**Міністерство освіти і науки України**

**Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського"**

**Факультет інформатики та обчислювальної техніки**

**Кафедра інформатики та програмної інженерії**

**Звіт**

з лабораторної роботи № 5 з дисципліни

«Проектування алгоритмів»

„**Проектування і аналіз алгоритмів для вирішення NP-складних задач ч.2**”

**Виконав(ла)**

(шифр, прізвище, ім'я, по батькові)

*ІП-14 Прокопенко Олексій*

**Перевірив**

(прізвище, ім'я, по батькові)

*Головченко М.Н.*

Київ 2022

Зміст

[1 Мета лабораторної роботи 3](#_Toc52291748)

[2 Завдання 4](#_Toc52291749)

[3 Виконання 10](#_Toc52291750)

[3.1 Покроковий алгоритм 10](#_Toc52291751)

[3.2 Програмна реалізація алгоритму 10](#_Toc52291752)

[3.2.1 Вихідний код 10](#_Toc52291753)

[3.2.2 Приклади роботи 10](#_Toc52291754)

[3.3 Тестування алгоритму 11](#_Toc52291755)

[Висновок 12](#_Toc52291756)

[Критерії оцінювання 13](#_Toc52291757)

# Мета лабораторної роботи

Мета роботи – вивчити основні підходи розробки метаеврестичних алгоритмів для типових прикладних задач. Опрацювати методологію підбору прийнятних параметрів алгоритму.

# Завдання

Згідно варіанту, формалізувати алгоритм вирішення задачі відповідно загальної методології.

Записати розроблений алгоритм у покроковому вигляді. З достатнім степенем деталізації.

Виконати його програмну реалізацію на будь-якій мові програмування.

Перелік задач наведено у таблиці 2.1.

Перелік алгоритмів і досліджуваних параметрів у таблиці 2.2.

Задача і алгоритм наведені в таблиці 2.3.

Змінюючи параметри алгоритму, визначити кращі вхідні параметри алгоритму. Для цього необхідно:

* обрати критерій зупинки алгоритму (кількість ітерацій або значення ЦФ);
* зафіксувати усі параметри крім одного і змінювати цей параметр, поки не буде досягнуто пікової ефективності;
* після цього параметр фіксується і змінюються інші параметри;
* далі повторюємо процедуру спочатку, з першого зафіксованого параметру;
* зупиняємось коли будуть знайдені оптимальні параметри для даної задачі або встановлена залежність одних параметрів від інших.

Зробити узагальнений висновок в якому обов’язково описати залежність якості розв’язку від вхідних параметрів.

Таблиця 2.1 – Прикладні задачі

|  |  |
| --- | --- |
| **№** | **Задача** |
| 2 | **Задача комівояжера** (300 вершин, відстань між вершинами випадкова від 5 до 150) полягає у знаходженні найвигіднішого маршруту, що проходить через вказані міста хоча б по одному разу. В умовах завдання вказуються критерій вигідності маршруту (найкоротший, найдешевший, сукупний критерій тощо) і відповідні матриці відстаней, вартості тощо. Зазвичай задано, що маршрут повинен проходити через кожне місто тільки один раз, в такому випадку розв'язок знаходиться серед гамільтонових циклів.  **Розглядається симетричний, асиметричний та змішаний варіанти.**  В загальному випадку, асиметрична задача комівояжера відрізняється тим, що ребра між вершинами можуть мати різну вагу в залежності від напряму, тобто, задача моделюється орієнтованим графом. Таким чином, окрім ваги ребер графа, слід також зважати і на те, в якому напрямку знаходяться ребра.  У випадку симетричної задачі всі пари ребер між одними й тими самими вершинами мають однакову вагу.  У випадку реальних міст може бути як симетричною, так і асиметричною в залежності від тривалості або довжини маршрутів і напряму руху.  Застосування:   * доставка товарів (в цьому випадку може бути більш доречна постановка транспортної задачі - доставка в кілька магазинів з декількох складів); * доставка води; * моніторинг об'єктів; * поповнення банкоматів готівкою; * збір співробітників для доставки вахтовим методом. |

Таблиця 2.2 – Варіанти алгоритмів і досліджувані параметри

|  |  |
| --- | --- |
| **№** | **Алгоритми і досліджувані параметри** |
| 3 | **Бджолиний алгоритм:**   * кількість ділянок; * кількість бджіл (фуражирів і розвідників). |

Таблиця 2.3 – Варіанти задач і алгоритмів

|  |  |
| --- | --- |
| **№** | **Задачі і алгоритми** |
| 13 | Задача комівояжера (змішана мережа) + Бджолиний алгоритм |

# Виконання

## Покроковий алгоритм

…

## Програмна реалізація алгоритму

### Вихідний код

import csv

import math

import random

import sys

from scipy.spatial import distance

class Bee:

def \_\_init\_\_(self, node\_set):

self.role = ''

self.path = list(node\_set)

self.distance = 0

self.cycle = 0

def get\_distance\_between\_nodes(n1, n2):

return distance.euclidean(n1, n2)

def make\_distance\_table(data\_list):

length = len(data\_list)

table = [[get\_distance\_between\_nodes(

(data\_list[i][1],data\_list[i][2]), (data\_list[j][1],data\_list[j][2]))

for i in range(0, length)] for j in range(0, length)]

return table

def get\_total\_distance\_of\_path(path, table):

new\_path = list(path)

new\_path.insert(len(path), path[0])

new\_path = new\_path[1:len(new\_path)]

coordinates = zip(path, new\_path)

distance = sum([table[i[0]][i[1]] for i in coordinates])

return round(distance, 3)

def initialize\_hive(population, data):

path = [x[0] for x in data]

hive = [Bee(path) for i in range (0, population)]

return hive

def assign\_roles(hive, role\_percentiles, table):

population = len(hive)

onlooker\_count = math.floor(population \* role\_percentiles[0])

forager\_count = math.floor(population \* role\_percentiles[1])

for i in range(0, onlooker\_count):

hive[i].role = 'O'

for i in range(onlooker\_count, (onlooker\_count + forager\_count)):

hive[i].role = 'F'

random.shuffle(hive[i].path)

hive[i].distance = get\_total\_distance\_of\_path(hive[i].path, table)

return hive

def mutate\_path(path):

random\_idx = random.randint(0, len(path) - 2)

new\_path = list(path)

new\_path[random\_idx], new\_path[random\_idx + 1] = new\_path[random\_idx + 1], new\_path[random\_idx]

return new\_path

def forage(bee, table, limit):

new\_path = mutate\_path(bee.path)

new\_distance = get\_total\_distance\_of\_path(new\_path, table)

if new\_distance < bee.distance:

bee.path = new\_path

bee.distance = new\_distance

bee.cycle = 0

else:

bee.cycle += 1

if bee.cycle >= limit:

bee.role = 'S'

return bee.distance, list(bee.path)

def scout(bee, table):

new\_path = list(bee.path)

random.shuffle(new\_path)

bee.path = new\_path

bee.distance = get\_total\_distance\_of\_path(new\_path, table)

bee.role = 'F'

bee.cycle = 0

def waggle(hive, best\_distance, table, forager\_limit, scout\_count):

best\_path = []

results = []

for i in range(0, len(hive)):

if hive[i].role == 'F':

distance, path = forage(hive[i], table, forager\_limit)

if distance < best\_distance:

best\_distance = distance

best\_path = list(hive[i].path)

results.append((i, distance))

elif hive[i].role == 'S':

scout(hive[i], table)

results.sort(reverse = True, key=lambda tup: tup[1])

scouts = [ tup[0] for tup in results[0:int(scout\_count)] ]

for new\_scout in scouts:

hive[new\_scout].role = 'S'

return best\_distance, best\_path

def recruit(hive, best\_distance, best\_path, table):

for i in range(0, len(hive)):

if hive[i].role == 'O':

new\_path = mutate\_path(best\_path)

new\_distance = get\_total\_distance\_of\_path(new\_path, table)

if new\_distance < best\_distance:

best\_distance = new\_distance

best\_path = new\_path

return best\_distance, best\_path

def print\_details(cycle, distance):

print("CYCLE: {}".format(cycle))

print("DISTANCE: {}".format(distance))

print("\n")

def generate\_towns():

return [[random.randint(5, 150) for j in range(3)] for i in range(300)]

def main():

population = 500

forager\_percent = 0.9

scout\_percent = 0.1

role\_percent = [scout\_percent, forager\_percent]

scout\_count = math.ceil(population \* scout\_percent)

forager\_limit = 500

cycle\_limit = 2000

cycle = 1

data = generate\_towns()

best\_distance = sys.maxsize

best\_path = []

result = ()

table = make\_distance\_table(data)

hive = initialize\_hive(population, data)

assign\_roles(hive, role\_percent, table)

print("Configuration: ")

print("The traveling salesman problem type: mixed")

print("Forager count: ", population \* forager\_percent)

print("Scout count: ", scout\_count);

while cycle < cycle\_limit:

waggle\_distance, waggle\_path = waggle(hive, best\_distance, table, forager\_limit, scout\_count)

if waggle\_distance < best\_distance:

best\_distance = waggle\_distance

best\_path = list(waggle\_path)

print\_details(cycle, best\_distance)

result = (cycle, best\_distance)

recruit\_distance, recruit\_path = recruit(hive, best\_distance, best\_path, table)

if recruit\_distance < best\_distance:

best\_distance = recruit\_distance

best\_path = list(recruit\_path)

print\_details(cycle, best\_distance)

result = (cycle, best\_distance)

if cycle % 1000 == 0:

print("CYCLE #: {}\n".format(cycle))

cycle += 1

print("Result: ")

print("Cycle: ", cycle)

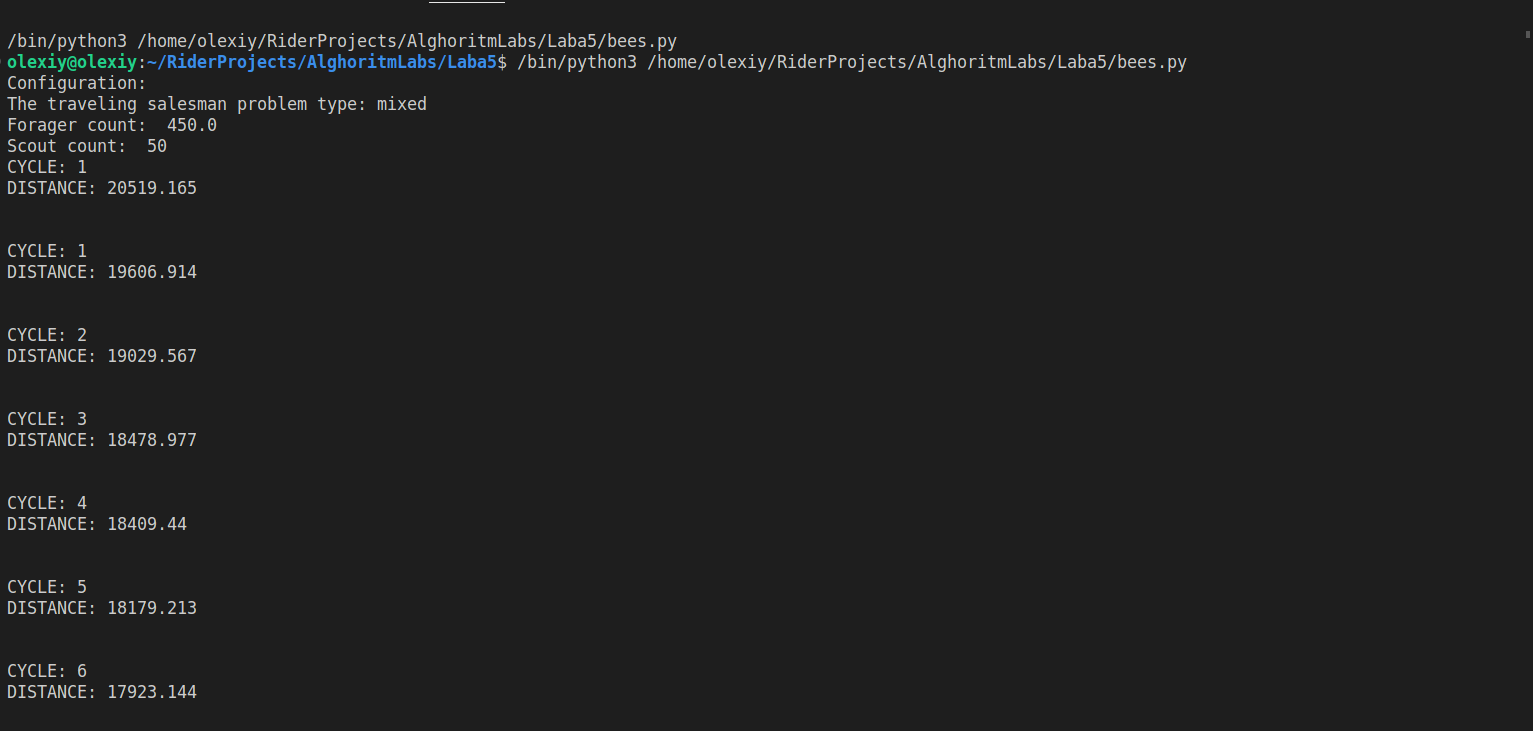
print("Best distance: ", best\_distance)

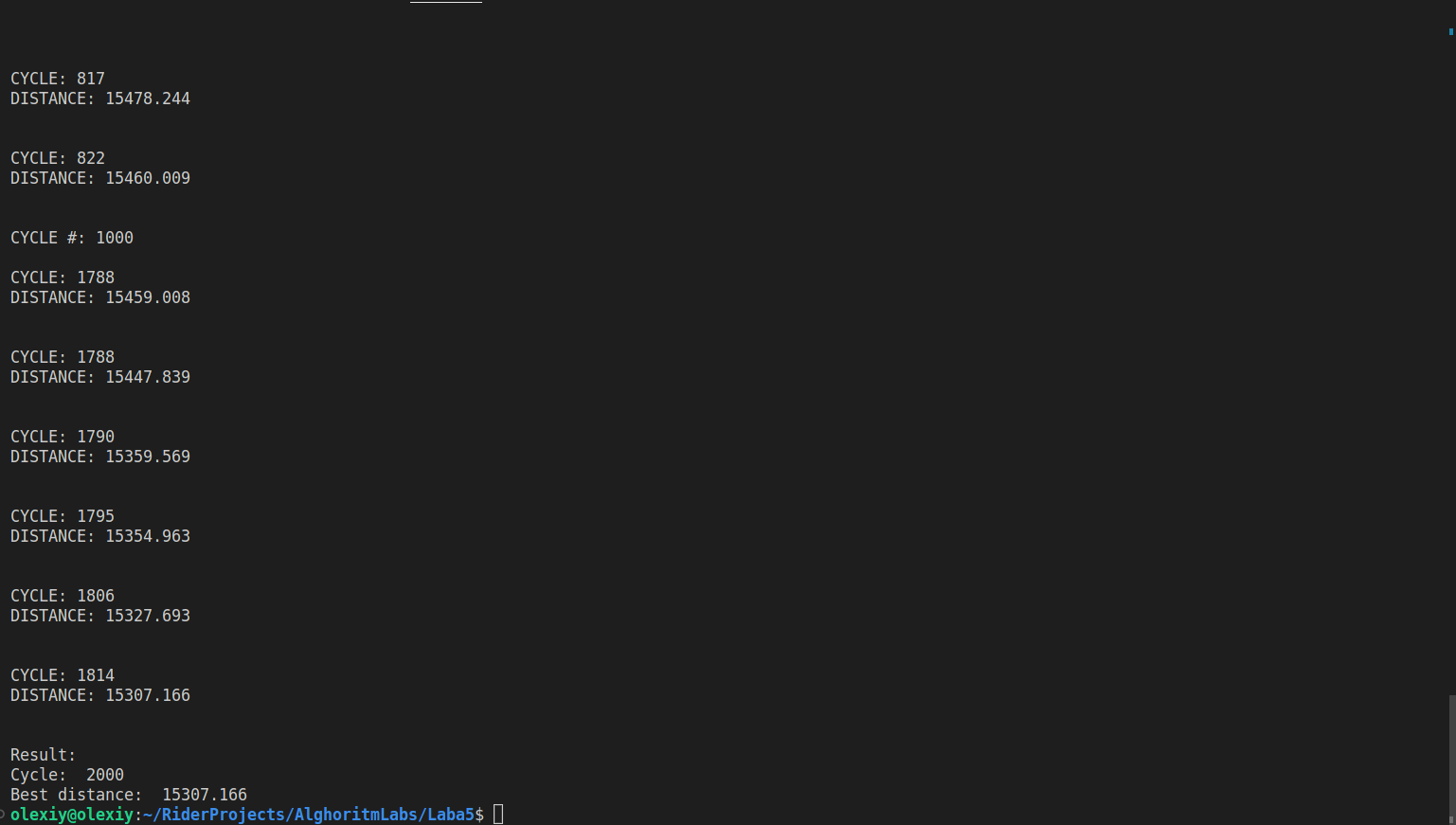
if \_\_name\_\_ == '\_\_main\_\_':

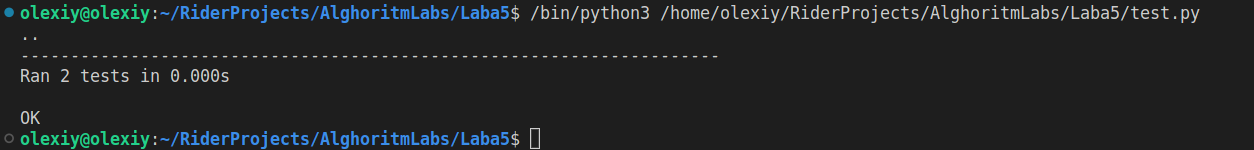
main()

### Приклади роботи

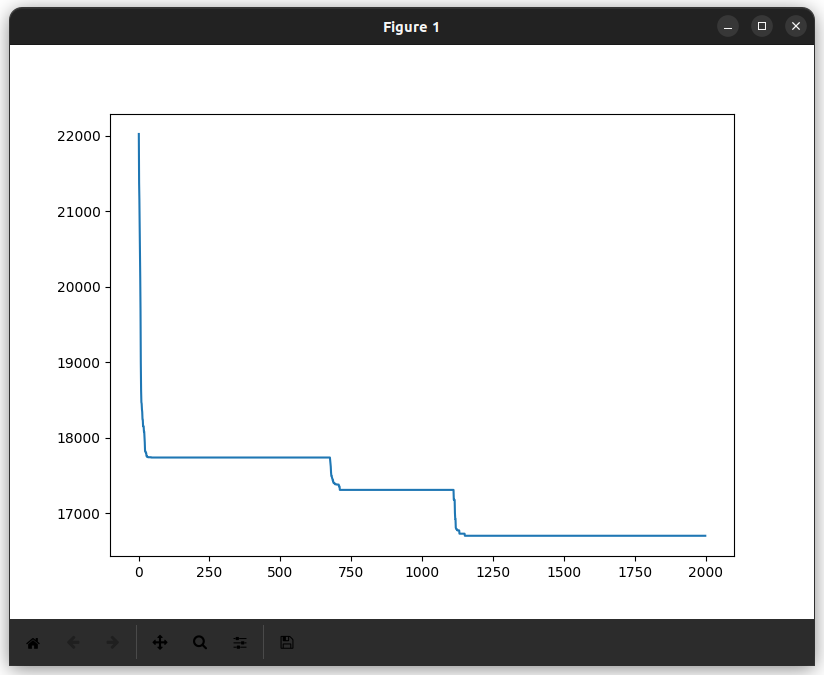
На рисунках 3.1 і 3.2 показані приклади роботи програми.

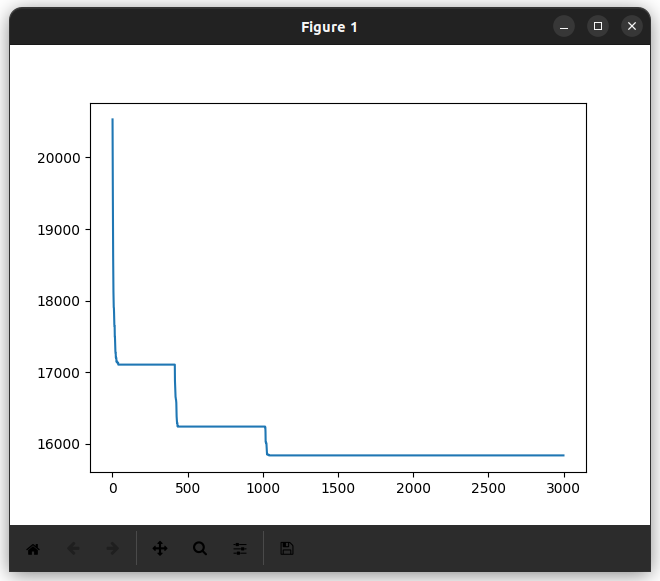
Рисунок 3.1 – початок роботи програми

Рисунок 3.2 – вивід результату роботи програми

Результат запуску юніт тестів

## Тестування алгоритму

Рисунок 3.3 – графік залежності відстані від кількості ітерацій (450 фуражирів, 50 розвідників, 2000 ітерацій)

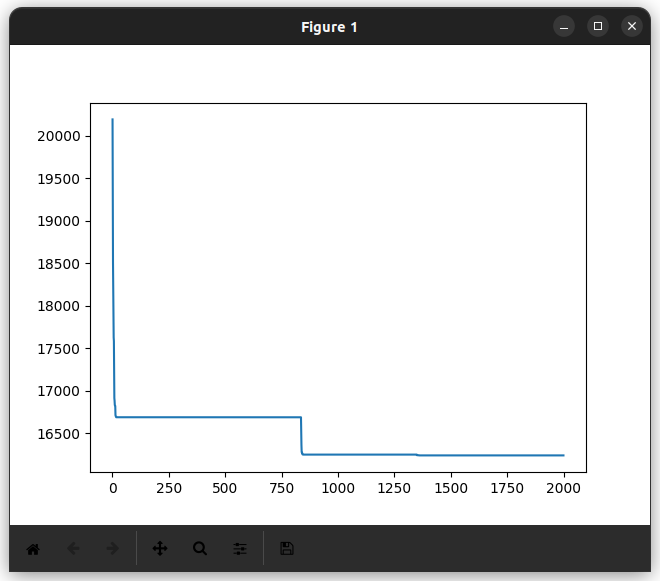
Рисунок 3.4 – графік залежності відстані від кількості ітерацій (450 фуражирів, 50 розвідників, 3000 ітерацій)

|  |  |
| --- | --- |
| Номер ітерації | Відстань |
| 1 | 20949.216 |
| 2 | 19768.86 |
| 20 | 17289.729 |
| 39-415 | 17106.428 |
| 416 | 17094.946 |
| 435-1015 | 16240.98 |
| 1016 | 16240.95 |
| 1027 | 15859.177 |
| 1041-2000 | 15838.414 |

Таблиця 3.1

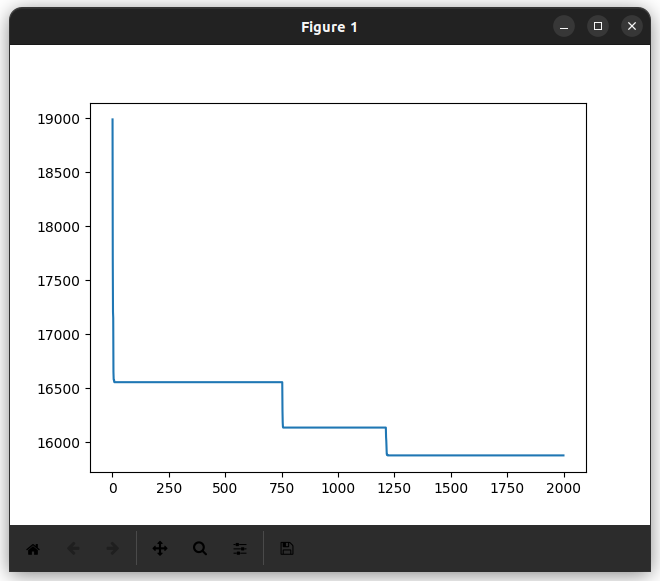
**Висновок для 450 фуражирів та 50 розвідників**

За результати проведеного тестування видно, що в перші приблизно 40 (в таблиці 3.1 цей результат 39) ітерацій відстань необхідна для подолання щоб відвідати всі міста миттєво зменшується. Після цього алгоритму велику цілкість ітерацій не вдається знайти вигідніший шлях, але приблизно через 400 (в таблиці 3.1 це 417) ітерацій він його знаходить, потім ще 600 (в таблиці 3.1 це 580) ітерацій не знаходить вигіднішого шляху. Приблизно на тисячній ітерації (в таблиці 3.1 це 1016-та) алгоритм знаходить правильний шлях, і протягом приблизно 20 (в таблиці 3.1 це 25) ітерацій покращує його, і десь на 1050 (в таблиці 3.1 це 1041) ітерації він знаходить результат.

Рисунок 3.5 - графік залежності відстані від кількості ітерацій (900 фуражирів, 100 розвідників, 2000 ітерацій)

**Висновок для 900 фуражирів та 100 розвідників**

При збільшенні кількості бджіл вдвічі зберігаючи пропорції фуражирів та розвідників видно що рішення знаходилося на приблизно 200 ітерацій довше.

Рисунок 3.6 - графік залежності відстані від кількості ітерацій (800 фуражирів, 200 розвідників, 2000 ітерацій)

**Висновок для 800 фуражирів та 200 розвідників**

При збільшенні кількості розвідників до 20% швидкість знаходження розв’язку зросла порівняно з 10%, але все одно менше ніж 10% і 500 бджіл.

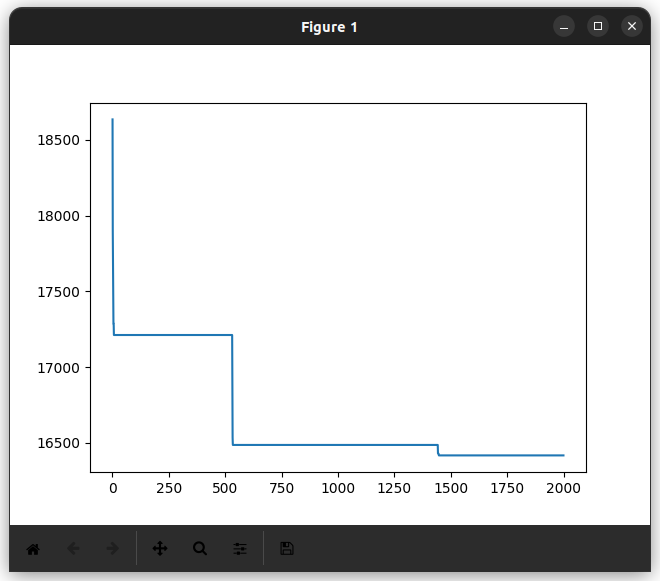
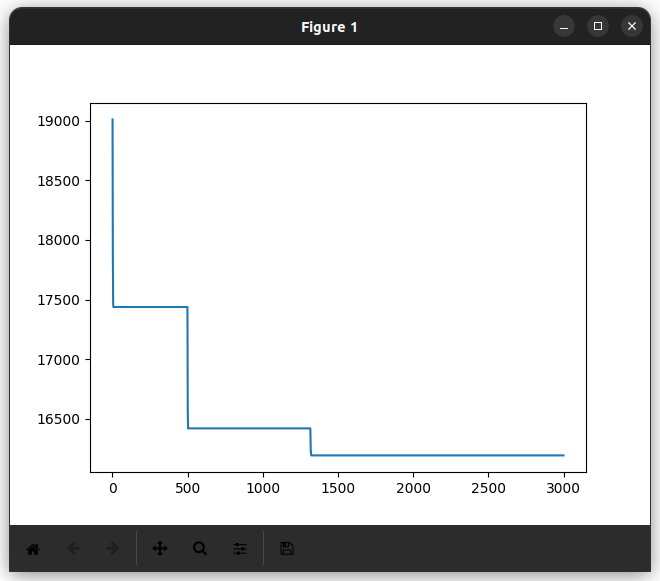


Рисунок 3.7 - графік залежності відстані від кількості ітерацій (700 фуражирів, 300 розвідників, 2000 ітерацій)

Рисунок 3.8 - графік залежності відстані від кількості ітерацій (700 фуражирів, 300 розвідників, 3000 ітерацій)

**Висновок для 700 фуражирів та 300 розвідників**

При збільшенні кількості розвідників до 30% швидкість знаходження розв’язку зменшилась порівняно з 20%.

Висновок

В рамках даної лабораторної роботи було реалізовано бджолиний алгоритм для задачі комівояжера. За дослідженням оптимальним рішенням було створити 10% розвідників та 90% фуражирів. При використанні такого набору бджіл розв’язок знаходився на приблизно 1050-ій ітерації. Також було досліджено його роботу з різною кількістю фуражирів та розвідників. Як показує дослідження, неефективно використовувати більше ніж 20% розвідників та робити більше ніж 1500 ітерацій.

Критерії оцінювання

При здачі лабораторної роботи до 11.12.2022 включно максимальний бал дорівнює – 5. Після 11.12.2022 максимальний бал дорівнює – 1.

Критерії оцінювання у відсотках від максимального балу:

* покроковий алгоритм – 15%;
* програмна реалізація алгоритму – 50%;
* тестування алгоритму– 30%;
* висновок – 5%.