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 № 3 з дисципліни

«Проектування алгоритмів»

„ Проектування структур даних”

Виконав студент: ІП-13 Паламарчук Олександр Олександрович

Київ 2022

ЗМІСТ

ВАРІАНТ 21

МЕТА

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.

ВИКОНАННЯ

Програмна реалізація

Вихідний код

```
package org.example.algorithm.bee.factory.impl;

import org.example.algorithm.bee.BeeColony;
import org.example.algorithm.bee.factory.BeeColonyFactory;
import org.example.exception.BeeColonyFactoryImplException;

import java.io.FileReader;
import java.io.IOException;
import java.util.Properties;

import static org.example.constant.Constant.*;
```

```
public class BeeColonyFactoryImpl extends BeeColonyFactory {
    @Override
    public BeeColony getBeeColony() throws BeeColonyFactoryImplException
{
        Properties props = new Properties();
        BeeColony beeColony;
        try {
            props.load(new FileReader(APPLICATION_PROPERTIES));
            int numberOfEmployedBees =
Integer.parseInt(props.getProperty(NUMBER_OF_EMPLOYED_BEES));
            int numberOfScoutBees =
Integer.parseInt(props.getProperty(NUMBER_OF_SCOUT_BEES));
            int numberOfIterations =
Integer.parseInt(props.getProperty(NUMBER_OF_ITERATIONS));
            int precision =
Integer.parseInt(props.getProperty(PRECISION));
            beeColony = new BeeColony(numberOfEmployedBees,
numberOfScoutBees, numberOfIterations, precision);
        } catch (IOException e) {
            throw new BeeColonyFactoryImplException(e);
        }
        return beeColony;
    }
}
```

```
package org.example.algorithm.bee.factory;

import org.example.algorithm.bee.BeeColony;
import org.example.algorithm.bee.factory.impl.BeeColonyFactoryImpl;
import org.example.exception.BeeColonyFactoryException;

public abstract class BeeColonyFactory {

    public static BeeColonyFactory newInstance() {
        return new BeeColonyFactoryImpl();
    }
```

```
    public abstract BeeColony getBeeColony() throws
BeeColonyFactoryException;

}
package org.example.algorithm.bee;

import org.example.algorithm.Area;
import org.example.graph.ColorGraph;
import org.example.graph.node.Color;
import org.example.graph.node.Vertex;

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

import static java.util.Objects.nonNull;
import static java.util.Objects.requireNonNull;
import static org.example.graph.node.Color.EMPTY;

public class BeeColony {

    private List<ColorGraph> colorGraphs;
    private final PriorityQueue<Area> areasQueue;
    private final int numberOfEmployedBees;
    private final int numberOfScoutBees;
    private final int numberOfIterations;
    private final int precision;

    public BeeColony(int numberOfEmployedBees, int numberOfScoutBees,
int numberOfIterations, int precision) {
        this.numberOfEmployedBees = numberOfEmployedBees;
        this.numberOfScoutBees = numberOfScoutBees;
        this.numberOfIterations = numberOfIterations;
        this.precision = precision;
        this.areasQueue = new
PriorityQueue<>(Comparator.comparing(Area::getChromaticNumber));
```

```
}

public void setColorGraphs(List<ColorGraph> colorGraphs) {
    this.colorGraphs = requireNonNull(colorGraphs);
}

public List<Integer> search() {
    scoutAll();
    List<Integer> chromaticNumbers = new ArrayList<>();
    for (int i = 0; i < numberOfIterations; i++) {
        if (i % precision == 0) {
            Area peek = areasQueue.peek();
            if (peek != null)
                chromaticNumbers.add(peek.getChromaticNumber());
        }

        int numberOfFreeBees = numberOfEmployedBees;
        List<Area> perspectiveAreas = new ArrayList<>();
        for (int j = 0; j < numberOfScoutBees - 1; j++)
            perspectiveAreas.add(areasQueue.poll());

        if (areasQueue.size() != 0)
            perspectiveAreas.add(getRandomAreaFromAreasQueue());

        for (Area area : perspectiveAreas) {
            if (numberOfFreeBees <= 0)
                break;

            if (nonNull(area)) {
                ColorGraph graph = area.getGraph();
                if (area.getVertices().size() == 0)
                    area.setVertices(enqueueVertices(graph));
                Vertex vertex = area.getVertices().poll();
                if (nonNull(vertex)) {
                    int power = graph.getPower(vertex);
                    int neededBees = Math.min(numberOfFreeBees,
```

```
power);
                        numberOfFreeBees -= neededBees;
                        paint(graph, vertex, neededBees);

area.setChromaticNumber(calculateChromaticNumber(graph));
                    }
                }
            }

            for (Area area : perspectiveAreas)
                if (nonNull(area)) areasQueue.add(area);

        }

        Area peek = areasQueue.peek();
        if (peek != null)
            chromaticNumbers.add(peek.getChromaticNumber());

        return chromaticNumbers;
    }

    private Area getRandomAreaFromAreasQueue() {
        ArrayList<Area> areas = new ArrayList<>(areasQueue);
        int randomI = ThreadLocalRandom.current().nextInt(areas.size());
        Area area = areas.get(randomI);
        areasQueue.remove(area);
        return area;
    }

    private void paint(ColorGraph graph, Vertex vertex, int iteration) {
        List<Vertex> adjacentVertices = graph.adjacentNodes(vertex)
                .stream().toList();

        for (int i = 0; i < iteration; i++) {
            Vertex adjacentVertex = adjacentVertices.get(i);
            boolean isSwapped = swapColors(graph, vertex,
```

```
adjacentVertex);
            if (isSwapped) tryToChangeColor(graph, adjacentVertex);
        }
    }

    private void tryToChangeColor(ColorGraph graph, Vertex vertex) {
        Set<Color> usedColors = graph.getUsedColors();
        Set<Color> adjacentColors = graph.getAdjacentColors(vertex);

        for (Color c : usedColors) {
            if (!adjacentColors.contains(c) &&
!c.equals(vertex.getColour())) {
                vertex.setColour(c);
                break;
            }
        }
    }

    private boolean swapColors(ColorGraph graph, Vertex v1, Vertex v2) {
        boolean canBeSwapped;

        Color v1Colour = v1.getColour();
        Color v2Colour = v2.getColour();

        Set<Color> v1AdjacentColors = graph.getAdjacentColorsExcept(v1,
v2);
        canBeSwapped = !v1AdjacentColors.contains(v2Colour);
        if (!canBeSwapped) return false;

        Set<Color> v2AdjacentColors = graph.getAdjacentColorsExcept(v2,
v1);
        canBeSwapped = !v2AdjacentColors.contains(v1Colour);
        if (!canBeSwapped) return false;

        v1.setColour(v2Colour);
        v2.setColour(v1Colour);
```

```
        return true;
    }

    private void scoutAll() {
        colorGraphs.forEach(colorGraph -> areasQueue.add(
                new Area(colorGraph,
calculateChromaticNumber(colorGraph), enqueueVertices(colorGraph))));
    }

    private PriorityQueue<Vertex> enqueueVertices(ColorGraph colorGraph)
{
        PriorityQueue<Vertex> vertices = new
PriorityQueue<>(Comparator.comparing(colorGraph::getPower).reversed());
        vertices.addAll(colorGraph.nodes());
        return vertices;
    }

    private static int calculateChromaticNumber(ColorGraph colorGraph) {
        return colorGraph.nodes().stream()
                .map(Vertex::getColour)
                .filter(colour -> !colour.equals(EMPTY))
                .collect(Collectors.toSet())
                .size();
    }

    public PriorityQueue<Area> getAreasQueue() {
        return areasQueue;
    }
}
package org.example.algorithm.dfs;

import org.example.graph.ColorGraph;
import org.example.graph.node.Color;
import org.example.graph.node.Vertex;

import java.util.*;
```

```java
import static org.example.algorithm.dfs.Indicator.FAILURE;
import static org.example.algorithm.dfs.Indicator.SOLUTION;
import static org.example.graph.node.Color.*;

public class DepthFirstSearch {

    private ColorGraph graph;
    private final List<Color> palette;

    public DepthFirstSearch(ColorGraph graph) {
        this.graph = graph;
        this.palette = new ArrayList<>(EnumSet.range(RED, KHAKI));
    }

    public ColorGraph search() {
        Result result = recursive(graph);

        if (result.hasSolution())
            return result.getSolution();

        throw new IllegalStateException("Cannot find solution");
    }

    private Result recursive(ColorGraph curr) {
        if (curr.isSolution())
            return Result.of(curr, SOLUTION);

        List<ColorGraph> successors = getSuccessors(curr);

        if (successors.isEmpty())
            return Result.of(FAILURE);

        for (ColorGraph successor : successors) {
            Result result = recursive(successor);
```

```
            if (result.hasSolution())
                return result;
        }

        return Result.of(FAILURE);
    }

    private List<ColorGraph> getSuccessors(ColorGraph curr) {
        Vertex vertex = getVertexWithoutColour(curr);

        Set<Color> adjacentColors = curr.getAdjacentColors(vertex);
        List<ColorGraph> successors = new ArrayList<>();

        for (Color c : palette) {
            if (!adjacentColors.contains(c)) {
                ColorGraph copy = curr.copy();
                paint(vertex.getId(), c, copy);
                successors.add(copy);
            }
        }
        Collections.shuffle(successors);
        return successors;
    }

    private void paint(int id, Color color, ColorGraph state) {
        Vertex vertex = state.nodes().stream()
                .filter(node -> node.getId() == id)
                .findFirst().orElseThrow();
        vertex.setColour(color);
    }

    private Vertex getVertexWithoutColour(ColorGraph state) {
        return state.nodes().stream()
                .filter(n -> n.getColour().equals(EMPTY))
                .findAny().orElseThrow(IllegalStateException::new);
```

```
    }

    public void setGraph(ColorGraph graph) {
        this.graph = graph;
    }
}
```

```
package org.example.algorithm.dfs;

public enum Indicator {

    SOLUTION,
    FAILURE

}
```

```
package org.example.algorithm.dfs;

import org.example.graph.ColorGraph;

import static org.example.algorithm.dfs.Indicator.FAILURE;
import static org.example.algorithm.dfs.Indicator.SOLUTION;

public class Result {

    private final ColorGraph solution;
    private final Indicator indicator;

    private Result(ColorGraph solution, Indicator indicator) {
        this.solution = solution;
        this.indicator = indicator;
    }

    public static Result of(Indicator indicator) {
        return new Result(null, indicator);
    }
```

```
    public static Result of(ColorGraph solution, Indicator indicator) {
        return new Result(solution, indicator);
    }

    public ColorGraph getSolution() {
        return solution;
    }

    public boolean hasSolution() {
        return indicator.equals(SOLUTION);
    }

    public boolean failure() {
        return indicator.equals(FAILURE);
    }

}
package org.example.algorithm;

import org.example.graph.ColorGraph;
import org.example.graph.node.Vertex;

import java.util.Objects;
import java.util.PriorityQueue;

public class Area {
    private int chromaticNumber;
    private PriorityQueue<Vertex> vertices;
    private final ColorGraph graph;

    public Area(ColorGraph graph, int chromaticNumber,
PriorityQueue<Vertex> vertices) {
        if (chromaticNumber < 0)
            throw new IllegalArgumentException("Chromatic number cannot
be < 0, but actually " + chromaticNumber);
```

```
        this.vertices = Objects.requireNonNull(vertices);
        this.graph = Objects.requireNonNull(graph);
        this.chromaticNumber = chromaticNumber;
    }

    public ColorGraph getGraph() {
        return graph;
    }

    public int getChromaticNumber() {
        return chromaticNumber;
    }

    public void setChromaticNumber(int chromaticNumber) {
        this.chromaticNumber = chromaticNumber;
    }

    public void setVertices(PriorityQueue<Vertex> vertices) {
        this.vertices = vertices;
    }

    public PriorityQueue<Vertex> getVertices() {
        return vertices;
    }
}
package org.example.constant;

public class Constant {

    public static final String APPLICATION_PROPERTIES =
"src/main/resources/application.properties";
    public static final String XSD_FILE_NAME = "xsd.file.name";
    public static final String XML_FILE_NAME = "xml.file.name";
    public static final String OUTPUT_FILE_NAME = "output.file.name";
    public static final String NUMBER_OF_EMPLOYED_BEES =
```

```
"number.of.employed.bees";
    public static final String NUMBER_OF_SCOUT_BEES =
"number.of.scout.bees";
    public static final String NUMBER_OF_AREAS = "number.of.areas";
    public static final String NUMBER_OF_ITERATIONS =
"number.of.iterations";
    public static final String PRECISION = "precision";

}
package org.example.exception;

public class BeeColonyFactoryException extends Exception {
    public BeeColonyFactoryException() {
    }

    public BeeColonyFactoryException(String message) {
        super(message);
    }

    public BeeColonyFactoryException(String message, Throwable cause) {
        super(message, cause);
    }

    public BeeColonyFactoryException(Throwable cause) {
        super(cause);
    }
}
package org.example.exception;

public class BeeColonyFactoryImplException extends
BeeColonyFactoryException {
    public BeeColonyFactoryImplException() {
    }

    public BeeColonyFactoryImplException(String message) {
        super(message);
```

```
    }

    public BeeColonyFactoryImplException(String message, Throwable
cause) {
        super(message, cause);
    }

    public BeeColonyFactoryImplException(Throwable cause) {
        super(cause);
    }
}
package org.example.exception;

public class GraphExporterFactoryException extends Exception {
    public GraphExporterFactoryException() {
    }

    public GraphExporterFactoryException(String message) {
        super(message);
    }

    public GraphExporterFactoryException(String message, Throwable
cause) {
        super(message, cause);
    }

    public GraphExporterFactoryException(Throwable cause) {
        super(cause);
    }
}
package org.example.exception;

public class GraphExporterFactoryImplException extends
GraphExporterFactoryException {
    public GraphExporterFactoryImplException() {
    }
```

```
    public GraphExporterFactoryImplException(String message) {
        super(message);
    }

    public GraphExporterFactoryImplException(String message, Throwable
cause) {
        super(message, cause);
    }

    public GraphExporterFactoryImplException(Throwable cause) {
        super(cause);
    }
}
package org.example.exception;

public class GraphFactoryException extends Exception {
    public GraphFactoryException() {
    }

    public GraphFactoryException(String message) {
        super(message);
    }

    public GraphFactoryException(String message, Throwable cause) {
        super(message, cause);
    }

    public GraphFactoryException(Throwable cause) {
        super(cause);
    }
}
package org.example.exception;

public class StAXControllerException extends Exception {
    public StAXControllerException() {
```

```
    }

    public StAXControllerException(String message) {
        super(message);
    }

    public StAXControllerException(String message, Throwable cause) {
        super(message, cause);
    }

    public StAXControllerException(Throwable cause) {
        super(cause);
    }
}
package org.example.graph.controller;

import org.example.exception.StAXControllerException;
import org.xml.sax.SAXException;

import javax.xml.stream.XMLEventReader;
import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLStreamException;
import javax.xml.stream.events.Attribute;
import javax.xml.stream.events.EndElement;
import javax.xml.stream.events.StartElement;
import javax.xml.stream.events.XMLEvent;
import javax.xml.transform.stream.StreamSource;
import javax.xml.validation.Schema;
import javax.xml.validation.SchemaFactory;
import javax.xml.validation.Validator;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Iterator;
```

```
import java.util.List;
import java.util.Objects;

import static javax.xml.XMLConstants.W3C_XML_SCHEMA_NS_URI;
import static org.example.graph.controller.XMLTag.*;

public class StAXController {

    private List<List<Integer>> adjacencyList;
    private final String xsdFileName;
    private final String xmlFileName;
    private final XMLInputFactory factory;
    public StAXController(String xsdFileName, String xmlFileName) {
        this.xsdFileName = Objects.requireNonNull(xsdFileName);
        this.xmlFileName = Objects.requireNonNull(xmlFileName);
        this.factory = XMLInputFactory.newInstance();
    }

    public void buildAdjacencyList() throws StAXControllerException {
        validate();

        int currId = 0;
        List<Integer> adjacentVertices = null;
        List<List<Integer>> adjacencyList = null;
        try {
            XMLEventReader reader =
                    factory.createXMLEventReader(new
FileReader(xmlFileName));
            while (reader.hasNext()) {
                XMLEvent event = reader.nextEvent();
                if (event.isStartElement()) {
                    StartElement startElement = event.asStartElement();
                    String tagName =
startElement.getName().getLocalPart();
                    XMLTag tag = XMLTag.from(tagName);
                    if (tag.equals(GRAPH)) {
```

```
                    adjacencyList = new ArrayList<>();
                } else if (tag.equals(VERTEX)) {
                    adjacentVertices = new ArrayList<>();
                    Iterator<Attribute> attributes =
                            startElement.getAttributes();
                    while (attributes.hasNext()) {
                        Attribute attribute = attributes.next();
                        String localPart =
attribute.getName().getLocalPart();
                        if (localPart.equals("id")) {
                            currId =
Integer.parseInt(attribute.getValue());
                        }
                    }
                } else if (tag.equals(ADJACENT_VERTEX)) {
                    Iterator<Attribute> attributes
                            = startElement.getAttributes();
                    assert adjacentVertices != null;
                    while (attributes.hasNext()) {
                        Attribute attribute = attributes.next();
                        String localPart =
attribute.getName().getLocalPart();
                        if (localPart.equals("id")) {

adjacentVertices.add(Integer.parseInt(attribute.getValue()));
                        }
                    }
                }
            } else if (event.isEndElement()) {
                EndElement endElement = event.asEndElement();
                String tagName =
endElement.getName().getLocalPart();
                XMLTag tag = from(tagName);
                if (tag.equals(GRAPH)) {
                    this.adjacencyList = adjacencyList;
                } else if (tag.equals(VERTEX)) {
                    assert adjacencyList != null;
```

```
                        adjacencyList.add(currId, adjacentVertices);
                    }
                }
            }
        } catch (XMLStreamException | FileNotFoundException e) {
            throw new StAXControllerException(e);
        }
    }

    private void validate() throws StAXControllerException {
        SchemaFactory schemaFactory =
SchemaFactory.newInstance(W3C_XML_SCHEMA_NS_URI);
        try {
            Schema schema = schemaFactory.newSchema(new
File(xsdFileName));
            Validator validator = schema.newValidator();
            validator.validate(new StreamSource(xmlFileName));
        } catch (SAXException | IOException e) {
            throw new StAXControllerException(e);
        }
    }


    public List<List<Integer>> getAdjacencyList() {
        return adjacencyList;
    }
}
package org.example.graph.controller;

public enum XMLTag {

    GRAPH("graph"),
    VERTEX("vertex"),
    ADJACENT_VERTEX("adjacentVertex");

    private final String name;
```

```
    XMLTag(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public static XMLTag from(String name) {
        return switch (name) {
            case "graph" -> GRAPH;
            case "vertex" -> VERTEX;
            case "adjacentVertex" -> ADJACENT_VERTEX;
            default -> throw new IllegalStateException("Unknown tag
name");
        };
    }
}
package org.example.graph.exporter.factory.impl;

import org.example.exception.GraphExporterFactoryImplException;
import org.example.graph.exporter.GraphExporter;
import org.example.graph.exporter.factory.GraphExporterFactory;

import java.io.FileReader;
import java.io.IOException;
import java.util.Properties;

import static org.example.constant.Constant.APPLICATION_PROPERTIES;
import static org.example.constant.Constant.OUTPUT_FILE_NAME;

public class GraphExporterFactoryImpl extends GraphExporterFactory {

    private String fileName;
```

```
    public GraphExporterFactoryImpl() throws
GraphExporterFactoryImplException {
        Properties props = new Properties();
        try {
            props.load(new FileReader(APPLICATION_PROPERTIES));
            this.fileName = props.getProperty(OUTPUT_FILE_NAME);
        } catch (IOException e) {
            throw new GraphExporterFactoryImplException(e);
        }
    }

    @Override
    public GraphExporter newGraphExporter() {
        return new GraphExporter(fileName);
    }
}
```

```
package org.example.graph.exporter.factory;

import org.example.exception.GraphExporterFactoryException;
import org.example.graph.exporter.GraphExporter;
import org.example.graph.exporter.factory.impl.GraphExporterFactoryImpl;

public abstract class GraphExporterFactory {

    public static GraphExporterFactory newInstance() throws
GraphExporterFactoryException {
        return new GraphExporterFactoryImpl();
    }

    public abstract GraphExporter newGraphExporter();

}
```

```
package org.example.graph.exporter;

import com.google.common.graph.EndpointPair;
```

```java
import org.example.graph.ColorGraph;
import org.example.graph.node.Vertex;

import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.LinkedList;
import java.util.List;
import java.util.Objects;

public class GraphExporter {

    private ColorGraph colorGraph;
    private List<String> lines;
    private final String fileName;

    public GraphExporter(String fileName) {
        this.fileName = Objects.requireNonNull(fileName);
        this.lines = new LinkedList<>();
    }

    public void setColourGraph(ColorGraph colorGraph) {
        this.colorGraph = Objects.requireNonNull(colorGraph);
    }

    public void export() {
        build();
        write();
    }

    private void build() {
        List<String> lines = new LinkedList<>();
        lines.add("graph G {");
        for (Vertex v : colorGraph.nodes())
            lines.add("  %d [style=filled,
color=%s]".formatted(v.getId(), v.getColour().name().toLowerCase()));
```

```
            lines.add("");

            for (EndpointPair<Vertex> endpointPair : colorGraph.edges())
                lines.add("  %d --
%d".formatted(endpointPair.nodeU().getId(),
endpointPair.nodeV().getId()));
            lines.add("}");
            this.lines = lines;
        }

        private void write() {
            try {
                Files.write(Path.of(fileName), lines);
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
        }
}
package org.example.graph.factory.impl;

import com.google.common.graph.GraphBuilder;
import com.google.common.graph.MutableGraph;
import org.example.exception.GraphFactoryException;
import org.example.exception.StAXControllerException;
import org.example.graph.ColorGraph;
import org.example.graph.controller.StAXController;
import org.example.graph.factory.GraphFactory;
import org.example.graph.node.Vertex;

import java.io.FileReader;
import java.io.IOException;
import java.util.List;
import java.util.Properties;

import static org.example.constant.Constant.*;
```

```
public final class GraphFactoryImpl extends GraphFactory {

    private final String DEFAULT_XML_FILE_NAME;
    private final String DEFAULT_XSD_FILE_NAME;

    public GraphFactoryImpl() throws GraphFactoryException {
        Properties props = new Properties();
        try {
            props.load(new FileReader(APPLICATION_PROPERTIES));
            DEFAULT_XML_FILE_NAME = props.getProperty(XML_FILE_NAME);
            DEFAULT_XSD_FILE_NAME = props.getProperty(XSD_FILE_NAME);
        } catch (IOException e) {
            throw new GraphFactoryException(e);
        }
    }

    public ColorGraph newColourGraph() throws GraphFactoryException {
        StAXController stAXController = new
StAXController(DEFAULT_XSD_FILE_NAME, DEFAULT_XML_FILE_NAME);

        try {
            stAXController.buildAdjacencyList();
        } catch (StAXControllerException e) {
            throw new GraphFactoryException(e);
        }

        return buildGraph(stAXController.getAdjacencyList());
    }

    private static ColorGraph buildGraph(List<List<Integer>>
adjacencyList) {
        MutableGraph<Vertex> graph = GraphBuilder.undirected()
                .allowsSelfLoops(false).build();

        for (int i = 0; i < adjacencyList.size(); i++) {
```

```
            List<Integer> adjacentVertices = adjacencyList.get(i);
            for (Integer adjacentVertex : adjacentVertices) {
                graph.putEdge(new Vertex(i), new
Vertex(adjacentVertex));
            }
        }
        return new ColorGraph(graph);
    }
}
package org.example.graph.factory;

import org.example.graph.ColorGraph;
import org.example.exception.GraphFactoryException;
import org.example.graph.factory.impl.GraphFactoryImpl;

public abstract class GraphFactory {

    public static GraphFactory newInstance() throws
GraphFactoryException {
        return new GraphFactoryImpl();
    }

    public abstract ColorGraph newColourGraph() throws
GraphFactoryException;
}
package org.example.graph.node;

public enum Color {
    EMPTY(0),
    RED(1),
    BLUE(2),
    YELLOW(3),
    GREEN(4),
    CHARTREUSE(5),
    ORANGE(6),
    PURPLE(7),
```

```
    GREY(8),
    BROWN(9),
    PINK(10),
    CADETBLUE(11),
    DARKSLATEGRAY(12),
    DARKSLATEBLUE(13),
    GOLD4(14),
    AQUAMARINE3(15),
    ANTIQUEWHITE4(16),
    BROWN3(17),
    MAROON(18),
    MEDIUMPURPLE(19),
    SKYBLUE(20),
    SPRINGGREEN(21),
    YELLOWGREEN(22),
    TOMATO(23),
    NAVY(24),
    MISTYROSE(25),
    OLIVE(26),
    DARKORANGE(27),
    CHOCOLATE(28),
    DARKRED(29),
    KHAKI(30);

    private final int number;

    Color(int number) {
        this.number = number;
    }

    public int getNumber() {
        return number;
    }
}
package org.example.graph.node;
```

```java
public class Vertex implements Comparable<Vertex> {
    private int id;
    private Color color;

    private Vertex(int id, Color color) {
        this.id = id;
        this.color = color;
    }

    public Vertex(int id) {
        this(id, Color.EMPTY);
    }

    public void setId(int id) {
        if (id < 0)
            throw new IllegalArgumentException("Id must be positive, but
actually " + id);
        this.id = id;
    }

    public int getId() {
        return id;
    }

    public Vertex copy() {
        return new Vertex(id, color);
    }

    public void setColour(Color color) {
        this.color = color;
    }

    public Color getColour() {
        return color;
    }
```

```java
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Vertex vertex)) return false;

        return id == vertex.id;
    }

    @Override
    public int hashCode() {
        int result = id;
        result = 31 * result;
        return result;
    }

    @Override
    public String toString() {
        return String.valueOf(id);
    }

    @Override
    public int compareTo(Vertex o) {
        return Integer.compare(this.id, o.id);
    }
}
package org.example.graph;

import com.google.common.graph.ElementOrder;
import com.google.common.graph.EndpointPair;
import com.google.common.graph.GraphBuilder;
import com.google.common.graph.MutableGraph;
import com.google.errorprone.annotations.CanIgnoreReturnValue;
import org.example.graph.node.Color;
import org.example.graph.node.Vertex;

import javax.annotation.CheckForNull;
```

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

import static java.util.Objects.isNull;

public class ColorGraph implements MutableGraph<Vertex> {
    private final MutableGraph<Vertex> sourceGraph;

    public ColorGraph(MutableGraph<Vertex> sourceGraph) {
        this.sourceGraph = sourceGraph;
    }

    public boolean isSolution() {
        return sourceGraph.nodes().stream()
                .noneMatch(node ->
node.getColour().equals(Color.EMPTY));
    }

    public ColorGraph copy() {
        MutableGraph<Vertex> sourceCopy = GraphBuilder.undirected()
                .allowsSelfLoops(false).build();

        Set<EndpointPair<Vertex>> sourceEdges = sourceGraph.edges();
        Map<Integer, Vertex> copyVertices = new TreeMap<>();

        sourceEdges.forEach(edge -> {
            Vertex vertexU = edge.nodeU();
            Vertex vertexV = edge.nodeV();

            Vertex u = copyVertices.get(vertexU.getId());
            Vertex v = copyVertices.get(vertexV.getId());

            if (isNull(u)) {
                vertexU = vertexU.copy();
                copyVertices.put(vertexU.getId(), vertexU);
```

```
            } else vertexU = u;

            if (isNull(v)) {
                vertexV = vertexV.copy();
                copyVertices.put(vertexV.getId(), vertexV);
            } else vertexV = v;

            sourceCopy.putEdge(vertexU, vertexV);
        });


        return new ColorGraph(sourceCopy);
    }

    public Set<Color> getAdjacentColors(Vertex vertex) {
        checkVertexExists(vertex);
        return adjacentNodes(vertex).stream()
                .map(Vertex::getColour).collect(Collectors.toSet());
    }

    public Vertex getVertexWithMaxPower() {
        return nodes().stream()
                .max(Comparator.comparing(vertex ->
adjacentNodes(vertex).size()))
                .orElseThrow(() -> new IllegalStateException("Cannot
find vertex with max power"));
    }

    public int getPower(Vertex vertex) {
        checkVertexExists(vertex);
        return adjacentNodes(vertex).size();
    }

    private void checkVertexExists(Vertex vertex) {
        if (!nodes().contains(vertex))
            throw new IllegalArgumentException("Vertex does not exist");
```

```
}

public Set<Color> getUsedColors() {
    return sourceGraph.nodes().stream()
            .map(Vertex::getColour)
            .collect(Collectors.toSet());
}

@Override
@CanIgnoreReturnValue
public boolean addNode(Vertex vertex) {
    return sourceGraph.addNode(vertex);
}

@Override
@CanIgnoreReturnValue
public boolean putEdge(Vertex vertex, Vertex n1) {
    return sourceGraph.putEdge(vertex, n1);
}

@Override
@CanIgnoreReturnValue
public boolean putEdge(EndpointPair<Vertex> endpointPair) {
    return sourceGraph.putEdge(endpointPair);
}

@Override
@CanIgnoreReturnValue
public boolean removeNode(Vertex vertex) {
    return sourceGraph.removeNode(vertex);
}

@Override
@CanIgnoreReturnValue
public boolean removeEdge(Vertex vertex, Vertex n1) {
    return sourceGraph.removeEdge(vertex, n1);
```

```
}

@Override
@CanIgnoreReturnValue
public boolean removeEdge(EndpointPair<Vertex> endpointPair) {
    return sourceGraph.removeEdge(endpointPair);
}

@Override
public Set<Vertex> nodes() {
    return sourceGraph.nodes();
}

@Override
public Set<EndpointPair<Vertex>> edges() {
    return sourceGraph.edges();
}

@Override
public boolean isDirected() {
    return sourceGraph.isDirected();
}

@Override
public boolean allowsSelfLoops() {
    return sourceGraph.allowsSelfLoops();
}

@Override
public ElementOrder<Vertex> nodeOrder() {
    return sourceGraph.nodeOrder();
}

@Override
public ElementOrder<Vertex> incidentEdgeOrder() {
    return sourceGraph.incidentEdgeOrder();
```

```
}

@Override
public Set<Vertex> adjacentNodes(Vertex vertex) {
    return sourceGraph.adjacentNodes(vertex);
}

@Override
public Set<Vertex> predecessors(Vertex vertex) {
    return sourceGraph.predecessors(vertex);
}

@Override
public Set<Vertex> successors(Vertex vertex) {
    return sourceGraph.successors(vertex);
}

@Override
public Set<EndpointPair<Vertex>> incidentEdges(Vertex vertex) {
    return sourceGraph.incidentEdges(vertex);
}

@Override
public int degree(Vertex vertex) {
    return sourceGraph.degree(vertex);
}

@Override
public int inDegree(Vertex vertex) {
    return sourceGraph.inDegree(vertex);
}

@Override
public int outDegree(Vertex vertex) {
    return sourceGraph.outDegree(vertex);
}
```

```
    @Override
    public boolean hasEdgeConnecting(Vertex vertex, Vertex n1) {
        return sourceGraph.hasEdgeConnecting(vertex, n1);
    }

    @Override
    public boolean hasEdgeConnecting(EndpointPair<Vertex> endpointPair)
{
        return sourceGraph.hasEdgeConnecting(endpointPair);
    }

    @Override
    public boolean equals(@CheckForNull Object o) {
        return sourceGraph.equals(o);
    }

    @Override
    public int hashCode() {
        return sourceGraph.hashCode();
    }

    @Override
    public String toString() {
        StringBuffer sb = new StringBuffer();
        nodes().stream()
                .sorted(Comparator.comparing(Vertex::getId))
                .forEach(vertex -> {
                    List<Vertex> vertices =
adjacentNodes(vertex).stream()
                            .sorted(Comparator.comparing(Vertex::getId))
                            .toList();
                    sb.append(vertex).append(" ==>
").append(vertices).append('\n');
                });
        return sb.toString();
```

```
    }

    public Set<Color> getAdjacentColorsExcept(Vertex vertex, Vertex
except) {
        return adjacentNodes(vertex).stream()
                .filter(v -> v.getId() != except.getId())
                .map(Vertex::getColour)
                .collect(Collectors.toSet());
    }
}
package org.example;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.chart.LineChart;
import javafx.scene.chart.NumberAxis;
import javafx.scene.chart.XYChart;
import javafx.stage.Stage;
import org.example.algorithm.Area;
import org.example.algorithm.bee.BeeColony;
import org.example.algorithm.bee.factory.BeeColonyFactory;
import org.example.algorithm.dfs.DepthFirstSearch;
import org.example.graph.ColorGraph;
import org.example.graph.exporter.GraphExporter;
import org.example.graph.exporter.factory.GraphExporterFactory;
import org.example.graph.factory.GraphFactory;

import java.io.FileReader;
import java.util.ArrayList;
import java.util.List;
import java.util.Properties;

import static org.example.constant.Constant.*;

public class Main extends Application {
```

```
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage stage) throws Exception {

        Properties props = new Properties();
        props.load(new FileReader(APPLICATION_PROPERTIES));
        int numberOfAreas =
Integer.parseInt(props.getProperty(NUMBER_OF_AREAS));
        int precision = Integer.parseInt(props.getProperty(PRECISION));

        GraphFactory graphFactory = GraphFactory.newInstance();
        ColorGraph colorGraph = graphFactory.newColourGraph();

        List<ColorGraph> colorGraphs = new ArrayList<>();
        for (int i = 0; i < numberOfAreas; i++) {
            DepthFirstSearch dfs = new DepthFirstSearch(colorGraph);
            ColorGraph graph = dfs.search();
            colorGraphs.add(graph);
        }

        BeeColonyFactory colonyFactory = BeeColonyFactory.newInstance();
        BeeColony beeColony = colonyFactory.getBeeColony();
        beeColony.setColorGraphs(colorGraphs);
        List<Integer> result = beeColony.search();

        Area peek = beeColony.getAreasQueue().peek();
        if (peek != null) {
            ColorGraph graph = peek.getGraph();
            GraphExporterFactory factory =
GraphExporterFactory.newInstance();
            GraphExporter graphExporter = factory.newGraphExporter();
            graphExporter.setColourGraph(graph);
```

```
            graphExporter.export();
        }

        stage.setTitle("Graphic");

        NumberAxis xAxis = new NumberAxis();
        NumberAxis yAxis = new NumberAxis();
        xAxis.setLabel("Iterations");
        yAxis.setLabel("Chromatic numbers");

        LineChart<Number, Number> lineChart
                = new LineChart<>(xAxis, yAxis);
        lineChart.setTitle("Quality of result");

        XYChart.Series<Number, Number> series
                = new XYChart.Series<>();

        for (int i = 0; i < result.size(); i++)
            series.getData().add(new XYChart.Data<>((i * precision),
result.get(i)));

        Scene scene = new Scene(lineChart, 800, 600);
        lineChart.getData().add(series);

        stage.setScene(scene);
        stage.show();
    }
}
```

ПРИКЛАДИ РОБОТИ

На рисунках 3.1 і 3.2 показані приклади роботи програми.

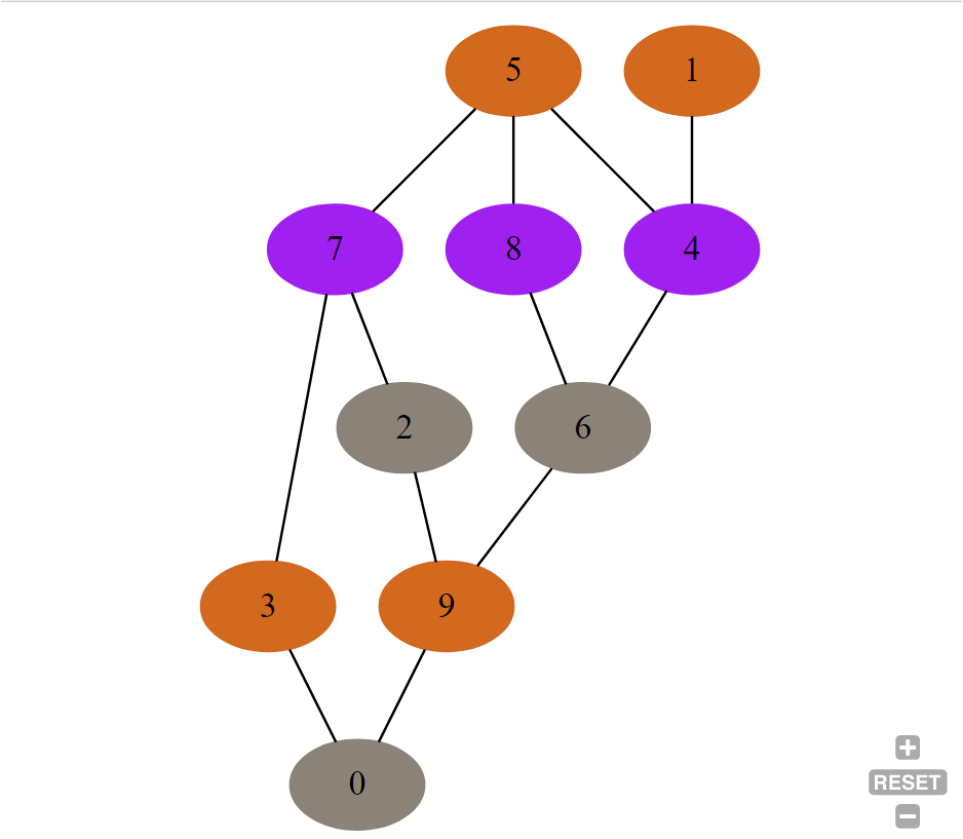

Рисунок 3.1 – Граф на 10 вершин

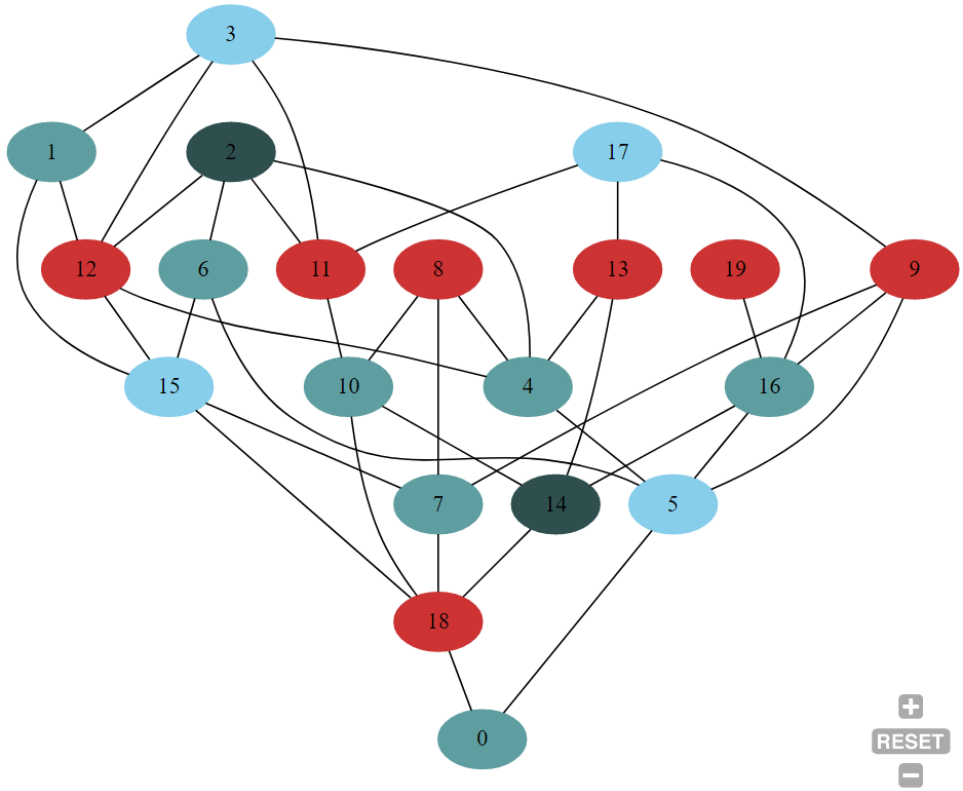

Рисунок 3.2 – Граф на 20 вершин

ТСТУВАННЯ АЛГОРИТМУ

Значення цільової функції зі збільшенням кількості ітерацій

У таблиці 3.1 наведено значення цільової функції зі збільшенням кількості ітерацій.

Таблиця 3.1

Ітерація	Значення ЦФ
0	30
20	29
40	25
60	23

80	21
100	19
120	16
140	16
160	16
180	14
200	14
220	14
240	14
260	14
280	14
300	14
320	14
340	13
360	13
380	13
400	13
420	13
440	12
460	12
480	12
500	12
520	12
540	12
560	12
580	12
600	12
620	12
640	12
660	12
680	12
700	11
720	11
740	11
760	11
780	11
800	11
820	11
840	11

860	11
880	11
900	11
920	11
940	11
960	11
980	11
1000	11

Графіки залежності розв'язку від числа ітерацій

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

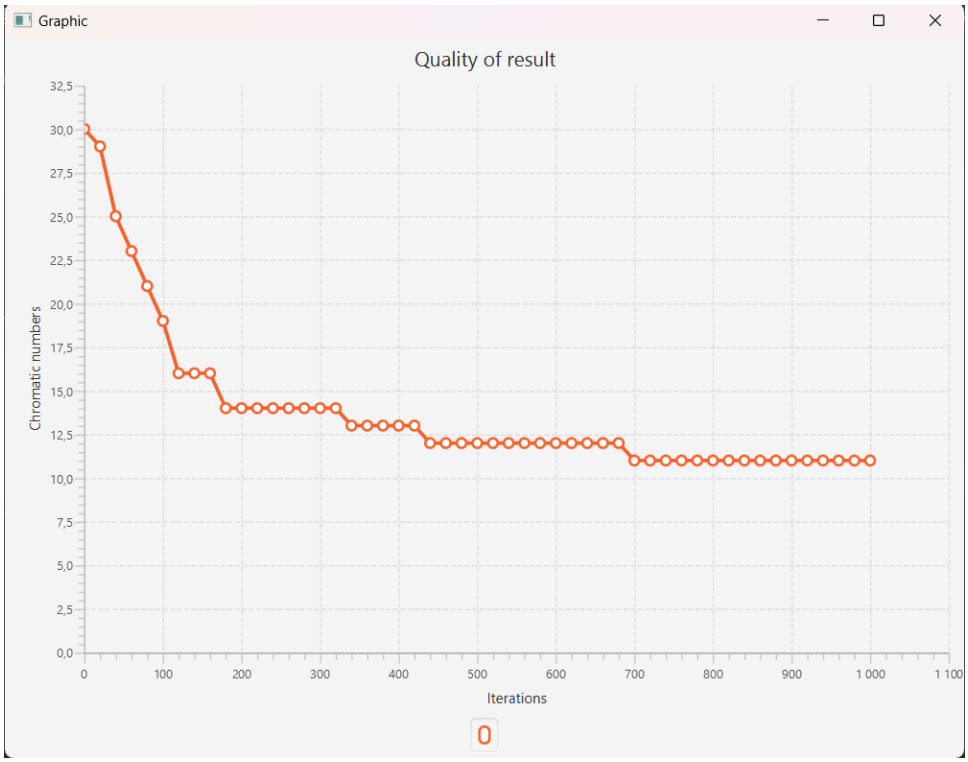

Рисунок 3.3 – Графіки залежності розв'язку від числа ітерацій

ВИСНОВОК

На лабораторній роботі було вивчено основні підходи формалізації метаевристичних алгоритмів і вирішення типових задач з їхньою допомогою. Було досліджено та спроектовано алгоритм “Класичний бджолиний

алгоритм”. Було наведено приклади роботи алгоритму, а також проведено тестування.