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

ОТЧЕТ

по учебной практике

Тема: Реализация алгоритма Краскала на языке Java с визуализацией

Студент гр. 2384 _____ Дамакин Р.П.

Студент гр. 2384 _____ Шурыгин Д.Л.

Студентка гр. 2384 _____ Матеюк Д.С.

Руководитель _____ Шестопалов Р.П.

Санкт-Петербург

2024

ЗАДАНИЕ

НА УЧЕБНУЮ ПРАКТИКУ

Студент Дамакин Р.П. группы 2384

Студент Шурыгин Д.Л. группы 2384

Студентка Матеюк Д.С. группы 2384

Тема практики: наименование темы

Задание на практику:

Командная итеративная разработка визуализатора алгоритма на Java с графическим интерфейсом.

Алгоритм: Краскала.

Сроки прохождения практики: 26.06.2024 – 09.07.2024

Дата сдачи отчета: 08.07.2024

Дата защиты отчета: 08.07.2024

Студент гр. 2384	_____	Дамакин Р.П.
Студент гр. 2384	_____	Шурыгин Д.Л.
Студентка гр. 2384	_____	Матеюк Д.С.
Руководитель	_____	Шестопалов Р.П.

АННОТАЦИЯ

Целью практики является изучение нового языка программирования, приобретения опыта коллективной работы над разработкой собственного проекта. В ходе работы разрабатывался визуализатор алгоритма Краскала на ранее неизвестном языке программирования Java. Работа была разбита на этапы: составление спецификации и плана разработки, прототипа, первой версии, финальной версии. Каждый этап совершенствует предыдущий при этом согласовывается с руководителем, после чего в проект вносятся правки. В отчёте приведена информация о ходе выполнения практической работы.

SUMMARY

The purpose of the practice is to study a new programming language, to gain experience of teamwork on the development of their own project. In the course of the work we developed a visualizer of Kruskal's algorithm in the previously unknown programming language Java. The work was divided into stages: specification and development plan, prototype, first version, final version. Each stage improves the previous one and is coordinated with the supervisor, after which the project is amended. The report provides information on the progress of the practical work.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цель практики – визуализация алгоритма на языке программирования Java. Для выполнения цели были выполнены следующие шаги:

1) Самостоятельное изучение языка Java и библиотеки Java Swing.
2) Изучение алгоритма Краскала.
3) Составление спецификации и плана разработки приложения, распределение ролей.
4) Написание первого этапа.
5) Совершенствование программы в соответствии с ТЗ.
6) Сдача проекта руководителю.

Алгоритм Краскала — эффективный алгоритм построения минимального остовного дерева. В начале текущее множество рёбер устанавливается пустым. Затем, пока это возможно, проводится следующая операция: из всех рёбер, добавление которых к уже имеющемуся множеству не вызовет появление в нём цикла, выбирается ребро минимального веса и добавляется к уже имеющемуся множеству. Когда таких рёбер больше нет, алгоритм завершён. Подграф данного графа, содержащий все его вершины и найденное множество рёбер, является его остовного дерева минимального веса. Подробное описание алгоритма можно найти в литературе. Сложность алгоритма O(n*log(n)). [1]

1. ТРЕБОВАНИЯ К ПРОГРАММЕ

1.1. Исходные Требования к программе

Визуализация алгоритма Краскала для поиска минимального покрывающего дерева на языке Java.

1.1.1 – требования к вводу исходных данных

Граф можно вручную нарисовать в программе, либо импортировать файлом. ЛКМ используется для добавления вершины, ПКМ для удаление, колесико для перемещения вершин графа по области.

1.1.2 – требования к визуализации

Графический интерфейс содержит следующие составляющие:

1) Область редактирования графа: подсказка, объясняющая как нарисовать граф.

2) Область с инструментами работы с графом: кнопка редактирования графа, кнопка редактирования веса графа, кнопка очищения области, кнопки импорта/экспорта графа.

3) Область работы с консолью: кнопка запуска алгоритма, кнопки для перехода на шаг вперед/шаг назад.

1.2. Уточнение требований после сдачи прототипа

Добавить цвет при выделении вершины.

1.3. Уточнение требований после сдачи первой версии

1) Окошко для добавления веса должно создаваться сразу после создания ребра

2) Импорт/экспорт должен осуществляться через проводник

2. ПЛАН РАЗРАБОТКИ И РАСПРЕДЕЛЕНИЕ РОЛЕЙ В БРИГАДЕ

2.1. План разработки

1) Разработка дизайна. (до 27.06)
2) Разработка GUI. (до 01.07)
3) Реализация рисования и редактирования графов. (до 02.07)
4) Реализация алгоритма Краскала. (до 03.07)
5) Реализация отображения шагов и вывода в консоль информации о шаге. (до 04.07)
6) Полировка (до 04.07)

2.2. Распределение ролей в бригаде

Дамакин Р.П. – разработка интерфейса.

Шурыгин Д.Л. – реализация алгоритма Краскала.

Матеюк Д.С. – разработка дизайна, написание отчета.

3. ОСОБЕННОСТИ РЕАЛИЗАЦИИ

3.1. Структуры данных

Класс *Main*

Главный класс, реализующий главную функцию запуска программы.

Методы класса

1) *public static void main(String[] args)* - точка входа, функция, с помощью которой осуществляется запуск программы. В методе осуществляется установка системной темы для программы и создание экземпляра класса *App*.

Класс *App*

Класс *App* наследуется от класса *JPanel*. *App* - класс приложения, осуществляющий конструирование основного окна приложения, объединяющий создание всех компонентов для интерфейса приложения.

Поля класса

1) *int DEFAULT_WIDTH* = 800 - ширина основного окна приложения;

2) *int DEFAULT_HEIGHT* = 600 - высота основного окна приложения;

3) *String Title* = "KruskalLimitedEdition.official.feat(Romchik,Danchik,Dianchik)" - название приложения, которое высвечивается в углу основного окна приложения;

4) *private JPanel MainPanel* - экземпляр основного окна приложения, задается с помощью формы *App.form*;

5) *private JPanel ToolsPane* - экземпляр окна панели инструментов, задается с помощью формы *App.form*, объект создается с помощью класса *ToolkitPanel* (будет описан ниже);

6) *private JButton DrawButton* - экземпляр кнопки редактирования графа, задается с помощью формы *App.form*;

7) *private JButton EditButton* - экземпляр кнопки изменения весов ребер графа, задается с помощью формы *App.form*;

8) *private JButton DeleteButton* - экземпляр кнопки удаления графа, задается с помощью формы *App.form*;

9) *private JButton DownloadButton* - экземпляр кнопки загрузки графа из файла, задается с помощью формы *App.form*;

10) *private JButton UploadButton* - экземпляр кнопки сохранения графа в файл, задается с помощью формы *App.form*;

11) *private JPanel GraphEditPanel* - экземпляр области редактирования графа, задается с помощью формы *App.form*, объект создается как экземпляр класса *GraphPanel* (будет описан ниже);

12) *private JScrollPane ConsolePanel* - экземпляр области консоли графа, задается с помощью формы *App.form*;

13) *private JButton StartButton* - экземпляр кнопки запуска алгоритма Краскала на построенном графе, задается с помощью формы *App.form*;

14) *private JButton PrevStepButton* - экземпляр кнопки перехода к предыдущему шагу алгоритма Краскала, задается с помощью формы *App.form*;

15) *private JButton NextStepButton* - экземпляр кнопки перехода к следующему шагу алгоритма Краскала, задается с помощью формы *App.form*;

16) *private JPanel ControlPanel* - экземпляр области, в которую помещаются кнопки запуска алгоритма и переходов к следующему и предыдущему полям, задается с помощью формы *App.form*;

17) *private JTextArea ConsoleTextArea* - экземпляр текстового поля, располагающегося в *ConsolePanel* для помещения туда вывода текстовой информации о шагах алгоритма, задается с помощью формы *App.form*;

18) *private JTextArea HelpText* - экземпляр текстового поля, содержащего подсказку для области редактирования графа, задается с помощью формы *App.form*, объект задается с помощью класса *HelpTextArea* (будет описан ниже);

Методы класса

1) *public App()* - конструктор класса. В конструкторе вызывается метод *setResizable(false)* для предотвращения изменения размера окна пользователем. Используется *Toolkit.getDefaultToolkit().getScreenSize()* для получения размера экрана и центрирования окна относительно него. Устанавливаются иконка и заголовок окна. Для текстового поля *HelpText* задается шрифт и текст с инструкциями по использованию. Для текстового поля *ConsoleTextArea* также задается шрифт и включены переносы строк по словам. Кнопкам назначаются слушатели действий (addActionListener), которые меняют состояние графа или выполняют другие действия в зависимости от нажатой кнопки. Также для кнопок устанавливаются настройки отображения - установка иконок с помощью метода *setIcon* (описан ниже), отключение отображения стандартного фона и границ кнопок.

2) *public static void setIcon(JButton button, String image_path)* - метод для настройки отображения кнопок.

3) *private void createUIComponents()* - создаются все классы с переопределенным и расширенным функционалом.

Класс ToolkitPanel

Класс *ToolkitPanel* является подклассом *JPanel*, предназначенным для отрисовки панели с закругленными углами. Этот класс может быть использован в качестве контейнера для других компонентов интерфейса.

Поля класса

1) *private final int cornerRadius*; - это поле хранит радиус закругления углов панели.

Методы класса

1) *public ToolkitPanel()* - конструктор без параметров, который инициализирует панель с заданным радиусом закругления углов и устанавливает её прозрачность в false для корректного отображения закругленных краев.

2) *@Override protected void paintComponent(Graphics g)* - метод paintComponent переопределен для добавления специальной логики рисования, которая позволяет отобразить панель с закругленными углами. В этом методе происходит вызов *super.paintComponent(g)* для выполнения базовой логики рисования компонента, создание объекта *Dimension* с радиусом закругления для использования в методе *fillRoundRect*, получение текущих размеров панели, отрисовка закругленного прямоугольника, используя полученный радиус закругления и цвета фона панели.

Класс HelpTextArea

Класс *HelpTextArea* представляет собой подкласс *JTextArea,* модифицированный для отображения текста внутри области с закругленными углами. Это позволяет создать уникальный вид для элемента ввода текста, соответствующий общему дизайну приложения.

Методы класса

1) *@Override protected void paintComponent(Graphics g)* - метод paintComponent переопределен для добавления специальной логики рисования, которая позволяет отобразить область текста с закругленными углами. В этом методе происходит создание объекта *Shape* типа *RoundRectangle2D.Double,* который представляет закругленный прямоугольник с заданными размерами и радиусом скругления углов, установка этого закругленного прямоугольника в качестве области обрезки (clip) для Graphics2D, что ограничивает область рисования до формы закругленного прямоугольника, вызов super.paintComponent(g2) для выполнения базовой логики рисования компонента *JTextArea* поверх закругленной области, если переданный графический объект не является экземпляром *Graphics2D*, вызывается обычная реализация *paintComponent*, чтобы обеспечить совместимость с другими компонентами.

Класс GraphPanel

GraphPanel представляет собой подкласс *JPanel*, который служит для создания области визуализации и редактирования графов. Он предоставляет интерфейс пользователя для добавления узлов и ребер, а также для выполнения алгоритма Краскала для построения минимального остовного дерева графа.

Поля класса

1) *graph*: Граф, который хранится в этом компоненте.

Используемые обработчики событий мыши

mousePressed(MouseEvent e): Обрабатывает нажатие мыши для добавления узлов, удаления узлов и ребер, и выбора узла для перемещения.

mouseReleased(MouseEvent e): Обрабатывает отпускание кнопки мыши после добавления ребра или окончания выбора узла.

mouseClicked(MouseEvent e): Обрабатывает клик мышью для изменения веса ребра.

mouseDragged(MouseEvent e): Обрабатывает перетаскивание узла.

2) *selectedNode*: Выбранный узел, который может быть перемещен или изменен.

3) *current_state*: Текущее состояние панели, определяющее, можно ли рисовать, редактировать или удалять элементы графа.

4) *step, old_graph_hash*: Переменные для управления процессом выполнения алгоритма Краскала. *step* отвечает за текущий шаг алгоритма симуляции, *old_graph_hash* сохраняет хеш графа для проверки, изменился ли граф в конкретный момент (используется для того, чтобы не нужно было при каждом шаге симуляции заново запускать полное прохождения алгоритма).

5) *kruskal, kruskal_steps*: Содержат информацию для выполнения алгоритма Краскала. *kruskal* хранит объект класса *KruskalAlgorithm* для выполнения алгоритма. *kruskal_steps* - массив, сохраняющий индексы выделенных в результате работы алгоритма ребер (результат алгоритма Краскала).

6) *DEFAULT_GRAPH_COLOR, DEFAULT_COLOR_FOR_OST, DEFAULT_PREV_STEP_COLOR*: Цвета для различных состояний узлов и ребер: стандартный цвет графа, стандартный цвет выделенного остовного дерева, стандартный цвет выделения предыдущего шага для пошагового выполнения.

Методы класса

7) *public GraphPanel()* - метод инициализирует граф, устанавливает начальное состояние и добавляет обработчики событий мыши для взаимодействия с графом. Обработчики позволяют добавлять узлы, удалять узлы и ребра, изменять веса ребер и перемещать узлы с помощью определенных ниже методов и использования возможностей библиотек swing и awt.

8) *private void setEdgesColor(Node node, Color color)* - метод для покраски ребер, смежных с переданной вершиной, в указанный цвет. Используется для выделения вершины и смежных ребер при перетаскивании вершины.

9) *public void setState(GraphPanelStates state)* - изменяет текущее состояние панели. Метод используется кнопками *DrawButton* и *EditButton* для установки режима редактирования.

10) *public void clearGraph()* - очищает граф и сбрасывает состояние. Используется кнопкой *DeleteButton*.

11) *public void loadGraph(),* public void *saveGraph()* - методы загружают и сохраняют граф из файла соответственно. Используются кнопками *DownloadButton* и *UploadButton*. Для реализации используют объект *FileChooser* для создания окна выбора места сохранения и названия файла или же файла, который нужно загрузить. При реализации также используют соответствующие методы класса *Graph*.

12) *private String showInputDialog(String message, String initialValue)* - отображает диалоговое окно для ввода данных. Используется для отображения окна ввода веса ребра в режиме редактирования весов.

13) *private JFormattedTextField createNumberTextField()* - создает текстовое поле для ввода числовых значений. Используется для проверки и редактирования введенного значения при редактировании весов ребер.

14) *public void kruskalAlgorithmFunc(JTextArea console)* - метод, выполняющий алгоритм Краскала для построенного графа. В методе также осуществляется вывод диалогового окна с сообщением об ошибке в случае невозможности выполнения алгоритма на построенном графе. В переданную в качестве параметра консоль (обычное текстовое поле) при этом помещаются сразу все шаги выполнения алгоритма.

15) *public void kruskalNextStep(JTextArea console)* - метод, выполняющий следующий шаг алгоритма Краскала, в переданную в качестве параметра консоль (обычное текстовое поле) при этом помещается следующий шаг выполнения алгоритма.

16) *public void kruskalPrevStep(JTextArea console)* - метод, выполняющий предыдущий шаг алгоритма Краскала, в переданную в качестве параметра консоль (обычное текстовое поле) при этом помещается предыдущий шаг выполнения алгоритма.

17) *@Override protected void paintComponent(Graphics g)* - метод отвечает за отрисовку существующего графа с использованием библиотеки для отрисовки *Graphics2D*. Метод проходится по всем ребрам графа и отрисовывает линию и метку веса, помещенную в квадрат, с помощью встроенных функций библиотеки, далее проходится по всем вершинам и отрисовывает их в виде залитых кругов также с помощью встроенных функций библиотеки *Graphics2D*.

App.form - это форма, используемая для размещения и первичной настройки графических компонент приложения в IntelliJ Idea.

Класс Node

Класс *Node* представляет узел графа с различными атрибутами, такими как координаты, радиус и цвет.

Поля класса:

1) *private Point point* - координаты узла.
2) *private int radius* = 10 - радиус узла по умолчанию.
3) *private Color color* - цвет узла.

Методы класса:

1) *public Node(int x, int y)* - конструктор, который создает узел с заданными координатами (x, y) и цветом по умолчанию (черный).
2) *public Node(int x, int y, Color color)* - конструктор, который создает узел с заданными координатами (x, y) и цветом.
3) *public Point getPoint()* - возвращает координаты узла.
4) *public int getRadius()* - возвращает радиус узла.
5) *public Color getColor()* - возвращает цвет узла.
6) *public void setColor(Color color)* - устанавливает цвет узла.
7) *public void setPoint(int x, int y)* - устанавливает новые координаты узла.
8) *public boolean contains(int x, int y)* - проверяет, находится ли точка с координатами (x, y) внутри узла, используя радиус.
9) *@Override public boolean equals(Object obj)* - переопределяет метод *equals* для сравнения узлов по их координатам.
10) *@Override public int hashCode()* - переопределяет метод *hashCode*, чтобы он соответствовал методу *equals*.

Класс *Edge*

Класс *Edge* представляет ребро графа с различными атрибутами, такими как начальная и конечная вершины, цвет, толщина и метка.

Поля класса:

1) *private Node start* - начальная вершина ребра.
2) *private Node end* - конечная вершина ребра.
3) *private Color color* - цвет ребра.
4) *private int thickness* - толщина ребра.
5) *private String label* - метка ребра.

Методы класса:

1) *public Edge(Node start, Node end)* - конструктор, который создает ребро с заданными начальной и конечной вершинами, устанавливая цвет по умолчанию в черный, толщину в 2 и пустую метку.
2) *public Edge(Node start, Node end, Color color, int thickness, String label)* - конструктор, который создает ребро с заданными начальной и конечной вершинами, цветом, толщиной и меткой.
3) *public Node getStart()* - возвращает начальную вершину ребра.
4) *public Node getEnd()* - возвращает конечную вершину ребра.
5) *public Color getColor()* - возвращает цвет ребра.
6) *public int getThickness()* - возвращает толщину ребра.
7) *public String getLabel()* - возвращает метку ребра.
8) *public void setColor(Color color)* - устанавливает цвет ребра.
9) *public void setThickness(int thickness)* - устанавливает толщину ребра.
10) *public void setLabel(String label)* - устанавливает метку ребра.

Класс KruskalAlgorithm

Класс *KruskalAlgorithm* реализует алгоритм Крускала для поиска минимального остовного дерева (MST) в графе.

Поля класса:

1) *private List<Edge> edges* - список всех рёбер графа.
2) *private List<Edge> sort_edges* - список отсортированных рёбер графа.
3) *private List<Node> nodes* - список всех узлов графа.

Методы класса:

1) *public KruskalAlgorithm(Graph graph)* - конструктор, который инициализирует поля класса рёбрами и узлами из переданного графа. Также сортирует рёбра по весу (метка ребра).

2) *public ArrayList<Integer> KruskalOST()* - метод, который реализует алгоритм Краскала для поиска минимального остовного дерева в графе. Возвращает список индексов рёбер, которые входят в минимальное остовное дерево. Если граф несвязный, возвращает пустой список. Алгоритм прекращает работу, когда количество рёбер равняется количеству вершин, уменьшенному на 1. Если набор ребёр *KruskalOST* не образует дерево, метод возвращает пустой список.

Класс Graph

Класс *Graph* представляет граф с узлами и рёбрами и включает методы для их добавления, удаления и поиска.

Поля класса:

1) *private List<Node> nodes* - список всех узлов графа.

2) *private List<Edge> edges* - список всех рёбер графа.

3) *private static final int NODE_PROXIMITY_RADIUS* = 25 - радиус близости для поиска узлов.

Методы класса:

4) *public Graph()* - конструктор, который инициализирует пустые списки узлов и рёбер.

5) *public void addNode(int x, int y)* - добавляет узел с заданными координатами.

6) public void addNode(Node node) - добавляет заданный узел.

7) *public void removeNode(Node node)* - удаляет заданный узел и все связанные с ним рёбра.

8) *public void addEdge(Node start, Node end, Color color, int thickness, String label)* - добавляет ребро с заданными параметрами между двумя узлами, если такое ребро ещё не существует.

9) *public void removeEdge(Edge edge)* - удаляет заданное ребро.

10) *public Node findNode(int x, int y)* - ищет узел, который находится в заданных координатах или в радиусе близости.

11) *public Edge findEdge(int x, int y)* - ищет ребро, которое находится рядом с заданными координатами.

12) *public void clearGraph()* - очищает граф, удаляя все узлы и рёбра.

13) *public void loadGraph(String filename)* - загружает граф из файла. Сначала очищает текущий граф, затем добавляет узлы и рёбра из файла.

14) *public void saveGraph(String filename)* - сохраняет текущий граф в файл.

15) *public List<Node> getNodes()* - возвращает список узлов.

16) *public List<Edge> getEdges()* - возвращает список рёбер.

17) *private boolean isPointNearLine(int x, int y, int x1, int y1, int x2, int y2)* - проверяет, находится ли точка рядом с линией, заданной двумя координатами.

18) *private boolean isNearNode(Node node, int x, int y)* - проверяет, находится ли точка рядом с узлом в радиусе близости.

4. ТЕСТИРОВАНИЕ

4.1. Тестирование графического интерфейса

Тестирование базовых функций: добавление/удаление/перемещение вершин, добавление ребер и редактирование их веса, очистка поля.

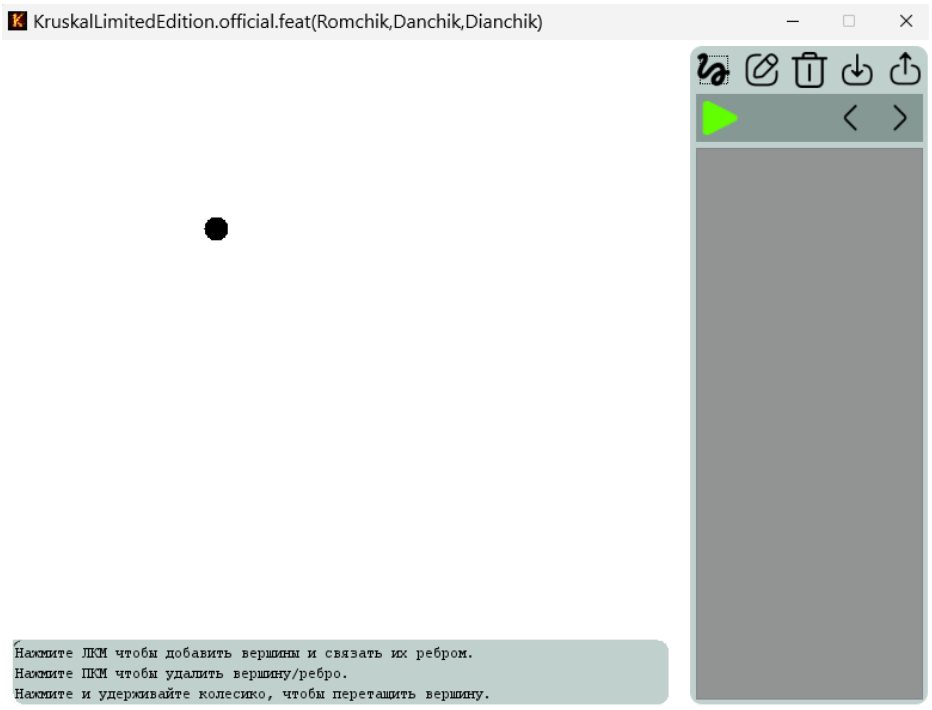

Рисунок 1 – Добавление вершины нажатием ЛКМ

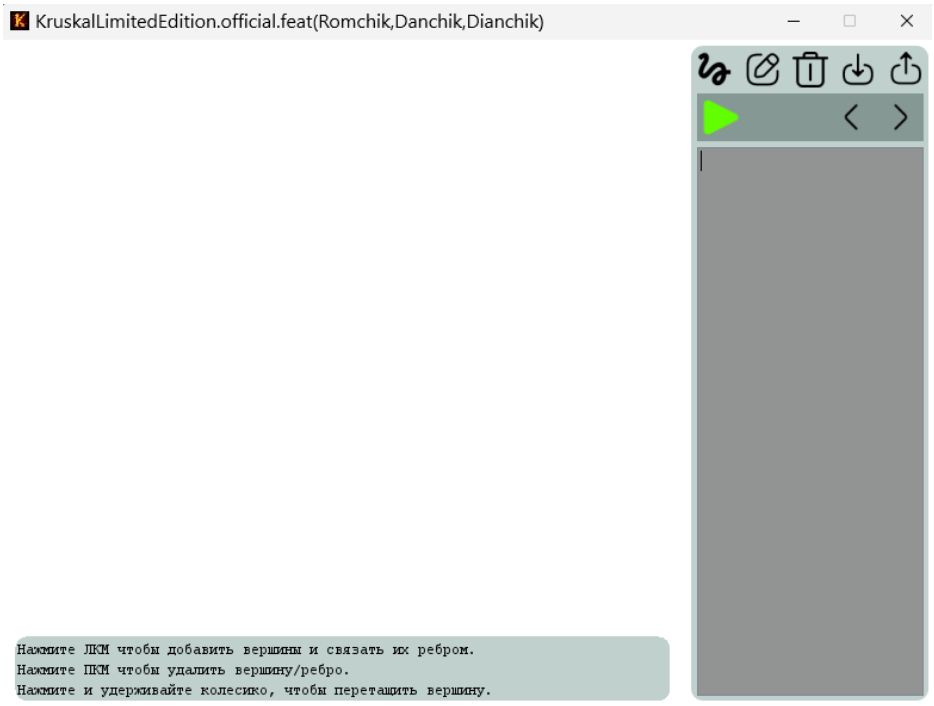

Рисунок 2 – Удаление вершины нажатием ПКМ

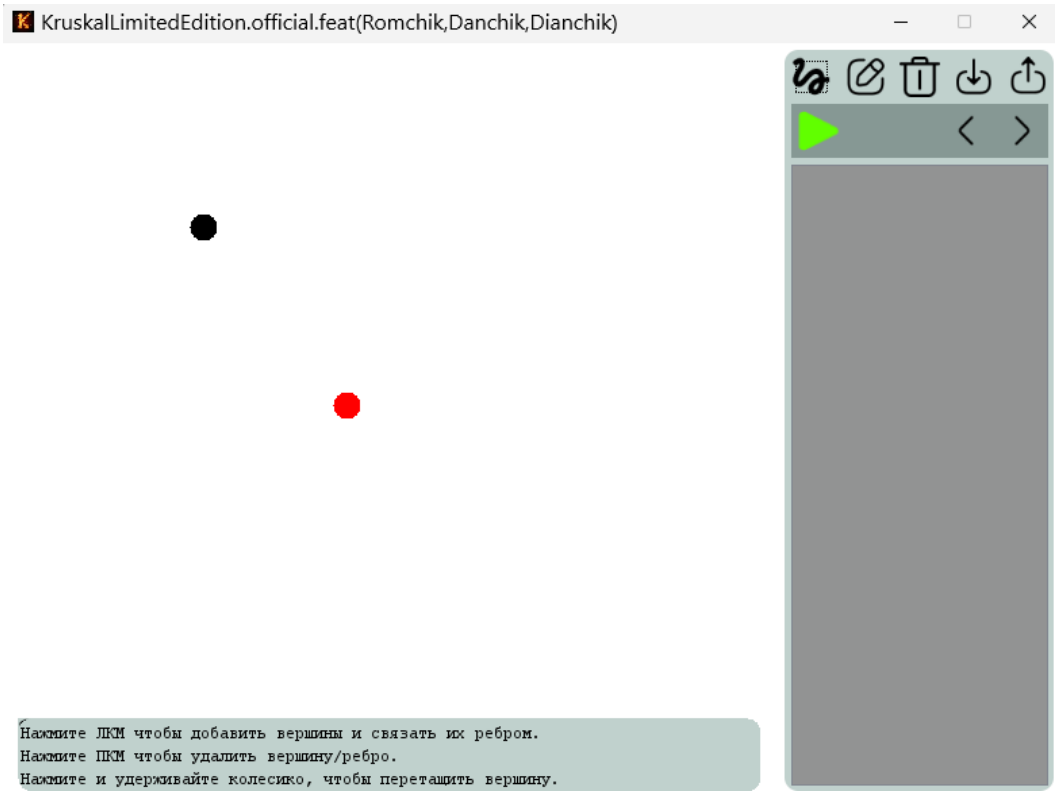

Рисунок 3 – Перемещение вершины нажатием на колесико

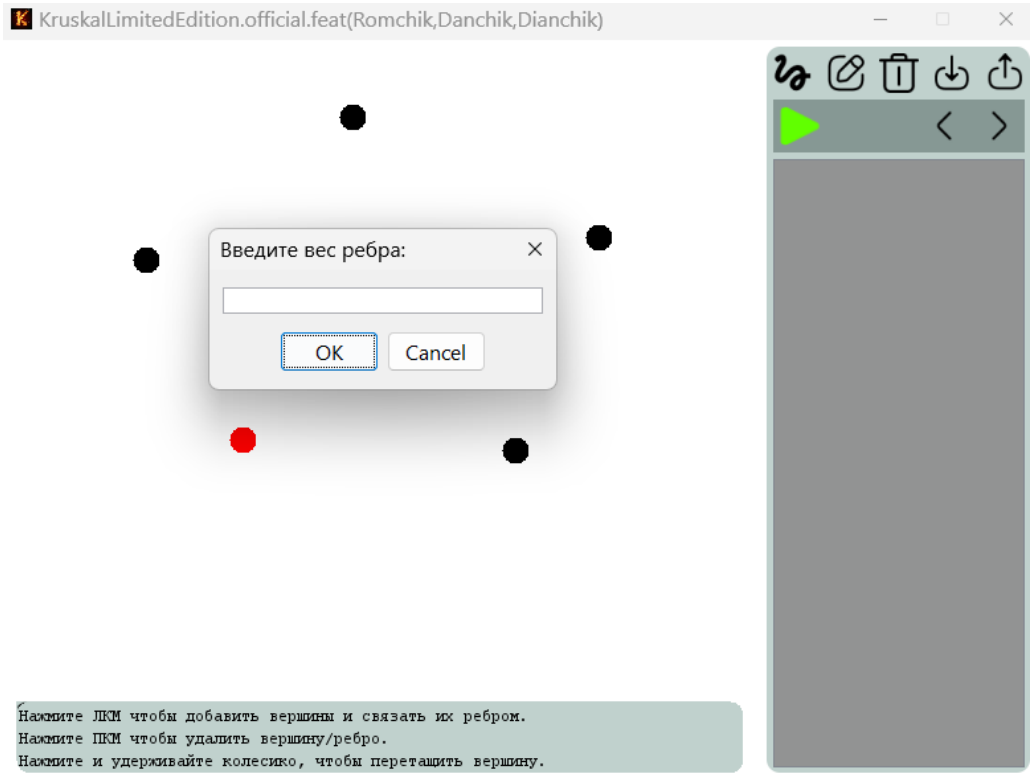

Рисунок 4 – При соединении двух вершин программа просит ввести вес ребра

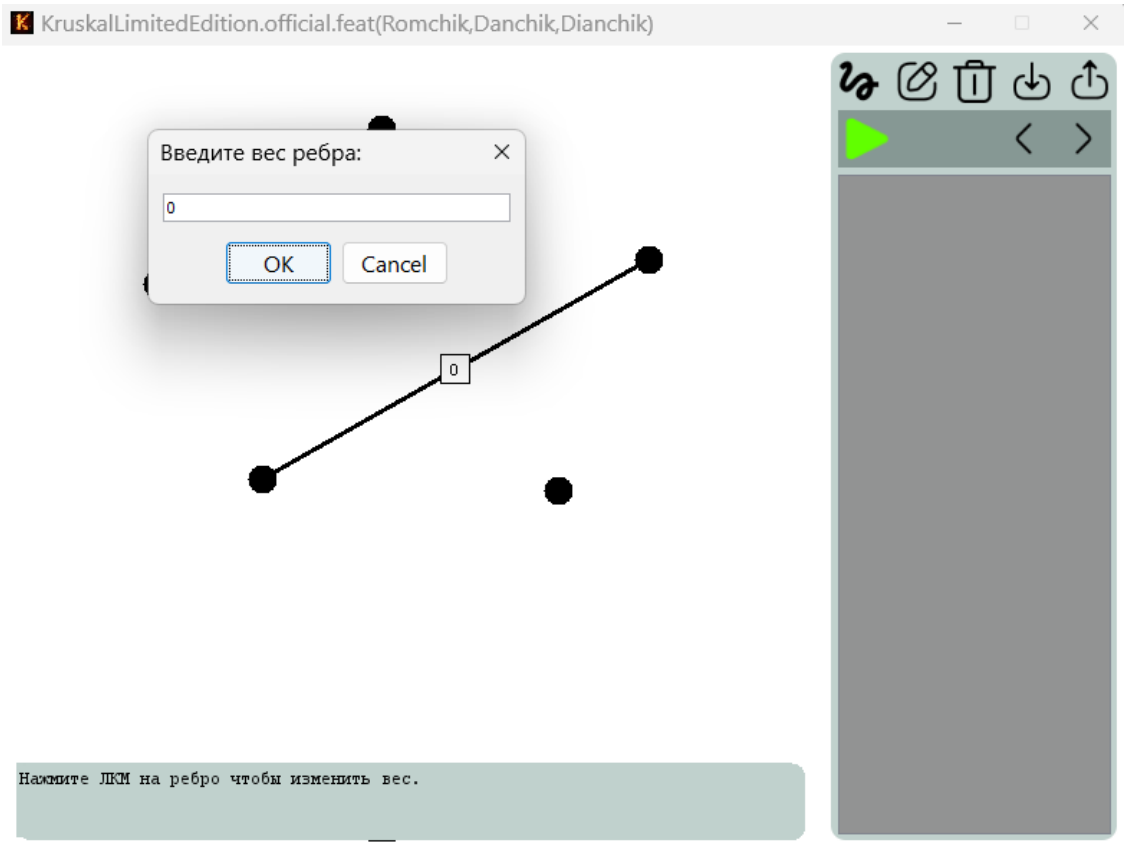

Рисунок 5 – При выборе инструмента для редактирования веса графа, обновляется подсказка и при нажатии на ребро можно изменить вес

Рисунок 6 – Очищение поля нажатием на кнопку с иконкой мусорного бака

4.2. Тестирование алгоритма

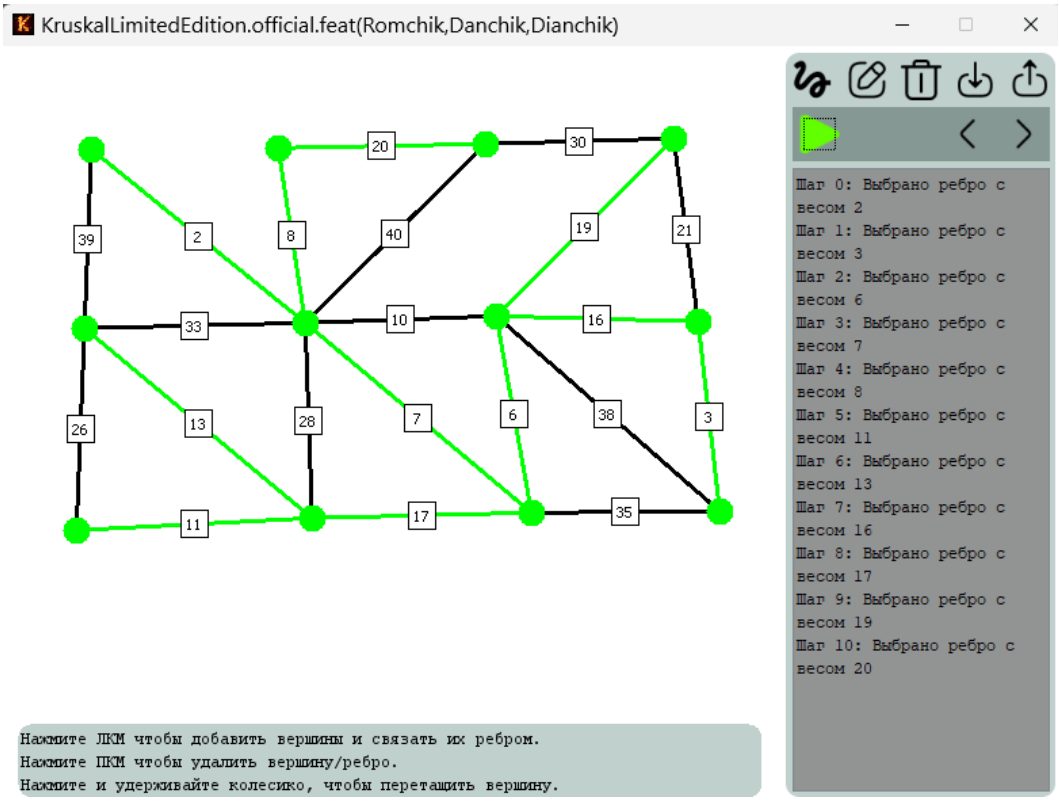

Рисунок 7 – Демонстрация правильной работы алгоритма

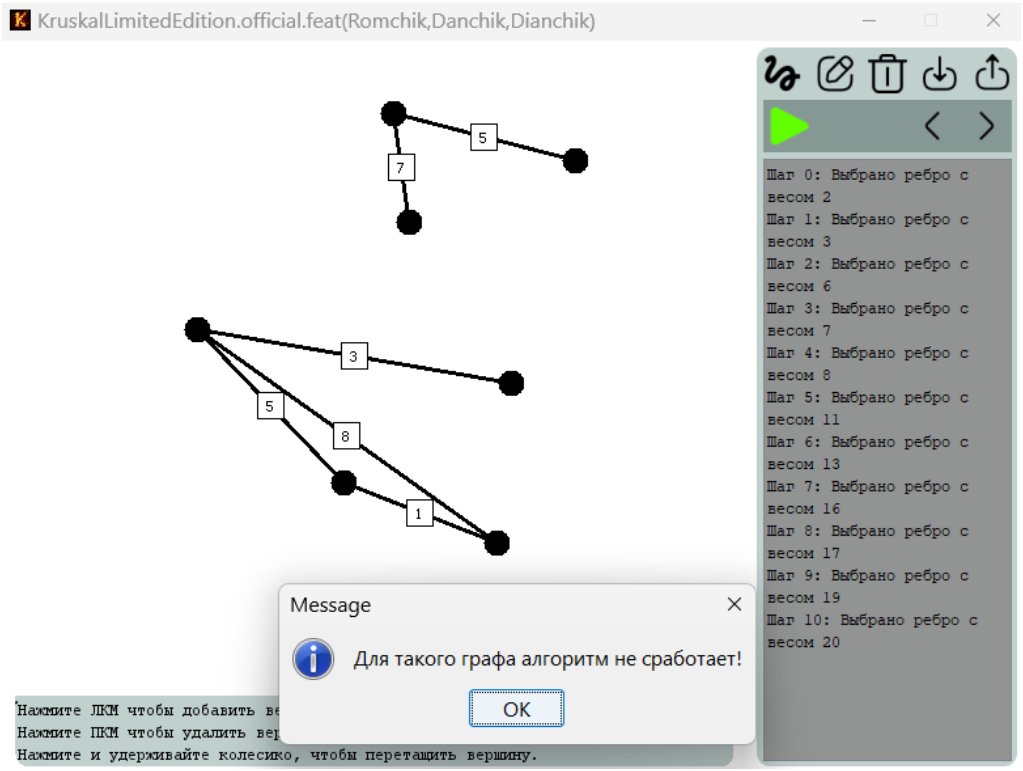

Рисунок 8 – При попытке запустить алгоритм на несвязном графе выводит сообщение

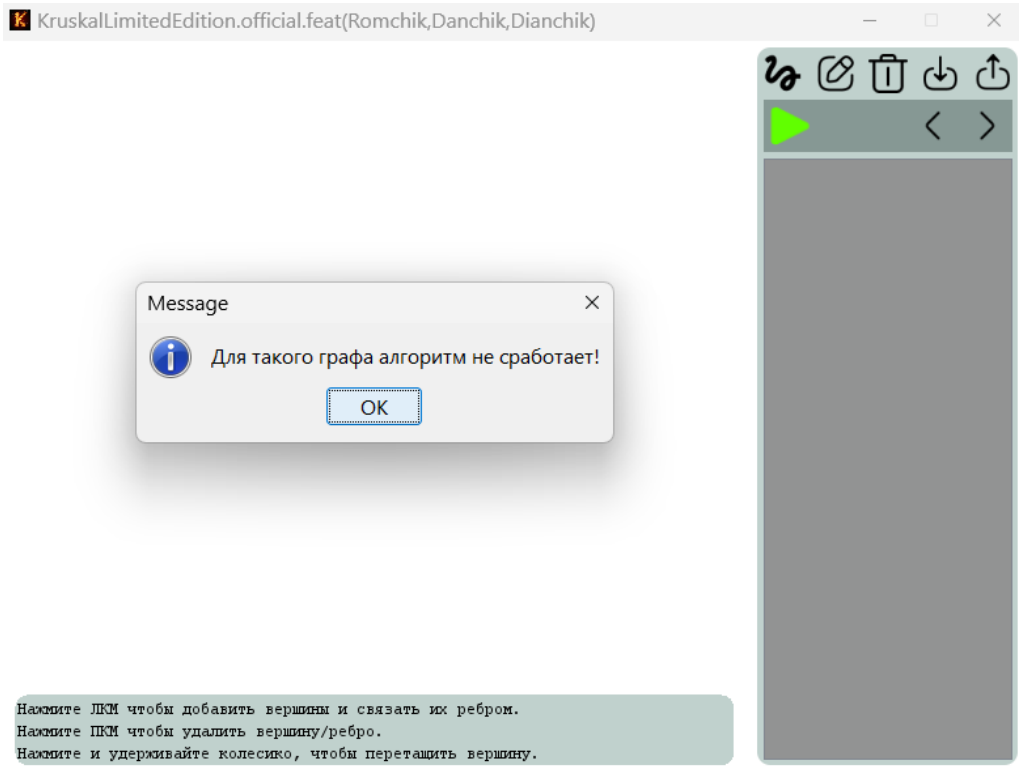

Рисунок 9 – На пустом графе алгоритм не сработает

ЗАКЛЮЧЕНИЕ

В ходе работы был изучен ранее неизвестный язык программирования Java. Для демонстрации и отработки полученных знаний был реализован графический интерфейс демонстрирующий работу алгоритма Краскала, что позволило систематизировать и отработать полученные знания.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Дискретная математика: учебник для студ. вузов / С. Н. Поздняков, С. В. Рыбин. – М.: Издательский центр «Академия», 2008. – 448 с.