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 Недельчев Євген Олександрович* _____
(шифр, прізвище, ім'я, по батькові)

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

Київ 2022

МЕТА ЛАБОРАТОРНОЇ РОБОТИ

Мета роботи – вивчити основні підходи формалізації метаевристичних алгоритмів і вирішення типових задач з їхньою допомогою.

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**
18	Задача розфарбовування графу (300 вершин, степінь вершини не більше 50, але не менше 1), класичний бджолиний алгоритм (число бджіл 60 із них 5 розвідники).

ВИКОНАННЯ

Варіант 18

Програмна реалізація алгоритму

bee_colony:

```
from utils import *
from constant import *


class BeeColony:
    def __init__(self, coloring, f):
        self.coloring = coloring
        self.f = f

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

    @staticmethod
    def initialize_scouts(colony_graph, colony_scouts, n):
        while colony_scouts.qsize() < n:
            colony_scout = BeeColony(*initial_coloring(colony_graph))
            if colony_scout not in colony_scouts.queue:
                colony_scouts.put(colony_scout)

    @staticmethod
    def select_best_scouts(colony_scouts):
        best = []
        for i in range(BEST_SCOUTS_COUNT):
            best.append(colony_scouts.get())
        return best

    @staticmethod
    def _get_queue(colony_graph, local_foragers_n):
        queue = []
        for node, neighbours in sorted(list(colony_graph.items()), key=lambda
x: len(x[1]), reverse=True):
            if len(queue) >= local_foragers_n:
                break
            for neighbour in neighbours:
                queue.append((node, neighbour))
        return queue

    @staticmethod
    def _get_results(colony_scout, queue, colony_graph):
        results = []
        for node, neighbour in queue:
            new_coloring = dict(colony_scout.coloring)
            new_coloring[neighbour], new_coloring[node] = new_coloring[node],
new_coloring[neighbour]

            if are_colors_in_neighbours(new_coloring[node],
colony_graph[node], new_coloring) or \
                    are_colors_in_neighbours(new_coloring[neighbour],
colony_graph[neighbour], new_coloring):
                continue
            else:
```

```
                for color in range(1, colony_scout.f + 1):
                    if not are_colors_in_neighbours(color,
colony_graph[neighbour], new_coloring):
                        new_coloring[neighbour] = color
                        results.append(BeeColony(new_coloring,
len(set(new_coloring.values()))))
        return results

    @staticmethod
    def discover_neighbours(colony_scout, colony_graph, local_foragers_n):
        queue = BeeColony._get_queue(colony_graph, local_foragers_n)
        results = BeeColony._get_results(colony_scout, queue, colony_graph)
        return min(results, key=lambda x: x.f) if results else colony_scout

    @staticmethod
    def perform_local_search(colony_graph, new_colony_scouts, colony_scouts,
n):
        for colony_scout in colony_scouts:
            colony_scout = BeeColony.discover_neighbours(colony_scout,
colony_graph, n)
            new_colony_scouts.put(colony_scout)

    @staticmethod
    def add_best_scouts_to_queue(colony_scouts, best_colony_scouts):
        for colony_scout in best_colony_scouts:
            colony_scouts.put(colony_scout)
```

main:

```
from queue import PriorityQueue
from bee_colony import BeeColony
from constant import *
from utils import *


def print_best_scout(scouts, iteration):
    if not iteration % 20:
        best = scouts.get()
        print(f"Iter: [{iteration}], best chromatic number: [{best.f}]")
        scouts.put(best)


def print_best_coloring(scouts):
    best = scouts.get()
    coloring = {k: v for k, v in sorted(best.coloring.items(), key=lambda
item: item[0])}
    print(f"Best chromatic number: [{best.f}], graph coloring: {coloring}")


def main():
    graph = generate_random_graph()

    scouts = PriorityQueue()
    BeeColony.initialize_scouts(graph, scouts, INITIAL_SCOUTS_COUNT)

    for i in range(1000):
        if not i % 20:
            print_best_scout(scouts, i)

        best_scouts = BeeColony.select_best_scouts(scouts)
        random_scouts = scouts.queue

        # Create new scouts queue
        scouts = PriorityQueue()
```

```
        # Local search for best foragers
        BeeColony.perform_local_search(graph, scouts, best_scouts,
BEST_FORAGERS_COUNT)

        # Local search for random foragers
        BeeColony.perform_local_search(graph, scouts, random_scouts,
RANDOM_FORAGERS_COUNT)

        # Find best scouts
        best_scouts = BeeColony.select_best_scouts(scouts)

        # Create new scouts queue
        scouts = PriorityQueue()

        # Generate random scouts
        BeeColony.initialize_scouts(graph, scouts, RANDOM_SCOUTS_COUNT)

        # Add best scouts to the new scouts queue
        BeeColony.add_best_scouts_to_queue(scouts, best_scouts)

    print_best_coloring(scouts)


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

utils:

```
import networkx as nx
from random import shuffle


def generate_random_graph():
    graph_generator = nx.watts_strogatz_graph(300, 30, 1)
    graph = {}
    for edge in graph_generator.edges:
        u, v = edge
        u += 1
        v += 1
        if u not in graph:
            graph[u] = []
        if v not in graph:
            graph[v] = []
        graph[u].append(v)
        graph[v].append(u)
    return graph


def are_colors_in_neighbours(color, neighbours, coloring):
    if neighbours:
        for neighbour in neighbours:
            if neighbour in coloring and coloring[neighbour] == color:
                return True
    return False


def initial_coloring(graph):
    coloring = {node: 0 for node in graph.keys()}
    graph = list(graph.items())
    shuffle(graph)
    max_color = 0
    for node, neighbours in graph:
        for color in range(1, 10000):
            if not are_colors_in_neighbours(color, neighbours, coloring):
```

```
                coloring[node] = color
                if color > max_color:
                    max_color += 1
                break
    return coloring, max_color
```

Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми.

```
Iter: [0], best chromatic number: [13]
Iter: [20], best chromatic number: [13]
Iter: [40], best chromatic number: [13]
Iter: [60], best chromatic number: [13]
Iter: [80], best chromatic number: [13]
Iter: [100], best chromatic number: [13]
Iter: [120], best chromatic number: [13]
Iter: [140], best chromatic number: [13]
Iter: [160], best chromatic number: [13]
Iter: [180], best chromatic number: [13]
Iter: [200], best chromatic number: [13]
Iter: [220], best chromatic number: [13]
Iter: [240], best chromatic number: [13]
Iter: [260], best chromatic number: [13]
Iter: [280], best chromatic number: [13]
Iter: [300], best chromatic number: [13]
Iter: [320], best chromatic number: [13]
Iter: [340], best chromatic number: [13]
Iter: [360], best chromatic number: [13]
Iter: [380], best chromatic number: [13]
Iter: [400], best chromatic number: [13]
Iter: [420], best chromatic number: [13]
Iter: [440], best chromatic number: [13]
Iter: [460], best chromatic number: [13]
Iter: [480], best chromatic number: [13]
Iter: [500], best chromatic number: [13]
Iter: [520], best chromatic number: [13]
Iter: [540], best chromatic number: [13]
Iter: [560], best chromatic number: [13]
```

```
Iter: [580], best chromatic number: [13]
Iter: [600], best chromatic number: [13]
Iter: [620], best chromatic number: [13]
Iter: [640], best chromatic number: [13]
Iter: [660], best chromatic number: [13]
Iter: [680], best chromatic number: [12]
Iter: [700], best chromatic number: [12]
Iter: [720], best chromatic number: [12]
Iter: [740], best chromatic number: [12]
Iter: [760], best chromatic number: [12]
Iter: [780], best chromatic number: [12]
Iter: [800], best chromatic number: [12]
Iter: [820], best chromatic number: [12]
Iter: [840], best chromatic number: [12]
Iter: [860], best chromatic number: [12]
Iter: [880], best chromatic number: [12]
Iter: [900], best chromatic number: [12]
Iter: [920], best chromatic number: [12]
Iter: [940], best chromatic number: [12]
Iter: [960], best chromatic number: [12]
Iter: [980], best chromatic number: [12]
```

```
Best chromatic number: [12], graph coloring: {1: 2, 2: 7, 3: 10, 4: 6, 5: 8, 6: 1, 7: 11, 8: 12, 9: 6, 10: 8, 11: 1, 12: 7, 13: 4, 14: 1, 15: 6, 16: 5, 17: 6, 18: 3, 19: 8,
 20: 4, 21: 11, 22: 6, 23: 6, 24: 8, 25: 1, 26: 4, 27: 9, 28: 1, 29: 2, 30: 4, 31: 8, 32: 6, 33: 10, 34: 3, 35: 4, 36: 12, 37: 6, 38: 8, 39: 8, 40: 9, 41: 4, 42: 9, 43: 11,
 44: 3, 45: 1, 46: 6, 47: 2, 48: 10, 49: 3, 50: 2, 51: 9, 52: 5, 53: 3, 54: 3, 55: 4, 56: 4, 57: 2, 58: 1, 59: 4, 60: 8, 61: 12, 62: 8, 63: 5, 64: 2, 65: 8, 66: 3, 67: 5, 68:
 3, 69: 12, 70: 11, 71: 5, 72: 6, 73: 2, 74: 2, 75: 11, 76: 1, 77: 3, 78: 5, 79: 6, 80: 6, 81: 7, 82: 5, 83: 8, 84: 11, 85: 3, 86: 3, 87: 2, 88: 9, 89: 4, 90: 7, 91: 8, 92: 7,
  93: 2, 94: 6, 95: 8, 96: 7, 97: 7, 98: 10, 99: 4, 100: 3, 101: 6, 102: 9, 103: 5, 104: 5, 105: 6, 106: 10, 107: 5, 108: 2, 109: 2, 110: 3, 111: 1, 112: 2, 113: 6, 114: 10,
 115: 9, 116: 2, 117: 9, 118: 3, 119: 1, 120: 10, 121: 1, 122: 5, 123: 4, 124: 12, 125: 5, 126: 9, 127: 10, 128: 1, 129: 4, 130: 2, 131: 1, 132: 1, 133: 5, 134: 6, 135: 3,
 136: 1, 137: 3, 138: 9, 139: 2, 140: 5, 141: 11, 142: 2, 143: 4, 144: 11, 145: 7, 146: 11, 147: 1, 148: 1, 149: 5, 150: 3, 151: 3, 152: 4, 153: 3, 154: 7, 155: 8, 156: 7,
 157: 6, 158: 4, 159: 1, 160: 4, 161: 4, 162: 12, 163: 5, 164: 3, 165: 2, 166: 3, 167: 5, 168: 10, 169: 6, 170: 4, 171: 2, 172: 8, 173: 4, 174: 7, 175: 11, 176: 5, 177: 7,
 178: 1, 179: 2, 180: 2, 181: 5, 182: 5, 183: 9, 184: 7, 185: 10, 186: 5, 187: 1, 188: 10, 189: 3, 190: 7, 191: 2, 192: 4, 193: 1, 194: 1, 195: 6, 196: 11, 197: 6, 198: 2,
 199: 6, 200: 1, 201: 10, 202: 1, 203: 4, 204: 6, 205: 9, 206: 3, 207: 7, 208: 6, 209: 10, 210: 5, 211: 12, 212: 5, 213: 7, 214: 7, 215: 10, 216: 7, 217: 3, 218: 6, 219: 5,
 220: 12, 221: 2, 222: 5, 223: 7, 224: 11, 225: 1, 226: 4, 227: 3, 228: 10, 229: 7, 230: 6, 231: 6, 232: 7, 233: 3, 234: 2, 235: 11, 236: 11, 237: 5, 238: 4, 239: 3, 240: 1,
 241: 1, 242: 7, 243: 2, 244: 11, 245: 6, 246: 9, 247: 3, 248: 2, 249: 2, 250: 6, 251: 6, 252: 1, 253: 9, 254: 7, 255: 9, 256: 5, 257: 8, 258: 9, 259: 4, 260: 8, 261: 8, 262:
 5, 263: 7, 264: 9, 265: 10, 266: 1, 267: 3, 268: 9, 269: 6, 270: 9, 271: 4, 272: 5, 273: 8, 274: 4, 275: 1, 276: 5, 277: 11, 278: 9, 279: 7, 280: 1, 281: 9, 282: 7, 283: 5,
 284: 4, 285: 1, 286: 12, 287: 8, 288: 3, 289: 10, 290: 4, 291: 3, 292: 8, 293: 10, 294: 9, 295: 8, 296: 2, 297: 5, 298: 7, 299: 8, 300: 1}
```

Рисунок 3.1 – Приклад роботи програми для випадкового графу

Рисунок 3.2 – Приклад роботи програми для випадкового графу

Тестування алгоритму

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
0	14	520	12
20	13	540	12
40	13	560	12
60	13	580	12
80	13	600	12
100	13	620	12
120	13	640	12
140	13	660	12
160	13	680	12
180	13	700	12
200	13	720	12
220	13	740	12
240	13	760	12
260	13	780	12
280	13	800	12
300	13	820	12
320	13	840	12
340	13	860	12
360	12	880	12
380	12	900	12
400	12	920	12
420	12	940	12
440	12	960	12
460	12	980	12
480	12	1000	12
500	12		

3.2.2 Графіки залежності розв'язку від числа ітерацій

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

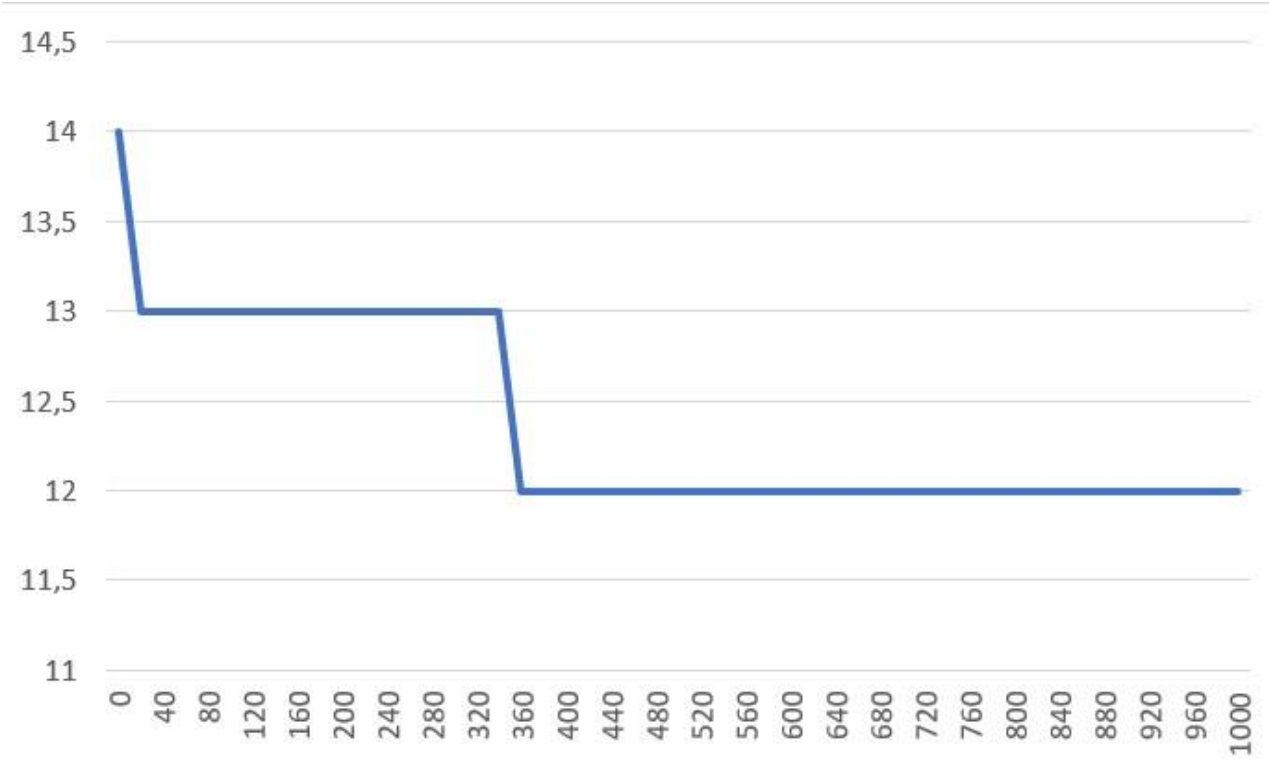

Рисунок 3.3 – Графік залежності розв'язку від числа ітерацій

ВИСНОВОК

В рамках даної лабораторної роботи було розроблено алгоритм вирішення задачі розфарбування графу класичним бджолиним алгоритмом та виконано його програмну реалізацію на мові програмування Python. В алгоритмі використовуються 5 розвідників, з ділянок, що вони знайшли, обирається 2 найкращі для подальшого пошуку в околиці 20-ма фуражирами кожен, пошук в околиці тих, що лишилися, виконується 5-ма фуражирами кожен. Під час пошуку в околиці кожен фуражир перевіряє, чи можна замінити кольори двох суміжніх вершин, і змінити колір однієї з них на кращий. В якості евристики було вибрано починати з вершини, яка має найбільший степінь.

Було зафіксовано якість отриманого розв’язку після кожних 20 ітерацій до 1000 і побудовано графік залежності якості розв’язку від числа ітерацій. Через специфіку задачі і ефективність евристичного алгоритму початкового розфарбування випадково обираючи вершини і розфарбовуючи першим найкращим кольором, вдалось покращити розв’язок з 14 до 12 кольорів.

Було зроблено висновок, що бджолиний алгоритм може бути використаний в якості метаевристичного алгоритму розв’язання задачі розфарбування графу для знаходження щонайоптимальнішого розв’язку.

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%.