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-12 Бобрик Максим Геннадійович* _____
(шифр, прізвище, ім'я, по батькові)

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

Київ 2022

ЗМІСТ

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.

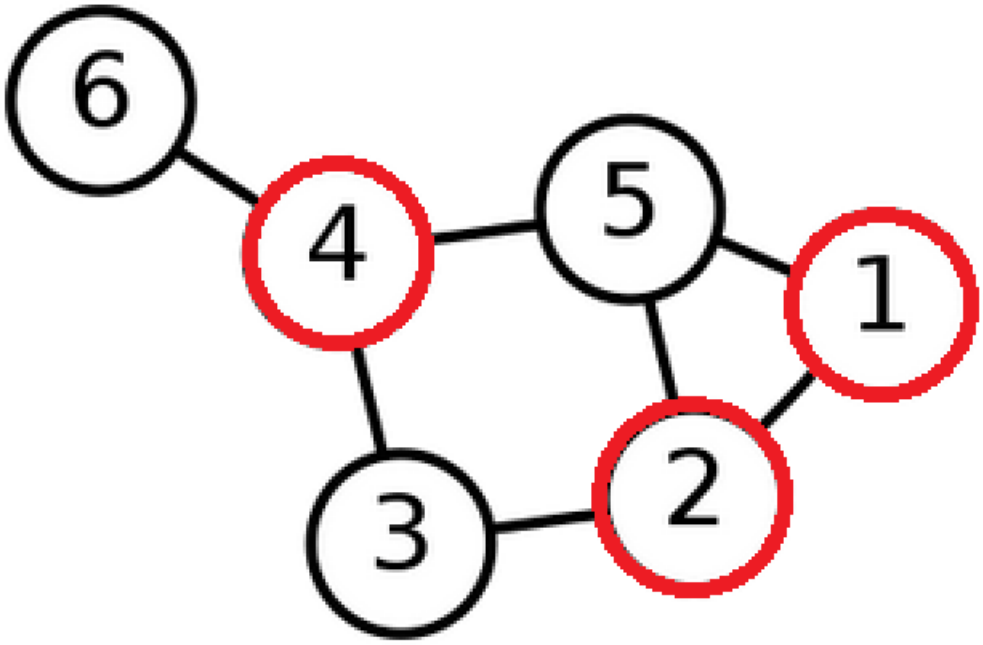
Зробити узагальнений висновок в якому обов’язково описати залежність якості розв’язку від вхідних параметрів.

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. Генерація симетричної матриці суміжность
2. Генерація популяції заданого розміру
3. Кросоверінг випадково обраних індувідумів (випадково обирається точка розриву хромосом у батьків, з двох частин хромосом двох батьків «зклеюється» хромосома нащадка, перевіряючи кожний ген на унікальність)

4. Мутація випадково обраних індувідумів (з випадково обраних індувідумів створюються нащадки, за алгоритмом описаним в п.3, в нащадках випадковим чином зміняються місцями два гени) та їх локальне покращення
5. Сортування популяції за цінністю індувідума та видалення останніх зайвих(для збереження розміру популяції)
6. Пункти 3-5 відбуваються в циклі задану кількість раз (1 ітерація – 1 покоління)

3.2 Програмна реалізація алгоритму

3.2.1 Вихідний код

```
import matplotlib.pyplot as plt
import numpy as np
from random import random, sample


class Individual:
    def __init__(self, path, matrix):
        self.path = path
        self.matrix = matrix
        self.fitness = self.calculate_fitness()

    def calculate_fitness(self):
        fitness = 0
        index = 0
        for i in range(len(self.path) - 1):
            fitness += self.matrix[index][self.path[i + 1]]
            index = self.path[i]
        return fitness

    def __lt__(self, other):
```

```
        return self.fitness < other.fitness

    def __gt__(self, other):
        return self.fitness > other.fitness

    def __getitem__(self, offset):
        return self.path[offset]

    def __len__(self):
        return len(self.path)

    def __getslice__(self, low, high):
        return Individual(self.path[low:high], self.matrix)


class Graph:
    def __init__(self, size):
        self.matrix = self.generate_matrix(size)
        self.print_matrix(self.matrix)

    @staticmethod
    def generate_matrix(size):
        matrix = []
        for _ in range(size):
            matrix.append([0 for _ in range(size)])

        for i in range(size):
            for j in range(size):
                if i >= j:
                    if i == j:
                        matrix[i][j] = float('inf')
                    else:
                        matrix[i][j] = round(random() * 149 + 5)
                        matrix[j][i] = matrix[i][j]
        return matrix

    @staticmethod
    def print_matrix(graph):
        print('Graph adjacency matrix:')
        print('\n'.join([''.join(['{:6}'.format(item) for item in row])
                         for row in graph]))
        print()

    @staticmethod
    def fill_with_infs(matrix, size):
        for i in range(size):
            matrix.append([])

        for i in matrix:
            for j in range(size):
                i.append(float('inf'))


class GeneticAlgorithm:
    def __init__(self, size):
        self.graph size = size
        self.graph = Graph(size)

    def launch_genetic_selection(self, iterations, population_size,
crossovers_number, mutations_number, mutation_probability):
        population = self.generate_population(population_size)
        best_results = [population[0].fitness]
        for i in range(iterations):
            self.generate_crossover(crossovers_number, population)
```

```
            self.mutations(mutations_number, mutation_probability, population)
            population.sort()
            population = population[:population_size]
            best_results.append(population[0].fitness)
            if i % 10 == 0:
                print(f'Current best route length: {population[0].fitness}')
        return population, best_results

    def generate_crossover(self, crossovers_number, population):
        for j in range(crossovers_number):
            childs = self.crossover(population)
            population.append(Individual(self.local_improvement(childs[0]),
self.graph.matrix))
            population.append(Individual(self.local_improvement(childs[1]),
self.graph.matrix))

    def mutations(self, mutations_number, mutation_probability, population):
        for j in range(mutations_number):
            childs = self.crossover(population)
            mutants = [self.try_mutation(mutation_probability, childs[0]),
                       self.try_mutation(mutation_probability, childs[1])]
            if mutants[0] is not None:
                population.append(Individual(self.local_improvement(mutants[0]),
self.graph.matrix))
            elif childs[0] is not None:
                population.append(Individual(self.local_improvement(childs[0]),
self.graph.matrix))
            if mutants[1] is not None:
                population.append(Individual(self.local_improvement(mutants[1]),
self.graph.matrix))
            elif childs[1] is not None:
                population.append(Individual(self.local_improvement(childs[1]),
self.graph.matrix))

    def generate_population(self, population_size):
        population = []
        for i in range(population_size - 1):
            population.append(Individual(self.create_random_path(),
self.graph.matrix))
        return population

    def crossover(self, population):
        parents = sample(population, 2)
        breaking_point = round(random() * (len(parents[0]) - 1) + 1)
        first_child = parents[0][:breaking_point]
        self.add_chromosome_part(first_child, parents, breaking_point, 1)
        second_child = parents[1][:breaking_point]
        self.add_chromosome_part(second_child, parents, breaking_point, 0)
        return first_child, second_child

    def try_mutation(self, probability, child):
        if random() < probability:
            return self.start_mutation(child)

    def create_random_path(self):
        random_path = [0]
        while len(random_path) <= self.graph_size:
            if len(random_path) == self.graph_size:
                random_path.append(0)
            else:
                temp = round(random() * (self.graph_size - 1))
                if temp not in random_path:
                    random_path.append(temp)
        return random_path
```

```
    def add_chromosome_part(self, child, parents, breaking_point,
parent_number):
        i = breaking_point
        while i < len(parents[parent_number]):
            if parents[parent_number][i] not in child:
                child.append(parents[parent_number][i])
            i += 1

        if len(child) < len(parents[parent_number]) - 1:
            self.fill_till_end(child, parents, parent_number)
        child.append(0)

    def check_child(self, child):
        matrix = self.graph.matrix
        index = 0
        for i in range(len(child) - 1):
            if matrix[index][child[i + 1]] == float('inf'):
                return False
        return True

    def start_mutation(self, child):
        indexes = sample([*range(1, len(child))], 2)
        child[indexes[0]], child[indexes[1]] = child[indexes[1]],
child[indexes[0]]
        if self.check_child(child):
            return child

    @staticmethod
    def fill_till_end(child, parents, parent_number):
        if parent_number == 0:
            another_parent_number = 1
        else:
            another_parent_number = 0
        i = 0
        while i < len(parents[another_parent_number]):
            if parents[another_parent_number][i] not in child:
                child.append(parents[another_parent_number][i])
            i += 1

    @staticmethod
    def local_improvement(improved):
        i = round(random() * (len(improved) - 2) + 1)
        j = round(random() * (len(improved) - 2) + 1)
        improved[i], improved[j] = improved[j], improved[i]
        return improved


gr = GeneticAlgorithm(300)
results = gr.launch_genetic_selection(1000, 8, 8, 8, 0.2)
ypoints = np.array(results[1])
plt.plot(ypoints)
plt.show()
print(f'The best route length: {results[0][0].fitness}')
print(f'The best route: {results[0][0].path}')
```

3.2.2 Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми.

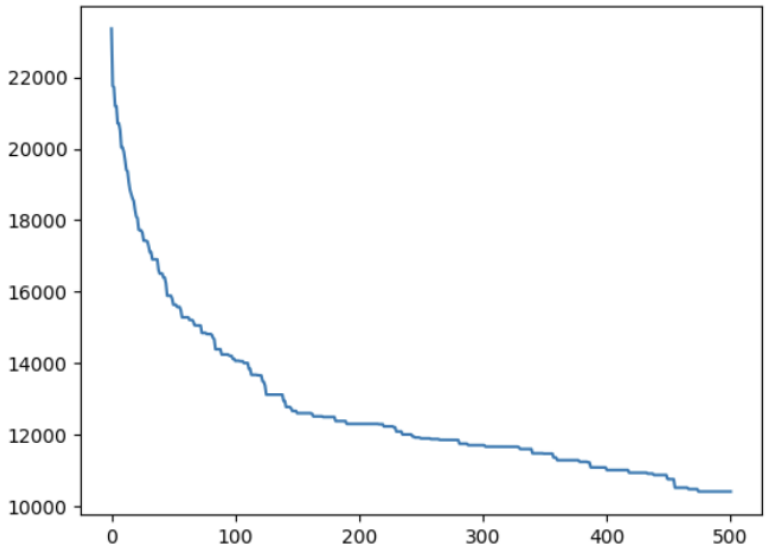


Рисунок 3.1 – Найкращі індувідуми в кожному поколінні

Тестування алгоритму

Зі збільшенням кількості поколінь будемо отримувати й кращі результати, тобто між цим параметром і результатом прямопропорційна залежність. Теж саме можно сказати і про розмір популяції, кількість схрещувань та мутацій. Тому ці параметри обираємо враховуючи показник продуктивності, тобто результат/час.

Кількість схрещувань	Результат
2	11516
4	11249
8	10913
16	10819

Як бачимо найоптимальнішою кількістю схрещувань є 8, так як далі цей показник майже не впливає на результат. Розмір популяції та мутацій також 8. Кількість поколінь – 1000.

Щодо коефіцієнту мутації:

Коефіцієнт мутації	Результат
0,2	10825
0,4	10744
0,6	10830
0,8	11005

Як бачимо після коефіцієнту мутації 0,6 результат роботи алгоритму стає гіршим, тому фіксуємо цей параметр на значенні 0,6.

ВИСНОВОК

В рамках даної лабораторної роботи я вирішив задачу комівояжера з симметричною мережою за допомогою генетичного алгоритму. Для цього було реалізовано класси: Graph, Individual та GeneticAlgorithm. Створив власні оператори кросоверінгу, мутації та локального покращення. Експерементально визначив оптимальні параметри роботи алгоритму.

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%.