# GRAPH-VISUALIZER PROJECT

# Overview

GraphVisualizer is a Java Swing-based application that provides a graphical interface for creating, visualizing, and manipulating graphs. Users can add nodes and edges, find paths using Depth-First Search (DFS) and Dijkstra's algorithm, and animate traversal between nodes.

## Features

1. **Node and Edge Management**
   - Add nodes by clicking on the panel when "Enable Add Nodes" is checked.
   - Add edges between nodes by specifying node indices.

2. **Pathfinding Algorithms**
   - Depth-First Search (DFS)
   - Dijkstra's Algorithm

3. **Traversal Animation**
   - Animate traversal along a specified path between nodes.

4. **Undo/Redo Functionality**
   - Undo and redo actions for adding nodes and edges.

5. **Background Image Support**
   - Load a custom background image for the graph panel.

6. **Clearing the Graph**
   - Clear all nodes and edges to start fresh.

## *Usage Instructions*

1. **Adding Nodes**
   - Enable node addition by checking the "Enable Add Nodes" menu item.
   - Click on the graph panel to add nodes.

2. **Adding Edges**
   - Enter the indices of the start and end nodes in the "Edge from" and "Edge to" fields, respectively.
   - Click the "Add Edge" button to add the edge.

3. **Finding Paths**
   - Enter the start and end node indices in the "Start Node" and "End Node" fields, respectively.
   - Select the desired algorithm (DFS or Dijkstra) from the dropdown.
   - Click the "Find Path" button to find and animate the path.

4. **Undo/Redo Actions**
   - Use the "Undo" and "Redo" menu items to undo or redo the last action.

5. **Setting Background Image**
   - Use the "Open Background Image" menu item to load a background image.

6. **Clearing the Graph**
   - Use the "Clear" menu item to clear all nodes and edges.

## Dependencies

- Java Development Kit (JDK) 8 or higher
- Swing framework (part of the JDK)

## Running the Application

Compile and run the application using your preferred Java IDE or command line. The main class is `GraphVisualizer`.

```
javac newpackage/*.java
java newpackage.GraphVisualizer
```

Enjoy using GraphVisualizer to create and explore graph structures!

## CODE:

### 1. GraphPanel Class:

```
package newpackage;

import java.awt.*;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.image.BufferedImage;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
import java.util.PriorityQueue;
import java.util.Stack;
import javax.swing.*;

public class GraphPanel extends JPanel {
    private List<Node> nodes;
    private List<Edge> edges;
    private BufferedImage backgroundImage;
    private boolean addNodesEnabled;
    private javax.swing.Timer traversalTimer;
    private Point traversalPoint;
    private List<Integer> traversalPath;
    private int traversalIndex;
    private double traversalProgress;
    private Stack<Action> undoStack;
    private Stack<Action> redoStack;

    public GraphPanel() {
        this.nodes = new ArrayList<>();
        this.edges = new ArrayList<>();
        this.traversalPoint = null;
        this.traversalPath = new ArrayList<>();
        this.undoStack = new Stack<>();
        this.redoStack = new Stack<>();
        setPreferredSize(new Dimension(600, 400));

        addMouseListener(new MouseAdapter() {
            @Override
            public void mouseClicked(MouseEvent e) {
                if (addNodesEnabled) {
                    Node newNode = new Node(e.getX(), e.getY(),
String.valueOf(nodes.size() + 1));
                    addNode(newNode);
                    undoStack.push(new AddNodeAction(newNode));
                    redoStack.clear();
                    repaint();
                }
            }
        });
    }

    public void startTraversal() {
        if (traversalPath.size() < 2) {
            JOptionPane.showMessageDialog(null, "Traversal path must contain
at least two nodes.");
            return;
        }

        traversalIndex = 0;
        traversalProgress = 0.0;
        traversalPoint = null;

        if (traversalTimer != null && traversalTimer.isRunning()) {
            traversalTimer.stop();
        }
```

```
        traversalTimer = new javax.swing.Timer(30, e -> {
            traversalProgress += 0.02;
            if (traversalProgress >= 1.0) {
                traversalProgress = 0.0;
                traversalIndex = (traversalIndex + 1) % (traversalPath.size() - 1);
            }

            int fromIndex = traversalPath.get(traversalIndex);
            int toIndex = traversalPath.get(traversalIndex + 1);
            if (isEdgeExist(fromIndex, toIndex)) {
                Node traversalStart = nodes.get(fromIndex);
                Node traversalEnd = nodes.get(toIndex);

                int x = (int) (traversalStart.x + traversalProgress * (traversalEnd.x -
traversalStart.x));
                int y = (int) (traversalStart.y + traversalProgress * (traversalEnd.y -
traversalStart.y));
                traversalPoint = new Point(x, y);
                repaint(); // Repaint the panel to show updated traversal point
            } else {
                traversalTimer.stop();
                JOptionPane.showMessageDialog(null, "Edge does not exist
between selected nodes.");
            }
        });

        traversalTimer.start();
    }

    private boolean isEdgeExist(int fromIndex, int toIndex) {
        for (Edge edge : edges) {
            if ((edge.from == nodes.get(fromIndex) && edge.to ==
nodes.get(toIndex)) ||
                    (edge.from == nodes.get(toIndex) && edge.to ==
nodes.get(fromIndex))) {
                return true;
            }
        }
        return false;
    }

    public void addNode(Node node) {
        nodes.add(node);
        repaint();
    }

    public void removeNode(Node node) {
        nodes.remove(node);
        repaint();
    }

    public void addEdge(Edge edge) {
        edges.add(edge);
        repaint();
    }

    public void removeEdge(Edge edge) {
        edges.remove(edge);
        repaint();
    }

    public void addEdgeByIndices(int fromIndex, int toIndex) {
        if (fromIndex >= 0 && fromIndex < nodes.size() && toIndex >= 0 &&
toIndex < nodes.size()) {
            Edge newEdge = new Edge(nodes.get(fromIndex),
nodes.get(toIndex));
            addEdge(newEdge);
```

```
            undoStack.push(new AddEdgeAction(newEdge));
            redoStack.clear();
        } else {
            JOptionPane.showMessageDialog(null, "Node indices out of
bounds.");
        }
    }

    public void setBackgroundImage(BufferedImage image) {
        this.backgroundImage = image;
        repaint();
    }

    public void setAddNodesEnabled(boolean enabled) {
        this.addNodesEnabled = enabled;
    }

    public void clear() {
        nodes.clear();
        edges.clear();
        backgroundImage = null;
        traversalPoint = null;
        traversalPath.clear();
        traversalProgress = 0.0;
        if (traversalTimer != null && traversalTimer.isRunning()) {
            traversalTimer.stop();
        }
        undoStack.clear();
        redoStack.clear();
        repaint();
    }

    // Action interface and concrete actions for undo/redo
    private interface Action {
        void undo();
        void redo();
    }

    private class AddNodeAction implements Action {
        private Node node;

        AddNodeAction(Node node) {
            this.node = node;
        }

        @Override
        public void undo() {
            removeNode(node);
        }

        @Override
        public void redo() {
            addNode(node);
        }
    }

    private class AddEdgeAction implements Action {
        private Edge edge;

        AddEdgeAction(Edge edge) {
            this.edge = edge;
        }

        @Override
        public void undo() {
            removeEdge(edge);
        }
```

```
            @Override
            public void redo() {
                addEdge(edge);
            }
        }

        public void undo() {
            if (!undoStack.isEmpty()) {
                Action action = undoStack.pop();
                action.undo();
                redoStack.push(action);
                repaint();
            }
        }

        public void redo() {
            if (!redoStack.isEmpty()) {
                Action action = redoStack.pop();
                action.redo();
                undoStack.push(action);
                repaint();
            }
        }

        public void findAndSetPathDFS(int startNodeIndex, int endNodeIndex) {
            if (startNodeIndex < 0 || startNodeIndex >= nodes.size() || endNodeIndex
    < 0 || endNodeIndex >= nodes.size()) {
                JOptionPane.showMessageDialog(null, "Invalid node indices.");
                return;
            }

            Node startNode = nodes.get(startNodeIndex);
            Node endNode = nodes.get(endNodeIndex);

            boolean[] visited = new boolean[nodes.size()];
            List<Integer> path = new ArrayList<>();
            Stack<Integer> stack = new Stack<>();

            stack.push(startNodeIndex);
            visited[startNodeIndex] = true;

            boolean found = false;
            outer:
            while (!stack.isEmpty()) {
                int currentNodeIndex = stack.peek();

                if (currentNodeIndex == endNodeIndex) {
                    found = true;
                    break;
                }

                List<Integer> neighbors = getNeighbors(currentNodeIndex);
                boolean allVisited = true;

                for (int neighbor : neighbors) {
                    if (!visited[neighbor]) {
                        stack.push(neighbor);
                        visited[neighbor] = true;
                        path.add(neighbor);
                        allVisited = false;
                        break;
                    }
                }

                if (allVisited) {
                    stack.pop();
                }
            }

            if (found) {
                path.add(0, startNodeIndex);
                setTraversalPath(path);
            } else {
                JOptionPane.showMessageDialog(null, "Path not found.");
            }
        }

        private List<Integer> getNeighbors(int nodeIndex) {
            List<Integer> neighbors = new ArrayList<>();
            for (int i = 0; i < edges.size(); i++) {
                Edge edge = edges.get(i);
                if (edge.from == nodes.get(nodeIndex)) {
                    neighbors.add(nodes.indexOf(edge.to));
                } else if (edge.to == nodes.get(nodeIndex)) {
                    neighbors.add(nodes.indexOf(edge.from));
                }
            }
            return neighbors;
        }
        public void findAndSetPathDijkstra(int startNodeIndex, int endNodeIndex) {
        if (startNodeIndex < 0 || startNodeIndex >= nodes.size() || endNodeIndex <
    0 || endNodeIndex >= nodes.size()) {
            JOptionPane.showMessageDialog(null, "Invalid node indices.");
            return;
        }

        Node startNode = nodes.get(startNodeIndex);
        Node endNode = nodes.get(endNodeIndex);

        // Initialize distances and predecessors
        double[] distances = new double[nodes.size()];
        int[] predecessors = new int[nodes.size()];
        PriorityQueue<Node> priorityQueue = new
    PriorityQueue<>(Comparator.comparingDouble(n ->
    distances[nodes.indexOf(n)]));

        Arrays.fill(distances, Double.POSITIVE_INFINITY);
        distances[startNodeIndex] = 0;
        predecessors[startNodeIndex] = -1;
        priorityQueue.add(startNode);

        while (!priorityQueue.isEmpty()) {
            Node currentNode = priorityQueue.poll();
            int currentNodeIndex = nodes.indexOf(currentNode);

            if (currentNodeIndex == endNodeIndex) {
                break; // Found the shortest path to the end node
            }

            List<Integer> neighbors = getNeighbors(currentNodeIndex);

            for (int neighborIndex : neighbors) {
                Node neighborNode = nodes.get(neighborIndex);
                double weight = 1; // Assuming all edges have weight = 1

                if (distances[currentNodeIndex] + weight < distances[neighborIndex]) {
                    distances[neighborIndex] = distances[currentNodeIndex] + weight;
                    predecessors[neighborIndex] = currentNodeIndex;
                    priorityQueue.add(neighborNode);
                }
            }
        }

        // Reconstruct the shortest path
        List<Integer> path = new ArrayList<>();
        for (int at = endNodeIndex; at != -1; at = predecessors[at]) {
            path.add(at);
        }
        Collections.reverse(path);

        if (!path.isEmpty() && path.get(0) == startNodeIndex) {
            setTraversalPath(path);
        } else {
            JOptionPane.showMessageDialog(null, "Path not found.");
        }
    }


        private void setTraversalPath(List<Integer> path) {
            this.traversalPath = path;
            startTraversal(); // Start traversal animation
        }

        private static class Node {
            int x, y;
            String label;

            Node(int x, int y, String label) {
                this.x = x;
                this.y = y;
                this.label = label;
            }
        }

        private static class Edge {
            Node from, to;

            Edge(Node from, Node to) {
                this.from = from;
                this.to = to;
            }
        }

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

            if (backgroundImage != null) {
                g2d.drawImage(backgroundImage, 0, 0, getWidth(), getHeight(), this);
            }

            for (Edge edge : edges) {
                g2d.drawLine(edge.from.x, edge.from.y, edge.to.x, edge.to.y);
            }

            for (Node node : nodes) {
                g2d.fillOval(node.x - 5, node.y - 5, 10, 10);
                g2d.drawString(node.label, node.x + 5, node.y - 5);
            }

            if (traversalPoint != null) {
                g2d.setColor(Color.RED);
                g2d.fillOval(traversalPoint.x - 5, traversalPoint.y - 5, 10, 10);
            }
        }

        public static void main(String[] args) {
            SwingUtilities.invokeLater(() -> {
                JFrame frame = new JFrame("Graph Panel");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.getContentPane().add(new GraphPanel());
                frame.pack();
                frame.setVisible(true);
            });
        }
    }
```

## 2. GraphVisualizer Class:

```
package newpackage;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;

public class GraphVisualizer extends JFrame {
    private GraphPanel graphPanel;
    private JTextField edgeStartField;
    private JTextField edgeEndField;
    private JButton addEdgeButton;
    private JCheckBoxMenuItem enableAddNodesMenuItem;
    private JTextField startNodeField;
    private JTextField endNodeField;
    private JButton findPathButton;
    private JMenuItem undoMenuItem;
    private JMenuItem redoMenuItem;
    private JComboBox<String> algorithmComboBox;

    public GraphVisualizer() {
        graphPanel = new GraphPanel();
        add(graphPanel, BorderLayout.CENTER); // Add graphPanel to the center

        setTitle("Graph Visualizer");
        setSize(800, 600);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null);

        JMenuBar menuBar = new JMenuBar();
        setJMenuBar(menuBar);

        JMenu fileMenu = new JMenu("File");
        menuBar.add(fileMenu);

        JMenuItem openBackgroundMenuItem = new JMenuItem("Open Background
Image");
        fileMenu.add(openBackgroundMenuItem);
        openBackgroundMenuItem.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                JFileChooser fileChooser = new JFileChooser();
                int result = fileChooser.showOpenDialog(GraphVisualizer.this);
                if (result == JFileChooser.APPROVE_OPTION) {
                    File selectedFile = fileChooser.getSelectedFile();
                    try {
                        BufferedImage backgroundImage = ImageIO.read(selectedFile);
                        graphPanel.setBackgroundImage(backgroundImage);
                    } catch (IOException ex) {
                        JOptionPane.showMessageDialog(GraphVisualizer.this, "Error loading
image: " + ex.getMessage());
                    }
                }
            }
        });

        JMenuItem clearMenuItem = new JMenuItem("Clear");
        fileMenu.add(clearMenuItem);
        clearMenuItem.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                graphPanel.clear();
            }
        });

        JMenu editMenu = new JMenu("Edit");
        menuBar.add(editMenu);

        enableAddNodesMenuItem = new JCheckBoxMenuItem("Enable Add Nodes");
        editMenu.add(enableAddNodesMenuItem);
        enableAddNodesMenuItem.addItemListener(new ItemListener() {
            @Override
            public void itemStateChanged(ItemEvent e) {

graphPanel.setAddNodesEnabled(enableAddNodesMenuItem.isSelected());
            }
        });

        undoMenuItem = new JMenuItem("Undo");
        editMenu.add(undoMenuItem);
        undoMenuItem.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                graphPanel.undo();
            }
        });

        redoMenuItem = new JMenuItem("Redo");
        editMenu.add(redoMenuItem);
        redoMenuItem.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                graphPanel.redo();
            }
        });

        JPanel controlPanel = new JPanel();
        controlPanel.setLayout(new FlowLayout(FlowLayout.RIGHT)); // Align
components to the right

        algorithmComboBox = new JComboBox<>(new String[]{"DFS", "Dijkstra"});
        controlPanel.add(new JLabel("Algorithm:"));
        controlPanel.add(algorithmComboBox);

        edgeStartField = new JTextField(5);
        edgeEndField = new JTextField(5);
        addEdgeButton = new JButton("Add Edge");

        controlPanel.add(new JLabel("Edge from:"));
        controlPanel.add(edgeStartField);
        controlPanel.add(new JLabel("Edge to:"));
        controlPanel.add(edgeEndField);
        controlPanel.add(addEdgeButton);

        addEdgeButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                try {
                    int fromIndex = Integer.parseInt(edgeStartField.getText()) - 1;
                    int toIndex = Integer.parseInt(edgeEndField.getText()) - 1;
                    graphPanel.addEdgeByIndices(fromIndex, toIndex);
                } catch (NumberFormatException ex) {
                    JOptionPane.showMessageDialog(GraphVisualizer.this, "Please enter
valid node indices.");
                }
            }
        });

        startNodeField = new JTextField(5);
        endNodeField = new JTextField(5);
        findPathButton = new JButton("Find Path");

        controlPanel.add(new JLabel("Start Node:"));
        controlPanel.add(startNodeField);
        controlPanel.add(new JLabel("End Node:"));
        controlPanel.add(endNodeField);
        controlPanel.add(findPathButton);

        findPathButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                try {
                    int startNodeIndex = Integer.parseInt(startNodeField.getText()) - 1;
                    int endNodeIndex = Integer.parseInt(endNodeField.getText()) - 1;
                    if ("DFS".equals(algorithmComboBox.getSelectedItem())) {
                        graphPanel.findAndSetPathDFS(startNodeIndex, endNodeIndex);
                    } else if ("Dijkstra".equals(algorithmComboBox.getSelectedItem())) {
                        graphPanel.findAndSetPathDijkstra(startNodeIndex, endNodeIndex);
                    }
                } catch (NumberFormatException ex) {
                    JOptionPane.showMessageDialog(GraphVisualizer.this, "Please enter
valid node indices.");
                }
            }
        });

        controlPanel.add(new JLabel("Algorithm:"));
        controlPanel.add(algorithmComboBox);
        add(controlPanel, BorderLayout.SOUTH); // Add controlPanel to the bottom

        pack(); // Adjust frame size to fit contents
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
            @Override
            public void run() {
                new GraphVisualizer().setVisible(true);
            }
        });
    }
}
```

## 3. NODE CLASS:

```
package newpackage;

public class Edge {
   Node from, to;

   public Edge(Node from, Node to) {
      this.from = from;
      this.to = to;          }
}
```

## 4. EDGE CLASS:

```
package newpackage;

public class Node {
   int x, y;
   String label;

   public Node(int x, int y, String label) {
      this.x = x;
      this.y = y;
      this.label = label;
   }
}
```