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. 8303 _____ Крыжановский К.Е.

Студент гр. 8303 _____ Кибардин А.Б.

Студент гр. 8303 _____ Спирин Н.В.

Руководитель _____ Жангиров Т.Р.

Санкт-Петербург

2020

ЗАДАНИЕ

на учебную практику

Студент Крыжановский К.Е. группы 8303

Студент Кибардин А.Б. группы 8303

Студент Спирин Н.В. группы 8303

Тема практики: «Визуализация алгоритма Краскала»

Задание на практику:

Написать программу – визуализатор алгоритма Краскала. Данная программа позволяет пользователю строить граф, добавлять и удалять вершины и рёбра, перемещать вершины. Затем требуется возможность пошагово смотреть, как работает алгоритм Краскала, либо сразу посмотреть на конечный результат построения минимального остовного дерева для построенного графа.

Алгоритм: Алгоритм Краскала.

Дата сдачи отчёта: 10.07.2020

Дата защиты отчёта: 10.07.2020

Студент	_____	Крыжановский К.Е.
Студент	_____	Кибардин А.Б.
Студент	_____	Спирин Н.В.
Руководитель	_____	Жангиров Т.Р.

АННОТАЦИЯ

Целью работы является получение умения работать в команде и закрепление навыков написания кода в стиле объектно-ориентированного программирования на языке Java. Для получения данного опыта выполняется один из вариантов мини-проекта. В процессе выполнения мини-проекта необходимо реализовать графический интерфейс, организовать ввод и вывод данных с его помощью, реализовать сам алгоритм, произвести тестирование для проверки корректности алгоритма и научиться работать в команде. В данной работе в качестве мини-проекта выступает визуализация алгоритма Краскала.

SUMMARY

The aim of the work is to gain the ability to work in a team and to consolidate the skills of writing code in the style of object-oriented programming in Java. To obtain this experience, one of the options of the mini-project is carried out. In the process of implementing a mini-project, it is necessary to implement a graphical interface, organize the input and output of data with its help, implement the algorithm itself, perform testing to verify the correctness of the algorithm and learn how to work in a team. In this work, a mini-project is the visualization of the Kruskal algorithm.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Основная цель практики – реализация мини-проекта, который является визуализацией алгоритма. В данной работе такой алгоритм – алгоритм Краскала, находящий минимальное остовное дерево для построенного графа. Работа над алгоритмом ведётся итеративно, поэтому проект меняется, начиная от прототипа до финальной версии, приобретая с каждой последующей итерацией обновленный функционал.

1. ФУНКЦИОНАЛ ПРОГРАММЫ

1.1. Функционал прототипа

1.1.1. Методы ввода данных

Для корректной работы алгоритма имеется возможность с помощью консоли:

- вводить рёбра исходного графа с заданными весами;

1.1.2 Реализация визуализации

В прототипе отсутствует визуализация, но имеется прототип будущего интерфейса:

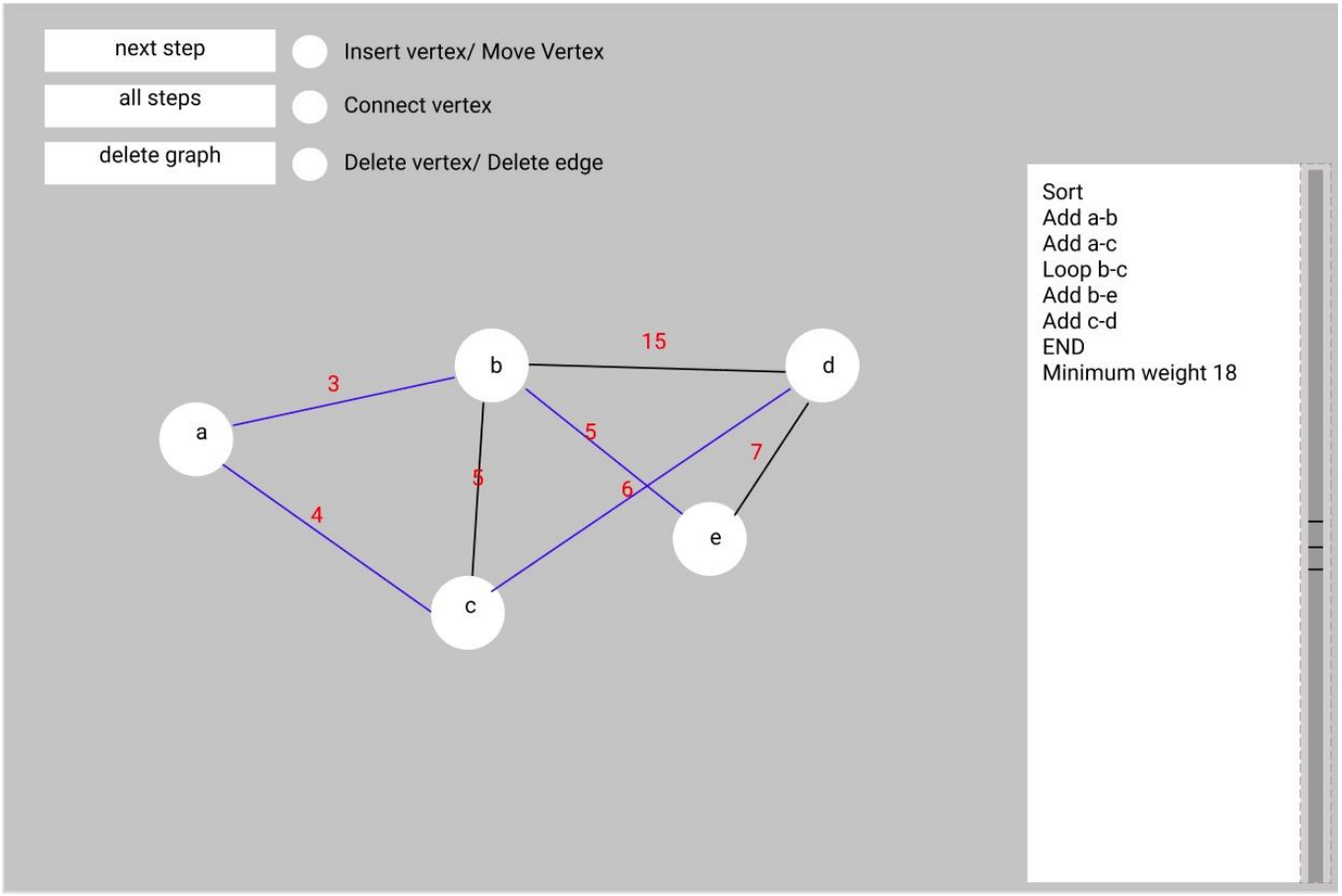


Рисунок 1 - прототип интерфейса

1.1.3 Реализация алгоритма и представление данных

Реализованный алгоритм Краскала сортирует рёбра введенного графа, а затем, начиная с рёбер с минимальным весом, стягивает их, формируя

минимальное остовное дерево. В случае образования цикла, ребро, приводящее к зацикливанию, не добавляется в минимальное остовное дерево, а алгоритм продолжает свою работу.

Граф хранится с помощью списка вершин и списка рёбер.

1.1.4 Представление выходных данных

Выходными данными программы является рёбра с их весами полученного минимального остовного дерева.

1.2. Функционал промежуточной версии программы

1.2.1. Методы управления программой

Для корректной работы алгоритма имеется возможность с помощью графического интерфейса:

- добавлять вершины;
- стягивать вершины рёбрами с заданием их веса;
- перемещать вершины;
- удалять граф;
- визуализировать алгоритм (без шагов);

Кнопка “Следующий шаг” пока не реализована.

1.2.2 Реализация визуализации

В промежуточной версии появился интерфейс, обладающий урезанным функционалом по отношению к финальной версии.

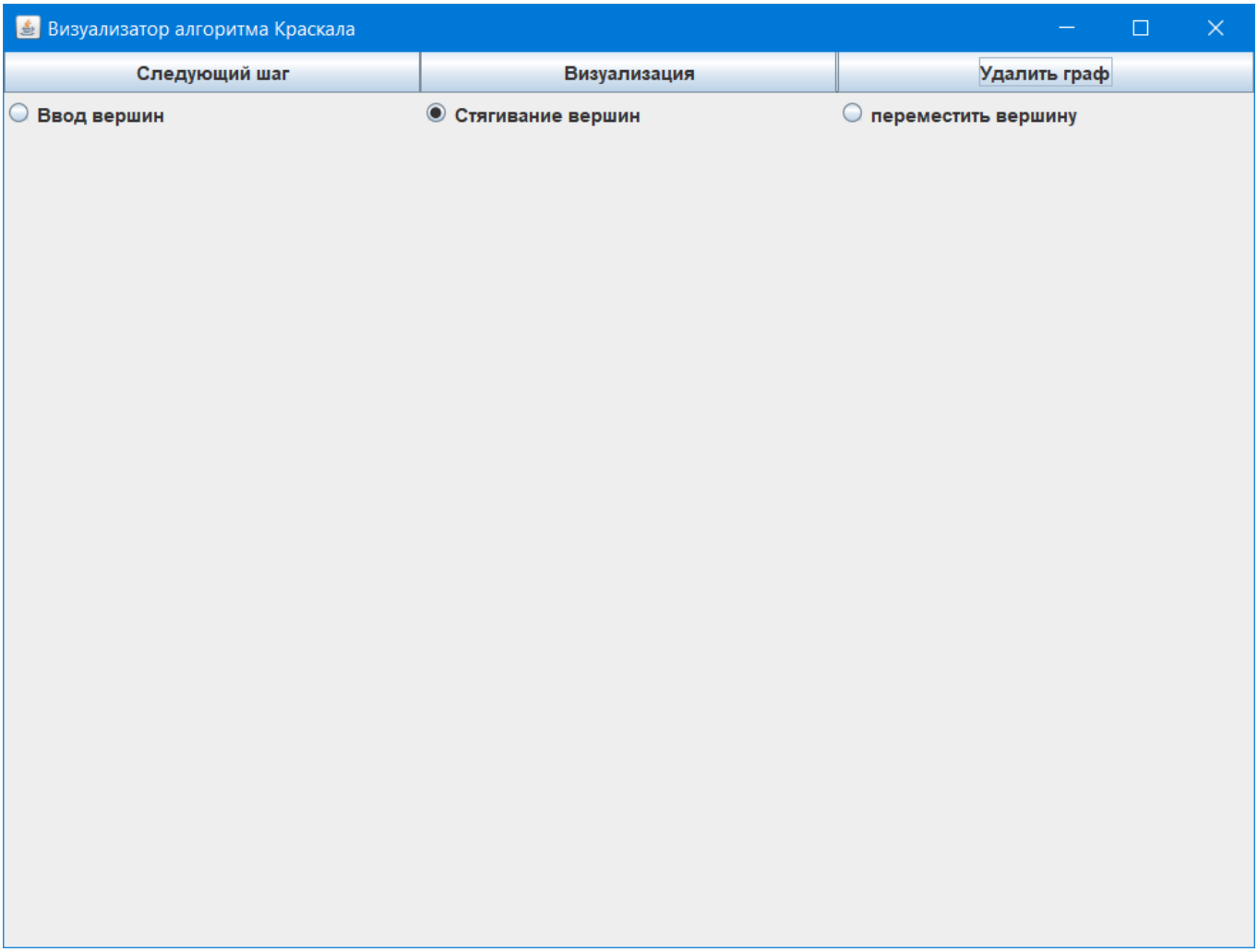


Рисунок 2 – интерфейс промежуточной версии

1.2.3 Представление выходных данных

С помощью нажатия кнопки “Визуализация” запускается ранее описанный алгоритм Краскала, а в построенном графе синим цветом выделяются рёбра, принадлежащие минимальному остовному дереву.

1.3. Функционал финальной версии программы

1.3.1. Методы управления программой

Для корректной работы алгоритма имеется возможность с помощью графического интерфейса:

- добавлять вершины;

- стягивать вершины рёбрами с заданием их веса;
- перемещать вершины;
- удалить граф;
- удалить ребро/вершину;
- визуализировать алгоритм (без шагов);
- визуализировать следующий шаг алгоритма;

Кнопка “Следующий шаг” пока не реализована.

1.3.2 Реализация визуализации

В финальной версии в программе появился простой и удобный интерфейс.

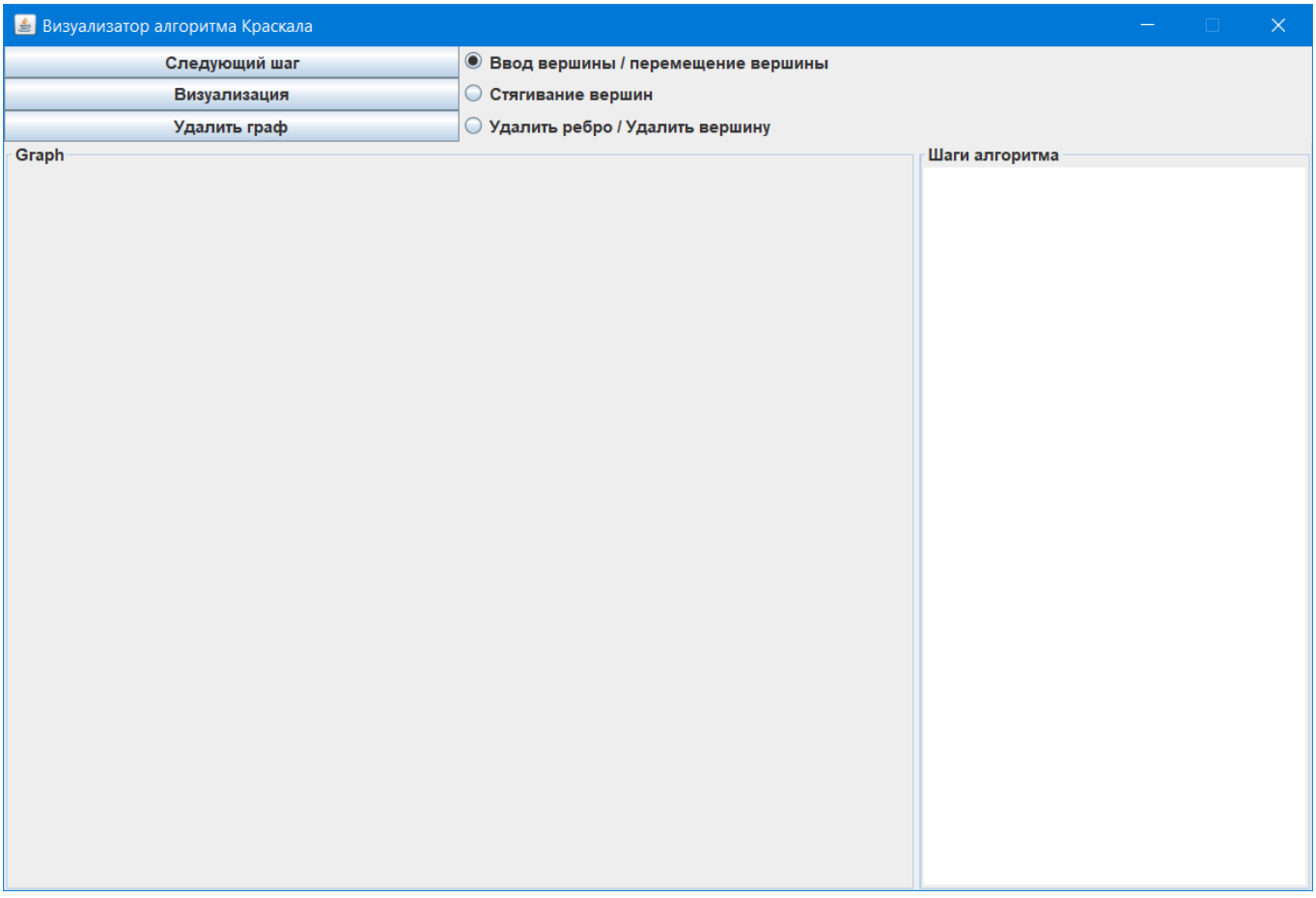


Рисунок 3 – интерфейс финальной версии

1.3.3 Представление выходных данных

С помощью нажатия кнопки “Визуализация” запускается алгоритм Краскала,

а в построенном графе синим цветом выделяются рёбра, принадлежащие минимальному остовному дереву. С помощью кнопки “Следующий шаг” алгоритм переходит к следующему шагу алгоритма. При этом справа в окне программы появилось поле для логирования действий алгоритма.

2. ПЛАН РАЗРАБОТКИ И РАСПРЕДЕЛЕНИЕ РОЛЕЙ В БРИГАДЕ

2.1. План разработки

1. Обсуждение задания, распределение ролей, выбор необходимых средств разработки и структур данных. Данный пункт необходимо выполнить к 1 июля 2020 года.

2. Реализация структур данных. Данный пункт необходимо выполнить к 2 июля 2020 года.

3. Реализация алгоритма Краскала. Данный пункт необходимо выполнить к 3 июля 2020 года.

4. Реализация прототипа GUI. Данный пункт необходимо выполнить к 4 июля 2020 года.

5. Реализация графического ввода графа. Данный пункт необходимо выполнить к 6 июля 2020 года.

6. Реализация основного GUI. Данный пункт необходимо выполнить к 7 июля 2020 года.

7. Реализация дополнительного функционала GUI. Данный пункт необходимо выполнить к 9 июля 2020 года.

8. Тестирование. Данный пункт необходимо выполнить к 10 июля 2020 года.

Распределение ролей в бригаде

- Крыжановский Кирилл, гр. 8303
    - разработка алгоритма
    - gui
- Кибардин Антон, гр. 8303
    - расширение возможностей gui
    - оптимизирование алгоритмов

- Спирин Никита, гр. 8303
    - тестирование
    - слияние наработок

3. ОСОБЕННОСТИ РЕАЛИЗАЦИИ

3.1. Использованные структуры данных

Для реализации проекта потребовалось разработать следующие структуры данных:

Класс Node, который хранит информацию о вершине:

- Поля:
    - private final char name – имя вершины
- Конструкторы:
    - public Node(char name) – инициализирует имя вершины
- Методы:
    - public char getName() – получить имя вершины

Класс Edge, который хранит информацию о ребре:

- Поля:
    - public Node from – вершина, из которой выходит ребро;
    - public Node to – вершина, в которую входит ребро;
    - public int distance – вес ребра между вершинами from и to
- Конструкторы:
    - public Edge(Node from, Node to, int distance) – инициализирует ребро двумя вершинами и весом
- Методы:
    - public boolean equalsWay(Edge other) – проверка на то, что ребро other совпадает с ребром текущим(например, ребро ab то же, что и ba)

Класс Graph, который хранит информацию о графе:

- Поля:
  - private List<Edge> inputEdges – список рёбер графа;
  - private List<Node> inputVertices – список вершин графа;
  - public boolean isModified – переменная, говорящая о том, был модифицирован граф или нет
- Конструкторы:
  - public Graph() – инициализирует пустые списки рёбер и вершин
- Методы:
  - public List<Edge> getInputEdges() – возвращает список рёбер графа;
  - public List<Node> getInputVertices() – возвращает список вершин графа;
  - public void initGraph(List<Edge> inputEdges, List<Node> inputVertices) – инициализирует пустые списки рёбер и вершин;
  - public void addEdge(Node from, Node to, int distance) – добавляет в граф ребро с концами в вершинах from и to и весом distance;
  - public void addVertex(Node node) – добавляет в граф вершину node;
  - public void addVertex() – добавляет в граф вершину с именем, следующим в английском алфавите после последней вершины в графе;
  - public void clear() – удаляет все рёбра и вершины в графе

3.2. Основные методы

Основные методы для работы алгоритма Краскала были реализованы в классе Kruskal:

- public List<Object> nextStep() – определяет что необходимо сделать на следующем шаге алгоритма и в зависимости от выбранного действия и состояния алгоритма возвращает список объектов, идентифицирующих их;
- public static String addStepInfo(List<Object> tuple) – в качестве аргумента принимает список объектов, полученный в nextStep() и преобразует их в строку;
- public String kruskal() – в цикле формирует ответ в виде строки выполняемых действий и состояний алгоритма используя методы nextStep() и addStepInfo(List<Object> tuple) пока минимальное остовное дерево не будет построено;

4. ТЕСТИРОВАНИЕ

4.1. Написание юнит-тестов

Юнит-тесты были написаны с помощью библиотеки JUnit с целью покрыть основные методы кода алгоритма для того, чтобы убедиться в корректности его работы.

Были написаны тесты для нескольких нетривиальных методов класса Kruskal: sortEdges(), nextStep() и kruskal(), а также для метода equals() из класса Edge. Тесты покрывают различные ситуации: от запуска алгоритма при отсутствии графа до запуска алгоритма для конфигурации графа с 40 рёбрами.

Получаемые в результате работы алгоритма списки рёбер сравниваются с теми, что ожидаются при соответствующих вводных данных, с помощью функции equals() класса Edge, которая вызывается в функции assertEquals от двух параметров: списка ожидаемых и полученных в результате алгоритма рёбер. В случае неравенства этих списков проверяемый тест будет провален.

Юнит-тесты к проекту представлены в Приложении Б.

4.2 Ручное тестирование программы

Ручное тестирование кода проводилось для выявления слабых мест программы, непокрытых юнит-тестами. Проводились тесты графического интерфейса на непредсказуемое и иногда приводящее к ошибкам поведение пользователя. В результате ручного тестирования были обнаружены многие проблемы программы, которые затем были успешно разрешены. Пример ручного тестирования представлен на рисунке 4.

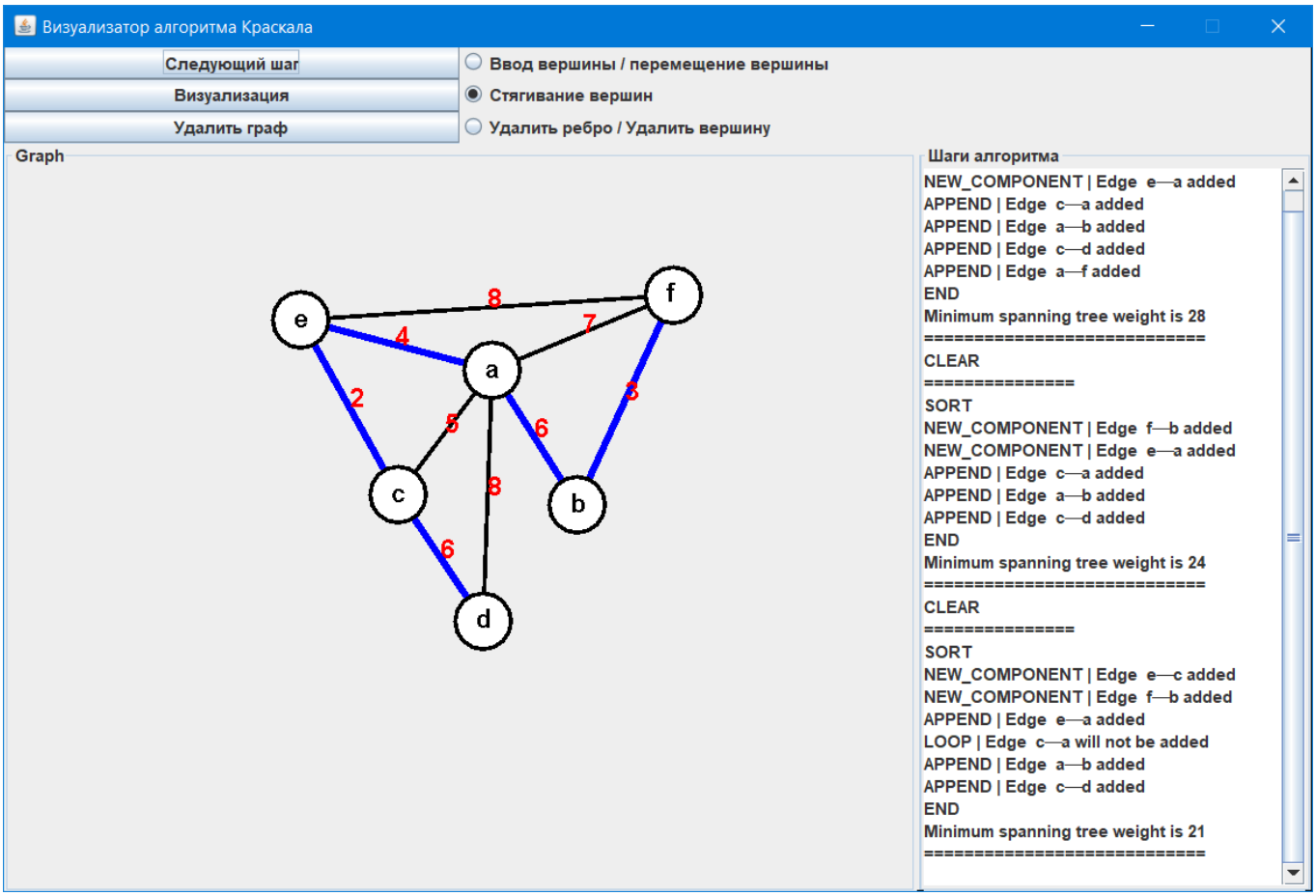


Рисунок 4 – Ручное тестирование

ЗАКЛЮЧЕНИЕ

Разработка поставленной задачи была выполнена в соответствии с планом. Было спроектировано и запрограммировано приложение визуализации алгоритма Краскала. Основной алгоритм, логика которого находится в классе Kruskal, был покрыт юнит-тестами, а графический интерфейс был оттестирован вручную. Поставленные задачи были выполнены полностью.

Таким образом разработка приложения была завершена успешно с полным выполнением плана и реализацией дополнительного функционала.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. ГОСТ 7.32-2017 Система стандартов по информации, библиотечному и издательскому делу. Отчет о научно-исследовательской работе. Структура и правила оформления

2. Официальная документация к Java: https://docs.oracle.com/en/java/javase/

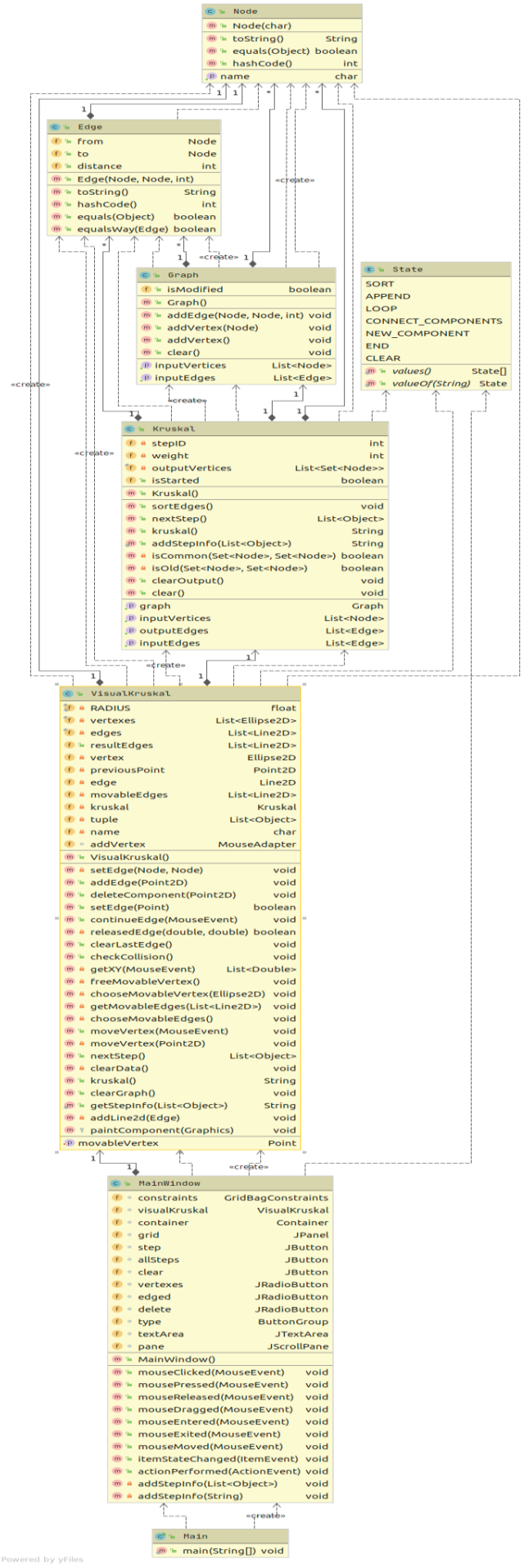
3. Учебный курс по основам Java на Stepik: https://stepik.org/course/187/

4. Википедия: https://ru.wikipedia.org

5. https://ru.stackoverflow.com/

6. https://habr.com/ru/

ПРИЛОЖЕНИЕ А. UML ДИАГРАММА



ПРИЛОЖЕНИЕ Б. Исходный код

Файл Main.java

```
package ru.etu.practice;

public class Main {
    public static void main(String[] args) {
        MainWindow mainWindow = new MainWindow();
    }
}
```

Файл MainWindow.java

```
package ru.etu.practice;

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

public class MainWindow extends JFrame
        implements MouseListener, MouseMotionListener, ItemListener, ActionListener {

    GridBagConstraints constraints = new GridBagConstraints();

    VisualKruskal visualKruskal = new VisualKruskal();

    Container container = getContentPane();

    JPanel grid = new JPanel(new GridLayout(3, 2));

    JButton step = new JButton("Следующий шаг");
    JButton allSteps = new JButton("Визуализация");
    JButton clear = new JButton("Удалить граф");

    JRadioButton vertexes = new JRadioButton("Ввод вершины / перемещение вершины", true);
    JRadioButton edged = new JRadioButton("Стягивание вершин");
    JRadioButton delete = new JRadioButton("Удалить ребро / Удалить вершину");

    ButtonGroup type = new ButtonGroup();

    JTextArea textArea = new JTextArea(10, 50);
    JScrollPane pane = new JScrollPane(textArea);

    public MainWindow() {
        super("Визуализатор алгоритма Краскала");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(950, 640);
```

```
        setResizable(false);
        setVisible(true);

        pane.setPreferredSize(new Dimension(150, 200));
        pane.setBorder(BorderFactory.createTitledBorder("Шаги алгоритма"));
        textArea.setLineWrap(true);
        textArea.setWrapStyleWord(true);
        textArea.setEditable(false);
        pane.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED);
        grid.setPreferredSize(new Dimension(200, 200));

        container.setComponentOrientation(ComponentOrientation.LEFT_TO_RIGHT);

        container.setLayout(new GridBagLayout());
        constraints.fill = GridBagConstraints.HORIZONTAL;
        constraints.anchor = GridBagConstraints.NORTH;
        constraints.weightx = 0.5;
        constraints.weighty = 0.5;
//        constraints.ipady = 200;
//        constraints.ipadx = 200;
        constraints.gridheight = 3;
        constraints.gridwidth = 3;

        constraints.gridx = 0;  // нулевая ячейка таблицы по вертикали
        constraints.gridy = 0;      // нулевая ячейка таблицы по горизонтали
        container.add(grid, constraints);

        constraints.fill = GridBagConstraints.BOTH;
        constraints.gridx = 0;
        constraints.gridy = 3;
        constraints.ipady = 500;
        constraints.ipadx = 800;
        constraints.gridheight = 4;
        constraints.gridwidth = 4;
        container.add(visualKruskal, constraints);

        constraints.fill = GridBagConstraints.BOTH;
        constraints.gridx = 4;
        constraints.gridy = 3;
        constraints.ipadx = 400;
        constraints.gridheight = 4;
        constraints.gridwidth = 1;
        container.add(pane, constraints);
        setLocationRelativeTo(null);

        type.add(vertexes);
        type.add(edged);
        type.add(delete);

        visualKruskal.setPreferredSize(new Dimension(700, 500));
        visualKruskal.setBorder(BorderFactory.createTitledBorder("Graph"));

//
        grid.add(step, BorderLayout.NORTH);
        grid.add(vertexes, BorderLayout.NORTH);
        grid.add(allSteps, BorderLayout.NORTH);
```

```
    grid.add(edged, BorderLayout.NORTH);
    grid.add(clear, BorderLayout.NORTH);

    grid.add(delete, BorderLayout.NORTH);

    vertexes.addItemListener(this);
    edged.addItemListener(this);

    step.addActionListener(this);
    allSteps.addActionListener(this);
    clear.addActionListener(this);
    delete.addActionListener(this);

    visualKruskal.addMouseListener(this);
    visualKruskal.addMouseMotionListener(this);

    Font font = new Font(null, Font.BOLD, 12);
    textArea.setFont(font);
}

@Override
public void mouseClicked(MouseEvent mouseEvent) {
    if (vertexes.isSelected()) {
        visualKruskal.addVertex.mouseClicked(mouseEvent);
    } else if (edged.isSelected()) {

    } else if (delete.isSelected()) {
        visualKruskal.deleteComponent(mouseEvent.getPoint());
    } else {
        assert false;
    }
}

@Override
public void mousePressed(MouseEvent mouseEvent) {
    if (edged.isSelected()) {
        boolean hasFound = visualKruskal.setEdge(mouseEvent.getPoint());
        if (!hasFound) {
            JOptionPane.showMessageDialog(
                    this,
                    "Кажется, что Вы не попали в область вершины, попробуйте ещё раз",
                    "Сообщение",
                    JOptionPane.ERROR_MESSAGE
            );
        }
    } else if (vertexes.isSelected()) {
        visualKruskal.setMovableVertex(mouseEvent.getPoint());
    }
}

@Override
public void mouseReleased(MouseEvent mouseEvent) {
    if (edged.isSelected()) {
        visualKruskal.addEdge(mouseEvent.getPoint());
    } else if (vertexes.isSelected()) {
        visualKruskal.checkCollision();
```

```
    }
}


@Override
public void mouseDragged(MouseEvent mouseEvent) {
    if (edged.isSelected()) {
        visualKruskal.continueEdge(mouseEvent);
    } else if (vertexes.isSelected()) {
        visualKruskal.moveVertex(mouseEvent);
    }
}


@Override
public void mouseEntered(MouseEvent mouseEvent) {

}

@Override
public void mouseExited(MouseEvent mouseEvent) {

}

@Override
public void mouseMoved(MouseEvent mouseEvent) {

}

@Override
public void itemStateChanged(ItemEvent itemEvent) {

}

@Override
public void actionPerformed(ActionEvent actionEvent) {
    if (actionEvent.getSource() == allSteps) {
        addStepInfo(visualKruskal.kruskal());
        visualKruskal.repaint();
    } else if (actionEvent.getSource() == clear) {
        visualKruskal.clearGraph();
        addStepInfo(String.valueOf(State.CLEAR));
        addStepInfo("=========================");
    } else if (actionEvent.getSource() == step) {
        List<Object> tuple = visualKruskal.nextStep();
        if (tuple != null) {
            addStepInfo(tuple);
            if(tuple.get(0) == State.END) {
                addStepInfo("=========================");
            }
            if (tuple.get(0) == State.CLEAR) {
                addStepInfo(String.valueOf(State.CLEAR));
                addStepInfo("=========================");
            }
        }
    } else {
```

```
            assert false;
        }
    }

    private void addStepInfo(List<Object> tuple) {
        textArea.setText(textArea.getText() + VisualKruskal.getStepInfo(tuple));
    }

    private void addStepInfo(String msg) {
        textArea.setText(textArea.getText() + msg + "\n");
    }
}
```

Файл VisualKruskal.java

```
package ru.etu.practice;

import javax.swing.*;
import java.awt.*;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.geom.*;
import java.util.LinkedList;
import java.util.List;

public class VisualKruskal extends JComponent {
    private final static float RADIUS = 40f;
    private final List<Ellipse2D> vertexes = new LinkedList<>();
    private final List<Line2D> edges = new LinkedList<>();
    public List<Line2D> resultEdges = new LinkedList<>();

    private Ellipse2D vertex = null;
    private Point2D previousPoint = new Point2D.Double();
    private Line2D edge = null;
    private List<Line2D> movableEdges = new LinkedList<>();
    private Kruskal kruskal;
    private List<Object> tuple;
    private char name;

    public VisualKruskal() {
        super();
        this.kruskal = new Kruskal();
    }

    MouseAdapter addVertex = new MouseAdapter() {
        public void mouseClicked(MouseEvent e) {
            for (Ellipse2D vertexTmp : vertexes) {
                if (vertexTmp.getBounds2D().contains(e.getPoint())) {
                    return;
                }
            }
            name = 'a';
            for (Node tmpChar : kruskal.getInputVertices()) {
                if (name != tmpChar.getName()) {
```

```
                break;
            } else {
                name++;
            }
        }

        if (name > 'z') {
            return;
        }

        kruskal.getInputVertices().add(name - 'a', new Node(name));
        List<Double> xy = getXY(e);

        vertex = new Ellipse2D.Double(
                xy.get(0) - RADIUS / 2,
                xy.get(1) - RADIUS / 2,
                RADIUS,
                RADIUS
        );
        vertexes.add(name - 'a', vertex);
        repaint();
    }
};

private void setEdge(Node from, Node to) {
    String result;
    boolean isFirst = true;
    do {
        if (!isFirst) {
            JOptionPane.showMessageDialog(
                    this,
                    "Нужно ввести положительно число от 1 до 99999999.\n" +
                            "Попробуйте еще раз",
                    "Ошибка",
                    JOptionPane.ERROR_MESSAGE
            );
        }
        result = JOptionPane.showInputDialog(
                this,
                "<html><h2>Введите вес ребра");
        if (result == null) {
            clearLastEdge();
            return;
        }
        isFirst = false;
    } while (!result.matches("\\d{1,8}") || Integer.parseInt(result) == 0);

    int distance = Integer.parseInt(result);
    kruskal.getGraph().addEdge(from, to, distance);

}

public void addEdge(Point2D point2D) {
    Node fromVertex = null, toVertex = null;
    for (Ellipse2D vertex : vertexes) {
        if (vertex.getBounds2D().contains(point2D)) {
```

```
            toVertex = kruskal.getGraph().getInputVertices().get(vertexes.indexOf(vertex));
            if (!releasedEdge(vertex.getBounds().getCenterX(), vertex.getBounds().getCenterY())) {
                clearLastEdge();
                JOptionPane.showMessageDialog(
                        this,
                        "Ввод петель не возможен.",
                        "Сообщение",
                        JOptionPane.WARNING_MESSAGE
                );
                return;
            }
            break;
        }

    }
    for (Ellipse2D vertexFrom : vertexes) {
        if (vertexFrom.getBounds().contains(previousPoint)) {
            fromVertex = kruskal.getGraph().getInputVertices().get(vertexes.indexOf(vertexFrom));

            break;
        }
    }
    if (fromVertex == null || toVertex == null) {
        clearLastEdge();
        JOptionPane.showMessageDialog(
                this,
                "Кажется, что Вы не попали в область вершины, попробуйте ещё раз",
                "Сообщение",
                JOptionPane.ERROR_MESSAGE
        );
        return;
    }

    if (kruskal.getGraph().getInputEdges().size() > 0) {
        Edge currentEdge = new Edge(fromVertex, toVertex, 0);
        for (Edge outEdge : kruskal.getGraph().getInputEdges()) {
            if (currentEdge.equalsWay(outEdge)) {
                clearLastEdge();
                JOptionPane.showMessageDialog(
                        this,
                        "Кажется, что ребро между этими вершинами уже существует",
                        "Сообщение",
                        JOptionPane.ERROR_MESSAGE
                );
                return;
            }
        }
    }
    setEdge(fromVertex, toVertex);
    repaint();
}

public void deleteComponent(Point2D e) {
    for (Ellipse2D vertex : vertexes) {
        if (vertex.contains(e)) {
            final char cr = kruskal.getInputVertices().get(vertexes.indexOf(vertex)).getName();
```

```
                kruskal.getInputVertices().remove(vertexes.indexOf(vertex));
                resultEdges.removeIf(elem -> vertex.contains(elem.getP1()) || vertex.contains(elem.getP2()));
                kruskal.getOutputEdges().removeIf(elem -> elem.from.getName() == cr || elem.to.getName() ==
cr);
                edges.removeIf(elem -> vertex.contains(elem.getP1()) || vertex.contains(elem.getP2()));
                kruskal.getGraph().getInputEdges().removeIf(elem -> elem.from.getName() == cr ||
elem.to.getName() == cr);
                kruskal.getInputEdges().removeIf(elem -> elem.from.getName() == cr || elem.to.getName() == cr);
                vertexes.remove(vertex);
                kruskal.getGraph().isModified = true;
                repaint();
                break;
            }
        }

        for (Line2D edge : edges) {
            if (edge.intersects(e.getX() - 3, e.getY() - 3, 6, 6)) {
                kruskal.getGraph().getInputEdges().remove(edges.indexOf(edge));
                edges.remove(edge);
                kruskal.getGraph().isModified = true;
                repaint();
                break;
            }
        }

        for (Line2D edge : resultEdges) {
            if (edge.intersects(e.getX() - 3, e.getY() - 3, 6, 6)) {
                resultEdges.remove(edge);
                kruskal.getGraph().isModified = true;
                repaint();
                break;
            }
        }

        repaint();
    }

    public void setMovableVertex(Point e) {
        chooseMovableVertex(null);
        for (Ellipse2D vertex : vertexes) {
            if (vertex.getBounds2D().contains(e)) {
                chooseMovableVertex(vertex);
                chooseMovableEdges();
                break;
            }
        }
    }

    public boolean setEdge(Point e) {
        for (Ellipse2D vertex : vertexes) {
            if (vertex.getBounds2D().contains(e)) {
                previousPoint = new Point2D.Double(vertex.getCenterX(), vertex.getCenterY());
                edge = new Line2D.Double(previousPoint, previousPoint);
                edges.add(edge);
                repaint();
                return true;
```

```
            }
        }
        return false;
    }

    public void continueEdge(MouseEvent e) {
        List<Double> xy = getXY(e);
        edge.setLine(edge.getP1(), new Point2D.Double(xy.get(0), xy.get(1)));
        repaint();
    }

    private boolean releasedEdge(double x, double y) {
        Point2D point2D = new Point2D.Double(x, y);
        if (point2D.equals(edge.getP1())) {
            JOptionPane.showMessageDialog(
                    this,
                    "Ввод петель не возможен.",
                    "Сообщение",
                    JOptionPane.WARNING_MESSAGE
            );
            return false;
        }
        for (Ellipse2D vertexTmp : vertexes) {
            if (vertexTmp.contains(point2D)) {
                edge.setLine(edge.getP1(), point2D);
                edge = null;
                repaint();
                return true;
            }
        }
        return false;
    }

    public void clearLastEdge() {
        edges.remove(edges.size() - 1);
        repaint();
    }

    public void checkCollision() {
        if (vertex != null) {
            for (Ellipse2D anotherVertex : vertexes) {
                if (!anotherVertex.equals(vertex) &&
(anotherVertex.getBounds2D().intersects(vertex.getBounds2D()) || anotherVertex.equals(vertex))) {
                    moveVertex(previousPoint);
                    break;
                }
            }
            for (Ellipse2D vertex : vertexes) {
                for (Line2D edge : edges) {
                    if (!(vertex.getBounds2D().contains(edge.getP1()) ||
vertex.getBounds2D().contains(edge.getP2()))) {
                        if (edge.intersects(vertex.getBounds2D())) {
                            moveVertex(previousPoint);
                            break;
                        }
                    }
```

```
            }
        }
        freeMovableVertex();
    }
}

private List<Double> getXY(MouseEvent event) {
    List<Double> xy = new LinkedList<>();
    double x = event.getPoint().getX();
    double y = event.getPoint().getY();

    if (x < 5 + RADIUS / 2) {
        x = 5 + RADIUS / 2;
    } else if (x > 650 - RADIUS / 2) {
        x = 650 - RADIUS / 2;
    }

    if (y < 10 + RADIUS / 2) {
        y = 10 + RADIUS / 2;
    } else if (y > 530 - RADIUS / 2) {
        y = 530 - RADIUS / 2;
    }

    xy.add(x);
    xy.add(y);

    return xy;
}

private void freeMovableVertex() {
    movableEdges.clear();
    vertex = null;
}

private void chooseMovableVertex(Ellipse2D chosenVertex) {
    vertex = chosenVertex;
    if (chosenVertex != null) {
        previousPoint.setLocation(vertex.getCenterX(), vertex.getCenterY());
    }
}

private void getMovableEdges(List<Line2D> edges) {
    for (Line2D line : edges) {
        if (vertex.getBounds2D().contains(line.getP1())) {
            previousPoint = line.getP1();
            line.setLine(line.getP2(), previousPoint);
            movableEdges.add(line);
        } else if (vertex.getBounds2D().contains(line.getP2())) {
            movableEdges.add(line);
        }
    }
}

private void chooseMovableEdges() {
    getMovableEdges(edges);
    getMovableEdges(resultEdges);
```

```
    }

    public void moveVertex(MouseEvent mouseEvent) {
        if (vertex == null)
            return;

        List<Double> xy = getXY(mouseEvent);

        double x = xy.get(0);
        double y = xy.get(1);

        for (Line2D line : movableEdges) {
            line.setLine(line.getP1(), new Point2D.Double(x, y));
        }

        vertex.setFrame(x - RADIUS / 2, y - RADIUS / 2, RADIUS, RADIUS);
        repaint();
    }

    private void moveVertex(Point2D point2D) {
        if (vertex == null)
            return;
        for (Line2D line : movableEdges) {
            line.setLine(line.getP1(), point2D);
        }
        vertex.setFrame(point2D.getX() - RADIUS / 2, point2D.getY() - RADIUS / 2, RADIUS, RADIUS);
        repaint();
    }

    public List<Object> nextStep() {
        if (kruskal.isStarted) {
            if (kruskal.getGraph().isModified) {
                    /*
                    delete result and back to start
                     */
                resultEdges.clear();
                kruskal.clearOutput();
                repaint();
                kruskal.getGraph().isModified = false;
                return nextStep();
            } else {
                    /*
                    next step
                     */
                tuple = kruskal.nextStep();
                if (tuple.get(0) == State.APPEND || tuple.get(0) == State.NEW_COMPONENT || tuple.get(0) ==
State.CONNECT_COMPONENTS) {
                    Edge edge = kruskal.getOutputEdges().get(kruskal.getOutputEdges().size() - 1);
                    addLine2d(edge);
                }
                if (tuple.get(0) == State.END) {
                    kruskal.getGraph().isModified = true;
                }
                repaint();
            }
        } else {
```

```
            /*
            init graph and do first step
             */
        tuple = null;
        kruskal.isStarted = true;
        return nextStep();
    }
    return tuple;
}

private void clearData() {
    kruskal.clearOutput();
    kruskal.isStarted = false;
    kruskal.getGraph().isModified = false;
}

public String kruskal() {
    if (kruskal.getGraph().isModified) {
        clearData();
        tuple = null;
        return kruskal();
    }

    String result = kruskal.kruskal();
    tuple = kruskal.nextStep();
    List<Edge> outEdges = kruskal.getOutputEdges();
    for (Edge edge : outEdges) {
        addLine2d(edge);
    }
    kruskal.getGraph().isModified = true;
    return result;
}

public void clearGraph() {
    kruskal.clear();
    vertexes.clear();
    edges.clear();
    resultEdges.clear();
    repaint();
}

public static String getStepInfo(List<Object> tuple) {
    return Kruskal.addStepInfo(tuple);
}

private void addLine2d(Edge edge) {
    Node from = edge.from;
    Node to = edge.to;
    double x1 = 0, y1 = 0, x2 = 0, y2 = 0;
    for (Ellipse2D vertex : vertexes) {
        Rectangle2D rectangle2D = vertex.getBounds2D();
        double x = rectangle2D.getCenterX();
        double y = rectangle2D.getCenterY();
        if (from == kruskal.getGraph().getInputVertices().get(vertexes.indexOf(vertex))) {
            x1 = x;
            y1 = y;
```

```
            } else if (to == kruskal.getGraph().getInputVertices().get(vertexes.indexOf(vertex))) {
                x2 = x;
                y2 = y;
            }
        }
        Point2D pointFrom = new Point2D.Double(x1, y1);
        Point2D pointTo = new Point2D.Double(x2, y2);
        Line2D line2D = new Line2D.Double(
                pointFrom, pointTo
        );
        resultEdges.add(line2D);
    }

    protected void paintComponent(Graphics g) {
        Graphics2D g2d = (Graphics2D) g;
        g2d.setPaint(Color.BLACK);

        // толщина линии
        g2d.setStroke(new BasicStroke(3));

        // размер шрифта
        g2d.setFont(new Font("TimesNewRoman", Font.BOLD, 18));

        int i = 0;
        int from, to;
        Line2D loop = null;
        if (tuple != null && tuple.size() == 2 && tuple.get(0) == State.LOOP) {
            from = kruskal.getGraph().getInputVertices().indexOf(((Edge) tuple.get(1)).from);
            to = kruskal.getGraph().getInputVertices().indexOf(((Edge) tuple.get(1)).to);
            loop = new Line2D.Double(vertexes.get(to).getBounds().getCenterX(),
                    vertexes.get(to).getBounds().getCenterY(),
                    vertexes.get(from).getBounds().getCenterX(),
                    vertexes.get(from).getBounds().getCenterY());
        }

        for (Line2D edge : edges) {
            if (tuple != null && tuple.get(0) == State.LOOP &&
                    (edge.getP1().equals(loop.getP1()) && edge.getP2().equals(loop.getP2()) ||
edge.getP1().equals(loop.getP2()) && edge.getP2().equals(loop.getP1())))
                g2d.setPaint(Color.RED);
            else
                g2d.setPaint(Color.BLACK);
            g2d.draw(edge);
        }


        // толщина линии
        g2d.setStroke(new BasicStroke(5));
        g2d.setPaint(Color.BLUE);

        for (Line2D edge : resultEdges) {
            g2d.draw(edge);
        }

        if (tuple != null && tuple.size() == 2) {
            if (tuple.get(0) == State.NEW_COMPONENT || tuple.get(0) == State.APPEND) {
```

```
            g2d.setPaint(Color.GREEN);
            g2d.draw(resultEdges.get(resultEdges.size() - 1));
        } else if (tuple.get(0) == State.END && resultEdges.size() > 0) {
            g2d.setPaint(Color.BLUE);
            g2d.draw(resultEdges.get(resultEdges.size() - 1));
        }
    }

    // толщина линии
    g2d.setStroke(new BasicStroke(3));

    // размер шрифта
    g2d.setFont(new Font("TimesNewRoman", Font.BOLD, 18));

    for (Ellipse2D vertex : vertexes) {
        g2d.setPaint(Color.WHITE);
        g2d.fill(vertex);
        g2d.setPaint(Color.BLACK);
        g2d.draw(vertex);
        int x = vertex.getBounds().x;
        int y = vertex.getBounds().y;

        g2d.drawString(String.valueOf(kruskal.getGraph().getInputVertices().get(vertexes.indexOf(vertex))),
x + 15, y + 25);
    }

    for (Line2D edge : edges) {
        if (kruskal.getGraph().getInputEdges().size() > i) {
            Edge outEdge = kruskal.getGraph().getInputEdges().get(i++);
            g2d.setPaint(Color.RED);
            g2d.drawString(
                    String.valueOf(outEdge.distance),
                    (float) (0.5 * (edge.getX1() + edge.getX2())),
                    (float) (0.5 * (edge.getY1() + edge.getY2()))
            );
        }
    }
  }
}
```

Файл Kruskal.java

```
package ru.etu.practice;

import java.util.*;

public class Kruskal {

  private final Graph graph;
  private int stepID = -1;
  private int weight = 0;

  public List<Edge> getInputEdges() {
    return inputEdges;
  }
```

```
private List<Edge> inputEdges;
private List<Node> inputVertices;
private final List<Edge> outputEdges;
private final List<Set<Node>> outputVertices;

public boolean isStarted = false;

public Kruskal() {
    this.graph = new Graph();
    inputEdges = new LinkedList<>(graph.getInputEdges());
    inputVertices = graph.getInputVertices();
    outputEdges = new LinkedList<>();
    outputVertices = new LinkedList<>();
}

public void initKruskal(List<Edge> inputEdges, List<Node> inputVertices) {
    this.inputEdges = inputEdges;
    this.inputVertices = inputVertices;
    this.graph.initGraph(inputEdges, inputVertices);
}

public void sortEdges() {
    inputEdges = new LinkedList<>(graph.getInputEdges());
    inputEdges.sort(new Comparator<Edge>() {
        @Override
        public int compare(Edge edge1, Edge edge2) {
            int distance1 = edge1.distance;
            int distance2 = edge2.distance;
            return Integer.compare(distance1, distance2);
        }
    });
}

public List<Edge> getOutputEdges() {
    return outputEdges;
}

public List<Node> getInputVertices() {
    return inputVertices;
}

public Graph getGraph() {
    return graph;
}

/**
 * tuple.get(0) instanceof State -- return state
 * tuple.get(1) instanceof Edge -- added edge
 *
 * @return tuple
 */
public List<Object> nextStep() {
    List<Object> tuple = new LinkedList<>();
    if (stepID == -1 || !isStarted) {
        isStarted = true;
        graph.isModified = false;
```

```
    sortEdges();
    stepID++;
    tuple.add(State.SORT);
    return tuple;
}

if (stepID >= inputEdges.size() || graph.isModified) {
    isStarted = false;
    tuple.add(State.END);
    tuple.add(weight);
    stepID = -1;
    graph.isModified = true;
    return tuple;
}

Edge edge = inputEdges.get(stepID++);
weight += edge.distance;
Set<Node> tempVertexes = new HashSet<>();
Node vertex1 = edge.from;
Node vertex2 = edge.to;
tempVertexes.add(vertex1);
tempVertexes.add(vertex2);

if (outputVertices.size() > 0) {
    if (outputVertices.get(0).size() == inputVertices.size()) {
        tuple.add(State.END);
        weight -= edge.distance;
        tuple.add(weight);
        stepID = -1;
        graph.isModified = true;
        return tuple;
    }
}

boolean hasFound = false;
for (Set<Node> currently : outputVertices) {
    if (isOld(currently, tempVertexes)) {
        tuple.add(State.LOOP);
        tuple.add(edge);
        weight -= edge.distance;
        return tuple;
    }
    if (isCommon(currently, tempVertexes)) {
        hasFound = true;
        currently.addAll(tempVertexes);
        tuple.add(State.APPEND);
        tuple.add(edge);
    }
}

for (int i = 0; i < outputVertices.size(); i++) {
    for (int j = i + 1; j < outputVertices.size(); j++) {
        Set<Node> first = outputVertices.get(i);
        Set<Node> second = outputVertices.get(j);
        if (isCommon(first, second)) {
            tuple.add(State.CONNECT_COMPONENTS);
```

```
                tuple.add(edge);
                first.addAll(second);
                second.clear();
            }
        }
    }

    outputEdges.add(edge);
    if (!hasFound) {
        tuple.add(State.NEW_COMPONENT);
        tuple.add(edge);
        outputVertices.add(tempVertexes);
    }
    return tuple;
}

public String kruskal() {
    stepID = -1;
    StringBuilder result = new StringBuilder();
    while (true) {
        List<Object> tuple = nextStep();
        result.append(addStepInfo(tuple));
        //System.err.println(tuple.get(0));
        if (tuple.get(0) == State.END) {
            graph.isModified = true;
            break;
        }
    }
    result.append("=========================");
    return result.toString();
}

public static String addStepInfo(List<Object> tuple) {
    State state = (State) tuple.get(0);
    StringBuilder addText = new StringBuilder();
    if (state == State.SORT) {
        addText.append(state);
    } else if (state == State.END) {
        assert tuple.size() > 1;
        addText.append(state);
        addText.append("\n");
        int value = (int) tuple.get(1);
        addText.append("Minimum spanning tree weight is ");
        addText.append(value);
    } else if (state == State.LOOP) {
        assert tuple.size() > 1;
        addText.append(state);
        addText.append(" | ");
        Edge edge = (Edge) tuple.get(1);
        addText.append(edge);
        addText.append(" will not be added");
    } else {
        assert tuple.size() > 1;
        addText.append(state);
        addText.append(" | ");
        Edge edge = (Edge) tuple.get(1);
```

```
            addText.append(edge);
            addText.append(" added");
        }
        return addText.toString() + "\n";
    }

    private boolean isCommon(Set<Node> first, Set<Node> second) {
        Set<Node> all = new HashSet<>();
        all.addAll(first);
        all.addAll(second);
        return all.size() != (first.size() + second.size());
    }

    private boolean isOld(Set<Node> first, Set<Node> second) {
        Set<Node> all = new HashSet<>();
        all.addAll(first);
        all.addAll(second);
        return all.size() == first.size();
    }

    public void clearOutput() {
        outputEdges.clear();
        outputVertices.clear();
        stepID = -1;
        weight = 0;
    }

    public void clear() {
        clearOutput();
        inputVertices.clear();
        inputEdges.clear();
        graph.clear();
    }
}
```

Файл Graph.java

```
package ru.etu.practice;

import java.util.*;

public class Graph {
//    private static final int SIZE = 26;

    private List<Edge> inputEdges;
    private List<Node> inputVertices;
    public boolean isModified = false;

//    public

    public List<Edge> getInputEdges() {
        return inputEdges;
    }

    public List<Node> getInputVertices() {
        return inputVertices;
```

```
    }

    public Graph() {
        this.inputEdges = new LinkedList<>();
        this.inputVertices = new LinkedList<>();
    }

    public void initGraph(List<Edge> inputEdges, List<Node> inputVertices) {
        this.inputEdges = new LinkedList<>(inputEdges);
        this.inputVertices = new LinkedList<>(inputVertices);
    }

    public void addEdge(Node from, Node to, int distance) {
        Edge newEdge = new Edge(from, to, distance);
        for (Edge edge : inputEdges)
            if (edge.equalsWay(newEdge))
                return;
        inputEdges.add(newEdge);
        isModified = true;
    }

    public void addVertex(Node node) {
        if (!inputVertices.contains(node))
            inputVertices.add(node);
        isModified = true;
    }

    public void addVertex() {
        int len = inputVertices.size();
        inputVertices.add(new Node((char)('a' + len)));
    }

    public void clear() {
        inputVertices.clear();
        inputEdges.clear();
        isModified = false;
    }
}
```

Файл Edge.java

```
package ru.etu.practice;

public class Edge {
    @Override
    public String toString() {
        return "Edge " + " " + from + "—" + to + " " + distance;
    }

    /**
     * way from -> to == way to -> to
     */
    public Node from;
    public Node to;
    public int distance;
```

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

    }

    @Override
    public int hashCode() {
        return super.hashCode();
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (o == null) {
            return false;
        }
        Edge other = (Edge) o;
        return this.to.getName() == other.to.getName() && this.from.getName() == other.from.getName() &&
this.distance == other.distance ||
                this.from.getName() == other.to.getName() && this.to.getName() == other.from.getName() &&
this.distance == other.distance;
    }

    public boolean equalsWay(Edge other){
        if (this == other) {
            return true;
        }
        if (other == null) {
            return false;
        }
        return this.to.getName() == other.to.getName() && this.from.getName() == other.from.getName() ||
                this.from.getName() == other.to.getName() && this.to.getName() == other.from.getName();
    }

}
```

Файл Node.java

```
package ru.etu.practice;

import java.util.Objects;

public class Node {
    private final char name;

    public char getName() {
        return name;
    }

    public Node(char name){
        this.name = name;
    }
```

```
    @Override
    public String toString() {
        return String.valueOf(name);
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Node node = (Node) o;
        return name == node.name;
    }

    @Override
    public int hashCode() {
        return Objects.hash(name);
    }
}
```

Файл State.java

```
package ru.etu.practice;

public enum State {
    SORT,
    APPEND,
    LOOP,
    CONNECT_COMPONENTS,
    NEW_COMPONENT,
    END,
    CLEAR
}
```

Файл KruskalTest.java

```
package ru.etu.practice;

import org.junit.jupiter.api.Test;

import java.util.*;

import static org.junit.jupiter.api.Assertions.*;

class KruskalTest {

    public void sortEdges(List<Edge> list) {
        list.sort(new Comparator<Edge>() {
            @Override
            public int compare(Edge edge1, Edge edge2) {
                int distance1 = edge1.distance;
                int distance2 = edge2.distance;
                return Integer.compare(distance1, distance2);
            }
        });
```

```
}

@Test
void equalsAbAb(){
   // arrange
   Edge ab = new Edge(new Node('a'), new Node('b'), 4);
   Edge ba = new Edge(new Node('a'), new Node('b'), 4);

   // act
   boolean equality = ab.equals(ba);

   // assert
   assertTrue(equality);
}

@Test
void equalsAbBa(){
   // arrange
   Edge ab = new Edge(new Node('a'), new Node('b'), 4);
   Edge ba = new Edge(new Node('b'), new Node('a'), 4);

   // act
   boolean equality = ab.equals(ba);

   // assert
   assertTrue(equality);
}

@Test
void notEqualsAb3Ab4(){
   // arrange
   Edge ab1 = new Edge(new Node('a'), new Node('b'), 3);
   Edge ab2 = new Edge(new Node('a'), new Node('b'), 4);

   // act
   boolean equality = ab1.equals(ab2);

   // assert
   assertFalse(equality);
}

@Test
void notEqualsAbAndNull(){
   // arrange
   Edge ab1 = new Edge(new Node('a'), new Node('b'), 3);
   Edge ab2 = null;

   // act
   boolean equality = ab1.equals(ab2);

   // assert
   assertFalse(equality);
}

@Test
void sort4Edges(){
```

```
    // arrange

    List<Edge> inputEdges = Arrays.asList(
            new Edge(new Node('b'), new Node('c'), 4),
            new Edge(new Node('a'), new Node('b'), 3),
            new Edge(new Node('a'), new Node('c'), 7),
            new Edge(new Node('c'), new Node('d'), 6)
    );

    List<Node> inputVertices = Arrays.asList(
            new Node('a'),
            new Node('b'),
            new Node('c')
    );

    // act
    Kruskal alg = new Kruskal();
    alg.initKruskal(inputEdges, inputVertices);
    alg.sortEdges();
    inputEdges = alg.getInputEdges();
    List<Edge> expectedEdges = Arrays.asList(
            new Edge(new Node('a'), new Node('b'), 3),
            new Edge(new Node('b'), new Node('c'), 4),
            new Edge(new Node('c'), new Node('d'), 6),
            new Edge(new Node('a'), new Node('c'), 7)
    );

    // assert
    assertEquals(inputEdges, expectedEdges);
}

@Test
void sort8Edges(){
    // arrange
    List<Edge> inputEdges = Arrays.asList(
            new Edge(new Node('a'), new Node('b'), 1),
            new Edge(new Node('b'), new Node('d'), 3),
            new Edge(new Node('a'), new Node('c'), 2),
            new Edge(new Node('c'), new Node('d'), 6),
            new Edge(new Node('d'), new Node('m'), 10),
            new Edge(new Node('c'), new Node('l'), 4),
            new Edge(new Node('l'), new Node('a'), 8),
            new Edge(new Node('c'), new Node('b'), 9)
    );

    List<Node> inputVertices = Arrays.asList(
            new Node('a'),
            new Node('b'),
            new Node('c'),
            new Node('d'),
            new Node('m'),
            new Node('l')
    );

    // act
    Kruskal alg = new Kruskal();
```

```
    alg.initKruskal(inputEdges, inputVertices);
    alg.sortEdges();
    inputEdges = alg.getInputEdges();
    List<Edge> expectedEdges = Arrays.asList(
            new Edge(new Node('a'), new Node('b'), 1),
            new Edge(new Node('a'), new Node('c'), 2),
            new Edge(new Node('b'), new Node('d'), 3),
            new Edge(new Node('c'), new Node('l'), 4),
            new Edge(new Node('c'), new Node('d'), 6),
            new Edge(new Node('l'), new Node('a'), 8),
            new Edge(new Node('c'), new Node('b'), 9),
            new Edge(new Node('d'), new Node('m'), 10)
    );

    // assert
    assertEquals(inputEdges, expectedEdges);
}

@Test
void firstNextStep(){
    // arrange

    List<Edge> inputEdges = Arrays.asList(
            new Edge(new Node('b'), new Node('c'), 4),
            new Edge(new Node('a'), new Node('b'), 3),
            new Edge(new Node('a'), new Node('c'), 7)
    );
    List<Node> inputVertices = Arrays.asList(
            new Node('a'),
            new Node('b'),
            new Node('c')
    );

    // act
    Kruskal alg = new Kruskal();
    alg.initKruskal(inputEdges, inputVertices);
    List<Object> tuple = alg.nextStep();
    List<Object> expectedTuple = new LinkedList<>();
    expectedTuple.add(State.SORT);

    // assert
    assertEquals(expectedTuple, tuple);
}

@Test
void newComponentNextStep(){
    // arrange

    List<Edge> inputEdges = Arrays.asList(
            new Edge(new Node('b'), new Node('c'), 4),
            new Edge(new Node('a'), new Node('b'), 3),
            new Edge(new Node('a'), new Node('c'), 7)
    );
    List<Node> inputVertices = Arrays.asList(
            new Node('a'),
            new Node('b'),
```

```
            new Node('c')
    );

    // act
    Kruskal alg = new Kruskal();
    alg.initKruskal(inputEdges, inputVertices);
    List<Object> tuple = alg.nextStep();
    List<Object> expectedTuple = new LinkedList<>();
    expectedTuple.add(State.SORT);

    // assert
    assertEquals(expectedTuple, tuple);

    // act
    tuple = alg.nextStep();
    expectedTuple.clear();
    expectedTuple.add(State.NEW_COMPONENT);
    expectedTuple.add(alg.getInputEdges().get(0));

    // assert
    assertEquals(expectedTuple, tuple);
}

@Test
void appendNextStep(){
    // arrange

    List<Edge> inputEdges = Arrays.asList(
            new Edge(new Node('b'), new Node('c'), 4),
            new Edge(new Node('a'), new Node('b'), 3),
            new Edge(new Node('a'), new Node('c'), 7)
    );
    List<Node> inputVertices = Arrays.asList(
            new Node('a'),
            new Node('b'),
            new Node('c')
    );

    // act
    Kruskal alg = new Kruskal();
    alg.initKruskal(inputEdges, inputVertices);
    List<Object> tuple = alg.nextStep();
    List<Object> expectedTuple = new LinkedList<>();
    expectedTuple.add(State.SORT);

    // assert
    assertEquals(expectedTuple, tuple);

    // act
    tuple = alg.nextStep();
    expectedTuple.clear();
    expectedTuple.add(State.NEW_COMPONENT);
    expectedTuple.add(alg.getInputEdges().get(0));

    // assert
    assertEquals(expectedTuple, tuple);
```

```
    // act
    tuple = alg.nextStep();
    expectedTuple.clear();
    expectedTuple.add(State.APPEND);
    expectedTuple.add(alg.getInputEdges().get(1));

    // assert
    assertEquals(expectedTuple, tuple);
}

@Test
void endNextStep(){
    // arrange

    List<Edge> inputEdges = Arrays.asList(
            new Edge(new Node('b'), new Node('c'), 4),
            new Edge(new Node('a'), new Node('b'), 3),
            new Edge(new Node('a'), new Node('c'), 7)
    );
    List<Node> inputVertices = Arrays.asList(
            new Node('a'),
            new Node('b'),
            new Node('c')
    );

    // act
    Kruskal alg = new Kruskal();
    alg.initKruskal(inputEdges, inputVertices);
    List<Object> tuple = alg.nextStep();
    List<Object> expectedTuple = new LinkedList<>();
    expectedTuple.add(State.SORT);

    // assert
    assertEquals(expectedTuple , tuple);

    // act
    tuple = alg.nextStep();
    expectedTuple.clear();
    expectedTuple.add(State.NEW_COMPONENT);
    expectedTuple.add(alg.getInputEdges().get(0));

    // assert
    assertEquals(expectedTuple , tuple);

    // act
    tuple = alg.nextStep();
    expectedTuple.clear();
    expectedTuple.add(State.APPEND);
    expectedTuple.add(alg.getInputEdges().get(1));

    // assert
    assertEquals(expectedTuple , tuple);

    // act
    tuple = alg.nextStep();
```

```
    expectedTuple.clear();
    expectedTuple.add(State.END);
    expectedTuple.add(7);

    // assert
    assertEquals(expectedTuple , tuple);
}

@Test
void loopNextStep(){
    // arrange

    List<Edge> inputEdges = Arrays.asList(
            new Edge(new Node('b'), new Node('c'), 5),
            new Edge(new Node('a'), new Node('b'), 6),
            new Edge(new Node('d'), new Node('b'), 2),
            new Edge(new Node('d'), new Node('c'), 3)
    );
    List<Node> inputVertices = Arrays.asList(new Node('a'),
            new Node('b'),
            new Node('c'),
            new Node('d')
    );

    // act
    Kruskal alg = new Kruskal();
    alg.initKruskal(inputEdges, inputVertices);
    List<Object> tuple = alg.nextStep();
    List<Object> expectedTuple = new LinkedList<>();
    expectedTuple.add(State.SORT);

    // assert
    assertEquals(expectedTuple, tuple);

    // act
    tuple = alg.nextStep();
    expectedTuple.clear();
    expectedTuple.add(State.NEW_COMPONENT);
    expectedTuple.add(alg.getInputEdges().get(0));

    // assert
    assertEquals(expectedTuple, tuple);

    // act
    tuple = alg.nextStep();
    expectedTuple.clear();
    expectedTuple.add(State.APPEND);
    expectedTuple.add(alg.getInputEdges().get(1));

    // assert
    assertEquals(expectedTuple, tuple);

    // act
    tuple = alg.nextStep();
    expectedTuple.clear();
    expectedTuple.add(State.LOOP);
```

```
    expectedTuple.add(alg.getInputEdges().get(2));

    // assert
    assertEquals(expectedTuple, tuple);

    // act
    tuple = alg.nextStep();
    expectedTuple.clear();
    expectedTuple.add(State.APPEND);
    expectedTuple.add(alg.getInputEdges().get(3));

    // assert
    assertEquals(expectedTuple, tuple);

    // act
    tuple = alg.nextStep();
    expectedTuple.clear();
    expectedTuple.add(State.END);
    expectedTuple.add(11);

    // assert
    assertEquals(expectedTuple, tuple);
}

@Test
void afterEndNextStep(){
    // arrange

    List<Edge> inputEdges = Arrays.asList(
            new Edge(new Node('b'), new Node('c'), 4),
            new Edge(new Node('a'), new Node('b'), 3),
            new Edge(new Node('a'), new Node('c'), 7)
    );
    List<Node> inputVertices = Arrays.asList(
            new Node('a'),
            new Node('b'),
            new Node('c')
    );

    // act
    Kruskal alg = new Kruskal();
    alg.initKruskal(inputEdges, inputVertices);
    List<Object> tuple = alg.nextStep();
    List<Object> expectedTuple = new LinkedList<>();
    expectedTuple.add(State.SORT);

    // assert
    assertEquals(expectedTuple , tuple);

    // act
    tuple = alg.nextStep();
    expectedTuple.clear();
    expectedTuple.add(State.NEW_COMPONENT);
    expectedTuple.add(alg.getOutputEdges().get(0));

    // assert
```

```
        assertEquals(expectedTuple , tuple);

        // act
        tuple = alg.nextStep();
        expectedTuple.clear();
        expectedTuple.add(State.APPEND);
        expectedTuple.add(alg.getInputEdges().get(1));

        // assert
        assertEquals(expectedTuple , tuple);

        // act
        tuple = alg.nextStep();
        expectedTuple.clear();
        expectedTuple.add(State.END);
        expectedTuple.add(7);

        // assert
        assertEquals(expectedTuple , tuple);

//        // act
//        tuple = alg.nextStep();
//        expectedTuple.clear();
//        expectedTuple.add(State.END);
//        expectedTuple.add(7);
//
//        // assert
//        assertEquals(expectedTuple , tuple);

        // act
        tuple = alg.nextStep();
        expectedTuple.clear();
        expectedTuple.add(State.SORT);

        // assert
        assertEquals(expectedTuple , tuple);
    }

    @Test
    void kraskalOfEmptyGraph() {
        // arrange

        List<Edge> inputEdges = new ArrayList<>();
        List<Node> inputVertices = new ArrayList<>();

        // act
        Kruskal alg = new Kruskal();
        alg.initKruskal(inputEdges, inputVertices);
        alg.kruskal();
        List<Edge> outputEdges = alg.getOutputEdges();
        sortEdges(outputEdges);
        List<Edge> expectedEdges = new ArrayList<>();

        // assert
        assertEquals(outputEdges, expectedEdges);
    }
```

```
@Test
void kraskalGraphOf3Edges() {
   // arrange

   List<Edge> inputEdges = Arrays.asList(
         new Edge(new Node('b'), new Node('c'), 4),
         new Edge(new Node('a'), new Node('b'), 3),
         new Edge(new Node('a'), new Node('c'), 7)
   );

   List<Node> inputVertices = Arrays.asList(
         new Node('a'),
         new Node('b'),
         new Node('c')
   );

   // act
   Kruskal alg = new Kruskal();
   alg.initKruskal(inputEdges, inputVertices);
   alg.kruskal();
   List<Edge> outputEdges = alg.getOutputEdges();
   sortEdges(outputEdges);
   List<Edge> expectedEdges = Arrays.asList(
         new Edge(new Node('a'), new Node('b'), 3),
         new Edge(new Node('b'), new Node('c'), 4)
   );

   // assert
   assertEquals(outputEdges, expectedEdges);
}

@Test
void kraskalGraphOf7Edges() {
   // arrange

   List<Edge> inputEdges = Arrays.asList(
         new Edge(new Node('a'), new Node('c'), 1),
         new Edge(new Node('a'), new Node('d'), 5),
         new Edge(new Node('a'), new Node('g'), 4),
         new Edge(new Node('b'), new Node('c'), 4),
         new Edge(new Node('b'), new Node('d'), 3),
         new Edge(new Node('c'), new Node('f'), 3),
         new Edge(new Node('f'), new Node('e'), 6)
   );

   List<Node> inputVertices = Arrays.asList(
         new Node('a'),
         new Node('b'),
         new Node('c'),
         new Node('d'),
         new Node('e'),
         new Node('f'),
         new Node( 'g')
   );
```

```
    // act
    Kruskal alg = new Kruskal();
    alg.initKruskal(inputEdges, inputVertices);
    alg.kruskal();
    List<Edge> outputEdges = alg.getOutputEdges();
    sortEdges(outputEdges);
    List<Edge> expectedEdges = Arrays.asList(
            new Edge(new Node('a'), new Node('c'), 1),
            new Edge(new Node('b'), new Node('d'), 3),
            new Edge(new Node('c'), new Node('f'), 3),
            new Edge(new Node('a'), new Node('g'), 4),
            new Edge(new Node('b'), new Node('c'), 4),
            new Edge(new Node('f'), new Node('e'), 6)
    );

    // assert
    assertEquals(outputEdges, expectedEdges);
}

@Test
void kraskalGraphOf8Edges() {
    // arrange

    List<Edge> inputEdges = Arrays.asList(
            new Edge(new Node('a'), new Node('b'), 2),
            new Edge(new Node('a'), new Node('c'), 1),
            new Edge(new Node('a'), new Node('d'), 3),
            new Edge(new Node('b'), new Node('c'), 6),
            new Edge(new Node('b'), new Node('d'), 5),
            new Edge(new Node('c'), new Node('d'), 4),
            new Edge(new Node('b'), new Node('e'), 9),
            new Edge(new Node('c'), new Node('e'), 10)
    );

    List<Node> inputVertices = Arrays.asList(
            new Node('a'),
            new Node('b'),
            new Node('c'),
            new Node('d'),
            new Node('e')
    );

    // act
    Kruskal alg = new Kruskal();
    alg.initKruskal(inputEdges, inputVertices);
    alg.kruskal();
    List<Edge> outputEdges = alg.getOutputEdges();
    sortEdges(outputEdges);
    List<Edge> expectedEdges = Arrays.asList(
            new Edge(new Node('a'), new Node('c'), 1),
            new Edge(new Node('a'), new Node('b'), 2),
            new Edge(new Node('a'), new Node('d'), 3),
            new Edge(new Node('b'), new Node('e'), 9)
    );

    // assert
```

```
    assertEquals(outputEdges, expectedEdges);
}

@Test
void kraskalGraphOf13Edges() {
    // arrange

    List<Edge> inputEdges = Arrays.asList(
            new Edge(new Node('a'), new Node('b'), 1),
            new Edge(new Node('a'), new Node('c'), 3),
            new Edge(new Node('a'), new Node('e'), 2),
            new Edge(new Node('b'), new Node('d'), 2),
            new Edge(new Node('c'), new Node('d'), 1),
            new Edge(new Node('c'), new Node('e'), 2),
            new Edge(new Node('e'), new Node('j'), 10),
            new Edge(new Node('f'), new Node('g'), 1),
            new Edge(new Node('f'), new Node('h'), 4),
            new Edge(new Node('f'), new Node('j'), 1),
            new Edge(new Node('g'), new Node('j'), 1),
            new Edge(new Node('g'), new Node('i'), 1),
            new Edge(new Node('h'), new Node('i'), 5)
    );

    List<Node> inputVertices = Arrays.asList(
            new Node('a'),
            new Node('b'),
            new Node('c'),
            new Node('d'),
            new Node('e'),
            new Node('f'),
            new Node('g'),
            new Node('h'),
            new Node('i'),
            new Node('j')
    );

    // act
    Kruskal alg = new Kruskal();
    alg.initKruskal(inputEdges, inputVertices);
    alg.kruskal();
    List<Edge> outputEdges = alg.getOutputEdges();
    sortEdges(outputEdges);
    List<Edge> expectedEdges = Arrays.asList(
            new Edge(new Node('a'), new Node('b'), 1),
            new Edge(new Node('c'), new Node('d'), 1),
            new Edge(new Node('f'), new Node('g'), 1),
            new Edge(new Node('f'), new Node('j'), 1),
            new Edge(new Node('g'), new Node('i'), 1),
            new Edge(new Node('a'), new Node('e'), 2),
            new Edge(new Node('b'), new Node('d'), 2),
            new Edge(new Node('f'), new Node('h'), 4),
            new Edge(new Node('e'), new Node('j'), 10)
    );

    // assert
    assertEquals(outputEdges, expectedEdges);
```

```
}

@Test
void kraskalGraphOf40Edges() {
   // arrange

   List<Edge> inputEdges = Arrays.asList(
         new Edge(new Node('a'), new Node('b'), 4),
         new Edge(new Node('a'), new Node('c'), 20),
         new Edge(new Node('b'), new Node('d'), 10),
         new Edge(new Node('b'), new Node('x'), 1),
         new Edge(new Node('c'), new Node('x'), 30),
         new Edge(new Node('c'), new Node('z'), 4),
         new Edge(new Node('d'), new Node('f'), 21),
         new Edge(new Node('d'), new Node('w'), 6),
         new Edge(new Node('e'), new Node('z'), 15),
         new Edge(new Node('f'), new Node('h'), 5),
         new Edge(new Node('g'), new Node('i'), 26),
         new Edge(new Node('g'), new Node('v'), 9),
         new Edge(new Node('h'), new Node('n'), 22),
         new Edge(new Node('h'), new Node('w'), 1),
         new Edge(new Node('i'), new Node('v'), 5),
         new Edge(new Node('i'), new Node('z'), 7),
         new Edge(new Node('j'), new Node('y'), 13),
         new Edge(new Node('k'), new Node('m'), 29),
         new Edge(new Node('k'), new Node('z'), 27),
         new Edge(new Node('l'), new Node('u'), 12),
         new Edge(new Node('m'), new Node('o'), 28),
         new Edge(new Node('m'), new Node('v'), 25),
         new Edge(new Node('n'), new Node('u'), 23),
         new Edge(new Node('o'), new Node('z'), 16),
         new Edge(new Node('p'), new Node('u'), 7),
         new Edge(new Node('q'), new Node('v'), 14),
         new Edge(new Node('r'), new Node('t'), 11),
         new Edge(new Node('r'), new Node('y'), 18),
         new Edge(new Node('r'), new Node('z'), 8),
         new Edge(new Node('s'), new Node('t'), 24),
         new Edge(new Node('s'), new Node('v'), 6),
         new Edge(new Node('t'), new Node('v'), 8),
         new Edge(new Node('t'), new Node('z'), 19),
         new Edge(new Node('u'), new Node('v'), 17),
         new Edge(new Node('v'), new Node('y'), 3),
         new Edge(new Node('v'), new Node('z'), 9),
         new Edge(new Node('w'), new Node('x'), 2),
         new Edge(new Node('x'), new Node('y'), 10),
         new Edge(new Node('x'), new Node('z'), 2),
         new Edge(new Node('y'), new Node('z'), 3)
   );

   List<Node> inputVertices = Arrays.asList(
         new Node('a'),
         new Node('b'),
         new Node('c'),
         new Node('d'),
         new Node('e'),
         new Node('f'),
```

```
        new Node('g'),
        new Node('h'),
        new Node('i'),
        new Node('j'),
        new Node('k'),
        new Node('l'),
        new Node('m'),
        new Node('n'),
        new Node('o'),
        new Node('p'),
        new Node('q'),
        new Node('r'),
        new Node('s'),
        new Node('t'),
        new Node('u'),
        new Node('v'),
        new Node('w'),
        new Node('x'),
        new Node('y'),
        new Node('z')
);

// act
Kruskal alg = new Kruskal();
alg.initKruskal(inputEdges, inputVertices);
alg.kruskal();
List<Edge> outputEdges = alg.getOutputEdges();
sortEdges(outputEdges);
List<Edge> expectedEdges = Arrays.asList(
        new Edge(new Node('b'), new Node('x'), 1),
        new Edge(new Node('h'), new Node('w'), 1),
        new Edge(new Node('w'), new Node('x'), 2),
        new Edge(new Node('x'), new Node('z'), 2),
        new Edge(new Node('v'), new Node('y'), 3),
        new Edge(new Node('y'), new Node('z'), 3),
        new Edge(new Node('a'), new Node('b'), 4),
        new Edge(new Node('c'), new Node('z'), 4),
        new Edge(new Node('f'), new Node('h'), 5),
        new Edge(new Node('i'), new Node('v'), 5),
        new Edge(new Node('d'), new Node('w'), 6),
        new Edge(new Node('s'), new Node('v'), 6),
        new Edge(new Node('p'), new Node('u'), 7),
        new Edge(new Node('r'), new Node('z'), 8),
        new Edge(new Node('t'), new Node('v'), 8),
        new Edge(new Node('g'), new Node('v'), 9),
        new Edge(new Node('l'), new Node('u'), 12),
        new Edge(new Node('j'), new Node('y'), 13),
        new Edge(new Node('q'), new Node('v'), 14),
        new Edge(new Node('e'), new Node('z'), 15),
        new Edge(new Node('o'), new Node('z'), 16),
        new Edge(new Node('u'), new Node('v'), 17),
        new Edge(new Node('h'), new Node('n'), 22),
        new Edge(new Node('m'), new Node('v'), 25),
        new Edge(new Node('k'), new Node('z'), 27)
);
```

```
        // assert
        assertEquals(outputEdges, expectedEdges);
    }
}
```