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 № 4 з дисципліни
«Проектування алгоритмів»

„Проектування і аналіз алгоритмів для вирішення NP-складних задач ч.1”

Виконав(ла) *ІП-13 Лисенко Анастасія* _____
(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 формалізації метаевристичних алгоритмів і вирішення типових задач з їхньою допомогою.

2 ЗАВДАННЯ

Згідно варіанту, розробити алгоритм вирішення задачі і виконати його програмну реалізацію на будь-якій мові програмування.

Задача, алгоритм і його параметри наведені в таблиці 2.1.

Зафіксувати якість отриманого розв'язку (значення цільової функції) після кожних 20 ітерацій до 1000 і побудувати графік залежності якості розв'язку від числа ітерацій.

Зробити узагальнений висновок.

Таблиця 2.1 – Варіанти алгоритмів

№	**Задача і алгоритм**
15	Задача розфарбовування графу (100 вершин, степінь вершини не більше 20, але не менше 1), класичний бджолиний алгоритм (число бджіл 30 із них 3 розвідники).

3 ВИКОНАННЯ

3.1 Програмна реалізація алгоритму

3.1.1 Вихідний код

representation_graph.py

```
import networkx as nx


class RepresentationGraph:
    def __init__(self):
        self.empty = 0

    @staticmethod
    def drawGraph(graph, vertex_colors):
        layout = nx.spring_layout(graph)
        colors = ['blue', 'yellow', 'red', 'orange', 'black', 'purple', 'green',
'grey', 'purple', 'pink', 'brown']
        values = [colors[vertex_colors[node]] for node in graph.nodes()]
        nx.draw(graph, layout, with_labels=True, node_color=values,
edge_color='black', width=1, alpha=0.8)

    @staticmethod
    def getAvailableColor(neighbours, vertex_colors, num_colors,
previous_color):
        available_colors = [color for color in range(num_colors)]
        for neighbor in neighbours:
            color = vertex_colors[neighbor]
            if color in available_colors:
                available_colors.remove(color)
        if previous_color in available_colors:
            available_colors.remove(previous_color)
        if len(available_colors) != 0:
            return available_colors[0]
        return -1

    @staticmethod
    def removeDuplicateEdges(lst):
        without_duplicates = []
        for key, val in lst:
            if key != val:
                if [key, val] and [val, key] not in without_duplicates:
                    without_duplicates.append([key, val])
        return without_duplicates
```

algo.py

```
import matplotlib.pyplot as plt
from representation_graph import *
from program_graph import *


class Algo:
    def __init__(self, graph: Program_Graph):
        self.n = graph.getNodes()
        self.lst = graph.getVertices()

    def getNeighbours(self, vertex):
```

```
        neighbour_list = []
        for key, val in self.lst:
            if key == vertex:
                neighbour_list.append(val)
        return neighbour_list

    def isCorrectColoredNeighbours(self, vertex, vertex_colors, current_color):
        neighbours = self.getNeighbours(vertex)
        neighbours_not_in_same_color = [vertex_colors[neighbor] != current_color
for neighbor in neighbours]
        if sum(neighbours_not_in_same_color) ==
len(neighbours_not_in_same_color):
            return True
        return False

    def maxPowerVertex(self):
        elements = [row[0] for row in self.lst]
        max_val = max(set(elements), key=elements.count)
        return max_val

    def try_to_reduce_num_of_colors(self, vertex, vertex_colors, num_colors):
        neighbours = self.getNeighbours(vertex)
        for neighbor in neighbours:
            temp_colors = vertex_colors.copy()
            temp_colors[vertex], temp_colors[neighbor] = temp_colors[neighbor],
temp_colors[vertex]
            if self.isCorrectColoredNeighbours(neighbor, temp_colors,
temp_colors[neighbor]):
                new_color =
RepresentationGraph.getAvailableColor(self.getNeighbours(neighbor), temp_colors,
num_colors, vertex_colors[neighbor])
                if new_color != -1:
                    temp_colors[neighbor] = new_color
                    vertex_colors = temp_colors.copy()

        return vertex_colors

    def createGraph(self):
        graph = nx.Graph()
        lst = RepresentationGraph.removeDuplicateEdges(self.lst)
        for row in lst:
            graph.add_edge(row[0], row[1])
        return graph

    def displayGraph(self, vertex_colors):
        print("Number of colors used:", len(set(vertex_colors)))
        print("\n_____\n")
        graph = self.createGraph()
        RepresentationGraph.drawGraph(graph, vertex_colors)
        plt.show()

    def greedy(self):
        current_color = 0
        vertex_colors = [-1 for _ in range(self.n)]
        while sum([val == -1 for val in vertex_colors]) != 0:
            for vertex in range(self.n):
                if vertex_colors[vertex] == -1:
                    if self.isCorrectColoredNeighbours(vertex, vertex_colors,
current_color):
                        vertex_colors[vertex] = current_color
            current_color += 1
        return vertex_colors, current_color

    def bees(self):
```

```
        vertex_colors, num_colors = self.greedy()
        self.displayGraph(vertex_colors)
        vertex = self.maxPowerVertex()
        nxt = [vertex]
        counter = 0
        parent = -1
        randomness = 1
        mutation = randomness
        while len(nxt) < 30:
            vertex = nxt[counter]
            neighbours = self.getNeighbours(vertex)
            for neighbor in neighbours:
                if neighbor != parent:
                    nxt.append(neighbor)
                if mutation == 0:
                    nxt.append(np.random.randint(0, self.n+1))
                    mutation = randomness
            parent = vertex
            counter += 1
            mutation -= 1
        for vertex in nxt:
            vertex_colors = self.try_to_reduce_num_of_colors(vertex,
vertex_colors, num_colors)
        self.displayGraph(vertex_colors)
```

program_graph.py

```
import numpy as np
from constants import *

class Program_Graph:
    def __init__(self, n=NODES_AMOUNT, min_pow=MIN_POW, max_pow=MAX_POW):
        self.n = n
        self.min_pow = min_pow
        self.max_pow = max_pow
        self.vertices = []

    def create(self):
        for i in range(self.n):
            num_of_edges = np.random.randint(1, self.max_pow + 1)
            all_vertices = np.arange(self.n)
            np.random.shuffle(all_vertices)
            neighbours = all_vertices[:num_of_edges]
            for neighbor in neighbours:
                self.vertices.append([i, neighbor])

    def getVertices(self):
        return self.vertices

    def getNodes(self):
        return self.n
```

constants.py

```
NODES_AMOUNT = 100
MAX_POW = 20
MIN_POW = 1
```

main.py

```
from algo import *
from program_graph import *


def main():
    graph = Program_Graph()
    graph.create()

    algo = Algo(graph)
    algo.bees()


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

3.1.2 Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми.

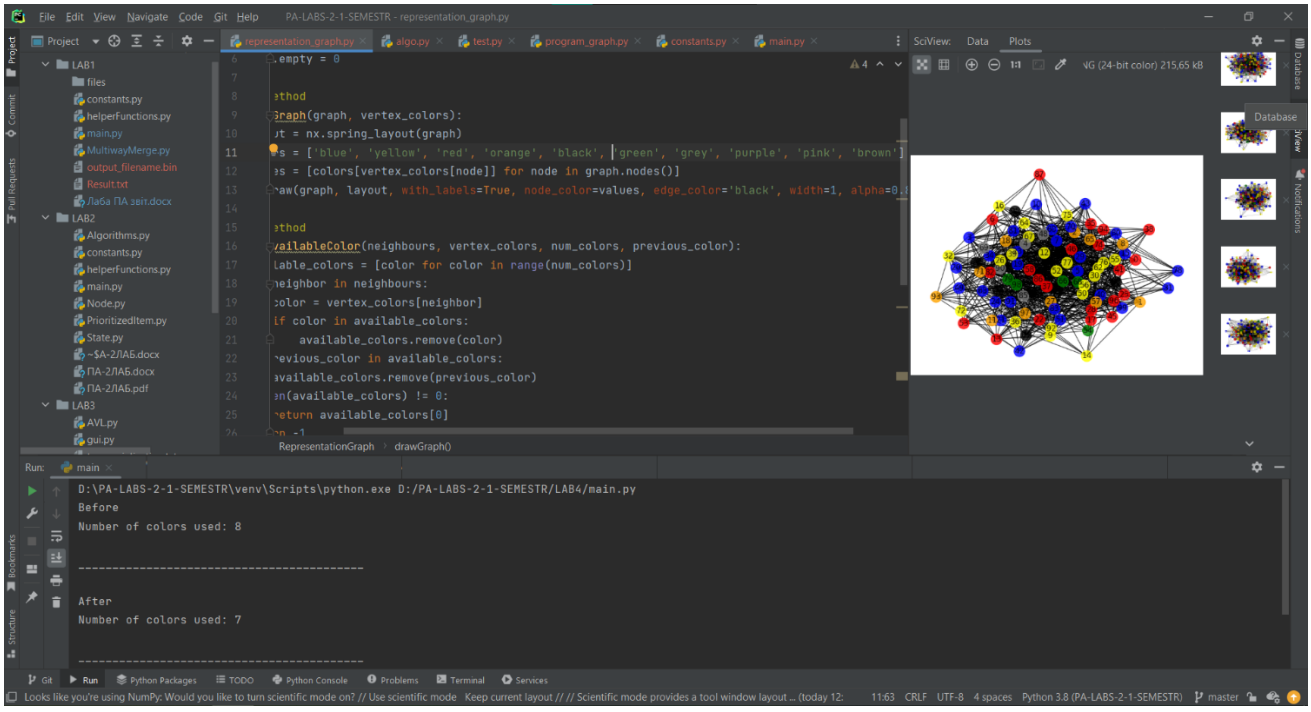

Рисунок 3.1 – Основне вікно програми

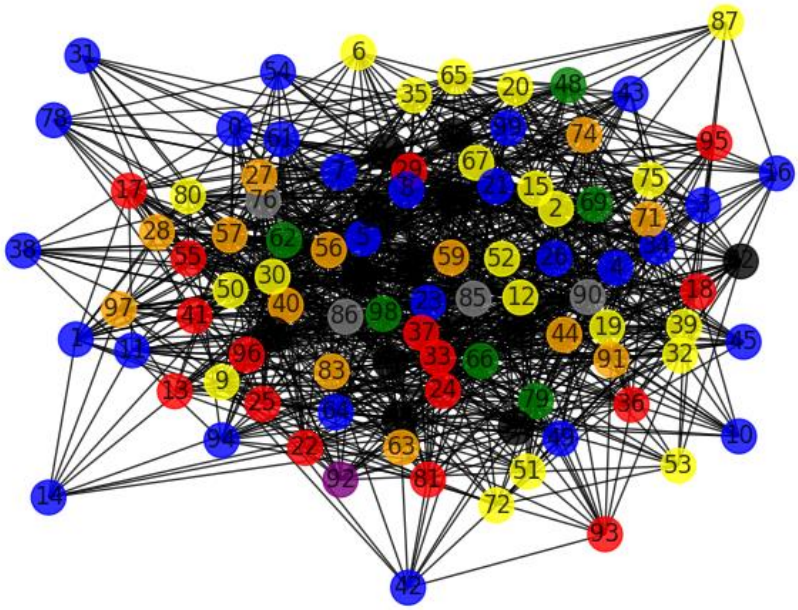

Рисунок 3.2 – Розмалювання жадібним алгоритмом

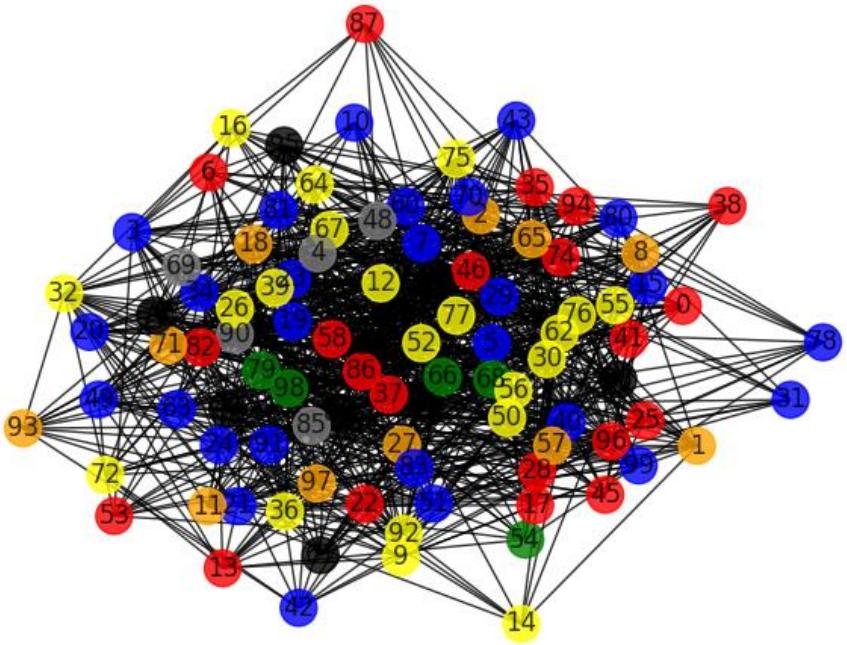

Рисунок 3.3 – Розмалювання бджолиним алгоритмом

3.2 Тестування алгоритму

3.2.1 Значення цільової функції зі збільшенням кількості ітерацій

У таблиці 3.1 наведено значення цільової функції зі збільшенням кількості ітерацій.

Ітерація	Значення функції	Ітерація	Значення функції	Ітерація	Значення функції
0	10	380	7	760	7
20	9	400	7	780	7
40	9	420	7	800	7
60	9	440	7	820	7
80	9	460	7	840	7
100	9	480	7	860	7
120	8	500	7	880	7
140	8	520	7	900	7
160	8	540	7	920	7
180	8	560	7	940	7
200	8	580	7	960	7
220	8	600	7	980	7
240	8	620	7	1000	7
260	8	640	7		
280	8	660	7		
300	8	680	7		
320	7	700	7		
340	7	720	7		
360	7	740	7		

3.2.2 Графіки залежності розв'язку від числа ітерацій

На рисунку 3.3 наведений графік, який показує якість отриманого розв'язку.

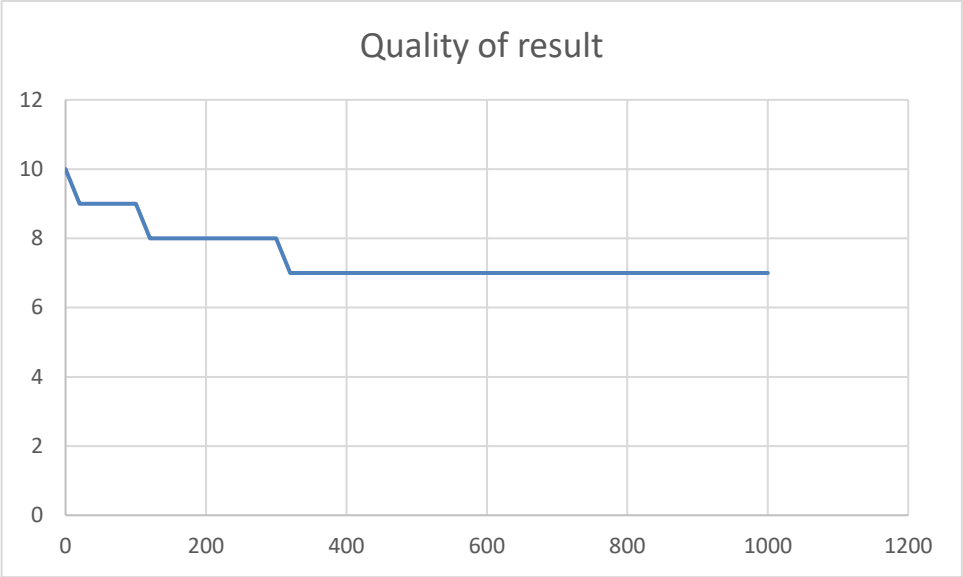

Рисунок 3.3 – Графіки залежності розв'язку від числа ітерацій

ВИСНОВОК

В рамках даної лабораторної роботи був розроблений класичний бджолиний алгоритм для вирішення задачі розмалювання графу. Реалізований алгоритм розбитий на декілька циклів: цикл розвідників, протягом якого обираються перспективні ділянки з початкових, які формуються за допомогою припустимого розмалювання жадібним алгоритмом, за допомогою евристики найменшого хроматичного числа, та цикл фуражирів, протягом якого виконуються дії щодо покращення хроматичного числа графу за допомогою спроб обміну кольорів між вершинами з максимальними степенями у графі з їх сусідами.

КРИТЕРІЇ ОЦІНЮВАННЯ

При здачі лабораторної роботи до 27.11.2021 включно максимальний бал дорівнює – 5. Після 27.11.2021 максимальний бал дорівнює – 1.

Критерії оцінювання у відсотках від максимального балу:

- програмна реалізація алгоритму – 75%;
- тестування алгоритму– 20%;
- висновок – 5%.