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-13 Недельчев Є.О.*
(шифр, прізвище, ім'я, по батькові)

Перевірив *Сопов О.О.*
(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**

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;

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).

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’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

Utils.py

```
from graph_io import num_nodes
from interior import Interior
import random as rand


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


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))
```

Graph_io.py

```
import random as rand
from itertools import combinations


def generate_random_graph(num_nodes, probability, min_degree, max_degree):
    nodes = list(range(1, num_nodes + 1))
    adjacency_list = {node: [] for node in nodes}
    possible_edges = combinations(nodes, 2)

    # Only add edges that meet the criteria
    for u, v in possible_edges:
        if rand.random() < probability and len(adjacency_list[v]) < max_degree
and len(adjacency_list[u]) < max_degree:
            adjacency_list[u].append(v)
            adjacency_list[v].append(u)

    # add nodes that meet the minimum degree criteria
    for node, neighbours in adjacency_list.items():
        while len(neighbours) < min_degree:
            potential_neighbor = rand.choice(nodes)
            if potential_neighbor not in neighbours:
                neighbours.append(potential_neighbor)
                adjacency_list[potential_neighbor].append(node)
    return adjacency_list
```

```
def dump_graph_to_file(graph, file_name):
    with open(file_name, "w") as f:
        f.write(str(graph))
    num_nodes = len(graph)
    return graph, num_nodes


def load_graph_from_file(file_name):
    with open(file_name, "r") as f:
        graph = eval(f.read())
    num_nodes = len(graph)
    return graph, num_nodes


graph, num_nodes = dump_graph_to_file(generate_random_graph(300, 0.90, 2, 30),
"graph.txt")
```

Interion.py

```
from graph_io import graph


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

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

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

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

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

    def __repr__(self):
        return f"{self.f}"

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

    def update(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)
```

Genetic_algorithm_utils.py

```
import random as rand
from interior import Interior
from graph_io import graph


#
#   Mutation
#
def randomly_flip_one_bit(chromosome):
    i = rand.randint(0, len(chromosome) - 1)
    chromosome = list(chromosome)
    chromosome[i] = 0 if chromosome[i] else 1
    return chromosome


def randomly_flip_interval_bits(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
    return chromosome



#
#   Local
#
def get_nodes_from_chromosome(chromosome):
    nodes = []
    for i, gene in enumerate(chromosome):
        if gene:
            nodes.append(i + 1)
    return nodes


def add_random_adjacent_node(chromosome):
    nodes = get_nodes_from_chromosome(chromosome)
    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(chromosome)
                chromosome[neighbour - 1] = 1
                return chromosome
    return None


def add_adjacent_node_by_heuristic(chromosome):
    nodes = get_nodes_from_chromosome(chromosome)
    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(chromosome)
            chromosome[neighbour - 1] = 1
            return chromosome
    return None


#
#   Crossover
#
def even_crossover(parent1, parent2):
    chromosome = []
    for x, y in zip(parent1.chromosome, parent2.chromosome):
        chromosome += rand.choice([x, y]),
    return Interior(chromosome)


def one_point_crossover(parent1, parent2):
    point = rand.randint(0, len(parent1.chromosome) - 1)
    offspring = Interior(parent1.chromosome[:point + 1] +
parent2.chromosome[point + 1:])
    return offspring


def two_point_crossover(parent1, parent2):
    point1 = rand.randint(0, len(parent1.chromosome) // 2)
    point2 = rand.randint(point1, len(parent2.chromosome) - 1)
    offspring = Interior(
        parent1.chromosome[:point1 + 1] + parent2.chromosome[point1 + 1:point2 +
1] + parent1.chromosome[point2 + 1:])
    return offspring
```

Main.py

```
from genetic_algorithm_utils import *
from utils import *


def main():
    iteration = 100_000
    crossover_func = two_point_crossover
    mutation_func = randomly_flip_one_bit
    local_func = add_adjacent_node_by_heuristic

    a, b, c = 100000, 100000, 100000
    population = create_population()
    record = max([i.f for i in population])
    for i in range(iteration):
        if not i % 10_000:
            print(i)

        parents = max_and_rand(population)
        child = crossover_func(*parents)
```

```
        if rand.random() <= 0.25:
            mutation_func(child.chromosome)

        if not child.f:
            continue

        local_func(child.chromosome)

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

        if child not in population:
            population.append(child)
            delete_rand_min(population)

    return a, b, c


if __name__ == '__main__':
    main()
```

3.2.2 Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми.

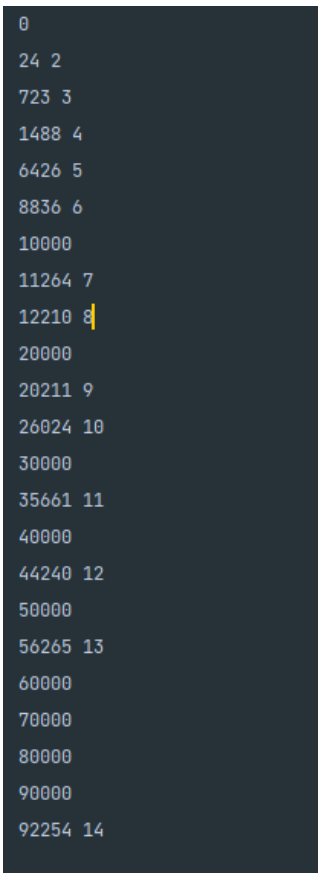

Рисунок 3.1 – Приклад роботи програми для випадкового графа

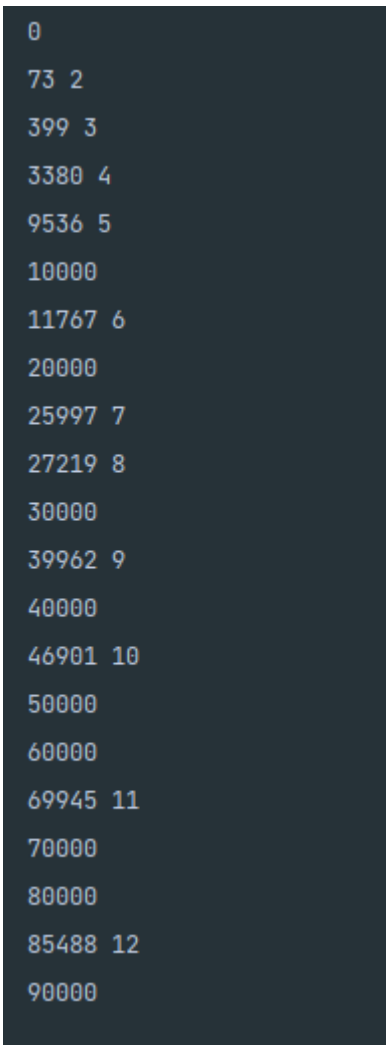

Рисунок 3.2 – Приклад роботи графа для випадкового графа.

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 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

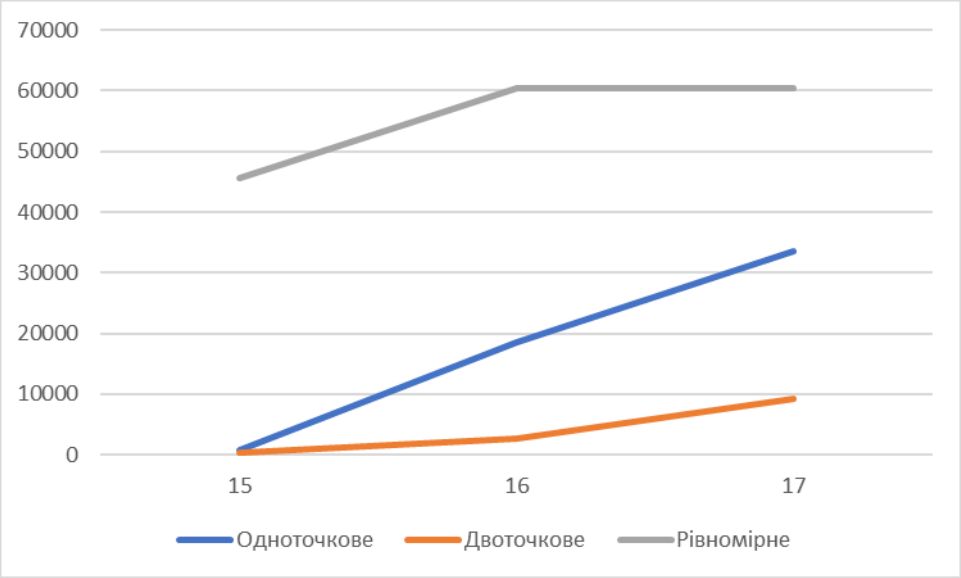

Рисунок 3.3 — Показники операторів схрещування

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

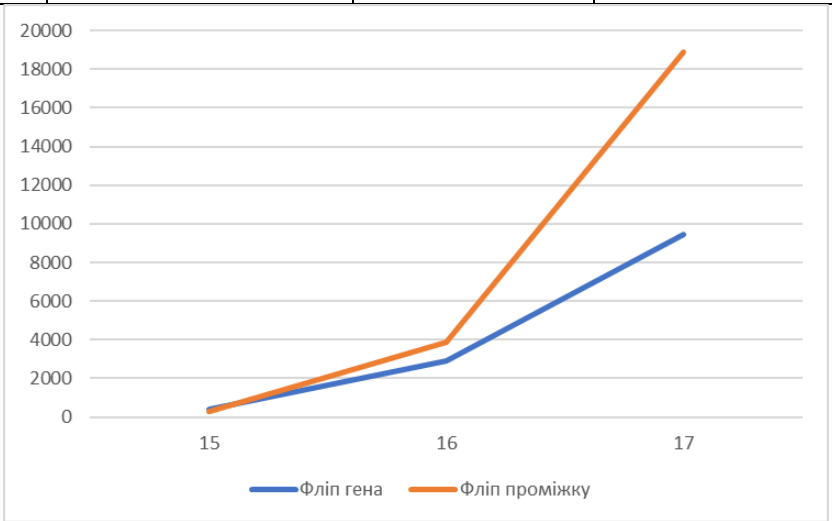

Рисунок 3.4 — Показники операторів мутації

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

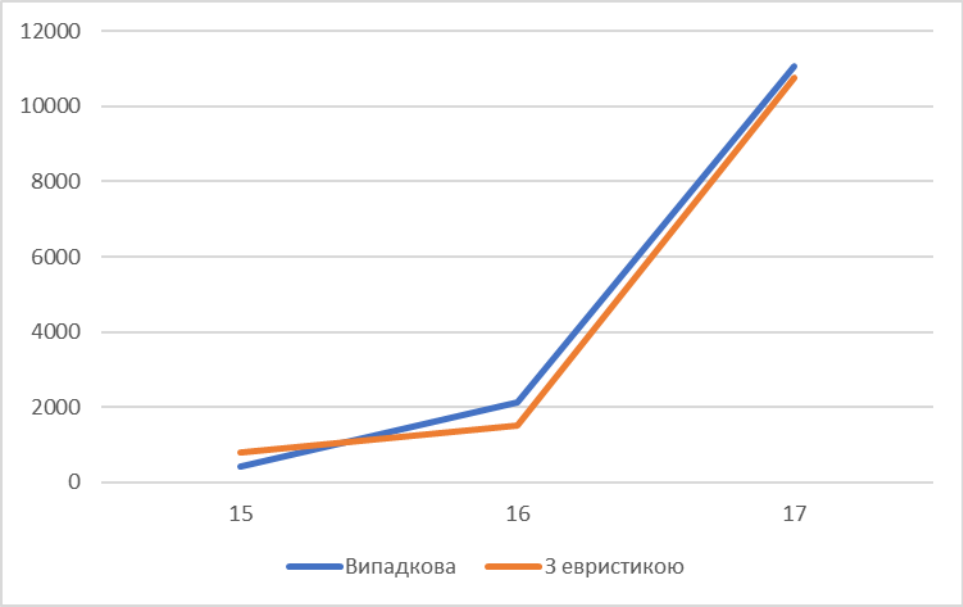

Рисунок 3.5 — Показники операторів локального покращення

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%.