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 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 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.

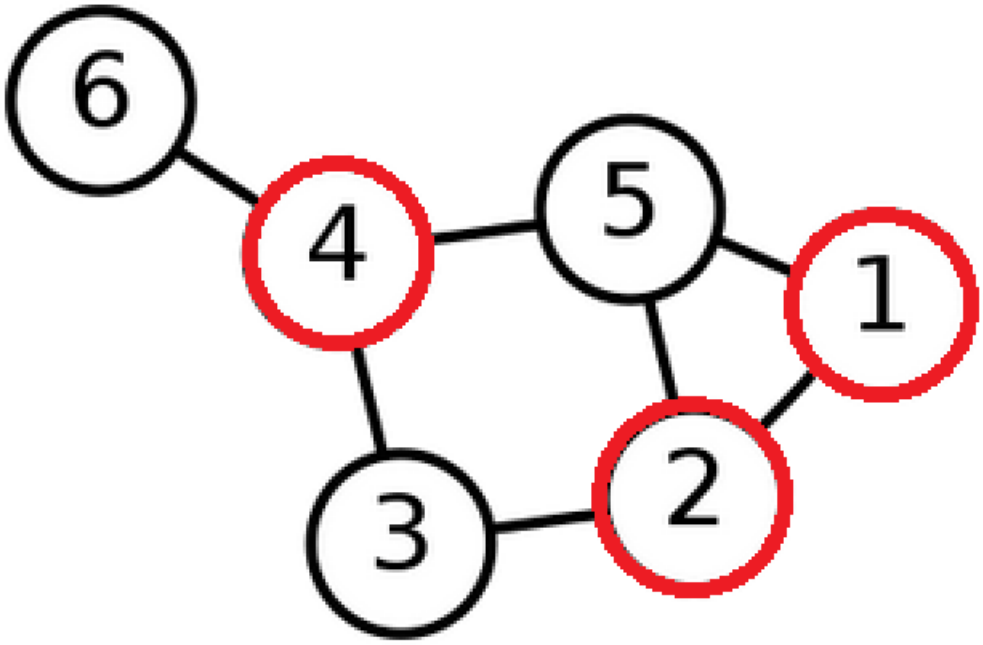
Зробити узагальнений висновок в якому обов’язково описати залежність якості розв’язку від вхідних параметрів.

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

3.1.1 Генерування графу

1. ПОЧАТОК
2. Створення деякого набору вершин
3. Визначення степені для кожної вершини випадковим чином(у межах від 2 до 30)
4. Створення зв’язків між вершинами у кількості відповідній степені кожної вершини
5. КІНЕЦЬ

3.1.2 Жадібний алгоритм для початкового розмалювання графу

1. ПОЧАТОК
2. Присвоїти кількість непофарбованих = кількості вершин у графі
3. ПОКИ є непофарбовані вершини:
   3.1. Обрати випадковим ще не пофарбовану вершину
   3.2. Обрати для вершини допустимий колір(перебрати усі кольори із палітри використаних та порівняти з кольорами сусідів. Якщо сусіди даної вершини пофарбовані в усі використані кольори, то пофарбувати дану вершину у новий колір та занести його у палітру.
   3.3. Зменшити кількість непофарбованих вершин на 1.
4. Визначити хроматичне число графу
5. КІНЕЦЬ

3.1.3 Класичний бджолиний алгоритм

1. ПОЧАТОК
2. Створити масив з деякої визначеної кількості ділянок, де ділянка – це конкретне розмалювання даного графа жадібним алгоритмом
3. Створити чергу з пріоритетом для визначення порядку обходу вершин у циклі фуражирів для кожної ділянки. (за степеню вершини)
4. Визначити розподіл фуражирів для кожної ділянки(можливий варіант - за остаточним принципом: для кожної ділянки не більше, ніж половина від максимальної степені вершини у графі)
5. ПОКИ не досягнутий ліміт по ітераціям:
   5.1. Отримати перелік обраних ділянок за допомогою запуску циклу роботи бджіл-розвідників
   5.2. Для ВСІХ обраних ділянок запустити цикл роботи бджіл-фуражирів
   5.3. Обрати ділянку з найкращим хроматичним числом як рекордний результат
6. ПОВЕРНУТИ рекордний результат
7. КІНЕЦЬ

3.1.4 Цикл роботи бджіл-розвідників

1. ПОЧАТОК
2. Відсортувати масив з ділянками за цільовою функцією(хроматичне число графа) в у порядку зростання
3. Визначити кількість бджіл-розвідників, які будуть обирати ділянки випадково, не дивлячись на значення цільової функції
4. Визначати кількість бджіл-розвідників, які будуть обирати ділянки за значеннями цільової функції, віднявши кількість бджіл-розвідників, які обирають ділянки випадково, від загальної кількості бджіл розвідників

5. Обрати найкращі ділянки у кількісті рівної кількості бджіл-розвідників, які обирають ділянки за значенням цільової функції
6. Обрати випадкові ділянки у кількості рівної кількості бджіл-розвідників, які обирають ділянки випадковим чином.
7. ПОВЕРНУТИ перелік обраних ділянок
8. КІНЕЦЬ

3.1.5 Цикл роботи бджіл-фуражирів

1. ПОЧАТОК
2. ПОКИ не закінчяться виділені на ділянку фуражири:
   2.1. Взяти ділянку з списку обраних
   2.2. З черги з пріоритетом для цієї ділянки взяти вершину для розгляду
   2.3. Спробувати обмінятись кольорами між обраною вершиною та її сусідами до першого вдалого обміну. Як результат повернути індекс сусіда, з яким вдалося обмінятись кольором. Якщо обмін невдалий – повернути ідентифікатор невдачі
   2.4. ЯКЩО індекс сусіда визначений, ТО спробувати пофарбувати спочатку обрану вершину у інший колір з палітри використаних кольорів, а потім так самого вершину з знайденим індексом і в кінці оновити перелік використаних кольорів та хроматичне число графа
   2.5. Зменшити кількість фуражирів на 1
3. КІНЕЦЬ

3.2 Програмна реалізація алгоритму

3.2.1 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()
```

3.2.2 Приклади роботи

На рисунках 3.1, 3.2 та 3.3 показані приклади роботи програми.

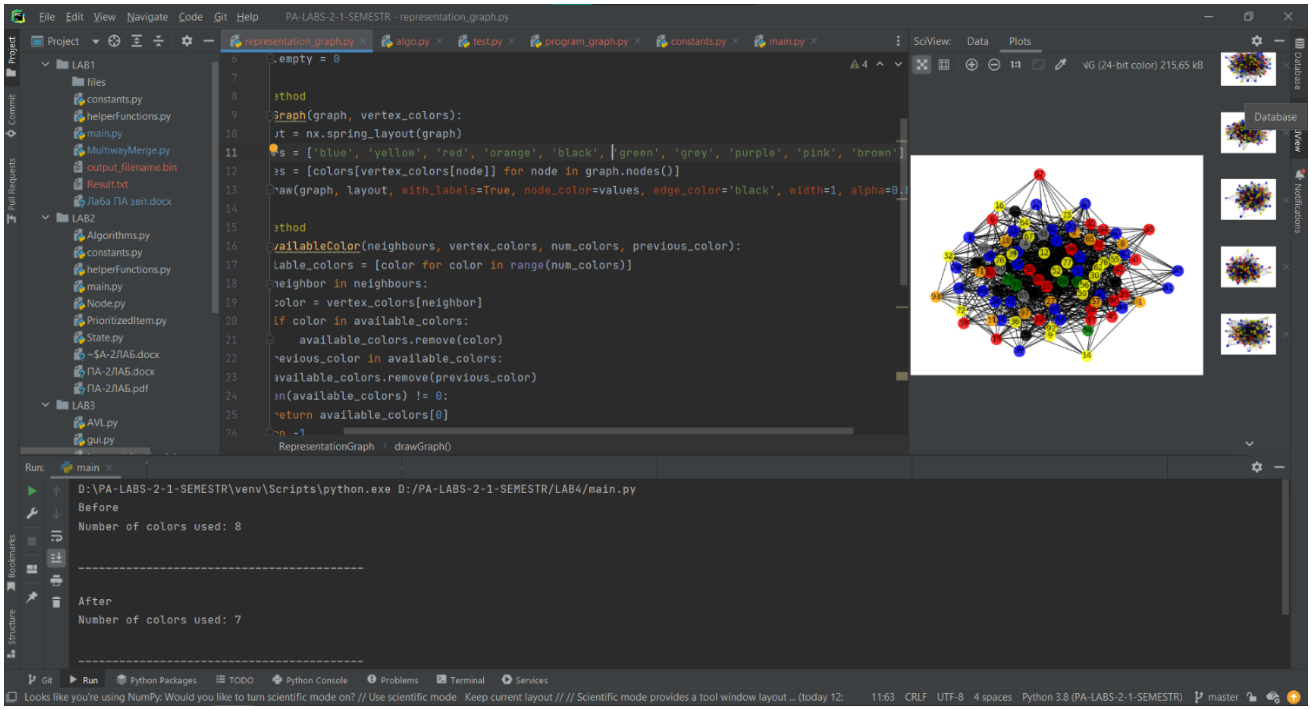


Рисунок 3.1 – Основне вікно програми

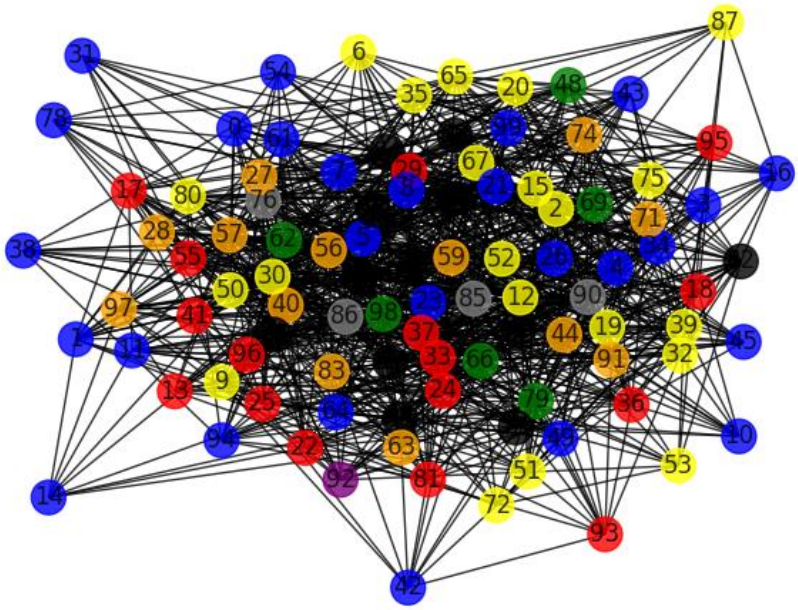


Рисунок 3.2 – Розмалювання жадібним алгоритмом

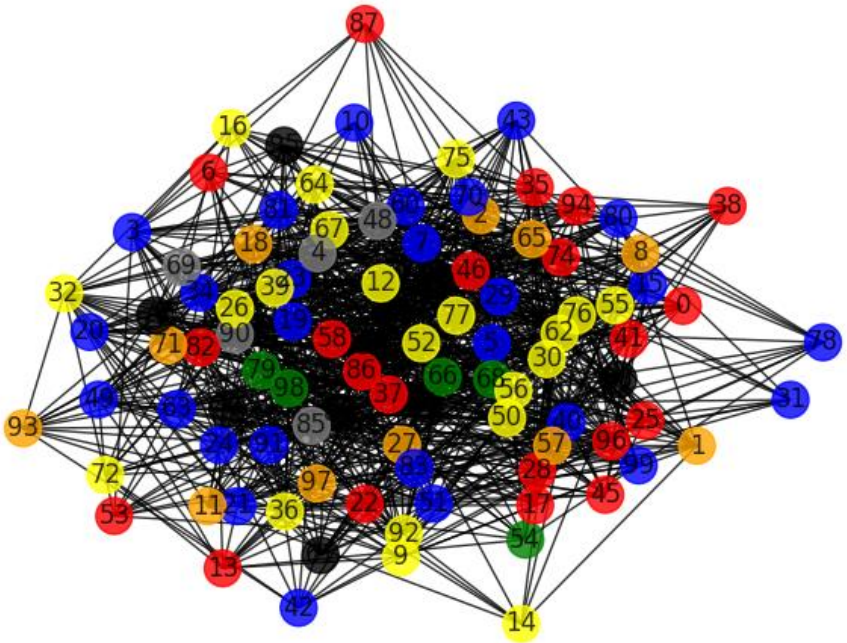


Рисунок 3.3 – Розмалювання бджолиним алгоритмом

3.3 Тестування алгоритму

Тестування алгоритму проведемо наступним чином: алгоритм має два основних параметри, які впливають на ефективність його роботи: кількість початкових ділянок, які розфарбовані жадібним алгоритмом, та кількість бджіл(розвідників і фуражирів) та їх розподіл за ролями. За допомогою перебору значень усіх параметрів при фіксуванні інших параметрів знайдемо оптимальні значення для кожного параметру.

Спочатку дізнаємось оптимальну кількість початкових ділянок. Результати іспитів будуть зафіксовані у таблиці 1.

Номер іспиту	Кількість ділянок	Кількість бджіл-розвідників	Кількість бджіл-фуражирів	Хроматичне число графу
1	1	*2*	*20*	15
2	2			14
3	3			14
4	4			13
5	8			13
6	16			12
7	**20**			**12**
8	25			13
9	35			14
10	40			14

Таблиця 1

За результатами іспитів маємо наступну картину: якість розв’язку покращується при середній кількості початкових ділянок – в районі 16 - 20. Подальше збільшення кількості ділянок лише погіршує якість розв’язку – це може бути зумовлено тим, що бджоли-розвідники, які обирають ділянки випадковим чином, частіше обирають нові неперспективні ділянки, не даючи

фуражирам покращити належним чином вже розглянуті на минулих ітераціях ділянки – це може заважати покинути локальний максимум. Результати іспитів візуалізовано за допомогою графіка на рисунку 3.3. Чим менше хроматичне число – тим краще розв’язок.

Рисунок 3.3

Наступним параметром є кількість бджіл і їх розподіл. Результати іспитів будуть зафіксовані у таблиці 2.

Номер іспиту	Кількість ділянок	Кількість бджіл-розвідників	Кількість бджіл-фуражирів	Хроматичне число графу
1	*20*	2	20	14
2			**30**	**13**
3			45	13
4			60	13
5			75	13
6		3	20	14
7			30	13
8			**45**	**12**
9			60	12
10			75	12
11		**4**	20	14
12			30	13
13			45	12
14			**60**	**11**
15			75	11
16		5	20	14
17			30	13
18			45	12
19			**60**	**11**
20			75	11

Таблиця 2

Під час обробки результатів іспитів досліджена закономірність: якість розв’язку досягає свого максимуму при наступному розподілу кількості бджіл: на одну бджолу-розвідника потрібна така кількість фуражирів, яка буде відправлена на ділянку(зазвичай як максимальна степінь вершини у графі або половина від цієї степені). Але найкращим розподілом бджіл для поставленої задачі є 4 на 60. Подальше пропорційне збільшення кількості бджіл лише подовжить час виконання ітерації алгоритму але суттєво не покращує розв’язок. Результати іспитів візуалізовано за допомогою графіка на рисунку 3.4, де по осі y – хроматичне число графу, а по осі x – кількість бджіл-розвідників при відповідності з бджолами-фуражирами 1 к 15. Чим менше хроматичне число – тим краще розв’язок.

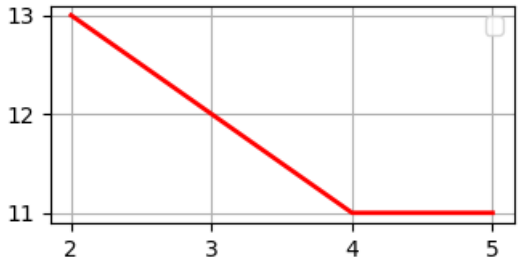


Рисунок 3.4

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. Також проведена робота щодо покращення роботи алгоритму за допомогою підбору оптимальних параметрів алгоритму, від яких залежить якість кінцевого розв’язку. Наочно досліджені значення оптимальних параметрів та встановлені закономірності між конкретними параметрами, які надають алгоритму максимальну ефективність.

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%.