**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 № 4 з дисципліни

«Проектування алгоритмів»

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

**Виконав(ла)**

(шифр, прізвище, ім'я, по батькові)

*ІТ-04 Стрільчук Михайло*

**Перевірив**

(прізвище, ім'я, по батькові)

*Сопов О.О.*

*.*

Київ 2021

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** |
| 1 | **Задача про рюкзак** (місткість P=500, 100 предметів, цінність предметів від 2 до 30 (випадкова), вага від 1 до 20 (випадкова)). Для заданої множини предметів, кожен з яких має вагу і цінність, визначити яку кількість кожного з предметів слід взяти, так, щоб сумарна вага не перевищувала задану, а сумарна цінність була максимальною.  Задача часто виникає при розподілі ресурсів, коли наявні фінансові обмеження, і вивчається в таких областях, як комбінаторика, інформатика, теорія складності, криптографія, прикладна математика. |
| 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. |
| 3 | **Розфарбовування графа** (300 вершин, степінь вершини не більше 30, але не менше 2) – називають таке приписування кольорів (або натуральних чисел) його вершинам, що ніякі дві суміжні вершини не набувають однакового кольору. Найменшу можливу кількість кольорів у розфарбуванні називають хроматичне число.  Застосування:   * розкладу для освітніх установ; * розкладу в спорті; * планування зустрічей, зборів, інтерв'ю; * розклади транспорту, в тому числі - авіатранспорту; * розкладу для комунальних служб; |
| 4 | **Задача вершинного покриття** (300 вершин, степінь вершини не більше 30, але не менше 2)**.** Вершинне покриття для неорієнтованого графа G = (V, E) - це множина його вершин S, така, що, у кожного ребра графа хоча б один з кінців входить в вершину з S.  Задача вершинного покриттяполягає в пошуку вершинного покриття найменшого розміру для заданого графа (цей розмір називається числом вершинного покриття графа).  На вході: Граф G = (V, E).  Результат: множина C ⊆ V - найменше вершинне покриття графа G.    Застосування:   * розміщення пунктів обслуговування; * призначення екіпажів на транспорт; * проектування інтегральних схем і конвеєрних ліній. |
| 5 | **Задача про кліку** (300 вершин, степінь вершини не більше 30, але не менше 2)**.** Клікою в неорієнтованому графі називається підмножина вершин, кожні дві з яких з'єднані ребром графа. Іншими словами, це повний підграф первісного графа. Розмір кліки визначається як число вершин в ній.  Задача про кліку існує у двох варіантах: у **задачі розпізнавання** потрібно визначити, чи існує в заданому графі G кліка розміру k, тоді як в **обчислювальному варіанті** потрібно знайти в заданому графі G кліку максимального розміру або всі максимальні кліки (такі, що не можна збільшити).  Застосування:   * біоінформатика; * електротехніка; |
| 6 | **Задача про найкоротший шлях** (300 вершин, відстань між вершинами випадкова від 5 до 150, степінь вершини не більше 10, але не менше 1) - задача пошуку найкоротшого шляху (ланцюга) між двома точками (вершинами) на графі, в якій мінімізується сума ваг ребер, що складають шлях.  Важливість задачі визначається її різними практичними застосуваннями. Наприклад, в GPS-навігаторах здійснюється пошук найкоротшого шляху між точкою відправлення і точкою призначення. Як вершин виступають перехрестя, а дороги є ребрами, які лежать між ними. Якщо сума довжин доріг між перехрестями мінімальна, тоді знайдений шлях найкоротший. |

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). |
| 2 | **Мурашиний алгоритм**:   * α; * β; * ρ; * Lmin; * кількість мурах М і їх типи (елітні, тощо…); * маршрути з однієї чи різних вершин. |
| 3 | **Бджолиний алгоритм:**   * кількість ділянок; * кількість бджіл (фуражирів і розвідників). |

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

Крок 1. Усі відстані між містами наведено в матриці, яка називається weights, тому кожна діагональна клітинка дорівнює 0, а ij-клітина — це відстань між містами i та j.

Крок 2. Для створення початкової популяції ми встановлюємо її розмір pop\_size та кількість поколінь (gen\_max) У методі generate\_population ми генеруємо початкову сукупність випадковим чином.

Крок 3. Кросовер. Ми беремо двох випадкових батьків (шляхи) і застосовуємо для них метод кросовера. кросовер (само, мати, батько) Кількість хромосом, взятих від матері та батька, генерується випадковим чином.

Крок 4. Мутація. Кожна дитина після схрещування піддається мутації - обмін двох хромосом в гені.

Крок 5. Придатність. Ми вимірюємо придатність кожного покоління, вимірюючи довжину запропонованого шляху. На кожній ітерації роботи алгоритму ми шукаємо найоптимальніший шлях

Крок 6. Вибір методу відбору. Відсортуйте нове покоління та попереднє покоління за допомогою функції пристосованості та виділяє найкращих представників кожного покоління та об'єднує їх, оновлюючи масив популяцій.

У циклі ми повторюємо кроки 3, 4, 5 і 6 1000 разів.

## Програмна реалізація алгоритму

https://github.com/Mishanych/Algorithms/tree/main/laba4

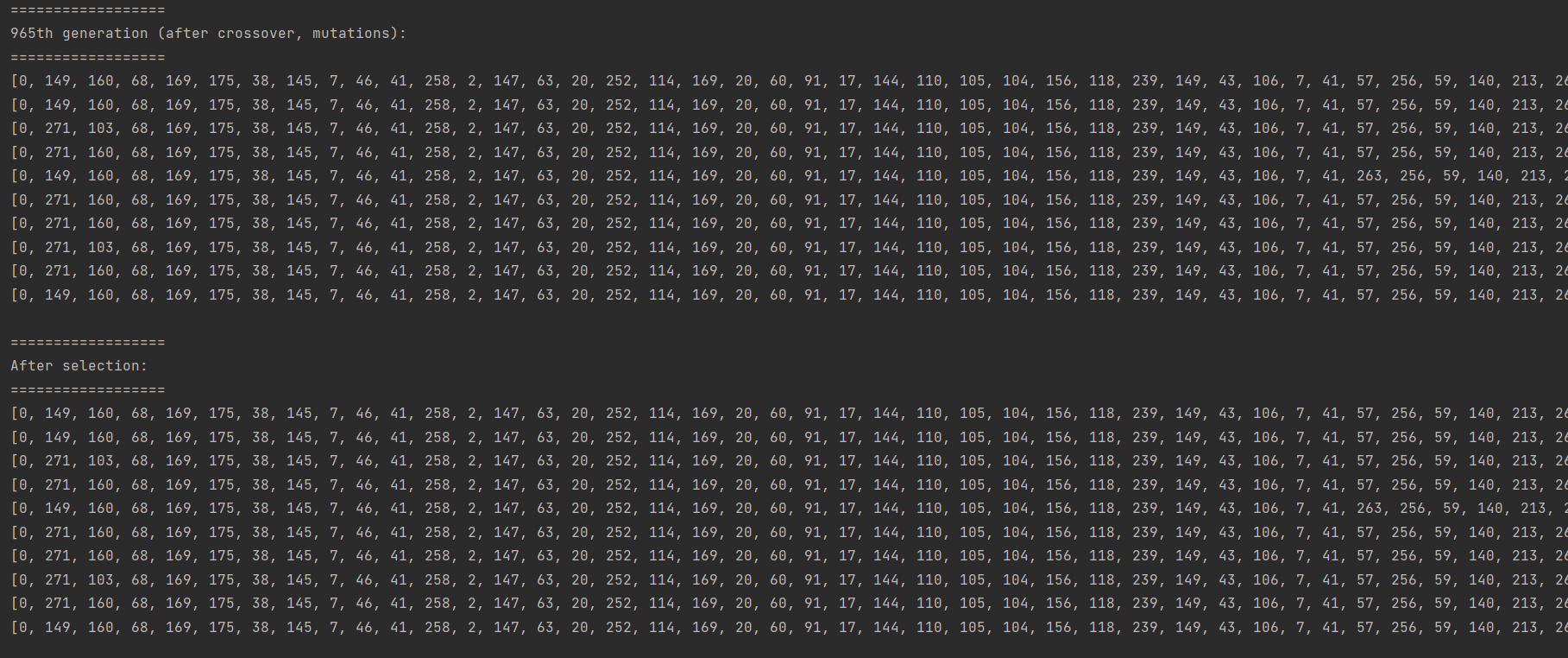
### Вихідний код

import random  
import functools  
from сhromosome import Chromosome  
  
  
number\_of\_vertex = 300  
chromosome\_length = number\_of\_vertex  
  
weights = [[0] \* number\_of\_vertex] \* number\_of\_vertex  
  
quite = False  
  
  
class GeneNetwork(object):  
 def \_\_init\_\_(self, dim, weights, chromosome\_length, source, destination):  
 *"""* ***:param*** *dim: number of vertex* ***:param*** *weights:* ***:param*** *chromosome\_length: length of chromosome* ***:param*** *source: source node* ***:param*** *destination: destination node* ***:return****:  
 """* if source >= dim or destination >= dim:  
 raise ValueError  
 self.chromosome\_length = chromosome\_length  
 self.dim = dim  
 self.weights = weights  
 self.source = source  
 self.destination = destination  
 self.population = []  
 self.population\_size = 0  
 self.results = []  
 self.best = None  
  
 def start(self, gen\_max, pop\_size):  
 *"""* ***:param*** *gen\_max: maximum number of generations* ***:param*** *pop\_size: initial population size* ***:return****: best solution found  
 """* gen = 1 # from first generation  
 self.generate\_population(pop\_size) # generate initial population  
 self.population\_size = pop\_size  
 if not quite:  
 pretty\_print('Initital:')  
 self.print\_chromosomes(self.population)  
  
 while gen <= gen\_max:  
 gen += 1  
 p = 1  
 new\_population = list()  
 while p <= self.population\_size:  
 p += 1  
 parents = random.sample(range(self.population\_size), 2)  
 newbie = self.crossover(self.population[parents[0]], self.population[parents[1]])  
 newbie.mutate()  
 fit = self.fitness(newbie)  
 self.results.append((newbie, fit))  
 new\_population.append(newbie)  
 if self.best is None or self.best[1] > fit:  
 self.best = (newbie, fit)  
 if not quite:  
 pretty\_print('%dth generation (after crossover, mutations): ' % gen)  
 self.print\_chromosomes(new\_population)  
 self.selection(self.population, new\_population)  
 if not quite:  
 pretty\_print('After selection: ')  
 self.print\_chromosomes(new\_population)  
 return self.best  
  
 def selection(self, prev, now):  
 *"""* ***:param*** *prev: previous generation* ***:param*** *now: new generation* ***:return****:  
 """* prev.extend(now)  
 keyfunc = functools.cmp\_to\_key(lambda x, y: (self.fitness(x) - self.fitness(y)))  
 prev.sort(key=keyfunc)  
 self.population = prev[:self.population\_size]  
  
 def generate\_population(self, n):  
 *"""* ***:param*** *n: number of chromosomes* ***:return****:  
 """* chromosomes = list()  
 for i in range(n):  
 chromosomes.append(self.\_gen\_chromosome())  
 self.population = chromosomes  
  
 def \_gen\_chromosome(self):  
 *"""* ***:return****: random path from start to destination  
 """* chromosome = random.sample(list(set(range(self.dim)) - {self.source, self.destination}),  
 self.chromosome\_length - 2)  
 chromosome.insert(0, self.source)  
 chromosome.append(self.destination)  
 return Chromosome(chromosome)  
  
 def crossover(self, mother, father):  
 *"""* ***:param*** *mother: first parent* ***:param*** *father: second parent* ***:return****: crossing over child  
 """* mother\_list = mother.get()  
 father\_list = father.get()  
 cut = random.randint(0, self.chromosome\_length - 1)  
 child = mother\_list[0:cut] + father\_list[cut:]  
 return Chromosome(child)  
  
 def fitness(self, chromosome):  
 chromosome\_list = chromosome.get()  
 return sum([self.weights[i][j] for i, j in zip(chromosome\_list[:-1], chromosome\_list[1:])])  
  
 def print\_chromosomes(self, chromosomes):  
 for chromosome in chromosomes:  
 print(str(chromosome) + ' ' + str(self.fitness(chromosome)))  
  
  
def pretty\_print(to\_print, hint=''):  
 print('')  
 print('==================')  
 print(hint + str(to\_print))  
 print('==================')  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
  
 weights = [[random.randint(5, 150) for x in range(0, number\_of\_vertex)] for y in range(0, number\_of\_vertex)]  
  
 for i in range(0, number\_of\_vertex):  
 for j in range(0, number\_of\_vertex):  
 if i == j:  
 weights[i][j] = 0  
 gene\_network = GeneNetwork(number\_of\_vertex, weights, chromosome\_length, 0, 299)  
 res = gene\_network.start(1000, 10) # start with 1000 generations and 10 initial chromosomes  
 pretty\_print(res, 'Solution: ')

import random  
  
  
class Chromosome(object):  
 def \_\_init\_\_(self, vector):  
 self.vector = vector  
  
 def \_\_repr\_\_(self):  
 return str(self.vector)  
  
 def get(self):  
 return self.vector  
  
 def size(self):  
 return len(self.vector)  
  
 def path(self):  
 return sum(self.vector)  
  
 def mutate(self):  
 pos = random.randint(1, self.size() - 2)  
 self.vector[pos] = random.randint(0, self.size() - 1)

### Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми.



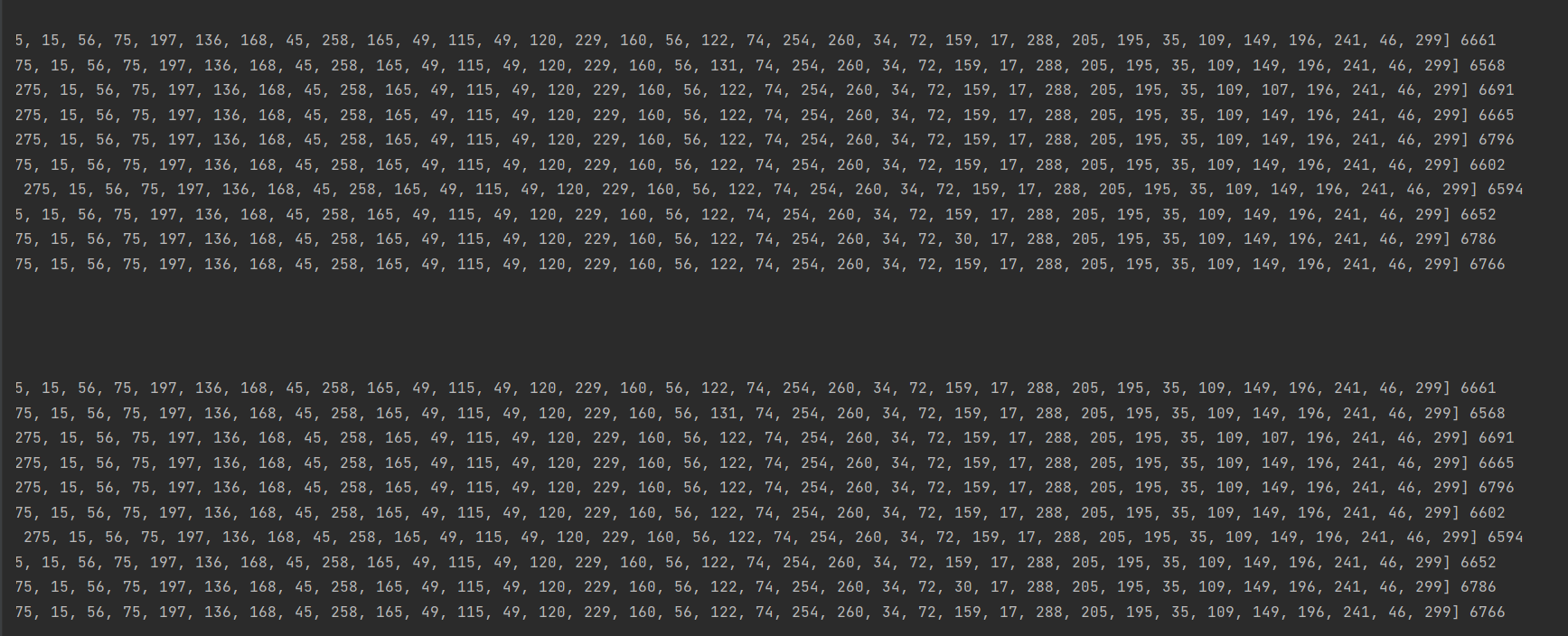
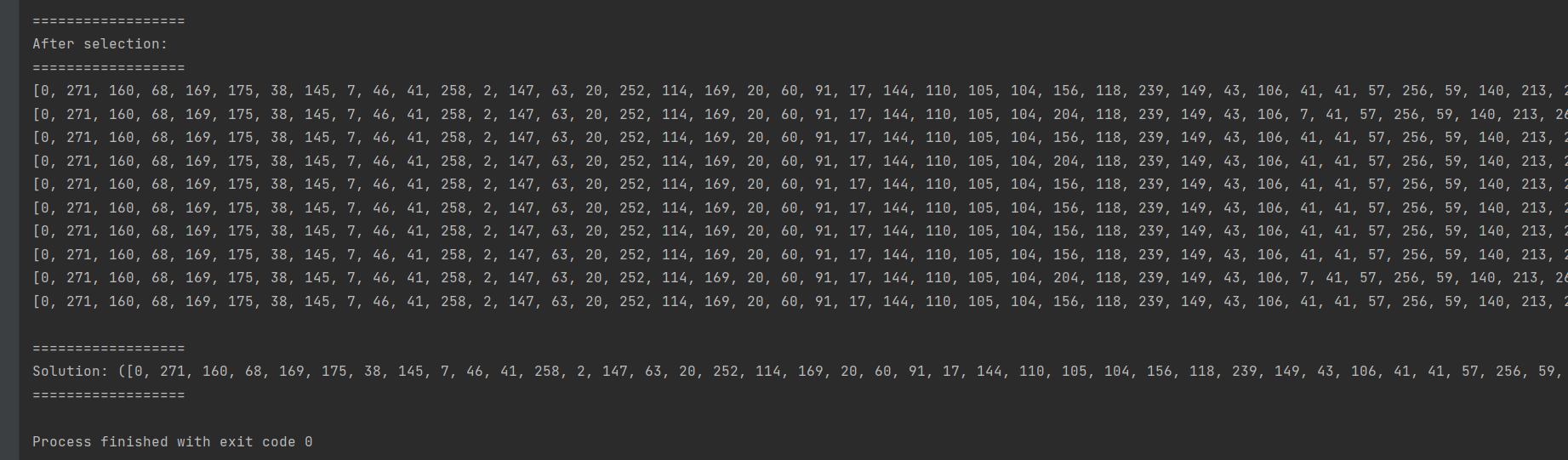


Рисунок 3.1 –



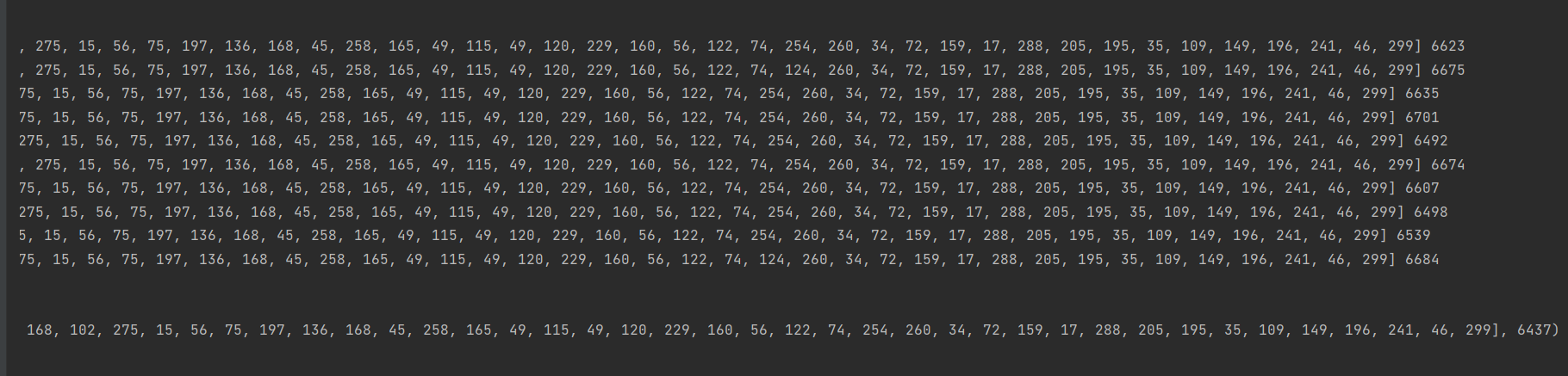


Рисунок 3.2 –

## Тестування алгоритму

***Залежність результату від кількості ітерацій та початкової популяції***

|  |  |  |
| --- | --- | --- |
| Кількість ітерацій | Кількість початкової популяції | Найкоротший шлях |
| 10 | 10 | 20481 |
|  | 20 | 20063 |
|  | 50 | 19733 |
| 100 | 10 | 15681 |
|  | 20 | 14483 |
|  | 50 | 12425 |
| 1000 | 10 | 6437 |
|  | 20 | 5445 |
|  | 50 | 4231 |

За цією таблицею ми можемо помітити, що результат більше залежить від зроблених ітерацій, тобто кросоверів, мутацій, вираховування пристосування, аніж від початкової популяції.

Висновок

В рамках даної лабораторної роботи я вивчив основні підходи розробки метаеврестичних алгоритмів для типових прикладних задач, опрацював методологію підбору прийнятних параметрів алгоритму.

Критерії оцінювання

При здачі лабораторної роботи до 26.11.2021 включно максимальний бал дорівнює – 5. Після 26.11.2021 максимальний бал дорівнює – 1.

Критерії оцінювання у відсотках від максимального балу:

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