**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-13 Вдовиченко С.Ю.*

**Перевірив**

(прізвище, ім'я, по батькові)

*Сопов О.О.*

Київ 2022

Зміст

[1 Мета лабораторної роботи 3](#_Toc52291748)

[2 Завдання 4](#_Toc52291749)

[3 Виконання 8](#_Toc52291750)

[3.1 Покроковий алгоритм 8](#_Toc52291751)

[3.2 Програмна реалізація алгоритму 8](#_Toc52291752)

[3.2.1 Вихідний код 8](#_Toc52291753)

[3.2.2 Приклади роботи 11](#_Toc52291754)

[3.3 Тестування алгоритму 13](#_Toc52291755)

[Висновок 16](#_Toc52291756)

[Критерії оцінювання 17](#_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** |
| 1 | **Генетичний алгоритм:**   * оператор схрещування (мінімум 3); * мутація (мінімум 2); * оператор локального покращення (мінімум 2). |

Таблиця 2.3 – Варіанти задач і алгоритмів

|  |  |
| --- | --- |
| **№** | **Задачі і алгоритми** |
| 1 | Задача про рюкзак + Генетичний алгоритм |
| 2 | Задача про рюкзак + Бджолиний алгоритм |
| 3 | Задача комівояжера (асиметрична мережа) + Генетичний алгоритм |
| 4 | Задача комівояжера (симетрична мережа) + Генетичний алгоритм |
| 5 | Задача комівояжера (змішана мережа) + Генетичний алгоритм |
| 6 | Задача комівояжера (асиметрична мережа) + Мурашиний алгоритм |
| 7 | Задача комівояжера (симетрична мережа) + Мурашиний алгоритм |
| 8 | Задача комівояжера (змішана мережа) + Мурашиний алгоритм |
| 9 | Задача вершинного покриття + Генетичний алгоритм |
| 10 | Задача вершинного покриття + Бджолиний алгоритм |
| 11 | Задача комівояжера (асиметрична мережа) + Бджолиний алгоритм |
| 12 | Задача комівояжера (симетрична мережа) + Бджолиний алгоритм |
| 13 | Задача комівояжера (змішана мережа) + Бджолиний алгоритм |
| 14 | Розфарбовування графа + Генетичний алгоритм |
| 15 | Розфарбовування графа + Бджолиний алгоритм |
| 16 | Задача про кліку (задача розпізнавання) + Генетичний алгоритм |
| 17 | Задача про кліку (задача розпізнавання) + Бджолиний алгоритм |
| 18 | Задача про кліку (обчислювальна задача) + Генетичний алгоритм |
| 19 | Задача про кліку (обчислювальна задача) + Бджолиний алгоритм |
| 20 | Задача про найкоротший шлях + Генетичний алгоритм |
| 21 | Задача про найкоротший шлях + Мурашиний алгоритм |
| 22 | Задача про найкоротший шлях + Бджолиний алгоритм |
| 23 | Задача про рюкзак + Генетичний алгоритм |
| 24 | Задача про рюкзак + Бджолиний алгоритм |
| 25 | Задача комівояжера (асиметрична мережа) + Генетичний алгоритм |
| 26 | Задача комівояжера (симетрична мережа) + Генетичний алгоритм |
| 27 | Задача комівояжера (змішана мережа) + Генетичний алгоритм |
| 28 | Задача комівояжера (асиметрична мережа) + Мурашиний алгоритм |
| 29 | Задача комівояжера (симетрична мережа) + Мурашиний алгоритм |
| 30 | Задача комівояжера (змішана мережа) + Мурашиний алгоритм |

# Виконання

## Покроковий алгоритм

* Створюємо клас City для міст та розрахунку відстані.
* Створюємо клас Fitness.
* Створюємо початкову популяцію.
* Використовуємо Fitness для сортування кожної особи, створюємо список з ідентифікаторами маршрутів та вартістю.
* Обираємо батьків методом **Fitness proportionate selection.**
* Використовуємо оператор схрещування.
* Застововуємо мутацію.
* Застосовуємо генетичний алгоритм для визначеної кількості ітерацій: створюємо нову популяцію, обираємо найкращу.

## Програмна реалізація алгоритму

### Вихідний код

import numpy as np, random, operator, pandas as pd  
  
class City:  
 def \_\_init\_\_(self, x, y):  
 self.x = x  
 self.y = y  
  
 def distance(self, city):  
 x\_distance = abs(self.x - city.x)  
 y\_distance = abs(self.y - city.y)  
 distance = np.sqrt((x\_distance \*\* 2) + (y\_distance \*\* 2))  
 return distance  
  
 def \_\_repr\_\_(self):  
 return "(" + str(self.x) + "," + str(self.y) + ")"  
  
class Fitness:  
 def \_\_init\_\_(self, route):  
 self.route = route  
 self.distance = 0  
 self.fitness = 0.0  
  
 def route\_distance(self):  
 if self.distance == 0:  
 path\_distance = 0  
 for i in range(0, len(self.route)):  
 from\_city = self.route[i]  
 to\_city = None  
 if i + 1 < len(self.route):  
 to\_city = self.route[i + 1]  
 else:  
 to\_city = self.route[0]  
 path\_distance += from\_city.distance(to\_city)  
 self.distance = path\_distance  
 return self.distance  
  
 def route\_fitness(self):  
 if self.fitness == 0:  
 self.fitness = 1 / float(self.route\_distance())  
 return self.fitness  
  
  
def create\_route(city\_list):  
 route = random.sample(city\_list, len(city\_list))  
 return route  
  
  
def initial\_population(pop\_size, city\_list):  
 population = []  
  
 for i in range(0, pop\_size):  
 population.append(create\_route(city\_list))  
 return population  
  
  
def rank\_routes(population):  
 fitness\_results = {}  
 for i in range(0,len(population)):  
 fitness\_results[i] = Fitness(population[i]).route\_fitness()  
 return sorted(fitness\_results.items(), key = operator.itemgetter(1), reverse = True)  
  
  
def selection(pop\_ranked, elite\_size):  
 selection\_results = []  
 df = pd.DataFrame(np.array(pop\_ranked), columns=["Index", "Fitness"])  
 df['cum\_sum'] = df.Fitness.cumsum()  
 df['cum\_perc'] = 100 \* df.cum\_sum / df.Fitness.sum()  
  
 for i in range(0, elite\_size):  
 selection\_results.append(pop\_ranked[i][0])  
 for i in range(0, len(pop\_ranked) - elite\_size):  
 pick = 100 \* random.random()  
 for i in range(0, len(pop\_ranked)):  
 if pick <= df.iat[i, 3]:  
 selection\_results.append(pop\_ranked[i][0])  
 break  
 return selection\_results  
  
  
def mating\_pool(population, selection\_results):  
 matingpool = []  
 for i in range(0, len(selection\_results)):  
 index = selection\_results[i]  
 matingpool.append(population[index])  
 return matingpool  
  
  
def breed(parent1, parent2):  
 child = []  
 child\_p1 = []  
 child\_p2 = []  
  
 gene\_a = int(random.random() \* len(parent1))  
 gene\_b = int(random.random() \* len(parent1))  
  
 start\_gene = min(gene\_a, gene\_b)  
 end\_gene = max(gene\_a, gene\_b)  
  
 for i in range(start\_gene, end\_gene):  
 child\_p1.append(parent1[i])  
  
 child\_p2 = [item for item in parent2 if item not in child\_p1]  
  
 child = child\_p1 + child\_p2  
 return child  
  
  
def breed\_population(mating\_pool, elite\_size):  
 children = []  
 length = len(mating\_pool) - elite\_size  
 pool = random.sample(mating\_pool, len(mating\_pool))  
  
 for i in range(0, elite\_size):  
 children.append(mating\_pool[i])  
  
 for i in range(0, length):  
 child = breed(pool[i], pool[len(mating\_pool) - i - 1])  
 children.append(child)  
 return children  
  
  
def mutate(individual, mutation\_rate):  
 for swapped in range(len(individual)):  
 if (random.random() < mutation\_rate):  
 swap\_with = int(random.random() \* len(individual))  
  
 city1 = individual[swapped]  
 city2 = individual[swap\_with]  
  
 individual[swapped] = city2  
 individual[swap\_with] = city1  
 return individual  
  
  
def mutatePopulation(population, mutation\_rate):  
 mutated\_pop = []  
  
 for ind in range(0, len(population)):  
 mutated\_ind = mutate(population[ind], mutation\_rate)  
 mutated\_pop.append(mutated\_ind)  
 return mutated\_pop  
  
  
def next\_generation(current\_gen, elite\_size, mutation\_rate):  
 pop\_ranked = rank\_routes(current\_gen)  
 selection\_results = selection(pop\_ranked, elite\_size)  
 matingpool = mating\_pool(current\_gen, selection\_results)  
 children = breed\_population(matingpool, elite\_size)  
 next\_generation = mutatePopulation(children, mutation\_rate)  
 return next\_generation  
  
  
def genetic\_algorithm(population, pop\_size, elite\_size, mutation\_rate, generations):  
 pop = initial\_population(pop\_size, population)  
 print(f"Initial distance: {str(int(1 / rank\_routes(pop)[0][1]))}")  
  
 for i in range(0, generations):  
 pop = next\_generation(pop, elite\_size, mutation\_rate)  
  
 print(f"Final distance: {str(int(1 / rank\_routes(pop)[0][1]))}")  
 best\_route\_index = rank\_routes(pop)[0][0]  
 best\_route = pop[best\_route\_index]  
 print(f"Number of iterations: {generations}")  
 # print(best\_route)  
 return best\_route  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 city\_list = []  
  
 for i in range(0, 300):  
 city\_list.append(City(x=int(random.random() \* 5 + random.random() \* 150 - 1),  
 y=int(random.random() \* 5 + random.random() \* 150 - 1)))  
 genetic\_algorithm(population=city\_list, pop\_size=100, elite\_size=60, mutation\_rate=0.3, generations=100)

### Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми.

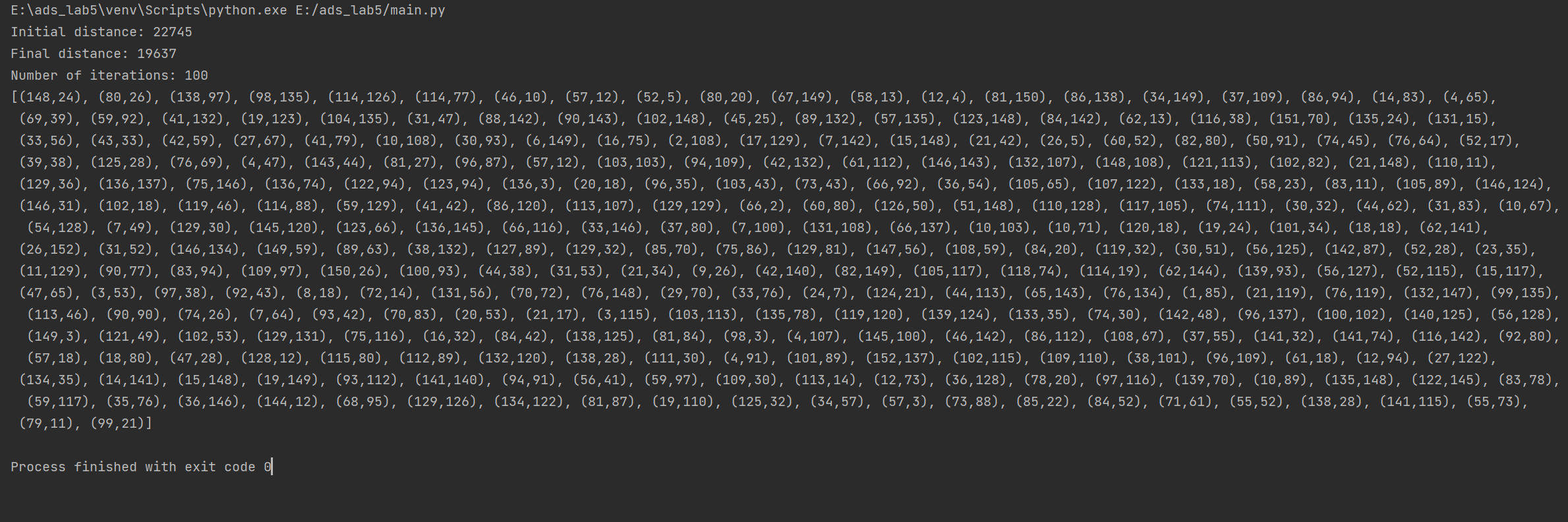


Рисунок 3.1 – Робота алгоритму при 100 ітераціях

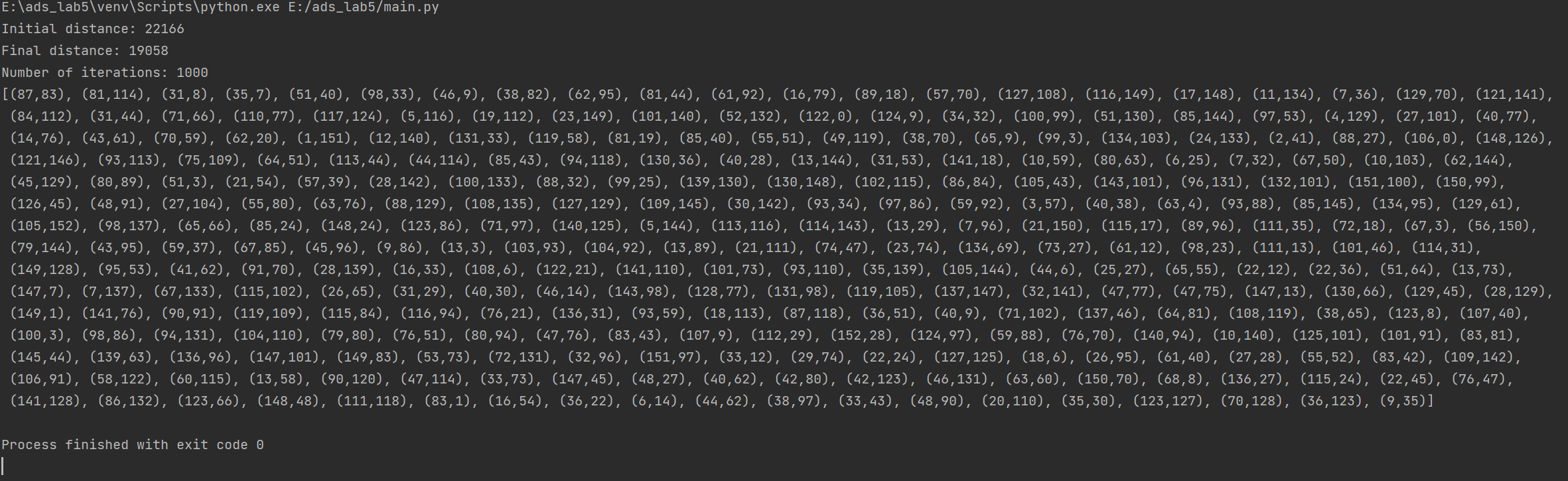


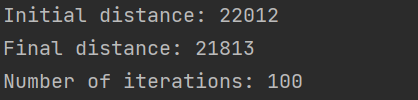
Рисунок 3.2 – Робота алгоритму при 1000 ітераціях

## Тестування алгоритму

Протестуємо алгоритм на різних вхідних даних. Тестування буде проходити на 100 ітераціях, популяція – 100 осіб.

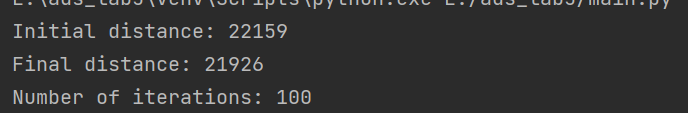
Вхідні дані: 20 найкращих осіб, 0.1 ймовірність мутації.

Результат:



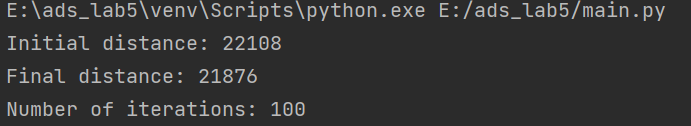
Вхідні дані: 20 найкращих осіб, 0.3 ймовірність мутації.

Результат:



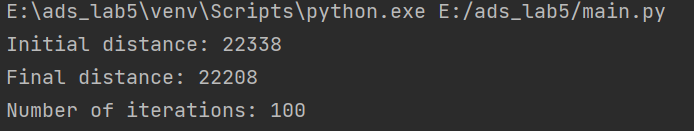
Вхідні дані: 20 найкращих осіб, 0.6 ймовірність мутації.

Результат:



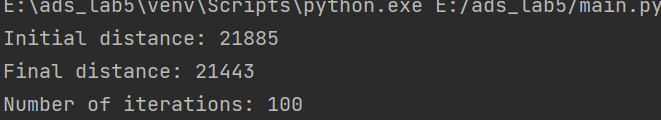
Вхідні дані: 40 найкращих осіб, 0.1 ймовірність мутації.

Результат:



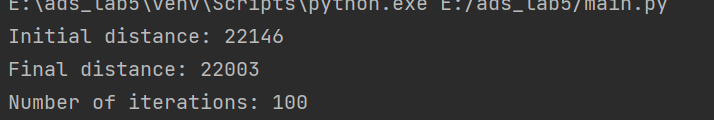
Вхідні дані: 40 найкращих осіб, 0.3 ймовірність мутації.

Результат:



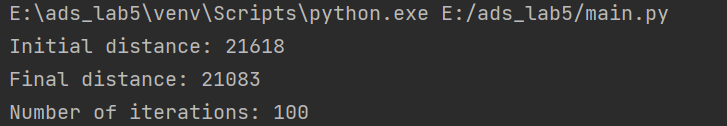
Вхідні дані: 40 найкращих осіб, 0.6 ймовірність мутації.

Результат:



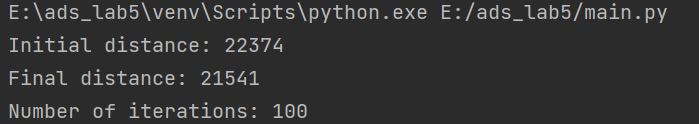
Вхідні дані: 60 найкращих осіб, 0.1 ймовірність мутації.

Результат:



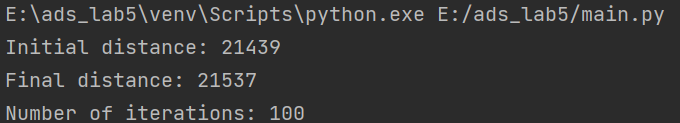
Вхідні дані: 60 найкращих осіб, 0.3 ймовірність мутації.

Результат:



Вхідні дані: 60 найкращих осіб, 0.6 ймовірність мутації.

Результат:



Отже, найкращі вхідні дані – 60 найкращих особин (60% від популяції) та 0.3 ймовірність мутації.

Проаналізуємо результати виконання програми, використовуючи дані вище.

Побудуємо графік залежності вартості від ітерацій.

Для цього побудуємо таблицю.

|  |  |
| --- | --- |
| Ітерації | Вартість |
| 100 | 22898 |
| 200 | 22550 |
| 300 | 22516 |
| 400 | 22439 |
| 500 | 22423 |
| 600 | 22477 |
| 700 | 22362 |
| 800 | 22301 |
| 900 | 22265 |
| 1000 | 22178 |

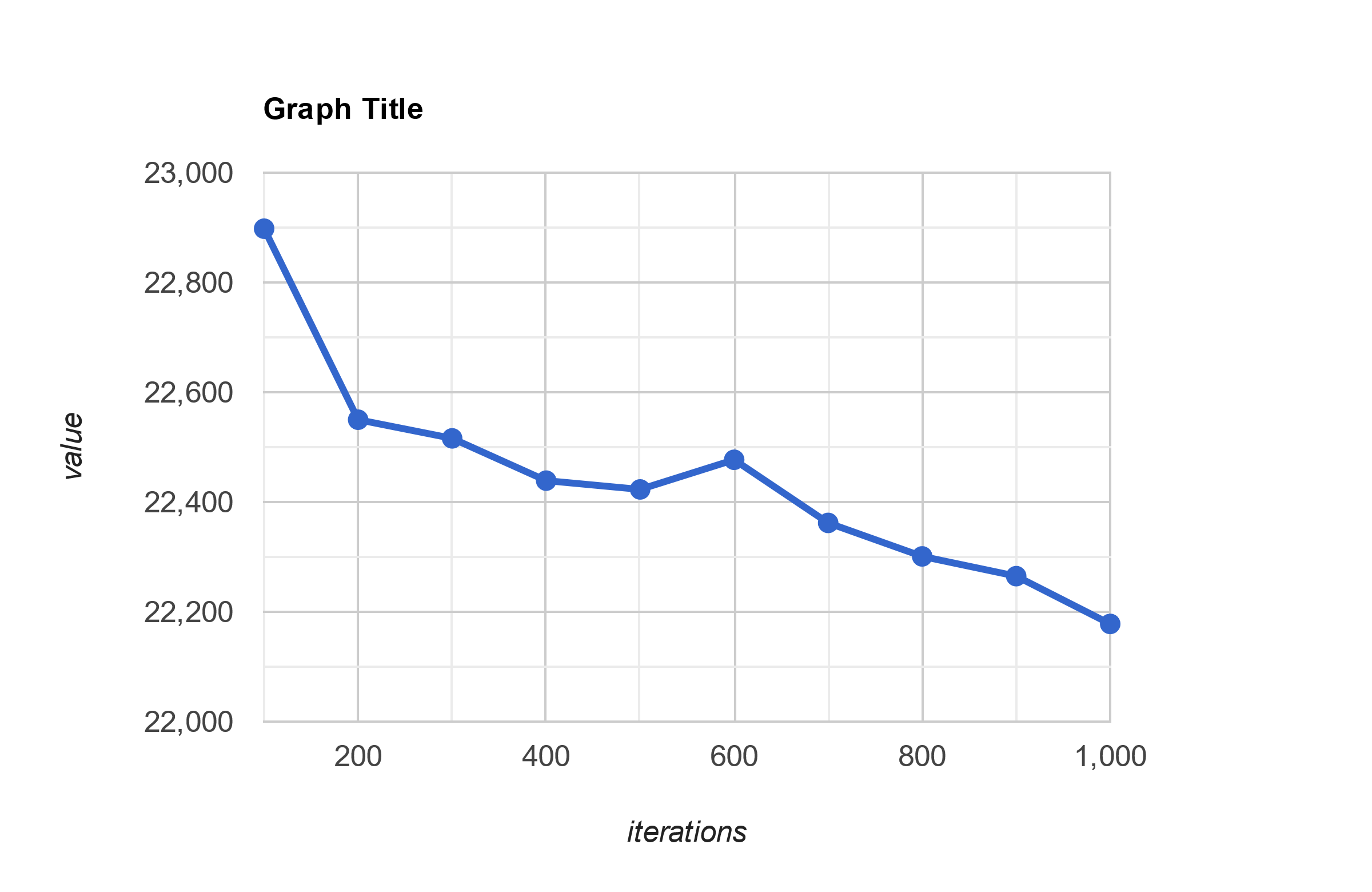


Рисунок 3.3 – Графік залежності результату від ітерацій

Висновок

В рамках даної лабораторної роботи лабораторної роботи я застосував на практиці метаевристичний алгоритм на прикладі задачі комівояжера. Алгоритм було програмно реалізовано та проведено повний аналіз його роботи з тестуванням виконання на найкращих вхідних параметрах.

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%.