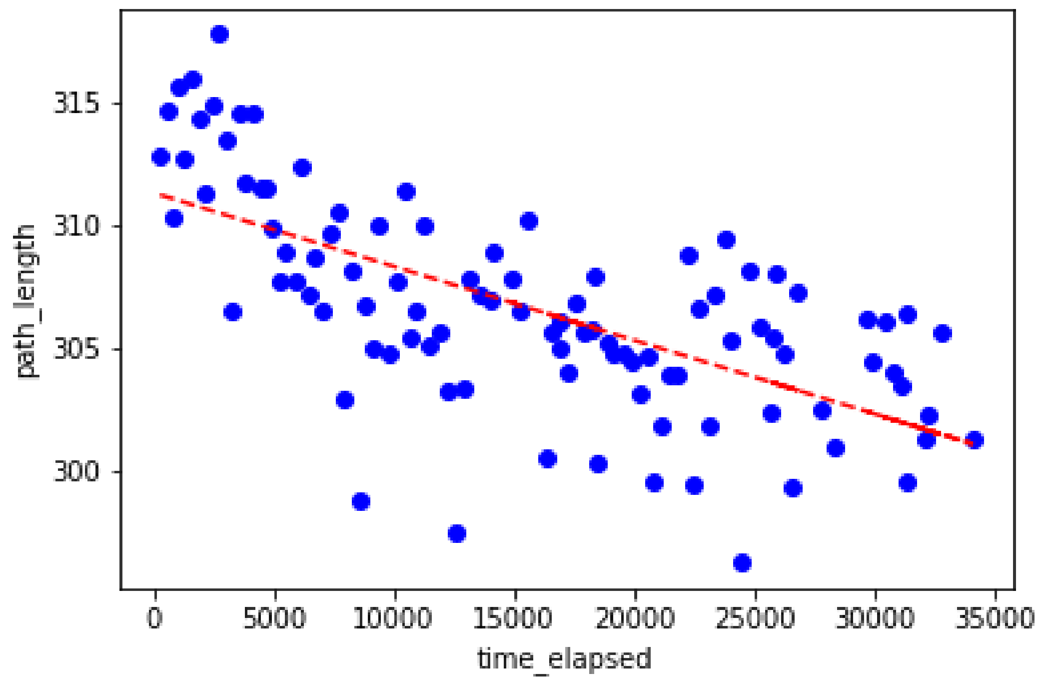


Рисунок 5.18 Результат досліджень МА для N = 500

5.2.2 Результати досліджень МА з точками з дискретними відстанями

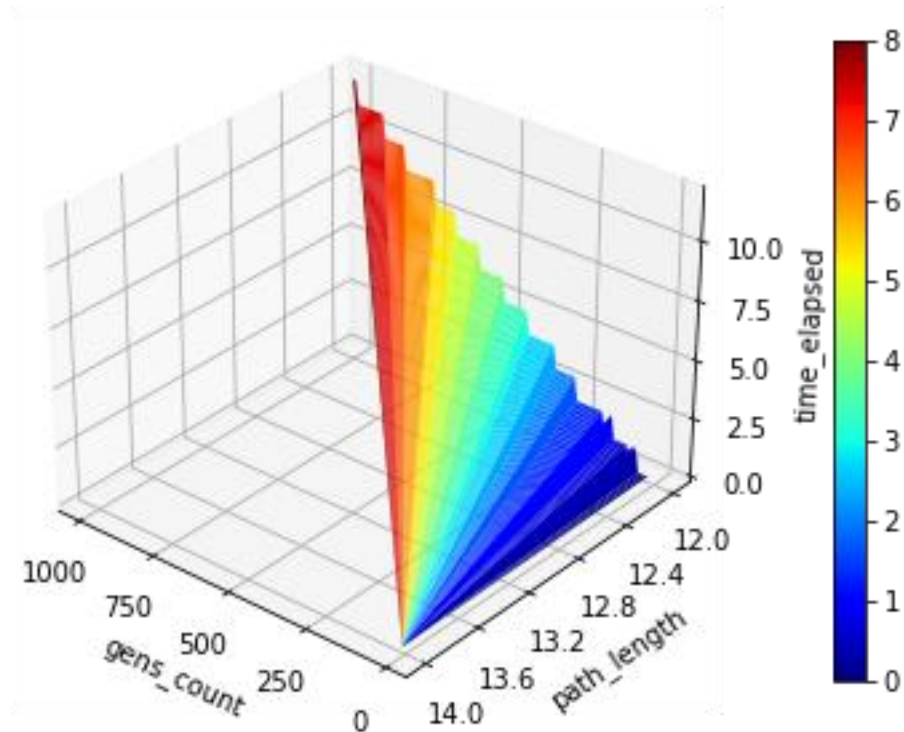


Рисунок 5.19 Результат досліджень МА для N = 10

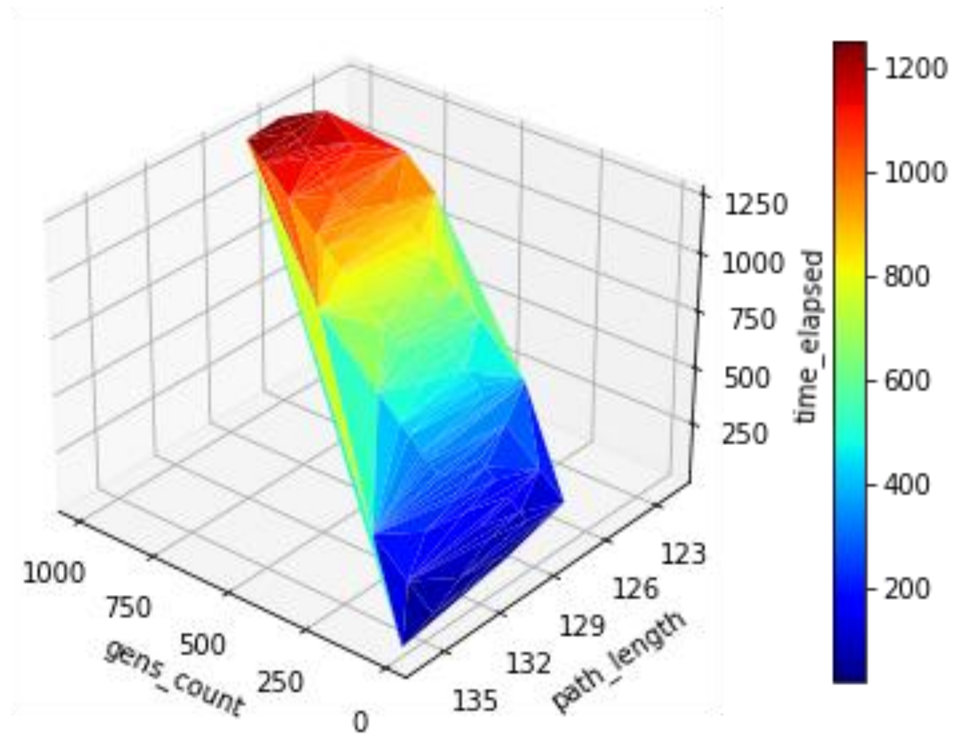


Рисунок 5.20 Результат досліджень MA для N = 100

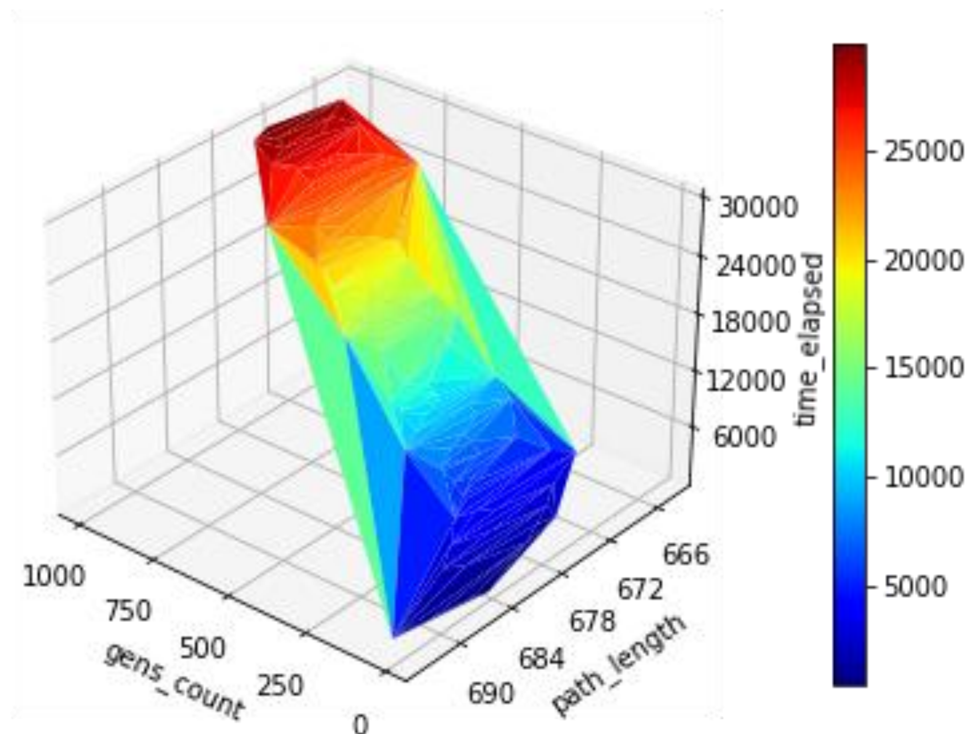


Рисунок 5.21 Результат досліджень MA для N = 500

По тій же причині, що і у підрозділах 5.2.1 та 5.2.2 ми розглянемо 2d графіки.

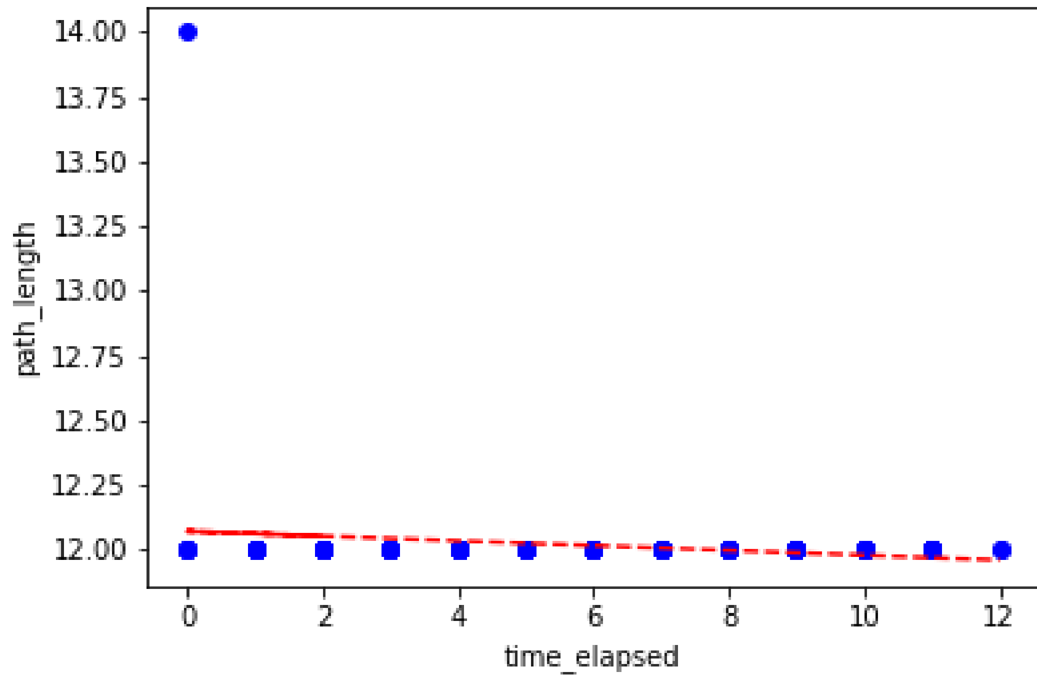


Рисунок 5.22 Результат досліджень МА для N = 10

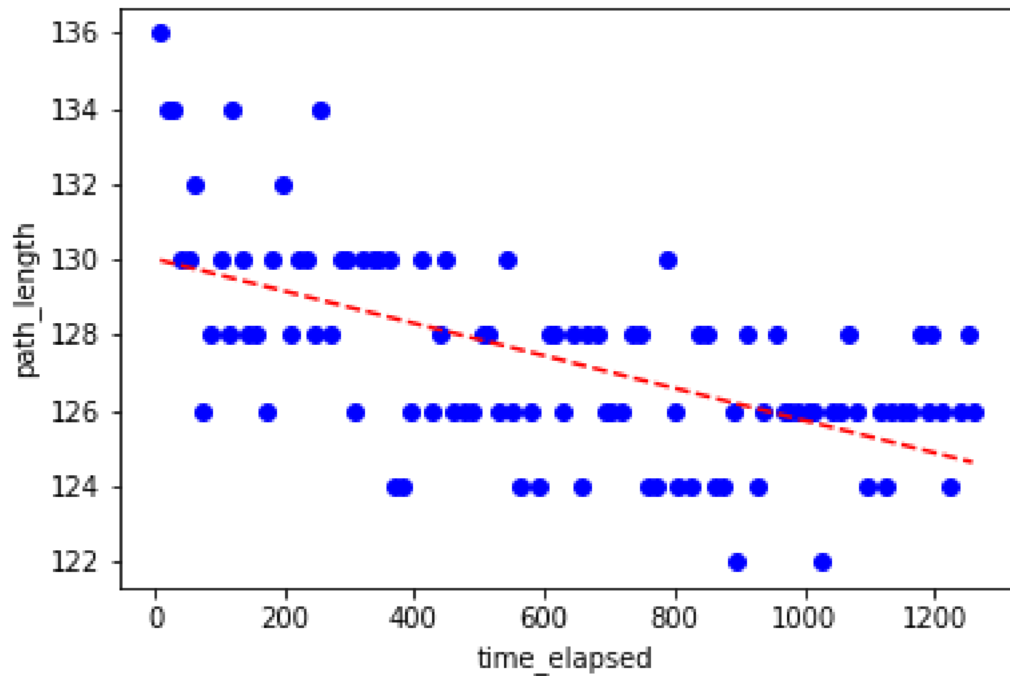


Рисунок 5.23 Результат досліджень МА для N = 100

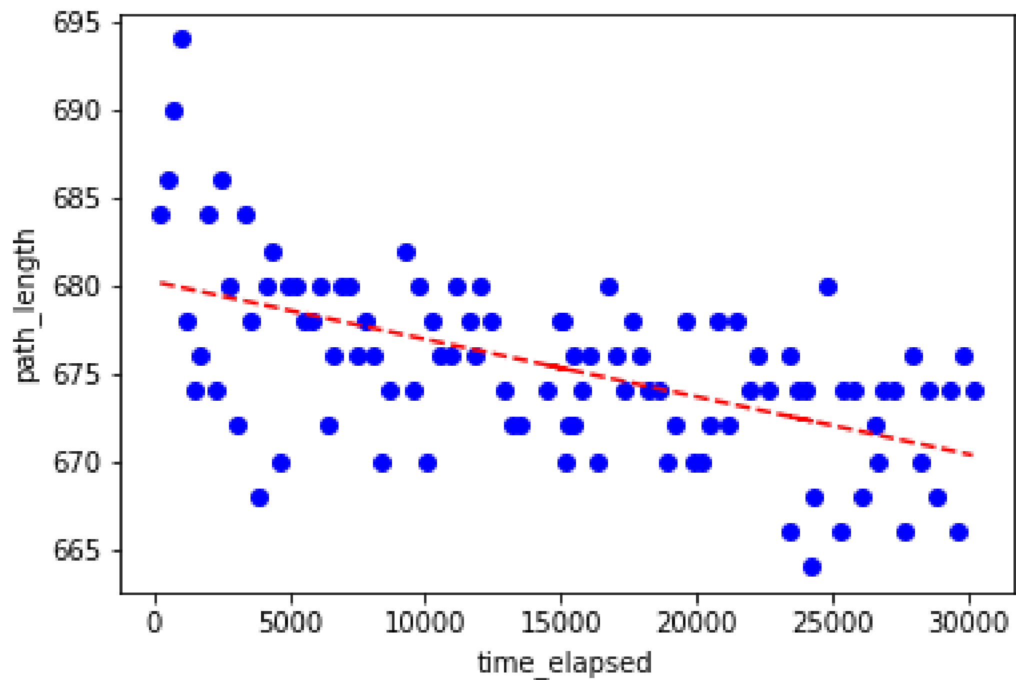


Рисунок 5.24 Результат досліджень МА для N = 500

5.2.3 Висновки досліджень МА

На усіх графіках чітко видно залежність між витраченим часом і довжиною знайденого шляху. Немає різниці простір з точками на сфері, чи відстані між точками дискретні, результат добре видно. Природа цієї залежності, чи лінійна вона, з одного лиш графіку зрозуміти складно. Та точно можна зробити висновок, що з кожною ітерацією розв’язок покращується.

7. ВИСНОВКИ

Метою кваліфікаційної роботи бакалавра була розробка програмного забезпечення для наближеного розв’язку геометричної задачі комівояжера, використовуючи генетичний алгоритм та мурашиний алгоритм.

У процесі виконання були виконані наступні завдання:

- поглиблені знання мови Rust та опануванні необхідні бібліотеки;
- згенеровані тестові дані для поставленої задачі комівояжера;
- вивчено та досліджено генетичний алгоритм та оператор кросоверу CX2;
- реалізовано генетичний алгоритм мовою програмування Rust;
- вивчено та досліджено мурашиний алгоритм;
- реалізовано мурашиний алгоритм мовою програмування Rust;
- описано використані структури даних;

Створена програмна дозволяє досліджувати розв’язки задачі комівояжера на ефективність, а саме – ГА та МА, залежно від різних параметрів, які використовуються у алгоритмах.

У системі реалізовано пошук розв’язку геометричної задачі комівояжера з дискретними відстанями. Отже, базовий функціонал повністю реалізований.

8. СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Github - rust/rust-lang [Електроний ресурс] // Github: [сайт]. URL: https://github.com/rust-lang/rust
2. Karla L. Hoffman M.P.G.R. Traveling salesman problem. Kluwer Academic Publishers, 2001.
3. Alexander Barvinok S.P.F.D.S.J.A.T. The Geometric Maximum Traveling Salesman Problem. 2003.
4. // Ant colony optimization algorithms: [сайт]. URL: https://en.wikipedia.org/wiki/Ant_colony_optimization_algorithms
5. Abid Hussain Y.S.M.M.N.S., "Operator, Genetic Algorithm for Traveling Salesman Problem with Modified Cycle Crossover," 2017.
6. Visual Studio Code [Електроний ресурс] // Wikipedia: [сайт]. URL: https://en.wikipedia.org/wiki/Visual_Studio_Code
7. Rust (programming language) [Електроний ресурс] URL: https://en.wikipedia.org/wiki/Rust_(programming_language)
8. Carpenter W. One Hundred Proofs that the Earth is Not a Globe. Read Books Ltd., 2017.
9. Haversine formula [Електроний ресурс] URL: https://en.wikipedia.org/wiki/Haversine_formula

9. ДОДАТКИ

ДОДАТОК 1

Код бібліотеки

1.1 Трейт TSPSolver

```
pub trait TSPSolver {
  fn solve_tsp<T: Distance + Clone>(&self, points: &Vec<T>, start_point_idx: usize) -> Vec<T>;
}
```

Рисунок А.1 Оголошення трейту TSPSolver.

Метод solve_tsp приймає на вхід вектор точок, що реалізують трейт Distance. І на вихід подає вектор точок цієї ж структури даних, що отримав на вхід.

1.2 Трейт Distance

```
pub trait Distance {
  fn distance_to(&self, point: &Self) -> f64;
}
```

Рисунок А.2 Оголошення трейту Distance.

Метод distance_to приймає на вхід точку типу Self (власного типу) і видає на вихід число з плаваючою точкою, що займає 64 біта.

1.3 Структура SpherePoint

У даній структурі зберігаємо довготу як lon: f64 та широту як lat: f64 в радіанах. Метод from_degrees слулує як допоміжний. Бо більшість карт задають значення широти і довготи точки у градусах.

```
pub struct SpherePoint {
  // in radians
  lon: f64,
  // in radians
  lat: f64,
}

impl SpherePoint {
  pub fn from_degrees(lon: f64, lat: f64) -> Self {
    SpherePoint {
      lon: degrees_to_radians(degrees: lon),
      lat: degrees_to_radians(degrees: lat),
    }
  }

  pub fn from_radians(lon: f64, lat: f64) -> Self {
    SpherePoint {
      lon,
      lat,
    }
  }
}

#[inline]
fn degrees_to_radians(degrees: f64) -> f64 {
  (degrees * PI) / 180.0
}
```

Рисунок А.3 Оголошення структури SpherePoint разом з методом ініціалізації.

1.4 Трейт Distance для структури SpherePoint

Реалізовує метод distance_to з трейту Distance згаданого у Додатку А.2

```
impl Distance for SpherePoint {
  fn distance_to(&self, point: &Self) -> f64 {
    let d_lat: f64 = point.lat - self.lat;
    let d_lon: f64 = point.lon - self.lon;

    let a: f64 =
      (d_lat / 2.0).sin().powi(2)
      +
      (d_lon / 2.0).sin().powi(2) * self.lat.cos() * point.lat.cos();


    a.sqrt().asin()
  }
}
```

Рисунок А.4 Реалізація трейту Distance для структури SpherePoint за формулою гаверсинуса.

1.5 Генерація випадкової точки

З допомогою бібліотеки rand ми генеруємо випадкову довготу (lon) і широту (lat) та передаємо їх у статичний метод структури SpherePoint from_radians.

```
use std::f64::consts::PI;

use rand::Rng;

use super::sphere_point::SpherePoint;

pub fn random_sphere_point() -> SpherePoint {
  let mut thread_rng: ThreadRng = rand::thread_rng();
  SpherePoint::from_radians(
    lon: thread_rng.gen::<f64>() * PI * 2.0 - PI,
    lat: thread_rng.gen::<f64>() * PI - PI / 2.0,
  )
}
```

Рисунок А.5 Реалізація функції генерації випадкової точки на сфері.

1.6 Генерація масиву випадкових точок

Функція приймає на вхід n типу usize і генерує n випадкових точок повертаючи їх у вигляді масиву (вектору).

```
pub fn random_sphere_points(n: usize) -> Vec<SpherePoint> {
  (0..n): Range<usize>
  .map(|_| random_sphere_point()): impl Iterator<Item = SpherePoint>
  .collect()
}
```

n – кількість точок які ми згенеруємо.

Рисунок А.6 Реалізація функції генерації масиву випадкових точок на сфері.

1.7 Структура AntAlgTSPSolver

Слугує для збереження параметрів МА.

```
pub struct AntAlgTSPSolver {
  alfa: f64,
  beta: f64,
  ro: f64,
  iters_count: usize,
  ant_capacity: f64,
  initial_pheromones_val: f64,
}

impl AntAlgTSPSolver {
  pub fn new(
    alfa: f64,
    beta: f64,
    ro: f64,
    iters_count: usize,
    ant_capacity: f64,
    initial_pheromones_val: f64
  ) -> Self {
    AntAlgTSPSolver {
      alfa,
      beta,
      ro,
      iters_count,
      ant_capacity,
      initial_pheromones_val,
    }
  }
}
```

Рисунок А.7 Реалізація структури AntAlgTSPSolver.

1.8 Реалізація мурашиного алгоритму

Алгоритм реалізовано мовою програмування Rust і він повністю повторює псевдо-код з підрозділу 4.4.1.2. Кроки алгоритму підписані під рисунками наведеними нижче.

```
impl TSPSolver for AntAlgTSPSolver {
  fn solve_tsp<T: Distance + Clone>(&self, points: &Vec<T>, start_point_idx: usize) -> Vec<T> {
    let distances_mat: Vec<Vec<f64>> = (0..points.len()): Range<usize>
        .map(|a: usize|
          (0..points.len()): Range<usize>
          .map(move |b: usize| points[a].distance_to(point: &points[b])): impl Iterator<Item = f64>
          .collect()
        ): impl Iterator<Item = Vec<…>>
        .collect();

    let mut pheromones_mat: Vec<Vec<f64>> = vec![vec![self.initial_pheromones_val; points.len()]; points.len()];

    let mut shortest_path: Vec<usize> = Vec::new();
    let mut shortest_path_length: f64 = f64::MAX;
```

Рисунок А.8 Оголошення матриці відстаней, матриці феромонів і змінних для зберігання даних про найкращий знайдений шлях.

```
let mut rand: ThreadRng = rand::thread_rng();
// each iteration is ant simulation
for _ in 0..self.iters_count {
    let mut current_point: usize = start_point_idx;
    let mut visited_points: HashSet<usize> = HashSet::new();

    let mut current_path: Vec<usize> = Vec::with_capacity(points.len());
    current_path.push(start_point_idx);

    let mut current_path_length: f64 = 0.0;
```

Рисунок А.9 Оголошення і перші команди циклу. Додавання до пройдених точок початкової точки.

```
while visited_points.len() < points.len() {
    visited_points.insert(current_point);

    let mut balance_sum: f64 = 0.0;

    for i: usize in 0..distances_mat.len() {
        if distances_mat[current_point][i].eq(&0.0) || visited_points.contains(&i) {
            continue;
        }

        balance_sum += pheromones_mat[current_point][i].powf(self.alfa)
            * (1.0 / distances_mat[current_point][i]).powf(self.beta);
    }

    for i: usize in 0..points.len() {
        if distances_mat[current_point][i].eq(&0.0) {
            continue;
        }

        let prob_to_move: f64 =
            if visited_points.contains(&i) {
                0.0
            } else {
                pheromones_mat[current_point][i].powf(self.alfa) *
                (1.0 / distances_mat[current_point][i]).powf(self.beta) / balance_sum
            };

        let random_number: f64 = rand.gen();

        if random_number < prob_to_move {
            current_path.push(i);

            current_path_length += distances_mat[current_point][i];
            current_point = i;
            break;
        }
    }
}
```

Рисунок А.10 Тіло циклу подорожі мурахи.

```
current_path.push(start_point_idx);
current_path_length += distances_mat[current_point][start_point_idx];

// leaving pheromones
for i: usize in 0..current_path.len() - 1 {
  pheromones_mat[current_path[i]][current_path[i + 1]] +=
    self.ant_capacity / current_path_length;
}

if current_path_length < shortest_path_length {
  shortest_path = current_path;
  shortest_path_length = current_path_length;
}

// vaporizing pheromones
for i: &Vec<f64> in pheromones_mat.iter() {
  for mut g: &f64 in i.iter() {
    g = &(g * (1.0 - self.ro));
  }
}
}
```

Рисунок А.11 Кінець циклу існування мурахи. Оновлення матриці феромонів і перевірка якості знайденого шляху.

```
  shortest_path.into_iter().map(|index: usize| points[index].clone()).collect()
 }
}
```

Рисунок А.12 Кінець алгоритму. Повернення найкоротшого шляху у тій же структурі даних, що й була на вхід.

1.9 Реалізація генетичного алгоритму

На наступних рисунках можна побачити реалізацію даного алгоритму мовою програмування Rust.

```
impl TSPSolver for GeneticAlgTSPSolver {
  fn solve_tsp<T: Distance + Clone>(&self, points: &Vec<T>, start_point_idx: usize) -> Vec<T> {
    let distances_mat: Vec<Vec<f64>> = (0..points.len()): Range<usize>
      .map(|a: usize|
        (0..points.len()): Range<usize>
        .map(move |b: usize| points[a].distance_to(point: &points[b])): impl Iterator<Item = f64>
        .collect()
      ): impl Iterator<Item = Vec<…>>
      .collect();
```

Рисунок А.13 Початок алгоритму і обчислення матриці відстаней.

```
// initializing first population
let mut population: Vec<Vec<usize>> = generate_population(&points, self.population_size);
```

Рисунок А.14 Генерація популяції за допомогою функції generate_population.

Популяція генерується наступним чином. Генерується вектор послідовного шляху. Далі цей вектор випадковим чином тасується. Повертається з функції *size* таких векторів. Функцію можна побачити на рис 7.9.

```
fn generate_population<T: Distance + Clone>(points: &Vec<T>, size: usize) -> Vec<Vec<usize>> {
  let mut rng: ThreadRng = thread_rng();

  (0..size): Range<usize>
    .map(|_| {
      let mut path: Vec<usize> = (0..points.len()).map(|i: usize| i).collect();
      path.shuffle(&mut rng);

      path
    }): impl Iterator<Item = Vec<…>>
    .collect()
}
```

Рисунок А.15 Функція генерації популяції розміру size.

Далі йде цикл який виконується *self.gens_count* кількість разів. Він має тіло, що відповідає подіям на одному поколінні індивідів. Цикл зображено на рис 7.10.

```
for _ in 0..self.gens_count {
  population = self.breed_population(&population);
  self.mutate_population(&mut population);

  // weak die
  let should_die_count: usize = (self.population_size as f64 * self.dying_rate) as usize;

  self.sort_population(&points, &distances_mat, &mut population);
  let mut should_alive: Vec<Vec<usize>> = population.iter(): Iter<Vec<usize>>
    .take(self.population_size - should_die_count): impl Iterator<Item = &Vec<…>>
    .cloned(): impl Iterator<Item = Vec<…>>
    .collect();
  // newbies arrive
  population = generate_population(&points, size: should_die_count);
  population.append(&mut should_alive);
}
```

Рисунок А.16 Тіло циклу поколінь ГА.

На початку тіла циклу виконується метод, що утворює пари з популяції для генерації наступного покоління. Вона тасує популяцію, бере пари і з шансом *self.crossover_rate* утворює пару. Рис 7.11 зображує дану функцію.

```
fn breed_population(&self, population: &Vec<Vec<usize>>) -> Vec<Vec<usize>> {
    let mut rng: ThreadRng = thread_rng();

    // shuffle population to get random pairs
    let mut shuffled_pop: Vec<Vec<usize>> = population.clone();
    shuffled_pop.shuffle(&mut rng);

    let mut result: Vec<Vec<usize>> = Vec::with_capacity(population.len());

    shuffled_pop.chunks(chunk_size: 2): Chunks<Vec<usize>>
        .map(|parents: &[Vec<usize>]| {
            if rng.gen::<f64>() < self.crossover_rate {
                return breed(parent1: &parents[0], parent2: &parents[1]);
            }
            (parents[0].clone(), parents[1].clone())
        }): impl Iterator<Item = (Vec<…>, …)>
        .for_each(|(offspring1: Vec<usize>, offspring2: Vec<usize>)| {
            result.push(offspring1);
            result.push(offspring2);
        });

    result
}
}
```

Рисунок А.17 Функція для утворення «дітей» покоління.

Утворення нової пари відбувається за алгоритмом згаданим у підрозділі 2.2. Також додано модифікацію, *step X*, яка виправляє забутий розробниками алгоритму крайовий випадок, коли перший елемент другої «дитини» є першим елементом першого «батька» на першій же ітерації. Функцію *breed* можна оглянути на рис 7.12, рис 7.13 та рис 7.14.

```
// breed algorithm published on https://www.hindawi.com/journals/cin/2017/7430125/#B9
fn breed(parent1: &Vec<usize>, parent2: &Vec<usize>) -> (Vec<usize>, Vec<usize>) {
  let mut offspring1: Vec<usize> = Vec::with_capacity(parent1.len());
  let mut offspring2: Vec<usize> = Vec::with_capacity(parent1.len());

  let mut bit_index_from_step_4: usize = 0;

  // step 2
  offspring1.push(parent2[0]);

  loop {
    // step 3
    let bit_index_in_parent1: usize = parent1.iter().position(|bit: &usize| *bit == parent2[bit_index_from_step_4]).unwrap();
    let bit_index_from_step_3: usize = parent1.iter().position(|bit: &usize| *bit == parent2[bit_index_in_parent1]).unwrap();
    offspring2.push(parent2[bit_index_from_step_3]);

    // step 5
    if parent2[bit_index_from_step_3] == parent1[0] {
      break;
    }

    // step 4
    bit_index_from_step_4 = parent1.iter().position(|bit: &usize| *bit == parent2[bit_index_from_step_3]).unwrap();
    offspring1.push(parent2[bit_index_from_step_4]);
  }
```

Рисунок А.18 Кроки 2, 3, 4, 5 алгоритму з підрозділу 2.2.

```
  // step X not mentioned in publication for cases like parent1 = [2, 1, 0] and parent2 = [1, 0, 2]
  let mut forgotted_bits_1: Vec<usize> = offspring1.iter().filter(|bit: &&usize| !offspring2.contains(bit)).copied().collect();
  let mut forgotted_bits_2: Vec<usize> = offspring2.iter().filter(|bit: &&usize| !offspring1.contains(bit)).copied().collect();
  offspring1.append(&mut forgotted_bits_2);
  offspring2.append(&mut forgotted_bits_1);
```

Рисунок А.19 Кроки X для виправлення алгоритму з підрозділу 2.2.

```
  // step 6
  if offspring1.len() != parent1.len() {
    let to_breed_1: Vec<usize> = parent1.iter(): Iter<usize>
      .filter(|bit: &&usize| !offspring2.contains(*bit)): impl Iterator<Item = &usize>
      .copied(): impl Iterator<Item = usize>
      .collect();
    let to_breed_2: Vec<usize> = parent2.iter(): Iter<usize>
      .filter(|bit: &&usize| !offspring1.contains(*bit)): impl Iterator<Item = &usize>
      .copied(): impl Iterator<Item = usize>
      .collect();

    let (mut offspring1_tail: Vec<usize>, mut offspring2_tail: Vec<usize>) = breed(parent1: &to_breed_1, parent2: &to_breed_2);
    offspring1.append(&mut offspring1_tail);
    offspring2.append(&mut offspring2_tail);
  }

  (offspring1, offspring2)
}
```

Рисунок А.20 Крок 6 алгоритму з підрозділу 2.2.

Далі йде сортування популяції по «якості» за допомогою методу *self sort_population*. Даний метод можна спостерігати на рис 7.15.

```
fn sort_population<T: Distance + Clone>(
    &self,
    points: &Vec<T>,
    distances_mat: &Vec<Vec<f64>>,
    population: &mut Vec<Vec<usize>>
) {
    population: &mut Vec<Vec<usize>>
        .sort_by(compare: |a: &Vec<usize>, b: &Vec<usize>|
            calc_fitness(&points, &distances_mat, path: &b): f64
                .partial_cmp(&calc_fitness(&points, &distances_mat, path: &a)): Option<Ordering>
                .unwrap()
        );
}
```

Рисунок А.21 Функція сортування покоління по «якості».

Після сортування ми беремо частину індивідів, що повинні перейти у наступне покоління і з’єднуємо їх із новозгенерованим поколінням.

Нове покоління утворене.

1.10 Реалізація алгоритму тестування ГА

```
pub fn genetic_alg_test<T: Distance + Clone>(test_data: &Vec<Vec<T>>, file_prefix: &str) {
    let population_size: usize = 100;
    let dying_rate: f64 = 0.2;
    let mut_rate: f64 = 0.1;
    let crossover_rate: f64 = 0.8;

    for points: &Vec<T> in test_data {
        let points: Vec<SpherePoint> = random_sphere_points(points.len());

        let write_file: File = std::fs::File::create(
            path: format!("{}_ma_test_{}.csv", file_prefix, points.len())
        ).unwrap();
        let mut writer: BufWriter<&File> = std::io::BufWriter::new(inner: &write_file);

        writeln!(&mut writer, "gens_count,path_length,time_elapsed");

        for gens_count: usize in (10..1000).step_by(step: 10) {
            let now: Instant = std::time::Instant::now();

            let solver: GeneticAlgTSPSolver = GeneticAlgTSPSolver::new(
                population_size,
                gens_count,
                crossover_rate,
                mut_rate,
                dying_rate
            );
            let result: Vec<SpherePoint> = solver.solve_tsp(&points, start_point_idx: 0);

            let elapsed: Duration = now.elapsed();

            let path_length: f64 = calculate_path_length(points: &result);

            writeln!(&mut writer, "{},{},{}", gens_count, path_length, elapsed.as_millis());
        }
    }
}
```

Рисунок А.22 Функція тестування ГА.

1.11 Функція тестування МА

```
pub fn ant_alg_test<T: Distance + Clone>(test_data: &Vec<Vec<T>>, file_prefix: &str) {
    let N: [usize; 3] = [10, 100, 500];
    let alfa: f64 = 0.5;
    let beta: f64 = 0.5;
    let ro: f64 = 0.1;
    let ant_capacity: f64 = 10.0;
    let initial_pheromones_val: f64 = 1.0;

    for points: &Vec<T> in test_data {
        let write_file: File = std::fs::File::create(
            path: format!("{}_ma_test_{}.csv", file_prefix, points.len())
        ).unwrap();
        let mut writer: BufWriter<&File> = std::io::BufWriter::new(inner: &write_file);

        writeln!(&mut writer, "iters_count,path_length,time_elapsed");

        for iters_count: usize in (10..1000).step_by(step: 10) {
            let now: Instant = std::time::Instant::now();

            let solver: AntAlgTSPSolver = AntAlgTSPSolver::new(
                alfa,
                beta,        You, 2 days ago • Initial commit
                ro,
                iters_count,
                ant_capacity,
                initial_pheromones_val,
            );
            let result: Vec<T> = solver.solve_tsp(&points, start_point_idx: 0);

            let elapsed: Duration = now.elapsed();

            let path_length: f64 = calculate_path_length(points: &result);

            writeln!(&mut writer, "{},{},{}", iters_count, path_length, elapsed.as_millis());
        }
    }
}
```

Рисунок А.23 Функція тестування МА.

1.12 Код для візуалізації отриманих даних

```
import sys
import matplotlib
import matplotlib.pyplot as plt
from matplotlib.ticker import MaxNLocator
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
from numpy.random import randn
from scipy import array, newaxis
from numpy import genfromtxt
```

Рисунок А.24 Імпорт бібліотек

```
def show_data(filepath, xlabel, ylabel, zlabel):
    DATA = genfromtxt(filepath, delimiter=',', skip_header=1)
    Xs = DATA[:,0] # gens_count or iters_counts
    Ys = DATA[:,1] # path_length
    Zs = DATA[:,2] # time_elapsed

    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    surf = ax.plot_trisurf(Xs, Ys, Zs, cmap=cm.jet, linewidth=0)
    fig.colorbar(surf, shrink=0.9, pad = 0.1)

    ax.set_xlabel(xlabel)
    ax.set_ylabel(ylabel)
    ax.set_zlabel(zlabel)

    ax.xaxis.set_major_locator(MaxNLocator(5))
    ax.yaxis.set_major_locator(MaxNLocator(6))
    ax.zaxis.set_major_locator(MaxNLocator(5))

    ax.view_init(30, 130)

    fig.tight_layout()
    # rotate the axes and update
    plt.show()
```

Рисунок А.25 Функція для зображення 3d графіку по даним із файлу за шляхом filepath.

```
def show_data2d(filepath, xlabel, ylabel, zlabel):
    DATA = genfromtxt(filepath, delimiter=',', skip_header=1)
    Xs = DATA[:,0] # gens_count or iters_counts
    Ys = DATA[:,1] # path_length
    Zs = DATA[:,2] # time_elapsed

    plt.xlabel(zlabel)
    plt.ylabel(ylabel)

    plt.plot(Zs, Ys, 'bo')

    z = np.polyfit(Zs, Ys, 1)
    p = np.poly1d(z)
    plt.plot(Zs ,p(Zs),"r--")

    plt.show()
```

Рисунок А.26 Функція для зображення 2d графіку по даним із файлу за шляхом filepath.

```
show_data('diplom/ga_test_10.csv', 'gens_count', 'path_length', 'time_elapsed')
```

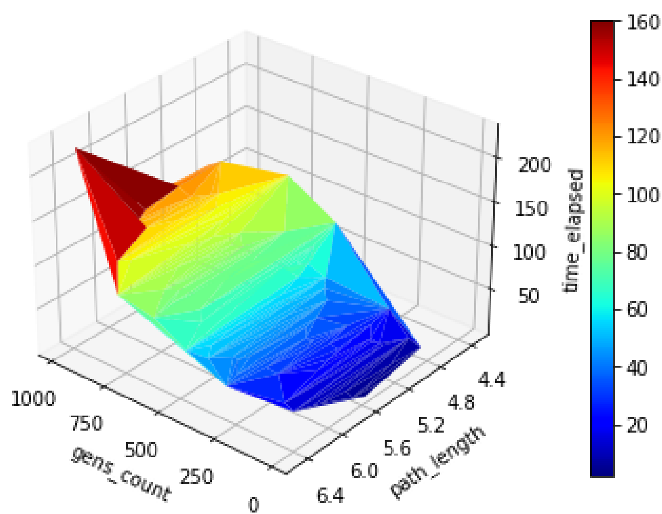


Рисунок А.27 Приклад виконання функції show_data.

1.13 Структура DiscretePoint

У даній структурі ми маємо єдине поле identifier типу usize. Також реалізовано трейт Distance використовуючи знання із підрозділу 1.5.

```
#[derive(Debug, Clone)]
pub struct DiscretePoint {
  identifier: usize,
}

impl DiscretePoint {
  fn new(identifier: usize) -> Self {
    DiscretePoint {
      identifier,
    }
  }
}

impl Distance for DiscretePoint {
  fn distance_to(&self, point: &Self) -> f64 {
    if self.identifier == point.identifier {
      0.0
    } else {
      ((self.identifier - point.identifier) % 2) as f64
    }
  }
}
```

Рисунок А.28 Структура DiscretePoint.

1.14 Генерація DiscretePoint

Дана функція генерує випадковий DiscretePoint використовуючи бібліотеку rand для генерації випадкового identifier.

```
pub fn random_discrete_point() -> DiscretePoint {
  let mut thread_rng: ThreadRng = rand::thread_rng();
  DiscretePoint::new(
    identifier: thread_rng.gen(),
  )
}
```

Рисунок А.29 Функція random_discrete_point.

1.15 Генерація масиву з екземплярами структури DiscretePoint

Дана функція генерує масив розміру n з випадковими точками використовуючи функцію з Додатку А.14.

```
pub fn random_discrete_points(n: usize) -> Vec<DiscretePoint> {
    (0..n): Range<usize>
        .map(|_| random_discrete_point()): impl Iterator<Item = DiscretePoint>
        .collect()
}
```

Рисунок А.30 Функція radom_discrete_points.

10.

ДОДАТОК 2

Посилання на код і дані

Посилання на репозиторій з програмним кодом кваліфікаційної роботи: https://github.com/ycatbink0t/diplom