Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

Лабораторна робота №5

з дисципліни
«Алгоритми і структури даних»

Виконав:

студент групи ІМ-43
Костеніч Степан Станіславович
номер у списку групи: 17

Перевірив:

Сергієнко А. М.

Київ 2025

Постановка задачі

1. Представити напрямлений граф із заданими параметрами так само, як у лабораторній роботі №3.

   Відмінність: коефіцієнт $k = 1.0 - n_3 * 0.01 - n_4 * 0.005 - 0.15$.

   Отже, матриця суміжності A_{dir} напрямленого графа за варіантом формується таким чином:

   1) встановлюється параметр (seed) генератора випадкових чисел, рівне номеру варіанту $n_1n_2n_3n_4$;
   2) матриця розміром n * n заповнюється згенерованими випадковими числами в діапазоні [0, 2.0);
   3) обчислюється коефіцієнт $k = 1.0 - n_3 * 0.01 - n_4 * 0.005 - 0.15$, кожен елемент матриці множиться на коефіцієнт k;
   4) елементи матриці округлюються: 0 — якщо елемент менший за 1.0, 1 — якщо елемент більший або дорівнює 1.0.

2. Створити програму, яка виконує обхід напрямленого графа вшир (BFS) та вглиб (DFS):

   - обхід починати з вершини із найменшим номером, яка має щонайменше одну вихідну дугу;
   - при обході враховувати порядок нумерації;
   - у програмі виконання обходу відображати покроково, черговий крок виконувати за натисканням кнопки у вікні або на клавіатурі.

3. Під час обходу графа побудувати дерево обходу. У програмі дерево обходу виводити покроково у процесі виконання обходу графа. Це можна виконати одним із двох способів:

   - або виділяти іншим кольором ребра графа;
   - або будувати дерево обходу поряд із графом.

4. Зміну статусів вершин у процесі обходу продемонструвати зміною кольорів вершин, графічними позначками тощо, або ж у процесі обходу виводити протокол обходу у графічне вікно або в консоль.

5. Якщо після обходу графа лишилися невідвідані вершини, продовжувати обхід з невідвіданої вершини з найменшим номером, яка має щонайменше одну вихідну дугу.

При проєктуванні програми **слід врахувати наступне:**

1) мова мова програмування обирається студентом самостійно;
2) графічне зображення усіх графів має формуватися програмою з тими ж вимогами, як у Лабораторній роботі номер 3;
3) всі графи обов’язково зображувати у графічному вікні;
4) типи та структури даних для внутрішнього представлення всіх даних у програмі слід вибрати самостійно.

Завдання для конкретного варіанту

Група 43, варіант №17:

$n_1n_2n_3n_4 = 4317$

Кількість вершин n: 11

Розміщення вершин: колом з вершиною в центрі

Текст програми

```
package org.arcctg;

import javax.swing.SwingUtilities;

public class Main {

    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> {
            GraphTraversalApp app = new GraphTraversalApp();
            app.setVisible(true);
        });
    }
}


package org.arcctg;

import javax.swing.*;
import javax.swing.border.EmptyBorder;
import java.awt.*;
import java.awt.geom.Ellipse2D;
import java.awt.geom.Line2D;
import java.awt.geom.Path2D;
import java.awt.geom.QuadCurve2D;
import java.util.*;

public class GraphTraversalApp extends JFrame {

    private static final int GROUP_NUMBER = 43;
    private static final int VARIANT_NUMBER = 17;
    private static final int VARIANT_CODE = GROUP_NUMBER * 100 +
VARIANT_NUMBER;

    private static final int N3 = 1;
    private static final int N4 = 7;

    private static final int VERTEX_COUNT = 10 + N3;

    private static final double K = 1.0 - N3 * 0.01 - N4 * 0.005 - 0.15;

    private static final int WINDOW_WIDTH = 1200;
    private static final int WINDOW_HEIGHT = 800;
    private static final int VERTEX_RADIUS = 20;
    private static final int ARROW_SIZE = 10;
```

```
private static final int SELF_LOOP_OFFSET = 40;
private static final int CURVE_CONTROL_OFFSET = 50;

private static final Color NOT_VISITED_COLOR = Color.WHITE;
private static final Color PROCESSED_COLOR = Color.GREEN;
private static final Color TREE_EDGE_COLOR = Color.BLUE;

private int[][] directedMatrix;
private boolean[] visited;
private int[] traversalOrder;
private boolean[][] treeEdges;
private int traversalIndex = 0;
private final Queue<Integer> queue = new LinkedList<>(); // For BFS
private final Stack<Integer> stack = new Stack<>(); // For DFS
private boolean traversalComplete = false;
private boolean isBfs = true;
private JTextArea outputArea = new JTextArea();
private GraphPanel graphPanel;
private boolean traversalStarted = false;

public GraphTraversalApp() {
    initializeWindow();
    generateMatrices();
    printMatrices();
    setupUI();
}

private void initializeWindow() {
    setTitle("Graph Traversal Visualization");
    setSize(WINDOW_WIDTH, WINDOW_HEIGHT);
    setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
    setLocationRelativeTo(null);
}

private void setupUI() {
    JPanel mainPanel = new JPanel(new BorderLayout());

    graphPanel = new GraphPanel();
    mainPanel.add(graphPanel, BorderLayout.CENTER);
    mainPanel.add(createControlPanel(), BorderLayout.NORTH);
    mainPanel.add(createOutputPanel(), BorderLayout.SOUTH);

    setFocusable(true);
    add(mainPanel);
}
```

```
    private JPanel createControlPanel() {
        JPanel controlPanel = new JPanel();

        JButton bfsButton = createButton("BFS", e -> {
            isBfs = true;
            startTraversal();
        });

        JButton dfsButton = createButton("DFS", e -> {
            isBfs = false;
            startTraversal();
        });

        JButton nextStepButton = createButton("Next Step", e -> {
            if (!traversalStarted) {
                traversalStarted = true;
                initializeTraversal();
            } else if (!traversalComplete) {
                performTraversalStep();
            }
            graphPanel.repaint();
        });

        JButton resetButton = createButton("Reset", e -> {
            resetTraversal();
            graphPanel.repaint();
        });

        controlPanel.add(bfsButton);
        controlPanel.add(dfsButton);
        controlPanel.add(nextStepButton);
        controlPanel.add(resetButton);

        return controlPanel;
    }

    private JButton createButton(String text,
java.awt.event.ActionListener listener) {
        JButton button = new JButton(text);
        button.addActionListener(listener);

        return button;
    }

    private JScrollPane createOutputPanel() {
        outputArea = new JTextArea(10, 50);
```

```
        outputArea.setEditable(false);
        JScrollPane scrollPane = new JScrollPane(outputArea);
        scrollPane.setBorder(new EmptyBorder(10, 10, 10, 10));

        return scrollPane;
    }

    private void startTraversal() {
        resetTraversal();
        traversalStarted = true;
        initializeTraversal();
        graphPanel.repaint();
    }

    private void generateMatrices() {
        directedMatrix = generateDirectedMatrix();
        resetTraversal();
    }

    private void resetTraversal() {
        visited = new boolean[VERTEX_COUNT];
        traversalOrder = new int[VERTEX_COUNT];
        Arrays.fill(traversalOrder, -1);
        treeEdges = new boolean[VERTEX_COUNT][VERTEX_COUNT];
        traversalIndex = 0;
        queue.clear();
        stack.clear();
        traversalComplete = false;
        traversalStarted = false;
        outputArea.setText("");
        outputArea.append("Generated Directed Graph Adjacency
Matrix:\n");
        printMatrixToOutput(directedMatrix);
    }

    private void initializeTraversal() {
        outputArea.append("\nStarting " + (isBfs ? "BFS" : "DFS") + "
Traversal\n");

        int startVertex = findStartVertex();

        if (!isValidStartVertex(startVertex)) {
            return;
        }

        outputArea.append("Starting from vertex " + (startVertex + 1) +
```

```
"\n");

        initializeAlgorithmDataStructure(startVertex);
        markVertexAsVisited(startVertex);
    }

    private boolean isValidStartVertex(int startVertex) {
        if (startVertex == -1) {
            outputArea.append("No vertex with outgoing edges found.
Cannot start traversal.\n");
            traversalComplete = true;

            return false;
        }
        return true;
    }

    private void initializeAlgorithmDataStructure(int vertex) {
        if (isBfs) {
            queue.add(vertex);
        } else {
            stack.push(vertex);
        }
    }

    private void markVertexAsVisited(int vertex) {
        visited[vertex] = true;
        traversalOrder[traversalIndex++] = vertex;
    }

    private int findStartVertex() {
        for (int i = 0; i < VERTEX_COUNT; i++) {
            if (hasOutgoingEdge(i)) {
                return i;
            }
        }
        return -1;
    }

    private boolean hasOutgoingEdge(int vertex) {
        for (int j = 0; j < VERTEX_COUNT; j++) {
            if (directedMatrix[vertex][j] == 1) {
                return true;
            }
        }
        return false;
```

```
    }

    private int findNextUnvisitedVertex() {
        for (int i = 0; i < VERTEX_COUNT; i++) {
            if (!visited[i]) {
                for (int j = 0; j < VERTEX_COUNT; j++) {
                    if (directedMatrix[i][j] == 1) {
                        return i;
                    }
                }
            }
        }

        return -1;
    }

    private void performTraversalStep() {
        if (isBfs) {
            performBFSStep();
        } else {
            performDFSStep();
        }
    }

    private void performBFSStep() {
        if (queue.isEmpty()) {
            handleEmptyQueue();
            return;
        }

        processVertexInBFS();
    }

    private void handleEmptyQueue() {
        int nextStart = findNextUnvisitedVertex();
        if (nextStart != -1) {
            startNewBfsComponent(nextStart);
        } else {
            completeBfsTraversal();
        }
    }

    private void startNewBfsComponent(int vertex) {
        outputArea.append("Continuing BFS from vertex " + (vertex + 1) +
"\n");
        queue.add(vertex);
```

```
        markVertexAsVisited(vertex);
    }

    private void completeBfsTraversal() {
        outputArea.append("BFS Traversal Complete\n");
        printTraversalOrder();
        printTreeMatrix();
        traversalComplete = true;
    }

    private void processVertexInBFS() {
        if (queue.isEmpty()) return;
        int currentVertex = queue.peek();

        boolean foundUnvisited =
findAndProcessUnvisitedNeighbor(currentVertex);

        if (!foundUnvisited) {
            outputArea.append("Processing vertex " + (currentVertex + 1)
+ " (all neighbors discovered)\n");
            queue.remove();
            processVertexInBFS();
        }
    }

    private boolean findAndProcessUnvisitedNeighbor(int currentVertex) {
        for (int i = 0; i < VERTEX_COUNT; i++) {
            if (directedMatrix[currentVertex][i] == 1 && !visited[i]) {
                outputArea.append("Discovered vertex " + (i + 1) + "
from " + (currentVertex + 1) + "\n");
                queue.add(i);
                markVertexAsVisited(i);
                treeEdges[currentVertex][i] = true;
                return true;
            }
        }
        return false;
    }

    private void performDFSStep() {
        if (stack.isEmpty()) {
            handleEmptyStack();
            return;
        }

        processVertexInDFS();
```

```
    }

    private void handleEmptyStack() {
        int nextStart = findNextUnvisitedVertex();
        if (nextStart != -1) {
            startNewDfsComponent(nextStart);
        } else {
            completeDfsTraversal();
        }
    }

    private void startNewDfsComponent(int vertex) {
        outputArea.append("Continuing DFS from vertex " + (vertex + 1) +
"\n");
        stack.push(vertex);
        markVertexAsVisited(vertex);
    }

    private void completeDfsTraversal() {
        outputArea.append("DFS Traversal Complete\n");
        printTraversalOrder();
        printTreeMatrix();
        traversalComplete = true;
    }

    private void processVertexInDFS() {
        if (stack.isEmpty()) return;

        int currentVertex = stack.peek();
        boolean foundUnvisited = exploreAdjacentVertices(currentVertex);

        if (!foundUnvisited) {
            outputArea.append("Processing vertex " + (currentVertex + 1)
+ " (all neighbors discovered)\n");
            stack.pop();
            processVertexInDFS();
        }
    }

    private boolean exploreAdjacentVertices(int currentVertex) {
        for (int i = 0; i < VERTEX_COUNT; i++) {
            if (directedMatrix[currentVertex][i] == 1 && !visited[i]) {
                outputArea.append("Discovered vertex " + (i + 1) + "
from " + (currentVertex + 1) + "\n");
                stack.push(i);
                markVertexAsVisited(i);
```

```
                treeEdges[currentVertex][i] = true;

                return true;
            }
        }

        return false;
    }

    private int[][] generateDirectedMatrix() {
        Random random = new Random(VARIANT_CODE);
        int[][] resultMatrix = new int[VERTEX_COUNT][VERTEX_COUNT];

        for (int i = 0; i < VERTEX_COUNT; i++) {
            for (int j = 0; j < VERTEX_COUNT; j++) {
                double value = random.nextDouble(0.0, 2.0) * K;
                resultMatrix[i][j] = value >= 1.0 ? 1 : 0;
            }
        }

        return resultMatrix;
    }

    private void printMatrices() {
        System.out.println("Directed Graph Adjacency Matrix (K = " + K +
"):");
        printMatrix(directedMatrix);
    }

    private void printMatrix(int[][] matrix) {
        for (int[] row : matrix) {
            for (int cell : row) {
                System.out.print(cell + " ");
            }
            System.out.println();
        }
    }

    private void printMatrixToOutput(int[][] matrix) {
        for (int[] row : matrix) {
            for (int cell : row) {
                outputArea.append(cell + " ");
            }
            outputArea.append("\n");
        }
    }
```

```
    private void printTraversalOrder() {
        outputArea.append("\nTraversal Order (Original Vertex Number ->
Order in Traversal):\n");
        for (int i = 0; i < traversalIndex; i++) {
            int vertex = traversalOrder[i];
            outputArea.append("Vertex " + (vertex + 1) + " -> " + (i +
1) + "\n");
        }
    }

    private void printTreeMatrix() {
        outputArea.append("\nTraversal Tree Adjacency Matrix:\n");
        for (int i = 0; i < VERTEX_COUNT; i++) {
            for (int j = 0; j < VERTEX_COUNT; j++) {
                outputArea.append(treeEdges[i][j] ? "1 " : "0 ");
            }
            outputArea.append("\n");
        }
    }

    class GraphPanel extends JPanel {
        @Override
        protected void paintComponent(Graphics g) {
            super.paintComponent(g);
            Graphics2D g2d = (Graphics2D) g;
            setupGraphics(g2d);

            int panelWidth = getWidth();
            int panelHeight = getHeight();

            drawDirectedGraph(g2d, 0, 0, panelWidth / 2, panelHeight);

            drawTraversalTree(g2d, panelWidth / 2, 0, panelWidth / 2,
panelHeight);

            drawTitles(g2d, panelWidth);
        }

        private void setupGraphics(Graphics2D g2d) {
            g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
RenderingHints.VALUE_ANTIALIAS_ON);
            g2d.setRenderingHint(RenderingHints.KEY_TEXT_ANTIALIASING,
RenderingHints.VALUE_TEXT_ANTIALIAS_ON);
            g2d.setStroke(new BasicStroke(1.5f));
        }
```

```
        private void drawTitles(Graphics2D g2d, int panelWidth) {
            g2d.setColor(Color.BLACK);
            g2d.setFont(new Font("Arial", Font.BOLD, 20));
            g2d.drawString("Directed Graph", 100, 30);
            g2d.drawString("Traversal Tree (" + (isBfs ? "BFS" : "DFS")
+ ")", panelWidth / 2 + 100, 30);
        }

        private void drawDirectedGraph(Graphics2D g2d, int x, int y, int
width, int height) {
            int centerX = x + width / 2;
            int centerY = y + height / 2 + 10;
            int radius = Math.min(width, height) / 3;

            Point[] vertexPositions = calculateVertexPositions(centerX,
centerY, radius);

            drawGraphEdges(g2d, vertexPositions);
            drawGraphVertices(g2d, vertexPositions);
        }

        private void drawGraphEdges(Graphics2D g2d, Point[]
vertexPositions) {
            boolean[][] bidirectionalEdges =
findBidirectionalEdges(directedMatrix);

            g2d.setColor(Color.BLACK);
            for (int i = 0; i < VERTEX_COUNT; i++) {
                for (int j = 0; j < VERTEX_COUNT; j++) {
                    if (directedMatrix[i][j] == 1) {
                        boolean isBidirectional =
bidirectionalEdges[i][j];
                        drawEdge(g2d, vertexPositions[i],
vertexPositions[j], i, j, vertexPositions[0],
                                true, isBidirectional);
                    }
                }
            }
        }

        private boolean[][] findBidirectionalEdges(int[][] matrix) {
            boolean[][] bidirectionalEdges = new
boolean[VERTEX_COUNT][VERTEX_COUNT];

            for (int i = 0; i < VERTEX_COUNT; i++) {
```

```
                for (int j = 0; j < VERTEX_COUNT; j++) {
                    if (matrix[i][j] == 1 && matrix[j][i] == 1 && i !=
j) {
                        bidirectionalEdges[i][j] = true;
                        bidirectionalEdges[j][i] = true;
                    }
                }
            }

            return bidirectionalEdges;
        }

        private void drawGraphVertices(Graphics2D g2d, Point[]
vertexPositions) {
            for (int i = 0; i < VERTEX_COUNT; i++) {
                Color vertexColor = determineVertexColor(i);
                drawVertex(g2d, vertexPositions[i], i + 1, vertexColor);
            }
        }

        private Color determineVertexColor(int vertexIndex) {
            if (traversalStarted && isVertexInTraversal(vertexIndex)) {
                return PROCESSED_COLOR;
            }
            return NOT_VISITED_COLOR;
        }

        private void drawTraversalTree(Graphics2D g2d, int x, int y, int
width, int height) {
            if (!traversalStarted) return;

            int centerX = x + width / 2;
            int centerY = y + height / 2 + 10;
            int radius = Math.min(width, height) / 3;

            Point[] vertexPositions = calculateVertexPositions(centerX,
centerY, radius);

            drawTraversalTreeEdges(g2d, vertexPositions);
            drawTraversalTreeVertices(g2d, vertexPositions);
        }

        private void drawTraversalTreeEdges(Graphics2D g2d, Point[]
vertexPositions) {
            g2d.setColor(TREE_EDGE_COLOR);
            for (int i = 0; i < VERTEX_COUNT; i++) {
```

```
                for (int j = 0; j < VERTEX_COUNT; j++) {
                    if (treeEdges[i][j]) {
                        drawEdge(g2d, vertexPositions[i],
vertexPositions[j], i, j, vertexPositions[0],
                                true, false);
                    }
                }
            }
        }

        private void drawTraversalTreeVertices(Graphics2D g2d, Point[]
vertexPositions) {
            for (int i = 0; i < VERTEX_COUNT; i++) {
                Color vertexColor = isVertexInTraversal(i) ?
PROCESSED_COLOR : NOT_VISITED_COLOR;
                drawVertex(g2d, vertexPositions[i], i + 1, vertexColor);
            }
        }

        private boolean isVertexInTraversal(int vertex) {
            for (int i = 0; i < traversalIndex; i++) {
                if (traversalOrder[i] == vertex) {
                    return true;
                }
            }
            return false;
        }

        private Point[] calculateVertexPositions(int centerX, int
centerY, int radius) {
            Point[] positions = new Point[VERTEX_COUNT];

            positions[0] = new Point(centerX, centerY);

            double angleStep = 2 * Math.PI / (VERTEX_COUNT - 1);
            for (int i = 1; i < VERTEX_COUNT; i++) {
                int vx = centerX + (int) (radius * Math.cos((i - 1) *
angleStep));
                int vy = centerY + (int) (radius * Math.sin((i - 1) *
angleStep));
                positions[i] = new Point(vx, vy);
            }

            return positions;
        }
```

```
        private void drawVertex(Graphics2D g2d, Point position, int
index, Color color) {
            g2d.setColor(color);
            Ellipse2D.Double circle = new Ellipse2D.Double(
                position.x - VERTEX_RADIUS,
                position.y - VERTEX_RADIUS,
                2.0 * VERTEX_RADIUS,
                2.0 * VERTEX_RADIUS
            );
            g2d.fill(circle);

            g2d.setColor(Color.BLACK);
            g2d.draw(circle);

            drawVertexLabel(g2d, position, index);
        }

        private void drawVertexLabel(Graphics2D g2d, Point position, int
index) {
            g2d.setFont(new Font("Arial", Font.BOLD, 14));
            String label = String.valueOf(index);
            FontMetrics metrics = g2d.getFontMetrics();
            int labelWidth = metrics.stringWidth(label);
            int labelHeight = metrics.getHeight();

            g2d.drawString(
                label,
                position.x - labelWidth / 2,
                position.y + labelHeight / 4
            );
        }

        private void drawEdge(Graphics2D g2d, Point from, Point to, int
fromIndex, int toIndex,
            Point centerPoint, boolean isDirected, boolean
isBidirectional) {
            boolean throughCenter = fromIndex != 0 && toIndex != 0 &&
linePassesThroughCenter(from, to, centerPoint);

            if (fromIndex == toIndex) {
                drawSelfLoop(g2d, from, fromIndex);
            } else if (isBidirectional || throughCenter) {
                drawCurvedEdge(g2d, from, to, isDirected,
isBidirectional);
            } else {
                drawStraightEdge(g2d, from, to, isDirected);
```

```
            }
        }

        private boolean linePassesThroughCenter(Point from, Point to,
Point center) {
            double distance = distanceFromPointToLine(center, from, to);
            return distance < 2 * VERTEX_RADIUS;
        }

        private double distanceFromPointToLine(Point point, Point
lineStart, Point lineEnd) {
            double numerator = Math.abs(
                (lineEnd.y - lineStart.y) * point.x -
                    (lineEnd.x - lineStart.x) * point.y +
                    lineEnd.x * lineStart.y -
                    lineEnd.y * lineStart.x
            );

            double denominator = Math.sqrt(
                Math.pow(lineEnd.y - lineStart.y, 2) +
                    Math.pow(lineEnd.x - lineStart.x, 2)
            );

            return numerator / denominator;
        }

        private void drawCurvedEdge(Graphics2D g2d, Point from, Point
to, boolean isDirected,
            boolean isBidirectional) {
            double dx = to.x - from.x;
            double dy = to.y - from.y;
            double length = Math.sqrt(dx * dx + dy * dy);

            double perpX = -dy / length;
            double perpY = dx / length;

            int curveOffset = isBidirectional ? 15 :
CURVE_CONTROL_OFFSET;

            double midX = (from.x + to.x) / 2.0;
            double midY = (from.y + to.y) / 2.0;
            int controlX = (int) (midX + perpX * curveOffset);
            int controlY = (int) (midY + perpY * curveOffset);

            double startAngle = Math.atan2(controlY - from.y, controlX -
from.x);
```

```
            int startX = from.x + (int) (VERTEX_RADIUS *
Math.cos(startAngle));
            int startY = from.y + (int) (VERTEX_RADIUS *
Math.sin(startAngle));

            double endAngle = Math.atan2(controlY - to.y, controlX -
to.x);
            int endX = to.x + (int) (VERTEX_RADIUS *
Math.cos(endAngle));
            int endY = to.y + (int) (VERTEX_RADIUS *
Math.sin(endAngle));

            g2d.draw(new QuadCurve2D.Double(startX, startY, controlX,
controlY, endX, endY));

            if (isDirected) {
                double t = 0.95;
                double curvePointX = (1-t)*(1-t)*startX + 2*(1-
t)*t*controlX + t*t*endX;
                double curvePointY = (1-t)*(1-t)*startY + 2*(1-
t)*t*controlY + t*t*endY;

                drawArrow(g2d, (int)curvePointX, (int)curvePointY, endX,
endY);
            }
        }

        private void drawStraightEdge(Graphics2D g2d, Point from, Point
to, boolean isDirected) {
            double dx = to.x - from.x;
            double dy = to.y - from.y;
            double length = Math.sqrt(dx * dx + dy * dy);

            double nx = dx / length;
            double ny = dy / length;

            int startX = from.x + (int) (nx * VERTEX_RADIUS);
            int startY = from.y + (int) (ny * VERTEX_RADIUS);
            int endX = to.x - (int) (nx * VERTEX_RADIUS);
            int endY = to.y - (int) (ny * VERTEX_RADIUS);

            g2d.draw(new Line2D.Double(startX, startY, endX, endY));

            if (isDirected) {
                drawArrow(g2d, startX, startY, endX, endY);
            }
```

```
        }

        private void drawArrow(Graphics2D g2d, int x1, int y1, int x2,
int y2) {
            double dx = x2 - x1;
            double dy = y2 - y1;
            double angle = Math.atan2(dy, dx);

            Path2D.Double path = new Path2D.Double();
            path.moveTo(x2, y2);
            path.lineTo(x2 - ARROW_SIZE * Math.cos(angle - Math.PI/6),
                y2 - ARROW_SIZE * Math.sin(angle - Math.PI/6));
            path.lineTo(x2 - ARROW_SIZE * Math.cos(angle + Math.PI/6),
                y2 - ARROW_SIZE * Math.sin(angle + Math.PI/6));
            path.closePath();

            g2d.fill(path);
        }

        private void drawSelfLoop(Graphics2D g2d, Point vertex, int
vertexIndex) {
            double angleOffset = (vertexIndex == 0) ? Math.PI / 4 :
getAngleForVertex(vertexIndex);

            int loopSize = VERTEX_RADIUS * 3;
            int offsetX = (int) (SELF_LOOP_OFFSET *
Math.cos(angleOffset));
            int offsetY = (int) (SELF_LOOP_OFFSET *
Math.sin(angleOffset));

            int loopX = vertex.x + offsetX - loopSize / 2;
            int loopY = vertex.y + offsetY - loopSize / 2;

            g2d.drawOval(loopX, loopY, loopSize, loopSize);
        }

        private double getAngleForVertex(int vertexIndex) {
            return vertexIndex == 0 ?
                Math.PI / 4 :
                2 * Math.PI * (vertexIndex - 1) / (VERTEX_COUNT - 1);
        }
    }
}
```

Згенерована матриця суміжності напрямленого графа

```
0 0 0 1 0 0 0 0 1 0 0
0 0 0 1 0 1 0 0 1 0 1
0 0 0 1 1 1 0 1 1 1 1
1 0 0 1 0 0 0 0 0 0 1
1 0 0 0 0 1 1 1 0 0 1
0 1 0 1 0 0 0 0 1 1 0
1 0 0 1 0 1 0 1 0 0 0
0 1 0 1 0 0 0 0 1 1 0
0 0 1 0 0 0 0 0 1 0 1
1 1 0 1 1 0 0 1 0 0 0
1 0 0 0 1 1 0 0 1 0 1
```

Матриця суміжності дерева обходу

BFS:

```
0 0 0 1 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 1 0
0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 1 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 1 0 0 0 0 0
```

DFS:

```
0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 1 1 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0
0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0
```

Список відповідності номерів вершин і їх нової нумерації

Traversal Order (Original Vertex Number -> Order in Traversal):

BFS:

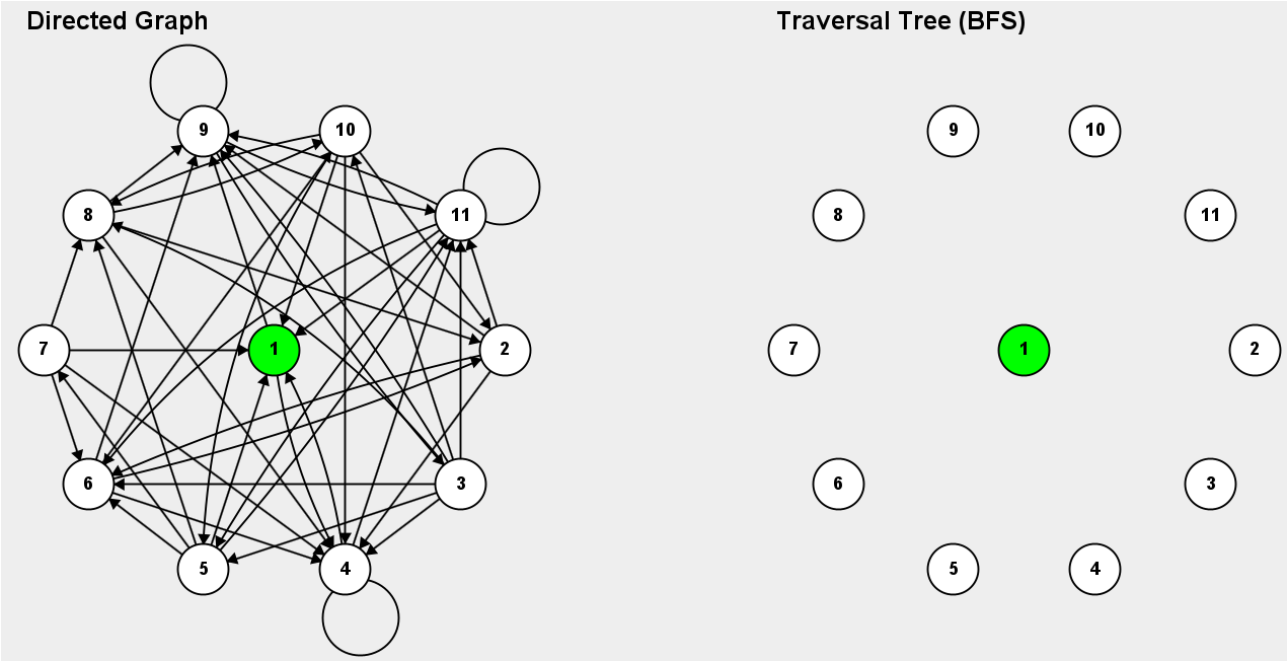
Vertex 1 -> 1
Vertex 4 -> 2
Vertex 9 -> 3
Vertex 11 -> 4
Vertex 3 -> 5
Vertex 5 -> 6
Vertex 6 -> 7
Vertex 8 -> 8
Vertex 10 -> 9
Vertex 7 -> 10
Vertex 2 -> 11

DFS:

Vertex 1 -> 1
Vertex 4 -> 2
Vertex 11 -> 3
Vertex 5 -> 4
Vertex 6 -> 5
Vertex 2 -> 6
Vertex 9 -> 7
Vertex 3 -> 8
Vertex 8 -> 9
Vertex 10 -> 10
Vertex 7 -> 11

Скриншоти зображення графа та дерева обходу

BFS на початку обходу:

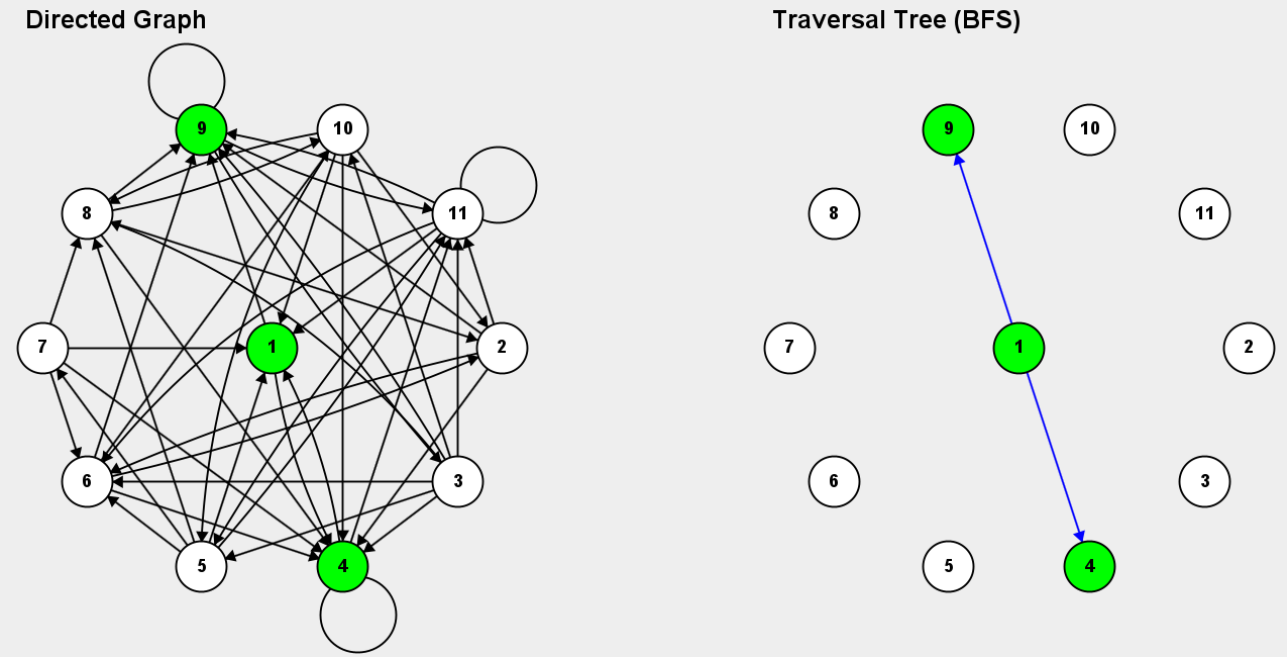


Консоль:

Starting BFS Traversal

Starting from vertex 1

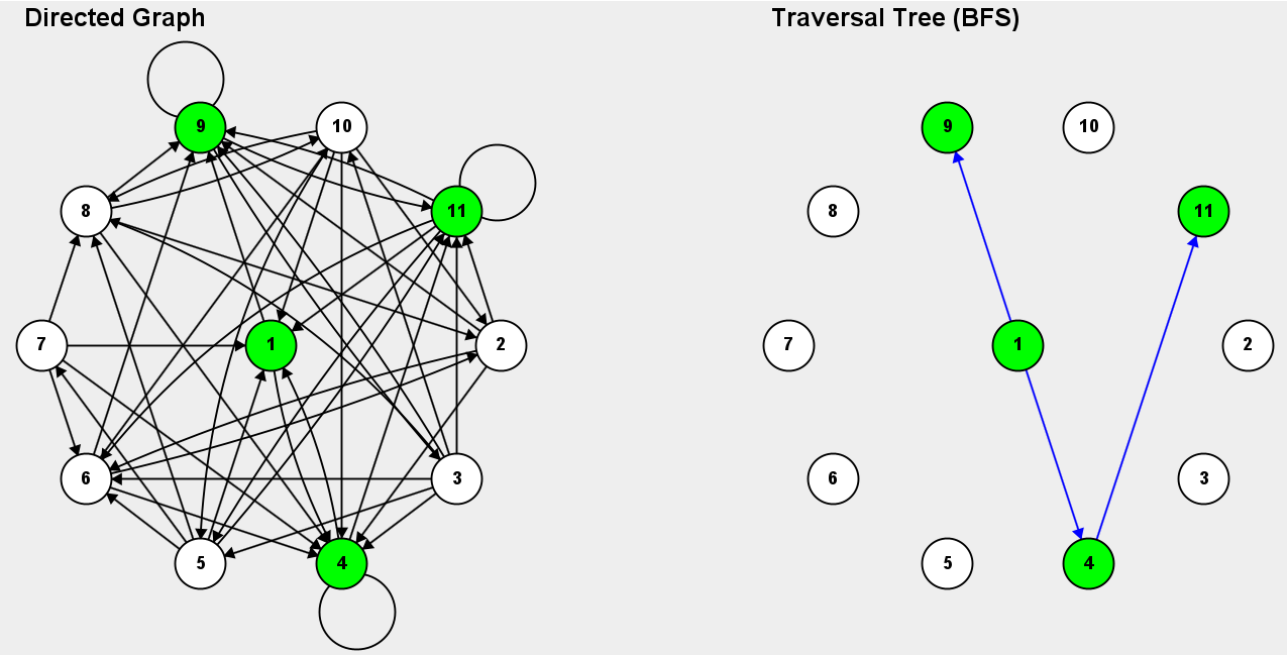
BFS у процесі обходу:



Консоль:

Discovered vertex 4 from 1

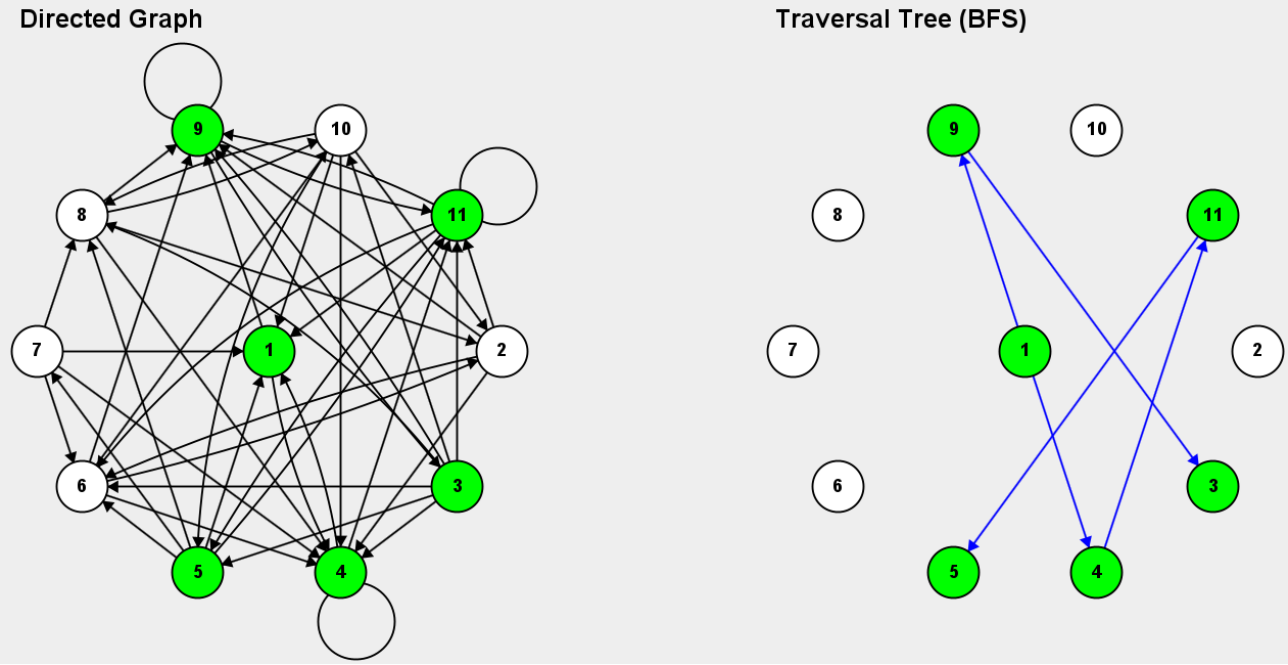
Discovered vertex 9 from 1



Консоль:

Processing vertex 1 (all neighbors discovered)

Discovered vertex 11 from 4



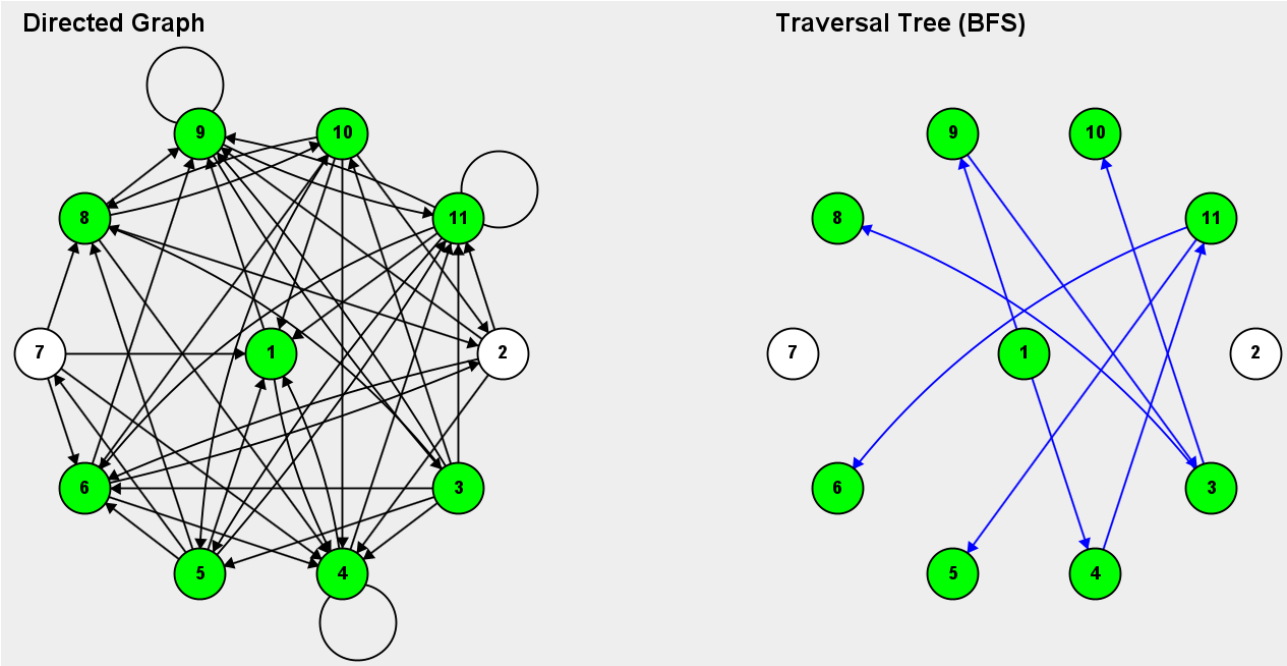
Консоль:

Processing vertex 4 (all neighbors discovered)

Discovered vertex 3 from 9

Processing vertex 9 (all neighbors discovered)

Discovered vertex 5 from 11



Консоль:

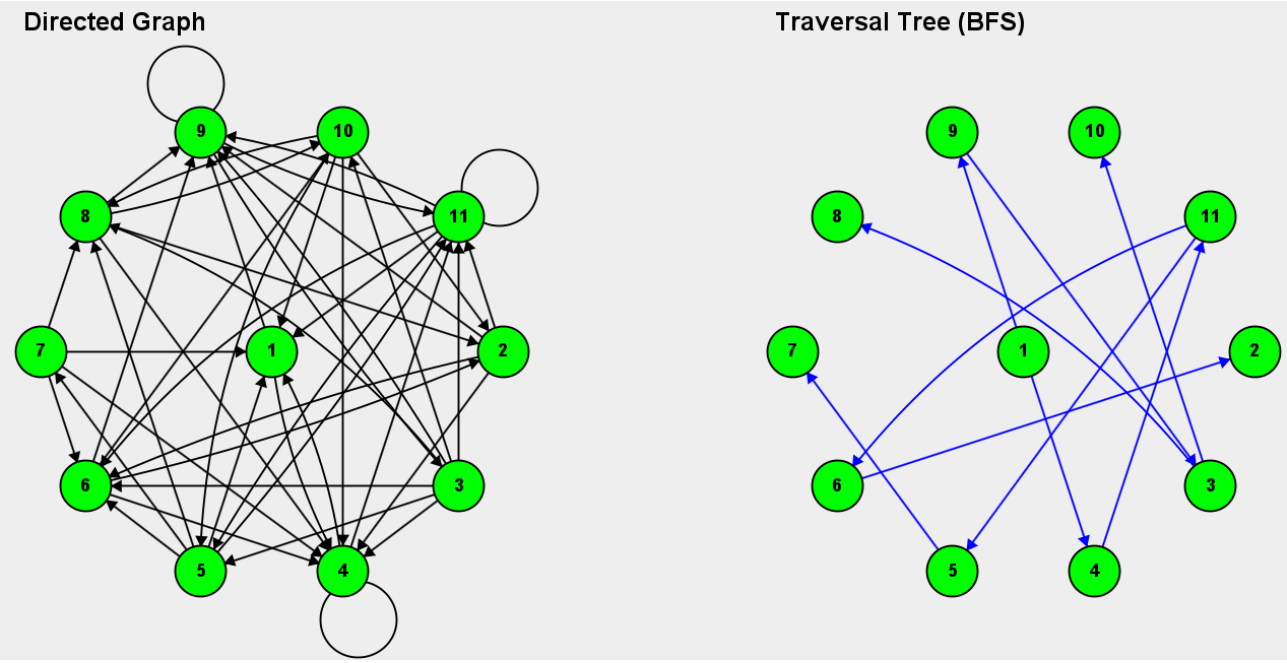
Discovered vertex 6 from 11

Processing vertex 11 (all neighbors discovered)

Discovered vertex 8 from 3

Discovered vertex 10 from 3

BFS після завершення обходу:



Консоль:

Processing vertex 3 (all neighbors discovered)

Discovered vertex 7 from 5

Processing vertex 5 (all neighbors discovered)

Discovered vertex 2 from 6

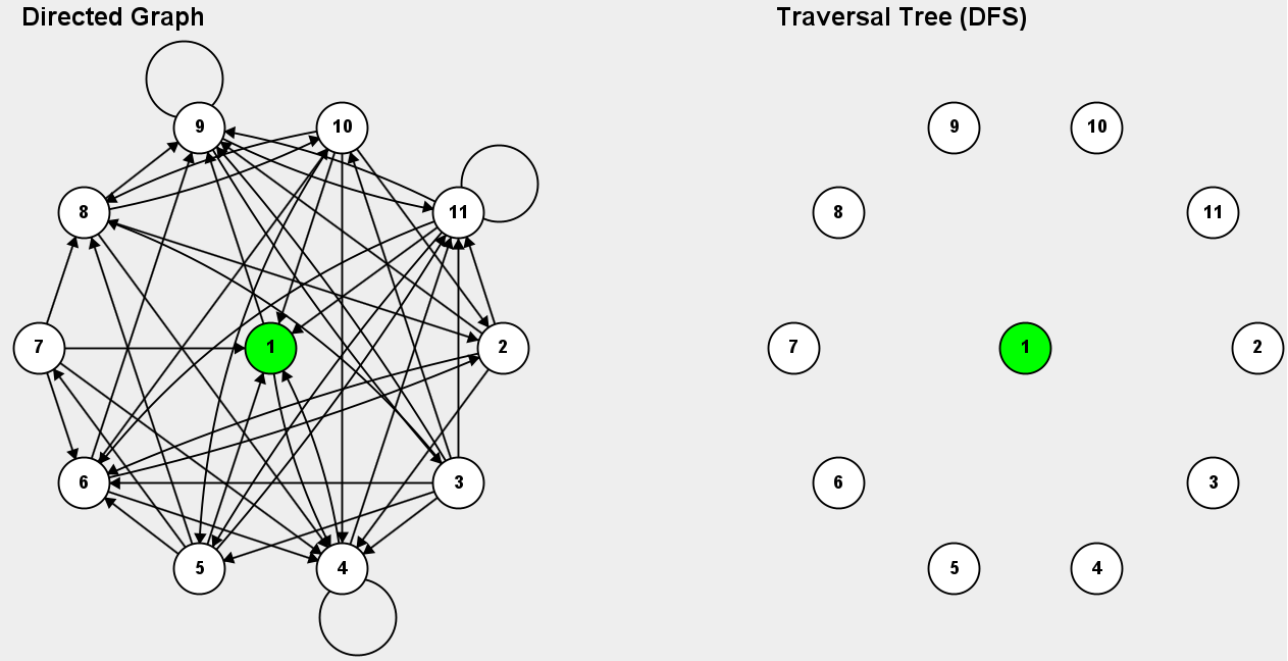
Processing vertex 6 (all neighbors discovered)

Processing vertex 8 (all neighbors discovered)

Processing vertex 10 (all neighbors discovered)

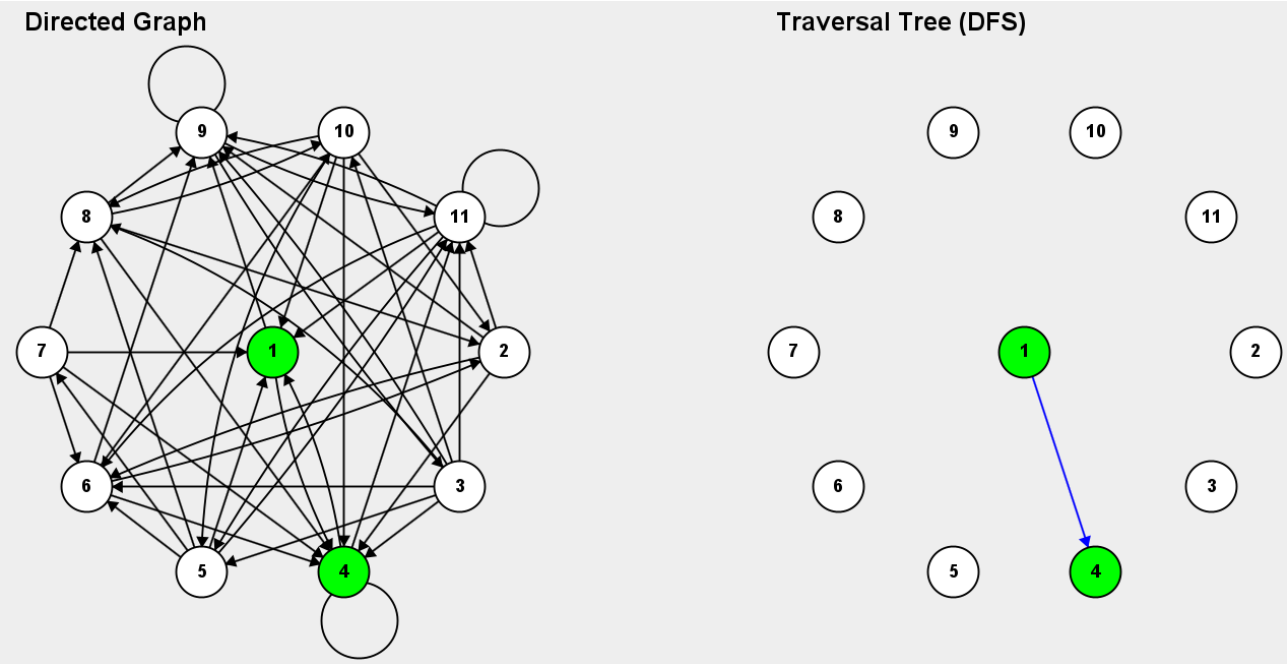
Processing vertex 7 (all neighbors discovered)
Processing vertex 2 (all neighbors discovered)
BFS Traversal Complete

DFS на початку обходу:

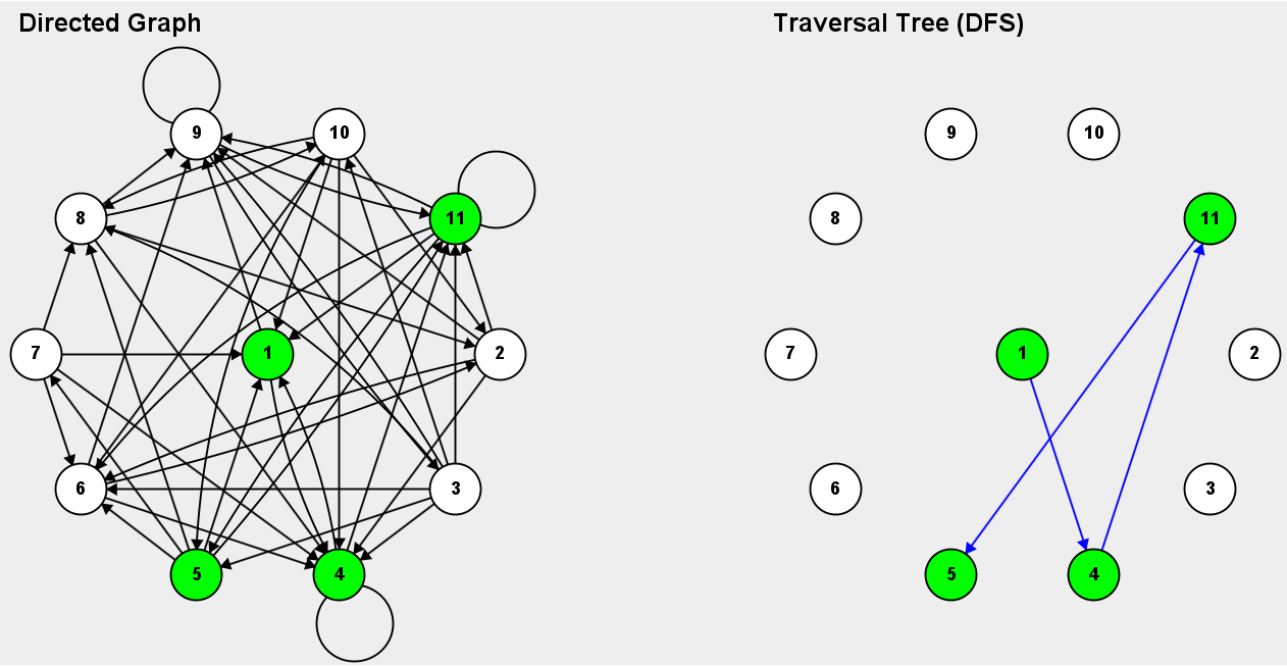


Консоль:
Starting DFS Traversal
Starting from vertex 1

DFS у процесі обходу:



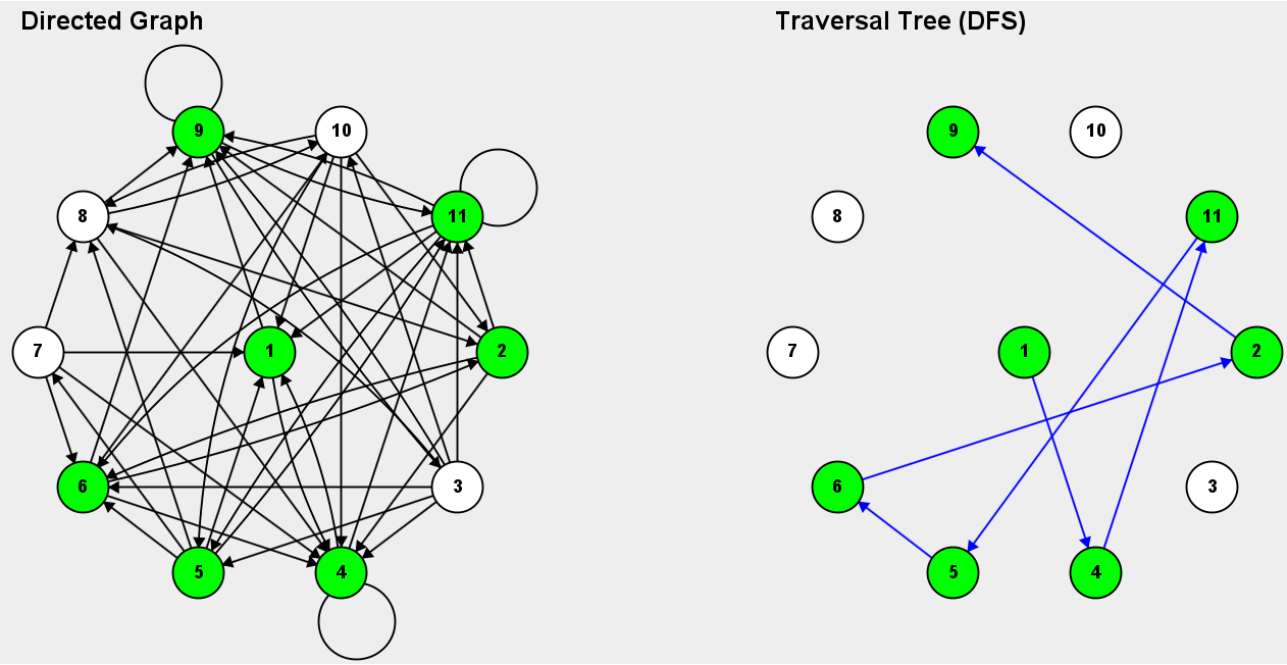
Консоль:
Discovered vertex 4 from 1



Консоль:

Discovered vertex 11 from 4

Discovered vertex 5 from 11

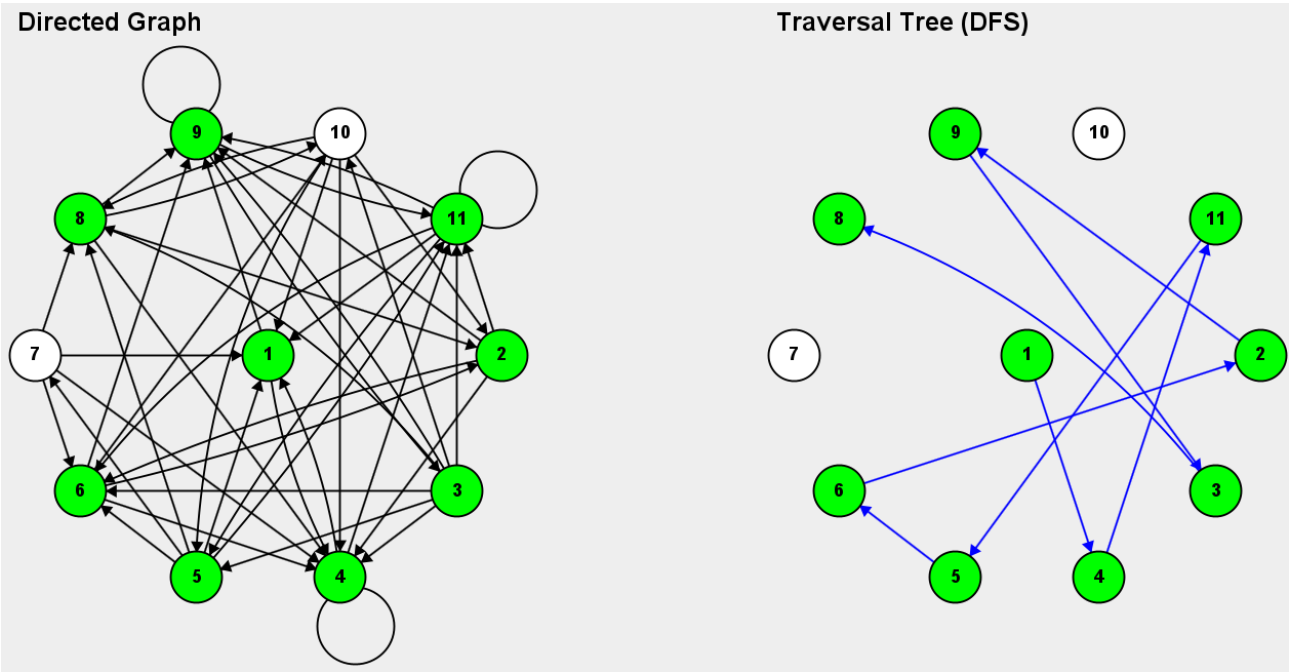


Консоль:

Discovered vertex 6 from 5

Discovered vertex 2 from 6

Discovered vertex 9 from 2

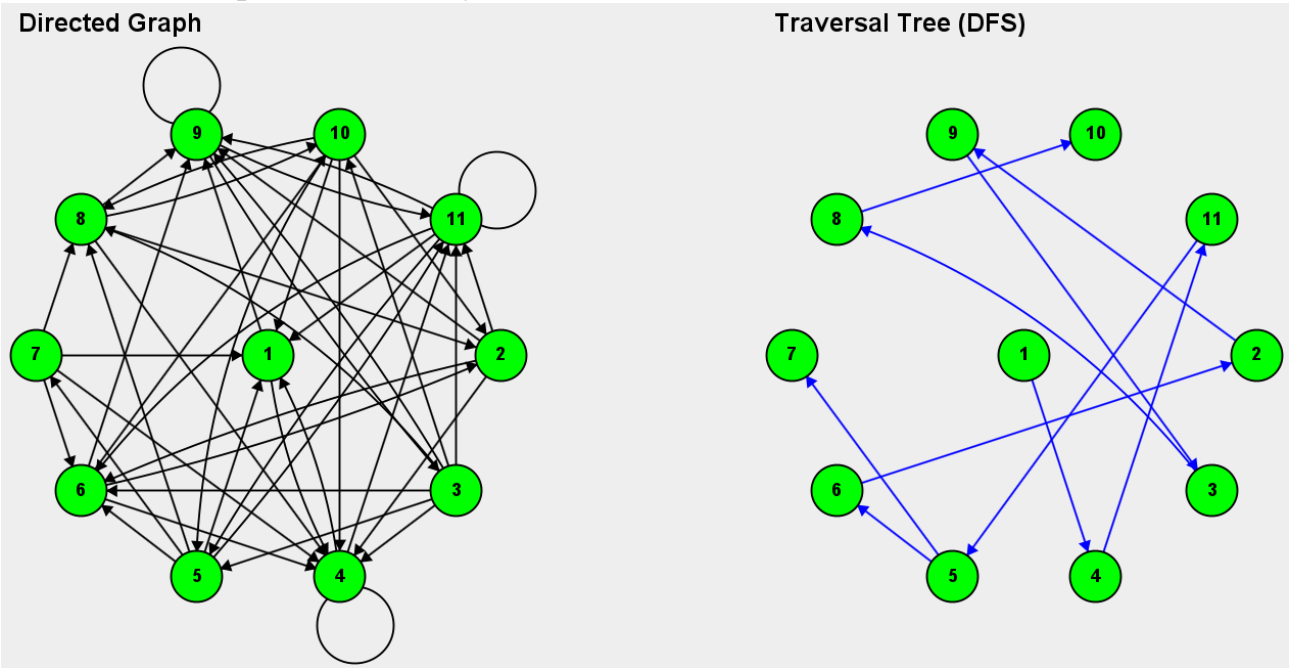


Консоль:

Discovered vertex 3 from 9

Discovered vertex 8 from 3

DFS після завершення обходу:



Консоль:

Discovered vertex 10 from 8

Processing vertex 10 (all neighbors discovered)

Processing vertex 8 (all neighbors discovered)

Processing vertex 3 (all neighbors discovered)

Processing vertex 9 (all neighbors discovered)

Processing vertex 2 (all neighbors discovered)

Processing vertex 6 (all neighbors discovered)

Discovered vertex 7 from 5

Processing vertex 7 (all neighbors discovered)

Processing vertex 5 (all neighbors discovered)
Processing vertex 11 (all neighbors discovered)
Processing vertex 4 (all neighbors discovered)
Processing vertex 1 (all neighbors discovered)
DFS Traversal Complete

Висновок

Під час виконання лабораторної роботи № 5 я засвоїв теоретичні основи та набув практичних навичок дослідження структури графа за допомогою фундаментальних алгоритмів обходу його вершин: обходу в ширину (BFS) та обходу в глибину (DFS). Я поглибив свої знання щодо механізмів роботи цих алгоритмів та їх застосування для систематичного відвідування всіх доступних вершин графа, а також навчився формувати дерево обходу як результат цього процесу.

Я набув практичного досвіду в програмній реалізації алгоритмів BFS та DFS для напрямленого графа, представленого матрицею суміжності. Це включало вибір початкової вершини згідно з умовою, врахування порядку нумерації вершин при обході, обробку ситуацій з незв'язними компонентами графа, а також візуалізацію зміни статусів вершин під час виконання алгоритмів.

Під час роботи я реалізував зазначені алгоритми обходу на основі матриці суміжності напрямленого графа, яка була згенерована відповідно до умов завдання з модифікованим k. Використовуючи мову програмування Java та бібліотеку Swing я розробив програму, яка візуалізує вихідний граф та забезпечує покрокове інтерактивне відображення процесу обходу (як BFS, так і DFS) та динамічну побудову відповідного дерева обходу безпосередньо у графічному вікні. Програма також виводить у графічне вікно необхідні для аналізу дані, такі як матриця суміжності дерева обходу та порядок відвідування вершин.

Виконання цього завдання допомогло мені краще зрозуміти принципи роботи ключових алгоритмів обходу графів, які є основою для вирішення багатьох інших задач, таких як пошук шляхів, перевірка зв'язності, тощо. Також я вдосконалив навички візуалізації алгоритмічних процесів та роботи з представленням графів у програмному коді.

Отже, виконання лабораторної роботи № 5 було корисним, дозволило закріпити теоретичні знання щодо алгоритмів BFS та DFS, набути цінних практичних навичок їх реалізації та візуалізації мовою Java, а також зрозуміти їх фундаментальну роль у теорії графів та комп'ютерних науках загалом.