

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 Мельник М.О.
(шифр, прізвище, ім'я, по батькові)

Перевірив

Сопов О.О.
(прізвище, ім'я, по батькові)

Київ 2022

ЗМІСТ

1	МЕТА ЛАБОРАТОРНОЇ РОБОТИ	3
2	ЗАВДАННЯ	4
3	ВИКОНАННЯ.....	6
3.1	ПОКРОКОВИЙ АЛГОРИТМ	6
3.2	ПРОГРАМНА РЕАЛІЗАЦІЯ АЛГОРИТМУ	8
3.2.1	<i>Вихідний код.....</i>	8
3.2.2	<i>Приклади роботи</i>	13
3.3	ТЕСТУВАННЯ АЛГОРИТМУ	15
	ВИСНОВОК	19
	КРИТЕРІЇ ОЦІНЮВАННЯ	20

1 МЕТА ЛАБОРАТОРНОЇ РОБОТИ

Мета роботи – вивчити основні підходи розробки метаврестичних 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 ступенем 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
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

	<p>вершин в ній.</p> <p>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).</p> <p>Застосування:</p> <ul style="list-style-type: none"> – біоінформатика; – електротехніка;
--	--

Таблиця 2.2 – Варіанти алгоритмів і досліджувані параметри

№	Алгоритми і досліджувані параметри
1	<p>Генетичний алгоритм:</p> <ul style="list-style-type: none"> - оператор схрещування (мінімум 3); - мутація (мінімум 2); - оператор локального покращення (мінімум 2).

Таблиця 2.3 – Варіанти задач і алгоритмів

№	Задачі і алгоритми
18	Задача про кліку (обчислювальна задача) + Генетичний 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'ятати \max ЦФ серед популяції.
3. ЦИКЛ ДЛЯ i 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'ятати 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 U вершині:
 - i. ЯКЩО $\text{ген} = 1$:
 1. Додати ген до масиву.
2. Перевірити чи вершини дійсно складають кліку:

- а. ДЛЯ вершини У кліці:
 - і. ДЛЯ сусіда У кліці:
 1. ЯКЩО сусід != вершина:
 - а. ЯКЩО сусід НЕ Є сусідом вершини у графі:
 - і. Повернути 0.
 - б. Повернути розмір кліки.
3. Кінець.

Оператори схрещування:

Одноточкове схрещування (a, b)

1. $p = \text{randint}(0, \text{size}(a))$.
2. Повернути $a[:p] + b[p:]$.

Двоточкове схрещування (a, b)

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

Рівномірне схрещування (a, b)

1. ДЛЯ генів x, y У a, b:
 - а. 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. $p1 = \text{randint}(0, \text{size}(c)-1)$.
2. $p2 = \text{randint}(p1, \text{size}(c))$.
3. ДЛЯ гена МІЖ $c[p1], c[p2]$:
 - а. 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 Y nodes:
 - i. ДЛЯ neighbour Y graph[nodes]:
 1. ЯКЩО усі елементи nodes Y 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 Y neighbours:
 - i. ЯКЩО усі елементи nodes Y graph[neighbour]:
 1. Додати neighbour до кліки.
 2. Кінець.
5. Кінець.

3.2 Програмна реалізація алгоритму

3.2.1 Вихідний код

individual

```
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

```

crossover

```

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:])

```

mutation

```

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)

```

local

```

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

```

main

```

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)

```

testing

```

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)

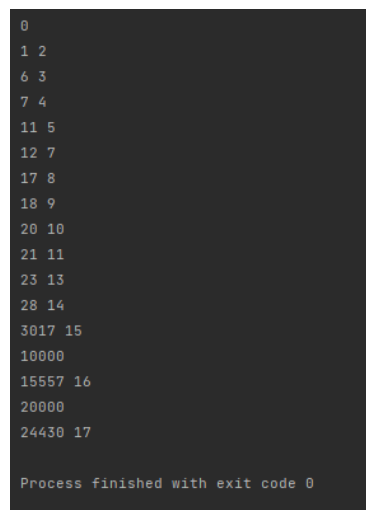
    for func, results in functions.items():
        print(func.__name__, ":", results[3], ":", results[0], results[1],
              results[2])

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
7 4
11 5
12 7
17 8
18 9
20 10
21 11
23 13
28 14
30 17
15 10000
16 15557
17 20000
17 24430

Process finished with exit code 0

```

Рисунок 3.1 – Приклад роботи програми для випадкового графа

```
0
1 2
5 3
8 4
14 5
15 6
18 7
22 8
33 9
35 10
36 11
37 12
45 13
47 14
50 15
51 16
10000
20000
30000
40000
50000
51420 17

Process finished with exit code 0
```

Рисунок 3.2 – Приклад роботи графа для випадкового графа.

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 покращення — 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

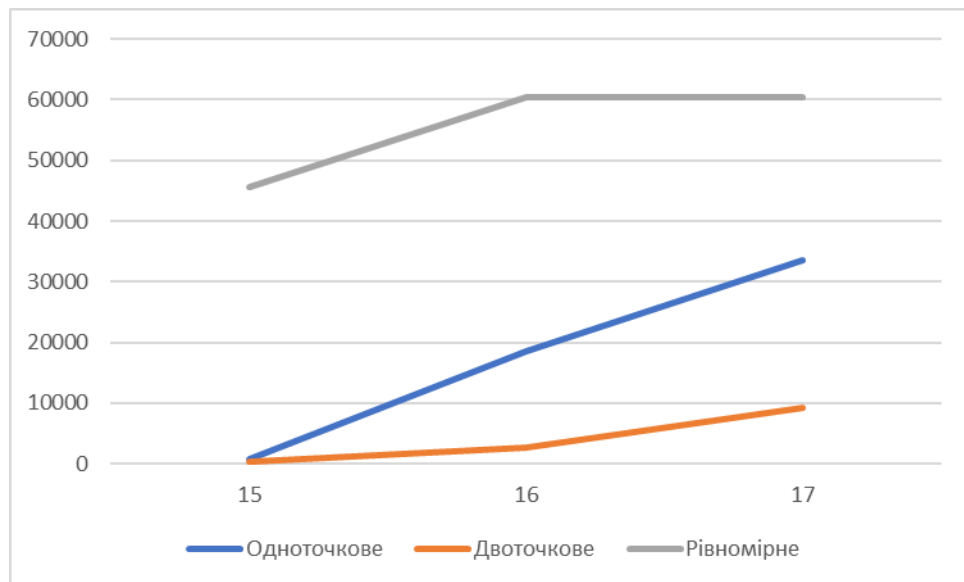


Рисунок 3.3 — Показники операторів схрещування

Обираємо оператор двоточковий 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

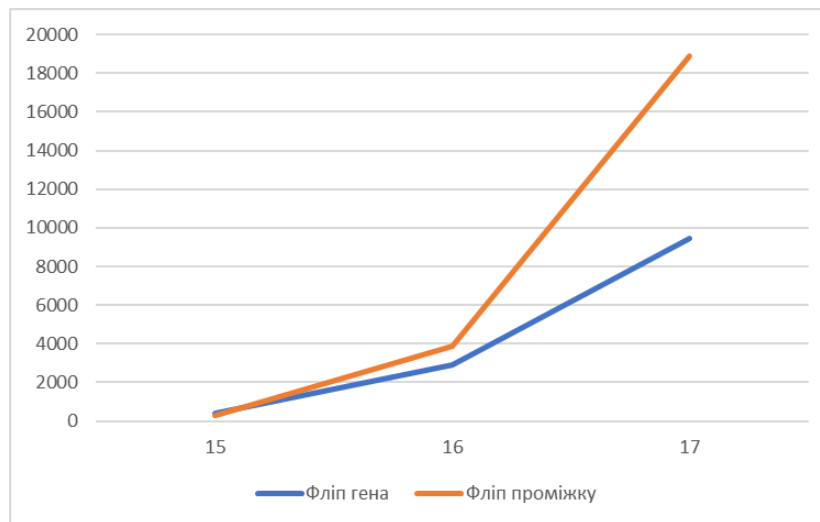


Рисунок 3.4 — Показники операторів мутації

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

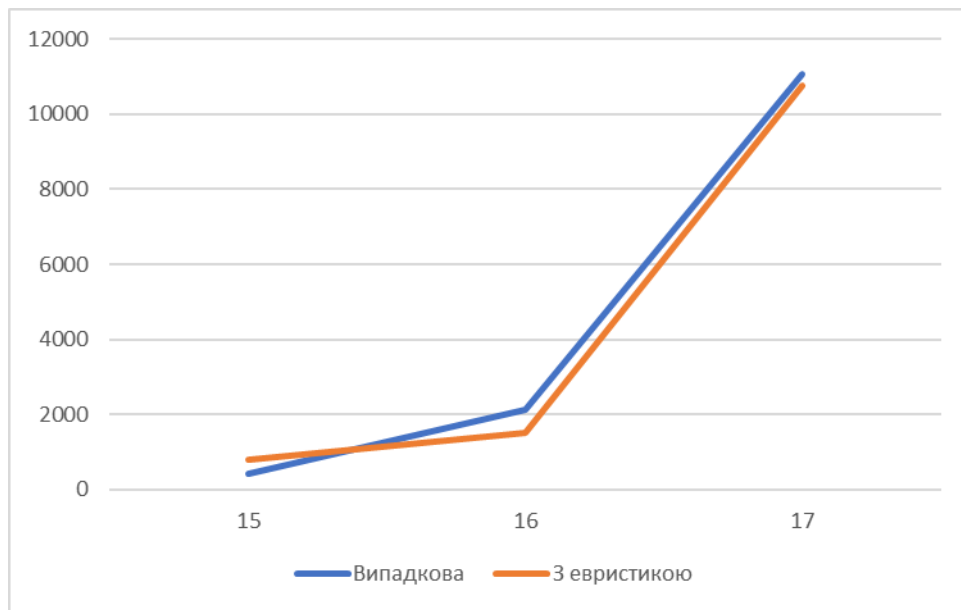


Рисунок 3.5 — Показники операторів локального покращення

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%.