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-14 Чорній Владислав* _____
(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.

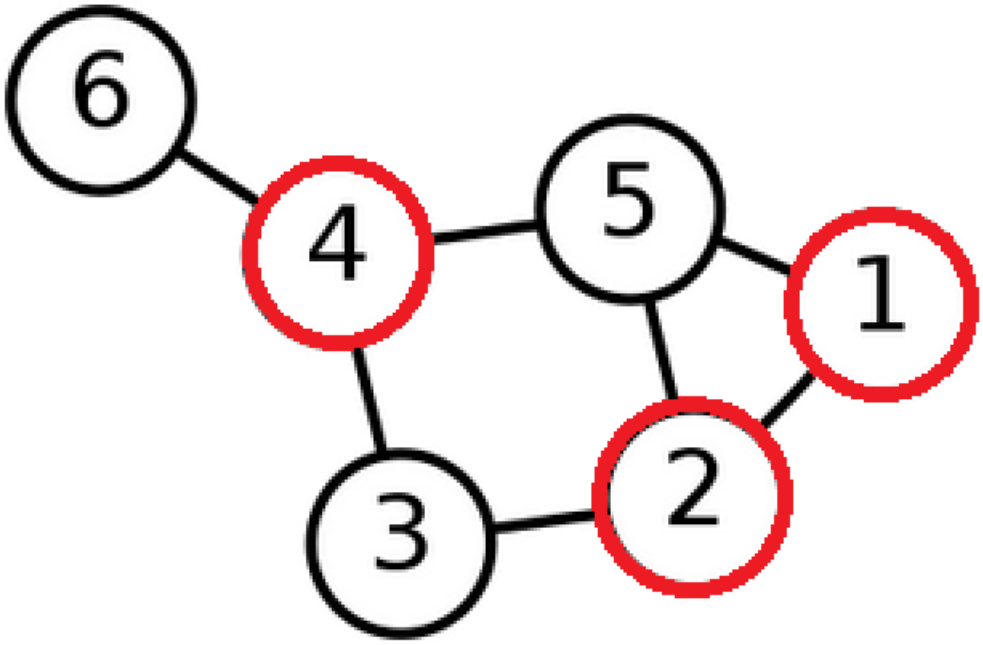
Зробити узагальнений висновок в якому обов’язково описати залежність якості розв’язку від вхідних параметрів.

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

Основний алгоритм

1. Створити популяцію шлях створення клік, які складаються з однієї вершини графа, для кожної вершини.
2. Запам’ятатати max ЦФ серед популяції.
3. ЦИКЛ ДЛЯ і ВІД 0 до 100 000:
    a. Вибрати батьків шляхом вибору найкращого і випадкового індивіда в популяції.
    b. Створити дитину шляхом схрещування батьків.
    c. З ймовірністю MUTATION_PROB застосувати оператор мутації до дитини.
    d. ЯКЩО ЦФ(дитина) = 0:
        i. Продовжити цикл.
    e. ІНАКШЕ:
        i. Застосувати оператор локального покращення.
    f. ЯКЩО ЦФ(дитина) > рекорд:
        i. Запам’ятатати новий рекорд
    g. ЯКЩО в популяції немає індивіда із генотипом, ідентичним генотипові дитини:
        i. Додати дитину до популяції.
        ii. Забрати з популяції особину з min ЦФ.
4. Кінець.

Алгоритм визначення ЦФ

1. Визначити вершини у кліці:
    a. ДЛЯ гену, номера гену У вершині:
        i. ЯКЩО ген = 1:
            1. Додати ген до масиву.
2. Перевірити чи вершини дійсно складають кліку:

a. ДЛЯ вершини У кліці:
    i. ДЛЯ сусіда У кліці:
        1. ЯКЩО сусід != вершина:
            a. ЯКЩО сусід НЕ Є сусідом вершини у графі:
                i. Повернути 0.
b. Повернути розмір кліки.

3. Кінець.

Оператори схрещування:

Одноточкове схрещування (a, b)

1. p = randint(0, size(a)).
2. Повернути a[:p] + b[p:].

Двоточкове схрещування (a, b)

1. p1 = randint(0, size(a)-1).
2. p2= randint(p1, size(a)).
3. Повернути a[:p1] + b[p1:p2] + a[p2:].

Рівномірне схрещування (a, b)

1. ДЛЯ генів x, y У a, b:
    a. Вибрати випадковим чином x або y і додати до нової хромосоми.
2. Повернути нову хромосому.

Оператори мутації

Фліп гена (c)

1. Вибрати випадковий ген.
2. Поміняти його на протилежний.
3. Оновити індивіда.

Фліп проміжку (c)

1. p1 = randint(0, size(c)-1).
2. p2= randint(p1, size(c)).
3. ДЛЯ гена МІЖ c[p1], c[p2]:
    a. Поміняти ген на протилежний.
4. Оновити індивіда.

Оператор локального покращення

Додавання випадкової вершини

1. Визначити вершини у кліці nodes.
2. Пройтись по сусідах шукаючи сумісного:
    a. ДЛЯ node У nodes:
        i. ДЛЯ neighbour У graph[nodes]:
            1. ЯКЩО усі елементи nodes У graph[neighbour]:
                a. Додати neighbour до кліки.
                b. Кінець.
3. Кінець.

Додавання вершини з евристикою

1. Визначити вершини у кліці nodes.
2. Визначити усі вершини, сусідні з nodes як neighbours.
3. Відсортувати neighbours за степенем у порядку спадання.
4. Пройтись по сусідах шукаючи сумісного:
    a. ДЛЯ neighbour У neighbours:
        i. ЯКЩО усі елементи nodes У graph[neighbour]:
            1. Додати neighbour до кліки.
            2. Кінець.
5. Кінець.

3.2 Програмна реалізація алгоритму

3.2.1 Вихідний код

```
from individual import Individual
from graph_module import nodes_n
import random as rand
import crossover, mutation, local

MUTATION_PROB = 0.25


def create_population(population):
    for i in range(nodes_n):
        chromosome = [0 for _ in range(nodes_n)]
        chromosome[i] = 1
        population.append(Individual(chromosome))
    return 1
```

```
def max_and_rand(population):
    a = max(population)
    b = rand.choice(population)
    while a == b:
        b = rand.choice((population))

    return a, b


def delete_rand_min(population):
    minimum = []
    m = population[0].f
    for ind in population:
        if ind.f < m:
            minimum.clear()
            m = ind.f
            minimum.append(ind)
        elif ind.f == m:
            minimum.append(ind)
    population.remove(rand.choice(minimum))


def run(crossover_func, mutation_func, local_func):
    a, b, c = 100000, 100000, 100000
    population = []
    record = create_population(population)

    for i in range(100_000):
        if not i % 10_000:
            print(i)

        parents = max_and_rand(population)
        kid = crossover_func(*parents)

        if rand.random() <= MUTATION_PROB:
            mutation_func(kid)

        if not kid.f:
            continue

        local_func(kid)

        if kid.f > record:
            record = kid.f
            print(i, record)
            if record == 15:
                a = i
            if record == 16:
                b = i
            if record >= 17:
                c = i
                break

        if kid not in population:
            population += kid,
            delete_rand_min(population)
```

```
    return a, b, c


if __name__ == '__main__':
    run(crossover.two_point, mutation.rand_change_one,
local.add_adj_node_heuristic)



import random as rand
from individual import Individual


def even(a, b):
    chromosome = []
    a, b = a.chromosome, b.chromosome
    for x, y in zip(a, b):
        chromosome += rand.choice([x, y]),
    return Individual(chromosome)


def one_point(a, b):
    a, b = a.chromosome, b.chromosome
    point = rand.randint(0, len(a)-1)
    return Individual(a[:point+1] + b[point+1:])


def two_point(a, b):
    a, b = a.chromosome, b.chromosome
    point1 = rand.randint(0, len(a)//2)
    point2 = rand.randint(point1, len(b) - 1)
    return Individual(a[:point1 + 1] + b[point1 + 1:point2+1] +
a[point2+1:])



import random as rand
from itertools import combinations


def generate_rand_graph(n, p, min_d, max_d):
    nodes = list(range(1, n+1))
    adj_list = {i: [] for i in nodes}
    possible_edges = combinations(nodes, 2)
    for u, v in possible_edges:
        if rand.random() < p and len(adj_list[v]) < max_d and
len(adj_list[u]) < max_d:
            adj_list[u].append(v)
            adj_list[v].append(u)
    for node, neighbours in adj_list.items():
        if len(neighbours) < min_d:
            nodes.remove(node)
            neighbours.append(rand.choice(nodes))
    return adj_list


def dump_graph():
```

```
    graph = generate_rand_graph(300, 0.90, 2, 30)
    with open("__graph_", "w") as f:
        f.write(str(graph))
    nodes_n = len(graph)
    return graph, nodes_n


def load_graph():
    with open("graph", "r") as f:
        graph = eval(f.read())
    nodes_n = len(graph)
    return graph, nodes_n


graph, nodes_n = dump_graph()



from graph_module import graph


class Individual:
    def __init__(self, chromosome):
        self.chromosome = chromosome
        self.f = self.max_clique(chromosome)

    @staticmethod
    def max_clique(chromosome):
        nodes = []
        for i, gene in enumerate(chromosome):
            if gene:
                nodes.append(i+1)
        for node in nodes:
            for neighbour in nodes:
                if node == neighbour:
                    continue
                else:
                    if neighbour not in graph[node]:
                        return 0
        return len(nodes)

    def update(self, chromosome):
        self.chromosome = chromosome
        self.f = self.max_clique(chromosome)

    def __lt__(self, other):
        return self.f.__lt__(other.f)

    def __gt__(self, other):
        return self.f.__gt__(other.f)

    def __le__(self, other):
        return self.f.__le__(other.f)

    def __ge__(self, other):
        return self.f.__ge__(other.f)

    def __repr__(self):
```

```
        return f"{self.f}"

    def __eq__(self, other):
        return self.chromosome == other.chromosome



from graph_module import graph
import random as rand


def add_rand_adj_node(c):
    nodes = []
    for i, gene in enumerate(c.chromosome):
        if gene:
            nodes.append(i + 1)

    rand.shuffle(nodes)
    for node in nodes:
        neighbours = graph[node]
        rand.shuffle(neighbours)
        for neighbour in neighbours:
            if neighbour in nodes:
                continue
            # if nodes in clique are all in neighbours of the neighbour
of the node
            if set(nodes) <= set(graph[neighbour]):
                chromosome = list(c.chromosome)
                chromosome[neighbour-1] = 1
                c.update(chromosome)
                return


def add_adj_node_heuristic(c):
    nodes = []
    for i, gene in enumerate(c.chromosome):
        if gene:
            nodes.append(i + 1)

    rand.shuffle(nodes)
    neighbours = []
    for node in nodes:
        neighbours += graph[node]

    neighbours = list(set(neighbours))
    rand.shuffle(neighbours)

    for neighbour in sorted(neighbours, key=lambda x: len(graph[x])):
        if neighbour in nodes:
            continue
        # if nodes in clique are all in neighbours of the neighbour of
the node
        if set(nodes) <= set(graph[neighbour]):
            chromosome = list(c.chromosome)
            chromosome[neighbour-1] = 1
            c.update(chromosome)
            return
```

```
import random as rand


def rand_change_one(c):
    i = rand.randint(0, len(c.chromosome)-1)
    chromosome = list(c.chromosome)
    chromosome[i] = 0 if chromosome[i] else 1
    c.update(chromosome)


def rand_change_interval(c):
    chromosome = list(c.chromosome)

    point1 = rand.randint(0, len(chromosome)-2)
    point2 = rand.randint(point1, len(chromosome))

    for i in range(point1, point2):
        chromosome[i] = 0 if chromosome[i] else 1

    c.update(chromosome)



from main import run
import crossover, local, mutation


def test1():
    functions = {crossover.one_point: (0, 0, 0, 0), crossover.two_point:
(0, 0, 0, 0), crossover.even: (0, 0, 0, 0)}
    for func in functions:
        print(func.__name__)
        a, b, c, stuck = 0, 0, 0, 0
        for i in range(10):
            print(i)
            res = run(func, mutation.rand_change_interval,
local.add_rand_adj_node)
            a += res[0]
            b += res[1]
            c += res[2]
            if res[2] == 100000:
                stuck += 1

        functions[func] = (a / 10, b / 10, c / 10, stuck)

    for func, results in functions.items():
        print(func.__name__, ":", results[3], ":", results[0],
results[1], results[2])


def test2():
    functions = {mutation.rand_change_one: (0, 0, 0, 0),
mutation.rand_change_interval: (0, 0, 0, 0)}
    for func in functions:
        print(func.__name__)
        a, b, c, stuck = 0, 0, 0, 0
```

```
        for i in range(20):
            print(i)
            res = run(crossover.two_point, func,
local.add_rand_adj_node)
            a += res[0]
            b += res[1]
            c += res[2]
            if res[2] == 100000:
                stuck += 1

        functions[func] = (a / 20, b / 20, c / 20, stuck)

    for func, results in functions.items():
        print(func.__name__, ":", results[3], ":", results[0],
results[1], results[2])


def test3():
    functions = {local.add_rand_adj_node: (0, 0, 0, 0),
local.add_adj_node_heuristic: (0, 0, 0, 0)}
    for func in functions:
        print(func.__name__)
        a, b, c, stuck = 0, 0, 0, 0
        for i in range(20):
            print(i)
            res = run(crossover.two_point, mutation.rand_change_one,
func)
            a += res[0]
            b += res[1]
            c += res[2]
            if res[2] == 100000:
                stuck += 1

        functions[func] = (a / 20, b / 20, c / 20, stuck)

    for func, results in functions.items():
        print(func.__name__, ":", results[3], ":", results[0],
results[1], results[2])


if __name__ == "__main__":
    test1()
```

3.2.2 Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми.

```
0
1 2
6 3
8 4
10 5
12 6
13 7
20 8
22 9
25 10
28 11
29 12
31 13
33 14
37 15
10000
20000
30000
40000
50000
60000
70000
80000
90000
```

Рисунок 3.1 –

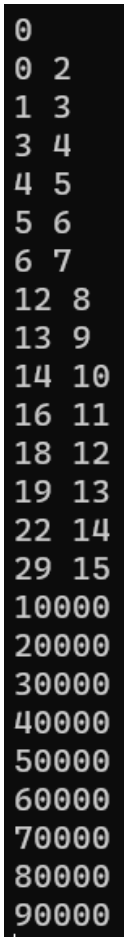


Рисунок 3.2 –

3.3 Тестування алгоритму

Маємо наступні досліджувані параметри:

1. Оператори схрещування
    a. Одноточкове схрещування
    b. Двоточкове схрещування
    c. Рівномірне схрещування
2. Оператор мутації
    a. Випадковий фліп гена
    b. Випадковий фліп проміжку з хромосоми
3. Оператори локального покращення
    a. Додавання до кліки випадкової вершини, сумісної з клікою
    b. Додавання до кліки вершини, сумісної з клікою, з евристикою перевірки спочатку вершин з найбільшими степенями

Зупиняємо виконання алгоритму коли досягли ЦФ=17 або к-ті ітерацій в 100 000.

Зафіксуємо оператор мутації — випадковий фліп гена, оператор локального покращеня — випадковий. Таблиця кількості тупиків (незнаходження глобального розв’язку) та середніх кількостей ітерацій t_{15}, t_{16} та t_{17} з 10 тестувань операторів схрещування наведена в таблиці 3.1. Графіки t(i) показані на рисунку 3.3.

Таблиця 3.1 — Показники тестування операторів схрещування.

Назва оператора	Кількість незнаходжень глобального розв’язку	t_{15}	t_{16}	t_{17}
Одноточкове	2	710,1	18 619,8	33 594,4
Двоточкове	0	460,3	2 688,3	9 199,0
Рівномірне	6	45 510,3	60 306,2	60 326,5

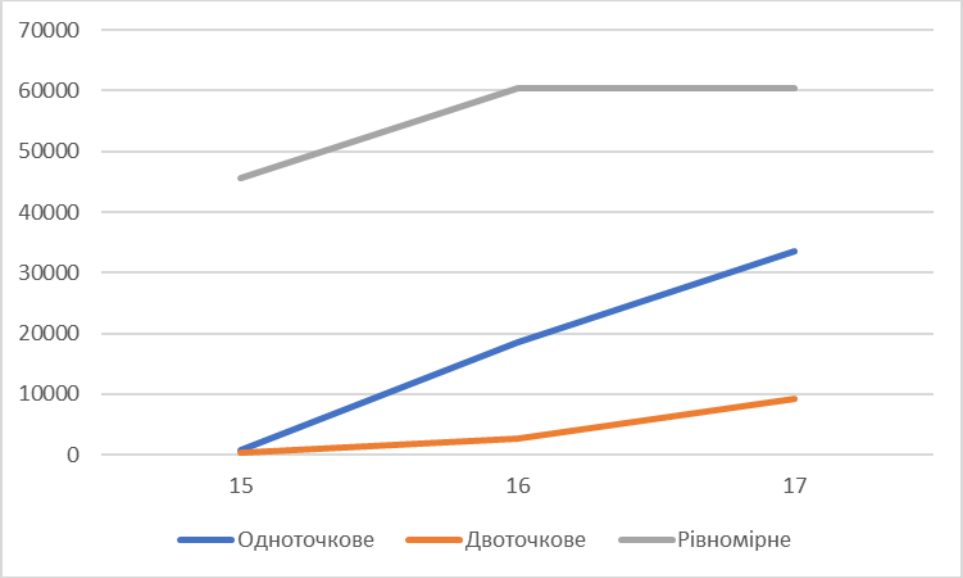


Рисунок 3.3 — Показники операторів схрещування

Обираємо оператор довоточковий оператор схрещування як найефективніший, фіксуємо разом із випадковим оператором локального покращення. Таблиця кількості тупиків (незнаходження глобального розв'язку) та середніх кількостей ітерацій t_{15}, t_{16} та t_{17} з 20 тестувань операторів мутації наведена в таблиці 3.2. Графіки t(i) показані на рисунку 3.4.

Таблиця 3.2 — Показники тестування операторів мутації.

Назва оператора	Кількість незнаходжень глобального розв'язку	t_{15}	t_{16}	t_{17}
Фліп гена	0	400,5	2 931,35	9 439,1
Фліп проміжку	1	287,45	3 876,9	18 890,3

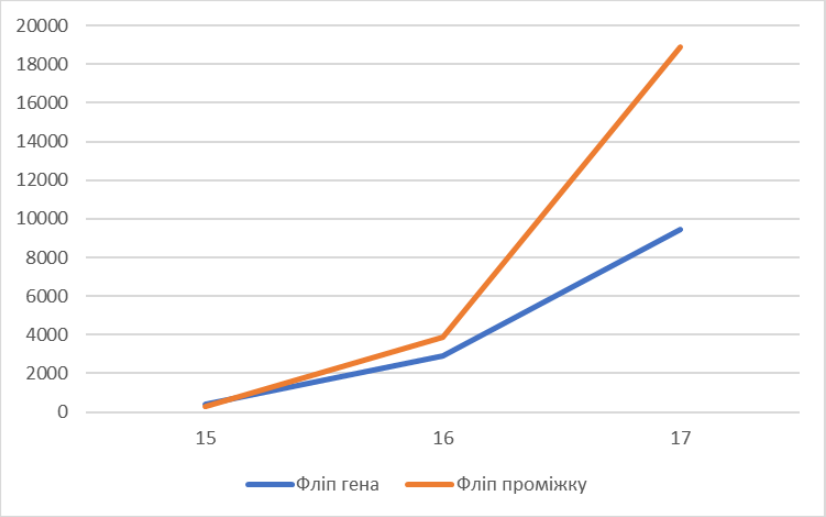


Рисунок 3.4 — Показники операторів мутації

Обираємо фліп гена, фіксуємо разом з двоточковим схрещуванням. Таблиця кількості тупиків (незнаходження глобального розв'язку) та середніх кількостей ітерацій t_{15}, t_{16} та t_{17} з 20 тестувань операторів локального покращення наведена в таблиці 3.3. Графіки t(i) показані на рисунку 3.5.

Таблиця 3.3 — Показники тестування операторів локального покращення.

Назва оператора	Кількість незнаходжень глобального розв'язку	t_{15}	t_{16}	t_{17}
Випадкова вершина	0	431,65	2 133,45	11 062,3
Вершина з евристикою	0	784,35	1 525,4	10 772,2

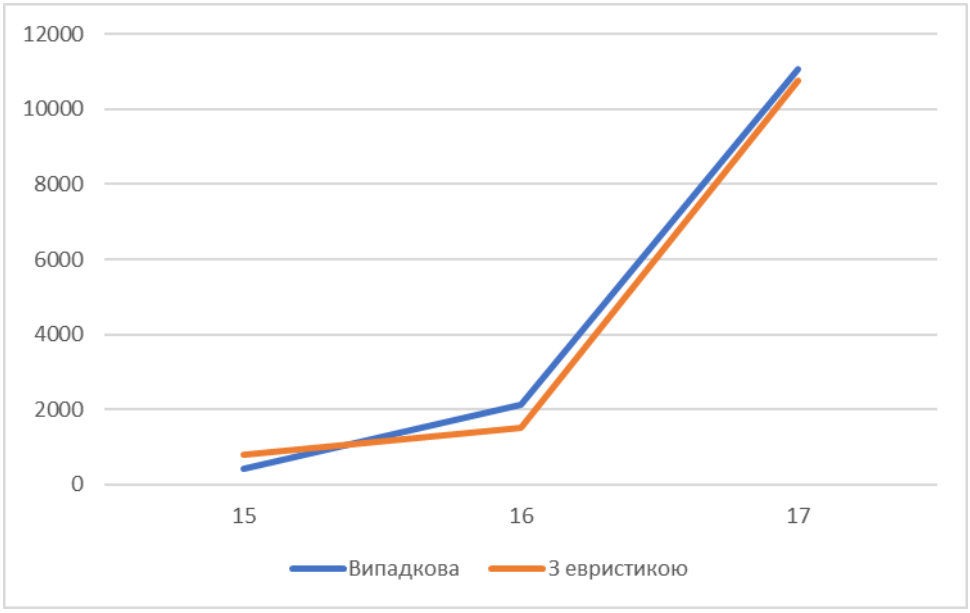


Рисунок 3.5 — Показники операторів локального покращення

Обираємо оператор локального покращення з евристикою.

В результаті отримали наступну оптимальну конфігурацію алгоритму: двоточкове схрещування, мутація, в якій мутує один ген, оператор локального покращення, в якому до кліки додається доступна вершина, починаючи з вершини із найбільшим степенем.

ВИСНОВОК

В рамках даної лабораторної роботи було формалізовано алгоритм вирішення обчислювальної задачі про кліку генетичним алгоритмом. Було записано розроблений алгоритм у покроковому вигляді та виконано його програмну реалізацію на мові програмування Python.

Змінюючи наступні параметри алгоритму: оператор схрещування, оператор мутації та оператор локального покращення, було визначено найкращі з них, ними виявилися двоточкове схрещування, оператор мутації, в якому мутує один випадковий ген та оператор локального покращення, в якому алгоритм намагається додати вершину до кліки, починаючи із сусідніх вершин з найбільшим степенем. В такій конфігурації популяція доволі рідко застряє в локальному максимумі. При цьому було зроблено висновок, що рівномірне схрещування та оператор мутації, в якому мутує проміжок генів у хромосомі, є далеко не оптимальними для вирішення нашої задачі.

Було зроблено висновок, що генетичний алгоритм є доволі ефективним метаевристичним алгоритмом розв’язування задач.

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%.