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-12 Мельник М.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 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**
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).

3 ВИКОНАННЯ

Варіант 18

3.1 Програмна реалізація алгоритму

3.1.1 Вихідний код

```
from random import shuffle
from queue import PriorityQueue
import networkx as nx

SCOUTS_N = 5
BEST_SCOUTS_N = 2
RANDOM_SCOUTS_N = SCOUTS_N - BEST_SCOUTS_N
FORAGERS_N = 55
BEST_FORAGERS_N = 20
RANDOM_FORAGERS_N = 5


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

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

   def __repr__(self):
      return f"\nFunction: {self.f} Coloring: {self.coloring}"


def generate_rand_graph():
   g = nx.watts_strogatz_graph(300, 30, 1)
   graph = {}
   for edge in g.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 in_neighbours(color, neighbours, coloring):
   if neighbours:
      for neighbour in neighbours:
         if neighbour in coloring and coloring[neighbour] == color:
            return True
   return False


def random_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 in_neighbours(color, neighbours, coloring):
            coloring[node] = color
            if color > max_color:
```

```
                max_color += 1
            break
    return coloring, max_color


def generate_scouts(graph, scouts, n):
    while scouts.qsize() < n:
        scout = Bee(*random_coloring(graph))
        if scout not in scouts.queue:
            scouts.put(scout)


def find_best_scouts(scouts):
    best = []
    for i in range(BEST_SCOUTS_N):
        best.append(scouts.get())
    return best


def discover(scout, graph, local_foragers_n):
    queue = []

    for node, neighbours in sorted(list(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))

    results = []

    for node, neighbour in queue:
        new_coloring = dict(scout.coloring)
        new_coloring[neighbour], new_coloring[node] = new_coloring[node],
new_coloring[neighbour]

        if in_neighbours(new_coloring[node], graph[node], new_coloring) or \
                in_neighbours(new_coloring[neighbour], graph[neighbour], new_coloring):
            continue
        else:
            for color in range(1, scout.f + 1):
                if not in_neighbours(color, graph[neighbour], new_coloring):
                    new_coloring[neighbour] = color
                    results.append(Bee(new_coloring, len(set(new_coloring.values()))))

    return min(results, key=lambda x: x.f) if results else scout


def local_search(graph, new_scouts, scouts, n):
    for scout in scouts:
        scout = discover(scout, graph, n)
        new_scouts.put(scout)


def add_best_scouts(scouts, best_scouts):
    for scout in best_scouts:
        scouts.put(scout)


def main():
    graph = generate_rand_graph()

    # generate initial scouts
    scouts = PriorityQueue()
    generate_scouts(graph, scouts, SCOUTS_N)

    for i in range(2000):
        if not i % 20:
            best = scouts.get()
            print(f"Iteration {i}, min f: {best.f}")
            scouts.put(best)
```

```
        # find the best scouts and remaining scouts
        best_scouts = find_best_scouts(scouts)
        random_scouts = scouts.queue

        # perform local search
        scouts = PriorityQueue()
        local_search(graph, scouts, best_scouts, BEST_FORAGERS_N)
        local_search(graph, scouts, random_scouts, RANDOM_FORAGERS_N)

        # save best solutions, look for new solutions
        best_scouts = find_best_scouts(scouts)
        scouts = PriorityQueue()
        generate_scouts(graph, scouts, RANDOM_SCOUTS_N)
        add_best_scouts(scouts, best_scouts)

    best = scouts.get()
    coloring = {k: v for k, v in sorted(best.coloring.items(), key=lambda item: item[0])}
    print(f"Best coloring function: {best.f}\nColoring: {coloring}")


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

3.1.2 Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми.

```
Iteration 0, min f: 14
Iteration 20, min f: 13
Iteration 40, min f: 13
Iteration 60, min f: 13
Iteration 80, min f: 13
Iteration 100, min f: 13
Iteration 120, min f: 13
Iteration 140, min f: 13
Iteration 160, min f: 13
Iteration 180, min f: 12
Iteration 200, min f: 12
Iteration 220, min f: 12
Iteration 240, min f: 12
Iteration 260, min f: 12
Iteration 280, min f: 12
Iteration 300, min f: 12
Iteration 320, min f: 12
Iteration 340, min f: 12
Iteration 360, min f: 12
Iteration 380, min f: 12
Iteration 400, min f: 12
Iteration 420, min f: 12
Iteration 440, min f: 12
Iteration 460, min f: 12
Iteration 480, min f: 12
Iteration 500, min f: 12
Iteration 520, min f: 12
Iteration 540, min f: 12
Iteration 560, min f: 12
Iteration 580, min f: 12
Iteration 600, min f: 12
Iteration 620, min f: 12
Iteration 640, min f: 12
Iteration 660, min f: 12
Iteration 680, min f: 12

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

Рисунок 3.1 – Приклад роботи програми для випадкового графу

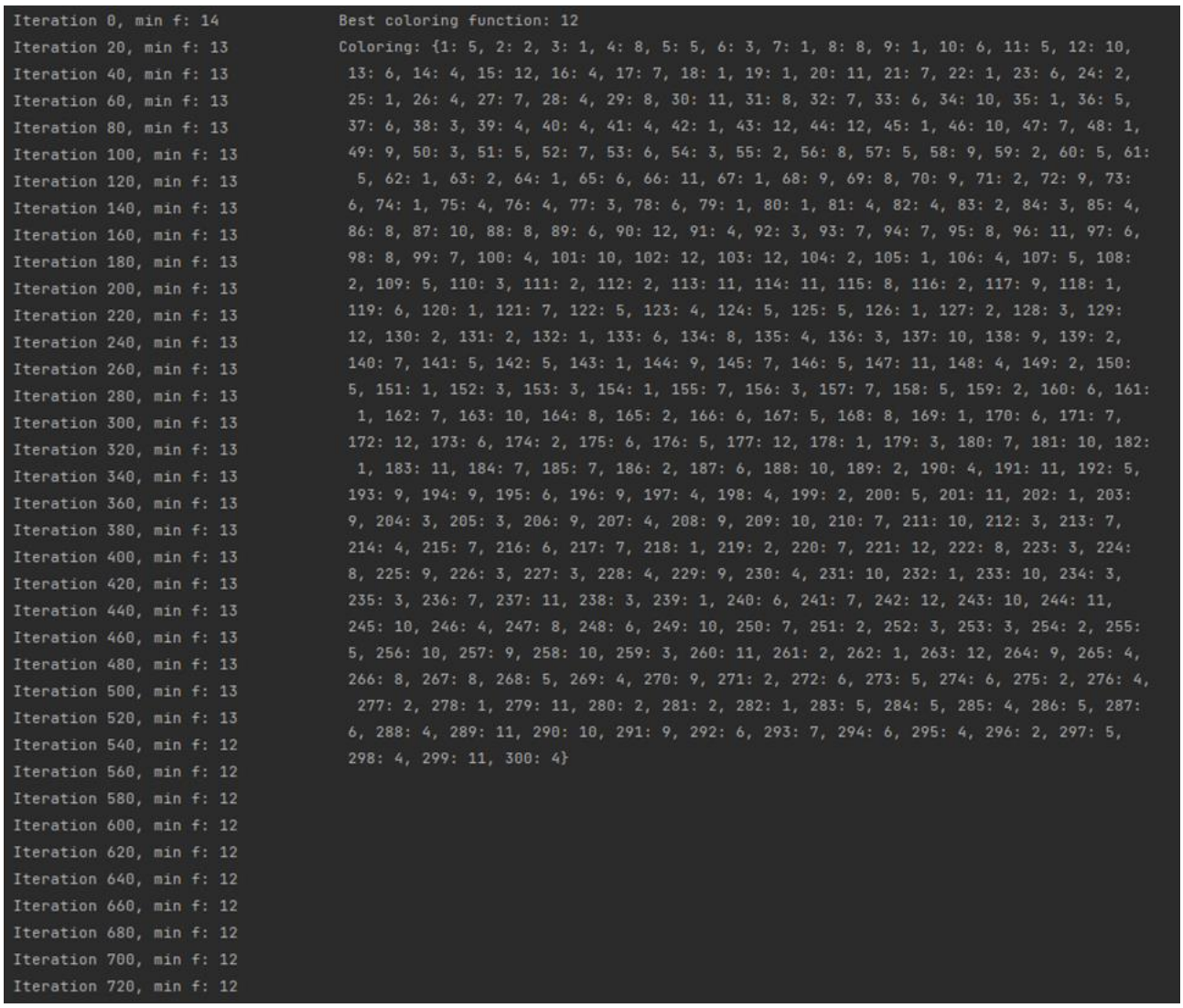

Рисунок 3.2 – Приклад роботи програми для випадкового графу

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
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.

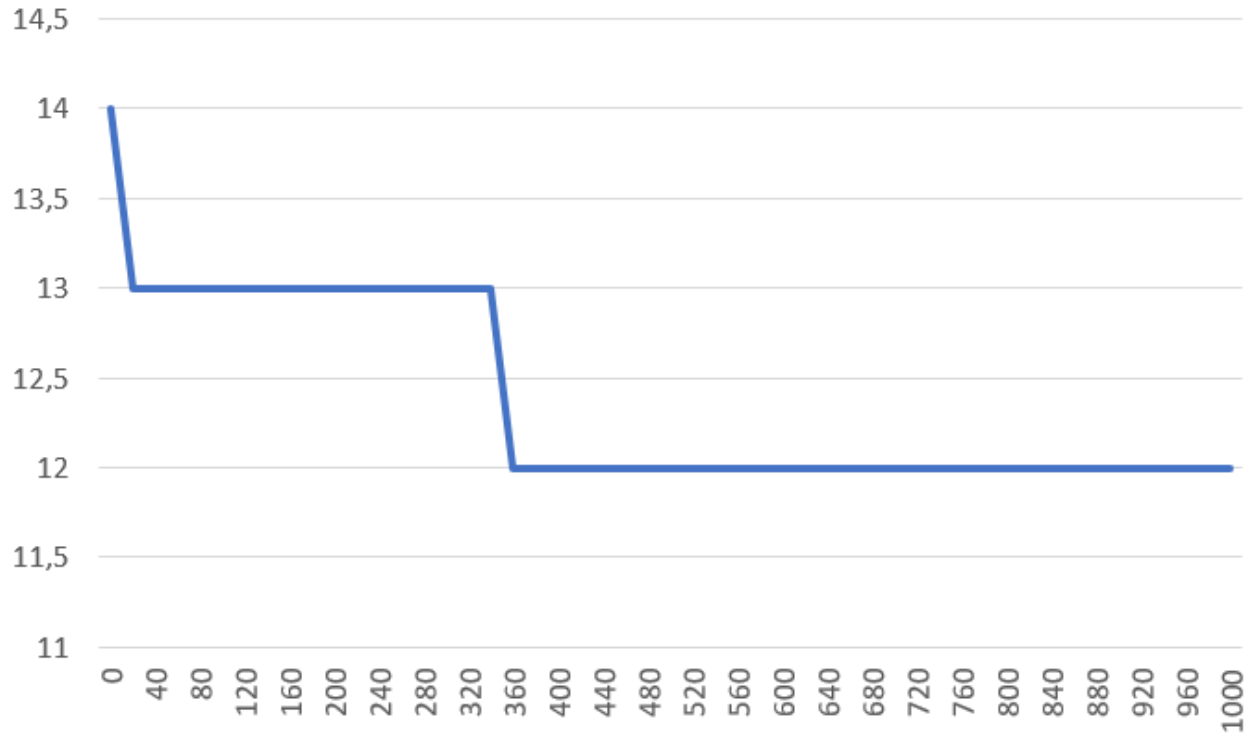

Рисунок 3.3 – Графік залежності розв'язку від числа ітерацій

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%.