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-з11 Химич Володимир Леонідович*
(шифр, прізвище, ім'я, по батькові)

Перевірив *Головченко М.Н.*
(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**
9	Задача розфарбовування графу (150 вершин, степінь вершини не більше 30, але не менше 1), бджолиний алгоритм ABC (число бджіл 25 із них 3 розвідники).

3 Виконання

3.1 Програмна реалізація алгоритму

3.1.1 Вихідний код

```
package Lab4.algorithm;

import Lab4.graph.BeeGraph;
import Lab4.graph.BeeNode;
import Lab4.utility.BeeNodeComparator;
import lombok.Getter;
import lombok.Setter;

import java.util.*;
import java.util.stream.Collectors;

import static Lab4.utility.Constants.*;
import static Lab4.utility.Util.createAllColors;
import static Lab4.utility.Util.createStartIndexes;

@Getter
public class ABCAlgorithm {
    private BeeGraph initGraph;
    private BeeGraph currentBeeGraph;
    @Setter
    private BeeGraph bestBeeGraph;
    private final int foragers = FORAGERS_NUMBER;
    private final int scouts = SCOUTS_NUMBER;
    private int[] allColors;
    private LinkedList<Integer> usedColors = new LinkedList<>();

    public ABCAlgorithm(BeeGraph initialGraph) {
        this.initGraph = initialGraph;
        this.currentBeeGraph = initialGraph.deepCopy();
        this.allColors = createAllColors(MAX_NODE_DEGREE);
    }

    public void resetAlgorithm() {
        this.currentBeeGraph = initGraph.deepCopy();
        this.usedColors = new LinkedList<>();
```

```
        }

        public void runAlgorithm() {
                                    ArrayList<Integer>    unvisitedIndexes    =
createStartIndexes(NODES_NUMBER);
            LinkedList<BeeNode> scouted = new LinkedList<>();
            PriorityQueue<BeeNode> nodesToVisit;
            BeeNode visitingNode;
            BeeNode scoutedNode;

            scoutNodes(unvisitedIndexes, scouted);
            nodesToVisit = findNodesToVisit(scouted);
            nodesToVisit = sortByPriority(nodesToVisit);

            while (nodesToVisit.size() != 0) {
                visitingNode = nodesToVisit.poll();

                paintNode(visitingNode);

                Optional<BeeNode> done = checkIfNodeDone(nodesToVisit, scouted);

                if (done.isPresent()) {
                    paintNode(done.get());
                    scouted.remove(done.get());

                    if (unvisitedIndexes.size() != 0) {
                        scoutedNode = scoutNode(unvisitedIndexes, scouted);

                        nodesToVisit.addAll(scoutedNode.getNeighbours());
                        nodesToVisit = sortByPriority(nodesToVisit);
                    }
                }
            }

            currentBeeGraph.updatePaintedNodesColors();
        }

        private void paintNode(BeeNode node) {
            List<BeeNode> neighbours = node.getNeighbours();

            int foundColor = getAppropriateColorFromUsed(neighbours);
```

```
        if (foundColor == -1) {
            foundColor = genNewColorFromAll();
        }

        currentBeeGraph.getNodes().get(node.getIndex()).setColor(foundColor);
    }

    private int genNewColorFromAll() {
        int random;

        while (true) {
            random = RAND.nextInt(MAX_NODE_DEGREE);
            if (usedColors.contains(random)) {
                continue;
            }
            usedColors.add(random);
            return random;
        }

    }

    private int getAppropriateColorFromUsed(List<BeeNode> neighbours) {
        boolean isSutable;

        for (Integer used : usedColors) {
            isSutable = true;
            for (BeeNode beeNode : neighbours) {
                if (beeNode.getColor() == used) {
                    isSutable = false;
                }
            }
            if (isSutable) {
                return used;
            }
        }

        return -1;
    }
```

```
    private Optional<BeeNode> checkIfNodeDone(PriorityQueue<BeeNode>
nodesToVisit, LinkedList<BeeNode> scoutedNodes) {

        boolean isPresent;
        for (BeeNode scouted : scoutedNodes) {
            isPresent = false;
            for (BeeNode toVisit : nodesToVisit) {
                if (scouted.hasNeighbour(toVisit)) {
                    isPresent = true;
                }
            }
            if (!isPresent) {
                return Optional.of(scouted);
            }
        }

        return Optional.empty();
    }

    private PriorityQueue<BeeNode> findNodesToVisit(LinkedList<BeeNode>
discoveredNodes) {
        PriorityQueue<BeeNode> nodesToVisit = new PriorityQueue<>();

        float pollenSum = 0;

        for (BeeNode node : discoveredNodes) {
            nodesToVisit.addAll(node.getNeighbours());
        }

        for (BeeNode node : nodesToVisit) {
            pollenSum += node.getDegree();
        }

        for (BeeNode node : nodesToVisit) {
            node.setPollenValue((float) node.getDegree() / pollenSum);
        }

        return nodesToVisit;
    }
```

```
            private  PriorityQueue<BeeNode>  sortByPriority(PriorityQueue<BeeNode>
nodesToVisit) {
            return nodesToVisit.stream().sorted(new BeeNodeComparator())
                    .collect(Collectors.toCollection(PriorityQueue::new));
        }

              private   BeeNode   scoutNode(ArrayList<Integer>   unvisitedIndexes,
LinkedList<BeeNode> scoutedNodes) {
            BeeNode richestNode;
            int foundNodesNumber;
            int random;
            int index;

            foundNodesNumber = (NODES_NUMBER - unvisitedIndexes.size()) % 3;

            if (foundNodesNumber == 0) {
                richestNode = currentBeeGraph.findRichestNode(unvisitedIndexes);
                scoutedNodes.add(richestNode);
                unvisitedIndexes.remove((Integer) richestNode.getIndex());
                return richestNode;
            } else {
                random = RAND.nextInt(unvisitedIndexes.size());

scoutedNodes.add(currentBeeGraph.getNodes().get(unvisitedIndexes.get(random)));
                index = unvisitedIndexes.get(random);
                unvisitedIndexes.remove(random);

                return currentBeeGraph.getNodes().get(index);
            }
        }

              private   void   scoutNodes(ArrayList<Integer>   unvisitedIndexes,
LinkedList<BeeNode> scoutedNodes) {
            ArrayList<BeeNode> currentNodes = currentBeeGraph.getNodes();

            BeeNode richestNode;
            int foundNodesNumber = (NODES_NUMBER - unvisitedIndexes.size()) % 3;
            int random;

            if (foundNodesNumber == 0) {
                richestNode = currentBeeGraph.findRichestNode(unvisitedIndexes);
```

```
                scoutedNodes.add(richestNode);
                unvisitedIndexes.remove((Integer) richestNode.getIndex());
                ++foundNodesNumber;
            }

            while (foundNodesNumber != scouts) {
                random = RAND.nextInt(unvisitedIndexes.size());
                scoutedNodes.add(currentNodes.get(unvisitedIndexes.get(random)));
                unvisitedIndexes.remove(random);
                ++foundNodesNumber;
            }
        }
    }
    package Lab4.graph;

    import lombok.Getter;

    import java.util.*;

    import static Lab4.utility.Constants.*;

    @Getter
    public class BeeGraph {
        private ArrayList<BeeNode> nodes = new ArrayList<>();
        private int[] paintedNodesColors;

        public BeeGraph() {
            fullInGraph(nodes);
            createNodeRelations(nodes);

            paintedNodesColors = new int[nodes.size()];
            Arrays.fill(paintedNodesColors, INIT_COLOR);
        }

        private BeeGraph(int[] paintedNodesColors) {
            this.paintedNodesColors = paintedNodesColors;
        }

        public BeeNode findRichestNode(ArrayList<Integer> indexes) {
            Integer bestIndex = 0;
            int bestDegree = 0;
```

```
        for (Integer index : indexes) {
            if (nodes.get(index).getDegree() > bestDegree) {
                bestIndex = index;
            }
        }

        return nodes.get(bestIndex);
    }

    private void fullInGraph(ArrayList<BeeNode> graph) {
        for (int i = 0; i < NODES_NUMBER; i++) {
            graph.add(new BeeNode(i));
        }
    }

    private void createNodeRelations(ArrayList<BeeNode> graph) {
        int randomizedDegree;
        int randomRightNodeIndex;
        BeeNode leftNode;
        BeeNode rightNode;

        for (int i = 0; i < graph.size(); i++) {
            randomizedDegree = 1;
            first:
            while (randomizedDegree > 0) {
                randomRightNodeIndex = RAND.nextInt(NODES_NUMBER);
                rightNode = graph.get(randomRightNodeIndex);
                if (rightNode.getNeighbours().size() < MAX_NODE_DEGREE) {

                    leftNode = graph.get(i);

                    if (leftNode.getNeighbours().size() > 0) {
                        for (BeeNode child : leftNode.getNeighbours()) {
                            if (child.getIndex() == rightNode.getIndex()) {
                                continue first;
                            }
                        }
                    }
                    leftNode.addNeighbour(rightNode);
                    rightNode.addNeighbour(leftNode);
```

```
                        --randomizedDegree;
                    }
                }
            }

            int randomNumberOfNodesToBeModified = RAND.nextInt(NODES_NUMBER);
            int leftNodeIndex;

            while (randomNumberOfNodesToBeModified > 0) {
                leftNodeIndex = RAND.nextInt(NODES_NUMBER);
                                                        randomizedDegree     =
countRandomDegree(graph.get(leftNodeIndex).getNeighbours().size());
                second:
                while (randomizedDegree > 0) {
                    randomRightNodeIndex = RAND.nextInt(NODES_NUMBER);
                    rightNode = graph.get(randomRightNodeIndex);
                    if (rightNode.getNeighbours().size() < MAX_NODE_DEGREE) {

                        leftNode = graph.get(leftNodeIndex);

                        if (leftNode.getNeighbours().size() > 0) {
                            for (BeeNode child : leftNode.getNeighbours()) {
                                if (child.getIndex() == rightNode.getIndex()) {
                                    continue second;
                                }
                            }
                        }

                        leftNode.addNeighbour(rightNode);
                        rightNode.addNeighbour(leftNode);
                        --randomizedDegree;
                    }
                }
                --randomNumberOfNodesToBeModified;
            }
        }

        private int countRandomDegree(int currentDegree) {
            int randomDegree = RAND.nextInt(MAX_NODE_DEGREE - MIN_NODE_DEGREE) +
MIN_NODE_DEGREE;
```

```
            if (currentDegree < randomDegree) {
                return randomDegree - currentDegree;
            }
            return 0;
        }

        public BeeGraph deepCopy() {
            int[] coloredByCopy = paintedNodesColors.clone();
            BeeGraph copy = new BeeGraph(coloredByCopy);
            fullInGraph(copy.getNodes());
            copyRelationsTo(copy);

            return copy;
        }

        private void copyRelationsTo(BeeGraph copyGraph) {
            ArrayList<BeeNode> copyNodes = copyGraph.getNodes();
            BeeNode copyNode;
            BeeNode currentNode;
            List<BeeNode> currentNeighbours;
            int currentNeighbourIndex;
            BeeNode copyRightNode;
            boolean containsSuchIndex;

            for (int i = 0; i < copyNodes.size(); i++) {
                copyNode = copyNodes.get(i);
                currentNode = this.nodes.get(i);
                copyNode.setUsed(currentNode.isUsed());
                copyNode.setColor(currentNode.getColor());

                currentNeighbours = currentNode.getNeighbours();

                for (int j = 0; j < currentNeighbours.size(); j++) {
                    currentNeighbourIndex = currentNeighbours.get(j).getIndex();
                    containsSuchIndex = false;

                    if (copyNode.getNeighbours().size() > 0) {
                        for (BeeNode copyNeighbour : copyNode.getNeighbours()) {
                                                    if  (copyNeighbour.getIndex()  ==
currentNeighbourIndex) {
                                containsSuchIndex = true;
```

```
                        }
                    }
                }

                if (!containsSuchIndex) {
                    copyRightNode = copyNodes.get(currentNeighbourIndex);
                    copyNode.addNeighbour(copyRightNode);
                    copyRightNode.addNeighbour(copyNode);
                }
            }
        }
    }

    public void updatePaintedNodesColors() {
        for (int i = 0; i < NODES_NUMBER; i++) {
            paintedNodesColors[i] = nodes.get(i).getColor();
        }
    }

    public int getChromaticNumber() {
        Set<Integer> uniqueColors = new HashSet<>();

        for (BeeNode node : nodes) {
            uniqueColors.add(node.getColor());
        }

        return uniqueColors.size();
    }
}
```

```
package Lab4.graph;

import lombok.Getter;
import lombok.Setter;

import java.util.LinkedList;
import java.util.List;
import java.util.Objects;

import static Lab4.utility.Constants.INIT_COLOR;

@Getter
```

```java
@Setter
public class BeeNode implements Comparable<BeeNode> {
    private int index;
    private List<BeeNode> neighbours = new LinkedList<>();
    private int color = INIT_COLOR;

    private int degree = 0;

    private boolean isUsed = false;

    private float pollenValue;

    public BeeNode(int index) {
        this.index = index;
    }

    public void addNeighbour(BeeNode neighbour) {
        this.neighbours.add(neighbour);
        degree++;
    }

    public boolean hasNeighbour(BeeNode node) {
        return this.neighbours.contains(node);
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        BeeNode node = (BeeNode) o;
        return index == node.index;
    }

    @Override
    public int hashCode() {
        return Objects.hash(index);
    }

    @Override
    public int compareTo(BeeNode o) {
        if (this.pollenValue > o.getPollenValue()) {
```

```
            return -1;
        } else if (this.pollenValue < o.getPollenValue()) {
            return 1;
        }

        return 0;
    }
}
package Lab4.utility;

import Lab4.graph.BeeNode;

import java.util.Comparator;

public class BeeNodeComparator implements Comparator<BeeNode> {

    @Override
    public int compare(BeeNode o1, BeeNode o2) {
        if (o1.getPollenValue() > o2.getPollenValue()) {
            return -1;
        } else if (o1.getPollenValue() < o2.getPollenValue()) {
            return 1;
        }

        return 0;
    }
}
package Lab4.utility;

import java.util.Random;

public interface Constants {
    int TOTAL_BEES = 35;
    int SCOUTS_NUMBER = 3;
    int FORAGERS_NUMBER = TOTAL_BEES - SCOUTS_NUMBER;
    int ITERATIONS_NUMBER = 1000;
    int NODES_NUMBER = 150;
    int MIN_NODE_DEGREE = 1;
    int MAX_NODE_DEGREE = 30;
    int INIT_COLOR = -1;
    Random RAND = new Random();
```

```
}
package Lab4.utility;

import Lab4.graph.BeeGraph;
import Lab4.graph.BeeNode;

import java.util.ArrayList;

public class Util {

    public static ArrayList<Integer> createStartIndexes(int size) {
        ArrayList<Integer> set = new ArrayList<>();

        for (int i = 0; i < size; i++) {
            set.add(i);
        }

        return set;
    }

    public static int[] createAllColors(int size) {
        int[] allColors = new int[size];

        for (int i = 0; i < allColors.length; i++) {
            allColors[i] = i;
        }

        return allColors;
    }

    public static void printGraph(BeeGraph graph, String message) {
        System.out.println(message + " Node Negrees");
        printGraphNodeDegrees(graph);

        System.out.println("\n" + message + " Node Colors:");
        printGraphColorsDegrees(graph);
    }

    public static void printGraphNodeDegrees(BeeGraph graph) {
```

```
            int i = 0;
            for (BeeNode node : graph.getNodes()) {
                i++;
                System.out.printf("%4d", node.getDegree());
                if (i / 30 == 1) {
                    System.out.println();
                    i = 0;
                }
            }
        }

        public static void printGraphColorsDegrees(BeeGraph graph, String
message) {
            System.out.println(message);
            printGraphColorsDegrees(graph);
        }

        public static void printGraphColorsDegrees(BeeGraph graph) {
            int i = 0;
            int[] colors = graph.getPaintedNodesColors();

            for (int j = 0; j < colors.length; j++) {
                i++;
                System.out.printf("%4d", colors[j]);
                if (i / 30 == 1) {
                    System.out.println();
                    i = 0;
                }
            }
        }
    }
    package Lab4;

    import Lab4.algorithm.ABCAlgorithm;
    import Lab4.graph.BeeGraph;

    import static Lab4.utility.Constants.ITERATIONS_NUMBER;
    import static Lab4.utility.Constants.NODES_NUMBER;
    import static Lab4.utility.Util.printGraph;
    import static Lab4.utility.Util.printGraphColorsDegrees;
```

```
public class Main {

    public static void main(String[] args) {
        BeeGraph beeGraph = new BeeGraph();
        ABCAlgorithm algorithm = new ABCAlgorithm(beeGraph);
        int best = NODES_NUMBER;
        int current;

        printGraph(algorithm.getInitGraph(), "Initial graph");

        for (int i = 1; i <= ITERATIONS_NUMBER; i++) {
            algorithm.runAlgorithm();
            current = algorithm.getCurrentBeeGraph().getChromaticNumber();
            if (best > current) {
                  System.out.println(i + " Better Chromatic Number found: " +
best);
                      printGraphColorsDegrees(algorithm.getCurrentBeeGraph(),
"Graph colors");
                  best = current;
                  algorithm.setBestBeeGraph(algorithm.getCurrentBeeGraph());
            } else {
                  System.out.println(i + " Chromatic Number: " + best);
            }

            algorithm.resetAlgorithm();
        }
    }
```

3.1.2 Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми.

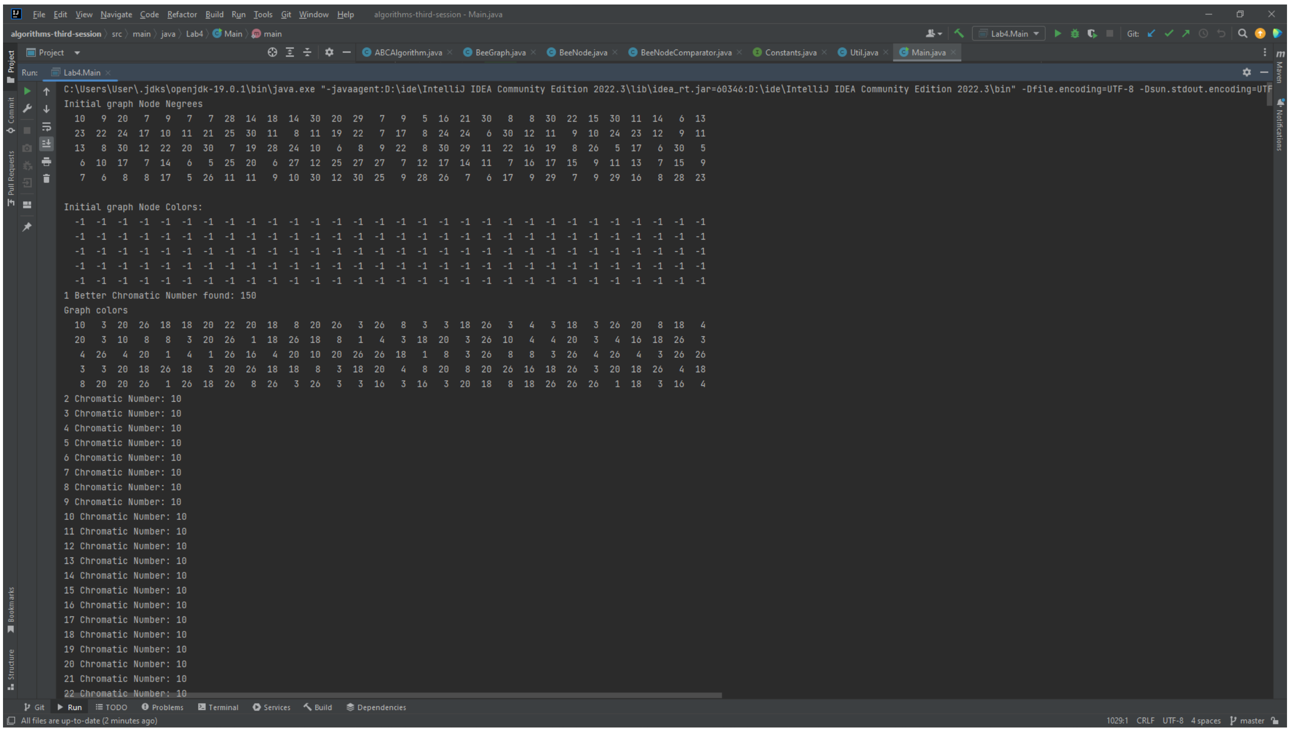

Рисунок 3.1 – Початок виводу результату роботи

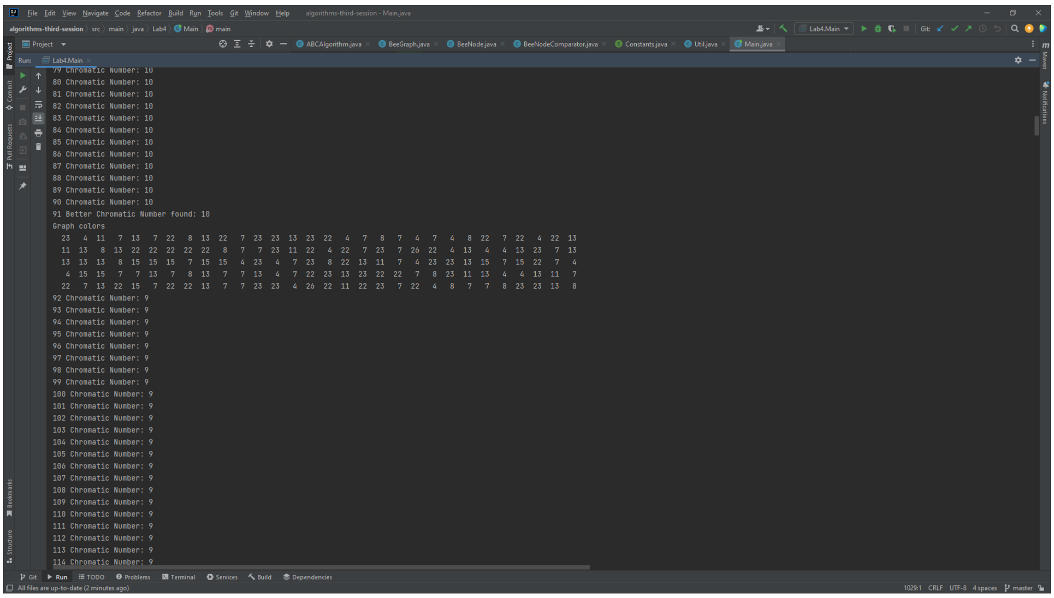

Рисунок 3.2 – вивід найкращого знайденого хроматичного числа

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.

Ітерації	**Хром. Число**	**Ітерації**	**Хром. Число**	**Ітерації**	**Хром. Число**	**Ітерації**	**Хром. Число**
0	150	260	8	520	8	780	8
20	12	280	8	540	8	800	8
40	10	300	8	560	8	820	8
60	9	320	8	580	8	840	8
80	9	340	8	600	8	860	8
100	9	360	8	620	8	880	8
120	9	380	8	640	8	900	8
140	9	400	8	660	8	920	8
160	9	420	8	680	8	940	8
180	9	440	8	700	8	960	8
200	9	460	8	720	8	980	8
220	9	480	8	740	8	1000	8
240	9	500	8	760	8	—	—

3.2.2 Графіки залежності розв'язку від числа ітерацій

На рисунку 3.3 наведений графік, який показує якість отриманого розв'язку, де вертикальна вісь - хроматичне число, горизонтальна вісь - кількість ітерацій

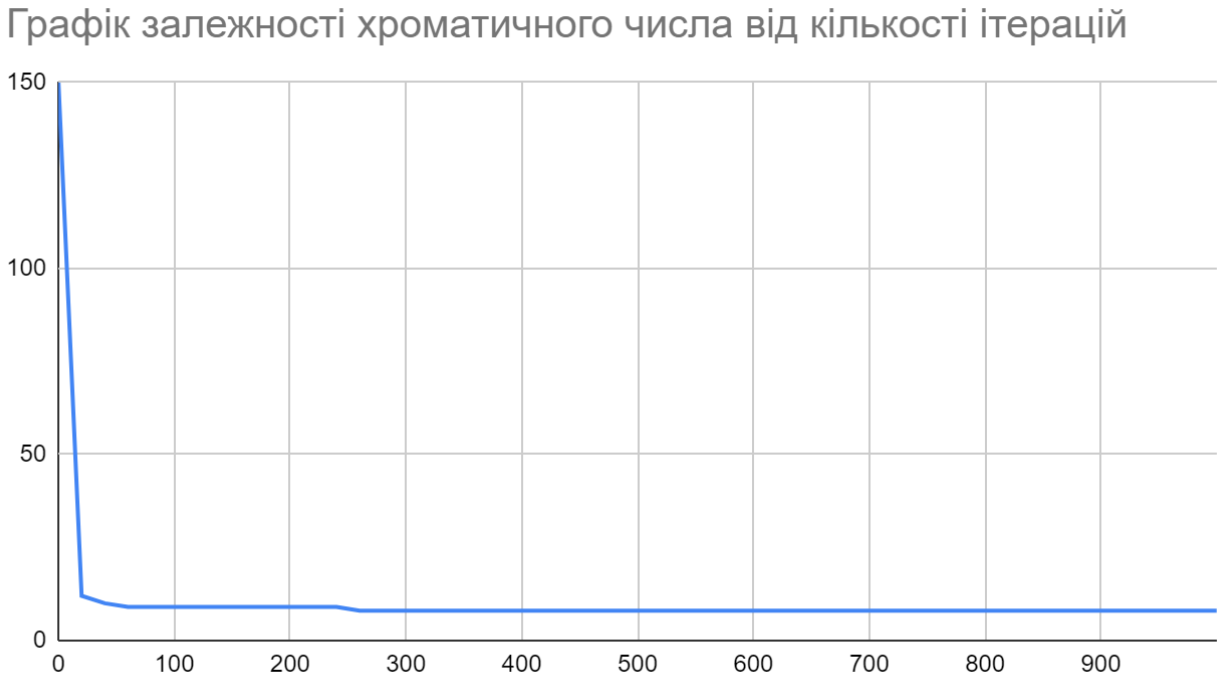

Рисунок 3.3 – Графіки залежності розв'язку від числа ітерацій

Висновок

В рамках даної лабораторної роботи я розв’язав задачу розфарбування графу на 150 вершин, де степінь кожної з них є у межах від 1 до 30 включно, виконавши програмну реалізацію на мові програмування Java, бджолиним алгоритмом, а саме його модифікацією – штучна бджолина колонія з такими параметрами: число бджіл 25 із них 3 розвідники. Також отримав графік залежності хроматичного числа від кількості ітерацій алгоритму, зафіксувавши на кожній двадцятій ітерації, яких всього 1000, значення цільової функції. З цього графіка видно, що вже після 240 ітерацій сенсу продовжувати виконувати алгоритм при таких параметрах немає, бо максимальне хроматичне число для цього графа вже було знайдено – 8. Було обдумано додаткові варіанти оптимізації, шляком проходження по вже розфарбованому графу ще декілька разів, але ця думка не була втілена в реальність, оскільки дана реалізація алгоритму і так чудово й швидко працює.

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%.