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 №6

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 зважений ненапрямлений граф із заданими параметрами так само, як у лабораторній роботі №3.

   Відмінність 1: коефіцієнт $k = 1.0 - n_3 * 0.01 - n_4 * 0.005 - 0.05$.

   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 $k = 1.0 - n_3 * 0.01 - n_4 * 0.005 - 0.05$, 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.

   Матриця A_{undir} ненапрямленого графа одержується з матриці A_{dir} так само, як у ЛР №3.

   Відмінність 2: матриця ваг W формується таким чином.

   1) матриця B розміром n * n заповнюється згенерованими випадковими числами в діапазоні [0, 2.0) (параметр генератора випадкових чисел той же самий, $n_1n_2n_3n_4$);
   2) одержується матриця C:

      $c_{(i, j)} = ceil(b_{(i, j)} * 100 * a_{undir(i, j)})$, $c_{(i, j)} \in C, b_{(i, j)} \in B, a_{undir(i, j)} \in A_{undir}$,

      де ceil — це функція, що округляє кожен елемент матриці до найближчого цілого числа, більшого чи рівного за дане;

   3) одержується матриця D, у якій

      $d_{(i, j)} = 0$, якщо $c_{(i, j)} = 0$,

      $d_{(i, j)} = 1$, якщо $c_{(i, j)} > 0$, $d_{(i, j)} \in D, c_{(i, j)} \in C$;

   4) одержується матриця H, у якій

      $h_{(i, j)} = 1$, якщо $d_{(i, j)} \neq d_{(j, i)}$,

      та $h_{(i, j)} = 0$ в іншому випадку;

   5) Tr — верхня трикутна матриця з одиниць ($tr_{(i, j)} = 1$ при $i < j$);
   6) матриця ваг W симетрична, і її елементи одержуються за формулою:

      $w_{(i, j)} = w_{(j, i)} = (d_{(i, j)} + h_{(i, j)} * tr_{(i, j)}) * c_{(i, j)}$

2. Створити програму для знаходження мінімального кістяка за алгоритмом Краскала при n_4 — парному і за алгоритмом Пріма — при непарному. При цьому у програмі:
   - графи представляти у вигляді динамічних списків, обхід графа, додавання, віднімання вершин, ребер виконувати як функції з вершинами відповідних списків;
   - у програмі виконання обходу відображати покроково, черговий крок виконувати за натисканням кнопки у вікні або на клавіатурі.
3. Під час обходу графа побудувати дерево його кістяка. У програмі дерево кістяка виводити покроково у процесі виконання алгоритму. Це можна виконати одним із двох способів:
   - або виділяти іншим кольором ребра графа;
   - або будувати кістяк поряд із графом.

   При зображенні як графа, так і його кістяка, вказати ваги ребер.

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 ЛР №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(() -> {
            MinimumSpanningTree app = new MinimumSpanningTree();
            app.setVisible(true);
        });
    }
}


package org.arcctg;

import javax.swing.*;
import java.awt.*;
import java.awt.geom.Ellipse2D;
import java.awt.geom.Line2D;
import java.awt.geom.QuadCurve2D;
import java.util.*;
import java.util.List;

public class MinimumSpanningTree 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 - 0.05;

    private static final int WINDOW_WIDTH = 1400;
    private static final int WINDOW_HEIGHT = 800;
    private static final int VERTEX_RADIUS = 20;
    private static final int SELF_LOOP_OFFSET = 40;
    private static final int CURVE_CONTROL_OFFSET = 50;
```

```
    private int[][] directedMatrix;
    private int[][] undirectedMatrix;
    private int[][] weightMatrix;
    private final List<Edge> mstEdges = new ArrayList<>();
    private final List<Edge> allEdges = new ArrayList<>();
    private final Set<Integer> mstVertices = new HashSet<>();
    private int currentStep = -1;
    private boolean algorithmCompleted = false;
    private int totalMstWeight = 0;

    private JLabel statusLabel;
    private JTextArea infoTextArea;

    private final Map<Integer, List<VertexConnection>> adjacencyList =
new HashMap<>();

    static class VertexConnection {
        int vertex;
        int weight;

        VertexConnection(int vertex, int weight) {
            this.vertex = vertex;
            this.weight = weight;
        }
    }

    static class Edge implements Comparable<Edge> {
        int src;
        int dest;
        int weight;

        Edge(int src, int dest, int weight) {
            this.src = src;
            this.dest = dest;
            this.weight = weight;
        }

        @Override
        public int compareTo(Edge other) {
            return Integer.compare(this.weight, other.weight);
        }

        @Override
        public boolean equals(Object o) {
            if (this == o) return true;
```

```
            if (o == null || getClass() != o.getClass()) return false;
            Edge edge = (Edge) o;
            return (src == edge.src && dest == edge.dest) ||
                (src == edge.dest && dest == edge.src);
        }

        @Override
        public int hashCode() {
            return src + dest;
        }
    }

    public MinimumSpanningTree() {
        initializeWindow();
        generateMatrices();
        createAdjacencyList();
        initializeUI();
        printMatrices();
    }

    private void initializeWindow() {
        setTitle("Minimum Spanning Tree - Prim's Algorithm");
        setSize(WINDOW_WIDTH, WINDOW_HEIGHT);
        setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
        setLocationRelativeTo(null);
        setLayout(new BorderLayout());
    }

    private void initializeUI() {
        JPanel controlPanel = createControlPanel();
        JScrollPane infoScrollPane = createInfoPanel();

        add(controlPanel, BorderLayout.NORTH);
        add(new GraphPanel(), BorderLayout.CENTER);
        add(infoScrollPane, BorderLayout.SOUTH);

        updateInfoTextArea();
    }

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

        JButton nextStepButton = new JButton("Next Step");
        JButton resetButton = new JButton("Reset");
        statusLabel = new JLabel("Press 'Next Step' to start Prim's
algorithm");
```

```
        nextStepButton.addActionListener(e -> {
            nextStep();
            repaint();
        });

        resetButton.addActionListener(e -> {
            resetAlgorithm();
            repaint();
        });

        controlPanel.add(nextStepButton);
        controlPanel.add(resetButton);
        controlPanel.add(statusLabel);

        return controlPanel;
    }

    private JScrollPane createInfoPanel() {
        infoTextArea = new JTextArea(10, 40);
        infoTextArea.setEditable(false);
        infoTextArea.setFont(new Font(Font.MONOSPACED, Font.PLAIN, 12));

        return new JScrollPane(infoTextArea);
    }

    private void updateInfoTextArea() {
        StringBuilder sb = new StringBuilder();

        sb.append("Undirected Graph Adjacency Matrix:\n");
        for (int i = 0; i < VERTEX_COUNT; i++) {
            for (int j = 0; j < VERTEX_COUNT; j++) {
                sb.append(undirectedMatrix[i][j]).append(" ");
            }
            sb.append("\n");
        }

        sb.append("\nWeight Matrix:\n");
        for (int i = 0; i < VERTEX_COUNT; i++) {
            for (int j = 0; j < VERTEX_COUNT; j++) {
                sb.append(String.format("%4d ", weightMatrix[i][j]));
            }
            sb.append("\n");
        }

        if (algorithmCompleted) {
```

```
            sb.append("\nMinimum Spanning Tree Edges:\n");
            for (Edge edge : mstEdges) {
                sb.append(String.format("(%d-%d) with weight: %d%n",
                    edge.src + 1, edge.dest + 1, edge.weight));
            }
            sb.append(String.format("%nTotal MST Weight: %d%n",
totalMstWeight));
        }

        infoTextArea.setText(sb.toString());
    }

    private void nextStep() {
        if (currentStep == -1) {
            initializePrimsAlgorithm();
            currentStep = 0;
            statusLabel.setText("Prim's algorithm initialized with
vertex 1");
        } else if (!algorithmCompleted) {
            executeNextPrimStep();
        }
        updateInfoTextArea();
    }

    private void initializePrimsAlgorithm() {
        mstEdges.clear();
        mstVertices.clear();
        mstVertices.add(0);
    }

    private void executeNextPrimStep() {
        if (isAlgorithmComplete()) {
            return;
        }

        Edge minEdge = findMinimumEdge();

        if (minEdge != null) {
            addEdgeToMST(minEdge);
        }
    }

    private boolean isAlgorithmComplete() {
        if (mstVertices.size() == VERTEX_COUNT) {
            algorithmCompleted = true;
            statusLabel.setText("Prim's algorithm completed. MST
```

```
found!");
            calculateTotalWeight();
            return true;
        }
        return false;
    }

    private Edge findMinimumEdge() {
        Edge minEdge = null;
        int minWeight = Integer.MAX_VALUE;

        for (int v : mstVertices) {
            for (VertexConnection connection : adjacencyList.get(v)) {
                if (isValidConnection(connection, minWeight)) {
                    minWeight = connection.weight;
                    minEdge = new Edge(v, connection.vertex,
connection.weight);
                }
            }
        }

        return minEdge;
    }

    private boolean isValidConnection(VertexConnection connection, int
currentMinWeight) {
        return !mstVertices.contains(connection.vertex) &&
            connection.weight < currentMinWeight &&
            connection.weight > 0;
    }

    private void addEdgeToMST(Edge edge) {
        mstEdges.add(edge);
        mstVertices.add(edge.dest);
        statusLabel.setText(String.format("Added edge (%d-%d) with
weight %d",
            edge.src + 1, edge.dest + 1, edge.weight));
        currentStep++;
    }

    private void calculateTotalWeight() {
        totalMstWeight = 0;
        for (Edge edge : mstEdges) {
            totalMstWeight += edge.weight;
        }
    }
```

```
    private void resetAlgorithm() {
        mstEdges.clear();
        mstVertices.clear();
        currentStep = -1;
        algorithmCompleted = false;
        totalMstWeight = 0;
        statusLabel.setText("Press 'Next Step' to start Prim's
algorithm");
        updateInfoTextArea();
    }

    private void createAdjacencyList() {
        for (int i = 0; i < VERTEX_COUNT; i++) {
            adjacencyList.put(i, new ArrayList<>());
        }

        for (int i = 0; i < VERTEX_COUNT; i++) {
            for (int j = 0; j < VERTEX_COUNT; j++) {
                if (weightMatrix[i][j] > 0) {
                    adjacencyList.get(i).add(new VertexConnection(j,
weightMatrix[i][j]));

                    if (i <= j) {
                        allEdges.add(new Edge(i, j,
weightMatrix[i][j]));
                    }
                }
            }
        }
    }

    private void generateMatrices() {
        directedMatrix = generateDirectedMatrix();
        undirectedMatrix = generateUndirectedMatrix(directedMatrix);
        weightMatrix = generateWeightMatrix(undirectedMatrix);
    }

    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 int[][] generateUndirectedMatrix(int[][] directedMatrix) {
        int[][] resultMatrix = new int[VERTEX_COUNT][VERTEX_COUNT];

        for (int i = 0; i < VERTEX_COUNT; i++) {
            for (int j = 0; j < VERTEX_COUNT; j++) {
                if (directedMatrix[i][j] == 1) {
                    resultMatrix[i][j] = 1;
                    resultMatrix[j][i] = 1;
                }
            }
        }

        return resultMatrix;
    }

    private int[][] generateWeightMatrix(int[][] undirMatrix) {
        Random random = new Random(VARIANT_CODE);

        double[][] matrixB = generateMatrixB(random);
        int[][] matrixC = generateMatrixC(matrixB, undirMatrix);
        int[][] matrixD = generateMatrixD(matrixC);
        int[][] matrixH = generateMatrixH(matrixD);
        int[][] matrixTr = generateUpperTriangularMatrix();

        return calculateFinalWeightMatrix(matrixD, matrixH, matrixTr,
matrixC);
    }

    private double[][] generateMatrixB(Random random) {
        double[][] matrixB = new double[VERTEX_COUNT][VERTEX_COUNT];
        for (int i = 0; i < VERTEX_COUNT; i++) {
            for (int j = 0; j < VERTEX_COUNT; j++) {
                matrixB[i][j] = random.nextDouble(0.0, 2.0);
            }
        }
        return matrixB;
    }

    private int[][] generateMatrixC(double[][] matrixB, int[][]
undirMatrix) {
```

```
        int[][] matrixC = new int[VERTEX_COUNT][VERTEX_COUNT];
        for (int i = 0; i < VERTEX_COUNT; i++) {
            for (int j = 0; j < VERTEX_COUNT; j++) {
                matrixC[i][j] = (int) Math.ceil(matrixB[i][j] * 100 *
undirMatrix[i][j]);
            }
        }
        return matrixC;
    }

    private int[][] generateMatrixD(int[][] matrixC) {
        int[][] matrixD = new int[VERTEX_COUNT][VERTEX_COUNT];
        for (int i = 0; i < VERTEX_COUNT; i++) {
            for (int j = 0; j < VERTEX_COUNT; j++) {
                matrixD[i][j] = matrixC[i][j] > 0 ? 1 : 0;
            }
        }
        return matrixD;
    }

    private int[][] generateMatrixH(int[][] matrixD) {
        int[][] matrixH = new int[VERTEX_COUNT][VERTEX_COUNT];
        for (int i = 0; i < VERTEX_COUNT; i++) {
            for (int j = 0; j < VERTEX_COUNT; j++) {
                matrixH[i][j] = matrixD[i][j] != matrixD[j][i] ? 1 : 0;
            }
        }
        return matrixH;
    }

    private int[][] generateUpperTriangularMatrix() {
        int[][] matrixTr = new int[VERTEX_COUNT][VERTEX_COUNT];
        for (int i = 0; i < VERTEX_COUNT; i++) {
            for (int j = 0; j < VERTEX_COUNT; j++) {
                matrixTr[i][j] = i < j ? 1 : 0;
            }
        }
        return matrixTr;
    }

    private int[][] calculateFinalWeightMatrix(int[][] matrixD, int[][]
matrixH,
        int[][] matrixTr, int[][] matrixC) {
        int[][] generatedWeightMatrix = new
int[VERTEX_COUNT][VERTEX_COUNT];
        for (int i = 0; i < VERTEX_COUNT; i++) {
```

```
            for (int j = 0; j < VERTEX_COUNT; j++) {
                int weight = (matrixD[i][j] + matrixH[i][j] *
matrixTr[i][j]) * matrixC[i][j];
                generatedWeightMatrix[i][j] = weight;
                generatedWeightMatrix[j][i] = weight;
            }
        }
        return generatedWeightMatrix;
    }

    private void printMatrices() {
        System.out.println("Directed Graph Adjacency Matrix:");
        printMatrix(directedMatrix);
        System.out.println("\nUndirected Graph Adjacency Matrix:");
        printMatrix(undirectedMatrix);
        System.out.println("\nWeight Matrix:");
        printMatrix(weightMatrix);
    }

    private void printMatrix(int[][] matrix) {
        for (int[] row : matrix) {
            for (int cell : row) {
                System.out.printf("%d ", cell);
            }
            System.out.println();
        }
    }

    class GraphPanel extends JPanel {

        public static final String FONT_NAME = "Arial";

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

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

            drawGraph(g2d, 0, 0, panelWidth / 2, panelHeight);
            drawMST(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(FONT_NAME, Font.BOLD, 20));
            g2d.drawString("Weighted Undirected Graph", 100, 30);
            g2d.drawString("Minimum Spanning Tree", panelWidth / 2 +
100, 30);
        }

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

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

            drawWeightedEdges(g2d, vertexPositions, false, allEdges);
            drawVertices(g2d, vertexPositions);
        }

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

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

            drawWeightedEdges(g2d, vertexPositions, true, mstEdges);
            drawVertices(g2d, vertexPositions);
        }

        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 drawWeightedEdges(Graphics2D g2d, Point[]
vertexPositions,
            boolean isMST, List<Edge> edgesToDraw) {

            setupEdgeDrawingStyle(g2d, isMST);

            for (Edge edge : edgesToDraw) {
                if (shouldSkipEdge(edge, isMST)) {
                    continue;
                }

                drawEdge(g2d, edge, vertexPositions);
            }
        }

        private void setupEdgeDrawingStyle(Graphics2D g2d, boolean
isMST) {
            Color edgeColor = isMST ? Color.RED : Color.BLACK;
            g2d.setColor(edgeColor);
            g2d.setStroke(new BasicStroke(isMST ? 2.5f : 1.5f));
        }

        private boolean shouldSkipEdge(Edge edge, boolean isMST) {
            return isMST && !mstEdges.contains(edge);
        }

        private void drawEdge(Graphics2D g2d, Edge edge, Point[]
vertexPositions) {
            int i = edge.src;
            int j = edge.dest;
```

```
            int weight = edge.weight;

            Point from = vertexPositions[i];
            Point to = vertexPositions[j];

            if (i == j) {
                drawSelfLoop(g2d, from, i);
            } else if (shouldDrawCurvedEdge(i, j, from, to,
vertexPositions[0])) {
                drawCurvedEdgeWithWeight(g2d, from, to, weight);
            } else {
                drawStraightEdgeWithWeight(g2d, from, to, weight);
            }
        }

        private boolean shouldDrawCurvedEdge(int i, int j, Point from,
Point to, Point center) {
            return i != 0 && j != 0 && linePassesThroughCenter(from, to,
center);
        }

        private void drawVertex(Graphics2D g2d, Point position, int
index) {
            boolean inMST = mstVertices.contains(index - 1);

            g2d.setColor(inMST ? Color.YELLOW : Color.WHITE);
            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 drawVertices(Graphics2D g2d, Point[]
vertexPositions) {
            for (int i = 0; i < VERTEX_COUNT; i++) {
                drawVertex(g2d, vertexPositions[i], i + 1);
            }
        }
```

```
        private void drawVertexLabel(Graphics2D g2d, Point position, int
index) {
            g2d.setColor(Color.BLACK);
            g2d.setFont(new Font(FONT_NAME, 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 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 drawCurvedEdgeWithWeight(Graphics2D g2d, Point
from, Point to, int weight) {
            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;

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

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

            drawWeightWithBackground(g2d, weight, controlX - 10,
controlY);
        }

        private void drawStraightEdgeWithWeight(Graphics2D g2d, Point
from, Point to, int weight) {
            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.setColor(Color.BLACK);
            g2d.draw(new Line2D.Double(startX, startY, endX, endY));
```

```
            int midX = (startX + endX) / 2;
            int midY = (startY + endY) / 2;

            drawWeightWithBackground(g2d, weight, midX, midY);
        }

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

            int loopSize = VERTEX_RADIUS * 3 - 10;
            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.setColor(Color.BLACK);
            g2d.drawOval(loopX, loopY, loopSize, loopSize);

            int weight = weightMatrix[vertexIndex][vertexIndex];
            drawWeightWithBackground(g2d, weight, loopX + loopSize,
loopY + loopSize - 20);
        }

        private void drawWeightWithBackground(Graphics2D g2d, int
weight, int x, int y) {
            String weightStr = String.valueOf(weight);
            Font weightFont = new Font(FONT_NAME, Font.BOLD, 14);
            g2d.setFont(weightFont);

            FontMetrics metrics = g2d.getFontMetrics(weightFont);
            int textWidth = metrics.stringWidth(weightStr);
            int textHeight = metrics.getHeight();

            int padding = 4;
            Rectangle textBounds = calculateTextBounds(x, y, textWidth,
textHeight, metrics, padding);

            drawTextBackground(g2d, textBounds);

            g2d.setColor(Color.BLUE);
```

```
            g2d.drawString(
                weightStr,
                x - textWidth / 2,
                y + metrics.getAscent() / 2
            );
        }

        private Rectangle calculateTextBounds(int x, int y, int
textWidth, int textHeight,
                                              FontMetrics metrics, int
padding) {
            return new Rectangle(
                x - textWidth / 2 - padding,
                y - textHeight / 2 + metrics.getDescent() - padding,
                textWidth + padding * 2,
                textHeight + padding
            );
        }

        private void drawTextBackground(Graphics2D g2d, Rectangle
bounds) {
            g2d.setColor(new Color(255, 255, 255, 220));
            g2d.fillRoundRect(
                bounds.x, bounds.y, bounds.width, bounds.height, 8, 8
            );

            g2d.setColor(new Color(200, 200, 200));
            g2d.drawRoundRect(
                bounds.x, bounds.y, bounds.width, bounds.height, 8, 8
            );
        }

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

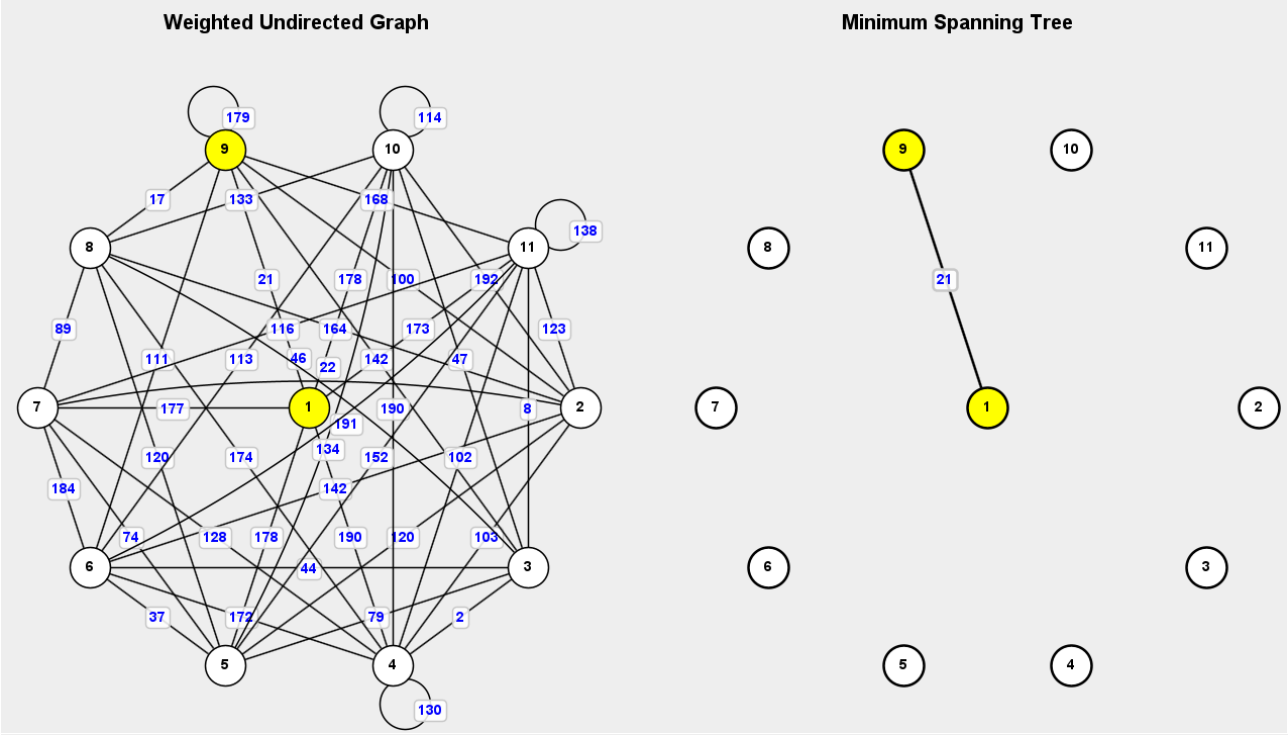
Згенерована матриця суміжності ненапрямленого графа

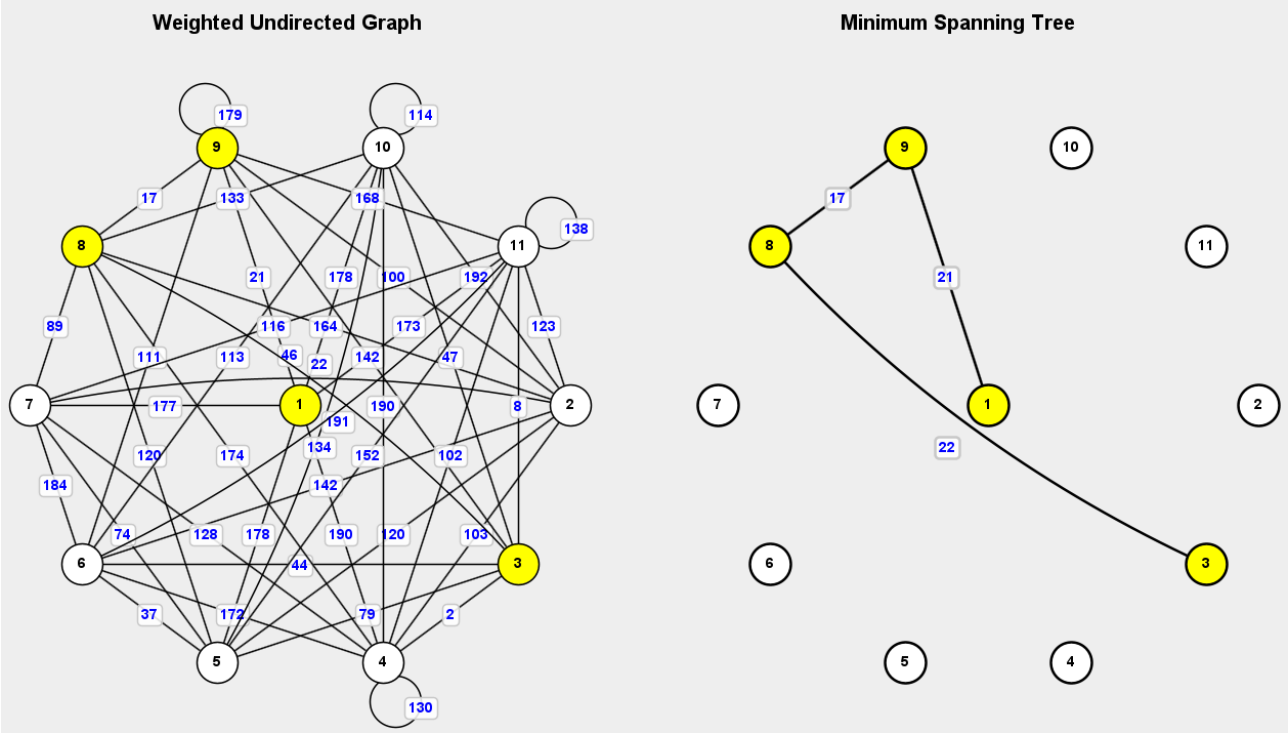
```
0 0 0 1 1 0 1 0 1 1 1
0 0 0 1 1 1 1 1 1 1 1
0 0 0 1 1 1 0 1 1 1 1
1 1 1 1 0 1 1 1 0 1 1
1 1 1 0 0 1 1 1 0 1 1
0 1 1 1 1 0 1 0 1 1 1
1 1 0 1 1 1 0 1 0 0 1
0 1 1 1 1 0 1 0 1 1 0
1 1 1 0 0 1 0 1 1 0 1
1 1 1 1 1 1 0 1 0 1 0
1 1 1 1 1 1 1 0 1 0 1
```

Згенерована матриця ваг графа

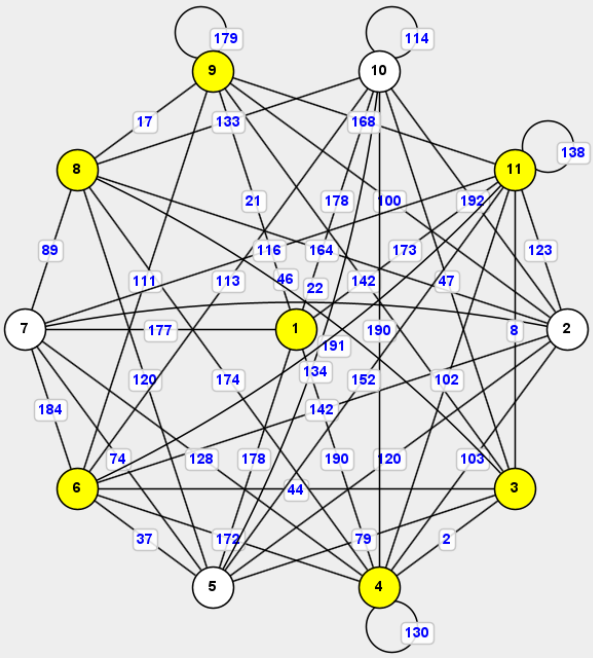
```
  0   0   0 190 178   0 177   0  21 178 173
  0   0   0 103 120 142  46 164 100 192 123
  0   0   0   2  79  44   0  22 142  47   8
190 103   2 130   0 172 128 174   0 190 102
178 120  79   0   0  37  74 120   0 191 152
  0 142  44 172  37   0 184   0 111 113 134
177  46   0 128  74 184   0  89   0   0 116
  0 164  22 174 120   0  89   0  17 133   0
 21 100 142   0   0 111   0  17 179   0 168
178 192  47 190 191 113   0 133   0 114   0
173 123   8 102 152 134 116   0 168   0 138
```

Скриншоти зображення графа та його мінімального кістяка

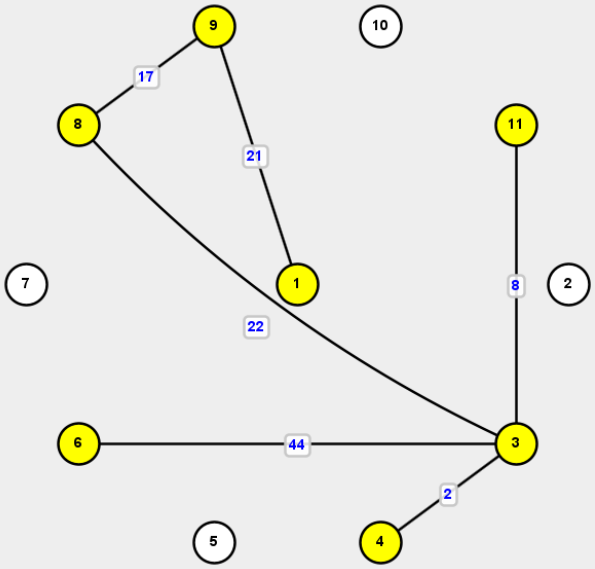




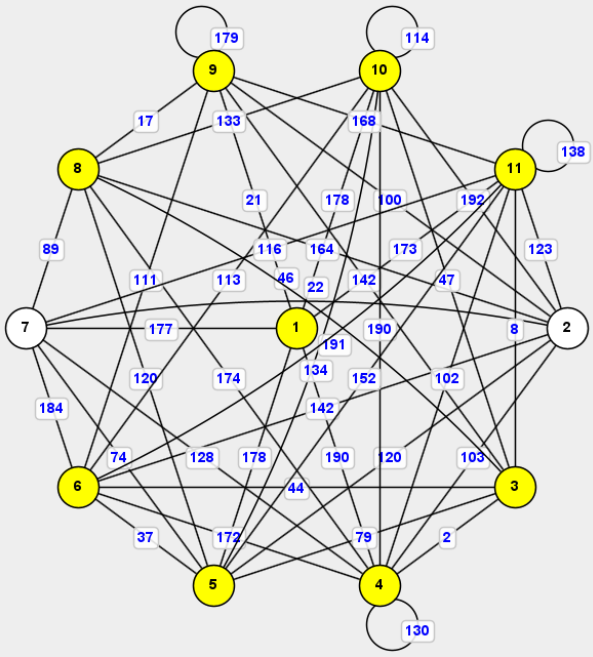
Weighted Undirected Graph



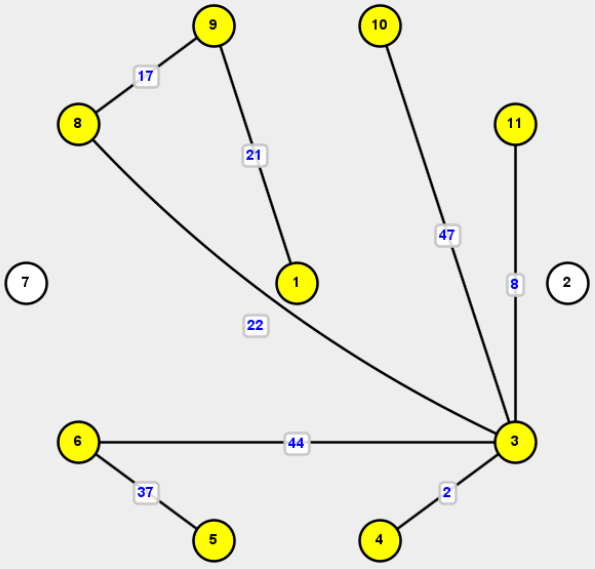
Minimum Spanning Tree

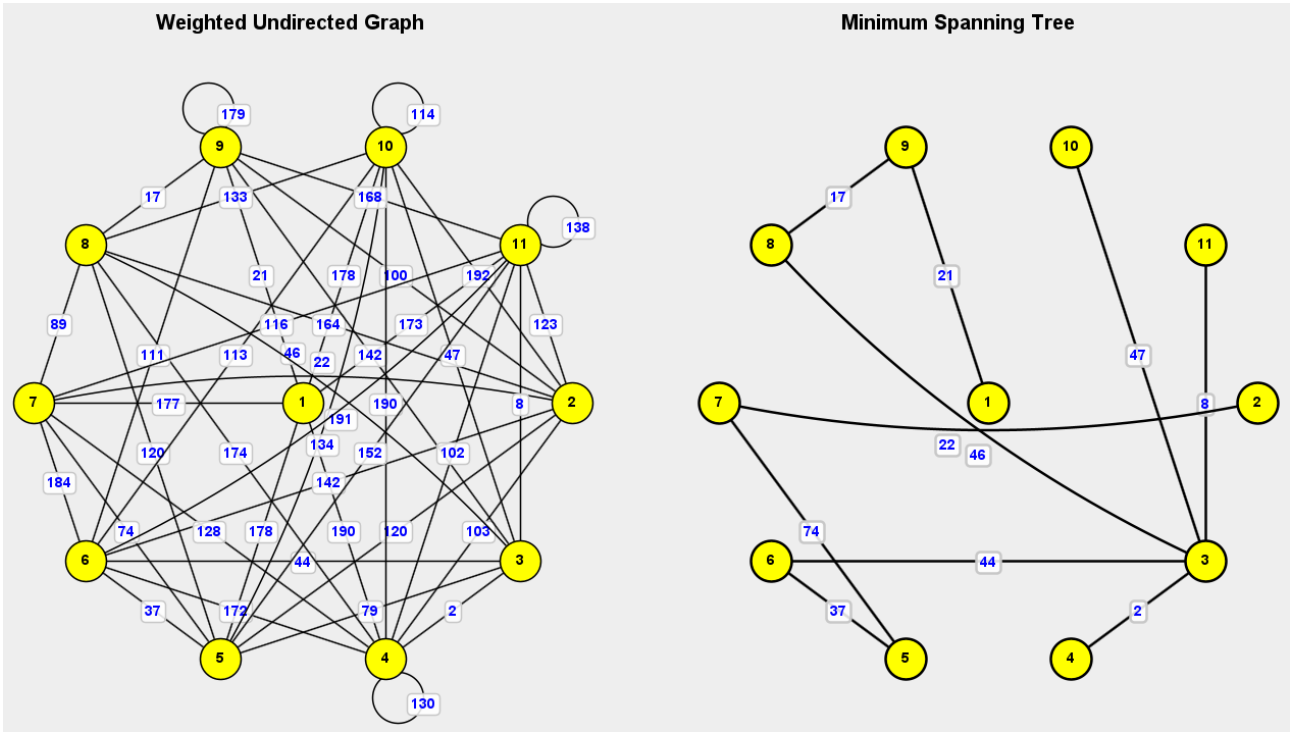


Weighted Undirected Graph



Minimum Spanning Tree




Weighted Undirected Graph
Minimum Spanning Tree
9
10
11
8
7
1
2
6
3
5
4
179
114
17
133
168
138
21
178
100
192
89
116
164
173
123
111
113
46
22
142
47
177
190
8
191
120
174
134
152
102
184
142
74
128
178
190
120
103
44
37
172
79
2
130
9
10
17
8
11
21
47
7
1
8
2
22
46
74
6
44
3
37
2
5
4

Сума ваг ребер знайденого мінімального кістяка

Minimum Spanning Tree Edges:
(1-9) with weight: 21
(9-8) with weight: 17
(8-3) with weight: 22
(3-4) with weight: 2
(3-11) with weight: 8
(3-6) with weight: 44
(6-5) with weight: 37
(3-10) with weight: 47
(5-7) with weight: 74
(7-2) with weight: 46

Total MST Weight: 318

Висновок

Під час виконання лабораторної роботи № 6 я засвоїв теоретичні основи та набув практичних навичок розв’язання задачі знаходження мінімального кістяка для зваженого ненапрямленого графа. Зокрема, я детально вивчив та застосував алгоритм Пріма. Я поглибив свої знання щодо принципів роботи цього алгоритму та його використання для знаходження дерева, що з'єднує всі вершини графа з мінімально можливою сумарною вагою ребер.

Я набув практичного досвіду реалізації алгоритму Пріма для знаходження мінімального кістяка з використанням динамічних списків для представлення графа. За допомогою мови програмування Java та бібліотеки Swing, я реалізував генерацію графа на основі обчислених матриць суміжності та ваг, покрокову візуалізацію процесу побудови мінімального кістяка. Ця візуалізація включала відображення графа, виділення ребер, що додаються до кістяка на кожному кроці, показ їх ваг та побудову самого дерева кістяка.

Виконання цього завдання допомогло мені краще зрозуміти концепцію мінімального кістяка та його практичне значення. Практична реалізація алгоритму Пріма та його покрокова візуалізація дозволили наочно побачити логіку його роботи та ефективність у знаходженні оптимальної структури, що з'єднує всі вершини графа з мінімальною сумарною вагою ребер.

Отже, виконання лабораторної роботи № 6 було корисним, дозволило закріпити теоретичні знання про мінімальні кістяки та алгоритми їх знаходження та набути практичних навичок у їх програмній реалізації мовою Java, включаючи генерацію даних, візуалізацію процесу та аналіз отриманих результатів.