МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по учебной практике

Тема: Поиск кратчайших путей. Алгоритм Дейкстры

Студент гр. 2383 _____ Сериков М.

Студент гр. 2303 _____ Мышкин Н.В.

Руководитель _____ Фирсов М.А.

Санкт-Петербург

2024

ЗАДАНИЕ
НА УЧЕБНУЮ ПРАКТИКУ

Студент Сериков М. группы 2383

Студент Мышкин Н.В. группы 2303

Тема практики: Алгоритм Дейкстры поиска кратчайших путей в графе.

Задание на практику:

Командная итеративная разработка визуализатора алгоритма на Java с графическим интерфейсом.

Алгоритм: Алгоритм Дейкстры поиска кратчайших путей в графе.

Сроки прохождения практики: 26.06.2024 – 09.07.2024

Дата сдачи отчета: 07.07.2024

Дата защиты отчета: 08.07.2024

Студент _____ Сериков М.

Студент _____ Мышкин Н.В.

Руководитель _____ Фирсов М.А.

АННОТАЦИЯ

Цель практики — создание программы с поддержкой графического интерфейса для нахождения кратчайшего пути с помощью алгоритма Дейкстры на графе с неторицательными весами ребер. Перед выполнением основного задания был составлен план разработки и спецификация программы согласно которым производилась работа.

SUMMARY

The goal of the practice is to create a program with graphical interface support to find the shortest path using Dijkstra's algorithm on a graph with non-negative edge weights. Before completing the main task, a development plan and program specification were drawn up according to which the work was carried out.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Главная цель практической работы — реализация графического представления работы алгоритма Дейкстры поиска кратчайших путей в графе. Для достижения поставленной цели необходимо реализовать рассматриваемый алгоритм, пользовательский интерфейс и визуализировать работу алгоритма, после чего произвести тестирование всех компонент проекта.

1. ТРЕБОВАНИЯ К ПРОГРАММЕ

1.1. Исходные Требования к программе

1.1.1. Требования к вводу исходных данных

На вход программе должен подаваться неотрицательно взвешенный неориентированный граф и исходная вершина. Название вершин состоят из одного символа латинского алфавита. Ввод начальных данных осуществляется двумя возможностями в зависимости от выбора пользователя: непосредственно в рабочей пространстве программы или посредством файла формата txt с данными о графе в следующем формате:

количество вершин

количество ребер

Название вершины координата X, координата Y

Вершина начала ребра, вершина конца ребра, вес ребра

Пример:

2

1

A 50 50

B 100 100

A B 10

1.1.2. Требования к визуализации

Алгоритм в ходе работы сохраняет промежуточные решения в виде списка по которому будет пошаговая визуализация с контролем шагов со стороны пользователя. На каждом шагу вывод одного из следующих этапов работы алгоритма:

Проверка соседей текущей вершины: Для каждого соседа проверяется нахождение нового более оптимального пути. (цветом выделяется проверяемая вершина и ребро до соседней вершины).

Обновление данных соседней вершины: Если найден более оптимальный путь, то обновляется метка стоимости пути. (изменение цвета вершины)

Переход к следующей вершине: Из очереди выбирается новая вершина в качестве текущей с наименьшим весом (отрисовка пути с нуля путем перекрашивания соответствующих ребер).

На каждом шаге алгоритма сохраняется текущее состояние алгоритма и очередь вершин.

1.1.3. Требования к интерфейсу

Эскиз интерфейса представлен на рисунке 1.

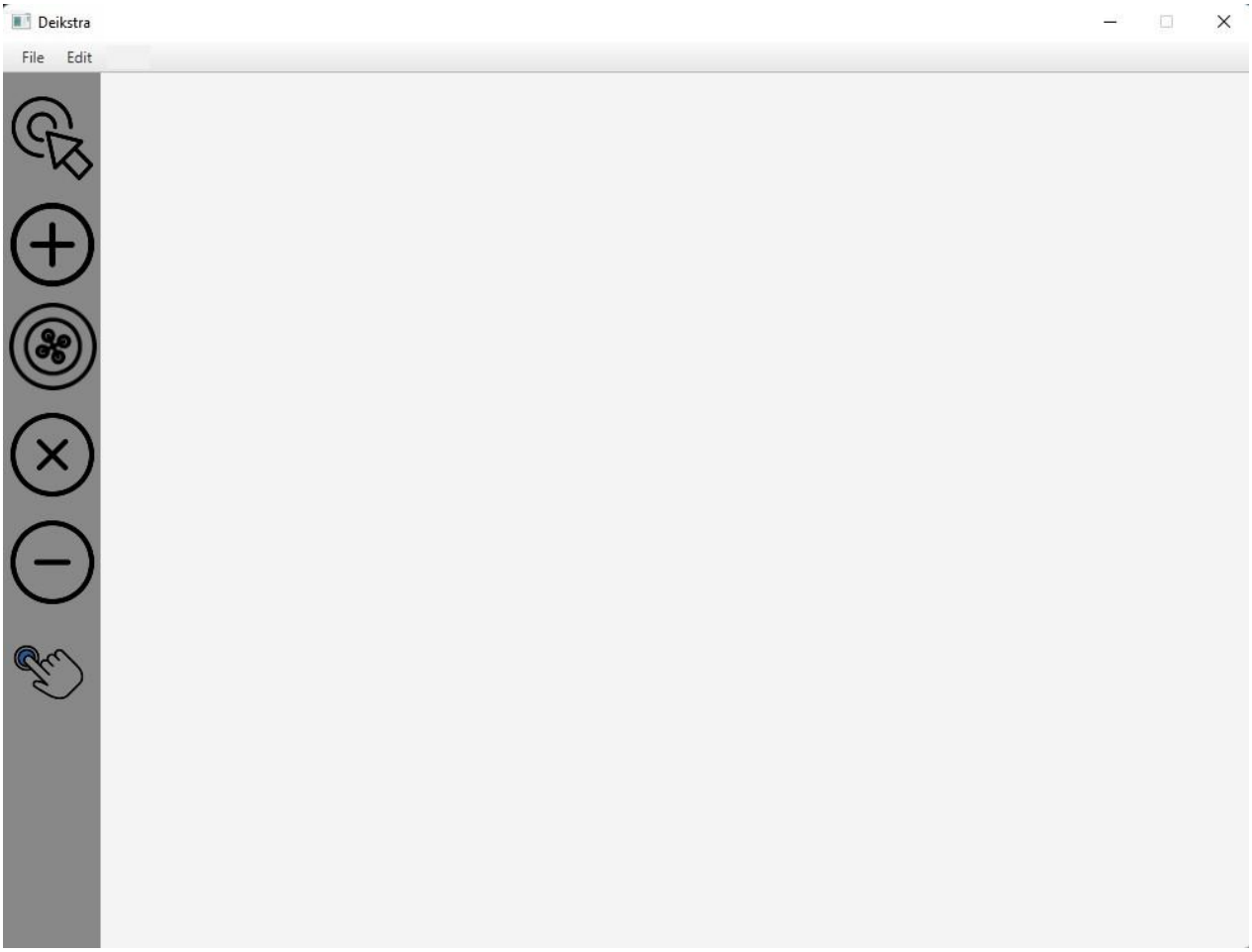

Рисунок 1 — интерфейс программы

В верхней панели программы находится меню с вкладками File, Edit.

Вкладка File: содержит в себе инструменты для загрузки графа из файла, кнопку выхода из программы

Вкладка Edit: содержит в себе инструменты для перехода к окну визуализации

Основная панель инструментов содержит в себе кнопки для построения графа, рассмотрим все кнопки.

Рисунок 2 — кнопка перемещения вершин

Кнопка на рис.2 необходима для перемещения вершин графа, пока активен режим каждую вершину можно перетащить посредством мыши.

Рисунок 3 — кнопка добавления вершин

Кнопка на рис.3 необходима для создания вершин графа, пока активен режим нажатием ЛКМ можно создавать вершины.

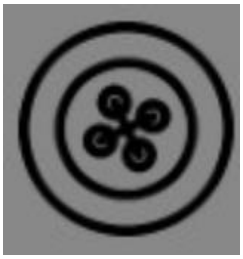

Рисунок 4 — кнопка добавления ребра

Кнопка на рис.4 необходима для добавления ребер графа, пока активен режим выделение двух вершин создает ребро.

Рисунок 5 — кнопка удаления вершин

Кнопка на рис.5 необходима для удаления вершин и ребер графа, пока активен режим выделение вершины или ребра удаляет его.

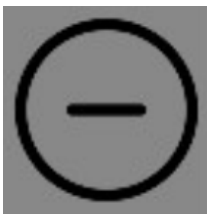

Рисунок 6 — кнопка очистки графа

Кнопка на рис.6 необходима для очистки графа, она очищает рабочую область.

Рисунок 7 — кнопка выбора начальной вершины

Кнопка на рис.7 необходима для выбора начальной вершины для работы алгоритма, она выделяет одну вершину меняя её цвет.

Эскиз окна визуализации работы алгоритма представлен на рисунке 8

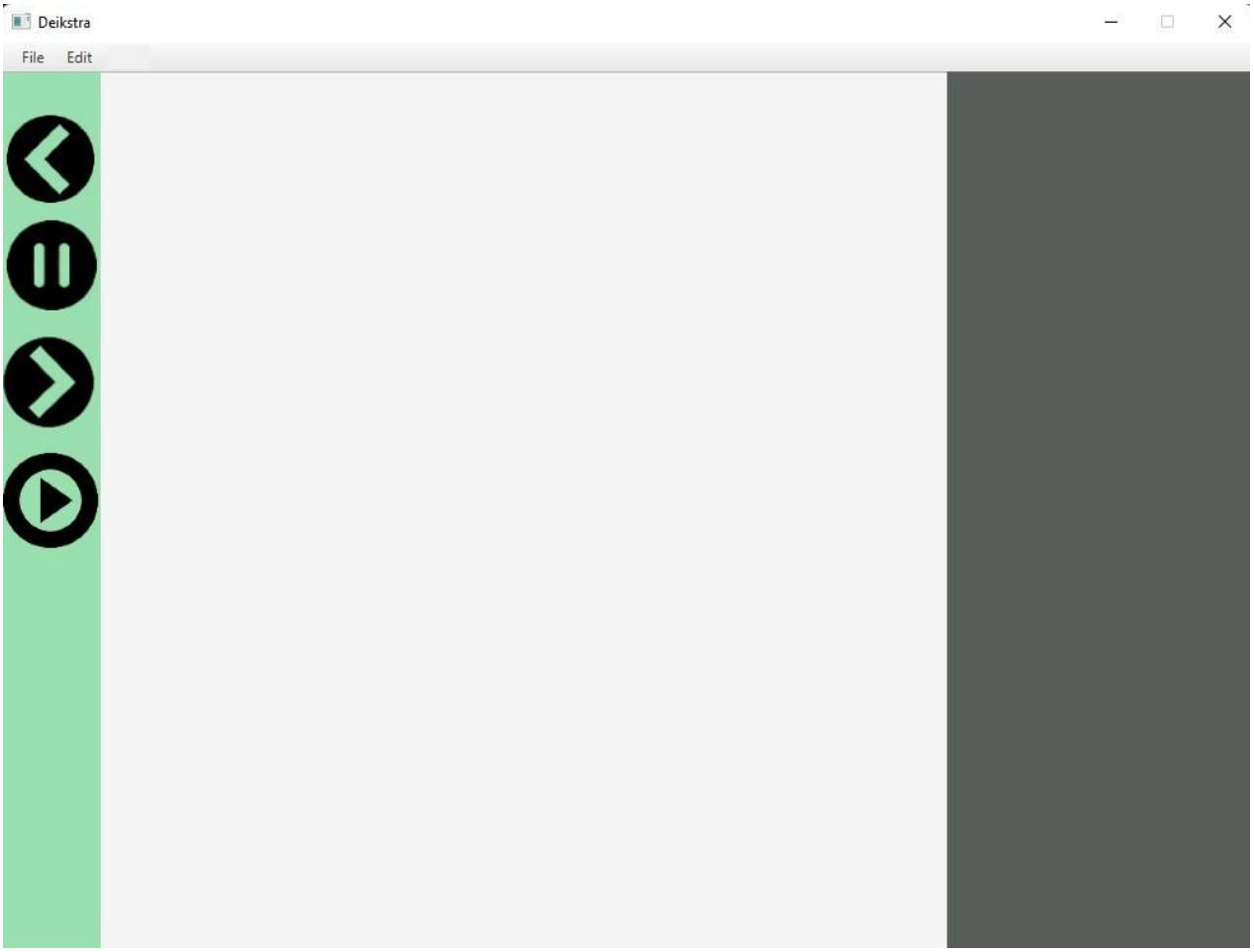

Рисунок 8 — окно визуализации программы

В данном окне представлены инструменты для контроля визуализации работы алгоритма, в случае если выбрана вершина старта алгоритма, то кнопки становятся активными, для начала запуска визуализации алгоритма используется нижняя кнопка, также имеются кнопки паузы и переключения между следующим и предыдущим шагами визуализации. Кнопка начала визуализации запускает автоматическое воспроизведение шагов с небольшой задержкой, при нажатии паузы можно перемещаться между шагами работы алгоритма. Справа от рабочей области находится поле лога работы алгоритма.

1.1.4. Требования к построению графа

Для построения будут использоваться кнопки:

Режим перетаскивания вершин — нажимая ЛКМ и удерживая курсор на вершине возможно её перемещение внутри рабочей области программы (перемещение вершины вслед за курсором)

Режим добавления вершины — нажатие ЛКМ по свободному пространству создась вершину, которой можно будет присвоить название.

Режим добавления ребра — Необходимо выбрать пару вершин, выделение будет демонстрироваться как изменение цвета вершины, после выделения второй вершины будет построено ребро с возможностью ввода его веса, являющимся неотрицательным числом.

Режим удаления объекта — выделения ЛКМ вершины или ребра удаляет его из графа.

Кнопка очистки рабочей области — удаление всего графа.

1.1.5. Требования к работе алгоритма

За основу рассматриваемого алгоритма взят следующий псевдокод:

```
function Dijkstra(Graph, source):
      dist[source] ← 0
      for each vertex v in Graph:
          if v ≠ source:
              dist[v] ← ∞
              prev[v] ← undefined
      Q ← the set of all nodes in Graph
      while Q is not empty:
          u ← vertex in Q with min dist[u]
          remove u from Q
          for each neighbor v of u:
              alt ← dist[u] + length(u, v)
              if alt < dist[v]:
                  dist[v] ← alt
                  prev[v] ← u
      return dist[], prev[]
```

Алгоритм должен уметь работать с неориентированными графами, с ограничением на не более чем 1 ребро между двумя вершинами и без петель, он должен реализовывать алгоритм Дейкстры, поиск кратчайших путей с заданной вершины. Алгоритм должен сохранять промежуточные результаты работы для пошаговой визуализации.

1.1.6. Требования к выводимым пояснениям

В ходе работы алгоритма пояснения должны включать следующие моменты:

1. **Начало алгоритма**:
   - Сообщение о старте алгоритма.
   - Указание начальной вершины, с которой начинается выполнение алгоритма.
2. **Проверка соседей текущей вершины**:
   - Сообщение о проверке соседей текущей вершины.
   - Перечисление всех соседних вершин и текущее расстояние до них.
3. **Обновление данных соседней вершины**:
   - Сообщение о нахождении более оптимального пути к соседней вершине.
   - Обновление метки стоимости пути до соседней вершины.
   - Сообщение о новом значении метки для соседней вершины.
4. **Переход к следующей вершине**:
   - Сообщение о переходе к следующей вершине с наименьшим весом.
   - Указание новой текущей вершины.
5. **Межшаговые действия**:
   - Сохранение текущего состояния алгоритма.
   - Вывод текущей очереди вершин.
   - Информация о промежуточных результатах.
6. **Завершение алгоритма**:
   - Сообщение о завершении работы алгоритма.
   - Вывод итоговых меток для всех вершин (кратчайших расстояний от начальной вершины до всех остальных вершин).
7. **Общие сообщения**:
   - Сообщение об ошибке в случае некорректных входных данных (например, отрицательные веса, несоответствие формата данных).
   - Информационные сообщения о текущем статусе и действиях программы.

Эти пояснения должны отображаться в логовом поле, находящемся справа от рабочей области, и обновляться в реальном времени по мере выполнения шагов алгоритма.

2. ПЛАН РАЗРАБОТКИ И РАСПРЕДЕЛЕНИЕ РОЛЕЙ В БРИГАДЕ

2.1. План разработки

Дата	Этап проекта	Реализованные возможности	Выполнено
27.06.24	Согласование спецификации		
02.06.24	Сдача прототипа	Графический интерфейс программы	
04.07.24	Сдача версии 1	Работа алгоритма без контроля шагов, вывод лога работы алгоритма, возможность построения графа, обработка нажатий кнопок рабочей среды.	
05.07.24	Сдача версии 2	Пошаговый контроль работы алгоритма, вывод пояснений, обработка исключений, улучшенный лог работы алгоритма, полное функционирование всех частей программы.	
06.07.24	Сдача версии 3	Подсказки к инструментам двух окон, сохранение визуализации в формате gif, возобновление анимации с любого шага или паузы.	
07.07.24	Сдача отчёта		
08.07.24	Защита отчёта		

2.2. Распределение ролей в бригаде

Сериков М. - пользовательский интерфейс программы, визуализация алгоритма, связь графической состовляющей программы с основной логикой.

Мышкин Н.В. - реализация алгоритма, структур данных, тестирование

3. ОСОБЕННОСТИ РЕАЛИЗАЦИИ

3.1. Структуры данных

3.1.1. Граф

Структура графа в программе представлена двумя основными компонентами: вершинами и ребрами. Вершина — это объект, содержащий свои координаты и данные, такие как имя. Ребро — это объект, соединяющий две вершины и хранящий вес ребра. Все вершины и ребра хранятся в отдельных списках. Граф предоставляет методы для управления вершинами и ребрами, такие как добавление, удаление и получение списка всех вершин и ребер (см. п. 3.2.1).

3.1.2. Отображение графа

Для отображения графа используются следующие классы:

Вершины: Класс Vertex наследуется от базового графического объекта и дополнительно хранит имя вершины и её координаты. Также, вершина может отображать свою метку расстояния и изменять цвет для различных состояний (рассмотренная, начальная, обычная).

Ребра: Класс Edge представляет собой линию, соединяющую две вершины, и включает в себя вес ребра. Ребра могут изменять цвет в зависимости от состояния (рассмотренное, обычное).

Графическая панель: Класс GraphPanel отвечает за отображение всех вершин и ребер. Он также реализует обработку событий мыши для добавления, перемещения и удаления элементов графа.

3.1.3. Сохранение графа

Для сохранения и загрузки графа используется механизм сериализации. Графы сохраняются в текстовом формате, где сначала записываются вершины,

затем ребра. Формат данных: количество вершин, количество ребер, далее координаты, имена вершин, затем данные о ребрах (начальная и конечная вершины, вес).

3.1.4. Алгоритм и разбиение на шаги

Для работы с алгоритмом Дейкстры был создан класс GraphPanel, который включает в себя методы для реализации данного алгоритма. Алгоритм состоит из следующих шагов:

Инициализация всех вершин и ребер.

Выбор начальной вершины и установка её расстояния до самой себя равным нулю.

Итеративное обновление расстояний до соседних вершин.

Обновление списка посещенных вершин и продолжение до тех пор, пока все вершины не будут посещены.

3.1.5. Интерпретатор шагов

Интерпретатор шагов, реализованный в классе GraphPanel, обеспечивает пошаговое выполнение алгоритма Дейкстры с визуализацией каждого шага. Шаги включают в себя выбор текущей вершины, обновление расстояний до соседних вершин и изменение цветов вершин и ребер для индикации их состояний.

3.2. Основные методы

3.2.1. Граф

Граф предоставляет методы для управления вершинами и ребрами:

addVertex(int x, int y, String name): добавление вершины.

addEdge(Vertex start, Vertex end, int weight): добавление ребра.

removeVertex(Vertex v): удаление вершины.

removeEdge(Edge e): удаление ребра.

getVertices(): получение списка всех вершин.

getEdges(): получение списка всех ребер.

3.2.2. Отображение графа

Основные методы для отображения графа включают:

paintComponent(Graphics g): отрисовка всех вершин и ребер на панели.

setTool(String tool): установка текущего инструмента (добавление вершины, ребра, перемещение и т.д.).

handleMousePressed(MouseEvent e): обработка нажатия мыши для различных инструментов.

handleMouseDragged(MouseEvent e): обработка перетаскивания мыши для перемещения вершин.

3.2.3. Сохранение графа

Для сохранения графа используется метод saveGraph(File file), который сохраняет данные о вершинах и ребрах в указанный файл. Для загрузки графа используется метод loadGraph(File file), который читает данные из файла и восстанавливает граф.

3.2.4. Алгоритм и разбиение на шаги

Класс GraphPanel включает методы для реализации алгоритма Дейкстры:

startDijkstra(JTextArea logTextArea): запуск алгоритма и логирование шагов.

resetGraph(): сброс состояния графа перед началом алгоритма.

paintPathToVertex(Vertex vertex, Map<Vertex, Vertex> prev): отрисовка кратчайшего пути до указанной вершины.

3.2.5. Интерпретатор шагов

Методы интерпретатора шагов включают:

startAutomaticVisualization(): автоматическое воспроизведение шагов алгоритма.

showNextStep(JTextArea logTextArea): выполнение следующего шага.

prevStep(JTextArea logTextArea): возврат к предыдущему шагу.

pauseVisualization(): пауза автоматического воспроизведения.

3.2.6. Интерфейс

Интерфейс программы реализован с использованием Swing и включает в себя основные окна и панели:

Главное окно с меню для загрузки, сохранения графа и переключения между режимами редактирования и визуализации.

Панель инструментов для выбора текущего инструмента.

Графическая панель для отображения и редактирования графа.

Панель управления визуализацией для воспроизведения шагов алгоритма Дейкстры и логирования действий.

4. ТЕСТИРОВАНИЕ

4.1. Тестирование графического интерфейса

Проведено тестирование интерфейса на различных операционных системах Windows, Linux, всех следующих элементов:

1. Режим изменения графа
2. Окно визуализации алгоритма
3. Панель инструментов
4. Загрузка графа из файла
5. Сохранение визуализации алгоритма

При тестировании режима изменения графа проверялось корректная работа всех инструментов: создание вершины, перемещение вершин графа, создание ребра, выбор начальной вершины алгоритма, удаление вершины или ребра, очистка графа. Проверка, что при выборе инструмента кнопка меняет цвет, проверка невозможности одноверменного выбора более чем одного инструмента.

Тестирование окна визуализации алгоритма включает также проверку всех составляющих инструментов: предыдущий, следующий шаг, пауза, старт, возобновление анимации. Если не выбрана начальная вершина проверяется, что панель инструментов заблокирована. Проверка правильной работоспособности программы при проигрывании анимации посредством взаимодействия со всеми доступными во время проигрывания инструментами, при переходе в режим редактирования визуализация останавливается. Также протестированы элементы загрузки графа и сохранения анимации, при загрузке файла с отрицательными весами ребер выводилось соответствующее окно ошибки, при попытках сохранения анимации, до просмотра визуализациии также выводилось сообщение о неправильном пользовании инструментом.

4.2. Тестирование кода алгоритма

При тестировании кода алгоритма использовались Юнит-тесты реализованные при помощи библиотеки JUnit. При тестировании была

проверка работоспособности алгоритма на неориентированных графах, обработаны ситуации когда пути не существует, либо имеются несколько кратчайших путей, тесты охватывают все описанные случаи.

4.3. Тестирование кода графа

При тестировании также используется библиотека JUnit, для проверки правильной работоспособности кода неориентированного графа были произведены тесты на получение ошибки при передаче значения null, на правильную реакцию при передаче нормальных данных и на получение ошибки при других сценариях.

ЗАКЛЮЧЕНИЕ

В ходе выполнения задания учебной практики была разработана программа для визуализации и анализа работы алгоритма Дейкстры. Основные этапы работы включали в себя изучение принципов и основ алгоритма Дейкстры, создание графического интерфейса с использованием библиотеки Swing, а также разработку и тестирование функциональности программы.

Была реализована интерактивная графическая оболочка, позволяющая пользователю добавлять вершины и ребра в граф, назначать стартовую вершину и запускать визуализацию работы алгоритма. Программа предоставляет пользователю возможность наблюдать пошаговое выполнение алгоритма.

Кроме того, были разработаны требования к входным данным и обеспечена удобная и интуитивно понятная визуализация. В ходе разработки также были написаны и выполнены юнит-тесты, которые проверили корректность работы алгоритма, интерфейса.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Курс обучения Java: https://stepik.org/course/Java-Базовый-курс-187/syllabus

2. Сайт с иконками для визуализации: https://www.flaticon.com/icon-fonts-most-downloaded

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: Main.java:

```
package org.example.MainProgramm;

import javax.swing.*;

public class Main {
    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> {
            GraphEditor editor = new GraphEditor();
            editor.setVisible(true);
        });
    }
}
```

Название файла: GraphEditor.java:

```
package org.example.MainProgramm;

import com.madgag.gif.fmsware.AnimatedGifEncoder;

import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.image.BufferedImage;
import java.io.File;
import java.net.URL;

public class GraphEditor extends JFrame {
    private JPanel toolPanel;
    private GraphPanel workPanel;
    private GraphPanel visualizationPanel;
    private JPanel mainPanel;
    private CardLayout cardLayout;
    private JButton activeButton;

    private boolean isVisualizationViewed = false;
    private static final Color CUSTOM_LIGHT_GREEN = new Color(144, 238,
144);
    private static final Color ACTIVE_BUTTON_COLOR = Color.YELLOW;

    private JTextArea logTextArea;

    public GraphEditor() {
        setTitle("Deikstra");
        setSize(800, 600);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JMenuBar menuBar = new JMenuBar();
        JMenu fileMenu = new JMenu("File");
        JMenu editMenu = new JMenu("Edit");
        JMenu saveMenu = new JMenu("Save");

        JMenuItem loadGraphItem = new JMenuItem("Load Graph");
```

```
        JMenuItem exitItem = new JMenuItem("Exit");
        fileMenu.add(loadGraphItem);
        fileMenu.add(exitItem);

        JMenuItem visualizeItem = new JMenuItem("Visualize");
        JMenuItem graphEditorItem = new JMenuItem("Graph Editor");
        editMenu.add(visualizeItem);
        editMenu.add(graphEditorItem);

        JMenuItem saveVisualizationItem = new JMenuItem("Save
Visualization");
        saveMenu.add(saveVisualizationItem);

        exitItem.addActionListener(e -> System.exit(0));
        visualizeItem.addActionListener(e -> showVisualizationWindow());
        graphEditorItem.addActionListener(e -> showEditWindow());
        loadGraphItem.addActionListener(e -> loadGraphFromFile());
        saveVisualizationItem.addActionListener(e ->
saveVisualization());

        menuBar.add(fileMenu);
        menuBar.add(editMenu);
        menuBar.add(saveMenu);

        setJMenuBar(menuBar);

        cardLayout = new CardLayout();
        mainPanel = new JPanel(cardLayout);

        toolPanel = new JPanel();
        toolPanel.setLayout(new BoxLayout(toolPanel, BoxLayout.Y_AXIS));
        toolPanel.setBackground(Color.GRAY);
        toolPanel.setPreferredSize(new Dimension(70, getHeight()));

        JButton moveButton = createToolButton("Move",
"/Icons/MoveVertex.png", "Move");
        JButton addVertexButton = createToolButton("Add Vertex",
"/Icons/AddVertex.png", "Add Vertex");
        JButton addEdgeButton = createToolButton("Add Edge",
"/Icons/AddEdge.png", "Add Edge");
        JButton deleteButton = createToolButton("Delete",
"/Icons/DeleteObject.png", "Delete");
        JButton clearButton = createToolButton("Clear",
"/Icons/ClearArea.png", "Clear");
        JButton startVertexButton = createToolButton("Start Vertex",
"/Icons/SetStartVertex.png", "Start Vertex");

        toolPanel.add(moveButton);
        toolPanel.add(addVertexButton);
        toolPanel.add(addEdgeButton);
        toolPanel.add(deleteButton);
        toolPanel.add(clearButton);
        toolPanel.add(startVertexButton);

        workPanel = new GraphPanel();
        clearButton.addActionListener(e -> workPanel.clearArea());
```

```
        JPanel editPanel = new JPanel(new BorderLayout());
        editPanel.add(toolPanel, BorderLayout.WEST);
        editPanel.add(workPanel, BorderLayout.CENTER);
        mainPanel.add(editPanel, "Edit");

        getContentPane().add(mainPanel);
        showEditWindow();
    }

    private JButton createToolButton(String text, String iconPath, String
toolTipText) {
        JButton button = new JButton();
        button.setPreferredSize(new Dimension(50, 50));
        button.setMaximumSize(new Dimension(50, 50));
        button.setAlignmentX(Component.CENTER_ALIGNMENT);
        URL iconURL = getClass().getResource(iconPath);
        if (iconURL != null) {
            ImageIcon icon = new ImageIcon(iconURL);
            button.setIcon(icon);
        }
        button.setToolTipText(toolTipText);
        button.setBackground(Color.GRAY);
        button.setOpaque(true);
        button.setBorderPainted(false);
        button.addActionListener(new ToolButtonListener(text));
        return button;
    }

    private class ToolButtonListener implements ActionListener {
        private String tool;

        public ToolButtonListener(String tool) {
            this.tool = tool;
        }

        @Override
        public void actionPerformed(ActionEvent e) {
            JButton source = (JButton) e.getSource();
            if (activeButton != null) {
                activeButton.setBackground(Color.GRAY);
            }
            activeButton = source;
            activeButton.setBackground(ACTIVE_BUTTON_COLOR);
            System.out.println("Button clicked: " +
source.getToolTipText());
            workPanel.setTool(tool);
        }
    }

    private void showVisualizationWindow() {
        isVisualizationViewed = false;
        visualizationPanel = new GraphPanel(workPanel);
        JPanel visualizationContainer = new JPanel(new BorderLayout());

        JPanel controlPanel = new JPanel();
        controlPanel.setLayout(new BoxLayout(controlPanel,
BoxLayout.Y_AXIS));
```

```
        controlPanel.setBackground(CUSTOM_LIGHT_GREEN);
        controlPanel.setPreferredSize(new Dimension(70, getHeight()));

        JButton playButton = createControlButton("Play",
"/Icons/PlayButton.png", "Start automatic visualization");
        JButton nextStepButton = createControlButton("Next Step",
"/Icons/NextStepButton.png", "Next step");
        JButton prevStepButton = createControlButton("Prev Step",
"/Icons/PrevStepButton.png", "Previous step");
        JButton pauseButton = createControlButton("Pause",
"/Icons/StopButton.png", "Pause visualization");
        JButton resumeButton = createControlButton("Resume",
"/Icons/ResumeButton.png", "Resume automatic visualization");

        controlPanel.add(playButton);
        controlPanel.add(nextStepButton);
        controlPanel.add(prevStepButton);
        controlPanel.add(pauseButton);
        controlPanel.add(resumeButton);

        JPanel logPanel = new JPanel();
        logPanel.setBackground(Color.DARK_GRAY);
        logPanel.setPreferredSize(new Dimension(300, getHeight()));
        logPanel.setLayout(new BoxLayout(logPanel, BoxLayout.Y_AXIS));

        logTextArea = new JTextArea();
        logTextArea.setEditable(false);
        JScrollPane logScrollPane = new JScrollPane(logTextArea);
        logPanel.add(logScrollPane);

        playButton.addActionListener(e ->
startVisualization(logTextArea));
        nextStepButton.addActionListener(e -> nextStep(logTextArea));
        prevStepButton.addActionListener(e -> prevStep(logTextArea));
        pauseButton.addActionListener(e -> pauseVisualization());
        resumeButton.addActionListener(e -> resumeVisualization());

        visualizationContainer.add(controlPanel, BorderLayout.WEST);
        visualizationContainer.add(logPanel, BorderLayout.EAST);
        visualizationContainer.add(visualizationPanel,
BorderLayout.CENTER);

        mainPanel.add(visualizationContainer, "Visualize");

        cardLayout.show(mainPanel, "Visualize");
    }

    private void showEditWindow() {
        cardLayout.show(mainPanel, "Edit");
    }

    private JButton createControlButton(String text, String iconPath,
String toolTipText) {
        JButton button = new JButton();
        button.setPreferredSize(new Dimension(50, 50));
        button.setMaximumSize(new Dimension(50, 50));
```

```
        button.setAlignmentX(Component.CENTER_ALIGNMENT);
        URL iconURL = getClass().getResource(iconPath);
        if (iconURL != null) {
            ImageIcon icon = new ImageIcon(iconURL);
            button.setIcon(icon);
        }
        button.setToolTipText(toolTipText);
        button.setBackground(CUSTOM_LIGHT_GREEN);
        button.setOpaque(true);
        button.setBorderPainted(false);
        button.addActionListener(new ControlButtonListener(text));
        return button;
    }

    private class ControlButtonListener implements ActionListener {
        private String action;

        public ControlButtonListener(String action) {
            this.action = action;
        }

        @Override
        public void actionPerformed(ActionEvent e) {
            JButton source = (JButton) e.getSource();
            System.out.println("Control button clicked: " +
source.getToolTipText());

            switch (action) {
                case "Play":
                    startVisualization(logTextArea);
                    break;
                case "Next Step":
                    nextStep(logTextArea);
                    break;
                case "Prev Step":
                    prevStep(logTextArea);
                    break;
                case "Pause":
                    pauseVisualization();
                    break;
                case "Resume":
                    resumeVisualization();
                    break;
            }
        }
    }

    private void loadGraphFromFile() {
        JFileChooser fileChooser = new JFileChooser();
        int result = fileChooser.showOpenDialog(this);
        if (result == JFileChooser.APPROVE_OPTION) {
            File selectedFile = fileChooser.getSelectedFile();
            workPanel.loadGraph(selectedFile);
        }
    }

    private void saveVisualization() {
```

```
        if (!isVisualizationViewed) {
            JOptionPane.showMessageDialog(this, "Saving is possible only
after viewing the animation.");
            return;
        }

        JFileChooser fileChooser = new JFileChooser();
        fileChooser.setDialogTitle("Save Visualization");
        fileChooser.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);
        int result = fileChooser.showSaveDialog(this);
        if (result == JFileChooser.APPROVE_OPTION) {
            File directory = fileChooser.getSelectedFile();
            saveGraphAsGif(directory);
        }
    }


    private void saveGraphAsGif(File directory) {
        AnimatedGifEncoder encoder = new AnimatedGifEncoder();
        encoder.start(new File(directory,
"visualization.gif").getPath());
        encoder.setDelay(1000);
        encoder.setRepeat(0);

        visualizationPanel.startDijkstra(logTextArea);


        for (int i = 0; i < visualizationPanel.getSteps().size(); i++) {
            visualizationPanel.getSteps().get(i).run();
            encoder.addFrame(getPanelImage(visualizationPanel));
        }

        encoder.finish();
        JOptionPane.showMessageDialog(this, "Visualization saved
successfully.");
    }

    private BufferedImage getPanelImage(JPanel panel) {
        int w = panel.getWidth();
        int h = panel.getHeight();
        BufferedImage image = new BufferedImage(w, h,
BufferedImage.TYPE_INT_RGB);
        Graphics2D g2 = image.createGraphics();
        panel.paint(g2);
        g2.dispose();
        return image;
    }

    private void startVisualization(JTextArea logTextArea) {
        isVisualizationViewed = true;
        visualizationPanel.startDijkstra(logTextArea);
        visualizationPanel.startAutomaticVisualization();
    }

    private void nextStep(JTextArea logTextArea) {
        if (!visualizationPanel.isVisualizationStarted()) {
```

```
            visualizationPanel.startDijkstra(logTextArea);
        }
        isVisualizationViewed = true;
        visualizationPanel.showNextStep(logTextArea);
    }

    private void prevStep(JTextArea logTextArea) {
        isVisualizationViewed = true;
        visualizationPanel.prevStep(logTextArea);
    }

    private void pauseVisualization() {
        visualizationPanel.pauseVisualization();
    }

    private void resumeVisualization() {
        visualizationPanel.startAutomaticVisualization();
    }

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

Название файла: GraphPanel.java:

```
package org.example.MainProgramm;

import javax.swing.*;
import javax.swing.Timer;
import java.awt.*;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.io.File;
import java.io.FileNotFoundException;
import java.util.*;
import java.util.List;

public class GraphPanel extends JPanel {
    private List<Vertex> vertices;
    private List<Edge> edges;
    private Vertex selectedVertex;
    private Vertex firstSelectedVertex;
    private Vertex startVertex;
    private String currentTool;
    private static final Color SELECTED_COLOR = Color.RED;
    private static final Color START_VERTEX_COLOR = Color.BLUE;
    private static final Color NEIGHBOR_COLOR = Color.ORANGE;
    private static final Color DEFAULT_VERTEX_COLOR = Color.BLACK;
    private static final Color DEFAULT_EDGE_COLOR = Color.BLACK;
    private Timer timer;
    private int stepIndex;
    private List<Runnable> steps;
    private List<String> logSteps;
    private boolean isPaused;
    private boolean isVisualizationStarted;
```

```
private JTextArea logTextArea;

public GraphPanel() {
    vertices = new ArrayList<>();
    edges = new ArrayList<>();
    currentTool = "Move";
    steps = new ArrayList<>();
    logSteps = new ArrayList<>();
    stepIndex = 0;
    isPaused = true;
    isVisualizationStarted = false;

    addMouseListener(new MouseAdapter() {
        @Override
        public void mousePressed(MouseEvent e) {
            handleMousePressed(e);
        }
    });

    addMouseMotionListener(new MouseAdapter() {
        @Override
        public void mouseDragged(MouseEvent e) {
            handleMouseDragged(e);
        }
    });
}

public GraphPanel(GraphPanel original) {
    this();
    this.vertices = new ArrayList<>(original.vertices);
    this.edges = new ArrayList<>(original.edges);
    this.startVertex = original.startVertex;
}

public List<Vertex> getVertices() {
    return vertices;
}

public List<Edge> getEdges() {
    return edges;
}

public void setStartVertex(Vertex startVertex) {
    this.startVertex = startVertex;
    startVertex.setColor(START_VERTEX_COLOR);
    repaint();
}

public void setTool(String tool) {
    currentTool = tool;
    firstSelectedVertex = null;
    deselectAllVertices();
    repaint();
}

private void handleMousePressed(MouseEvent e) {
```

```
        if (currentTool.equals("Add Vertex")) {
            String vertexName = JOptionPane.showInputDialog(this, "Enter
vertex name:");
            if (vertexName != null && !vertexName.trim().isEmpty()) {
                vertices.add(new Vertex(e.getX(), e.getY(), vertexName));
                repaint();
            }
        } else if (currentTool.equals("Move")) {
            for (Vertex vertex : vertices) {
                if (vertex.contains(e.getPoint())) {
                    selectedVertex = vertex;
                    break;
                }
            }
        } else if (currentTool.equals("Add Edge")) {
            for (Vertex vertex : vertices) {
                if (vertex.contains(e.getPoint())) {
                    if (firstSelectedVertex == null) {
                        firstSelectedVertex = vertex;
                        vertex.setColor(SELECTED_COLOR);
                    } else if (firstSelectedVertex != vertex) {
                        String weightStr =
JOptionPane.showInputDialog(this, "Enter edge weight:");
                        try {
                            int weight = Integer.parseInt(weightStr);
                            if (weight < 0) {
                                throw new IllegalArgumentException("Edge
weight must be a positive number.");
                            }
                            edges.add(new Edge(firstSelectedVertex,
vertex, weight));

firstSelectedVertex.setColor(DEFAULT_VERTEX_COLOR); // Reset color
                            firstSelectedVertex = null;
                            repaint();
                        } catch (NumberFormatException ex) {
                            JOptionPane.showMessageDialog(this, "Invalid
weight. Please enter a number.");
                        } catch (IllegalArgumentException ex) {
                            JOptionPane.showMessageDialog(this,
ex.getMessage());
                        }
                    }
                    break;
                }
            }
            repaint();
        } else if (currentTool.equals("Delete")) {
            for (Vertex vertex : vertices) {
                if (vertex.contains(e.getPoint())) {
                    vertices.remove(vertex);
                    edges.removeIf(edge -> edge.getStart() == vertex ||
edge.getEnd() == vertex);
                    repaint();
                    return;
                }
            }
```

```
            for (Edge edge : edges) {
                if (edge.contains(e.getPoint())) {
                    edges.remove(edge);
                    repaint();
                    return;
                }
            }
        } else if (currentTool.equals("Start Vertex")) {
            for (Vertex vertex : vertices) {
                if (vertex.contains(e.getPoint())) {
                    if (startVertex != null) {
                        startVertex.setColor(DEFAULT_VERTEX_COLOR);
                    }
                    startVertex = vertex;
                    startVertex.setColor(START_VERTEX_COLOR);
                    repaint();
                    break;
                }
            }
        }
    }

    private void handleMouseDragged(MouseEvent e) {
        if (currentTool.equals("Move") && selectedVertex != null) {
            selectedVertex.setLocation(e.getX(), e.getY());
            repaint();
        }
    }

    private void deselectAllVertices() {
        for (Vertex vertex : vertices) {
            if (!vertex.isFinalized()) {
                vertex.setColor(DEFAULT_VERTEX_COLOR);
            }
        }
        if (startVertex != null && !startVertex.isFinalized()) {
            startVertex.setColor(START_VERTEX_COLOR);
        }
    }

    public void clearArea() {
        vertices.clear();
        edges.clear();
        startVertex = null;
        repaint();
    }

    public void loadGraph(File file) {
        clearArea();
        try (Scanner scanner = new Scanner(file)) {
            int vertexCount = scanner.nextInt();
            int edgeCount = scanner.nextInt();
            scanner.nextLine();

            for (int i = 0; i < vertexCount; i++) {
                String[] vertexData = scanner.nextLine().split(" ");
                String name = vertexData[0];
```

```
                int x = Integer.parseInt(vertexData[1]);
                int y = Integer.parseInt(vertexData[2]);
                vertices.add(new Vertex(x, y, name));
            }

            for (int i = 0; i < edgeCount; i++) {
                String[] edgeData = scanner.nextLine().split(" ");
                Vertex start = getVertexByName(edgeData[0]);
                Vertex end = getVertexByName(edgeData[1]);
                int weight = Integer.parseInt(edgeData[2]);
                if (weight < 0) {
                    throw new IllegalArgumentException("Edge weight must
be a positive number.");
                }
                if (start != null && end != null) {
                    edges.add(new Edge(start, end, weight));
                }
            }

            repaint();
        } catch (FileNotFoundException e) {
            JOptionPane.showMessageDialog(this, "File not found: " +
e.getMessage());
        } catch (IllegalArgumentException e) {
            JOptionPane.showMessageDialog(this, e.getMessage());
        } catch (Exception e) {
            JOptionPane.showMessageDialog(this, "Error loading graph: " +
e.getMessage());
        }
    }

    private Vertex getVertexByName(String name) {
        for (Vertex vertex : vertices) {
            if (vertex.getName().equals(name)) {
                return vertex;
            }
        }
        return null;
    }

    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        for (Edge edge : edges) {
            edge.draw(g);
        }
        for (Vertex vertex : vertices) {
            vertex.draw(g);
        }
    }

    public void resetGraph() {
        for (Vertex vertex : vertices) {
            vertex.reset();
        }
        for (Edge edge : edges) {
            edge.setColor(DEFAULT_EDGE_COLOR);
```

```
        }
    }

    public void startDijkstra(JTextArea logTextArea) {
        this.logTextArea = logTextArea;
        resetGraph();
        isVisualizationStarted = true;
        if (startVertex == null) {
            logTextArea.append("No start vertex selected.\n");
            return;
        }

        Map<Vertex, Integer> dist = new HashMap<>();
        Map<Vertex, Vertex> prev = new HashMap<>();
        PriorityQueue<Vertex> queue = new
PriorityQueue<>(Comparator.comparingInt(dist::get));

        for (Vertex vertex : vertices) {
            dist.put(vertex, Integer.MAX_VALUE);
            prev.put(vertex, null);
            vertex.setDistance(Integer.MAX_VALUE); // Установить
начальное расстояние для всех вершин
        }
        dist.put(startVertex, 0);
        startVertex.setDistance(0); // Установить начальное расстояние
для стартовой вершины

        queue.addAll(vertices);

        steps.clear();
        logSteps.clear();
        stepIndex = 0;

        steps.add(() -> {
            log("Starting Dijkstra's algorithm from vertex " +
startVertex.getName() + ".\n");
            startVertex.setDistance(0);
            repaint();
        });

        while (!queue.isEmpty()) {
            Vertex u = queue.poll();
            final String currentVertexLog = "Current vertex: " +
u.getName() + " (distance: " + dist.get(u) + ")\n";
            steps.add(() -> {
                log(currentVertexLog);
                if (!u.isFinalized()) {
                    u.setColor(SELECTED_COLOR);
                }
                repaint();
            });

            for (Edge edge : edges) {
                Vertex v = null;
                if (edge.getStart() == u) {
                    v = edge.getEnd();
                } else if (edge.getEnd() == u) {
```

```
                    v = edge.getStart();
                }
                if (v != null) {
                    final Vertex neighbor = v;
                    final String checkingNeighborLog = "Checking
neighbor: " + neighbor.getName() + " (current distance: " +
dist.get(neighbor) + ")\n";
                    steps.add(() -> {
                        edge.setColor(NEIGHBOR_COLOR);
                        if (!neighbor.isFinalized()) {
                            neighbor.setColor(NEIGHBOR_COLOR);
                        }
                        log(checkingNeighborLog);
                        repaint();
                    });
                    int alt = dist.get(u) + edge.getWeight();
                    if (alt < dist.get(neighbor)) {
                        final int newDist = alt;
                        dist.put(neighbor, newDist);
                        prev.put(neighbor, u);
                        final String shorterPathLog = "Found a shorter
path to vertex " + neighbor.getName() + ": " + newDist + "\n";
                        steps.add(() -> {
                            log(shorterPathLog);
                            neighbor.setDistance(newDist);
                            repaint();
                        });
                        queue.remove(neighbor);
                        queue.add(neighbor);
                    }
                }
            }

            final String queueState = getQueueNames(queue);
            final String updatedDistancesLog = "Updated distances: " +
getDistances(dist) + "\n";
            final String queueLog = "Queue: " + queueState + "\n";
            steps.add(() -> {
                log(updatedDistancesLog);
                log(queueLog);
                repaint();
            });

            final Vertex currentVertex = u;
            steps.add(() -> {
                resetEdgeColors();
                paintPathToVertex(currentVertex, prev);
                currentVertex.setFinalized(true);
                currentVertex.setColor(SELECTED_COLOR);
                repaint();
            });
        }
        steps.add(() -> {
            log("Dijkstra's algorithm completed.\n");
        });

        isPaused = false;
```

```
    }

    private void resetEdgeColors() {
        for (Edge edge : edges) {
            edge.setColor(DEFAULT_EDGE_COLOR);
        }
    }

    private void paintPathToVertex(Vertex vertex, Map<Vertex, Vertex>
prev) {
        Vertex current = vertex;
        while (prev.get(current) != null) {
            Vertex previous = prev.get(current);
            for (Edge edge : edges) {
                if ((edge.getStart() == current && edge.getEnd() ==
previous) ||
                        (edge.getStart() == previous && edge.getEnd() ==
current)) {
                    edge.setColor(Color.BLUE);
                }
            }
            current = previous;
        }
    }

    public void startAutomaticVisualization() {
        if (timer != null) {
            timer.stop();
        }
        timer = new Timer(1000, e -> {
            if (stepIndex < steps.size()) {
                SwingUtilities.invokeLater(() -> {
                    steps.get(stepIndex).run();
                    stepIndex++;
                    updateLogDisplay(logTextArea);
                });
            } else {
                timer.stop();
            }
        });
        timer.start();
    }

    public void showNextStep(JTextArea logTextArea) {
        if (stepIndex < steps.size()) {
            SwingUtilities.invokeLater(() -> {
                steps.get(stepIndex).run();
                stepIndex++;
                updateLogDisplay(logTextArea);
            });
        }
    }

    public void prevStep(JTextArea logTextArea) {
        if (stepIndex > 0) {
            stepIndex--;
            resetGraph();
```

```
            for (int i = 0; i <= stepIndex; i++) {
                steps.get(i).run();
            }
            updateLogDisplay(logTextArea);
        }
    }

    public void pauseVisualization() {
        if (timer != null) {
            isPaused = true;
            timer.stop();
        }
    }

    public boolean isVisualizationStarted() {
        return isVisualizationStarted;
    }

    public void updateLogDisplay(JTextArea logTextArea) {
        logTextArea.setText("");
        for (int i = 0; i <= stepIndex; i++) {
            logTextArea.append(logSteps.get(i));
        }
    }

    private void log(String message) {
        while (logSteps.size() <= stepIndex) {
            logSteps.add("");
        }
        logSteps.set(stepIndex, message);
        logTextArea.append(message);
    }

    private String getDistances(Map<Vertex, Integer> dist) {
        StringBuilder sb = new StringBuilder();
        for (Map.Entry<Vertex, Integer> entry : dist.entrySet()) {
            sb.append("Vertex
").append(entry.getKey().getName()).append(": distance =
").append(entry.getValue()).append(", ");
        }
        return sb.toString();
    }

    private String getQueueNames(PriorityQueue<Vertex> queue) {
        StringBuilder sb = new StringBuilder();
        for (Vertex vertex : queue) {
            sb.append(vertex.getName()).append(", ");
        }
        return sb.toString();
    }

    public List<Runnable> getSteps() {
        return steps;
    }
}
```

Название файла: Vertex.java:

```
package org.example.MainProgramm;

import java.awt.*;
import java.awt.geom.Ellipse2D;

public class Vertex {
    private int x, y;
    private String name;
    private static final int SIZE = 20;
    private Color color;
    private int distance;
    private boolean finalized;

    public Vertex(int x, int y, String name) {
        this.x = x;
        this.y = y;
        this.name = name;
        this.color = Color.BLACK;
        this.distance = Integer.MAX_VALUE;
        this.finalized = false;
    }

    public Vertex(Vertex original) {
        this.x = original.x;
        this.y = original.y;
        this.name = original.name;
        this.color = original.color;
        this.distance = original.distance;
        this.finalized = original.finalized;
    }

    public void setLocation(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public String getName() {
        return name;
    }

    public Color getColor() {
        return color;
    }

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

```
    public void setDistance(int distance) {
        this.distance = distance;
    }

    public int getDistance() {
        return distance;
    }

    public boolean isFinalized() {
        return finalized;
    }

    public void setFinalized(boolean finalized) {
        this.finalized = finalized;
    }

    public void reset() {
        this.color = Color.BLACK;
        this.distance = Integer.MAX_VALUE;
        this.finalized = false;
    }

    public boolean contains(Point p) {
        return new Ellipse2D.Float(x - SIZE / 2, y - SIZE / 2, SIZE,
SIZE).contains(p);
    }

    public void draw(Graphics g) {
        g.setColor(color);
        g.fillOval(x - SIZE / 2, y - SIZE / 2, SIZE, SIZE);
        g.setColor(Color.WHITE);
        g.drawString(name, x - SIZE / 4, y + SIZE / 4);
        g.drawString(distance == Integer.MAX_VALUE ? "∞" :
String.valueOf(distance), x - SIZE / 2, y - SIZE / 2 - 5);
    }
}
```

Название файла: Edge.java:

```
package org.example.MainProgramm;

import java.awt.*;
import java.awt.geom.Line2D;

public class Edge {
    private Vertex start, end;
    private int weight;
    private Color color;

    public Edge(Vertex start, Vertex end, int weight) {
        if (start == null || end == null) {
            throw new NullPointerException("Vertices cannot be null");
        }
        if (weight < 0) {
            throw new IllegalArgumentException("Edge weight must be a
positive number.");
        }
        this.start = start;
```

```
        this.end = end;
        this.weight = weight;
        this.color = Color.BLACK;
    }

    public Edge(Edge original) {
        this.start = original.start;
        this.end = original.end;
        this.weight = original.weight;
        this.color = original.color;
    }

    public Vertex getStart() {
        return start;
    }

    public Vertex getEnd() {
        return end;
    }

    public int getWeight() {
        return weight;
    }

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

    public boolean contains(Point p) {
        int x1 = start.getX();
        int y1 = start.getY();
        int x2 = end.getX();
        int y2 = end.getY();
        int px = p.x;
        int py = p.y;

        double distance = Line2D.ptSegDist(x1, y1, x2, y2, px, py);
        return distance < 5;
    }

    public void draw(Graphics g) {
        g.setColor(color);
        g.drawLine(start.getX(), start.getY(), end.getX(), end.getY());
        g.drawString(String.valueOf(weight), (start.getX() + end.getX())
/ 2, (start.getY() + end.getY()) / 2);
    }
```