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 Кушнір Ганна Вікторівна* _____
(шифр, прізвище, ім'я, по батькові)

Перевірив *Сопов Олексій Олександрович* _____
(прізвище, ім'я, по батькові)

Київ 2023

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.

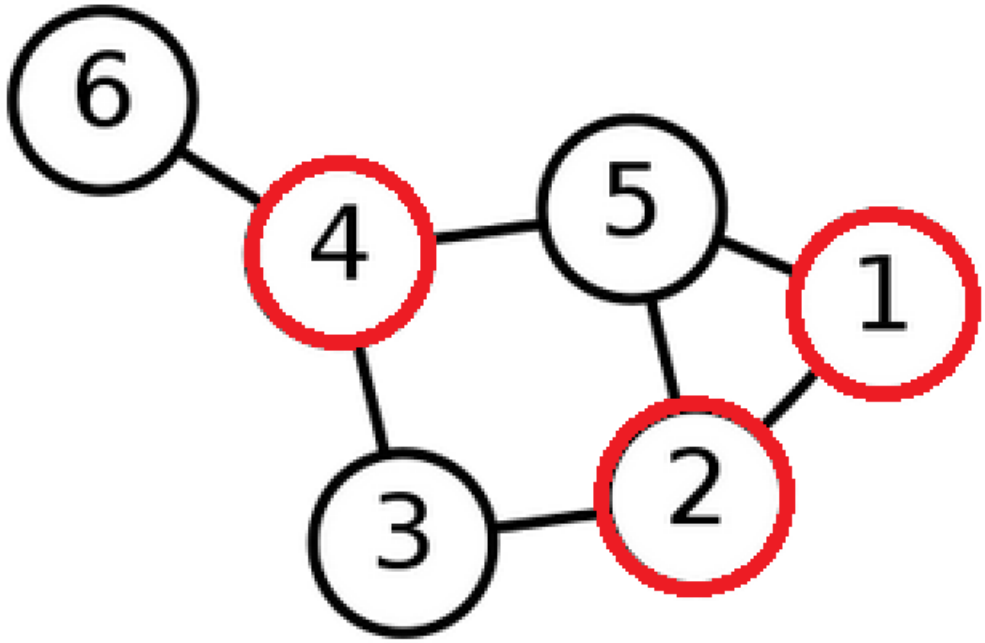
Зробити узагальнений висновок в якому обов’язково описати залежність якості розв’язку від вхідних параметрів.

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

Бджолиний алгоритм:

1 Початкова ініціалізація:
- 1.1 Ініціалізувати вхідний граф
- 1.2 Ініціалізувати параметри scouts_num, foragers_num, solutions_num, iterations_num
- 1.3 Ініціалізувати чергу з пріоритетом vertices – набір вершин графу в порядку спадання кількості сусідів
- 1.4 Ініціалізувати чергу з пріоритетом solutions – перелік згенерованих жадібним алгоритмом ділянок (розфарбувань графу) у порядку спадання цільової функції

2 Присвоїти foragers_on_solution = ⌈ foragers_num / scouts_num ⌉

3 Повторити від 0 до iterations_num не включно:
- 3.1 Ініціалізувати змінні scouts_sent і foragers_sent і присвоїти їм 0
- 3.2 Поки scouts_sent < scouts_num і scouts_sent < solutions_num виконувати:
  - 3.2.1 З імовірністю 0.5 обрати ділянку з найменшим значенням цільової функції або випадкову ділянку, за умови, що ділянка не розглядалась на даній ітерації
  - 3.2.2 Відправити фуражирів кількістю foragers_on_solution на обрану ділянку (див. наступний алгоритм)
  - 3.2.3 Збільшити значення scouts_sent на 1
  - 3.2.4 Збільшити значення foragers_sent на кількість відправлених фуражирів (foragers_on_solution)

4 Повернути першу ділянку зі списку solutions

Алгоритм відправки фуражирів

1 Ініціалізувати змінну foragers і присвоїти їй 0

2 Повторити для і від 0 до кількості вершин графу:

2.1 Якщо foragers = foragers_sent, тоді перейти до пункту 3

2.2 Ініціалізувати змінну vertex як елемент масиву vertices, що знаходиться під індексом і

2.3 Ініціалізувати масив neighbors – перелік сусідів вершини vertex

2.4 Повторити для кожного сусіда neighbor з масиву neighbors:

2.4.1 Якщо foragers = foragers_sent, тоді перейти до пункту 3

2.4.2 Поміняти місцями кольори вершин vertex та neighbor

2.4.3 Якщо при зміні не виникло конфліктів з іншими вершинами, тоді:

2.4.3.1 Спробувати замінити нові кольори вершин на інші, також використані раніше

2.4.3.2 Зберегти зміни для ділянки

2.4.4 Інакше відмінити всі зміни, здійснені з ділянкою на даній ітерації

3.2 Програмна реалізація алгоритму

3.2.1 Вихідний код

Файл «main.py»:

```
from graph import *
from algorithm import *

if __name__ == "__main__":
    path = 'file_lab5_test2.txt'
    f = CreateGraph()
    f.create_and_save_to_file(path)
    scout_bees = 2
    foragers = scout_bees * 30
    solutions = 1
    iterations = 100
```

```
algorithm = Algorithm(path, scout_bees, foragers, solutions, iterations)
algorithm.bee_algorithm()
```

Файл «graph.py»:

```
import numpy as np

class CreateGraph:
    def __init__(self, num: int = 300, min_pow: int = 2, max_pow: int = 30):
        self.num = num
        self.min_pow = min_pow
        self.max_pow = max_pow
        self.edges = []

    def create_and_save_to_file(self, path: str):
        self._generate_graph()
        with open(path, 'w') as f:
            f.write(str(self.num) + '\n')
            for edge in self.edges:
                f.write(str(edge[0]) + ' ' + str(edge[1]) + '\n')

    def _generate_graph(self):
        counts = [0 for i in range(self.num)]
        for vertex in range(self.num):
            num_of_neighbors = np.random.randint(self.min_pow, self.max_pow)
            if counts[vertex] + num_of_neighbors > self.max_pow:
                num_of_neighbors = self.max_pow - counts[vertex]
            counts[vertex] += num_of_neighbors
            i = 0
            neighbors = []
            while i < num_of_neighbors:
                neighbor = np.random.randint(0, self.num)
                if vertex != neighbor and ([vertex, neighbor] not in self.edges) and ([neighbor, vertex] not in self.edges):
                    if counts[neighbor] < self.max_pow:
                        neighbors.append(neighbor)
                        counts[neighbor] += 1
                        i += 1
            for neighbor in neighbors:
                self.edges.append([vertex, neighbor])
```

Файл «algorithm.py»:

```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt

"""
Задача розфарбовування графу класичним бджолиним алгоритмом.
"""

class Graph:
    """
    Слугує для збереження структури графу.
    """
    def __init__(self, path: str):
        """Ініціалізує граф."""
        self.edges = []
        self.vertex_num = 0
        self.counts = []
        with open(path, 'r') as f:
            lines = f.readlines()
            self.vertex_num = int(lines[0])
            lines = lines[1:]
            self.edges = [list(map(int, line.split())) for line in lines]
            self._count_occurence_of_vertices_and_sort()

    def get_neighbors(self, vertex: int):
        """Повертає всіх сусідів заданої вершини vertex."""
        neighbors = []
        for start, end in self.edges:
            if start == vertex:
                neighbors.append(end)
            if end == vertex:
                neighbors.append(start)
        return neighbors
    def _count_occurence_of_vertices_and_sort(self):
        """Рахує кількість зв'язків з кожною із вершин і
        сортує створений список кількостей у порядку спадання."""
        vertices_in_edges = [edge[0] for edge in self.edges] + [edge[1] for edge in self.edges]
        self.counts = []
        for vertex in set(vertices_in_edges):
            self.counts.append([vertex, vertices_in_edges.count(vertex)])
        def func(elem: list[int]):
            return elem[1]
        self.counts.sort(key = func, reverse = True)
```

```
class Solution:
    """
    Слугує для збереження даних про конкретне рішення (певне розфарбування графу).
    """
    def __init__(self, graph: Graph, solution: list[int] = []):
        """Ініціалізує рішення."""
        self.graph = graph
        self.solution = solution
        self.colors_num = self.count_colors()
        self.last_color_num = self.count_occurrences_of_last_color()

    def count_colors(self):
        """Рахує кількість використаних кольорів."""
        return len(set(self.solution))

    def count_occurrences_of_last_color(self):
        """Рахує кількість появ кольору, що був доданий останнім."""
        return self.solution.count(self.colors_num - 1)

    def greedy_algorithm(self):
        """Жадібний алгоритм розфарбування графа."""
        self.solution = [-1] * self.graph.vertex_num
        curr_color = 0
        vertices = np.arange(self.graph.vertex_num)
        np.random.shuffle(vertices)
        while -1 in self.solution:
            for vertex in vertices:
                if self.solution[vertex] == -1:
                    if self._is_color_available(vertex, curr_color):
                        self.solution[vertex] = curr_color
            curr_color += 1
        self.colors_num = self.count_colors()
        self.last_color_num = self.count_occurrences_of_last_color()
    def get_neighbor_solution(self, foragers: int):
        """Повертає рішення, близьке до поточного, але з деякими змінами
        та покращеннями."""
        if len(self.solution) == 0:
            return Solution(self.graph)
        foragers_sent = 0
        solution_to_return = self.solution.copy()
        for i in range(self.graph.vertex_num):
            if foragers_sent == foragers:
                break
            vertex = self.graph.counts[i][0]
            neighbors = self.graph.get_neighbors(vertex)
```

```
        for neighbor in neighbors:
            if foragers_sent == foragers:
                break
            temp_solution = solution_to_return.copy()
            temp_solution[vertex], temp_solution[neighbor] = temp_solution[neighbor], temp_solution[vertex]
            if self._is_color_available(neighbor, temp_solution[neighbor], temp_solution) and self._is_color_available(vertex, temp_solution[vertex], temp_solution):
                foragers_sent += 1
                new_color1 = self._get_available_color(vertex, temp_solution)
                if new_color1 != -1:
                    temp_solution[vertex] = new_color1
                new_color2 = self._get_available_color(neighbor, temp_solution)
                if new_color2 != -1:
                    temp_solution[neighbor] = new_color2
                solution_to_return = temp_solution.copy()
    return Solution(self.graph, solution_to_return)

  def _is_color_available(self, vertex: int, color: int, solution: list[int] = []):
    """Перевіряє, чи можна розмалювати дану вершину даним кольором
    без виникнення конфліктів."""
    if solution == []:
        solution = self.solution
    neighbors = self.graph.get_neighbors(vertex)
    for neighbor in neighbors:
        if solution[neighbor] == color:
            return False
    return True

  def _get_available_color(self, vertex: int, solution: list[int] = []):
    """Шукає колір, доступний для розмалювання даної вершини."""
    if solution == []:
        solution = self.solution
    neighbors = self.graph.get_neighbors(vertex)
    available_colors = list(np.arange(len(set(solution))))
    if solution[vertex] in available_colors:
        available_colors.remove(solution[vertex])
    for neighbor in neighbors:
        if solution[neighbor] in available_colors:
            available_colors.remove(solution[neighbor])
    if len(available_colors) == 0:
        return -1
    return available_colors[0]
```

```
class Algorithm:
    """
    Слугує для ініціалізації та запуску алгоритму.
    """
    def __init__(self, path: str, scouts_num: int, foragers_num: int, solutions_num: int, iterations_num: int):
        self.scouts_num = scouts_num
        self.foragers_num = foragers_num
        self.solutions_num = solutions_num
        self.iterations_num = iterations_num
        self.probability = 0.5
        self.graph = Graph(path)
        self.solutions: list[Solution] = self._create_list_of_solutions()
        self._sort_solutions()

    def _create_list_of_solutions(self):
        """Ініціалізує ділянки для розвідки та пошуку."""
        solutions: list[Solution] = []
        for i in range(self.solutions_num):
            solutions.append(Solution(self.graph))
            solutions[i].greedy_algorithm()
        return solutions

    def _sort_solutions(self):
        """Сортує рішення у порядку зростання цільової функції
        (кількості використаних кольорів)."""
        for i in range(len(self.solutions)):
            for j in range(i + 1, len(self.solutions)):
                if (self.solutions[i].colors_num > self.solutions[j].colors_num or
                self.solutions[i].colors_num == self.solutions[j].colors_num and self.solutions[i].last_color_num >=
self.solutions[j].last_color_num):
                    temp = self.solutions[i]
                    self.solutions[i] = self.solutions[j]
                    self.solutions[j] = temp
    def bee_algorithm(self):
        """Запускає бджолиний алгоритм."""
        foragers_on_solution = int(self.foragers_num / self.scouts_num + 0.5)
        for i in range(self.iterations_num):
            visited_indexes = []
            curr_best_sltn_index = 0
            scouts_sent = 0
            foragers_sent = 0
            while scouts_sent < self.scouts_num and scouts_sent < self.solutions_num:
                if foragers_sent + foragers_on_solution >= self.foragers_num:
                    if self._send_scout(curr_best_sltn_index, visited_indexes, self.foragers_num - foragers_sent):
                        scouts_sent += 1
```

```
                break
            else:
                if self._send_scout(curr_best_sltn_index, visited_indexes, foragers_on_solution):
                    scouts_sent += 1
                    foragers_sent += foragers_on_solution
                    self._sort_solutions()
        self._sort_solutions()
        print('Iteration: ' + str(i + 1))
        print('Best number of colors:', self.solutions[0].colors_num)
    return self.solutions[0]

def _send_scout(self, curr_best_sltn_index: int, visited_indexes: list[int], foragers_sent: int):
    """Відправляє бджолу-розвідника на ділянку (рішення).
    Повертає True, якщо ділянку ще не відвідували інші і
    на неї вдалося відправити розвідника."""
    if np.random.random_sample() > self.probability:
        if curr_best_sltn_index in visited_indexes:
            curr_best_sltn_index += 1
            return False
        visited_indexes.append(curr_best_sltn_index)
        self._send_foragers(curr_best_sltn_index, foragers_sent)
    else:
        index = np.random.randint(self.solutions_num)
        if index in visited_indexes:
            return False
        visited_indexes.append(index)
        self._send_foragers(index, foragers_sent)
    return True

def _send_foragers(self, sltn_index: int, foragers_sent: int):
    """Відправляє фуражирів на ділянку для дослідження її околу."""
    neighbor_solution = self.solutions[sltn_index].get_neighbor_solution(foragers_sent)
    self.solutions[sltn_index] = neighbor_solution
```

3.2.2 Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми.

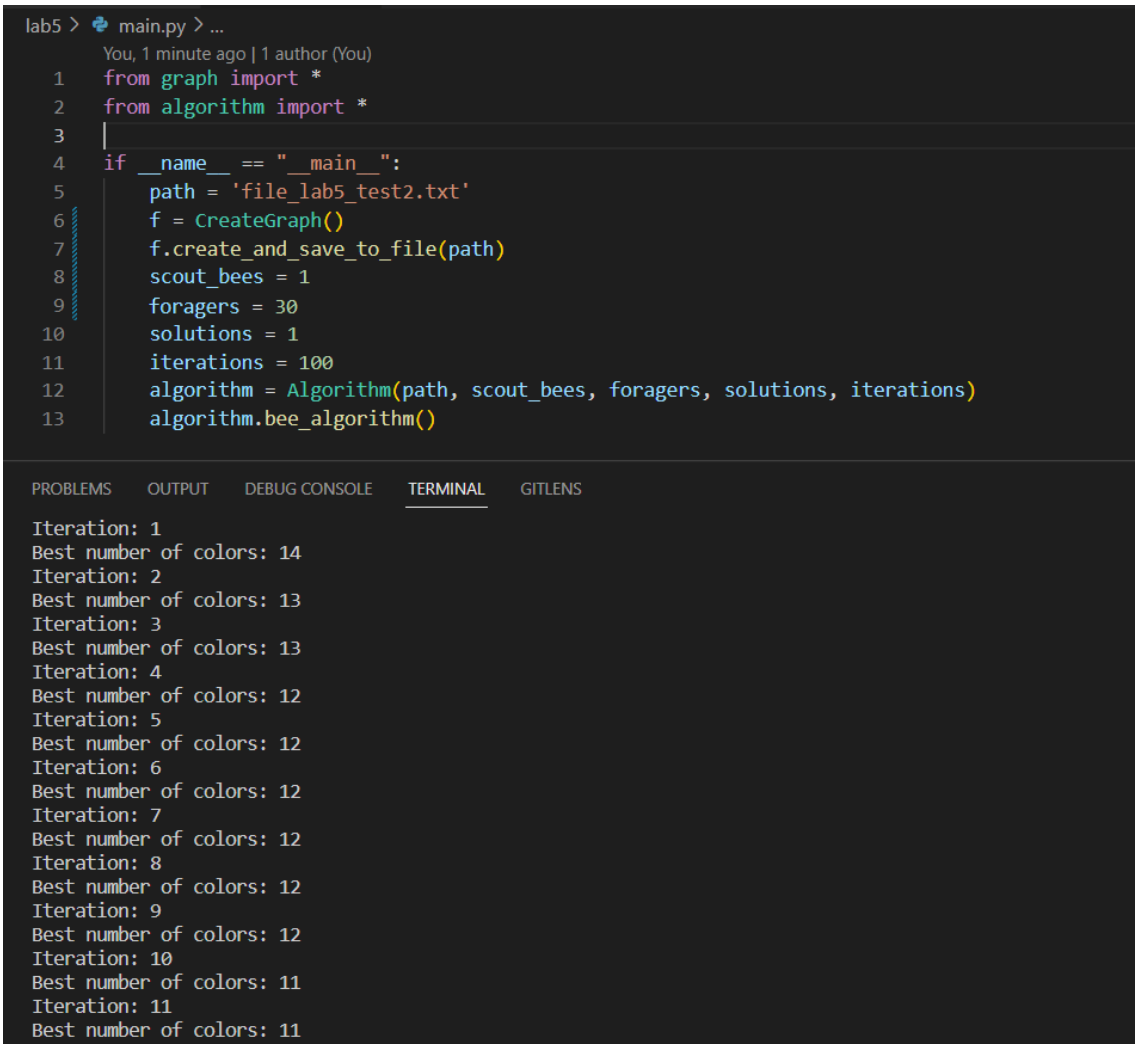


Рисунок 3.1 – Приклад роботи програми при вхідних даних, вказаних у верхній частині рисунка

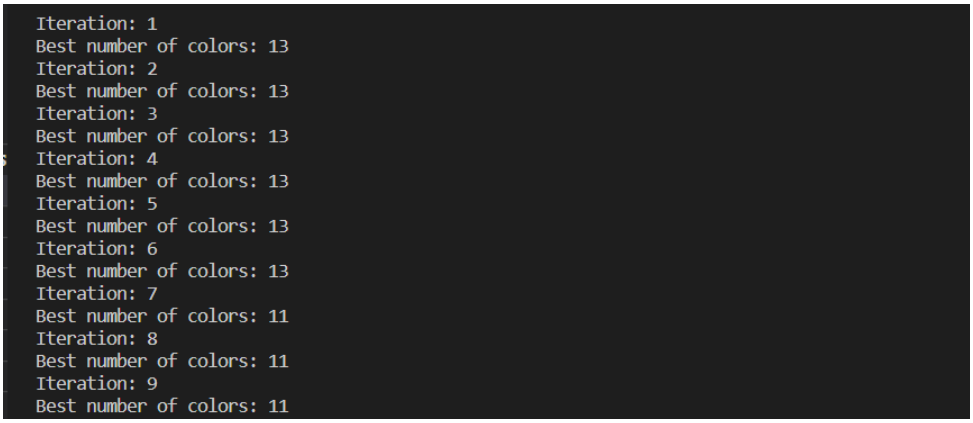


Рисунок 3.2 – Інший приклад роботи програми для тих самих вхідних даних

3.3 Тестування алгоритму

Перед початком проведення тестування був згенерований граф за визначеними умовами (300 вершин, мінімальний степінь – 2, максимальний – 30), згенерований граф буде залишатися сталим для кожного з наступних тестів.

Критерієм зупинки алгоритму було обрано значення цільової функції – 11 кольорів у розфарбуванні графу. Дослідження буде проведено на основі кількості ітерацій, зроблених алгоритмом для досягнення встановленого критерію.

Дослідження буде проводитися для наступних параметрів: кількість ділянок (рішень), кількість бджіл-розвідників та кількість бджіл-фуражирів. Для початку зафіксуємо значення кількості бджіл (розвідників – 2, фуражирів – 60) та будемо змінювати число ділянок, поки не буде досягнуто пікової ефективності алгоритму. Результат даного етапу наведений у таблиці 3.1.

Таблиця 3.1 – Пошук оптимального числа ділянок

№ спроби	Кількість ділянок	Кількість ітерацій	Середня кількість ітерацій
1	1	>100	>100
1	2	12	12,25
2		20	
3		10	
4		7	
1	3	24	16,25
2		20	
3		10	
4		11	
1	4	11	28,3
2		54	
3		20	
1	5	10	14
2		16	
3		16	

Продовження таблиці 3.1

№ спроби	Кількість ділянок	Кількість ітерацій	Середня кількість ітерацій
1	6	31	21,5
2		30	
3		7	
4		18	
1	7	10	38,6
2		19	
3		87	
1	8	24	47,3
2		18	
3		>100	

На рисунку 3.1 наведений графік, що наочно відображає, яка кількість ділянок є найоптимальнішою при таких вхідних значеннях.

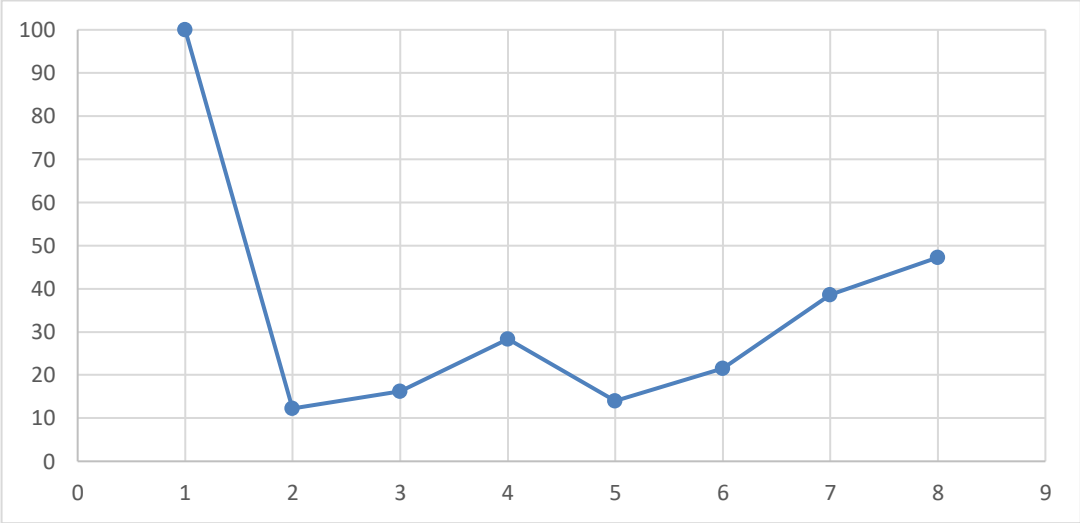


Рисунок 3.1 – Графік залежності кількості ітерацій від кількості ділянок

Із таблиці та побудованого графіку видно, що найкращою у даній ситуації є кількість ділянок, рівна 2.

Тепер зафіксуємо значення кількості ділянок (знайдене на попередньому кроці – 2) та бджіл-фуражирів (початково зафіксоване – 60) і будемо змінювати число бджіл-розвідників. У таблиці 3.2 наведені результати даного етапу.

Побудований алгоритм передбачає, що, якщо кількість бджіл-розвідників перевищує кількість ділянок, то зайві розвідники не будуть використані. Тому перевіряти кількість розвідників більшу за 2 немає сенсу.

Таблиця 3.2 – Пошук оптимального числа бджіл-розвідників

№ спроби	Кількість розвідників	Кількість ітерацій	Середня кількість ітерацій
1	1	9	53
2		28	
3		>100	
4		75	
1	2	10	12
2		11	
3		7	
4		20	

Тепер визначимо оптимальну кількість бджіл-фуражирів. Зафіксуємо значення кількості ділянок (2) та розвідників (2). Результат даного етапу представлений у таблиці 3.3 та на рисунку 3.2.

Оскільки кількість фуражирів, відправлених на кожну ділянку, рівна загальній кількості фуражирів, поділеній на кількість розвідників, а максимальний степінь вершини заданого графу – 30, оптимальне число фуражирів на одну ділянку краще за все шукати в околі даного значення. Також врахуємо, що визначена кількість розвідників – 2, тому оптимальне значення шукатимемо в околі 30 * 2 = 60 фуражирів (від 50 до 70).

Таблиця 3.3 – Пошук оптимального числа бджіл-фуражирів

№ спроби	Кількість фуражирів	Кількість ітерацій	Середня кількість ітерацій
1	50	15	11
2		7	
1	51	9	11,5
2		14	
1	52	17	16
2		15	
1	53	10	10,5
2		11	
1	54	9	14
2		19	
1	55	23	17,5
2		12	

Продовження таблиці 3.3

№ спроби	Кількість фуражирів	Кількість ітерацій	Середня кількість ітерацій
1	56	7	18,5
2		30	
1	57	24	21,5
2		19	
1	58	9	10,5
2		12	
1	59	14	18,5
2		23	
1	60	11	9,5
2		8	
1	61	8	17,5
2		27	
1	62	12	12,5
2		13	
1	63	45	34
2		23	
1	64	17	17,5
2		18	
1	65	29	25
2		21	
1	66	24	17,5
2		11	
1	67	6	10,5
2		15	
1	68	19	17,5
2		16	
1	69	14	13
2		12	
1	70	18	13
2		8	

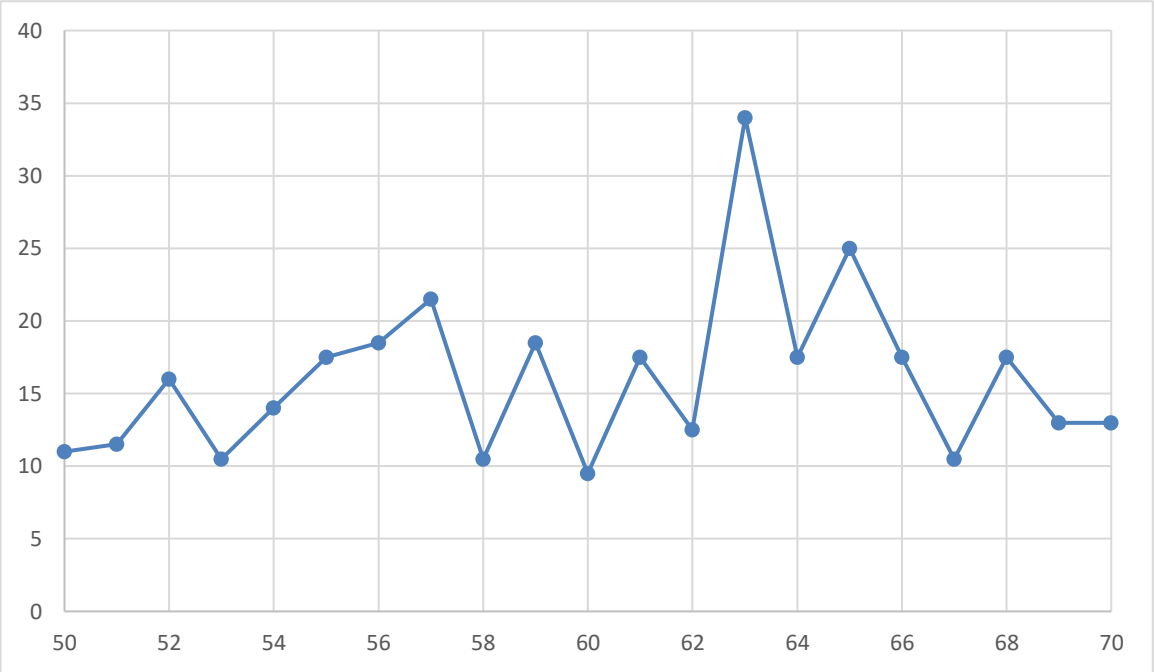


Рисунок 3.2 – Графік залежності кількості ітерацій від кількості фуражирів

Можна зробити висновок, що неможливо точно визначити найоптимальніші вхідні параметри для даного алгоритму, оскільки результат його роботи напряму залежить від початково згенерованих ділянок (рішень) і може сильно різнитися для однакових вхідних параметрів. Але за результатом виконання даного етапу можна обрати оптимальне число фуражирів – 60.

ВИСНОВОК

В рамках даної лабораторної роботи було вивчено основні підходи до розробки метаевристичних алгоритмів для типових прикладних задач. Також було опрацьовано методологію підбору прийнятних параметрів для алгоритму.

У ході роботи було розроблено детальний алгоритм для розв’язання задачі розфарбовування графу з використанням бджолиної колонії (класичним бджолиним алгоритмом). Було здійснено програмну реалізацію побудованого алгоритму мовою програмування Python.

Далі готову програму було протестовано на різних вхідних параметрах, у результаті чого було отримано найбільш оптимальні значення: кількість ділянок – 2, бджіл-розвідників – 2, бджіл-фуражирів – 60. Також було визначено, що якість розв’язку задачі більшою мірою залежить не від вхідних параметрів (таких як кількість бджіл та ділянок), а саме від згенерованих програмою початкових ділянок. Звичайно, у даному випадку ймовірність отримати точний розв’язок більша, якщо використовувати більшу кількість ділянок (так як оптимальний розв’язок може згенеруватися жадібним алгоритмом і бути одразу обраним як найкращий), але якщо число ділянок невелике, вони будуть частіше змінюватися і зменшиться шанс зайти в локальний мінімум цільової функції і ніколи не знайти глобальний мінімум або ж знайти його лише через дуже велику кількість ітерацій.

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%.