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

ОТЧЕТ

по учебной практике

Тема: Визуализация алгоритма Краскала

Студент гр. 3341 _____ Ягудин Д.Р.

Студент гр. 3341 _____ Че М.Б.

Студент гр. 3341 _____ Костромитин М.М.

Руководитель _____ Фирсов М.А.

Санкт-Петербург

2025

ЗАДАНИЕ

НА УЧЕБНУЮ ПРАКТИКУ

Студент Ягудин Д.Р. группы 3341

Студент Че М.Б. группы 3341

Студент Фирсов М.А. группы 3341

Тема практики: Визуализация алгоритма Краскала

Задание на практику:

Командная итеративная разработка визуализатора алгоритма(ов) на Java с графическим интерфейсом.

Алгоритм: алгоритм Краскала.

Сроки прохождения практики: 25.06.2024 – 08.07.2024

Дата сдачи отчета: 04.07.2024

Дата защиты отчета: 04.07.2024

Студент гр. 3341	_____	Ягудин Д.Р.
Студент гр. 3341	_____	Че М.Б.
Студент гр. 3341	_____	Костромитин М.М.
Руководитель	_____	Фирсов М.А.

АННОТАЦИЯ

Цель практики:

Разработка визуализации алгоритма Краскала для построения минимального остовного дерева (MST) с пошаговым отображением процесса.

Основное содержание:

1. Реализация графа с вершинами и рёбрами (классы Vertex, Edge).
2. Логика алгоритма Краскала (Kruskal), включая детекцию циклов (CycleDetector).
3. Сохранение состояний (State) для пошаговой визуализации.
4. GUI (Swing) с панелью графа (GraphPanel) и управляющими кнопками.
5. Взаимодействие: добавление/удаление вершин и рёбер, загрузка из файла, генерация случайного графа.
6. Визуализация включённых (зелёные), исключённых (красные) рёбер и циклов (розовые).
7. Логирование шагов алгоритма в текстовое поле.

SUMMARY

Practice Objective:

Development of a visualization for Kruskal's algorithm to construct a Minimum Spanning Tree (MST) with step-by-step process display.

Main Content:

1. Implementation of a graph with vertices and edges (classes Vertex, Edge).
2. Kruskal's algorithm logic (Kruskal), including cycle detection (CycleDetector).
3. Saving states (State) for step-by-step visualization.
4. GUI (Swing) with a graph panel (GraphPanel) and control buttons.
5. Interaction: adding/removing vertices and edges, loading from a file, generating a random graph.
6. Visualization of included (green), excluded (red) edges, and cycles (pink).
7. Logging algorithm steps in a text field.

СОДЕРЖАНИЕ

	Введение	5
1.	Требования к программе	6
1.1.	Исходные требования к программе	6
1.2.	Уточнение требований после сдачи прототипа	8
1.3.	Уточнение требований после сдачи 1-ой версии	8
1.4	Уточнение требований после сдачи 2-ой версии	8
2.	План разработки и распределение ролей в бригаде	9
2.1.	План разработки	9
2.2.	Распределение ролей в бригаде	10
3.	Особенности реализации	11
3.1.	Структуры данных	11
3.2.	Основные методы	13
4.	Тестирование	17
4.1	Тестирование графического интерфейса	17
4.2	Тестирование кода алгоритма	22
	Заключение	23
	Список использованных источников	24
	Приложение А. Исходный код – только в электронном виде	25

ВВЕДЕНИЕ

Цель практики:

Разработать интерактивное приложение для визуализации алгоритма Краскала, позволяющее наглядно изучать процесс построения минимального остовного дерева (MST) в графе.

Задачи:

1. Реализовать структуры данных для представления графа (вершины, рёбра).
2. Разработать алгоритм Краскала с детекцией циклов.
3. Обеспечить пошаговое выполнение и сохранение состояний.
4. Создать графический интерфейс (GUI) для взаимодействия.
5. Добавить функции загрузки данных и генерации случайных графов.

Алгоритм Краскала:

6. Сортирует рёбра по весу.
7. Последовательно добавляет рёбра в MST, избегая циклов.
8. Использует систему непересекающихся множеств (через поиск путей) для проверки связности.

1. ТРЕБОВАНИЯ К ПРОГРАММЕ

1.1 Исходные требования к программе:

1.1.1 Требование к вводу исходных данных:

1.1.1.1 Ввод через взаимодействие с холстом:

1.1.1.1.1 Добавление вершин кликом по холсту.

1.1.1.1.2 Добавление ребер кликом по двум вершинам.

1.1.1.1.3 Добавление и изменение веса ребра кликом по ребру.

1.1.1.2 Ввод через текстовый файл – матрицей смежности. Вершины размещаются в форме правильного многоугольника.

1.1.1.3 Генерация матрицы смежности для входных данных. С указанием количества вершин и рёбер.

1.1.2 Требования к визуализации:

1.1.2.1 Основные элементы интерфейса:

1.1.2.1.1 Холст для отображения графа. При рассмотрении потенциального ребра оно рисуется светло-серым цветом. Если ребро не добавляется из-за цикла(цикл показывается светло-розовым цветом), то само ребро перекрашивается в красный цвет.

1.1.2.1.2 Панель управления с кнопками взаимодействия.

1.1.2.1.3 Поле для отображения текущего шага алгоритма.

1.1.2.2 Функционал интерфейса:

1.1.2.2.1 При успешном добавлении ребра в МОД – раскрашивать его в зеленый цвет на холсте.

1.1.2.2.2 На каждом шаге алгоритма выводить текстовую информацию о данном шаге – какие ребра уже были добавлены в МОД до шага; какое ребро рассматривается; добавляется ли рассматриваемое ребро в итоговый МОД или нет.

1.1.2.2.3 Функционал свободного перемещения между шагами алгоритма.

1.1.2.2.4 При отмене ввода веса ребра не появляется окно с ошибкой неверного веса ребра.

1.1.2.2.5 При последовательном перемещении вершин не совершается попытка создания ребра.

1.1.3 Требование к выходным данным:

1.1.3.1 Текстовый файл с ребрами входящими в МОД.

1.1.3.2 Графическое представление МОД в приложении.

1.1.4 Схематическое представление графического представления

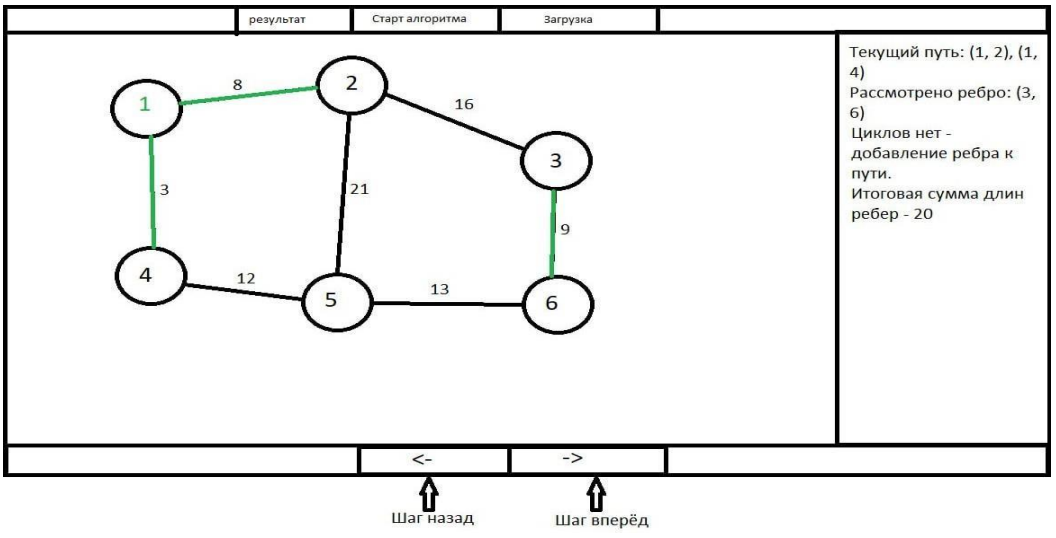

Рисунок 1 – графический интерфейс

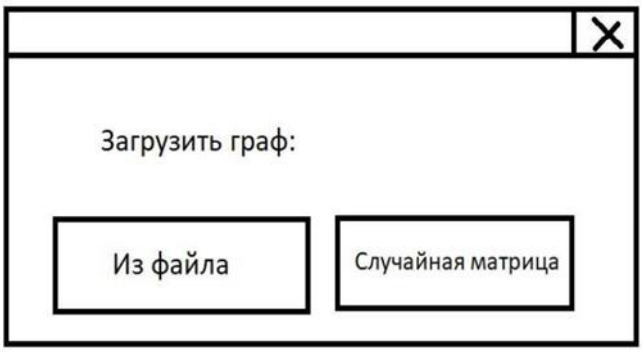

Рисунок 2 – Окно загрузки графа

1.2 Уточнение требований после сдачи прототипа

1.2.1 Добавить сверху приложения кнопку “Результат”.

1.2.2 Если нажата "Cancel" или закрыто окно для ввода веса, то не надо сообщать о некорректном вводе веса ребра.

1.2.3 Если пользователь сначала переместил одну вершину, то должна быть возможность затем сразу переместить другую, без диалога ввода ребра (программа должна учитывать, был ли щелчок по вершине или пользователь её двигал).

1.3 Уточнение требований после сдачи версии 1.0

1.3.1 Реализовать функцию запроса-подтверждения на прерывание алгоритма.

1.3.2 Добавить возможность удалять отдельные ребра.

1.4 Уточнение требований после сдачи версии 2.0

1.4.1 Исправить баг:

1) переместить вершину а;

2) щелчок по вершине б;

3) щелчок по вершине в — ребро не создаётся;

4) попытка переместить вершину а — вместо этого создаётся ребро.

1.4.2 Отвергнутые ребра рисовать немного более темным оттенком.

1.4.3 Исправить баг: “если сделать первый шаг и вернуться в начало, то появляется возможность редактировать граф: изменения не учитываются при дальнейшем выполнении алгоритма, и возможность редактирования не пропадает при последующих шагах.”

2. ПЛАН РАЗРАБОТКИ И РАСПРЕДЕЛЕНИЕ РОЛЕЙ В БРИГАДЕ

2.1. План разработки

Дата	Этап проекта	Реализованные возможности	Выполнено
27.06.2025	Согласование спецификации		Выполнено
30.06.2025	Сдача прототипа	Разработан интерфейс а так же сам алгоритм, возможность запуска алгоритма в интерфейсе. Предоставление списка ребер входящих в МОД в текстовом окне.	Выполнено
02.07.2025	Сдача версии 1	Визуализация готового МОД с отображением рёбер на холсте, включая пошаговое выполнение алгоритма с добавлением новых рёбер на полотне и текстовым описанием каждого шага, с возможностью перехода вперёд и назад	Выполнено 01.07.2025
04.07.2025	Сдача версии 2	Загрузка графа из файла, добавиться возможность сразу	Выполнено 03.07.2025

		увидеть готовый ответ, визуализация шагов алгоритма на графе пользователя.	
	Сдача версии 3	-	
04.07.2025	Сдача отчёта		
04.07.2025	Защита отчёта		

2.2. Распределение ролей в бригаде

2.2.1 Реализация графического интерфейса:

2.2.1.1 Общий дизайн графического интерфейса и его реализация – Ягудин Д.Р.

2.2.2 Реализация алгоритма Краскала – Че М.Б.

2.2.3 Взаимодействие интерфейса с алгоритмом Краскала и тестирование – Костромитин М.М.

3. ОСОБЕННОСТИ РЕАЛИЗАЦИИ

3.1 Структуры данных

3.1.1 *public static class Vertex* – класс для реализации вершины графа на холсте, имеет следующие поля:

3.1.1.1 *public int x* – координата x на холсте.

3.1.1.2 *public int y* – координата y на холсте.

3.1.1.3 *public int radius* – радиус круга, который будет начерчен на холсте по координатам *x, y*.

3.1.1.4 *public int label* – номер вершины.

3.1.2 *public static class Edge implements Comparable<Edge>* - класс для реализации ребра графа на холсте, имеет следующие поля:

3.1.2.1 *public Vertex v1* – первая вершина ребра.

3.1.2.2 *public Vertex v2* – вторая вершина ребра.

3.1.2.3 *public int weight* – вес ребра.

3.1.3 *public record State(List<GraphPanel.Edge> includedEdges, List<GraphPanel.Edge> excludedEdges, int totalWeight, GraphPanel.Edge currentEdge, boolean isIncluded, List<GraphPanel.Edge> cycleEdges)* – хранит данные о шаге алгоритма, имеет следующие поля:

3.1.3.1 *List<GraphPanel.Edge> includedEdges* – список включенных в МОД ребер.

3.1.3.2 *List<GraphPanel.Edge> excludedEdges* – список исключенных ребер, которые образуют циклы при добавлении в частичный МОД.

3.1.3.3 *int totalWeight* – вес ребер частичного МОДа.

3.1.3.4 *GraphPanel.Edge currentEdge* – рассматриваемое на текущем шаге ребро.

3.1.3.5 *boolean isIncluded* – содержит true если ребро *currentEdge* было добавлено в МОД на этом шаге, иначе содержит false.

3.1.3.6 *List<GraphPanel.Edge> cycleEdges* – список ребер цикла, из-за которого ребро *currentEdge* не было добавлено, если *currentEdge* было добавлено, то содержит пустой список.

3.1.4 *class CycleDetector* – содержит методы для нахождения циклов в частичном МОДе.

3.1.5 *public class Kruskal* – класс реализующий алгоритм Краскала, имеет следующие поля:

3.1.5.1 *private final int numVertices* – количество вершин в графе.

3.1.5.2 *private final List<GraphPanel.Edge> edges* – список всех ребер графа.

3.1.5.3 private final List<State> states – список всех шагов алгоритма, хранящихся в виде класса *State.*

3.1.6 *public class GraphPanel extends JPanel* – класс холста, имеет следующие поля:

3.1.6.1 *private Vertex draggedVertex* – содержит вершину, которую перетаскивают на холсте, по умолчанию null.

3.1.6.2 *private Point offset* – точка для хранения разницы координат между оригинальным положением вершины и его будущим положением после перетаскивания.

3.1.6.3 *private boolean vertexMoved* – по умолчанию равно false, после перетаскивания любой вершины принимает значение true, нужно для устранения багов с последовательным перемещением и добавлением новых ребер.

3.1.6.4 *private boolean algorithmRunning* – по умолчанию равно false, если использованы кнопки для запуска алгоритма или результат, меняет значение на true.

3.1.6.5 *private java.util.ArrayList<Vertex> vertices* – содержит вершины графа.

3.1.6.6 *private java.util.ArrayList<Edge> edges* – содержит ребра графа.

3.1.6.7 *private Vertex selectedVertex* – содержит вершину, на которую нажал пользователь, нужно для реализации добавления ребра.

3.1.6.8 *private JTextArea logArea* – хранит поле для вывода текстового описания шагов в окне приложения.

3.1.6.9 *private java.util.List<State> algorithmSteps* – список всех шагов алгоритма.

3.1.6.10 *private int currentStep* – номер текущего шага алгоритма – индекс списка *algorithmSteps.*

3.1.6.11 *private java.util.List<Edge> shownEdges* – список ребер, включенных в частичный МОД, которые должны быть отображены зеленым на холсте.

3.1.6.12 *private java.util.List<Edge> excludedEdges* – аналогично поле для невключенных ребер.

3.1.6.13 *private java.util.List<Edge> cycleEdges* – аналогично поле для ребер цикла, из-за которого ребро не будет включено, выделяет ребра цикла только на одном шаге.

3.1.6.14 *private TreeSet<Integer> freeLabels* – содержит свободные индексы для вершин, пополняется если вершины удаляются.

3.1.6.15 *private int labelCounter* – по умолчанию 1, хранит индекс следующей вершины, при добавлении вершины увеличивается на 1.

3.1.7 *public class GraphApp extends JFrame* – класс приложения, имеет следующие поля:

3.1.7.1 *private GraphPanel graphPanel* – холст, на котором отображается граф.

3.1.7.2 *private JTextArea logArea* – текстовое окно, которое отображает информацию об очередном шаге алгоритма.

3.1.7.3 *private JButton stepBackButton* – кнопка возвращающая прошлый шаг алгоритма.

3.1.7.4 *private JButton stepForwardButton* – кнопка для перехода на следующий шаг алгоритма.

3.2 Основные методы

3.2.1 **Класс Main**

3.2.1.1 *public static void main(String[] args)*: Точка входа в программу. Запускает графическое приложение GraphApp в потоке обработки событий Swing.

3.2.2 Класс GraphApp

3.2.2.1 *public class GraphApp():* Конструктор, создает главное окно приложения с интерфейсом:

- Панель с кнопками "Старт Алгоритма", "Загрузка", "Результат"
- Панель управления с кнопками шагов "<--" и "-->"
- Область для вывода логов
- Основную панель для отображения графа (GraphPanel)

3.2.2.2 *private void showLoadOptions():* Показывает диалог выбора способа загрузки графа:

- "Из файла" - открывает файловый диалог
- "Случайная матрица" - запрашивает параметры генерации

3.2.2.3 *private void runAlgorithmResult():* Запускает алгоритм Краскала и сразу показывает результат

3.2.2.4 *private void runAlgorithm():* Запускает пошаговое выполнение алгоритма Краскала

3.2.2.5 *private void stepBack():* Пошаговая навигация по алгоритму

3.2.2.6 *private void stepForward():* Пошаговая навигация по алгоритму

3.2.3 Класс CycleDetector

3.2.3.1 *findPath():* Находит путь между двумя вершинами в графе с помощью DFS

3.2.3.2 *findCycleEdges():* Находит цикл, образованный добавлением нового ребра

3.2.4 Класс Kruskal

3.2.4.1 *public Kruskal():* Конструктор, инициализирует алгоритм с заданными ребрами и количеством вершин

3.2.4.2 *public void addEdge():* Добавляет ребро в граф

3.2.4.3 *public ArrayList<GraphPanel.Edge> computeMST():* Выполняет алгоритм Краскала:

- Сортирует ребра по весу
- Последовательно добавляет ребра, избегая циклов
- Сохраняет состояния алгоритма на каждом шаге

3.2.5 Класс State

3.2.5.1 *State():* Конструктор, сохраняет состояние алгоритма:

- Включенные и исключенные ребра
- Текущее ребро
- Флаг включения ребра
- Ребра цикла

3.2.5.2 *public String toString():* Форматирует состояние для вывода в лог

3.2.5.3 *public List<GraphPanel.Edge> getIncludedEdges():* Возвращает список рёбер, включённых в минимальное остовное дерево (MST) на текущем шаге алгоритма.

3.2.5.4 *public List<GraphPanel.Edge> getExcludedEdges():* Возвращает список рёбер, исключённых из MST на текущем шаге (из-за образования цикла).

3.2.5.5 *public int getTotalWeight():* Возвращает суммарный вес всех рёбер, включённых в MST на текущем шаге.

3.2.5.6 *public GraphPanel.Edge getCurrentEdge():* Возвращает ребро, которое обрабатывается на текущем шаге алгоритма. Для начального состояния (до обработки первого ребра) возвращает null.

3.2.5.7 *public boolean isIncluded():* Возвращает true, если текущее ребро (currentEdge) было добавлено в MST, и false — если исключено (из-за цикла).

3.2.5.8 *public List<GraphPanel.Edge> getCycleEdges():* Возвращает список рёбер, образующих цикл при попытке добавить currentEdge. Пустой список, если цикл не обнаружен или ребро было включено в MST.

3.2.6 **Класс GraphPanel**

3.2.6.1 *public GraphPanel():* Конструктор, инициализирует панель для отображения графа и обработки событий мыши

3.2.6.2 *private boolean confirmAlgorithmInterruption():* Показывает диалог подтверждения прерывания алгоритма

3.2.6.3 *private Edge getEdgeAt()/private Vertex getVertexAt():* Находят элементы графа по координатам

3.2.6.4 *public void generateRandomGraph():* Генерирует случайный связный граф с заданными параметрами

3.2.6.5 *private int getNextLabel():* Возвращает следующий свободный номер для вершины

3.2.6.6 *protected void paintComponent():* Отрисовывает граф с учетом текущего состояния алгоритма

3.2.6.7 *public void runAlgorithmResult():* Запускает алгоритм и сразу показывает итоговое MST

3.2.6.8 *public void runAlgorithm():* Запускает алгоритм Краскала и сохраняет шаги для визуализации

3.2.6.9 *public void step():* Переходит на указанный шаг алгоритма, обновляя отображение

3.2.6.10 *public void loadFromFile():* Загружает граф из файла (матрицы смежности)

3.2.6.11 *public int getCurrentStep():* Управляет состоянием выполнения алгоритма

3.2.6.12 *public void setAlgorithmRunning():* Управляет состоянием выполнения алгоритма

4. ТЕСТИРОВАНИЕ

4.1 Тестирование графического интерфейса

4.1.1 Тестирование кнопки “Загрузка”

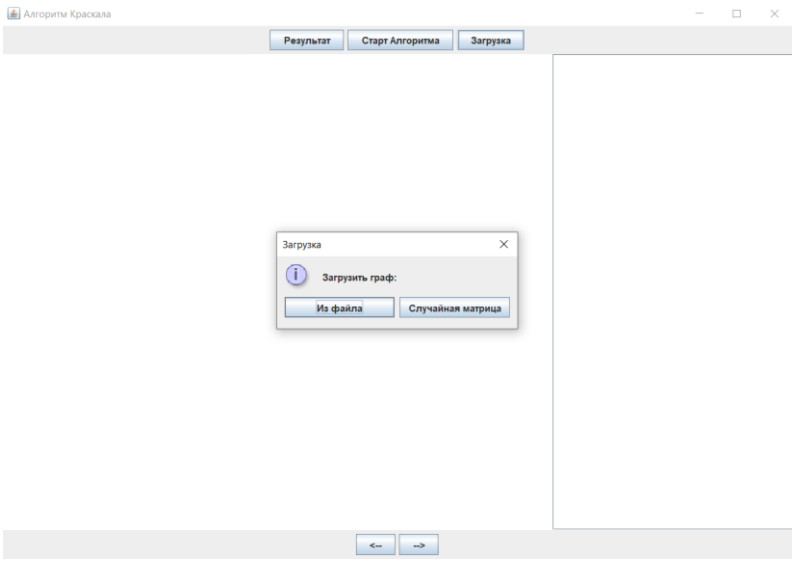

Рисунок 3 – работа кнопки “Загрузка”

4.1.1.1 Тестирование кнопки “Из файла”

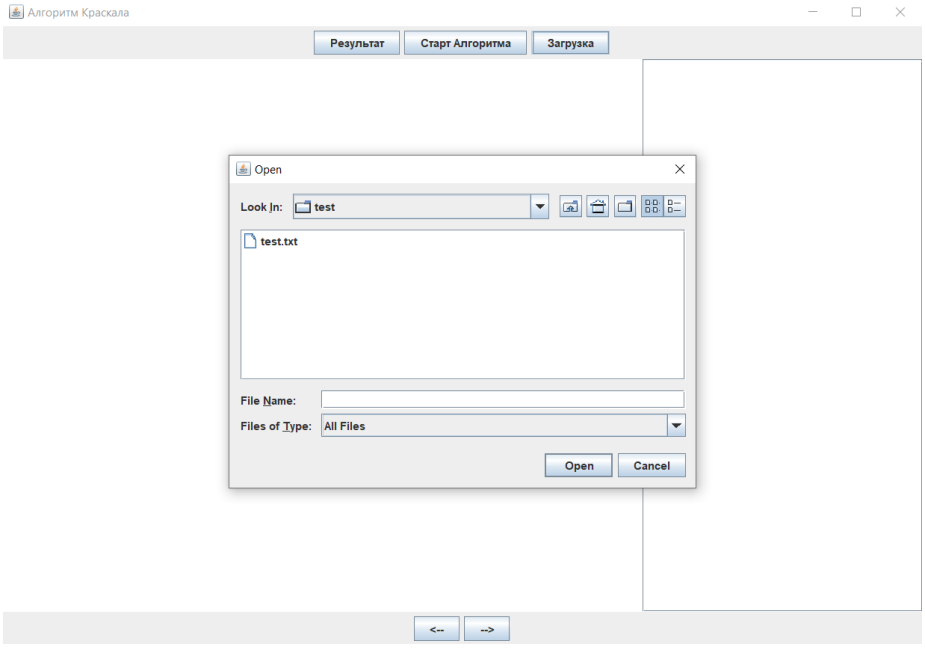

Рисунок 4 – выбор текстового документа

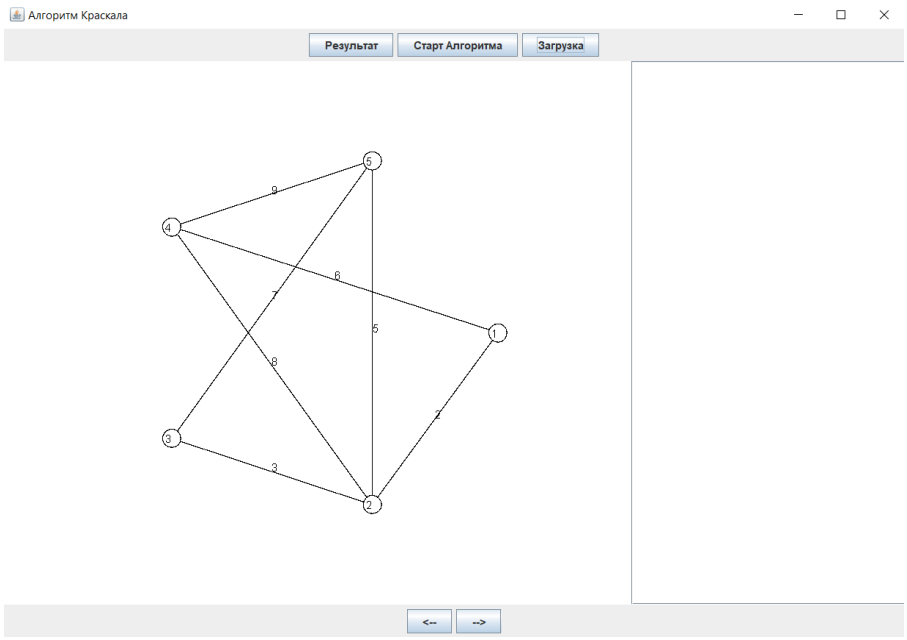

Рисунок 5 – визуализация из txt файла

4.1.1.2 Тестирование кнопки “Случайная матрица”

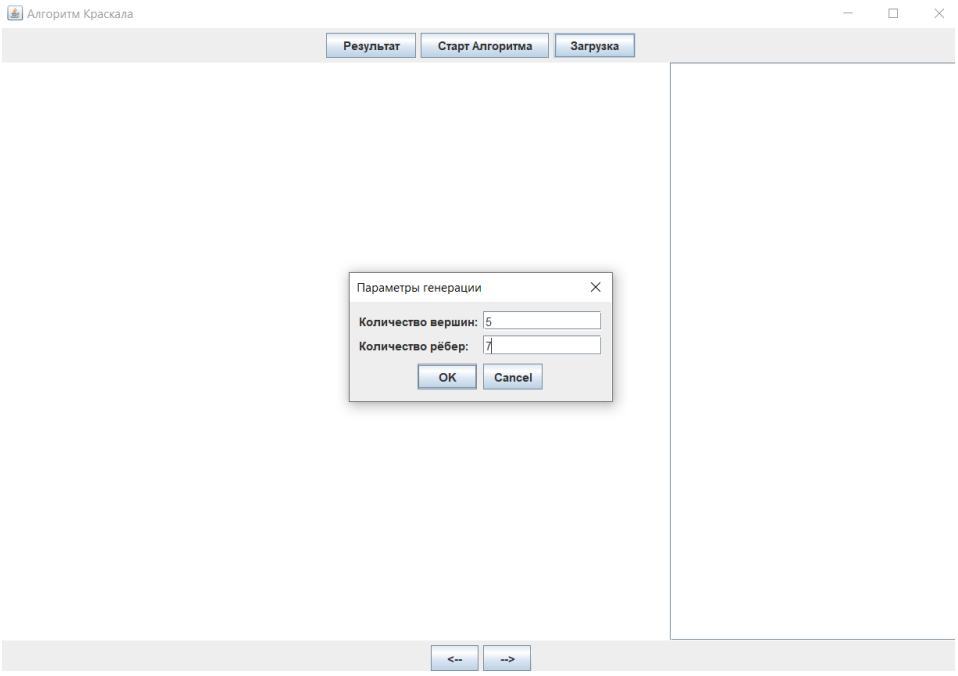

Рисунок 6 – настройка параметров генерации графа

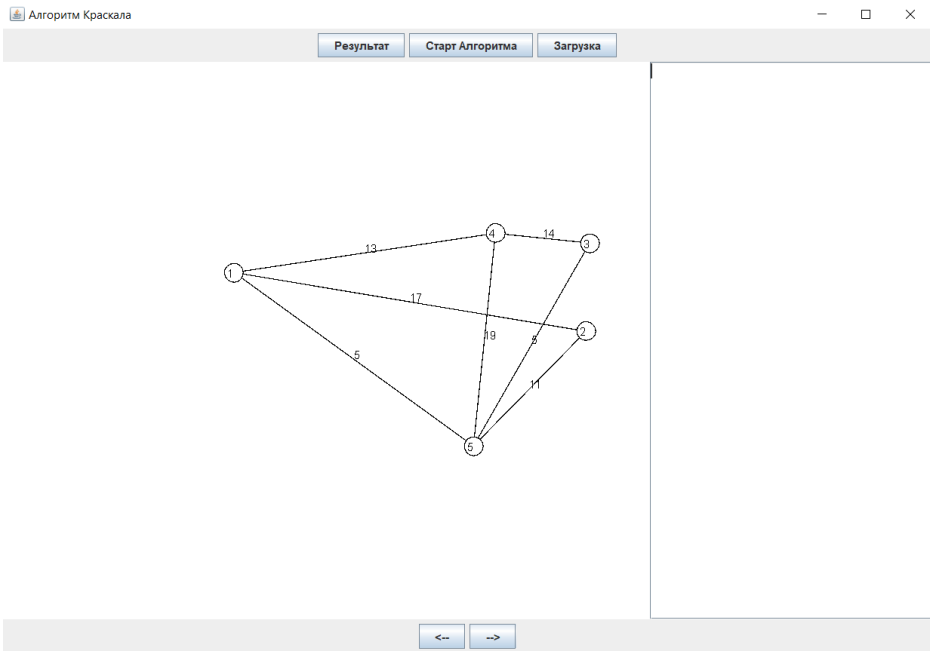

Рисунок 7 – визуализация сгенерированного графа

4.1.2 Тестирование кнопки “Старт алгоритма”

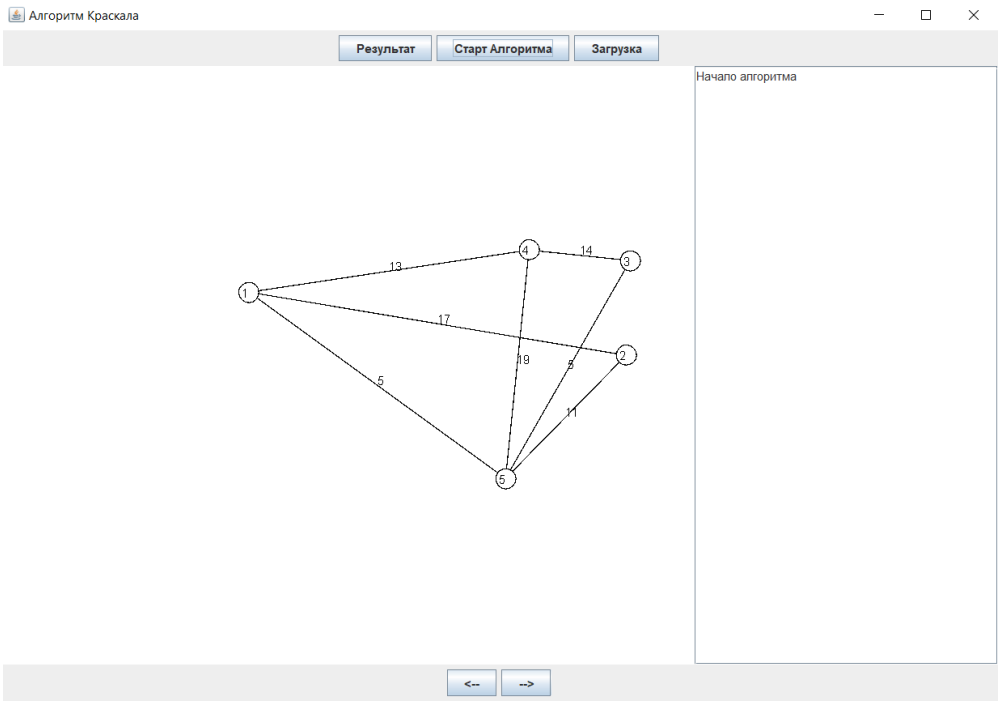

Рисунок 8 – Запуск алгоритма

4.1.3 Тестирование кнопок “<-”/“->”

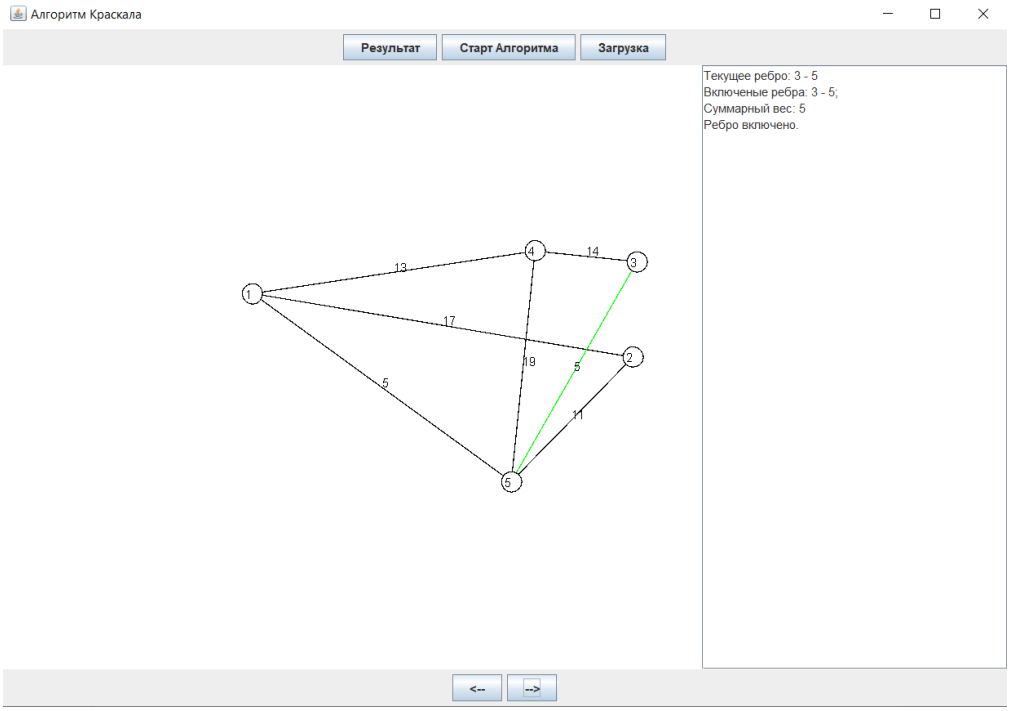

Рисунок 9 – Был сделан шаг по алгоритму вперед.

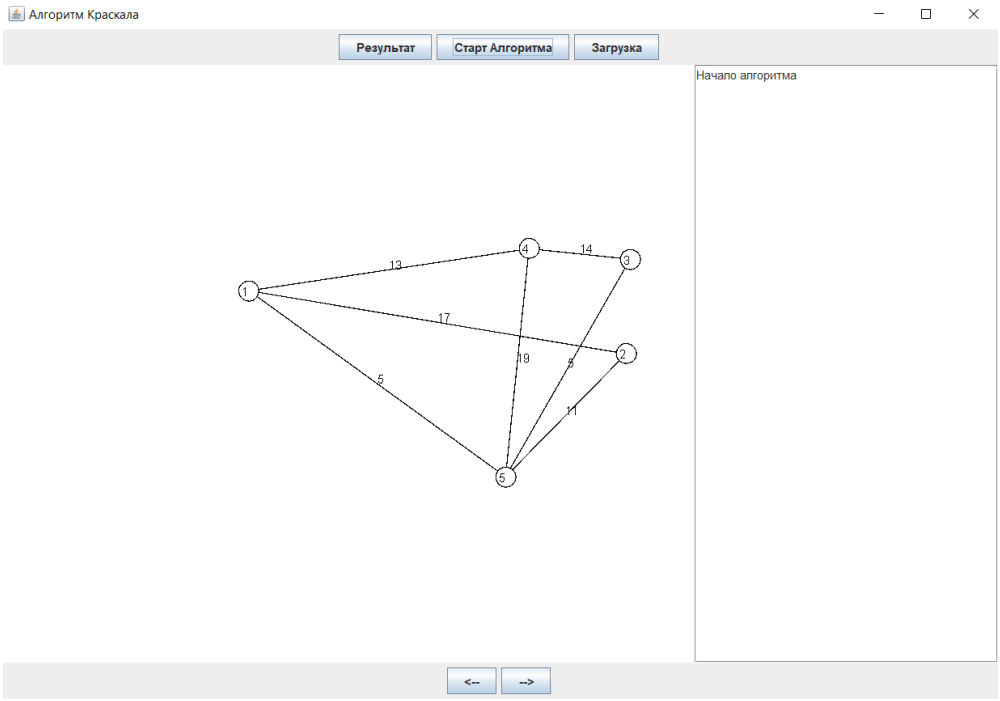

Рисунок 10 – Был сделан шаг по алгоритму назад.

4.1.4 Тестирование кнопки “Результат”

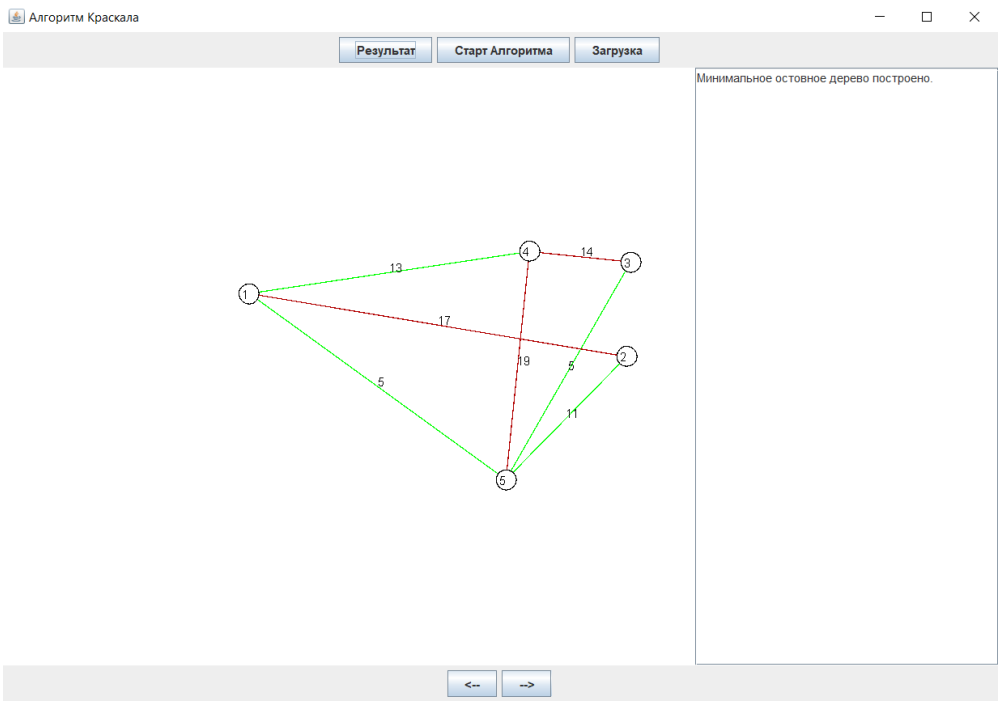

Рисунок 11 – Визуализация результата алгоритма

4.1.5 Тестирование окрашивания циклов алгоритма/отвергнутых рёбер

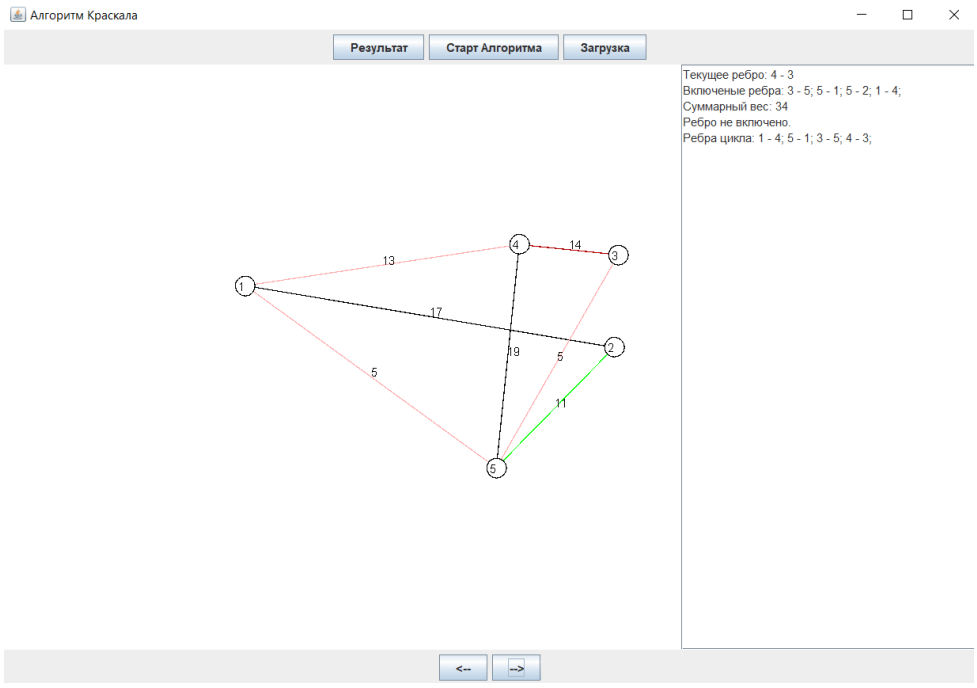

Рисунок 11 – Окрашивание циклов/рёбер

4.2 Тестирование алгоритма

№	Входные данные	Ребра МОД	Комментарий
1	0 2 0 6 0 2 0 3 8 5 0 3 0 0 7 6 8 0 0 9 0 5 7 9 0	1-2; 2-3; 2-5; 1-4	Связный неориентированный граф
2	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0		Нуль граф, никакие ребра не рассматривались алгоритмом
3	0 0 0 0 6 0 0 3 0 0 0 3 0 0 0 0 0 0 0 9 6 0 0 9 0	2-3; 1-5; 4-5	Несвязный неориентированный граф, создаст МОДы для каждой компоненты связности
4	0 5 2 3 6 5 0 3 9 10 2 3 0 6 4 3 9 6 0 9 6 10 4 9 0	1 - 3; 1 - 4; 2 - 3; 3 - 5	Полный граф с 5 вершинами
5	0 0 0 0 6 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 6 0 0 0 0	1-5	Граф с одни ребром
6	0		Граф с одной вершиной, ребер нет поэтому алгоритм работает аналогично случаю с нуль графом
7	0 7 1 3 5 6 1 3 6 9 2 2 3 6 5 7 0 9 5 4 1 7 9 1 4 8 9 3 9 9 1 9 0 3 8 5 5 6 1 7 7 8 5 6 9 3 5 3 0 8 7 7 1 3 9 9 7 8 6 8 5 4 8 8 0 2 4 6 5 4 2 2 7	1 - 3; 1 - 7; 2 - 6; 2 - 9; 3 - 9; 4 - 8; 7 - 8; 7 - 10; 8 - 11; 8 - 14; 12 - 13; 1 - 12; 5 - 6; 12 - 15	Граф с относительно большим количеством вершин (15)

	10 9 **6 1 5 7 2 0 3 8 7 2 5 10 8 5 6** **1 7 5 7 4 3 0 1 9 1 3 3 4 3 5** **3 9 6 1 6 8 1 0 6 9 1 4 10 1 5** **6 1 1 3 5 7 9 6 0 6 8 2 8 3 7** **9 4 7 9 4 2 1 9 6 0 10 7 8 3 6** **2 8 7 9 2 5 3 1 8 10 0 3 9 5 9** **2 9 8 7 2 10 3 4 2 7 3 0 1 3 2** **3 3 5 8 7 8 4 10 8 8 9 1 0 8 9** **6 9 6 6 10 5 3 1 3 3 5 3 8 0 6** **5 9 9 8 9 6 5 5 7 6 9 2 9 6 0**		

Заключение

Цель (визуализация алгоритма Краскала) достигнута:

Создано интерактивное приложение с пошаговым показом MST.

Задачи выполнены:

- Структуры «Вершина»/«Ребро» реализованы.
- Алгоритм Краскала с детекцией циклов через Union-Find (и альтернативно — обход) работает корректно.
- Каждое состояние (список рёбер, вес) сохраняется для навигации.
- GUI с контролами «Шаг»/«Авто»/ползунок обеспечил удобство взаимодействия.
- Добавлены загрузка из файла и генерация случайных графов.

Вывод: заявленная цель и все задачи выполнены; приложение готово к использованию и дальнейшему расширению.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Обучающий курс по основам языка Java // Stepik. URL: https://stepik.org/course/187/promo (дата обращения: 25.06.2025).

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД

Main.java:

```
import javax.swing.SwingUtilities;
import src.logic.GraphApp;

public class Main {
    public static void main(String[] args) {
        SwingUtilities.invokeLater(GraphApp::new);
    }
}
```

GraphPanel.java:

```
package src.gui;

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import javax.swing.*;
import src.logic.Kruskal;
import src.logic.State;
import static java.lang.Math.min;

public class GraphPanel extends JPanel {
    private Vertex draggedVertex = null;
    private Point offset = new Point();
    private boolean vertexMoved = false;
    private boolean algorithmRunning = false; // Флаг выполнения алгоритма

    public static class Vertex {
        public int x, y, radius = 10;
        public int label;
        Vertex(int x, int y, int label) {
            this.x = x;
            this.y = y;
            this.label = label;
        }
        boolean contains(int mx, int my) {
            return (mx - x)*(mx - x) + (my - y)*(my - y) <= radius * radius;
        }
    }

    public static class Edge implements Comparable<Edge> {
        public Vertex v1, v2;
        public int weight;

        Edge(Vertex v1, Vertex v2, int weight) {
            this.v1 = v1;
            this.v2 = v2;
            this.weight = weight;
        }

        boolean connects(Vertex v) {
            return v1 == v || v2 == v;
        }

        @Override
```

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

    // Метод для нахождения точки на ребре.
    boolean containsPoint(int x, int y, int threshold) {
        // Проверяем расстояние от точки до линии ребра
        int dx = v2.x - v1.x;
        int dy = v2.y - v1.y;
        int lengthSquared = dx*dx + dy*dy;

        // Если длина ребра 0 (вершины совпадают)
        if (lengthSquared == 0) {
            return (v1.x - x)*(v1.x - x) + (v1.y - y)*(v1.y - y) <= threshold*threshold;
        }

        // Вычисляем проекцию точки на линию ребра
        double t = ((double)((x - v1.x)*dx + (y - v1.y)*dy)) / lengthSquared;
        t = Math.max(0, min(1, t)); // Ограничиваем проекцию отрезком

        // Ближайшая точка на ребре
        int projX = (int)(v1.x + t*dx);
        int projY = (int)(v1.y + t*dy);

        // Квадрат расстояния до проекции
        int distSquared = (projX - x)*(projX - x) + (projY - y)*(projY - y);
        return distSquared <= threshold*threshold;
    }
}

private java.util.ArrayList<Vertex> vertices = new ArrayList<>();
private java.util.ArrayList<Edge> edges = new ArrayList<>();
private Vertex selectedVertex = null;
private int vertexCounter = 0;
private JTextArea logArea;

private java.util.List<State> algorithmSteps = new ArrayList<>();
private int currentStep = -1;
private java.util.List<Edge> shownEdges = new ArrayList<>();
private java.util.List<Edge> excludedEdges = new ArrayList<>();
private java.util.List<Edge> cycleEdges = new ArrayList<>();

private TreeSet<Integer> freeLabels = new TreeSet<>();
private int labelCounter = 1;

public GraphPanel(JTextArea logArea) {
    this.logArea = logArea;
    setBackground(Color.WHITE);
    setPreferredSize(new Dimension(800, 600));

    addMouseListener(new MouseAdapter() {
        public void mousePressed(MouseEvent e) {
            Vertex clicked = getVertexAt(e.getX(), e.getY());

            if (SwingUtilities.isLeftMouseButton(e)) {
                if (clicked == null) {
                    // Добавление новой вершины
                    if (algorithmRunning && !confirmAlgorithmInterruption()) return;
                    shownEdges.clear();
                    int label = getNextLabel();
                    vertices.add(new Vertex(e.getX(), e.getY(), label));
                    selectedVertex = null;
```

```
            algorithmSteps.clear();
        } else {
            if (selectedVertex == null) {
                selectedVertex = clicked;
            } else {
                if (selectedVertex != clicked) {
                    if (!vertexMoved) {
                        // Добавление нового ребра
                        if (algorithmRunning && !confirmAlgorithmInterruption()) {
                            selectedVertex = null;
                            return;
                        }
                        shownEdges.clear();
                        String input = JOptionPane.showInputDialog("Введите вес ребра:");
                        if (input != null && !input.trim().isEmpty()) {
                            try {
                                int weight = Integer.parseInt(input.trim());
                                edges.add(new Edge(selectedVertex, clicked, weight));
                            } catch (NumberFormatException ex) {
                                JOptionPane.showMessageDialog(null, "Вес должен быть числом");
                            }
                        }
                        selectedVertex = null;
                        algorithmSteps.clear();
                    } else {
                        selectedVertex = clicked;
                    }
                }
                //selectedVertex = null;
            }

            draggedVertex = clicked;
            if (draggedVertex != null) {
                offset.x = e.getX() - draggedVertex.x;
                offset.y = e.getY() - draggedVertex.y;
            }
        }
    } else if (SwingUtilities.isRightMouseButton(e)) {
        if (clicked != null) {
            // Удаление вершины
            if (algorithmRunning && !confirmAlgorithmInterruption()) return;
            cycleEdges.clear();
            excludedEdges.clear();
            shownEdges.clear();
            edges.removeIf(edge -> edge.connects(clicked));
            vertices.remove(clicked);
            freeLabels.add(clicked.label);
            selectedVertex = null;
            algorithmSteps.clear();
        } else {
            // Проверяем клик на ребре
            Edge edge = getEdgeAt(e.getX(), e.getY());
            if (edge != null) {
                // Удаление ребра
                if (algorithmRunning && !confirmAlgorithmInterruption()) return;
                cycleEdges.clear();
                excludedEdges.clear();
                shownEdges.clear();
                edges.remove(edge);
                selectedVertex = null;
                algorithmSteps.clear();
            }
```

```
                }
            }

            vertexMoved = false;
            repaint();
        }

        public void mouseReleased(MouseEvent e) {
            draggedVertex = null;
        }
    });

    addMouseMotionListener(new MouseMotionAdapter() {
        public void mouseDragged(MouseEvent e) {
            if (draggedVertex != null) {
                draggedVertex.x = e.getX() - offset.x;
                draggedVertex.y = e.getY() - offset.y;
                vertexMoved = true;
                repaint();
            }
        }
    });
}

/**Функция, подтверждающая остановку алгоритма.
 * @return Возвращает true если алгоритм был прерван. Иначе false.
 */
private boolean confirmAlgorithmInterruption() {
    Object[] options = {"Да", "Нет"};
    int response = JOptionPane.showOptionDialog(
            this,
            "Алгоритм выполняется. Прервать и продолжить редактирование?",
            "Подтверждение прерывания",
            JOptionPane.YES_NO_OPTION,
            JOptionPane.WARNING_MESSAGE,
            null,
            options,  // Массив с текстом кнопок
            options[0] // Первая кнопка по умолчанию
    );
    if (response == JOptionPane.YES_OPTION) {
        algorithmRunning = false;
        algorithmSteps.clear();
        return true;
    }
    return false;
}

/**Поиск ребра по координатам.
 * @param x координата x на полотне.
 * @param y координата y
 * @return Ребро находящееся по этим координатам, если ребра нет, то возвращает null.
 */
private Edge getEdgeAt(int x, int y) {
    for (Edge edge : edges) {
        if (edge.containsPoint(x, y, 5)) { // Порог 5 пикселей
            return edge;
        }
    }
    return null;
}

/**Генерация случайного графа по количеству вершин и ребер.
```

```
 * @param numVert количество вершин генерируемого графа.
 * @param numEdges количество ребер генерируемого графа.
 */
public void generateRandomGraph(int numVert, int numEdges) {
    if (algorithmRunning && !confirmAlgorithmInterruption()) return;

    shownEdges.clear();
    freeLabels.clear();
    vertices.clear();
    edges.clear();
    excludedEdges.clear();
    cycleEdges.clear();
    vertexCounter = 0;
    labelCounter = 1;

    Random rand = new Random();

    // 1. Создаём вершины в случайных позициях
    for (int i = 0; i < numVert; i++) {
        int x = 100 + rand.nextInt(600);
        int y = 100 + rand.nextInt(400);
        vertices.add(new Vertex(x, y, getNextLabel()));
    }

    // 2. Строим связное остовное дерево
    ArrayList<Integer> order = new ArrayList<>();
    for (int i = 0; i < min(numEdges, numVert); i++) order.add(i);
    Collections.shuffle(order, rand);

    Set<String> edgeSet = new HashSet<>();

    for (int i = 1; i < min(numEdges, numVert); i++) {
        int v1 = order.get(i);
        int v2 = order.get(rand.nextInt(i));
        int weight = 1 + rand.nextInt(20);
        Vertex a = vertices.get(v1);
        Vertex b = vertices.get(v2);
        edges.add(new Edge(a, b, weight));
        edgeSet.add(min(v1, v2) + "-" + Math.max(v1, v2));
    }

    // 3. Добавляем случайные рёбра до нужного числа
    int attempts = 0;
    while (edges.size() < numEdges && attempts < numEdges * 10) {
        int i = rand.nextInt(numVert);
        int j = rand.nextInt(numVert);
        if (i == j) {
            attempts++;
            continue;
        }
        String key = min(i, j) + "-" + Math.max(i, j);
        if (edgeSet.contains(key)) {
            attempts++;
            continue;
        }
        int weight = 1 + rand.nextInt(20);
        edges.add(new Edge(vertices.get(i), vertices.get(j), weight));
        edgeSet.add(key);
    }

    repaint();
}
```

```
    /**Поиск вершины по координатам.
     * @param x координата x на полотне.
     * @param y координата y на полотне.
     * @return Вершина находящаяся по этим координатам, если вершины нет, то возвращает null.
     */
    private Vertex getVertexAt(int x, int y) {
        for (Vertex v : vertices) {
            if (v.contains(x, y)) return v;
        }
        return null;
    }

    /**Находит первый свободный индекс для вершины.
     * @return Первый свободный индекс, если некоторая вершина была удалена, то ее индекс берется из
freeLabels.
     */
    private int getNextLabel() {
        if (!freeLabels.isEmpty()) {
            return freeLabels.pollFirst();
        }
        return labelCounter++;
    }

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

        for (Edge edge : edges) {
            g.setColor(Color.BLACK);
            g.drawLine(edge.v1.x, edge.v1.y, edge.v2.x, edge.v2.y);
            int midX = (edge.v1.x + edge.v2.x) / 2;
            int midY = (edge.v1.y + edge.v2.y) / 2;
            g.drawString(Integer.toString(edge.weight), midX, midY);
        }

        if (!shownEdges.isEmpty()) {
            for (Edge edge : shownEdges) {
                g.setColor(Color.GREEN);
                g.drawLine(edge.v1.x, edge.v1.y, edge.v2.x, edge.v2.y);
                int midX = (edge.v1.x + edge.v2.x) / 2;
                int midY = (edge.v1.y + edge.v2.y) / 2;
                g.setColor(Color.BLACK);
                g.drawString(Integer.toString(edge.weight), midX, midY);
            }
        }

        if (!cycleEdges.isEmpty()) {
            for (Edge edge : cycleEdges) {
                g.setColor(Color.PINK);
                g.drawLine(edge.v1.x, edge.v1.y, edge.v2.x, edge.v2.y);
                int midX = (edge.v1.x + edge.v2.x) / 2;
                int midY = (edge.v1.y + edge.v2.y) / 2;
                g.setColor(Color.BLACK);
                g.drawString(Integer.toString(edge.weight), midX, midY);
            }
        }

        if (!excludedEdges.isEmpty()){
            for (Edge edge : excludedEdges) {
                g.setColor(Color.RED.darker());
                g.drawLine(edge.v1.x, edge.v1.y, edge.v2.x, edge.v2.y);
```

```
                int midX = (edge.v1.x + edge.v2.x) / 2;
                int midY = (edge.v1.y + edge.v2.y) / 2;
                g.setColor(Color.BLACK);
                g.drawString(Integer.toString(edge.weight), midX, midY);
            }
        }

        for (Vertex v : vertices) {
            g.setColor(Color.WHITE);
            g.fillOval(v.x - v.radius, v.y - v.radius, v.radius * 2, v.radius * 2);
            g.setColor(Color.BLACK);
            g.drawOval(v.x - v.radius, v.y - v.radius, v.radius * 2, v.radius * 2);
            g.drawString(Integer.toString(v.label), v.x - 7, v.y + 5);
        }
    }

    /**Метод для прыжка сразу на последний шаг алгоритма и вывод результата на интерфейс.
     */
    public void runAlgorithmResult() {
        algorithmRunning = true;
        algorithmSteps.clear();
        currentStep = -1;
        runAlgorithm();
        step(algorithmSteps.size());
        if (!algorithmSteps.isEmpty()) {
            State lastState = algorithmSteps.get(algorithmSteps.size() - 1);
            shownEdges = new ArrayList<>(lastState.getIncludedEdges());
            excludedEdges = new ArrayList<>(lastState.getExcludedEdges());
            cycleEdges.clear();
            logArea.setText("Минимальное остовное дерево построено.\n");
            repaint();
        }
    }

    /**Метод для начала прохода по алгоритму,
     * вызывает алгоритм и запоминает все его шаги для последующей визуализации.
     */
    public void runAlgorithm() {
        algorithmRunning = true;
        algorithmSteps.clear();
        currentStep = -1;

        Kruskal kruskal = new Kruskal(edges, vertices.size());
        kruskal.computeMST();

        algorithmSteps.add(new State(
                Collections.emptyList(),
                Collections.emptyList(),
                0,
                null,
                false,
                Collections.emptyList()
        ));
        algorithmSteps.addAll(kruskal.getStates());

        step(1);
    }

    /**Метод для перехода между шагами алгоритма. Визуализирует шаги и выводит информацию о шаге в текстовом виде.
     * @param delta величина шага между состояниями алгоритма. Шаг всегда вперед.
     */
```

```
public void step(int delta) {
    int previousStep = currentStep;

    currentStep += delta;

    if (algorithmSteps.size() == 0) {
        return;
    }
    if (currentStep < 0) {
        currentStep = 0;
    } else if (currentStep > algorithmSteps.size()) {
        currentStep = algorithmSteps.size();
    }

    if (currentStep == previousStep) {
        return;
    }

    if (currentStep == algorithmSteps.size() && currentStep > 0) {
        if (!algorithmSteps.isEmpty()) {
            State lastState = algorithmSteps.get(algorithmSteps.size() - 1);
            shownEdges = new ArrayList<>(lastState.getIncludedEdges());
            excludedEdges = new ArrayList<>(lastState.getExcludedEdges());
        }
        cycleEdges.clear();
        logArea.setText("Минимальное остовное дерево построено.\n");
        repaint();
        return;
    }

    State currentState = algorithmSteps.get(currentStep);
    if (currentState.getCurrentEdge() == null) {
        shownEdges.clear();
        excludedEdges.clear();
        cycleEdges.clear();
        logArea.setText("Начало алгоритма\n");
    } else {
        shownEdges = new ArrayList<>(currentState.getIncludedEdges());
        excludedEdges = new ArrayList<>(currentState.getExcludedEdges());
        cycleEdges = new ArrayList<>(currentState.getCycleEdges());
        logArea.setText(currentState.toString());
    }
    repaint();
}

/**Загрузка графа из файла, точнее матрицы смежности из файла.
 * @param file файл из которого будет считана матрица смежности.
 */
public void loadFromFile(File file) {
    if (algorithmRunning && !confirmAlgorithmInterruption()) return;

    shownEdges.clear();
    excludedEdges.clear();
    cycleEdges.clear();
    vertexCounter = 0;
    labelCounter = 1;
    freeLabels.clear();

    try (BufferedReader br = new BufferedReader(new FileReader(file))) {
        vertices.clear();
        edges.clear();
        vertexCounter = 0;
```

```
            ArrayList<String[]> lines = new ArrayList<>();
            String line;
            while ((line = br.readLine()) != null) {
                lines.add(line.trim().split("\\s+"));
            }
            int n = lines.size();

            int centerX = getWidth() / 2;
            int centerY = getHeight() / 2;
            int radius = min(getWidth(), getHeight()) / 3;
            for (int i = 0; i < n; i++) {
                double angle = 2 * Math.PI * i / n;
                int x = centerX + (int)(radius * Math.cos(angle));
                int y = centerY + (int)(radius * Math.sin(angle));
                vertices.add(new Vertex(x, y, getNextLabel()));
            }

            for (int i = 0; i < n; i++) {
                for (int j = i + 1; j < n; j++) {
                    String val = lines.get(i)[j];
                    if (!val.equals("0")) {
                        try {
                            int weight = Integer.parseInt(val);
                            edges.add(new Edge(vertices.get(i), vertices.get(j), weight));
                        } catch (NumberFormatException e) {
                            JOptionPane.showMessageDialog(this, "Неверный формат веса: " + val);
                        }
                    }
                }
            }
            repaint();
        } catch (IOException e) {
            JOptionPane.showMessageDialog(this, "Ошибка чтения файла");
        }
    }

    /**Возвращает номер текущего шага алгоритма.
     */
    public int getCurrentStep() {
        return currentStep;
    }

    /**Ставит значение флага algorithmRunning на нужное.
     * @param algorithmRunning состояние которое нужно поставить.
     */
    public void setAlgorithmRunning(boolean algorithmRunning) {
        this.algorithmRunning = algorithmRunning;
    }
}
```

GraphApp.java:

```
package src.logic;

import java.awt.*;
import java.io.*;
import javax.swing.*;
import src.gui.GraphPanel;

public class GraphApp extends JFrame {
    private GraphPanel graphPanel;
    private JTextArea logArea;
```

```
private JButton stepBackButton, stepForwardButton;

public GraphApp() {
    setTitle("Алгоритм Краскала");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setSize(1000, 700);
    setResizable(true);
    setLayout(new BorderLayout());

    JPanel topPanel = new JPanel();
    JButton startButton = new JButton("Старт Алгоритма");
    JButton loadButton = new JButton("Загрузка");
    JButton resultButton = new JButton("Результат");

    resultButton.addActionListener(e -> runAlgorithmResult());
    startButton.addActionListener(e -> runAlgorithm());
    loadButton.addActionListener(e -> showLoadOptions());

    topPanel.add(resultButton);
    topPanel.add(startButton);
    topPanel.add(loadButton);

    JPanel bottomPanel = new JPanel();
    stepBackButton = new JButton("<--");
    stepForwardButton = new JButton("-->");
    stepBackButton.addActionListener(e -> stepBack());
    stepForwardButton.addActionListener(e -> stepForward());

    bottomPanel.add(stepBackButton);
    bottomPanel.add(stepForwardButton);

    logArea = new JTextArea();
    logArea.setEditable(false);
    JScrollPane logScroll = new JScrollPane(logArea);
    logScroll.setPreferredSize(new Dimension(300, getHeight()));

    graphPanel = new GraphPanel(logArea);

    add(topPanel, BorderLayout.NORTH);
    add(bottomPanel, BorderLayout.SOUTH);
    add(logScroll, BorderLayout.EAST);
    add(graphPanel, BorderLayout.CENTER);

    setLocationRelativeTo(null);
    setVisible(true);
}

private void showLoadOptions() {
    String[] options = {"Из файла", "Случайная матрица"};
    int choice = JOptionPane.showOptionDialog(
            this,
            "Загрузить граф:",
            "Загрузка",
            JOptionPane.DEFAULT_OPTION,
            JOptionPane.INFORMATION_MESSAGE,
            null,
            options,
            options[0]
    );

    if (choice == 0) {
        JFileChooser fileChooser = new JFileChooser();
```

```
        if (fileChooser.showOpenDialog(this) == JFileChooser.APPROVE_OPTION) {
            File file = fileChooser.getSelectedFile();
            graphPanel.loadFromFile(file);
        }
    } else if (choice == 1) {
        // Собираем панель с двумя полями ввода
        JPanel inputPanel = new JPanel(new GridLayout(2, 2, 5, 5));
        inputPanel.add(new JLabel("Количество вершин:"));
        JTextField vertField = new JTextField();
        inputPanel.add(vertField);
        inputPanel.add(new JLabel("Количество рёбер:"));
        JTextField edgeField = new JTextField();
        inputPanel.add(edgeField);

        int result = JOptionPane.showConfirmDialog(
                this,
                inputPanel,
                "Параметры генерации",
                JOptionPane.OK_CANCEL_OPTION,
                JOptionPane.PLAIN_MESSAGE
        );

        if (result == JOptionPane.OK_OPTION) {
            try {
                int numVert = Integer.parseInt(vertField.getText().trim());
                int numEdges = Integer.parseInt(edgeField.getText().trim());

                if (numVert < 1) {
                    throw new NumberFormatException("Необходимо numVert ≥ 1");
                }
                // Максимальное число рёбер в простом неориентированном графе: n*(n-1)/2
                int maxEdges = numVert * (numVert - 1) / 2;
                if (numEdges < 0 || numEdges > maxEdges) {
                    JOptionPane.showMessageDialog(
                            this,
                            String.format("Число рёбер должно быть в диапазоне [%d .. %d]", 0, maxEdges),
                            "Недопустимое значение",
                            JOptionPane.WARNING_MESSAGE
                    );
                    return;
                }

                graphPanel.generateRandomGraph(numVert, numEdges);
            } catch (NumberFormatException ex) {
                JOptionPane.showMessageDialog(
                        this,
                        "Пожалуйста, введите корректные целые числа.",
                        "Ошибка ввода",
                        JOptionPane.ERROR_MESSAGE
                );
            }
        }
    }
}

private void runAlgorithmResult() {
    graphPanel.runAlgorithmResult();
}

private void runAlgorithm() {
    graphPanel.runAlgorithm();
}
```

```
    private void stepBack() {
        graphPanel.step(-1);
        // Сброс флага при возврате к началу
        if (graphPanel.getCurrentStep() == 0) {
            graphPanel.setAlgorithmRunning(false);
        }
    }

    private void stepForward() {
        graphPanel.step(1);
    }
}
```

Kruskal.java:

```
package src.logic;

import java.util.*;
import src.gui.GraphPanel;
import src.logic.State;

// Класс для поиска пути и детекции цикла
class CycleDetector {
    /**
     * Ищет простой путь между start и target.
     * Возвращает список ребер пути или пустой список если пути нет.
     */
    public static List<GraphPanel.Edge> findPath(
            Collection<GraphPanel.Edge> edges,
            int start,
            int target
    ) {
        // Строим список смежности
        Map<Integer, List<GraphPanel.Edge>> adj = new HashMap<>();
        for (GraphPanel.Edge e : edges) {
            adj.computeIfAbsent(e.v1.label, k -> new ArrayList<>()).add(e);
            adj.computeIfAbsent(e.v2.label, k -> new ArrayList<>()).add(e);
        }

        // DFS для поиска пути
        Deque<Integer> stack = new ArrayDeque<>();
        Map<Integer, GraphPanel.Edge> edgeTo = new HashMap<>();
        Set<Integer> visited = new HashSet<>();

        stack.push(start);
        visited.add(start);

        while (!stack.isEmpty()) {
            int u = stack.pop();
            if (u == target) break;
            for (GraphPanel.Edge e : adj.getOrDefault(u, Collections.emptyList())) {
                int w = (e.v1.label == u ? e.v2.label : e.v1.label);
                if (!visited.contains(w)) {
                    visited.add(w);
                    edgeTo.put(w, e);
                    stack.push(w);
                }
            }
        }

        // Сбор пути
```

```
        List<GraphPanel.Edge> path = new ArrayList<>();
        if (!visited.contains(target)) {
            return path; // пути нет
        }
        Integer cur = target;
        while (cur != null && cur != start) {
            GraphPanel.Edge e = edgeTo.get(cur);
            path.add(e);
            cur = (e.v1.label == cur ? e.v2.label : e.v1.label);
        }
        Collections.reverse(path);
        return path;
    }

    /**
     * Возвращает ребра цикла: найденный путь + новое ребро.
     */
    public static List<GraphPanel.Edge> findCycleEdges(
            Collection<GraphPanel.Edge> includedEdges,
            GraphPanel.Edge newEdge
    ) {
        List<GraphPanel.Edge> path = findPath(includedEdges,
                newEdge.v1.label, newEdge.v2.label);
        if (path.isEmpty()) {
            return Collections.emptyList();
        }
        List<GraphPanel.Edge> cycle = new ArrayList<>(path);
        cycle.add(newEdge);
        return cycle;
    }
}


// Реализация Kruskal
public class Kruskal {

    private final int numVertices;
    private final List<GraphPanel.Edge> edges;
    private final List<State> states;

    public Kruskal(List<GraphPanel.Edge> inputEdges, int numVertices) {
        this.numVertices = numVertices;
        this.edges = new ArrayList<>(inputEdges);
        this.states = new ArrayList<>();
    }

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

    public ArrayList<GraphPanel.Edge> computeMST() {
        Collections.sort(edges);
        ArrayList<GraphPanel.Edge> mst = new ArrayList<>();
        ArrayList<GraphPanel.Edge> excludedEdges = new ArrayList<>();
        int totalWeight = 0;

        for (GraphPanel.Edge edge : edges) {
            // Проверяем: есть ли путь между концами ребра в текущем MST
            List<GraphPanel.Edge> path = CycleDetector.findPath(mst, edge.v1.label, edge.v2.label);
            boolean added = path.isEmpty();
            List<GraphPanel.Edge> cycle = Collections.emptyList();
```

```
            if (added) {
                mst.add(edge);
                totalWeight += edge.weight;
            } else {
                // Документируем цикл
                cycle = CycleDetector.findCycleEdges(mst, edge);
                excludedEdges.add(edge);
                // Если цикл найден, то ребро не добавляем в MST
            }

            // Сохраняем состояние после обработки ребра
            states.add(new State(mst,excludedEdges, totalWeight, edge, added, cycle));
        }
        return mst;
    }

    public List<State> getStates() {
        return states;
    }
}
```

State.java:

```
package src.logic;

import java.util.*;
import src.gui.GraphPanel;

public record State( List<GraphPanel.Edge> includedEdges, List<GraphPanel.Edge> excludedEdges,  int totalWeight,
                     GraphPanel.Edge currentEdge, boolean isIncluded, List<GraphPanel.Edge> cycleEdges) {
    public State(
            List<GraphPanel.Edge> includedEdges,
            List<GraphPanel.Edge> excludedEdges,
            int totalWeight,
            GraphPanel.Edge currentEdge,
            boolean isIncluded,
            List<GraphPanel.Edge> cycleEdges
    ) {
        this.includedEdges = new ArrayList<>(includedEdges);
        this.excludedEdges = new ArrayList<>(excludedEdges);
        this.totalWeight = totalWeight;
        this.currentEdge = currentEdge;
        this.isIncluded = isIncluded;
        this.cycleEdges = new ArrayList<>(cycleEdges);
    }

    @Override
    public String toString() {
        String finalString = "";

        finalString += "Текущее ребро: " + (currentEdge != null ? currentEdge.v1.label + " - " + currentEdge.v2.label :
"None") + "\n";
        finalString += "Включеные ребра: ";
        if (includedEdges.isEmpty()) {
            finalString += "----------\n";
        } else {
            for (GraphPanel.Edge edge : includedEdges) {
                finalString += edge.v1.label + " - " + edge.v2.label + "; ";
            }
            finalString += "\n";
        }
        finalString += "Суммарный вес: " + totalWeight + "\n";
```

```
            if (isIncluded) {
                finalString += "Ребро включено.\n";
            } else {
                finalString += "Ребро не включено.\n";
                finalString += "Ребра цикла: ";
                for (GraphPanel.Edge edge : cycleEdges) {
                    finalString += edge.v1.label + " - " + edge.v2.label + "; ";
                }
            }

            return finalString;
        }

        public List<GraphPanel.Edge> getIncludedEdges() {
            return Collections.unmodifiableList(includedEdges);
        }

        public List<GraphPanel.Edge> getExcludedEdges() {
            return Collections.unmodifiableList(excludedEdges);
        }

        public int getTotalWeight() {
            return totalWeight;
        }

        public GraphPanel.Edge getCurrentEdge() {
            return currentEdge;
        }

        public boolean isIncluded() {
            return isIncluded;
        }

        public List<GraphPanel.Edge> getCycleEdges() {
            return Collections.unmodifiableList(cycleEdges);
        }
    }
```