**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МОЭВМ**

**ОТЧЕТ**

**по учебной практике**

**“Визуализация алгоритма поиска в глубину и алгоритма Дейкстры в ориентированном графе на языке Java.”**

|  |  |  |
| --- | --- | --- |
| Студент гр. 6382 |  | Швайко Т.А. |
| Студент гр. 6383 |  | Любчук Д.В. |
| Студентка гр. 6382 |  | Кузьмина А.С. |
| Руководитель |  | Фирсов М.А. |

Санкт-Петербург

2018

**ЗАДАНИЕ**

**на учебную практику**

|  |  |  |
| --- | --- | --- |
| Студент группы 6382 Швайко Т.А. | | |
| Студент группы 6383 Любчук Д.В. | | |
| Студентка группы 6382 Кузьмина А.С. | | |
| Тема практики: визуализация алгоритмов на языке Java | | |
| Задание на практику:  Командная итеративная разработка визуализатора алгоритмов на Java с графическим интерфейсом.  Алгоритмы: обход ориентированного графа в глубину, алгоритм Дейкстры. | | |
| Сроки прохождения практики: 27.06.2018 – 10.08.2018 | | |
| Дата сдачи отчета: 05.07.2018 | | |
| Дата защиты отчета: 06.07.2018 | | |
|  | | |
| Студент гр.6382 |  | Швайко Т.А. |
| Студент гр.6383 |  | Любчук Д.В. |
| Студентка гр.6382 |  | Кузьмина А.С. |
| Руководитель |  | Фирсов М.А. |

**АННОТАЦИЯ**

Целью данной учебной практики является получение практических навыков в визуализации алгоритмов, изучение и получение навыков использования языка программирования Java, приобретение навыка работы в команде (распределение ролей, написание каждым участником группы участка кода, за который он отвечает, и сборка данных кодов в исполняемую программу). В результате выполнения данной работы была представлена визуализация алгоритма обхода ориентированного графа в глубину, а также алгоритма Дейкстры с пользовательским интерфейсом.

**SUMMARY**

The goal of the training practice is to gain practical skills in visualizing Java algorithms and team skills (the distribution of roles, the writing by each participant of a group of a section of code for which it is responsible, and the assembly of code data into an executable program). The work shows the visualization of the algorithm DFS, also Dijkstra's algorithm in an oriented graph with the user interface.

**СОДЕРЖАНИЕ**

[**ВВЕДЕНИЕ** 5](#_Toc518543605)

[**1.** **ТРЕБОВАНИЯ К ПРОГРАММЕ** 6](#_Toc518543606)

[**1.1 Исходные Требования к программе** 6](#_Toc518543607)

[**1.1.1 Требования к вводу исходных данных** 6](#_Toc518543608)

[**1.1.2 Требования к выводу исходных данных** 6](#_Toc518543609)

[**1.1.3 Требования к визуализации** 6](#_Toc518543610)

[**1.2. Уточнение требований после сдачи прототипа** 7](#_Toc518543611)

[**1.3. Уточнение требований после сдачи 1-ой версии** 7](#_Toc518543612)

[**1.4. Работа программы до внесения изменений** 8](#_Toc518543613)

[**2.ПЛАН РАЗРАБОТКИ И РАСПРЕДЕЛЕНИЕ РОЛЕЙ В БРИГАДЕ** 11](#_Toc518543614)

[**2.1. План разработки** 11](#_Toc518543615)

[**2.2. Распределение ролей в бригаде** 12](#_Toc518543616)

[**3. ОСОБЕННОСТИ РЕАЛИЗАЦИИ** 13](#_Toc518543617)

[**3.1 Используемые структуры данных** 13](#_Toc518543618)

[**3.2 Основные классы и их методы** 14](#_Toc518543619)

[**3.3 Использованные возможности библиотеки** 20](#_Toc518543620)

[**3.4 UML – диаграмма классов** 21](#_Toc518543621)

[**4. ТЕСТИРОВАНИЕ** 22](#_Toc518543622)

[**4.1 Тестирование графического интерфейса** 22](#_Toc518543623)

[**4.2 Тестирование кода алгоритма (поиск в глубину)** 29](#_Toc518543624)

[**4.3 Тестирование кода алгоритма (Дейкстра)** 35](#_Toc518543625)

[**5. ЗАКЛЮЧЕНИЕ** 38](#_Toc518543626)

[**ПРИЛОЖЕНИЕ А. Исходный код** 39](#_Toc518543627)

**ВВЕДЕНИЕ**

Темой данной учебной практики является командная итеративная разработка визуализатора алгоритма на языке программирования Java. Цель данной учебной практики состоит в получении практических навыков в визуализации алгоритмов, изучении и получении навыков использования языка программирования Java, получении навыков работы в команде.

Задачей данной работы является написание визуализированного алгоритма обхода ориентированного графа в глубину, а также алгоритма Дейкстры с пользовательским интерфейсом (поиск в глубину – один из методов обхода графа, алгоритм Дейкстры – один из способов нахождения кратчайшего пути от одной вершины до всех остальных).

1. **ТРЕБОВАНИЯ К ПРОГРАММЕ**

**1.1 Исходные Требования к программе**

### **1.1.1 Требования к вводу исходных данных**

В качестве исходных данных предполагается создание пользователем при помощи графического интерфейса приложения графа посредством добавления вершин и ребер либо считывание информации о графе из заданного файла (если файл пуст или отсутствует, то поле представление графа при запуске будет пустым). Граф в файле представляется списком пар инцидентных вершин.

### **1.1.2 Требования к выводу исходных данных**

В качестве выходных данных предполагается графическое представление графа с раскраской всех посещенных вершин и соответствующих пройденных ребер, а также вывод пошаговой работы алгоритма в текстовое окно и вывод порядка обхода в глубину.

### **1.1.3 Требования к визуализации**

При запуске проекта должно создаваться отдельное окно, которое имеет 5 полей:

1. Поле графического представления графа.
2. Поле с выводом результата.
3. Поле с выводом промежуточных результатов.
4. Поле входных данных, содержащее представление исходного графа.
5. Поле интерфейса создания графа:

Предполагаемый вид интерфейса представлен на рис.1.1.

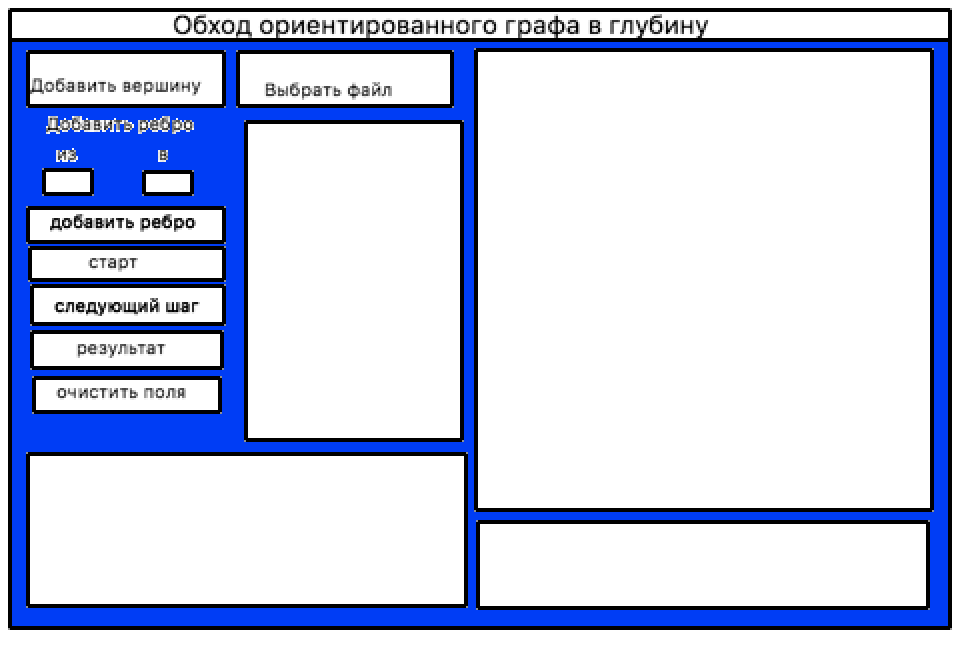


Рисунок 1.1 - Предполагаемый интерфейс программы.

**1.2. Уточнение требований после сдачи прототипа**

Для удобства пользователя было принято решение добавить кнопку «шаг назад» для возврата на предыдущий шаг алгоритма для того, чтобы пользователь мог полностью разобраться в работе алгоритма. Было решено объединить поля для вывода промежуточных данных, а также изменить цвета окрашиваемых вершин, для того, чтобы четко видеть элементы интерфейса. Также было принято решение добавить алгоритм Дейкстры на тот случай, если пользователю будет необходимо найти кратчайший путь между вершинами.

**1.3. Уточнение требований после сдачи 1-ой версии**

Было принято решение добавить всплывающие окна после окончания работы алгоритма, а также при неправильном вводе данных (которые при этом будут сообщать пользователю о том, что было сделано не так). А также было решено разделить кнопки по их назначению, чтобы пользователь во время ввода данных и во время работы алгоритма не отвлекался на соседние кнопки, которые на данном шаге не нужны.

## **1.4. Работа программы до внесения изменений**

Интерфейс программы после добавления вершин представлен на рис. 1.2.

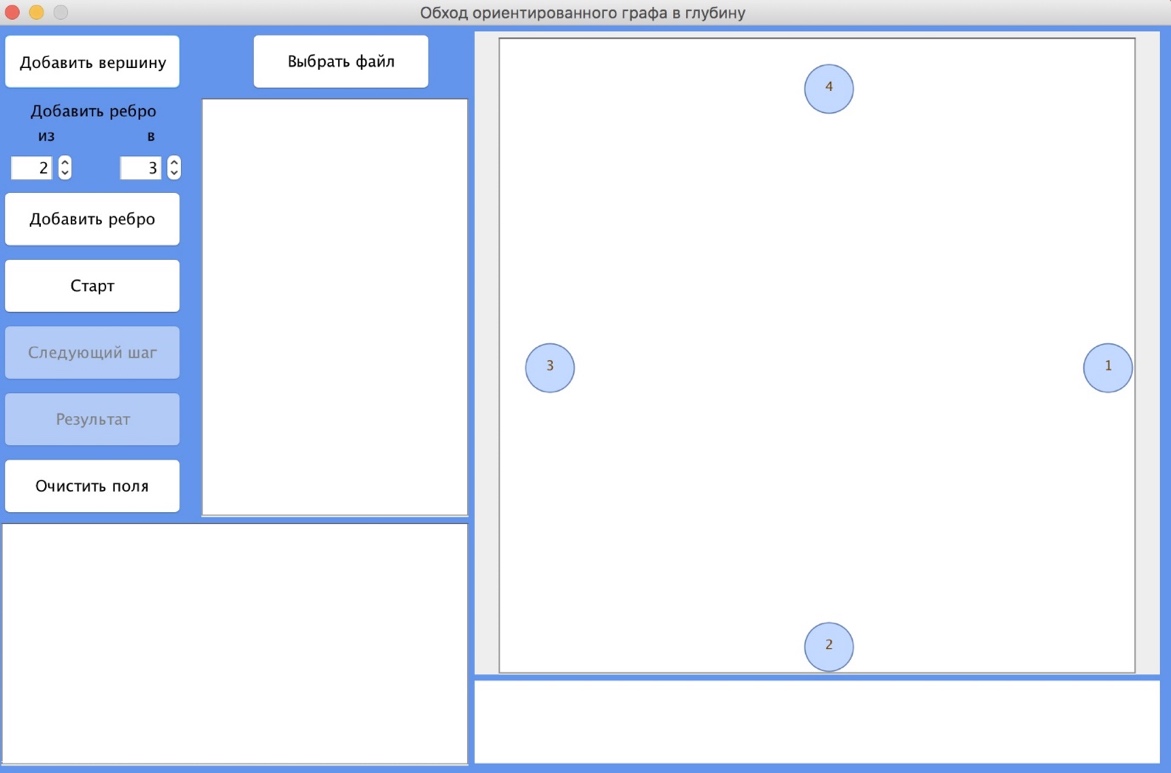


Рисунок 1.2 – добавление вершин

Интерфейс программы после добавления ребер представлен на рис. 1.3.

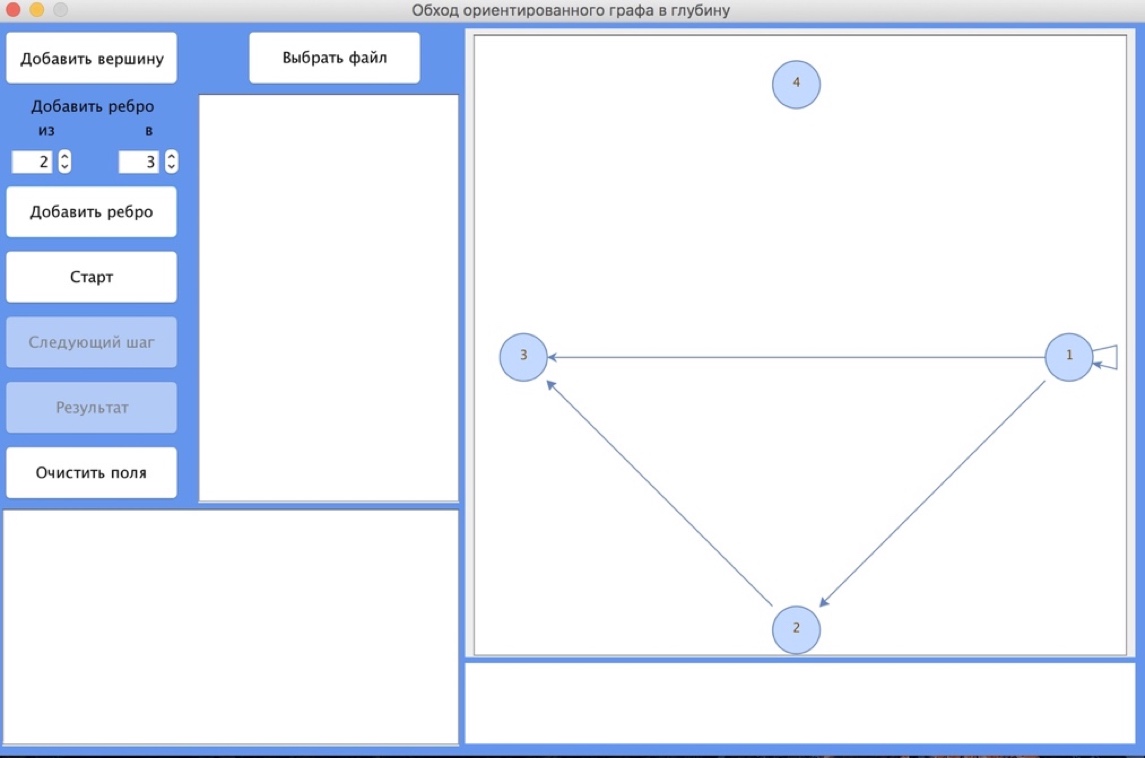


Рисунок 1.3 – добавление ребер

Интерфейс программы на 1-ом шаге алгоритма представлен на рис. 1.4.

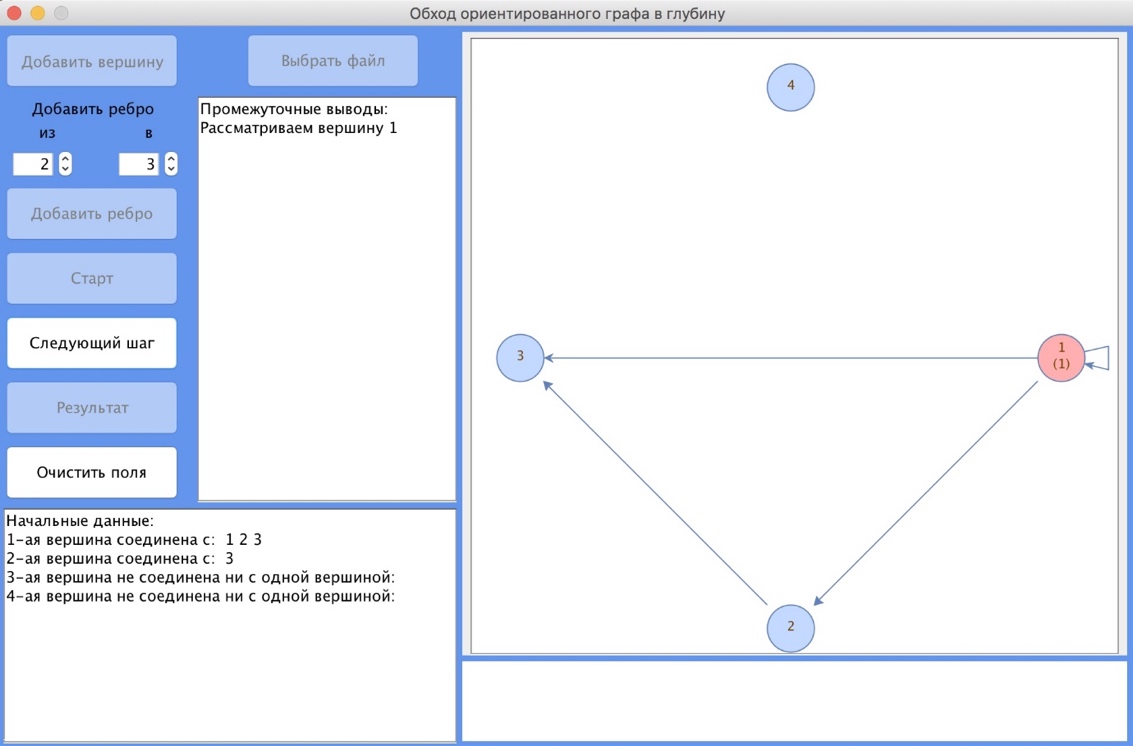


Рисунок 1.4 – 1-ый шаг алгоритма

Интерфейс программы на 4-ом шаге алгоритма представлен на рис. 1.5.

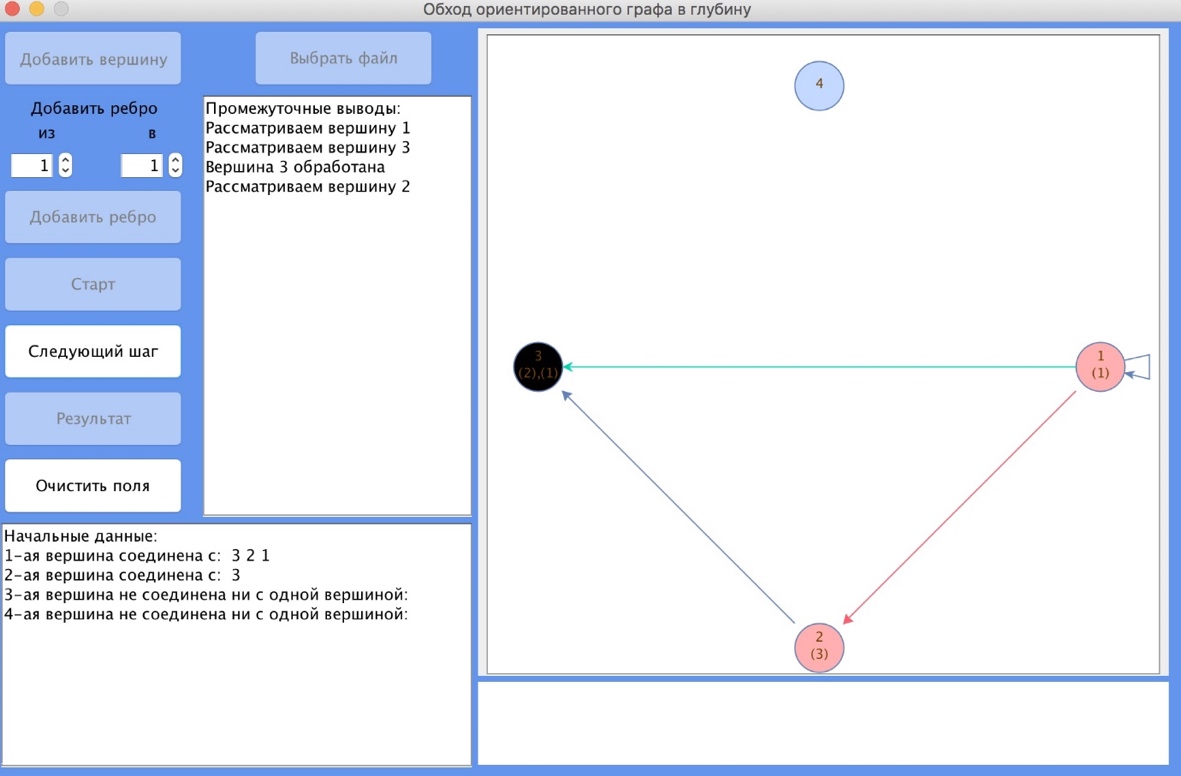


Рисунок 1.5 – 4-й шаг алгоритма

Интерфейс программы после завершения работы алгоритма представлен на рис. 1.6.

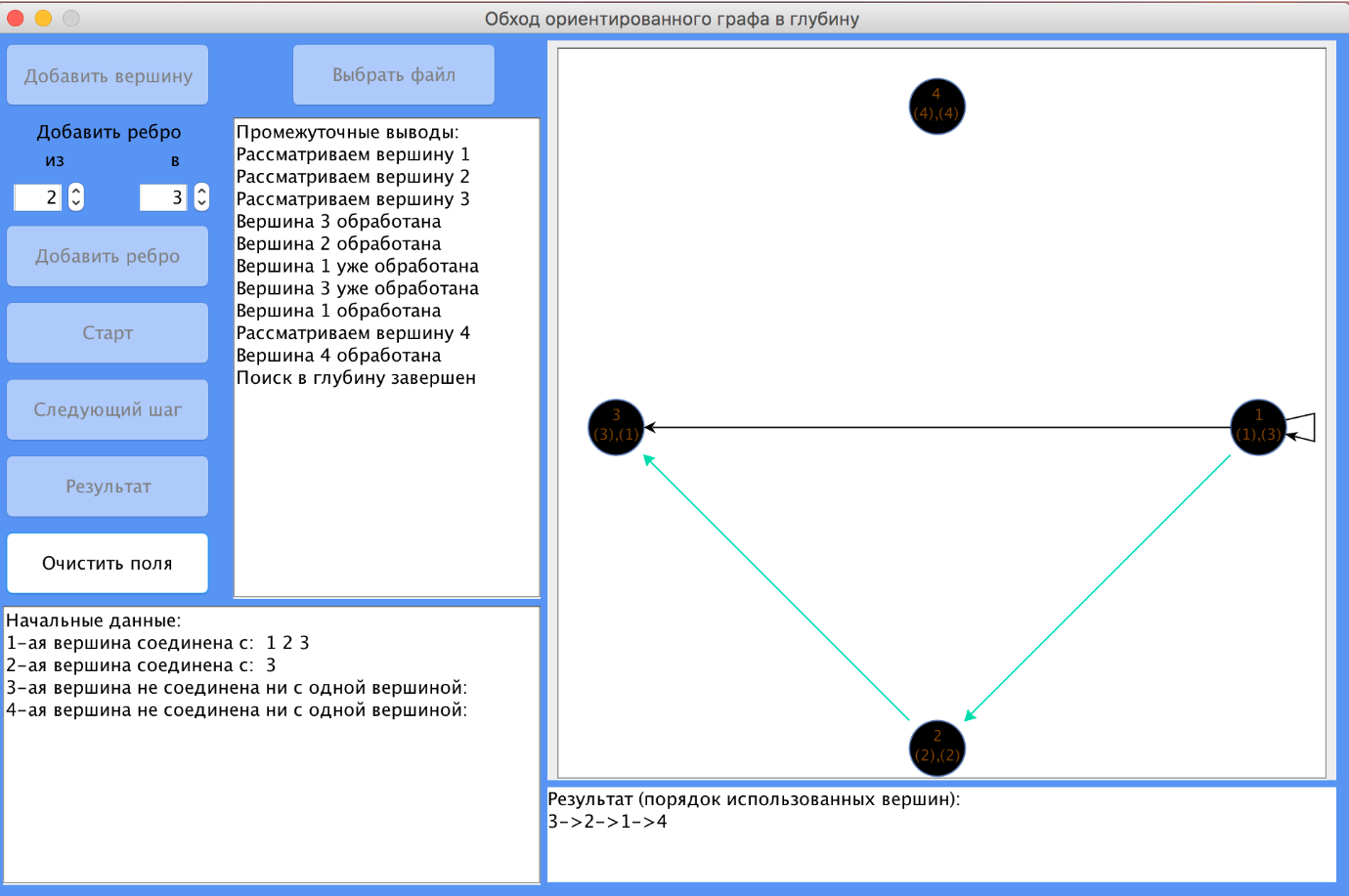


Рисунок 1.6 – завершение работы алгоритма

При нажатии кнопки «следующий шаг» происходят следующие действия: в поле промежуточных выводов указывается состояние вершины; на графе состояние вершины отображается соответствующим цветом (1 цвет – посещённая вершина, 2 цвет – использованная вершина) и номером порядка посещения и использования. При переходе к следующей вершине ребро, соединяющее вершины, подсвечивается в цвет 3. При возврате ребро будет окрашивается в цвет 4.

# **2.ПЛАН РАЗРАБОТКИ И РАСПРЕДЕЛЕНИЕ РОЛЕЙ В БРИГАДЕ**

**2.1. План разработки**

28.06.2018 – разработка спецификации

29.06.2018 – согласование спецификации с руководителем

29.06.2018 – разработка интерфейса

29.06.2018 - представление планируемого интерфейса (прототип), с отсутствующей функциональностью.

30.06.2018 – реализация графического представления графа, на основе входных данных с файла и осуществление работы алгоритма с выводом результата в соответствующее окно (частичная функциональность)

03.07.2018 - сдача 1-ой версии с добавлением частичной функциональности к пользовательскому интерфейсу;

03.07.2018 – добавление некоторых кнопок для удобства.

04.07.2018 – реализация алгоритма Дейкстры и соответствующей ему визуализации.

04.07.2018 - оформление отчета и тестирования.

05.07.2018 - сдача финальной версии. Представление проекта с полной функциональностью.

**2.2. Распределение ролей в бригаде**

В таблице 1 представлен состав бригады и распределение обязанностей.

Таблица 1 – распределение ролей

|  |  |
| --- | --- |
| Имя студента | Обязанность |
| Швайко Тимофей | Визуализация окна и реализация функциональности кнопок. |
| Любчук Денис | Реализация алгоритмов и визуализация представления графа. |
| Кузьмина Алёна | Тестирование алгоритма и программы и написание отчетной записки. |

**3. ОСОБЕННОСТИ РЕАЛИЗАЦИИ**

**3.1 Используемые структуры данных**

Граф для реализации алгоритма представляется в виде списка инцидентности. Для визуализации алгоритма граф представляется в виде четырех списков: список вершин прямого обхода, список вершин обратного обхода, список ребер, использованных при обходе, список ребер неиспользованных при обходе.

Для реализации графического интерфейса разрабатываемого приложения используется библиотека Swing, для изображения исходного графа и порядка его обхода – библиотека JGraphX.

Программа состоит из 6 классов: Main, SimpleGraph, Step, Node, GraphUtils, GraphVizualizer.

**3.2 Основные классы и их методы**

***Класс Main***

Класс Main отвечает за изображение графического интерфейса и выполнение программы в целом.

В основе работы класса лежат методы класса JFrame, запускаемые в отдельном потоке. Данные методы реализуют весь графический интерфейс, кроме изображения графа.

Кнопки, используемые в программе, представлены в таблице 2:

Таблица 2 – функциональность кнопок

|  |  |
| --- | --- |
| **Название кнопки** | **Функциональность** |
| addVertexButton | Добавление вершины |
| addEdgeButton | Добавление ребра |
| startButton | Начало алгоритма |
| nextButton | Следующий шаг алгоритма |
| resultButton | Результат |
| clearFieldButton | Очистить поля |
| prevButton | Предыдущий шаг алгоритма |
| dijkstraButton | Старт алгоритма Дейкстры |
| loadFromFileButton | Считать информацию из файла |

Метод addActionListener(new ActionListener) вызывается для добавления кнопкам функциональности.

Для изображения графа создан объект panel класса GraphVizualizer.

***Класс Node***

Класс Node содержит в себе поля для хранения промежуточных данных при реализации алгоритмов обхода.

Поля класса:

* private T value – номер вершины
* private int inTime – время входа в некоторую вершину
* private int outTime – время выхода из некоторой вершины

Методы класса описаны в таблице 3:

Таблица 3 – методы класса Node.

|  |  |
| --- | --- |
| **Метод** | **Предназначение** |
| Public Node(T value, int inTime, int outTime) | Метод присвоения ссылок, передаваемых значений (время входа и выхода). |
| public Node (T value) | Метод передачи значения. (Ничего не возвращает.) |
| Public T getValue() | Метод возвращения значения. |
| Public void setValue(T value) | Метод присвоения значения. (Ничего не возвращает.) |
| public int getInTime() | Метод возвращения времени входа. |
| public void setInTime() | Метод присвоения значения времени входа.  (Ничего не возвращает.) |
| public int getoutTime() | Метод возвращения времени выхода. |
| Public void setoutTime() | Метод присвоения значения времени выхода. (Ничего не возвращает.) |
| public Boolean equals(Object o) | Переопределение метода Equals. |
| public int hashCode() | Переопределение метода возврата (ключ, значение). |
| public String toString() | Переопределение метода toString для грамотного вывода промежуточных данных. |

***Класс Step***

Класс Step отвечает за корректные данные, которые описывают шаги алгоритмов.

Поля класса:

* private String description – описание шага
* private Node (T value) – описание номера вершины

Методы класса описаны в таблице 4:

Таблица 4 – методы класса Step.

|  |  |
| --- | --- |
| **Метод** | **Предназначение** |
| public Step(String description, Node<T> value) | Метод присвоения номера вершины, описания шага. (Возвращаемого значения нет.) |
| public String getDescription() | Метод возврата описания шага. |
| public void setDescription(String Description) | Метод присвоения шага описания. (Ничего не возвращает.) |
| public Node<T> getValue | Метод возврата номера вершины |
| public void SetValue(Node<T> Value) | Метод присвоения номера вершины. (Ничего не возвращает.) |
| public Boolean equals(Object o) | Переопределение метода Equals. |
| public int hashCode() | Переопределение метода возврата (ключ, значение). |
| public String toString() | Переопределение метода toString для грамотного вывода информации об описании шагов алгоритма. |

***Класс SimpleGraph***

*Класс SimpleGraph отвечает за визуализацию основного графа.*

**Поля класса:**

private int numberOfNode – количество создаваемых вершин

private List<Pair<Integer, Integer» edges = new ArrayList<>() – список инцидентных вершин.

Методы класса представлены в таблице 5:

Таблица 5 – методы классе SimpleGraph.

|  |  |
| --- | --- |
| **Метод** | **Предназначение** |
| Public void addNode() | Метод для увеличения числа вершин. (Ничего не возвращает.) |
| Public void delNode() | Метод для уменьшения числа вершин. (Ничего не возвращает.) |
| Public void addEdje(int from, int to) | Метод для добавления ориентированного ребра. (Принимает номер вершины, откуда исходит ребро и номер вершины, в которую входит ребро соответственно; ничего не возвращает.) |
| public void addEdge(JSpinner from, JSpinner to) | Аналогичен методу выше за исключением того, что номера вершины-истока и стока для ребра принимает не числом, а из «спиннера» |
| Public arrayList<Pair<Double,Double>> points = new ArrayList<>(); | Метод для размещения вершин графа на исходном поле для сохранения между ними равных расстояний (вершины размещаются по окружности). (Принимает и возвращает координаты центра вершин) |
| Public int getNumberOfNode() | Метод для возвращения количества вершин. |
| Public List<Pair<Integer, Integer» getEdges() | Метод для возвращения количества ориентированных ребер. |

Окончание таблицы 5 – методы классе SimpleGraph.

|  |  |
| --- | --- |
| Public void Clear() | Метод для очистки основного поля. (Ничего не возвращает.) |

***Класс GraphUtils:***

Класс GraphUtils отвечает за реализацию основных алгоритмов.

Методы класса представлены в таблице 6:

Таблица 6 – методы класса GraphUtils.

|  |  |
| --- | --- |
| **Метод** | **Предназначение** |
| public static <T> List<Step<T» deepFirstSearch(Graph<Node<T» graph, Node<T> startNode) | Метод, в котором описывается алгоритм работы поиска в глубину.  (Передаваемые значения - граф, начальная вершина соответственно. Возвращает список для сохранения истории шагов алгоритма.) |
| public static <T> List<Step<T» dijkstra(Graph<Node<T» graph, Node<T> startNode) | Метод, в котором описывается работа алгоритма Дейкстры.  (Передаваемые значения - граф, начальная вершина соответственно. Возвращает список для сохранения истории шагов алгоритма.) |
| private static <T> void saveToHistory(List<Step<T>> history, Node<T> current, String s) | Метод для внесения изменений в историю выполнения алгоритма. (Передаваемые значения – история, текущая вершина и строка, которую необходимо добавить к истории; ничего не возвращает.) |
| private static <T> Node<T> copy(Node<T> current) | Метод для создания копии истории. (Передаваемое значение –текущая вершина; возвращает элемент истории.) |

***Класс GraphVisualizer:***

Класс GraphVizualizer отвечает за визуализацию алгоритмов.

Методы класса представлены в таблице 7:

Таблица 7 – методы класса GraphVizualizer.

|  |  |
| --- | --- |
| **Метод** | **Предназначение** |
| public GraphVisualizer() | Устанавливает поле для размещения графа. (Ничего не возвращает.) |
| public void visualizeGraph(SimpleGraph simpleGraph, List<Step<Integer» history, int step, JTextArea info) | Метод визуализации алгоритмов.  (Передаваемые значения - граф, список шагов алгоритма, шаг, поле вывода соответственно; ничего не возвращает) |

**3.3 Использованные возможности библиотеки**

В данной работе для визуализации графа была использована сторонняя библиотека JGraphX.

*Использованные поля:*

mxIGraphModel -определение модели графа для установки ее в граф

mxGraph –создание объекта графа

*Использованные методы для графа:*

setModel - установка модели графа

setCellStyles - покраска вершины

getModel().setValue - установка текста внутрь вершины

getModel().setStyle - покраска ребра

graph.insertEdge - создание ребер

*Использованные методы для модели графа:*

beginUpdate - фиксирует начало изменения и конец изменен

endUpdate - фиксирует конец изменения

## **3.4 UML – диаграмма классов**

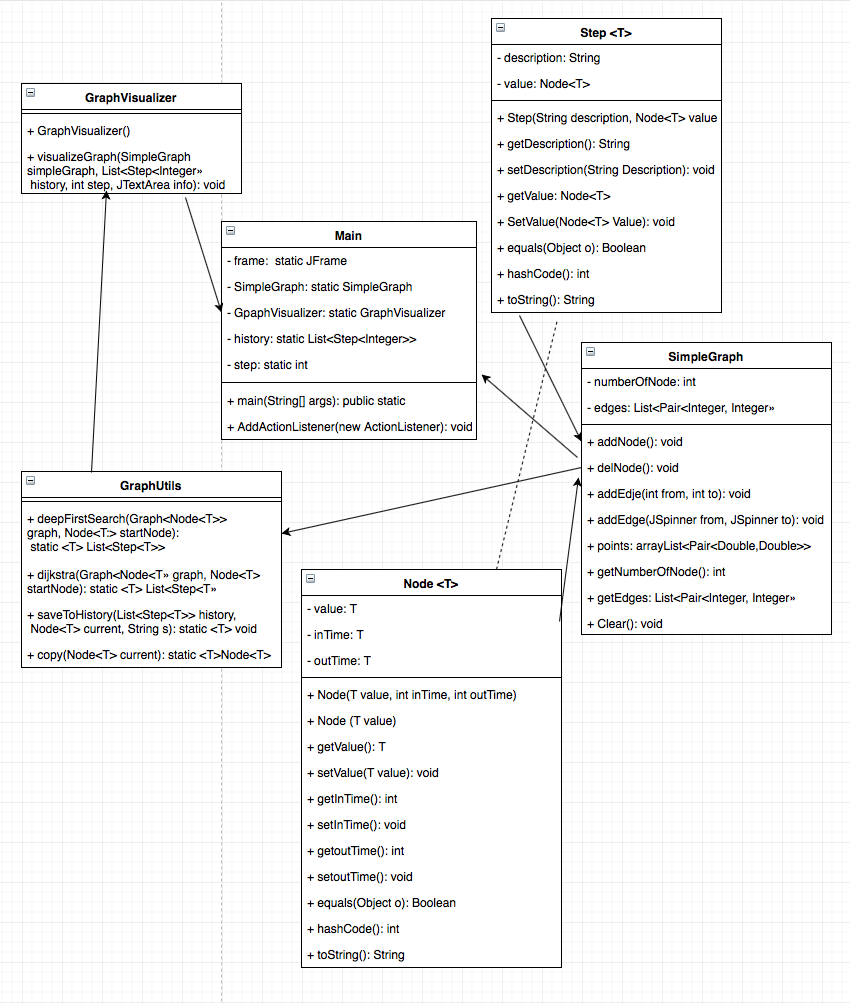


Рисунок 3.1 – uml-диаграмма классов.

**4. ТЕСТИРОВАНИЕ**

**4.1 Тестирование графического интерфейса**

При запуске программы открывается окно программы, имеющее интерфейс, представленный на рисунке 4.1. (Изначально кнопки из правого столбца, отвечающие за работу алгоритма, заблокированы, они открываются после введения данных.)

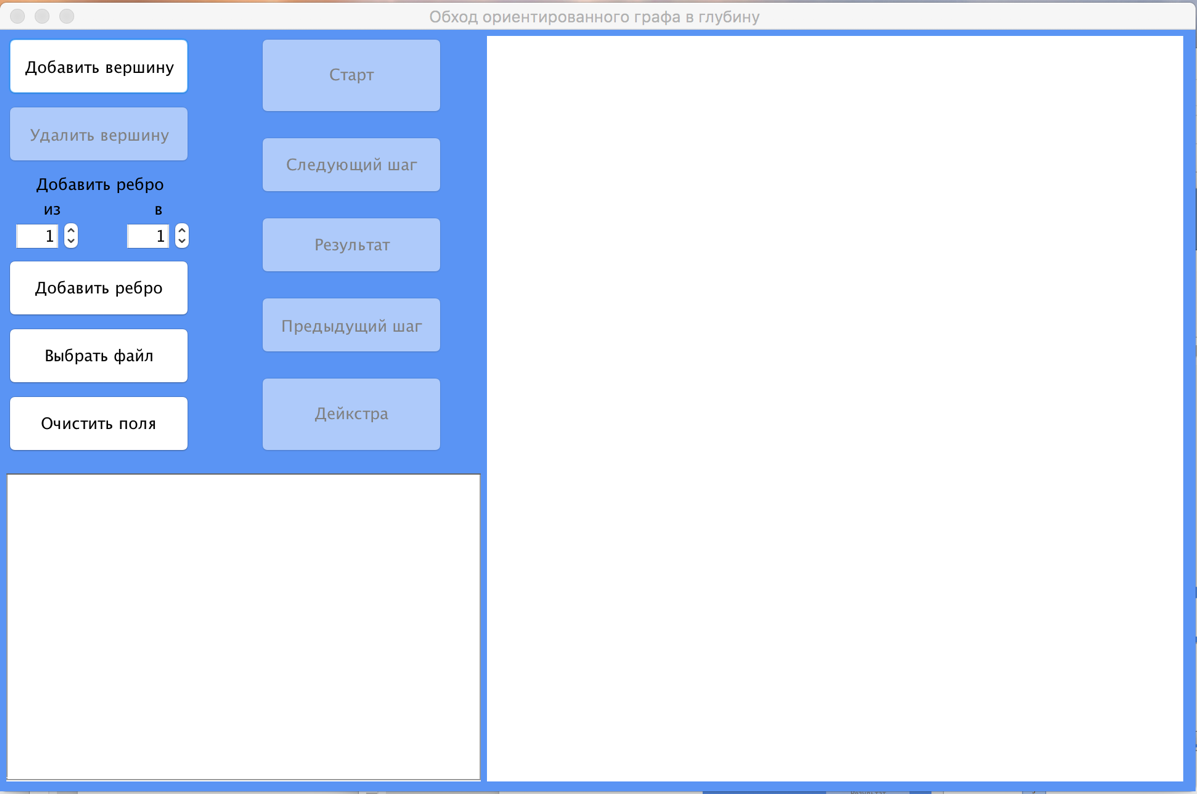


Рисунок 4.1 – интерфейс программы

Чтобы начать работу с программой, нужно ввести данные. Сделать это можно двумя способами: считать данные из файла или построить ориентированный граф при помощи кнопок. Чтобы считать данные из файла необходимо нажать кнопку «Выбрать файл» и указать файл, из которого необходимо считать данные. После этого появится изображение графа. Если в файле некорректные данные (например последовательность букв), программа выдаст следующее предупреждение (представлено на рисунке 4.2). Если файл пустой, то предупреждения не будет, но и никакого изображения графа не появится.

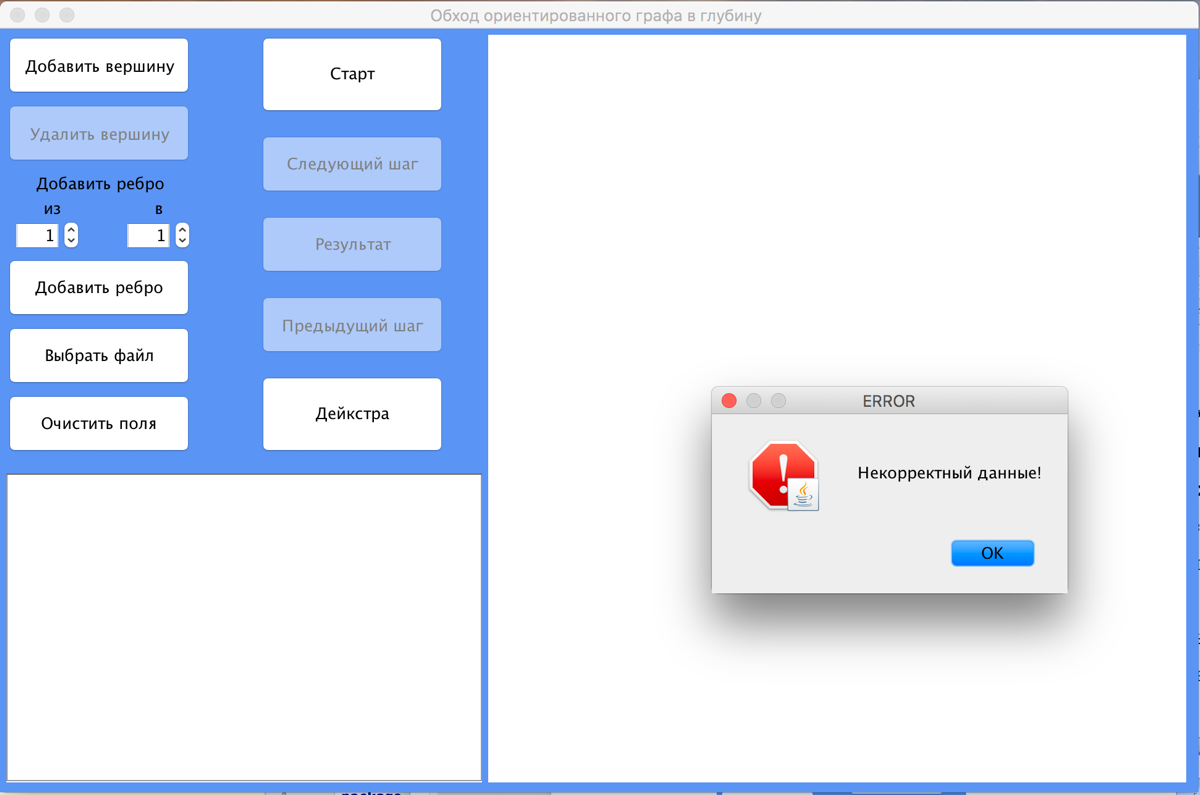


Рисунок 4.2 – интерфейс при некорректных данных в файле

Чтобы непосредственно самим построить граф, необходимо сначала добавить вершины при помощи кнопки «Добавить вершину» (вершины добавляются по одной, нумерация проводится согласно порядку добавления). Результат операции приведен на рисунке 4.3.

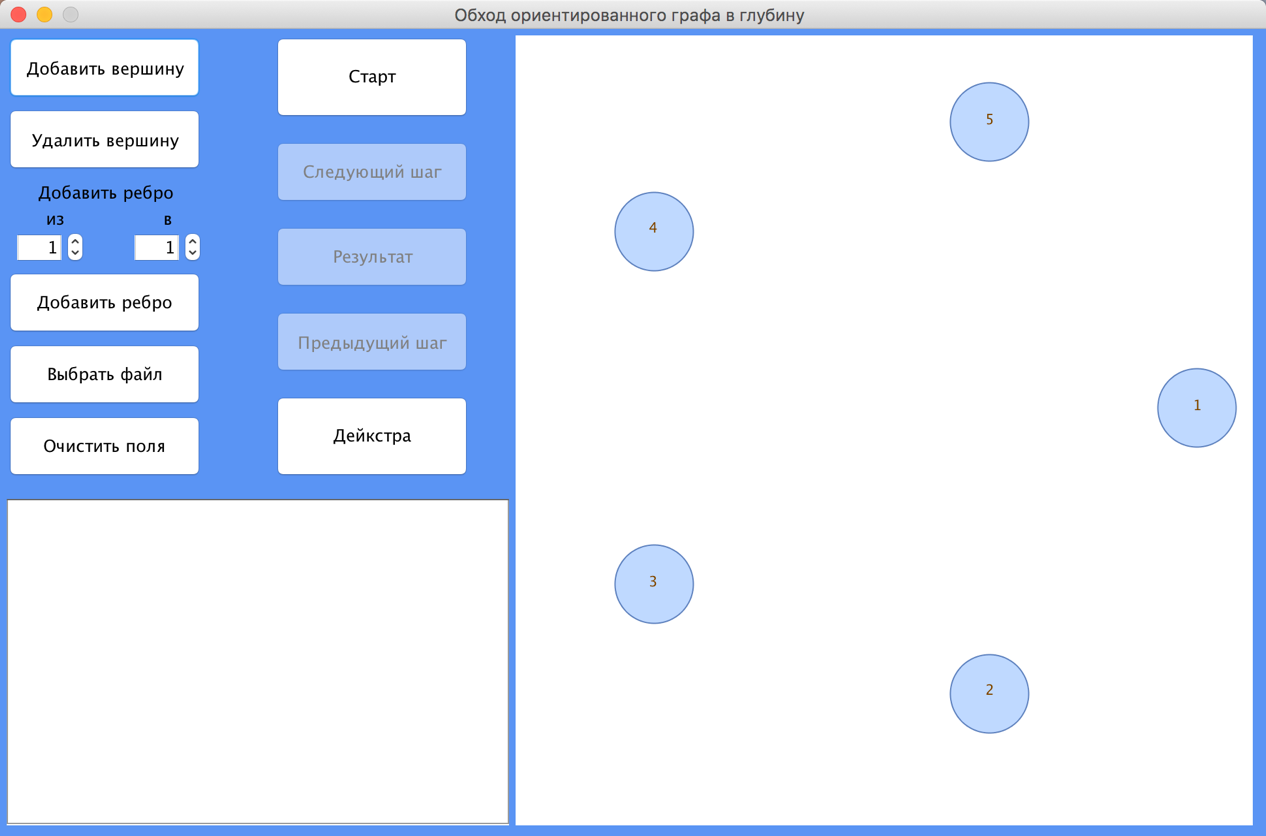
**

Рисунок 4.3 – интерфейс после добавления вершин

При желании пользователь может перемещать вершины для своего удобства. Пример приведен на рисунке 4.4.

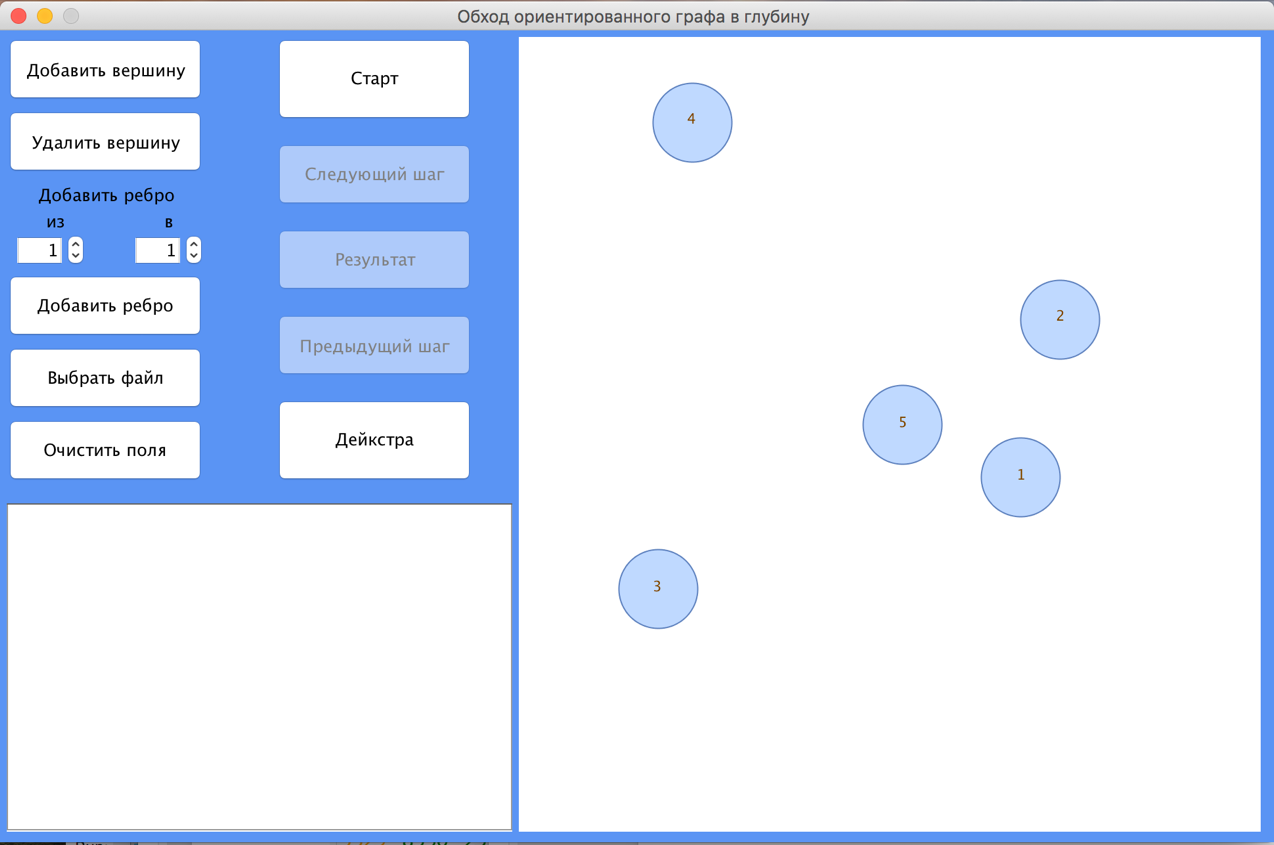


Рисунок 4.4 – интерфейс после перемещения вершин

Если пользователь ввел лишние вершины, он их может удалить про помощи кнопки «Удалить вершину», при этом удаление происходит в порядке, обратном добавлению вершин. Удалять можно, пока количество вершин не станет равным единице, после этого при нажатии «Удалить» ничего происходить не будет.

После того, как были добавлены вершины, необходимо добавить ребра при помощи кнопки «Добавить ребро», определив перед этим 2 вершины: из которой выходит ребро и в которую входит. Вершины устанавливаются при помощи кнопок последовательного выбора, подписанных «из» и «в» (верхнее ограничение – 50, нижнее - 1). При выставлении несуществующей вершины всплывет предупреждение «Некорректный ввод(не существует такой вершины)». Результаты корректной и некорректной вставки ребер представлены на рисунках 4.5 и 4.6 соответственно.

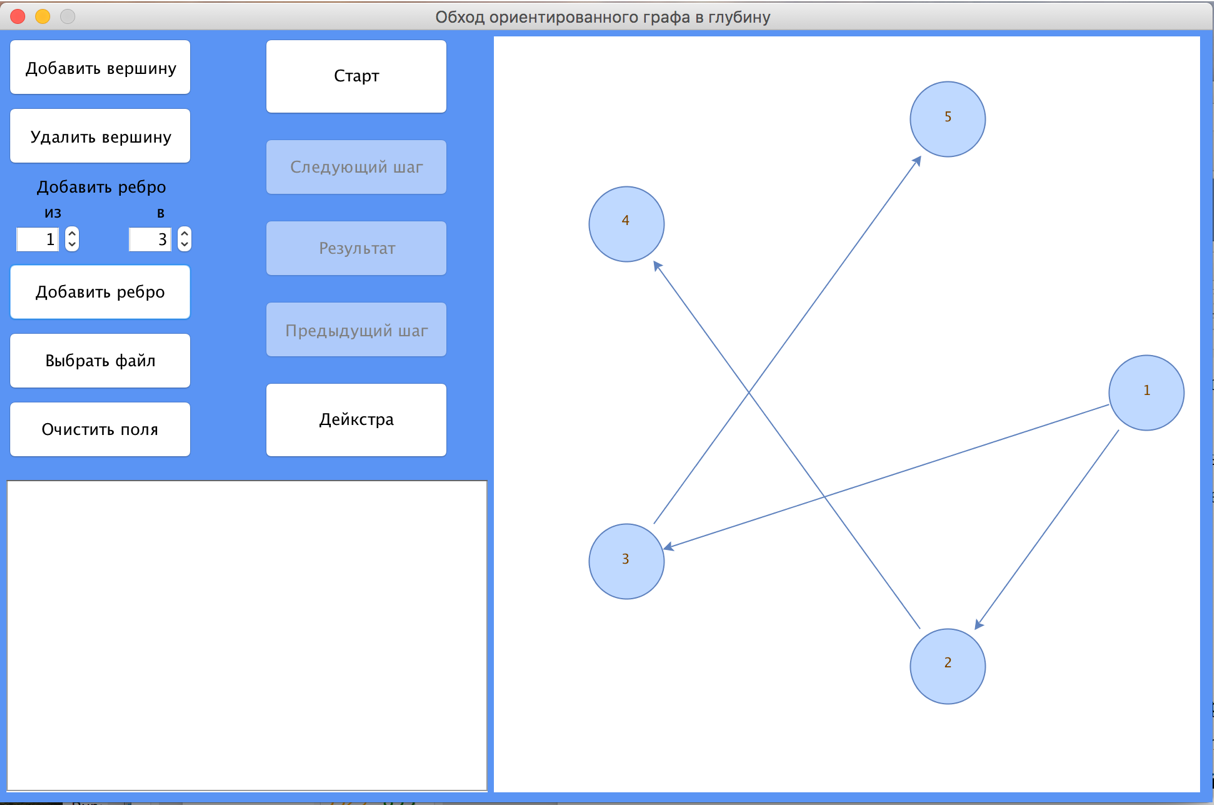


Рисунок 4.5 – интерфейс после вставки ребер

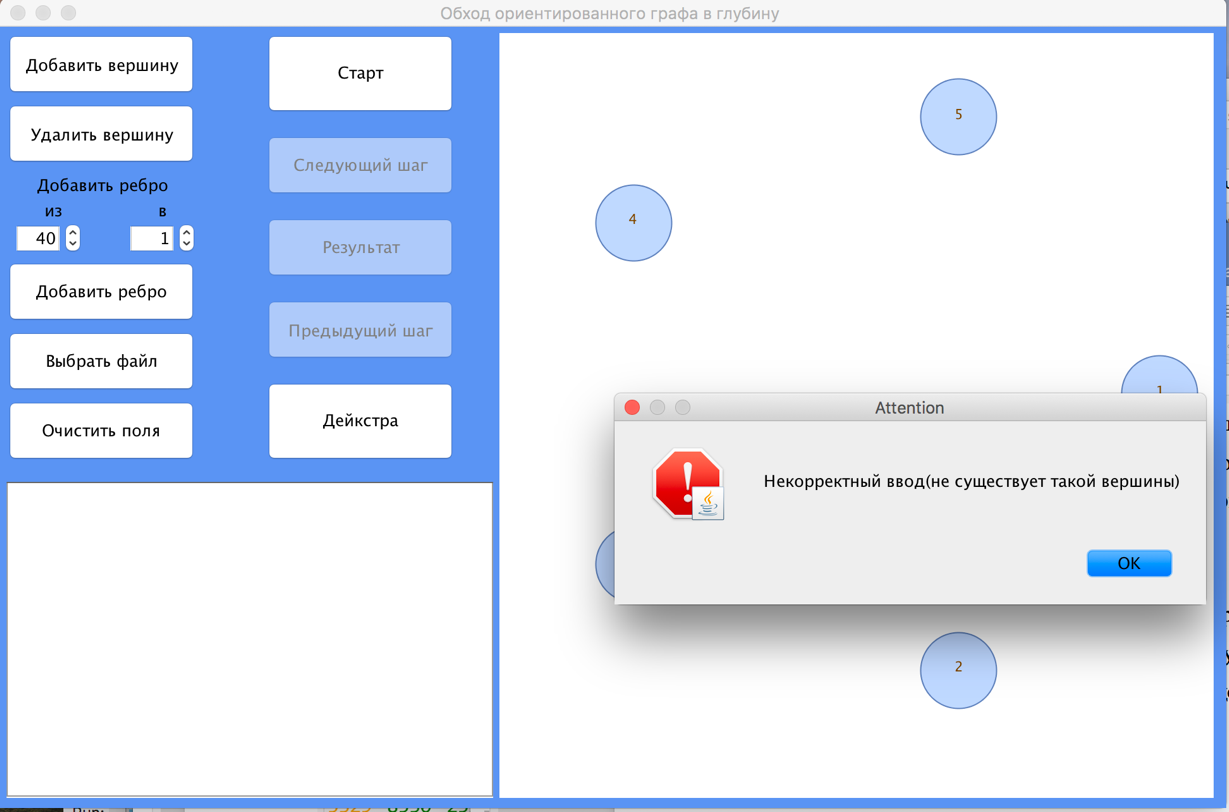


Рисунок 4.6 – интерфейс после попытки некорректной вставки ребра

При помощи кнопки «Очистить поля» пользователь может очистить поле, в котором выводится изображение графа, а также поле для вывода промежуточных данных о выполнении программы (если выполнялся алгоритм).

После введения данных пользователь может либо работать с алгоритмом поиска в глубину (кнопка «Старт»), либо с алгоритмом Дейкстры (кнопка «Дейкстра»), который позволяет найти кратчайший путь из первой вершины в любую другую. При нажатии любой из данных двух кнопок блокируются все кнопки кроме «Следующий шаг» (отвечает за пошаговое выполнение алгоритма), «Результат» (отвечает за выдачу конечного результата работы алгоритма сразу) и «Предыдущий шаг» (отвечает за отмену последнего шага).

Рассмотрим подробнее функциональность данных кнопок. Используем начальный граф с рисунка 4.5.

Результат выполнения на первом шаге (после нажатия кнопки «Следующий шаг» один раз) представлен на рисунке 4.7. Можно увидеть, что первая рассматриваемая вершина подсвечивается зеленым. В поле вывода промежуточных данных появляются комментарии к текущему шагу.

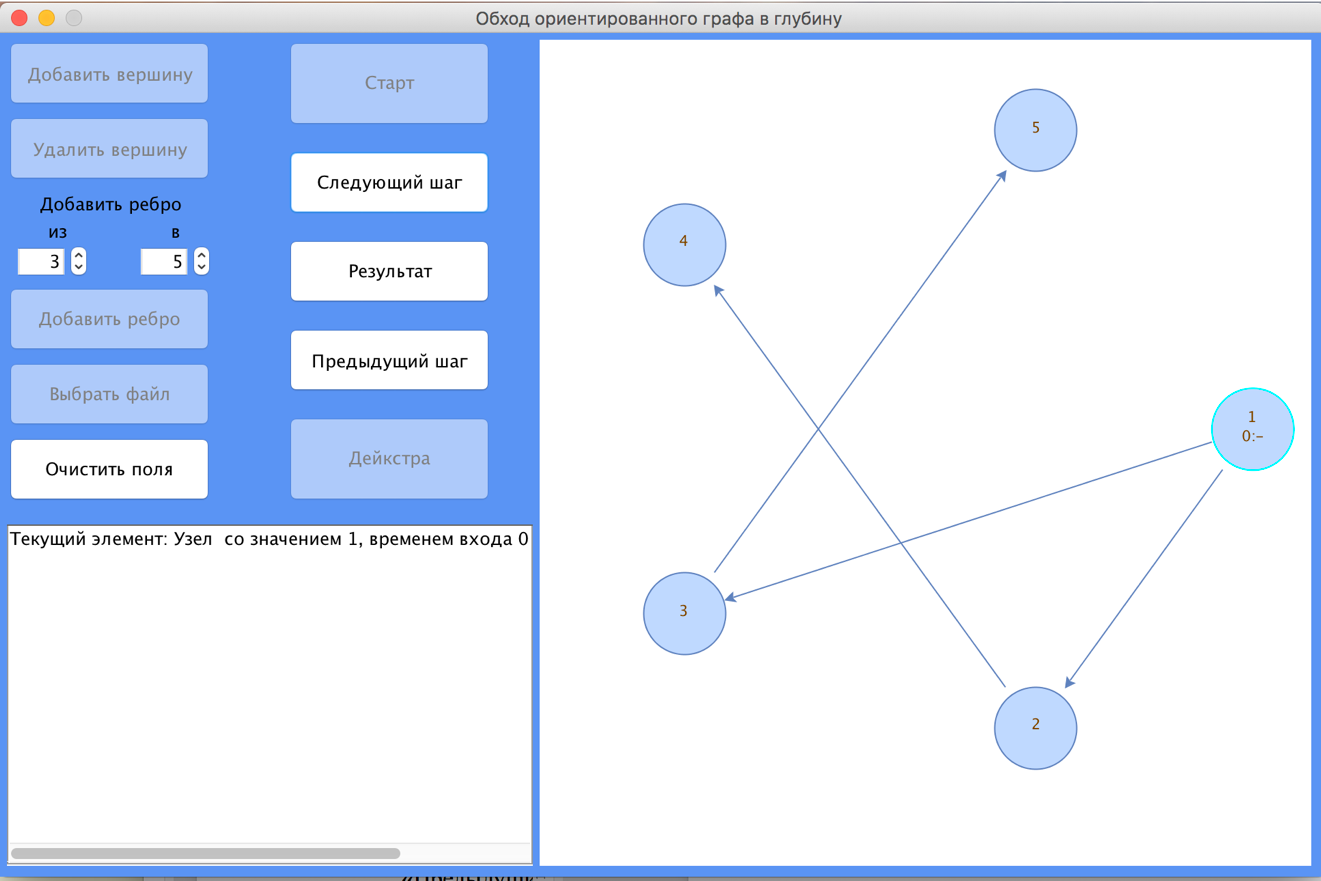


Рисунок 4.7 – интерфейс после 1-го шага

Нажмем кнопку следующего шага еще раз. Можно заметить, что вершина, в которую можно перейти из прошлой, также подсвечена зеленым, а в низу круга вершины появилось время входа (как и в первой рассматриваемой вершины на первом шаге). Результат представлен на рисунке 4.8.

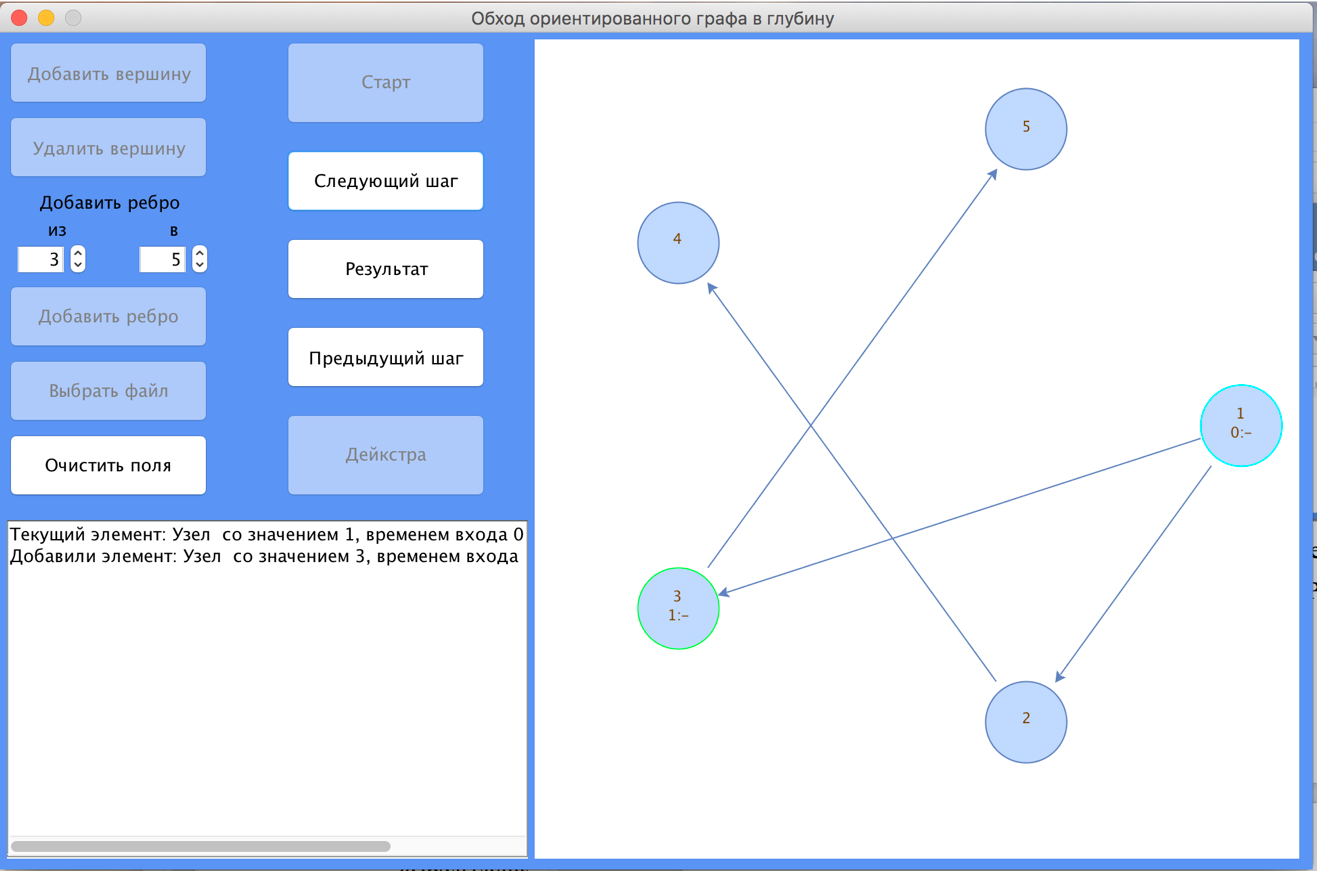


Рисунок 4.8 – интерфейс после 2-го шага

Проходим еще несколько шагов, дойдя из вершины 3 в вершину 5 и поняв, что из вершины 5 никуда попасть нельзя. При этом вершина 5, будучи до этого с зеленым контуром, как 1-я или 3-я вершины, приобретает красный цвет. И кроме времени входа в низу круга вершины появляется время выхода. Результат представлен на рисунке 4.9.

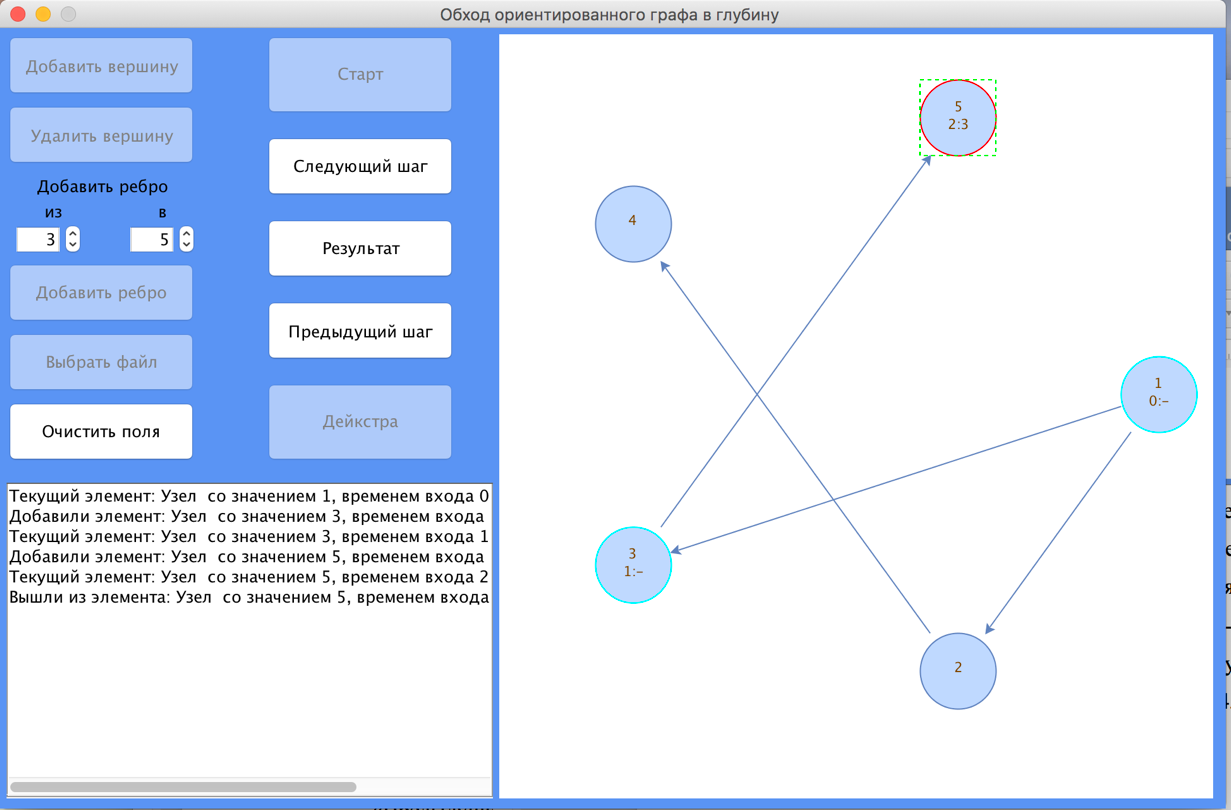


Рисунок 4.8 – интерфейс после 6-го шага

Проверим кнопку «предыдущий шаг». По логике, при ее нажатии вершина 5 должна стать зеленой, должно пропасть время выхода и из строки вывода промежуточных данных должна исчезнуть одна строка. Результат на рисунке 4.9.

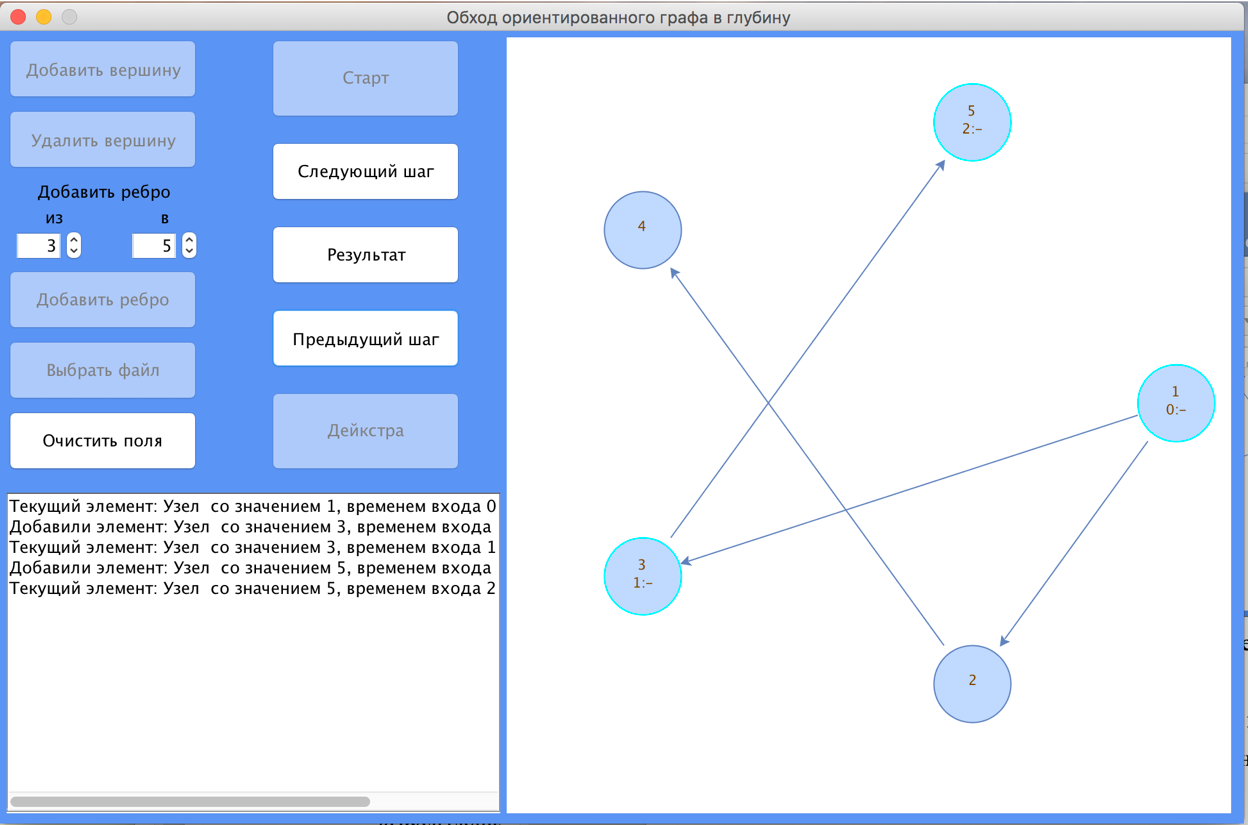


Рисунок 4.9 – интерфейс после отката к предыдущему шагу

Осталось проверить кнопку «Результат». Выполнение данного шага представлено на рисунке 4.10. Как можно заметить, все вершины красные и везде есть время входа и время выхода, поэтому можно считать завершение алгоритма успешным.

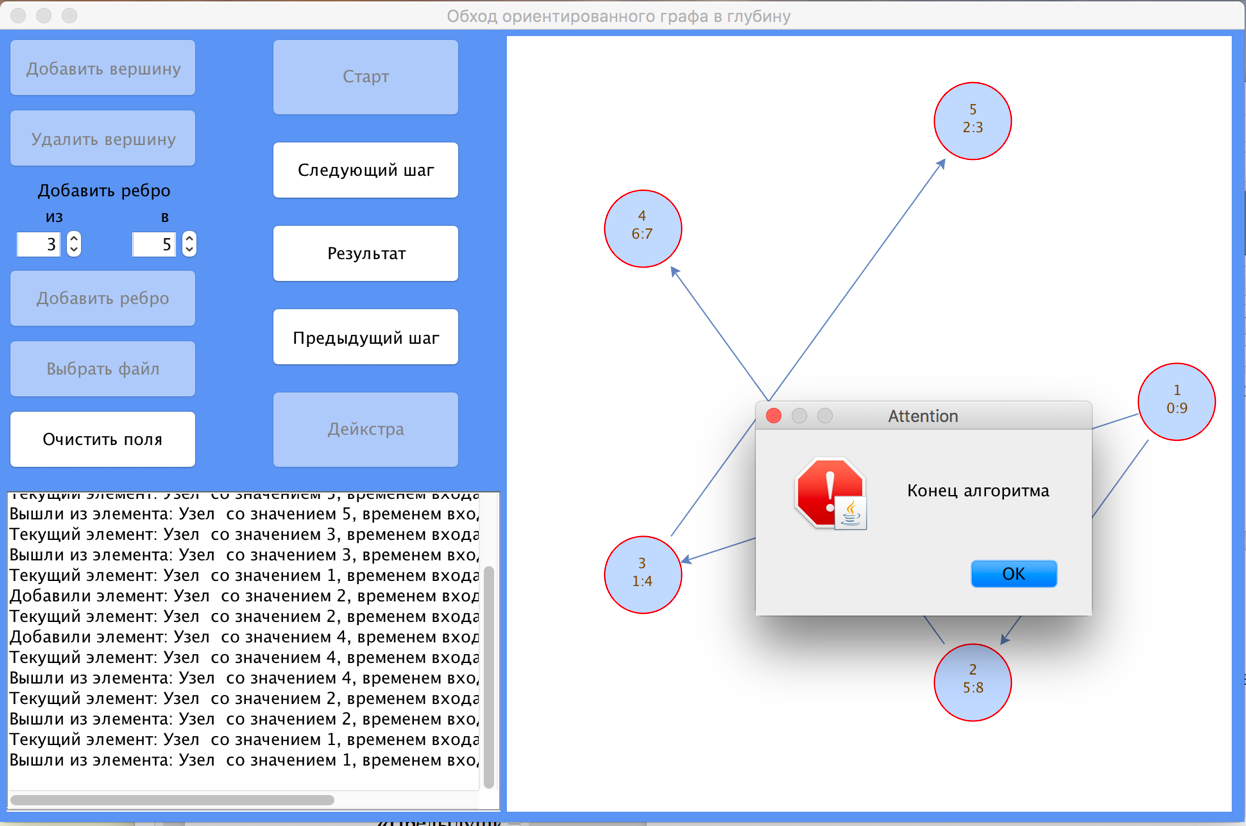


Рисунок 4.9 – интерфейс после нажатия «Результат»

При работе алгоритма Дейкстры функциональность рассмотренных трех кнопок такая же.

## **4.2 Тестирование кода алгоритма (поиск в глубину)**

Для представления работы данного алгоритма будут использоваться изображения графов после обхода в глубину (рисунки 4.10 – 4.14). Исходный граф не нужен, т.к. отличается от результата только отсутствием времен входа и выхода и подсветки вершин.

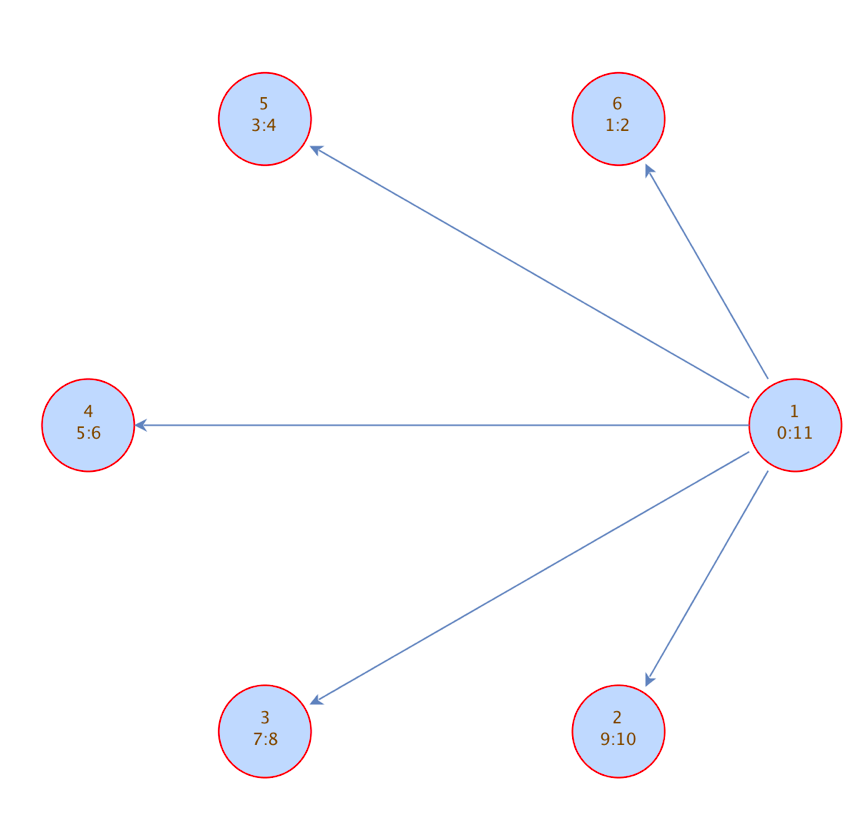


Рисунок 4.10 – тест алгоритма №1

Порядок обхода: 1-6-5-4-3-2

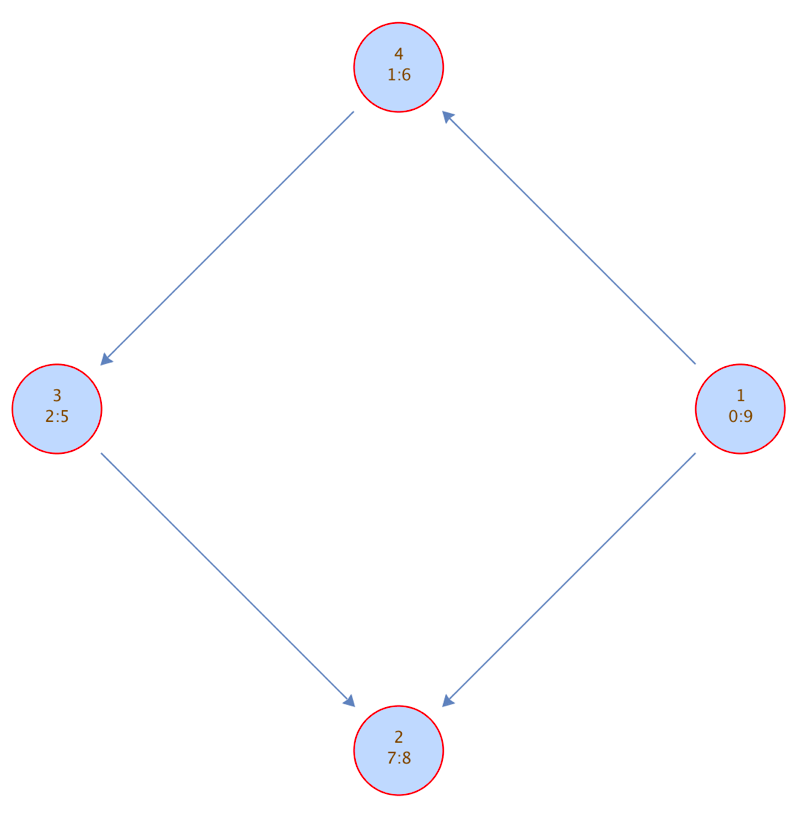


Рисунок 4.11 – тест алгоритма №2

Порядок обхода: 1-4-3-2

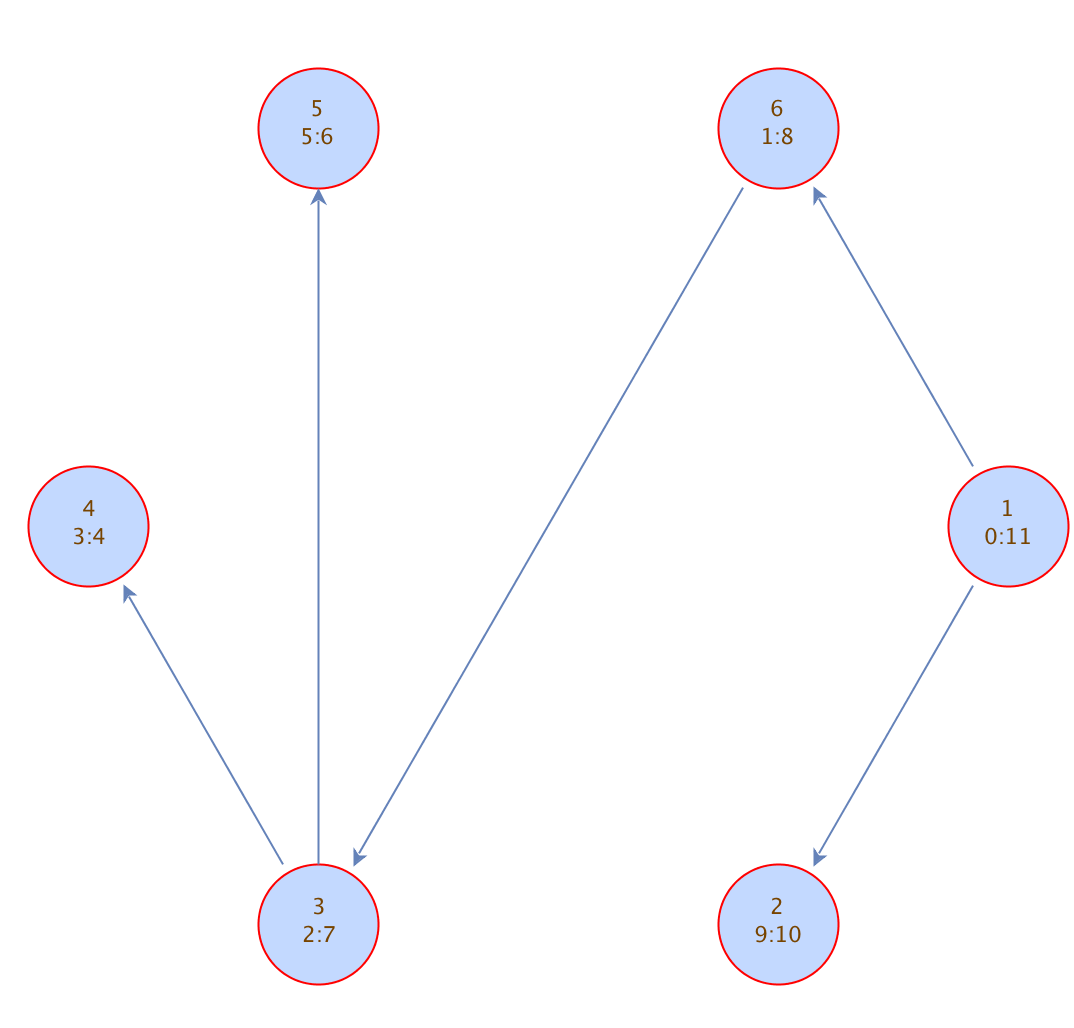


Рисунок 4.12 – тест алгоритма №3

Порядок обхода: 1-6-3-4-2-5

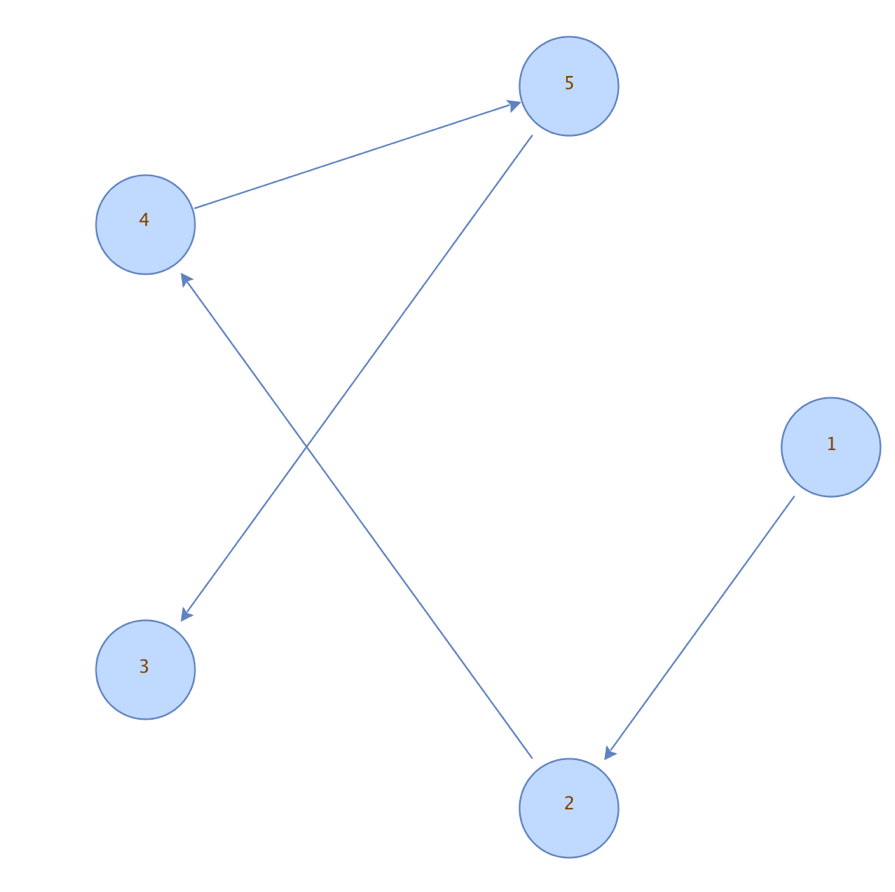


Рисунок 4.13 – тест алгоритма №4

Порядок обхода: 1-2-4-5-3

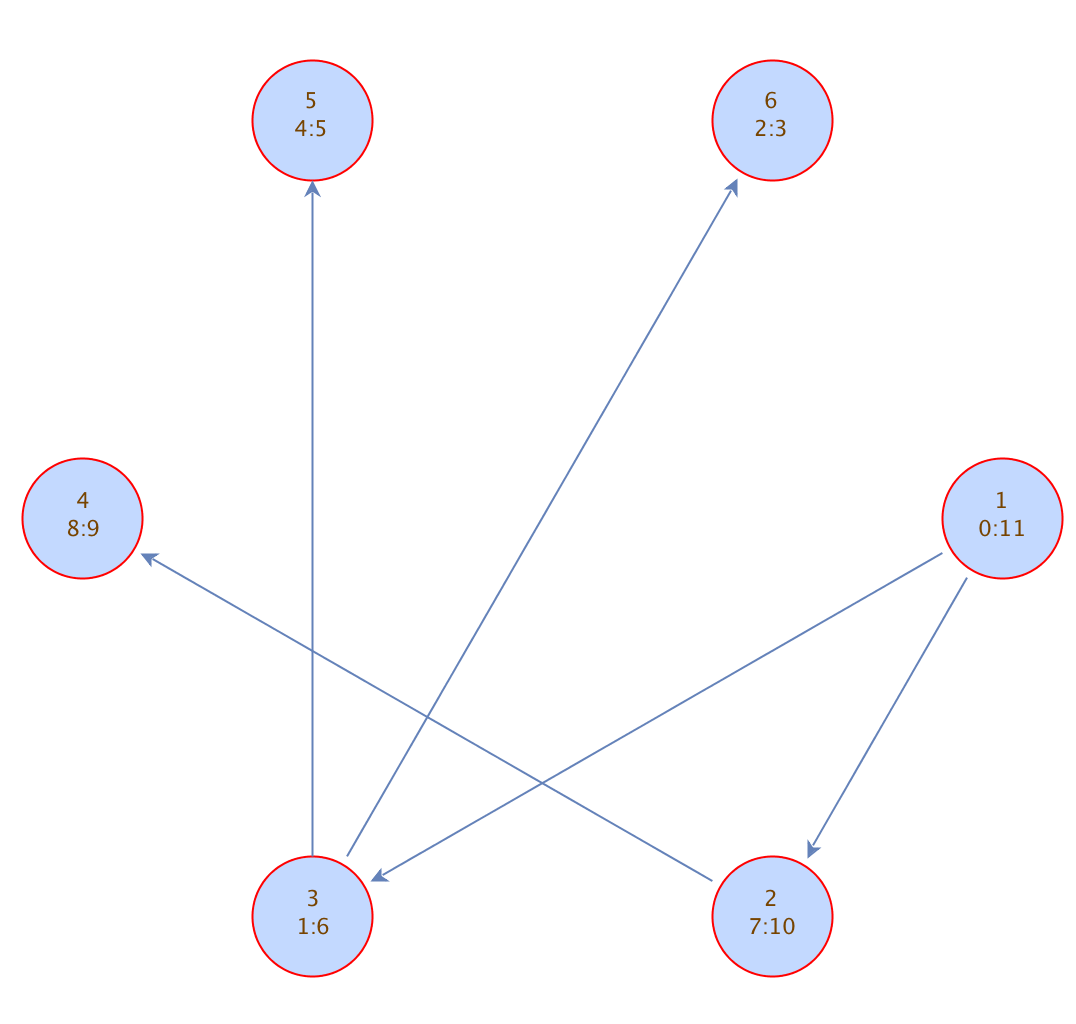


Рисунок 4.14 – тест алгоритма №5

Порядок обхода: 1-3-5-6-2-4

Приведем промежуточные данные для поиска в глубину на рисунке 4.14:  
*Текущий элемент: Узел со значением 1, временем входа 0, временем выхода*

*Добавили элемент: Узел со значением 3, временем входа 1, временем выхода*

*Текущий элемент: Узел со значением 3, временем входа 1, временем выхода*

*Добавили элемент: Узел со значением 6, временем входа 2, временем выхода*

*Текущий элемент: Узел со значением 6, временем входа 2, временем выхода*

*Вышли из элемента: Узел со значением 6, временем входа 2, временем выхода 3*

*Текущий элемент: Узел со значением 3, временем входа 1, временем выхода*

*Добавили элемент: Узел со значением 5, временем входа 4, временем выхода*

*Текущий элемент: Узел со значением 5, временем входа 4, временем выхода*

*Вышли из элемента: Узел со значением 5, временем входа 4, временем выхода 5*

*Текущий элемент: Узел со значением 3, временем входа 1, временем выхода*

*Вышли из элемента: Узел со значением 3, временем входа 1, временем выхода 6*

*Текущий элемент: Узел со значением 1, временем входа 0, временем выхода*

*Добавили элемент: Узел со значением 2, временем входа 7, временем выхода*

*Текущий элемент: Узел со значением 2, временем входа 7, временем выхода*

*Добавили элемент: Узел со значением 4, временем входа 8, временем выхода*

*Текущий элемент: Узел со значением 4, временем входа 8, временем выхода*

*Вышли из элемента: Узел со значением 4, временем входа 8, временем выхода 9*

*Текущий элемент: Узел со значением 2, временем входа 7, временем выхода*

*Вышли из элемента: Узел со значением 2, временем входа 7, временем выхода 10*

*Текущий элемент: Узел со значением 1, временем входа 0, временем выхода*

*Вышли из элемента: Узел со значением 1, временем входа 0, временем выхода 11*

Таким образом, можно убедиться, что сама работа алгоритма производится корректно.

Рассмотрим исключение. Т.к. алгоритм при поиске в глубину не работает на циклах, при введении такого графа в программу и запуске алгоритма последнее рассматриваемое ребро при обходе графа «исключается» из обхода. Таким образом программа не «ломается» и пользователь все равно получает обход в глубину данного графа, но без цикла, что для данного приложения является лучшим решением. Пример представлен на рисунке 4.15.

Еще одним исключительным случаем можно считать поведение программы, когда из начальной вершины (номер 1) нет исходящих ребер. В таком случае обход будет в себя включать только данную вершину (что логично, т.к. из нее попасть никуда нельзя). Однако т.к. по умолчанию обход начинается с первой вершины, данное поведение программы единственно правильное. Пример представлен на рисунке 4.16.

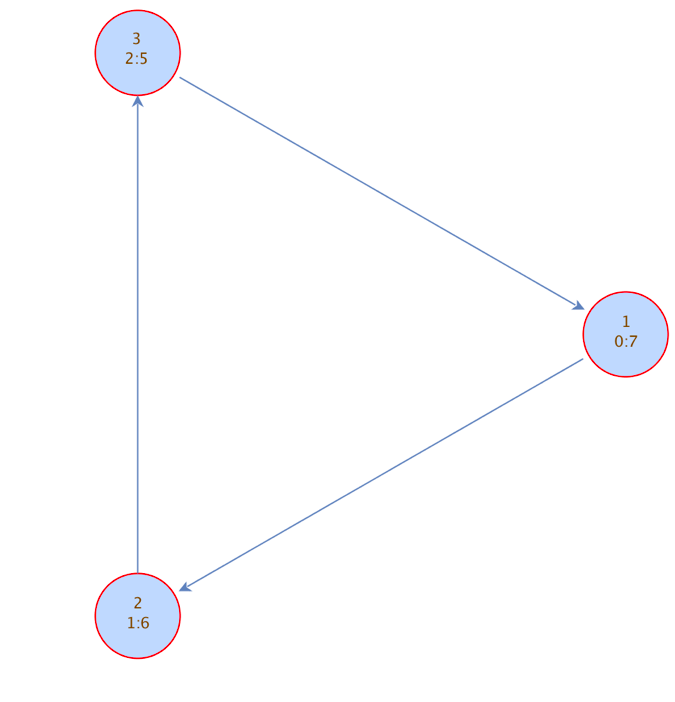


Рисунок 4.15 – работа алгоритма на графе с циклом

Порядок обхода: 1-2-3

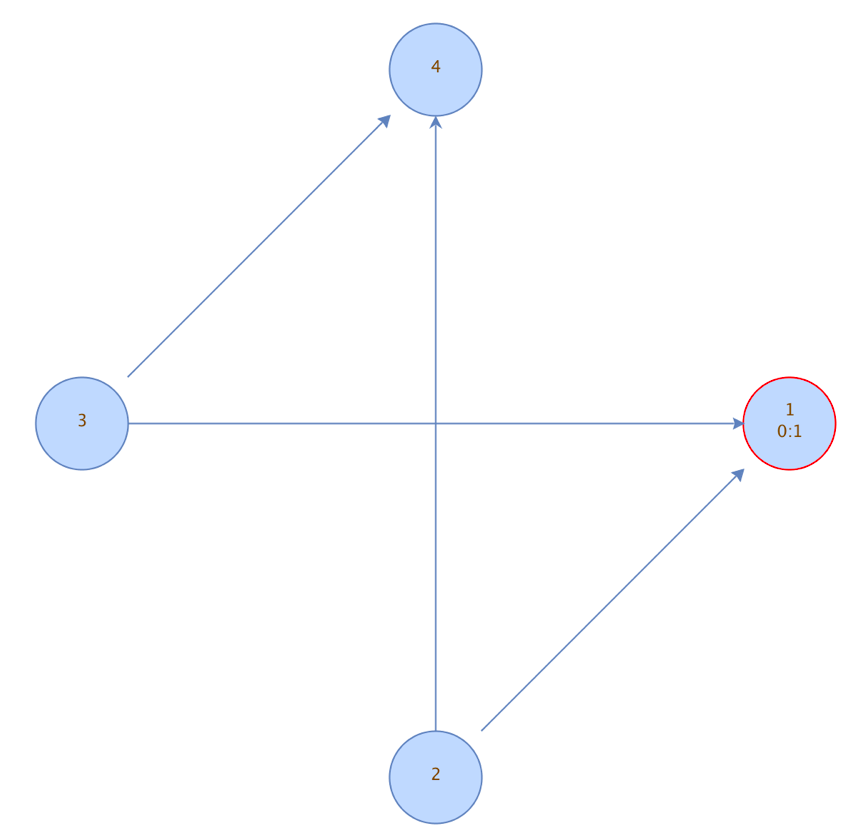


Рисунок 4.16 – работа алгоритма на графе без ребер из начальной вершины

## **4.3 Тестирование кода алгоритма (Дейкстра)**

Для тестирования данного алгоритма преимущественно будем подбирать тесты, где возможен не кратчайший путь от первой к другим вершинам. Результаты тестирования будут представлены на рисунках 4.16 - .

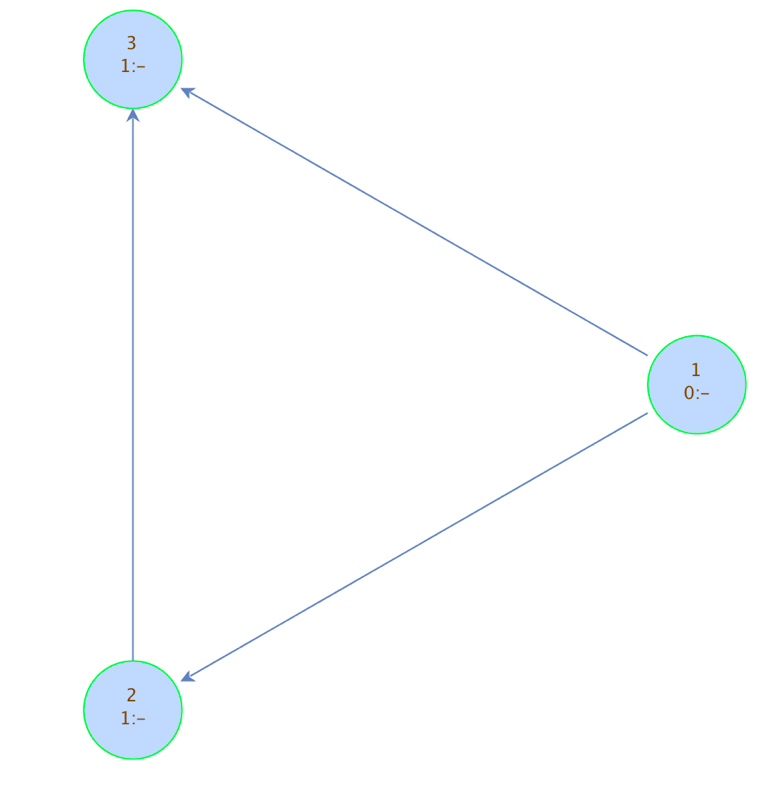


Рисунок 4.17 – тест алгоритма №1

Примечание: пути из вершины 1 в вершину 3 существует 2: прямой пусть и через вершину 2. Однако из рисунка 4.16 видно, что программа нашла кратчайший путь к вершине 3, равный единице.

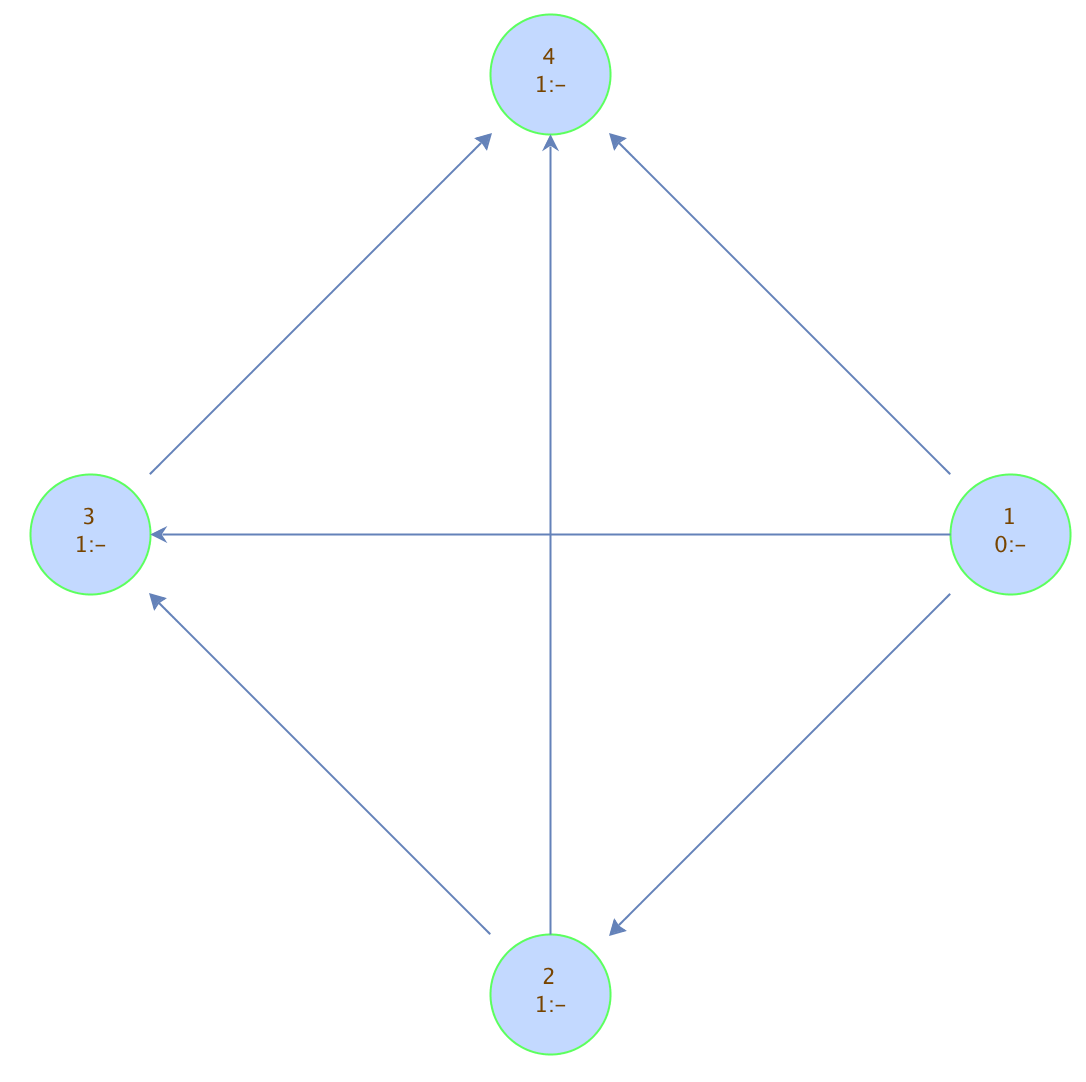


Рисунок 4.18 – тест алгоритма №2

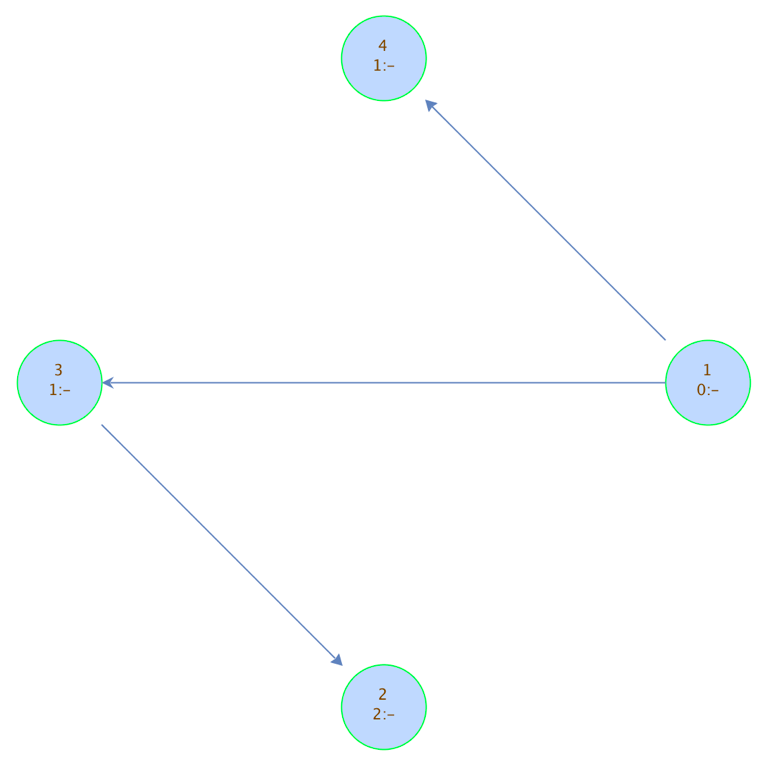


Рисунок 4.19 – тест алгоритма №3

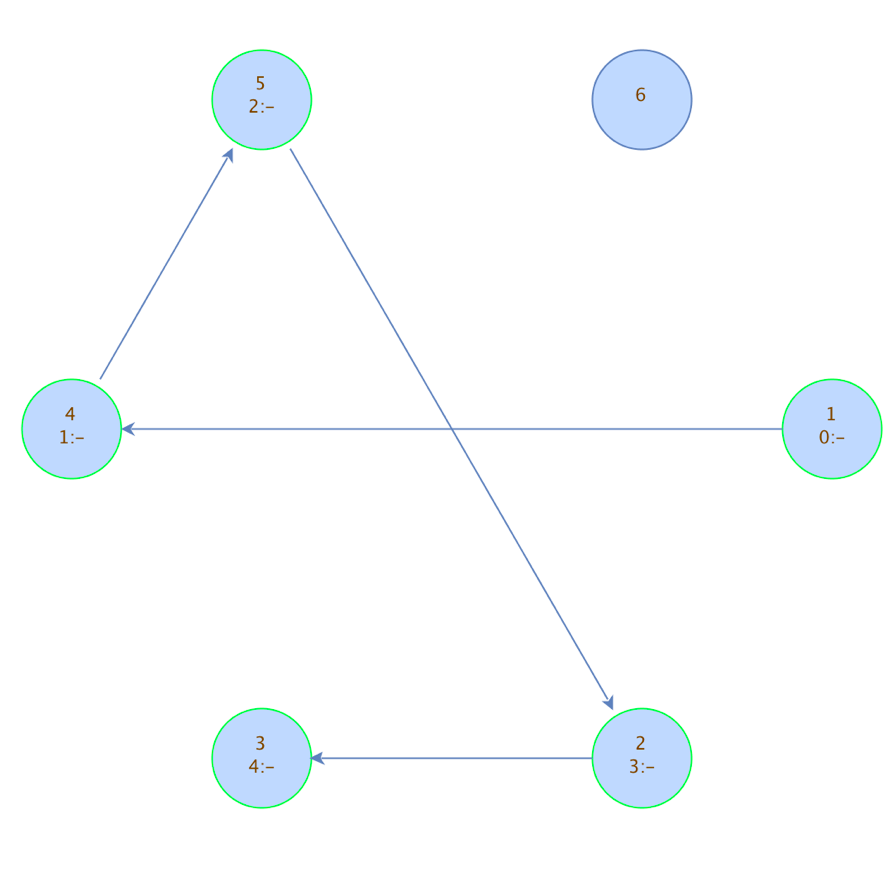


Рисунок 4.20 – тест алгоритма №4

Примечание: из теста 4 видно, что программа корректно работает и в том случае, если из вершины 1 пути нет (в данном случае – в вершину 6).

Из результатов тестирования можно сделать вывод, что и интерфейс программы, и сами алгоритмы работают корректно, интерфейс приложения прост в использовании, удобен и весьма понятен.

**5. ЗАКЛЮЧЕНИЕ**

В данной практической работе были реализованы алгоритм обхода орграфа в глубину и алгоритм Дейкстры с визуализацией. Был разработан интерфейс удобный и понятный для пользователя. Приложение с данным интерфейсом не ломается на всевозможных тестах, что подтверждают примеры из тестирования.

Также в ходе учебной практики были получены базовые знания по языку программирования Java и практически закреплены при написании визуализации алгоритмов обхода в глубину графа и алгоритма Дейкстры.

Была выполнена одна из основных задач данной учебной практики – освоение работы в команде (распределение ролей, написание каждый участником команды своей части кода, сборка данных частей кода в одно работающее приложение).

Таким образом результате учебной практики были выполнены все поставленные перед участниками команды цели (получение практических навыков в визуализации алгоритмов, изучение и получение навыков использования языка программирования Java, получение навыков работы в команде).

# **ПРИЛОЖЕНИЕ А. Исходный код**

**Step.java**

**package** com.eltech.practice.utils;  
**import** java.util.Objects;  
**public class** Step <T> {  
 **private** String **description**; *// описание шага* **private** Node<T> **value**; *// номер вершины* **public** Step(String description, Node<T> value) {  
 **this**.**description** = description;  
 **this**.**value** = value;  
 }  
  
 **public** String getDescription() {  
 **return description**;  
 } *// возврат описания* **public void** setDescription(String description) {  
 **this**.**description** = description;  
 } *// присвоение описания* **public** Node<T> getValue() {  
 **return value**;  
 } *// возврат номера вершины* **public void** setValue(Node<T> value) {  
 **this**.**value** = value;  
 } *// присвоение номера вершины* @Override  
 **public boolean** equals(Object o) { *// переопределение Equals* **if** (**this** == o) **return true**;  
 **if** (o == **null** || getClass() != o.getClass()) **return false**;  
 Step<?> step = (Step<?>) o;  
 **return** Objects.*equals*(**description**, step.**description**) &&  
 Objects.*equals*(**value**, step.**value**);  
 }  
  
 @Override *// переопределение возврата (ключ, значение)* **public int** hashCode() {  
  
 **return** Objects.*hash*(**description**, **value**);  
 }  
  
 @Override *// переопределение toString для выводы информации об описании шагов алгоритма* **public** String toString() {  
 **return "Step{"** +  
 **"description='"** + **description** + **'\''** +  
 **", value="** + **value** +  
 **'}'**;  
 }  
}

**Main.java**

**package** com.eltech.practice;  
  
**import** com.eltech.practice.graphs.SimpleGraph;  
**import** com.eltech.practice.utils.GraphsUtils;  
**import** com.eltech.practice.utils.Node;  
**import** com.eltech.practice.utils.Step;  
**import** com.eltech.practice.visualizers.GraphVisualizer;  
**import** com.google.common.graph.GraphBuilder;  
**import** com.google.common.graph.MutableGraph;  
**import** javafx.util.Pair;  
  
**import** javax.swing.\*;  
**import** java.awt.\*;  
**import** java.io.File;  
**import** java.io.IOException;  
**import** java.nio.file.Files;  
**import** java.nio.file.Paths;  
**import** java.util.ArrayList;  
**import** java.util.HashSet;  
**import** java.util.List;  
  
**public class** Main {  
 **private static** JFrame *frame*;  
 **private static** SimpleGraph *simpleGraph* = **new** SimpleGraph();  
 **private static** GraphVisualizer *graphVisualizer* = **new** GraphVisualizer();  
 **private static** List<Step<Integer>> *history* = **new** ArrayList<>();  
 **private static int** *step*;  
  
  
 **public static void** main(String[] args) {  
  
 *frame* = **new** JFrame(**"Обход ориентированного графа в глубину "**);  
  
 Color blueColor = **new** Color(0x6495ED);  
 *frame*.getContentPane().setBackground(blueColor);  
 *frame*.setSize(**new** Dimension(970, 640));  
 *frame*.setResizable(**false**);  
 *frame*.setDefaultCloseOperation(WindowConstants.***EXIT\_ON\_CLOSE***);  
 *frame*.setLayout(**null**);  
  
 JButton addVertexButton = **new** JButton(**"Добавить вершину"**);  
 addVertexButton.setSize(**new** Dimension(150, 50));  
 addVertexButton.setLocation(5, 5);  
  
 JButton delVertexButton = **new** JButton(**"Удалить вершину"**);  
 delVertexButton.setSize(**new** Dimension(150, 50));  
 delVertexButton.setLocation(5, 60);  
  
 JLabel addEdgeLabel = **new** JLabel(**"Добавить ребро"**);  
 addEdgeLabel.setSize(**new** Dimension(150, 50));  
 addEdgeLabel.setLocation(29, 100);  
  
 JLabel fromEdgeLabel = **new** JLabel(**"из"**);  
 fromEdgeLabel.setSize(**new** Dimension(75, 50));  
 fromEdgeLabel.setLocation(35, 120);  
  
 JLabel toEdgeLabel = **new** JLabel(**"в"**);  
 toEdgeLabel.setSize(**new** Dimension(75, 50));  
 toEdgeLabel.setLocation(125, 120);  
  
 SpinnerModel fromSpinnerModel = **new** SpinnerNumberModel(1, 1, 50, 1);  
 JSpinner fromSpinner = **new** JSpinner(fromSpinnerModel);  
 fromSpinner.setSize(**new** Dimension(55, 25));  
 fromSpinner.setLocation(10, 155);  
  
 SpinnerModel toSpinnerModel = **new** SpinnerNumberModel(1, 1, 50, 1);  
 JSpinner toSpinner = **new** JSpinner(toSpinnerModel);  
 toSpinner.setSize(**new** Dimension(55, 25));  
 toSpinner.setLocation(100, 155);  
  
 JButton addEdgeButton = **new** JButton(**"Добавить ребро"**);  
 addEdgeButton.setSize(**new** Dimension(150, 50));  
 addEdgeButton.setLocation(5, 185);  
  
 JButton loadFromFileButton = **new** JButton(**"Выбрать файл"**);  
 loadFromFileButton.setSize(**new** Dimension(150, 50));  
 loadFromFileButton.setLocation(5, 240);  
  
 JButton clearFieldButton = **new** JButton(**"Очистить поля"**);  
 clearFieldButton.setSize(**new** Dimension(150, 50));  
 clearFieldButton.setLocation(5, 295);  
  
 JButton startButton = **new** JButton(**"Старт"**);*//* startButton.setSize(**new** Dimension(150, 65));  
 startButton.setLocation(210, 5);  
  
 JButton resultButton = **new** JButton(**"Результат"**);  
 resultButton.setSize(**new** Dimension(150, 50));  
 resultButton.setLocation(210, 150);  
  
 JButton nextButton = **new** JButton(**"Следующий шаг"**);  
 nextButton.setSize(**new** Dimension(150, 50));  
 nextButton.setLocation(210, 85);  
  
 JButton prevButton = **new** JButton(**"Предыдущий шаг"**);  
 prevButton.setSize(**new** Dimension(150, 50));  
 prevButton.setLocation(210, 215);  
  
 JButton dijkstraButton = **new** JButton(**"Дейкстра"**);  
 dijkstraButton.setSize(**new** Dimension(150, 65));  
 dijkstraButton.setLocation(210, 280);  
  
 JTextArea infoTextArea = **new** JTextArea();  
 JScrollPane scrollPane = **new** JScrollPane(infoTextArea);  
 scrollPane.setBounds(5, 360, 385, 250);  
 infoTextArea.setEditable(**false**);  
  
  
*// JTextArea resultTextArea = new JTextArea();  
// resultTextArea.setSize(565, 68);  
// resultTextArea.setLocation(395, 540);  
// resultTextArea.setEditable(false);  
  
 frame*.add(addVertexButton);  
 *frame*.add(delVertexButton);  
 *frame*.add(addEdgeLabel);  
 *frame*.add(fromEdgeLabel);  
 *frame*.add(toEdgeLabel);  
 *frame*.add(fromSpinner);  
 *frame*.add(toSpinner);  
 *frame*.add(addEdgeButton);  
 *frame*.add(startButton);  
 *frame*.add(nextButton);  
 *frame*.add(prevButton);  
 *frame*.add(dijkstraButton);  
 *frame*.add(loadFromFileButton);  
 *// frame.add(resultTextArea);  
 frame*.add(resultButton);  
 *frame*.add(clearFieldButton);  
 *frame*.add(scrollPane);  
 *frame*.add(*graphVisualizer*);  
 *frame*.setVisible(**true**);  
 delVertexButton.setEnabled(**false**);  
 startButton.setEnabled(**false**);  
 dijkstraButton.setEnabled(**false**);  
  
  
 loadFromFileButton.addActionListener(e -> {  
 JFileChooser fileChooser = **new** JFileChooser();  
 **int** ret = fileChooser.showDialog(**null**, **"Open file"**);  
 **if** (ret == JFileChooser.***APPROVE\_OPTION***) {  
 **try** {  
 HashSet<Integer> vertexes = **new** HashSet<>();  
 File selectedFile = fileChooser.getSelectedFile();  
 **for** (String str : Files.*readAllLines*(Paths.*get*(selectedFile.getPath()))) {  
 String[] s = str.split(**" "**);  
 **int** v = Integer.*parseInt*(s[0]);  
 **int** w = Integer.*parseInt*(s[1]);  
 vertexes.add(v);  
 vertexes.add(w);  
 **for** (Pair<Integer, Integer> integerIntegerPair : *simpleGraph*.getEdges()) {  
 **if** (integerIntegerPair.getKey() == v && integerIntegerPair.getValue() == w) {  
 JOptionPane.*showMessageDialog*(**null**, **"Некорректный ввод"**, **"Attention"**, JOptionPane.***ERROR\_MESSAGE***);  
 resultButton.setEnabled(**false**);  
 nextButton.setEnabled(**false**);  
 prevButton.setEnabled(**false**);  
 startButton.setEnabled(**true**);  
 dijkstraButton.setEnabled(**true**);  
 loadFromFileButton.setEnabled(**true**);  
 addEdgeButton.setEnabled(**true**);  
 addVertexButton.setEnabled(**true**);  
 *simpleGraph*.clear();  
 *history* = **new** ArrayList<>();  
 *step* = 0;  
 *graphVisualizer*.visualizeGraph(*simpleGraph*, *history*, *step*, infoTextArea);  
 *frame*.repaint();  
 **return**;  
 }  
 }  
  
 *simpleGraph*.addEdge(v, w);  
 }  
 vertexes.forEach(integer -> *simpleGraph*.addNode());  
 startButton.setEnabled(**true**);  
 dijkstraButton.setEnabled(**true**);  
 } **catch** (NumberFormatException ex) {  
 JOptionPane.*showMessageDialog*(**null**, **"Некорректный данные!"**, **"ERROR"**, JOptionPane.***ERROR\_MESSAGE***);  
 } **catch** (NullPointerException ex) {  
 *//число пар меньше чем задано во 2 поле* JOptionPane.*showMessageDialog*(**null**, **"Несоответствие заданного и фактического количества ребер!"**, **"ERROR"**, JOptionPane.***ERROR\_MESSAGE***);  
 } **catch** (IndexOutOfBoundsException ex) {  
 *//связываются не существующие вершины* JOptionPane.*showMessageDialog*(**null**, **"Попытка связать несуществующие вершины!"**, **"ERROR"**, JOptionPane.***ERROR\_MESSAGE***);  
 } **catch** (IOException e1) {  
 e1.printStackTrace();  
 }  
 }  
 *graphVisualizer*.visualizeGraph(*simpleGraph*, *history*,*step*, infoTextArea);  
 });  
  
 startButton.setEnabled(**false**);  
 startButton.addActionListener(e -> {  
 resultButton.setEnabled(**true**);  
 nextButton.setEnabled(**true**);  
 prevButton.setEnabled(**true**);  
 startButton.setEnabled(**false**);  
 dijkstraButton.setEnabled(**false**);  
 loadFromFileButton.setEnabled(**false**);  
 addEdgeButton.setEnabled(**false**);  
 addVertexButton.setEnabled(**false**);  
 delVertexButton.setEnabled(**false**);  
  
 MutableGraph<Node<Integer>> graph = GraphBuilder.<Integer>*directed*().allowsSelfLoops(**true**).build();  
 **for** (**int** i = 0; i < *simpleGraph*.getNumberOfNode(); i++) {  
 graph.addNode(**new** Node<>(i));  
 }  
  
 **for** (Pair<Integer, Integer> edge : *simpleGraph*.getEdges()) {  
 graph.putEdge(**new** Node<>(edge.getKey()), **new** Node<>(edge.getValue()));  
 }  
  
 *history* = GraphsUtils.*deepFirstSearch*(graph, **new** Node<>(1));  
 *step* = 0;  
  
 *graphVisualizer*.visualizeGraph(*simpleGraph*, *history*, *step*, infoTextArea);  
 });  
 clearFieldButton.setEnabled(**true**);  
 clearFieldButton.addActionListener(e -> {  
 resultButton.setEnabled(**false**);  
 nextButton.setEnabled(**false**);  
 prevButton.setEnabled(**false**);  
 startButton.setEnabled(**true**);  
 dijkstraButton.setEnabled(**true**);  
 loadFromFileButton.setEnabled(**true**);  
 addEdgeButton.setEnabled(**true**);  
 addVertexButton.setEnabled(**true**);  
 *simpleGraph*.clear();  
 *history* = **new** ArrayList<>();  
 *step* = 0;  
 *graphVisualizer*.visualizeGraph(*simpleGraph*, *history*, *step*, infoTextArea);  
 *frame*.repaint();  
 startButton.setEnabled(**false**);  
 dijkstraButton.setEnabled(**false**);  
 });  
 addVertexButton.setEnabled(**true**);  
 addVertexButton.addActionListener(e -> {  
 *simpleGraph*.addNode();  
 startButton.setEnabled(**true**);  
 dijkstraButton.setEnabled(**true**);  
 *graphVisualizer*.visualizeGraph(*simpleGraph*, *history*, *step*, infoTextArea);  
 delVertexButton.setEnabled(**true**);  
 });  
  
  
 delVertexButton.addActionListener(e -> {  
 *simpleGraph*.delNode();  
 startButton.setEnabled(**true**);  
 dijkstraButton.setEnabled(**true**);  
 *graphVisualizer*.visualizeGraph(*simpleGraph*, *history*, *step*, infoTextArea);  
 });  
  
 addEdgeButton.setEnabled(**true**);  
 addEdgeButton.addActionListener(e -> {  
 **for** (Pair<Integer, Integer> integerIntegerPair : *simpleGraph*.getEdges()) {  
 **if** (integerIntegerPair.getKey() == fromSpinner.getValue() && integerIntegerPair.getValue() == toSpinner.getValue()) {  
 JOptionPane.*showMessageDialog*(**null**, **"Некорректный ввод(такое ребро уже существует)"**, **"Attention"**, JOptionPane.***ERROR\_MESSAGE***);  
 resultButton.setEnabled(**false**);  
 nextButton.setEnabled(**false**);  
 prevButton.setEnabled(**false**);  
 startButton.setEnabled(**true**);  
 dijkstraButton.setEnabled(**true**);  
  
  
 loadFromFileButton.setEnabled(**true**);  
 addEdgeButton.setEnabled(**true**);  
 addVertexButton.setEnabled(**true**);  
 *//simpleGraph.clear();  
 history* = **new** ArrayList<>();  
 *step* = 0;  
 *graphVisualizer*.visualizeGraph(*simpleGraph*, *history*, *step*, infoTextArea);  
 *frame*.repaint();  
 **return**;  
 }  
 }  
 **if** (((Integer)fromSpinner.getValue()).compareTo(0)<0  
 || ((Integer)toSpinner.getValue()).compareTo(0)<0  
 || ((Integer)fromSpinner.getValue()).compareTo(*simpleGraph*.getNumberOfNode())>0  
 || ((Integer)toSpinner.getValue()).compareTo(*simpleGraph*.getNumberOfNode())>0){  
 JOptionPane.*showMessageDialog*(**null**, **"Некорректный ввод(не существует такой вершины)"**, **"Attention"**, JOptionPane.***ERROR\_MESSAGE***);  
 } **else** {  
 *simpleGraph*.addEdge(fromSpinner, toSpinner);  
 }  
 *graphVisualizer*.visualizeGraph(*simpleGraph*, *history*, *step*, infoTextArea);  
 });  
 nextButton.setEnabled(**false**);  
 nextButton.addActionListener(e -> {  
 **if** (*step*<*history*.size()){  
 *step*++;  
 }**else** {  
 JOptionPane.*showMessageDialog*(**null**, **"Конец алгоритма"**, **"Attention"**, JOptionPane.***ERROR\_MESSAGE***);  
 }  
 *graphVisualizer*.visualizeGraph(*simpleGraph*, *history*, *step*, infoTextArea);  
 });  
 resultButton.setEnabled(**false**);  
 resultButton.addActionListener(e -> {  
 *step* = *history*.size();  
 *graphVisualizer*.visualizeGraph(*simpleGraph*, *history*, *step*, infoTextArea);  
 JOptionPane.*showMessageDialog*(**null**, **"Конец алгоритма"**, **"Attention"**, JOptionPane.***ERROR\_MESSAGE***);  
 });  
 prevButton.setEnabled(**false**);  
 prevButton.addActionListener(e -> {  
 **if** (*step*>0){  
 *step*--;  
  
 }  
 *graphVisualizer*.visualizeGraph(*simpleGraph*, *history*, *step*, infoTextArea);  
 });  
 dijkstraButton.setEnabled(**false**);  
 dijkstraButton.addActionListener(e -> {  
 resultButton.setEnabled(**true**);  
 nextButton.setEnabled(**true**);  
 prevButton.setEnabled(**true**);  
 startButton.setEnabled(**false**);  
 dijkstraButton.setEnabled(**false**);  
 loadFromFileButton.setEnabled(**false**);  
 addEdgeButton.setEnabled(**false**);  
 addVertexButton.setEnabled(**false**);  
 delVertexButton.setEnabled(**false**);  
  
 MutableGraph<Node<Integer>> graph = GraphBuilder.<Integer>*directed*().allowsSelfLoops(**true**).build();  
 **for** (**int** i = 0; i < *simpleGraph*.getNumberOfNode(); i++) {  
 graph.addNode(**new** Node<>(i));  
 }  
  
 **for** (Pair<Integer, Integer> edge : *simpleGraph*.getEdges()) {  
 graph.putEdge(**new** Node<>(edge.getKey()), **new** Node<>(edge.getValue()));  
 }  
  
 *history* = GraphsUtils.*dijkstra*(graph, **new** Node<>(1));  
 *step* = 0;  
  
 *graphVisualizer*.visualizeGraph(*simpleGraph*, *history*, *step*, infoTextArea);  
 });  
  
  
  
 }  
}

**Node.java**

**package** com.eltech.practice.utils;  
  
**import** java.util.Objects;  
  
**public class** Node <T> {  
 **private** T **value**;  
 **private int inTime** = Integer.***MAX\_VALUE***;  
 **private int outTime** = Integer.***MAX\_VALUE***;  
  
 **public** Node(T value, **int** inTime, **int** outTime) {  
 **this**.**value** = value;  
 **this**.**inTime** = inTime;  
 **this**.**outTime** = outTime;  
 }  
  
 **public** Node(T value) {  
 **this**.**value** = value;  
 } *// присвоение значения* **public** T getValue() {  
 **return value**;  
 } *// возврат значения* **public void** setValue(T value) {  
 **this**.**value** = value;  
 } *// присвоение значений* **public int** getInTime() {  
 **return inTime**;  
 } *// возвращение времени входа* **public void** setInTime(**int** inTime) {  
 **this**.**inTime** = inTime;  
 } *// присвоение времени входа* **public int** getOutTime() {  
 **return outTime**;  
 } *// возвращение времени выхода* **public void** setOutTime(**int** outTime) {  
 **this**.**outTime** = outTime;  
 } *// присвоение времени выхода* @Override *// переопределение метода equals* **public boolean** equals(Object o) {  
 **if** (**this** == o) **return true**;  
 **if** (o == **null** || getClass() != o.getClass()) **return false**;  
 Node<?> node = (Node<?>) o;  
 **return** Objects.*equals*(**value**, node.**value**);  
 }  
  
 @Override *// переопределение метода hashCode* **public int** hashCode() {  
 **return** Objects.*hash*(**value**, **outTime**);  
 }  
  
 @Override *// переопределение метода toString* **public** String toString() {  
 **return "Узел "** +  
 **" со значением "** + **value** +  
 **", временем входа "** + **inTime** +  
 **", временем выхода "** + (**outTime** == Integer.***MAX\_VALUE*** ? **""** : **outTime**);  
 }  
}

**SimpleGraph.java**

**package** com.eltech.practice.graphs;  
  
**import** javafx.util.Pair;  
  
**import** javax.swing.\*;  
**import** java.util.ArrayList;  
**import** java.util.List;  
  
**public class** SimpleGraph {  
 **private int numberOfNode**; *// количество создавваемых вершин* **private** List<Pair<Integer, Integer>> **edges** = **new** ArrayList<>(); *// список инцидентных вершин* **public void** addNode() { *// добавление вершины* **this**.**numberOfNode**++;  
 }  
  
 **public void** delNode(){  
 **if** (**this**.**numberOfNode**!=1) {  
 **this**.**numberOfNode**--;  
 }  
 }  
  
  
 **public void** addEdge(JSpinner from, JSpinner to) { *// добавление ребер* **this**.**edges**.add(**new** Pair<>((Integer) from.getValue(), (Integer) to.getValue()));  
 }  
  
 **public void** addEdge(**int** from, **int** to) {  
 **this**.**edges**.add(**new** Pair<>(from, to));  
 }  
  
 **public** ArrayList<Pair<Double, Double>> getNodes() { *// размещения вершин на поле для основного графа* ArrayList<Pair<Double, Double>> points = **new** ArrayList<>();  
 **double** phi0 = 0;  
 **double** phi = 2 \* Math.***PI*** / **numberOfNode**;  
 **int** r = 230;  
 **for** (**int** i = 0; i <**numberOfNode** ; i++) {  
 points.add(**new** Pair<>(250 + r \* Math.*cos*(phi0), 250 + r \* Math.*sin*(phi0)));  
 phi0 += phi;  
 }  
 **return** points;  
 }  
  
 **public int** getNumberOfNode() {  
 **return numberOfNode**;  
 } *// возвращения количества вершин* **public** List<Pair<Integer, Integer>> getEdges() {  
 **return edges**;  
 } *// возвращение количества ребер* **public void** clear() { *// очистка основного поля* **numberOfNode** = 0;  
 **edges** = **new** ArrayList<>();  
 }  
}

**GraphsUtils.java**

**package** com.eltech.practice.utils;  
  
**import** com.google.common.graph.EndpointPair;  
**import** com.google.common.graph.Graph;  
  
**import** java.util.\*;  
  
  
**public class** GraphsUtils {  
 **private** GraphsUtils() {  
 }  
  
 **public static** <T> List<Step<T>> deepFirstSearch(Graph<Node<T>> graph, Node<T> startNode) {  
 Integer time = 0;  
 List<Step<T>> history = **new** ArrayList<>(); *// список для хранения истории* Deque<Node<T>> deque = **new** LinkedList<>(); *// список обработанных вершин* deque.addLast(startNode);  
 startNode.setInTime(time++);  
  
 **while** (!deque.isEmpty()) {  
 Node<T> current = deque.getLast();  
 *saveToHistory*(history, current, **"Текущий элемент: %s"**);  
  
 Set<Node<T>> nodes = graph.successors(current); *// обработанный элемент* **if** (nodes.isEmpty()) {  
 current.setOutTime(time++);  
 deque.removeLast();  
 *saveToHistory*(history, current, **"Вышли из элемента: %s"**);  
 **continue**;  
 }  
  
 **boolean** foundNotVisited = **false**;  
 **for** (Node<T> node : nodes) {  
  
 **if** (node.getInTime() == Integer.***MAX\_VALUE***) {  
 deque.addLast(node);  
 node.setInTime(time++);  
 *saveToHistory*(history, node, **"Добавили элемент: %s"**);  
 foundNotVisited = **true**;  
 **break**;  
 }  
 }  
  
 **if** (!foundNotVisited) {  
 current.setOutTime(time++);  
 deque.removeLast();  
 *saveToHistory*(history, current, **"Вышли из элемента: %s"**);  
 }  
 }  
  
 **return** history;  
 }  
  
 **public static** <T> List<Step<T>> dijkstra(Graph<Node<T>> graph, Node<T> startNode) {  
 HashMap<T, Integer> map = **new** HashMap<>(); *// Map with weight corresponding to Nodes.* List<Step<T>> history = **new** ArrayList<>();  
 Deque<Node<T>> deque = **new** LinkedList<>();  
  
 deque.addLast(startNode);  
 startNode.setInTime(0);  
 map.put(startNode.getValue(), 0);  
  
 **while** (!deque.isEmpty()) {  
 Node<T> current = deque.getFirst();  
 *saveToHistory*(history, current, **"Текущий элемент: %s"**);  
  
 Set<Node<T>> nodes = **new** HashSet<>();  
 Set<EndpointPair<Node<T>>> edges = graph.incidentEdges(current);  
 **for** (EndpointPair<Node<T>> edge : edges) {  
 **if** (edge.target() != current) {  
 nodes.add(edge.target());  
 }  
 }  
  
 **for** (Node<T> node : nodes) {  
  
 *// Compare weights* **if** (map.getOrDefault(node.getValue(), Integer.***MAX\_VALUE***) > map.get(current.getValue()) + 1) {  
 deque.addLast(node);  
 *// Put them to the map* map.put(node.getValue(), current.getInTime() + 1);  
 node.setInTime(map.get(node.getValue()));  
  
 *saveToHistory*(history, node, **"Добавили элемент в очередь: %s"**);  
 }  
 }  
  
 deque.removeFirst();  
  
 *saveToHistory*(history, current, **"Вышли из элемента: %s"**);  
 }  
  
 **return** history;  
 }  
  
 **private static** <T> **void** saveToHistory(List<Step<T>> history, Node<T> current, String s) {  
 Node<T> toHistory = *copy*(current);  
 history.add(**new** Step<>(String.*format*(s + **"\n"**, toHistory), toHistory));  
 }  
  
 **private static** <T> Node<T> copy(Node<T> current) {  
 Node<T> toHistory = **new** Node<>(current.getValue());  
 toHistory.setInTime(current.getInTime());  
 toHistory.setOutTime(current.getOutTime());  
 **return** toHistory;  
 }  
}

**GraphVisualiser.java**

**package** com.eltech.practice.visualizers;  
  
**import** com.eltech.practice.graphs.SimpleGraph;  
**import** com.eltech.practice.utils.Step;  
**import** com.mxgraph.model.mxCell;  
**import** com.mxgraph.swing.mxGraphComponent;  
**import** com.mxgraph.view.mxGraph;  
**import** javafx.util.Pair;  
  
**import** javax.swing.\*;  
**import** javax.swing.border.EmptyBorder;  
**import** java.awt.\*;  
**import** java.util.ArrayList;  
**import** java.util.List;  
  
**public class** GraphVisualizer **extends** JPanel {  
  
 **public** GraphVisualizer() {  
 **this**.setSize(565, 605);  
 **this**.setLocation(**new** Point(395, 5));  
 **this**.setBackground(Color.***white***);  
 }  
  
 **public void** visualizeGraph(SimpleGraph simpleGraph, List<Step<Integer>> history, **int** step, JTextArea info) {  
  
 removeAll();  
  
 mxGraph graph = **new** mxGraph();  
 Object parent = graph.getDefaultParent();  
  
 graph.getModel().beginUpdate();  
  
 ArrayList<Pair<Double, Double>> nodes = simpleGraph.getNodes();  
 ArrayList<Object> vertexes = **new** ArrayList<>();  
  
 **int** i = 0;  
 **for** (Pair<Double, Double> node : nodes) {  
 ++i;  
 vertexes.add(graph.insertVertex(parent, **null**, i, node.getKey(), node.getValue(), 60, 60, **"shape=ellipse"**));  
  
 }  
  
 info.selectAll();  
 info.replaceSelection(**""**);  
  
 **for** (**int** j = 0; j < step; j++) {  
 Step<Integer> stepOne = history.get(j);  
 mxCell cell = (mxCell) vertexes.get(stepOne.getValue().getValue() - 1);  
 cell.setValue(history.get(j).getValue().getValue().toString() +  
 **"\n"** + history.get(j).getValue().getInTime() +  
 **":"** + (history.get(j).getValue().getOutTime() == Integer.***MAX\_VALUE*** ? **"-"** : history.get(j).getValue().getOutTime()));  
 **if** (stepOne.getValue().getInTime() != Integer.***MAX\_VALUE*** && stepOne.getValue().getOutTime() == Integer.***MAX\_VALUE***) {  
 graph.setCellStyle(**"shape=ellipse;strokeColor=#5cff5e"**, **new** Object[]{cell});  
 } **else if** (stepOne.getValue().getInTime() != Integer.***MAX\_VALUE*** && stepOne.getValue().getOutTime() != Integer.***MAX\_VALUE***) {  
 graph.setCellStyle(**"shape=ellipse;strokeColor=#ff0000"**, **new** Object[]{cell});  
 }  
 **if** (stepOne.getDescription().startsWith(**"Текущий"**)) {  
 graph.setCellStyle(**"shape=ellipse;strokeColor=#00ffff"**, **new** Object[]{cell});  
 }  
 info.append(stepOne.getDescription());  
 }  
 **for** (Pair<Integer, Integer> edge : simpleGraph.getEdges()) {  
 graph.insertEdge(parent, **null**, **null**, vertexes.get(edge.getKey() - 1), vertexes.get(edge.getValue() - 1));  
 }  
  
  
 graph.getModel().endUpdate();  
  
 mxGraphComponent graphComponent = **new** mxGraphComponent(graph);  
 graphComponent.setVisible(**true**);  
 graphComponent.setConnectable(**false**);  
 graphComponent.setSize(565, 605);  
 graphComponent.setLocation(**new** Point(395, 15));  
 graphComponent.setBorder(**new** EmptyBorder(0,0,0,0));  
  
 graph.setAllowDanglingEdges(**false**);  
 graph.setCellsResizable(**false**);  
 graph.setCellsDeletable(**false**);  
 graph.setCellsEditable(**false**);  
 graph.setCellsDisconnectable(**false**);  
 graph.setEdgeLabelsMovable(**false**);  
 graph.setConnectableEdges(**false**);  
  
 **this**.add(graphComponent);  
 **this**.revalidate();  
  
 }  
}