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) _____*ІП-*_____ _____
(шифр, прізвище, ім'я, по батькові)

Перевірив _____*Головченко М.Н.*_____ _____
(прізвище, ім'я, по батькові)

Київ 2022

ЗМІСТ

1 МЕТА ЛАБОРАТОРНОЇ РОБОТИ 3

2 ЗАВДАННЯ 4

3 ВИКОНАННЯ 11

3.1 Покроковий алгоритм 11

3.2 Програмна реалізація алгоритму 11

3.2.1 Вихідний код 11

3.2.2 Приклади роботи 15

3.3 Тестування алгоритму 19

ВИСНОВОК 23

КРИТЕРІЇ ОЦІНЮВАННЯ 28

1 МЕТА ЛАБОРАТОРНОЇ РОБОТИ

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

2 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.

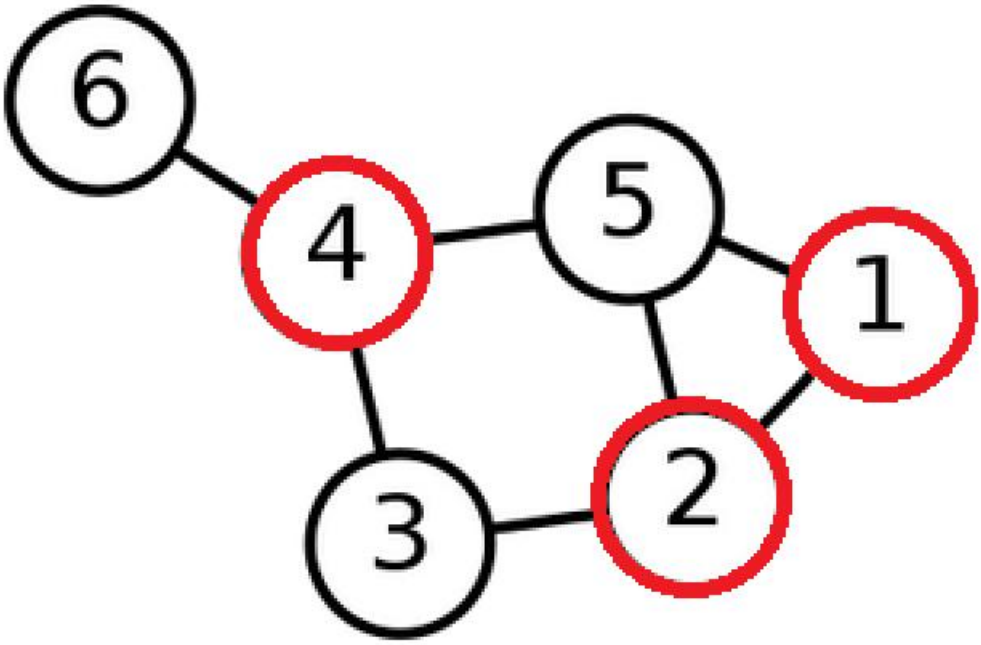
Зробити узагальнений висновок в якому обов’язково описати залежність якості розв’язку від вхідних параметрів.

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 покриття полягає 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 M 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

3 ВИКОНАННЯ

3.1 Покроковий алгоритм

1) Створюємо початкову популяцію з використанням функції generate_population(), яка приймає на вхід граф, розмір популяції та максимальну кількість кольорів.
2) Проходимо цикл заданої кількості поколінь (generation_amount).
3) Вибираємо родителів з використанням функції select_parents_proportional() та зберігаємо їх у змінній parents.
4) Створюємо пустий список children та виконуємо функцію crossover_function() для кожної пари родителів, результат записуємо у список children.
5) Проходимо цикл по всіх елементах списку children та виконуємо над ними функцію mutation_function().
6) Об'єднуємо списки parents та children та зберігаємо результат у змінній population.
7) Виконуємо функцію local_improvement_function() для найкращого елемента з популяції та графу та повертаємо результат.

3.2 Програмна реалізація алгоритму

3.2.1 Вихідний код

```
import random

import networkx as nx
import matplotlib.pyplot as plt
```

```
import numpy as np
```

```
def generate_random_adjacency_matrix(n, min, max):
```

```
    adjacency_matrix = [ [0 for i in range(n)] for j in range(n) ]

    for i in range(n):
        degree = 0
        while degree < 2 or degree > max:
```

```
            for j in range(n):
                if i != j:
                    r = np.random.random()
                    if r < 0.5 and degree < 2 :
```

```
                        adjacency_matrix[i][j] = 1
                        adjacency_matrix[j][i] = 1
                        degree += 1
                    elif r < 0.5 and degree < max:
                        adjacency_matrix[i][j] = 1
                        adjacency_matrix[j][i] = 1
                        degree += 1
    return adjacency_matrix
```

```
def generate_population(graph, size, n_colors):
    population = []
    for i in range(size):
        # Create a random coloring of the graph
        individual = [random.randint(0,n_colors) for i in range(len(graph[0]))]
```

```
        while(count_color_conflicts(individual, graph) != 0):
            individual = [random.randint(0,n_colors) for i in range(len(graph[0]))]
```

```
        population.append(individual)
    return population
```

```
def max_colors(graph):
```

```
    max_degree = 0
```

```
    for i in range(len(graph[0])):
        degree = 0
        for j in range(len(graph[0])):
            if graph[i][j] == 1:
                degree +=1

        if degree > max_degree:
            max_degree = degree
```

```
    return max_degree + 1
```

```
def select_parents_propotional(population):
    parents = []
    for i in range(2):
        r = random.random()
        total = 0
        for individual in population:
            total += individual[1]
            if total > r:
                parents.append(individual)
                break
    return parents
```

```
def evaluate_fitness(individual, graph):
```

```
    conflicts = count_color_conflicts(individual, graph)
```

```
    if conflicts == 0:
        return 1 + 1/count_colors(individual)
```

```
    return 1/conflicts + 1/count_colors(individual)
```

```
def count_colors(individual):
    return len(set(individual))
```

```
def count_color_conflicts(individual, graph):
    conflicts = 0
    for i in range(len(graph[0])):
```

```
        for j in range(len(graph[0])):
            if graph[i][j] == 1 and individual[i] == individual[j]:
                conflicts += 1
    return conflicts
```

```
def one_point_cross_over(parent1, parent2):
    crossover_point = random.randint(1, len(parent1)-1)
```

```
    child1 = parent1[:crossover_point] + parent2[crossover_point:]
```

```
    return child1

def two_point_cross_over(parent1, parent2):
    crossover_point1 = random.randint(1, len(parent1)-2)
    crossover_point2 = random.randint(crossover_point1+1, len(parent1)-1)
```

```
    child1 = parent1[:crossover_point1] + parent2[crossover_point1:crossover_point2] +
parent1[crossover_point2:]

    return child1
```

```
def even_crossover(parent1, parent2):
    child1 = []
    for i in range(len(parent1)):
        if i % 2 == 0:
            child1.append(parent1[i])
        else:
            child1.append(parent2[i])
    return child1
```

```
def probability_mutation(individual, probability=0.1):
    for i in range(len(individual)):
        if random.random() < probability:
            individual[i] = random.randint(0, max(individual))
```

```
def swap_mutation(individual):
    index1 = random.randint(0, len(individual)-1)
    index2 = random.randint(0, len(individual)-1)
    individual[index1], individual[index2] = individual[index2], individual[index1]
```

```
def draw_graph(graph):
    G = nx.Graph()
    for i in range(len(graph)):
        G.add_node(i)
        for j in range( len(graph)-i):
            if graph[j][i] == 1:
                G.add_edge(j, i)
```

```
    nx.draw(G, with_labels=True)
    plt.show()
```

```
def draw_graph_with_colors(graph, colors):
```

```
    G = nx.Graph()
    for i in range(len(graph)):
        G.add_node(i, color=ids_to_colors(colors)[i])
        for j in range( len(graph)-i):
            if graph[j][i] == 1:
                G.add_edge(j, i)
```

```
    nx.draw(G, with_labels=True, node_color=[G.nodes[i]['color'] for i in G.nodes])
    plt.show()
```

```
def genetic_algorithm(graph, population_size, generation_amount, max_colors,
crossover_function,  mutation_function, local_improvement_function):
    population = generate_population(graph, population_size, max_colors)
```

```
    for i in range(generation_amount):
        population = sorted(population, key=lambda x: evaluate_fitness(x, graph), reverse=True)
```

```
        for i in range(population_size):
            parents = select_parents_propotional(population)
            child1= crossover_function(parents[0], parents[1])
            mutation_function(child1)
```

```
            if evaluate_fitness(child1, graph) > evaluate_fitness(population[-1], graph):
                population[-1] = child1
                population = sorted(population, key=lambda x: evaluate_fitness(x, graph), reverse=True)
```

```
    population = sorted(population, key=lambda x: evaluate_fitness(x, graph), reverse=True)
    return population[0]
```

```
def local_improvement_random(individual, graph):
    for i in range(len(individual)):
        individual[i] = random.randint(0, max_colors)
        if count_color_conflicts(individual, graph) == 0:
            return individual
    return individual
```

```
def local_improvement_hill_climbing(individual, graph):
    ind = individual.copy()
    for i in range(len(ind)):
        for j in range(1, max_colors):
            ind[i] = j
            if count_color_conflicts(ind, graph) == 0:
                return ind
    return individual
```

```
def ids_to_colors(ids):
    colors = []
    for id in ids:
        colors.append(id_to_color(id))
    return colors
```

```
def id_to_color(id):
    if id == 0:
        return 'red'
    elif id == 1:
        return 'green'
    elif id == 2:
        return 'blue'
    elif id == 3:
        return 'yellow'
    elif id == 4:
        return 'orange'
    elif id == 5:
        return 'purple'
    elif id == 6:
        return 'pink'
    elif id == 7:
        return 'aqua'
    elif id == 8:
        return 'brown'
```

```
    elif id == 9:
        return 'grey'
    elif id == 10:
        return 'cyan'
    elif id == 11:
        return 'magenta'
    elif id == 12:
        return 'lime'
    elif id == 13:
        return 'olive'
    elif id == 14:
        return 'teal'
    elif id == 15:
        return 'coral'
    elif id == 16:
        return 'gold'
    elif id == 17:
        return 'khaki'
    elif id == 18:
        return 'maroon'
    elif id == 19:
        return 'navy'
    elif id == 20:
        return 'plum'
    elif id == 21:
        return 'salmon'
    elif id == 22:
        return 'sienna'
    elif id == 23:
        return 'tan'
    elif id == 24:
        return 'violet'
    elif id == 25:
        return 'wheat'
```

```
graph = generate_random_adjacency_matrix(8, 2, 30)
g1= graph.copy()
```

```
population_size = 50
generation_amount = 50
max_colors = max_colors(graph)
```

```
population = generate_population(graph, population_size, max_colors)
```

```
solution = genetic_algorithm(graph, population_size, generation_amount, max_colors, even_crossover,
swap_mutation, local_improvement_hill_climbing)
```

```
draw_graph(graph)
draw_graph_with_colors(graph, solution)
```

```
print(count_color_conflicts(solution, graph))
```

3.2.2 Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми.

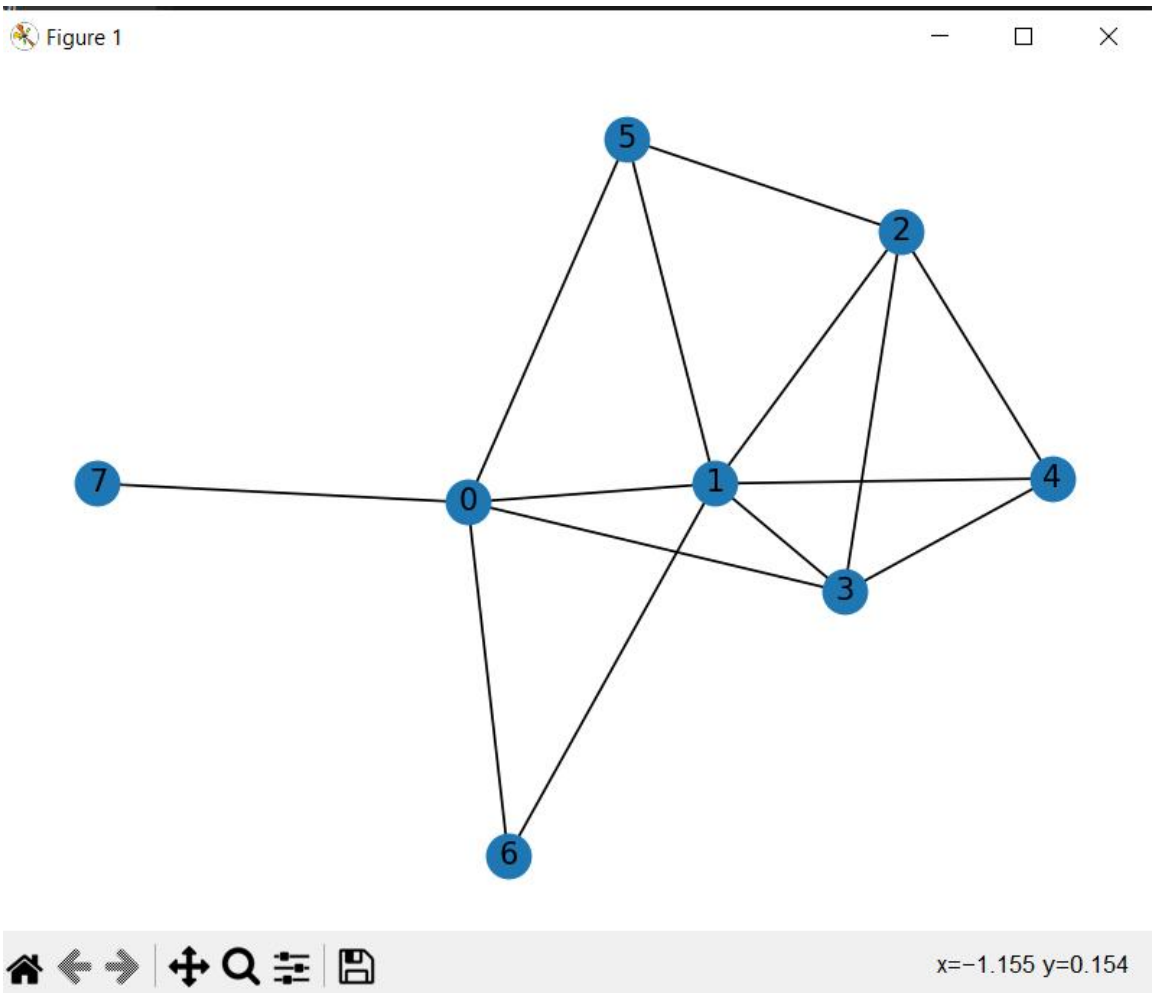


Рисунок 3.1 – випадково сгенерований граф на 8 вузлів

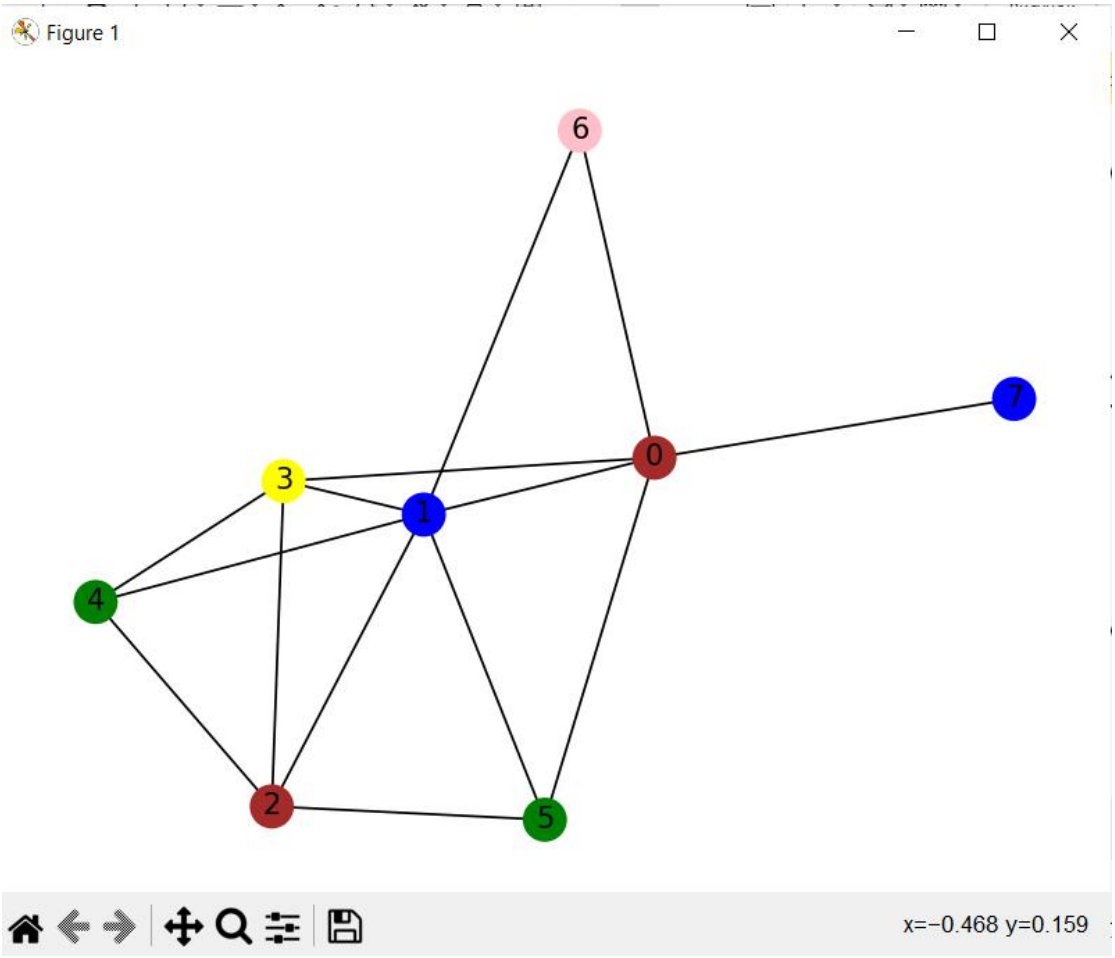


Рисунок 3.2 – Найкращий потомок після 300 поколінь(50 - розмір покоління).

1 точковий кросовер, мутація обміном, локальне покращення “hill climbing“

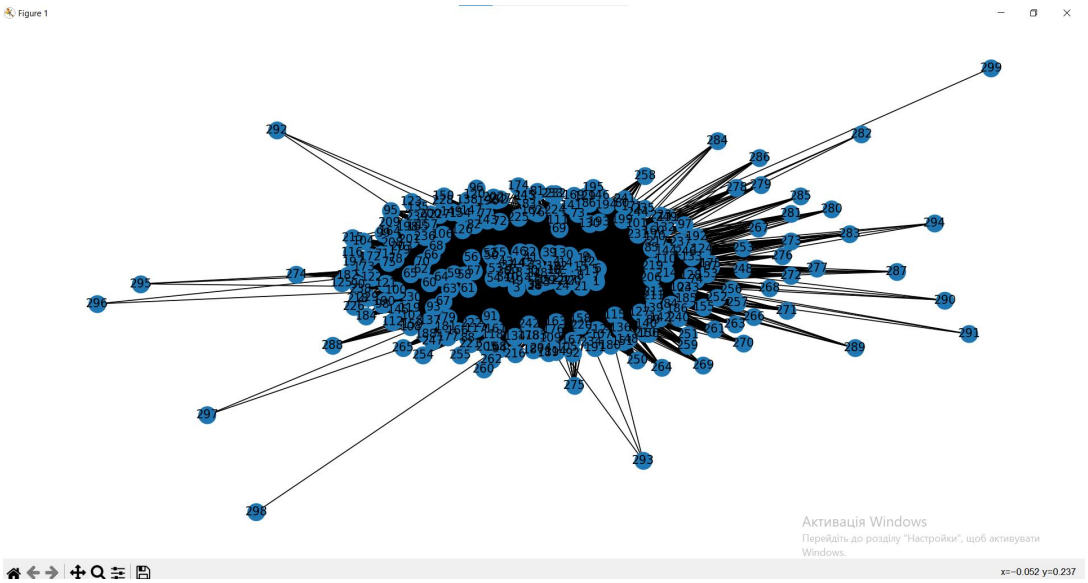


Рисунок 3.3 – випадково сгенерований граф на 300 вузлів

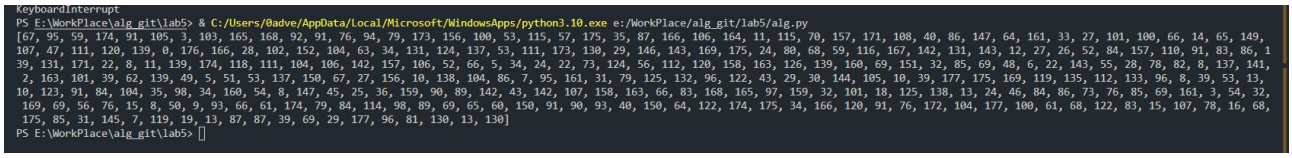


Рисунок 3.4 – Найкращий потомок після 50 поколінь(20 - розмір покоління). Використані оптимальні параметри

Тестування алгоритму

Генетичний алгоритм:

Маємо протестувати наступні парамерти:

1. Функції схрещування
    a) Одноточкове
    b) Двоточкове
    c) Рівномірне
2. Функції мутації
    a) Випадкова мутація
    b) Мутація обміном
3. Фунції локалького покращення
    a) Випадкове покращення
    b) ”Hill climbing”

3.3.1. Функції схрещування

Для порівняння методів схрещування запустимо алгоритм 3 рази
З одним і тим же набором початкових генів. Кожні 20 ітерацій будемо звмірювати середню вартість популяції.

Змінюється:

crossover = one_point_crossover, two_point_crossover, even_crossover

Постійні змінні:

mutation = probability_mutation

local_improvement = local_improvement_hill_climbing

Табл 1 - Порівняння методів схрещування

Номер покоління	Одноточкове схрещування (середня оцінка популяції)	Двоточкове схрещування (середня оцінка популяції)	Рівномірне схрещування (середня оцінка популяції)
1	0.02117511585	0.01928946142	0.0200007895
20	0.03746579933	0.03700015586	0.0367526202
40	0.04414546949	0.04775926546	0.03983310139
60	0.04957750011	0.05582392412	0.042148658
80	0.05777399908	0.0637380473	0.04966024075
100	0.06315470517	0.06765284524	0.0509182496
120	0.06608055058	0.0887657593	0.05545712201
140	0.0667937638	0.1039477249	0.06295127611
160	0.07065504436	0.1101864956	0.1023626263
180	0.07298926569	0.1101864956	0.1142949139
200	0.07485237055	0.1119038693	0.1442529663

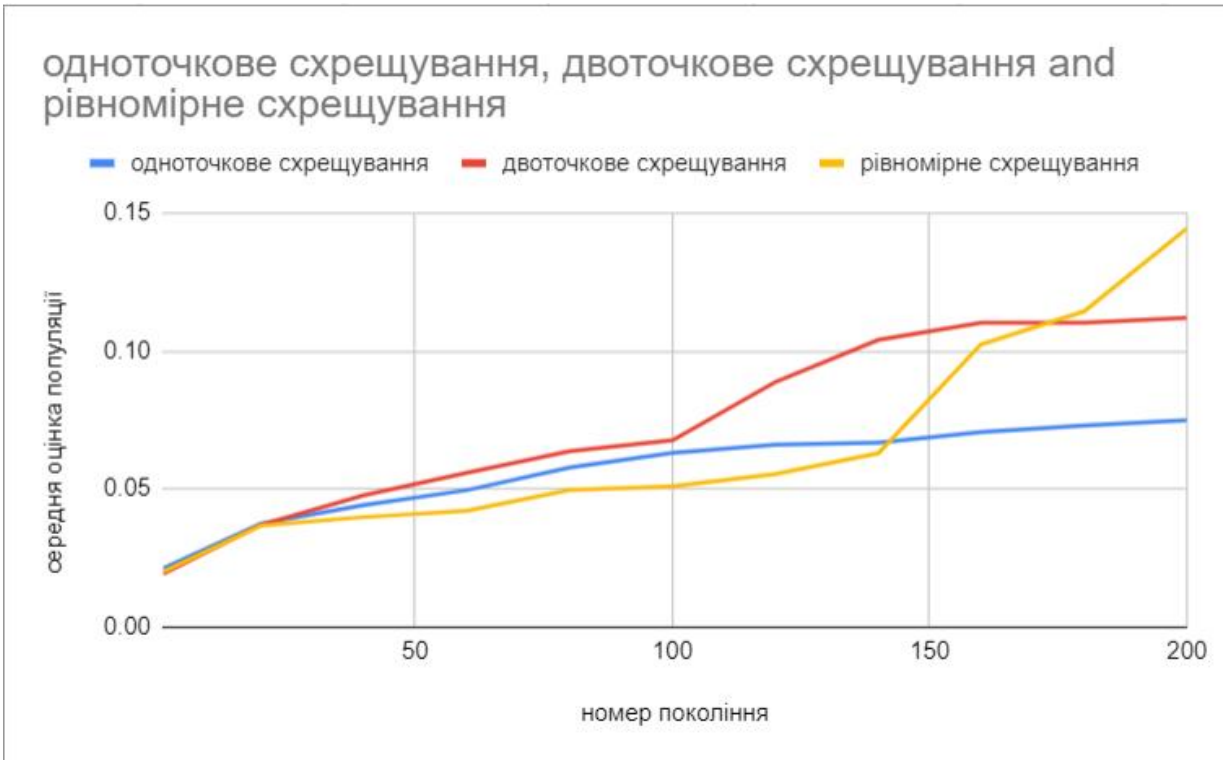


Рисунок 3.3 – Оцінка функцій схрещування

Бачимо, що на початку двоточкове схрещування “перемагає”, але ближче до 200го покоління рівномірне схрещування набуває значної переваги.

3.3.2. Функції мутації

Для порівняння методів мутації використаємо то й же метод що і для порівняння методів схрещування.

Змінюється:

mutation = probability_mutation, swap_mutation

Постійні змінні:

crossover = even_crossover

local_improvement = local_improvement_hill_climbing

Табл 2 - Порівняння методів мутації

Номер покоління	Випадкова мутація (середня оцінка популяції)	Мутація обміном (середня оцінка популяції)
1	0.02172088892	0.02288926799
20	0.03552702633	0.04199300699
40	0.04215982482	0.06254856255
60	0.05075869731	0.131993007
80	0.06132168297	0.1736596737
100	0.06992212908	0.506993007
120	0.07377235638	0.506993007
140	0.07470301088	0.506993007
160	0.07731677607	1.006993007
180	0.07731677607	1.006993007
200	0.07941773773	1.006993007

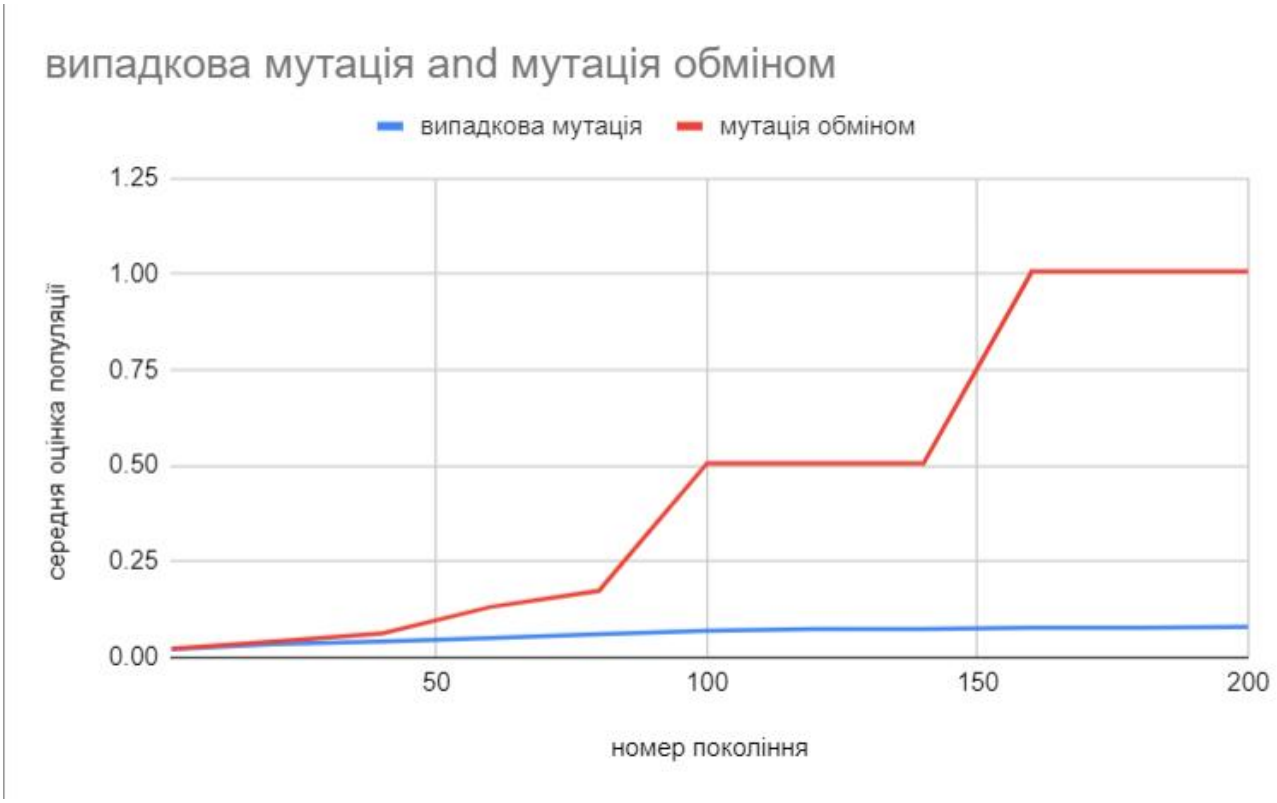


Рисунок 3.4 – Оцінка функцій мутації

Очевидно, що метод мутації обміном, має значну перевагу.

3.3.3. Функції локального покращення

Для порівняння методів локального покращення використаємо то й же метод що і для порівняння методів схрещування і мутації.

Змінюється:

local_improvement = local_improvement_hill_climbing,
local_improvement_random

Постійні змінні:

crossover = even_crossover

mutation = swap_mutation

Табл 3 - Порівняння методів локального покращення

Номер покоління	Локальне покращення “hill climbing” (середня оцінка популяції)	Випадкове локальне покращення (середня оцінка популяції)
1	0.0200405032	0.02009428457
20	0.04521829522	0.04718715969
40	0.1142567568	0.06925675676
60	0.5067567568	0.09675675676
80	1.006756757	0.5067567568
100	1.006756757	0.5067567568
120	1.006756757	1.006756757
140	1.006756757	1.006756757
160	1.006756757	1.006756757
180	1.006756757	1.006756757
200	1.006756757	1.006756757

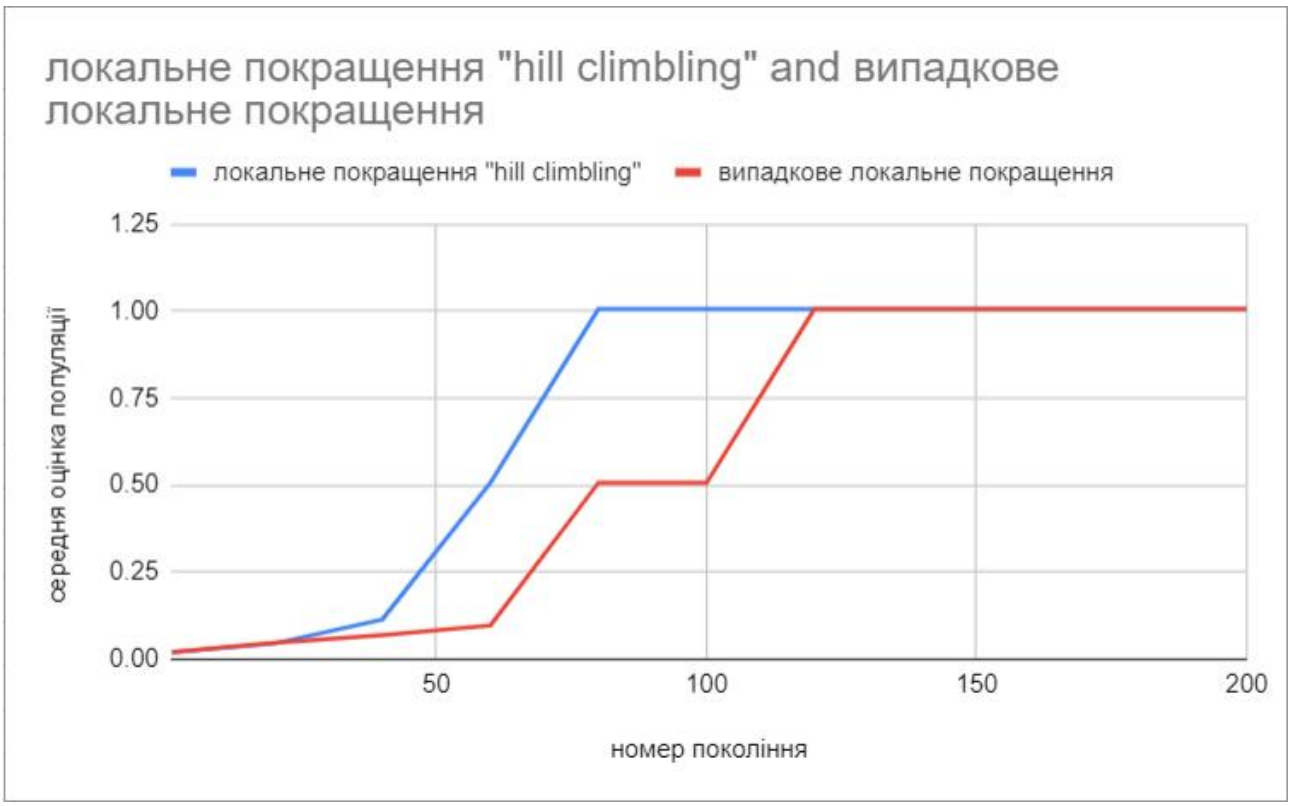


Рисунок 3.5 – Оцінка функцій локального покращення

Маємо досить схожі резульати, але локальне покращення “hill climbing” показує швидший приріст, тому воно є фаворитом.

ВИСНОВОК

При виконанні даної лабораторної роботи було вивчено основні підходи розробки метаевристичних алгоритмів для типових прикладних задач. Опрацьовано методологію підбору прийнятних параметрів алгоритму. Проведено аналіз ефективності використання алгоритма. Викоритсано евристичну функцію. Було помічено, що для генетивчного алгоритму опитимальний набів використаних методів такий :

- рівномірне схрещування
- мутація обміном
- локальне покращення “hill climbing”

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%.