**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по учебной практике**

Тема: Визуализация алгоритма Дейкстры поиска

Кратчайших путей в графе

|  |  |  |
| --- | --- | --- |
| Студент гр. 7381 |  | Дорох С.В. |
| Студент гр. 7381 |  | Ильясов А.В. |
| Студент гр. 7381 |  | Павлов А.П. |
| Руководитель |  | Жангиров Т.М. |

Санкт-Петербург

2019

**ЗАДАНИЕ**

**на учебную практику**

|  |  |  |
| --- | --- | --- |
| Студент Дорох С.В. группы 7381 | | |
| Студент Ильясов А.В. группы 7381 | | |
| Студент Павлов А.П. группы 7381  Тема практики: разработка визуализатора алгоритма на Java. | | |
| Задание на практику:  Командная итеративная разработка визуализатора алгоритма(ов) на Java с графическим интерфейсом.  Алгоритм: алгоритм Дейкстры. | | |
| Сроки прохождения практики: 01.07.2019 – 14.07.2019 | | |
| Дата сдачи отчета: 12.07.2019 | | |
| Дата защиты отчета: 12.07.2019 | | |
|  | | |
| Студент |  | Дорох С.В. |
| Студент |  | Ильясов А.В. |
| Студент |  | Павлов А.П. |
| Руководитель |  | Жангиров Т.М. |

**Аннотация**

Темой данной учебной практики является командная итеративная разработка визуализатора алгоритма на языке программирования Java. Цель учебной практики – получить практические навыки в визуализации алгоритмов, изучить и получить навыки использования языка программирования Java, получить навыки работы в команде. В работе представлена визуализация алгоритма Дейкстры поиска кратчайших путей в графе.

**Summary**

The topic of this practice is the team iterative development of the algorithm visualizer in Java programming language. The purpose of the training practice is to get practical skills in visualization of algorithms, to learn and get skills of using Java programming language, to get skills of working in a team. The work presents the visualization of the algorithm of Dijkstra search for the shortest paths in the graph.

**содержание**

|  |  |  |
| --- | --- | --- |
|  | Введение | 5 |
| 1.1 | Требования к программе | 6 |
| 1.1. | Исходные требования к программе | 7 |
| 1.1.1. | Требования к архитектуре проекта | 7 |
| 1.1.2. | Требования к вводу исходных данных | 10 |
| 1.1.3. | Требования к визуализации | 10 |
| 1.1.4. | Требования к выводу результата | 12 |
| 1.2. | Уточнение требований после сдачи прототипа | 12 |
| 1.3. | Уточнение требований после сдачи первой версии | 12 |
| 2. | План разработки и распределение ролей в бригаде | 13 |
| 2.1. | План разработки | 13 |
| 2.2. | Распределение ролей в бригаде | 13 |
| 3. | Особенности реализации | 15 |
| 3.1. | Использованные сторонние библиотеки | 15 |
| 3.2. | Использованные структуры данных | 15 |
|  | Заключение | 18 |
|  | Список использованных источников | 19 |

**введение**

**Формулировка задания:** требуется разработать программу, которая при помощи алгоритма Дейкстры будет находить кратчайшие пути в ориентированном графе до всех вершин от заданной. Также должен присутствовать графический интерфейс для визуализации работы алгоритма.

**Описание алгоритма Дейкстры:** алгоритм на графах, изобретённый нидерландским учёным Эдсгером Дейкстрой в 1959 году. Находит кратчайшие пути от одной из вершин графа до всех остальных. Алгоритм работает только для графов без рёбер отрицательного веса.

Алгоритм Дейкстры:

Шаг 1. Всем вершинам, за исключением стартовой, присваивается вес равный бесконечности, а стартовой – 0.

Шаг 2. Все вершины не выделены.

Шаг 3. Первая стартовая объявляется текущей.

Шаг 4. Вес всех невыделенных вершин пересчитывается по формуле: вес невыделенной вершины есть минимальное число из старого веса данной вершины, суммы веса текущей вершины и веса ребра, соединяющего текущую вершину с невыделенной.

Шаг 5. Среди невыделенных вершин ищется вершина с минимальным весом. Если таковая не найдена, то есть вес всех вершин равен бесконечности, то маршрут не существует. Следовательно, выход. Иначе, текущей становится найденная вершина. Она же выделяется.

Шаг 6. Если текущей вершиной оказывается конечная, то путь найден, и его вес есть вес конечной вершины.

Шаг 7. Переход на шаг 4.

**1. требования к программе**

**1.1. Исходные Требования к программе**

**1.1.1. Требования к архитектуре проекта**

На рис. 1 – 3 представлены uml-диаграммы, по которым и собирался проект.

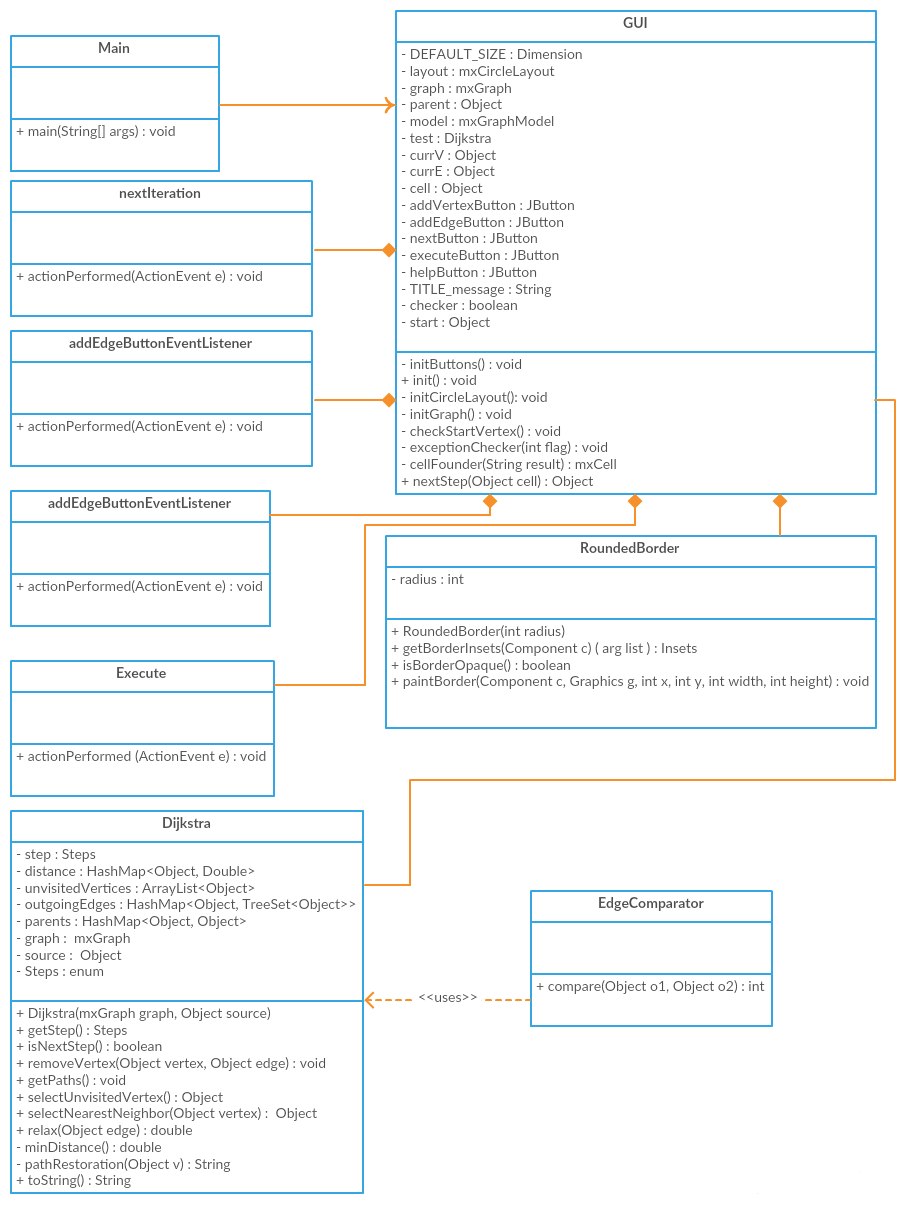
****

Рисунок 1 – uml-диаграмма классов

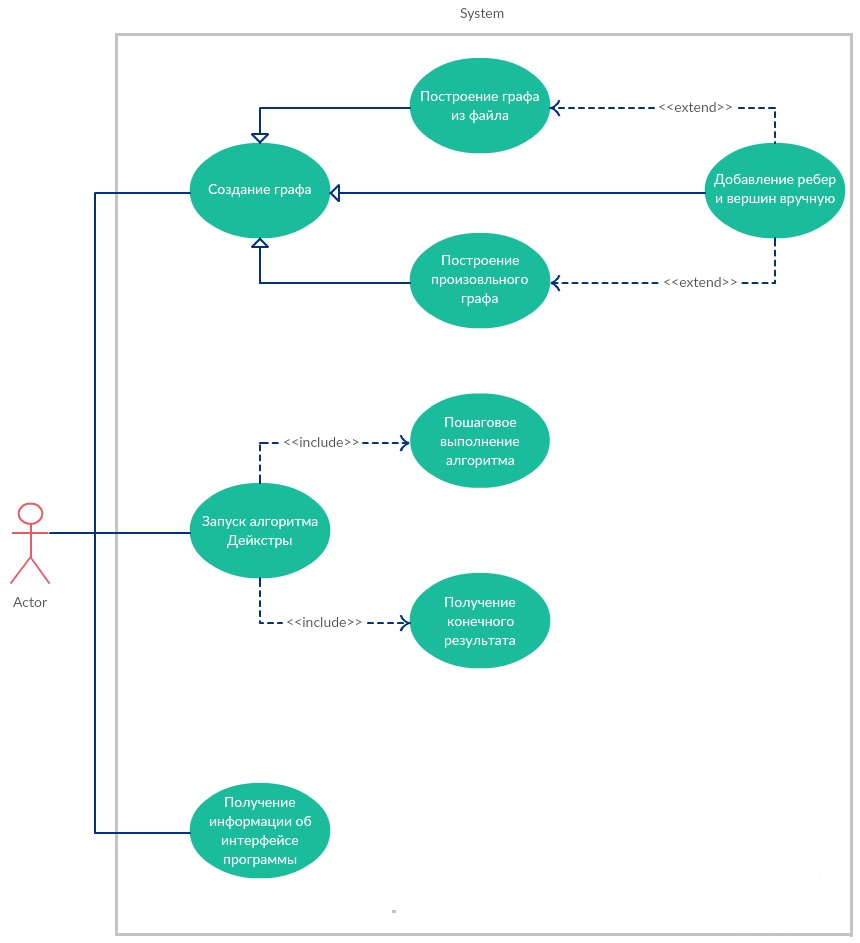
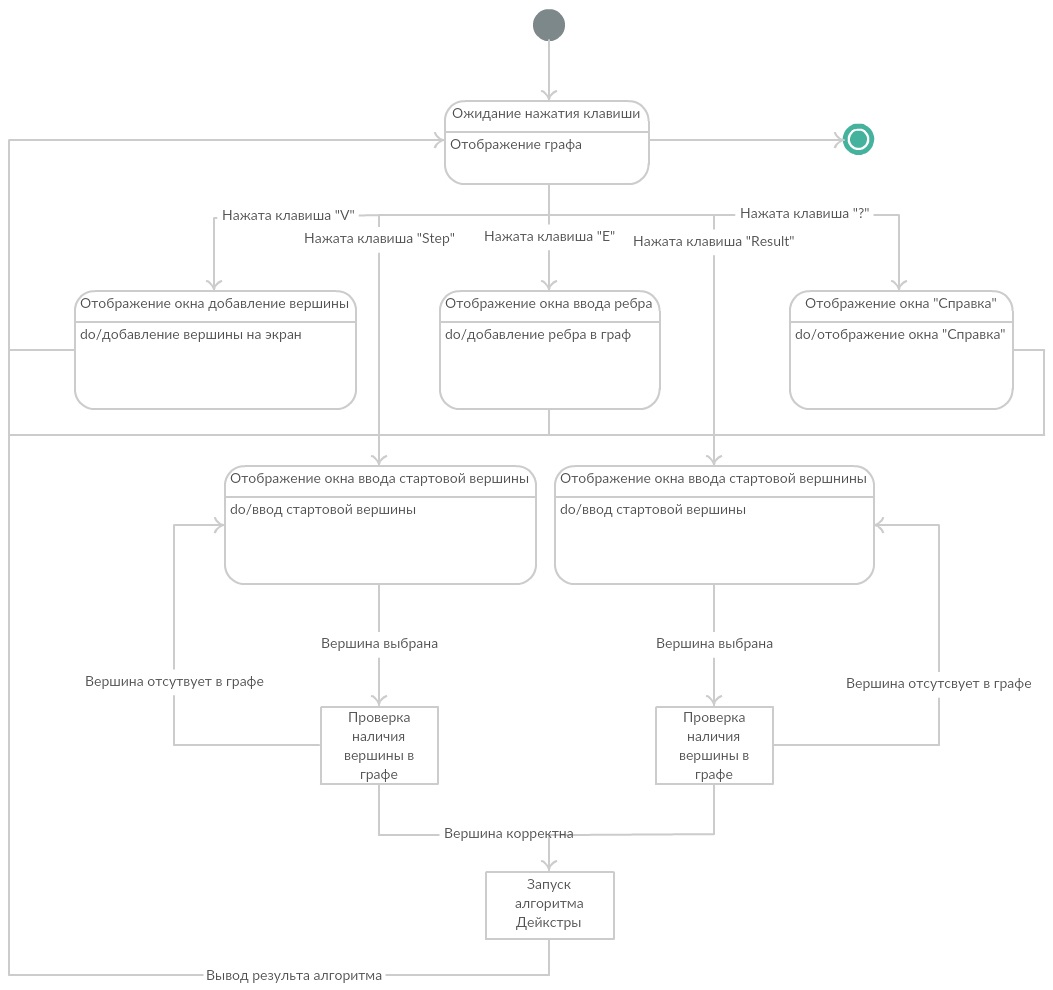


Рисунок 2 – use case диаграмма

Рисунок 3 – диаграмма состояний

**1.1.2. Требования к вводу исходных данных**

Ввод компонент графа должен осуществляться с файла, в котором записаны ребра в формате: (from, to, weight), либо же прямо в приложении с возможностью отдельного добавления вершин и ребер.

При неудачной попытке добавления вершины, должно выводится окно, сообщающее об этом. Также при попытке ввода ребра должны выводится окна со следующей информацией:

* Отсутствует начальная вершина – при вводе начальной вершины ребра, которая отсутствует в графе;
* Отсутствует конечная вершина – при вводе начальной вершины ребра, которая отсутствует в графе;
* Некорректное значение веса ребра – при вводе отрицательного веса ребра.

**1.1.3. Требования к визуализации**

Алгоритм Дейкстры можно разделить на 3 основных итерации: выбор не просмотренной вершины с наименьшим текущим расстоянии до начальной вершины, выбор ребра, исходящего из текущей просматриваемой вершины и имеющего наименьший вес, релаксация.

Следовательно, необходимо, чтобы программа визуализировала все 3 итерации алгоритма: при выборе следующей текущей вершины, она перекрашивается в другой цвет, при выборе ребра для перехода, оно также перекрашивается в другой цвет, при успешной релаксации, значение текущего расстояния должно обновляться около соответствующей вершины, ну и при завершении просмотра вершины, когда все исходящие из нее ребра просмотрены, она перекрашивается в новый цвет. Ниже представлены эскизы графического интерфейса.

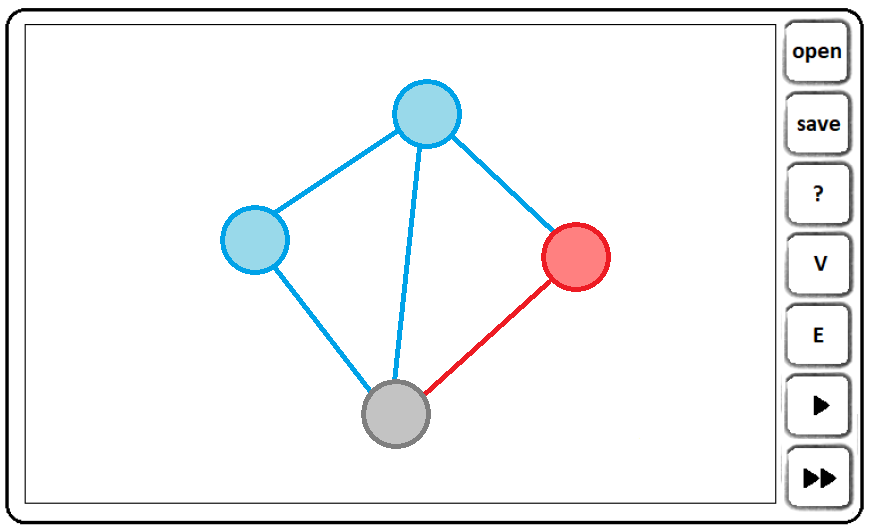
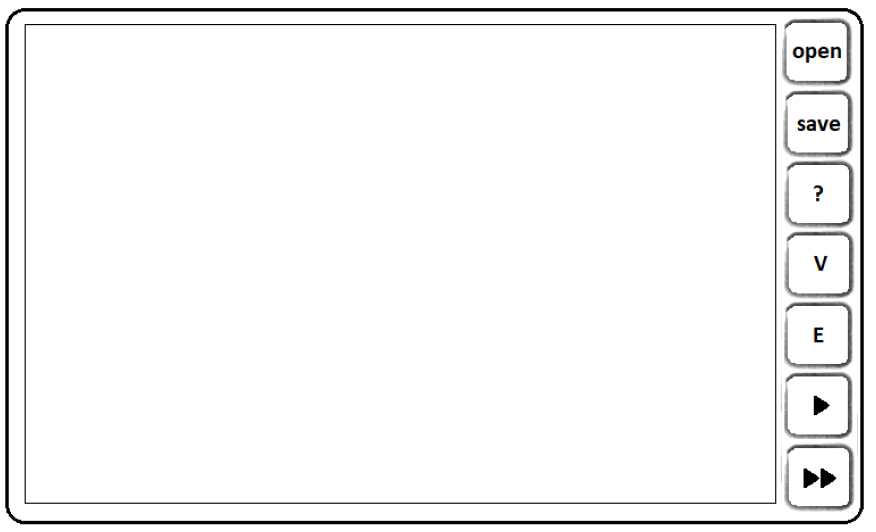
Рисунок 4 – основное окно приложения

Рисунок 5 – построенный граф, с выделенными элементами: серая вершина – просмотренная, красное ребро – просмотренное, красная вершина – текущая просматриваемая, синие ребра и вершины – непросмотренные.



**1.1.4. Требования к выводу результата**

В качестве вывода, программа должна выводить кратчайшие расстояния до каждой вершины в графе, кроме начальной, а также сами пути.

**1.2. Уточнение требований после сдачи прототипа**

Необходимо реализовать пошаговое выполнение алгоритма с визуализацией, каждого шага, добавление графа с файла и сохранение результатов в файл.

**1.3. Уточнение требований после сдачи первой версии**

Добавление логов, корректная обработка ввода графа с файла, структурирование проекта.

**2. План разработки и распределение ролей в бригаде**

**2.1. План разработки**

**Спецификация - 03.07.2019:**

Будет утверждена архитектура проекта, основные требования и возможности.

**Прототип – 04.07.2019:**

Будут реализованы все кнопки интерфейса, всплывающие окна с дополнительной информацией.

**Итерация 1 – 06.07.2019:**

Будет реализован ввод произвольного графа графическим способом. Планируется реализовать алгоритм Дейкстры без связи с графический интерфейсом.

**Итерация 2 – 08.07.2019:**

Будет реализована связь графического интерфейса с основным алгоритмом. Добавится подробный вывод информации о работе алгоритма на каждой итерации, а также ввод структуры графа из файла.

**Итерация 3 – 12.07.2019:**

Будет проведена отладка всего проекта.

**2.2. Распределение ролей в бригаде**

В табл. 1 представлены основные обязанности каждого члена бригады.

Таблица 1 – обязанности членов бригады

|  |  |
| --- | --- |
| Имя студента | Обязанность |
| Дорох С.В. | Разработка графического интерфейса.  Ведение Git-репозитория. |
| Ильясов А.В. | Разработка пошаговой работы алгоритма.  Выведение результата программы. |
| Павлов А.П. | Разработка архитектуры проекта(uml-диаграммы).  Разработка связи графического интерфейса и алгоритма. |

**3. Особенности реализации**

**3.1. Использованные сторонние библиотеки**

Использована библиотека JGraphX, из которой взяты классы mxGrpah и mxCell.

mxGraph – класс, представляющий собой граф, содержащий методы, для удобной реализации алгоритмов на графах.

mxCell – класс с помощью которого можно хранить вершины и ребра.

Эта библиотека была выбрана по 2 основным причинам:

1) удобный и понятный интерфейс;

2) совместимость с java.Swing.

**3.2. Использованные структуры данных**

В проекте реализованы 3 основных класса: Main, GUI, Dijkstra.

**Class Main:**

Главный класс, содержащий метод main, в котором происходит создание объекта графического интерфейса.

**Class GUI:**

Класс, наследующий класс JApplet. Реализует графический интерфейс.

Компоненты java.swing:

* JButton addVertexButton - кнопка для добавления вершины в граф
* JButton addEdgeButton - кнопка для добавления ребра в граф
* JButton nextButton - кнопка перехода к следующей итерации
* JButton executeButton - кнопка перехода к результату алгоритма
* JButton helpButton - справочная кнопка
* JButton fileButton – кнопка для выбора файла на вход
* JButton saveButton – кнопка для сохранения результатов в файл
* JButton showResultAlgoButton – кнопка для показа результатов работы алогритма

Вложенные классы:

RoundedBorder – класс, реализующий интерфейс Border и задающий форму кнопок меню.

Методы класса:

initButtons() - задание формы кнопок и их инициализация.

init() - метод для вызова инициализации графа, кнопок и кругового макета.

getAddVertexButton() – метод, возвращающий объект кнопки для вставки вершины.

getAddEdgeButton() – метод, возвращающий объект кнопки для вставки ребра.

getNextButton() – метод, возвращающий объект кнопки для перехода к следующей итерации.

getExecuteButton() – метод, возвращающий объект кнопки для перехода к конечному результату алгоритма.

getFileButton() – метод, возвращающий объект кнопки для ввода графа из файла.

getSaveButton() – метод, возвращающий объект кнопки для сохранения результатов работы программы в файл.

getShowResultAlgoButton() – метод, возвращающий объект кнопки для показа результатов программы.

getLogsPane() – метод, возвращающий объект для отображения логов программы.

getDimension() – метод, возвращающий объект разрешения окна.

createFileChooser(String exampleFileName) – метод выбора файла для считывания.

exceptionChecker(int flag) - метод для вывода сообщения в случае неудачного добавления ребра.

**Class Dijkstra:**

Класс, реализующий алгоритм Дейкстры.

Атрибуты:

enum Steps{UNVISITED\_VERTEX\_SELECTION, NEAREST\_NEIGHBOR\_SELECTION, RELAXATION} – перечисление, хранящее варианты итераций алгоритма.

step – переменная, хранящая элемент Steps, показывающая, какая итерация алгоритма должна выполняться следующей.

HashMap<Object, Double> distance - карта вершин и расстояний до них.

ArrayList<Object> unvisitedVertices - карта вершин и расстояний до них.

HashMap<Object, TreeSet<Object>> outgoingEdges - карта вершин и выходящих из них ребер.

HashMap<Object, Object> parents - список пар вершин, по которым восстанавливаются кратчайшие пути.

mxGraph graph - граф, в котором ищутся кратчайшие пути.

Object source - начальная вершина в алгоритме Дейкстры.

Методы класса:

Dijkstra(mxGraph graph, Object source) – конструктор класса.

getStep() – метод, возвращающий текущее состояние атрибута step.

isNextStep() – метод, проверяющий, не завершился ли алгоритм.

removeVertex(Object vertex, Object edge) – метод удаляющий вершину из списка непросмотренных.

getPaths() – метод, считающий расстояния до всех вершин. Может вызываться, если не требуется визуализация.

selectUnvisitedVertex() – метод выбора следующей просматриваемой вершины из еще непросмотренных.

selectNearestNeighbor(Object vertex) – метод выбора непросмотренной вершины, ближайшей к текущей просматриваемой.

relax(Object edge) – метод обновления расстояния до вершины.

minDistance() – метод поиска кратчайшего расстояния до непросмотренных вершин.

pathRestoration(Object v) – метод восстановления пути до вершины.

toString() – метод, возвращающий строковое состояние объекта(расстояния до вершин и сами пути).

**Class GAdapter:**

Класс, содержащий в себе методы и поля для создания и работы с графом.

Атрибуты:

Dijkstra dijkstra – объект класса Dijkstra.

mxCircleLayout layout – круговой макет заданного радиуса.

mxGraph graph – граф, которым необходимо работать.

Object parent – значение, которое будет использоваться как родитель для всех новых ячеек.

mxGraphComponent graphComponent - компонента графа, которую необходимо отобразить в окне.

Object currV –выбранная на данный момент вершина.

Object currE –выбранное на данный момент ребро.

Object cell – просматриваемая на данный момент ячейка.

boolean checker – флаг, для проверки наличия начальной вершины.

Object start – начальная вершина в графе.

Методы класса:

getDijkstra() - метод, возвращающий объект класса Dijkstra.

getStart() – метод, возвращающий объект начальной вершины.

initCircleLayout() - метод инициализации кругового макета.

initGraph() - метод инициализации графа.

fileReader(JFileChooser jFileChooser, Container contentPane) – метод для считывания данных из файла.

addVertexButtonEventLister(String input) – метод для добавления вершины в граф.

DoubleParser(mxCell From, mxCell To, String weight, String strLine) – метод для проверки введённого для ребра веса и добавления его в граф.

nextIteration() – метод для перехода к следующей итерации.

checkStartVertex() – метод для проверки наличия в графе начальной вершины.

cellFounder(String result) – метод для поиска вершины в графе.

nextStep(Object cell) – метод для выбора объекта на следующей итерации.

isEnable(Object v1, Object v2) – метод для проверки на наличие между двумя вершинами ребра.

**заключение**

В ходе данной учебной практики были изучены основы программирования на языке Java, пройден интерактивный курс «Java.Базовый курс». После чего была разработана программа, которая с помощью алгоритма Дейкстры находит кратчайшие пути в графе из начальной вершины с визуализацией шагов алгоритма.

**список использованных источников**

1)https://docs.oracle.com/en/java/

2) https://jgraph.github.io/mxgraph/docs/tutorial.html

3) https://www.baeldung.com/jgrapht

**Приложение**

**Исходный код класса Main**

**import javax.swing.**\*;  
**import java.awt.**\*;  
  
  
**public class Main** {  
 **public static void** main(**String**[] args) {  
 **GUI** applet = **new** GUI();  
 applet.init();  
 **JFrame** frame = **new** JFrame();  
 frame.getContentPane().add(applet);  
 Toolkit toolkit = Toolkit.getDefaultToolkit();  
 **Dimension** dimension = toolkit.getScreenSize();  
 frame.setLocation(dimension.width/2 - 400, dimension.height/2 - 250);  
 frame.setTitle("Dijkstra vizualizator");  
 frame.setDefaultCloseOperation(**JFrame**.EXIT\_ON\_CLOSE);  
 frame.pack();  
 frame.setVisible(**true**);  
 }  
}

**Исходный код класса EdgeComparator**

**Исходный код класса Dijkstra**

**package Dijkstra**;  
  
**import com.mxgraph.model.mxCell**;  
**import com.mxgraph.view.mxGraph**;  
  
**import javax.swing.**\*;  
**import java.util.**\*;  
  
**import static java.lang.Double**.POSITIVE\_INFINITY;  
**import static java.lang.Double**.min;  
  
**public class Dijkstra** {  
 **public enum** Steps{UNVISITED\_VERTEX\_SELECTION, NEAREST\_NEIGHBOR\_SELECTION, RELAXATION};  
  
 /\*\*  
 \* step - метка, указывающая, какой шаг алгоритма должен выполнятся следующим  
 \* distance - карта вершин и расстояний до них  
 \* unvisitedVertices - список непросмотренных вершин  
 \* outgoingEdges - карта вершин и выходящих из них ребер  
 \* parents - список пар вершин, по которым восстанавливаются кратчайшие пути  
 \* graph - граф, в котором ищутся кратчайшие пути  
 \* source - начальная вершина в алгоритме Дейкстры  
 \*/  
 **private** Steps step = Steps.UNVISITED\_VERTEX\_SELECTION;  
 **private HashMap**<**Object**, **Double**> distance;  
 **private ArrayList**<**Object**> unvisitedVertices;  
 **private HashMap**<**Object**, **TreeSet**<**Object**>> outgoingEdges;  
 **private HashMap**<**Object**, **Object**> parents;  
 **private mxGraph** graph;  
 **private Object** source;  
  
 **public** Dijkstra(**mxGraph** graph, **Object** source) {  
 distance = **new** HashMap<>();  
 unvisitedVertices = **new** ArrayList<>();  
 outgoingEdges = **new** HashMap<>();  
 parents = **new** HashMap<>();  
 **this**.graph = graph;  
 **this**.source = source;  
  
 /\*\*  
 \* изначально все вершины считаются непросмотренными и расстояния до них равно бесконечности  
 \*/  
 **for** (**Object** v : graph.getChildVertices(graph.getDefaultParent())) {  
 distance.put(v, POSITIVE\_INFINITY);  
 unvisitedVertices.add(v);  
 }  
  
 /\*\*  
 \* расстояние от начальной вершины до себя, очевидно, равно 0  
 \*/  
 distance.put(source, 0.0);  
  
 /\*\*  
 \* дублирование вершин с выходящими из них ребрами для возможности реализации визуализации  
 \*/  
 **for** (**Object** v : graph.getChildVertices(graph.getDefaultParent())) {  
 **TreeSet**<**Object**> set = **new** TreeSet<>(**new** EdgeComparator());  
 set.addAll(**Arrays**.asList(graph.getOutgoingEdges(v)));  
 outgoingEdges.put(v, set);  
 }  
  
 }  
  
 /\*\*  
 \* получение текущего значения step для координирования выполнения алгоритма  
 \*/  
 **public** Steps getStep() {  
 **return** step;  
 }  
  
 /\*\*  
 \* алгоритм заканчивается, когда все вершины просмотрены,  
 \* либо когда расстояния до всех оставшихся непросмотренных вершин равно бесконености  
 \*/  
 **public boolean** isNextStep() {  
 **return** !unvisitedVertices.isEmpty() && minDistance() < POSITIVE\_INFINITY;  
 }  
  
 /\*\*  
 \* после просмотра всех исходящих ребер из вершины  
 \* она удаляется из списка непросмотренных  
 \*/  
 **public void** removeVertex(**Object** vertex, **Object** edge) {  
 **if** (edge.equals(vertex))  
 unvisitedVertices.remove(vertex);  
 }  
  
 /\*\*  
 \* выбор следующей просматриваемой вершины из еще непросмотренных  
 \*/  
 **public Object** selectUnvisitedVertex(**JTextPane** textPane) {  
 step = Steps.NEAREST\_NEIGHBOR\_SELECTION;  
  
 /\*\*  
 \* выбирается непросмотренная вершина с наименьшим текущем расстоянии до начальной вершины  
 \*/  
 **Object** vertex = **new** mxCell();  
 **double** mindistance = minDistance();  
  
 **for** (**Object** v: distance.keySet()) {  
 **if** (unvisitedVertices.contains(v) && distance.get(v).equals(mindistance)) {  
 vertex = v;  
 **break**;  
 }  
 }  
  
 textPane.setText(textPane.getText() + "выбрана вершина '" + ((**mxCell**) vertex).getId() + "'" + "\n");  
 **return** vertex;  
 }  
  
 /\*\*  
 \* выбор непросмотренной вершины, ближайшей к текущей просматриваемой  
 \*/  
 **public Object** selectNearestNeighbor(**Object** vertex, **JTextPane** textPane) {  
 **if** (outgoingEdges.get(vertex).isEmpty()) {  
 step = Steps.UNVISITED\_VERTEX\_SELECTION;  
 textPane.setText(textPane.getText() + " все исходящие ребра просмотрены" + "\n");  
 **return** vertex;  
 }  
 **else if** (unvisitedVertices.contains(((**mxCell**) outgoingEdges.get(vertex).first()).getTarget())) {  
 step = Steps.RELAXATION;  
 **Object** result = outgoingEdges.get(vertex).first();  
 outgoingEdges.get(vertex).remove(result);  
 textPane.setText(textPane.getText() + " выбрано ребро до вершины '" + ((**mxCell**) result).getTarget().getId() + "'" + "\n");  
 **return** result;  
 }  
 **else** {  
 **Object** result = outgoingEdges.get(vertex).first();  
 outgoingEdges.get(vertex).remove(result);  
 **return** vertex;  
 }  
  
 }  
  
 /\*\*  
 \* обновление расстояния до вершины  
 \*/  
 **public double** relax(**Object** edge, **JTextPane** textPane) {  
 **Object** source = ((**mxCell**) edge).getSource();  
 **Object** target = ((**mxCell**) edge).getTarget();  
 **double** value = (**double**)((**mxCell**) edge).getValue();  
  
 **double** newDistance = **Math**.min(distance.get(target), distance.get(source) + value);  
  
 **if** (newDistance < distance.get(target)) {  
 textPane.setText(textPane.getText() + " релаксация прошла успешно:" + "\n" + " " + distance.get(source) +  
 " + " + value + " < " + distance.get(target) + "\n");  
 parents.put(target, source);  
 }  
 **else** {  
 textPane.setText(textPane.getText() + " релаксация прошла неуспешно:" + "\n" + " " + distance.get(source)  
 + " + " + value + " ≥ " + distance.get(target) + "\n");  
 }  
 distance.put(target, newDistance);  
 step = Steps.NEAREST\_NEIGHBOR\_SELECTION;  
  
 **return** newDistance;  
 }  
  
 /\*\*  
 \* поиск кратчайшего расстояния до непросмотренных вершин  
 \*/  
 **private double** minDistance() {  
 **double** result = POSITIVE\_INFINITY;  
 **for** (**Object** v: unvisitedVertices)  
 **if** (distance.get(v) < result)  
 result = distance.get(v);  
  
 **return** result;  
 }  
  
 /\*\*  
 \* восстановление пути до вершины  
 \*/  
 **public String** pathRestoration(**Object** v) {  
 **StringBuilder** builder = **new** StringBuilder();  
 **Stack**<**Object**> stack = **new** Stack<>();  
  
 **do** {  
 stack.push(v);  
 v = parents.get(v);  
 } **while** (!v.equals(source));  
 stack.push(source);  
  
 **while** (!stack.empty())  
 builder.append(((**mxCell**) stack.pop()).getId().toString() + " ");  
 builder.append("\n");  
  
 **return** builder.toString();  
 }  
  
 **public String** toString() {  
 **StringBuilder** builder = **new** StringBuilder();  
 **for** (Map.Entry e: distance.entrySet()) {  
 **if** (!((**mxCell**) e.getKey()).equals(source)) {  
 builder.append("вершина = ");  
 builder.append(((**mxCell**) e.getKey()).getId());  
 builder.append(", расстояние = ");  
 builder.append(e.getValue().toString());  
 builder.append("\n");  
 }  
 }  
  
 builder.append("\n" + "пути:" + "\n");  
  
 **for** (**Object** v: parents.keySet())  
 builder.append(pathRestoration(v));  
  
 **return** builder.toString();  
 }  
}

**Исходный код класса GUI**

**import com.mxgraph.model.**\*;  
**import javax.swing.**\*;  
**import java.awt.**\*;  
**import java.awt.event.**\*;  
**import java.io.**\*;  
**import javax.swing.border.**Border;  
**import javax.swing.filechooser.**FileFilter;  
  
/\*\*  
 \* Класс реализации графического интерфейса  
 \*/  
**public class GUI extends JApplet** {  
 **private static final Dimension** DEFAULT\_SIZE = **new** Dimension(800, 500);  
 **private GAdapter** graph = **new** GAdapter();  
 /\*\*  
 \* addVertexButton - кнопка для добавления вершины в граф  
 \* addEdgeButton - кнопка для добавления ребра в граф  
 \* nextButton - кнопка перехода к следующей итерации  
 \* executeButton - кнопка перехода к результату алгоритма  
 \* helpButton - справочная кнопка  
 \* saveButton -кнопка для сохранения файла  
 \* fileButton - кнопка для считывания из файла  
 \* showResultAlgoButton - кнопка для показа результата программы  
 \* TITLE\_message - титульник справочного окна  
 \*/  
 **private static JButton** *addVertexButton* = **new** JButton("V");  
 **private static JButton** *addEdgeButton* = **new** JButton("E");  
 **private static JButton** *nextButton* = **new** JButton("▶");  
 **private static JButton** *executeButton* = **new** JButton("▶▶");  
 **private static JButton** *helpButton* = **new** JButton(" ? ");  
 **private static JButton** *fileButton* = **new** JButton("\uD83D\uDCC1");  
 **private static JButton** *saveButton* = **new** JButton("\uD83D\uDCBE");  
 **private static JButton** *showResultAlgoButton* = **new** JButton("\uD83C\uDFC1");  
 **private static JCheckBox** *logChecker* = **new** JCheckBox("логи");  
 **private static JTextPane** *logsPane* = **new** JTextPane();  
 **private JScrollPane** scrollPane = **new** JScrollPane(*logsPane*, **JScrollPane**.VERTICAL\_SCROLLBAR\_AS\_NEEDED, **JScrollPane**.HORIZONTAL\_SCROLLBAR\_NEVER);  
  
 **private final String** TITLE\_message = "Справка";  
  
 /\*\*  
 \* Задание формы кнопок и их инициализация  
 \*/  
 **private void** initButtons() {  
 *addVertexButton*.setBounds(10, 135, 50, 50);  
 *addVertexButton*.addActionListener(e -> {  
 **String** input = **JOptionPane**.showInputDialog(  
 **GUI**.**this**,  
 "<html><h2>Введите вершину:");  
 graph.addVertexButtonEventListener(input);  
 });  
 getContentPane().add(*addVertexButton*);  
 *addVertexButton*.setBorder(**new** RoundedBorder(10));  
  
 *addEdgeButton*.setBounds(10, 190, 50, 50);  
 *addEdgeButton*.addActionListener(e -> {  
 **String** from = **JOptionPane**.showInputDialog(  
 **GUI**.**this**,  
 "<html><h2>Введите первую вершину:");  
 **if**(from == **null**) {  
 exceptionChecker(1);  
 **return**;  
 }  
 **mxCell** vFrom = graph.cellFounder(from);  
 **if**(vFrom == **null**) {  
 exceptionChecker(1);  
 **return**;  
 }  
 **String** to = **JOptionPane**.showInputDialog(  
 **GUI**.**this**,  
 "<html><h2>Введите вторую вершину:");  
 **if**(to == **null**) {  
 exceptionChecker(2);  
 **return**;  
 }  
 **mxCell** vTo = graph.cellFounder(to);  
 **if**(vTo == **null**) {  
 exceptionChecker(2);  
 **return**;  
 }  
 **String** weight = **JOptionPane**.showInputDialog(  
 **GUI**.**this**,  
 "<html><h2>Введите вес ребра:");  
 **if**(weight == **null**)  
 **return**;  
 graph.DoubleParser(vFrom, vTo, weight, weight);  
 });  
 getContentPane().add(*addEdgeButton*);  
 *addEdgeButton*.setBorder(**new** RoundedBorder(10));  
  
 *nextButton*.setBounds(10, 245, 50, 50);  
 *nextButton*.addActionListener(e -> {  
 graph.nextIteration();  
 });  
 getContentPane().add(*nextButton*);  
 *nextButton*.setBorder(**new** RoundedBorder(10));  
 *nextButton*.setEnabled(**false**);  
  
 *executeButton*.setBounds(10, 300, 50, 50);  
 *executeButton*.addActionListener(e -> {  
 **while**(graph.nextIteration());  
 *executeButton*.setEnabled(**false**);  
 *saveButton*.setEnabled(**true**);  
 });  
 getContentPane().add(*executeButton*);  
 *executeButton*.setBorder(**new** RoundedBorder(10));  
 *executeButton*.setEnabled(**false**);  
  
 *showResultAlgoButton*.setBounds(10, 355, 50, 50);  
 *showResultAlgoButton*.addActionListener(e -> {  
 **JOptionPane** jOptionPane = **new** JOptionPane();  
 jOptionPane.showMessageDialog(**GUI**.**this**,  
 "<html><h2>Результат работы агоритма Дейкстры:</h2><p>" + ((**mxCell**) graph.getStart()).getId() + " - стартовая вершина\n" + graph.getDijkstra().toString(), "Вывод", **JOptionPane**.INFORMATION\_MESSAGE);  
  
 });  
 getContentPane().add(*showResultAlgoButton*);  
 *showResultAlgoButton*.setBorder(**new** RoundedBorder(10));  
 *showResultAlgoButton*.setEnabled(**false**);  
  
 *helpButton*.setBounds(10, 25, 50, 50);  
 *helpButton*.addActionListener(e -> **JOptionPane**.showMessageDialog(**GUI**.**this**,  
 "<html><h2>Справка:</h2><p>Данный графический интерфейс визуализирует алгоритм<br> поиска кратчайшего пути в графе - алгоритм Дейкстры.<br>" +  
 "<i><br>Описание кнопок:</i> <br>" +  
 "1) V - кнопка добавления вершины в граф;<br>2) E - кнопка добавления ребра в граф;<br>" +  
 "3) ▶ - кнопка для перехода к следующей итерации алгоритма;<br>4) ▶▶ - кнопка вывода конечного результата алгоритма.<br>" +  
 "4) \uD83D\uDCC1 - кнопка для считывания из файла.<br>" +  
 "5) \uD83C\uDFC1 - кнопка показа конечного результата алгоритма<br></p>" +  
 "6) \uD83D\uDCBE - кнопка вывода результата алгоритма в файл<br></p>" +  
 "<br>Значения в () - минимальное расстояние до вершин из начальной.", TITLE\_message, **JOptionPane**.INFORMATION\_MESSAGE));  
  
 getContentPane().add(*helpButton*);  
 *helpButton*.setBorder(**new** RoundedBorder(10));  
  
 *fileButton*.setBounds(10, 80, 50, 50);  
 *fileButton*.addActionListener(e -> {  
 **JFileChooser** jFileChooser = createFileChooser("file.txt");  
 graph.fileReader(jFileChooser, getContentPane());  
 });  
 getContentPane().add(*fileButton*);  
 *fileButton*.setBorder(**new** RoundedBorder(10));  
  
 *saveButton*.setBounds(10, 410, 50, 50);  
 *saveButton*.addActionListener(e -> {  
 **JFileChooser** jFileChooser = createFileChooser("result.txt");  
 **int** i = jFileChooser.showSaveDialog(getContentPane());  
 **File** file = jFileChooser.getSelectedFile();  
 **JOptionPane** jOptionPane = **new** JOptionPane();  
 **if**(i == jFileChooser.APPROVE\_OPTION && file.getName().endsWith("txt")) {  
 **try** {  
 **FileWriter** fw = **new** FileWriter(file);  
 fw.write("Результат работы агоритма Дейкстры:\n" + '\n' + ((**mxCell**) graph.getStart()).getId() + " - стартовая вершина\n"+ '\n' + graph.getDijkstra().toString());  
 jOptionPane.showMessageDialog(**GUI**.**this**, "<html><h2>Файл сохранен успешно!</h2><p>", "Сохранение файла", **JOptionPane**.INFORMATION\_MESSAGE);  
 fw.flush();  
 } **catch** (**Exception** ex) {  
 ex.printStackTrace();  
 }  
 }  
 **else if**(i == jFileChooser.CANCEL\_OPTION){}  
 **else** {  
 jOptionPane.showMessageDialog(**GUI**.**this**, "<html><h2>Ошибка сохранения файла!</h2><p>" + "<html><h2>Выберете файл с расширение \*.txt!</h2><p>", "Ошибка сохранения файла", **JOptionPane**.INFORMATION\_MESSAGE);  
 }  
 });  
 getContentPane().add(*saveButton*);  
 *saveButton*.setBorder(**new** RoundedBorder(10));  
 *saveButton*.setEnabled(**false**);  
  
 *logChecker*.setBounds(740, 3, 55, 25);  
 getContentPane().add(*logChecker*);  
 *logChecker*.addItemListener(e -> {  
 **if** (e.getStateChange() == **ItemEvent**.SELECTED) {  
 scrollPane.setVisible(**true**);  
 }  
 **else** {  
 scrollPane.setVisible(**false**);  
 }  
 });  
 *logsPane*.setBounds(555, 30, 220, 1000);  
 *logsPane*.setBackground( **new** Color(238, 238, 238));  
  
  
 scrollPane.setBounds(555, 30, 220, 400);  
 getContentPane().add(scrollPane);  
 scrollPane.setVisible(**false**);  
 }  
  
 /\*\*  
 \* Метод для вызова инициализации графа, кнопок и кругового макета  
 \*/  
 **public void** init() {  
 initButtons();  
 graph.initGraph();  
 graph.initCircleLayout();  
 getContentPane().add(graph.getGraphComponent());  
 setPreferredSize(DEFAULT\_SIZE);  
 }  
  
 /\*\*  
 \*Геттер возвращающий объект кнопки для вставки вершины  
 \*/  
 **static JButton** getAddVertexButton() { **return** *addVertexButton*; }  
 /\*\*  
 \*Геттер возвращающий объект кнопки для вставки ребра  
 \*/  
 **static JButton** getAddEdgeButton() { **return** *addEdgeButton*; }  
 /\*\*  
 \*Геттер возвращающий объект кнопки для перехода к следующей итерации  
 \*/  
 **static JButton** getNextButton() { **return** *nextButton*; }  
 /\*\*  
 \*Геттер возвращающий объект кнопки для перехода к конечному результату алгоритма  
 \*/  
 **static JButton** getExecuteButton() { **return** *executeButton*; }  
 /\*\*  
 \*Геттер возвращающий объект кнопки для ввода графа из файла  
 \*/  
 **static JButton** getFileButton() { **return** *fileButton*; }  
 /\*\*  
 \*Геттер возвращающий объект кнопки для сохранения результатов в файл  
 \*/  
 **static JButton** getSaveButton() { **return** *saveButton*; }  
 /\*\*  
 \*Геттер возвращающий объект кнопки для показа результатов программы  
 \*/  
 **static JButton** getShowResultAlgoButton() { **return** *showResultAlgoButton*; }  
 /\*\*  
 \*Геттер возвращающий объект для отображения логов прогрммы.  
 \*/  
 **static JTextPane** getLogsPane() { **return** *logsPane*; }  
 /\*\*  
 \*Геттер возвращающий объект разрешения окна.  
 \*/  
 **static Dimension** getDimension() {  
 **return** DEFAULT\_SIZE;  
 }  
  
 /\*\*  
 \*Метод выбора файла для считывания  
 \*/  
 **private JFileChooser** createFileChooser(**String** exampleFileName){  
 **JFileChooser** jFileChooser = **new** JFileChooser();  
 jFileChooser.setCurrentDirectory(**new** File("."));  
 jFileChooser.setSelectedFile(**new** File(exampleFileName));  
 jFileChooser.setFileFilter(**new** FileFilter() {  
 @Override  
 **public boolean** accept(**File** f) {  
 **return** f.getName().endsWith("txt");  
 }  
 @Override  
 **public String** getDescription() {  
 **return** "Текстовые файлы (\*.txt)";  
 }  
 });  
 **return** jFileChooser;  
 }  
  
 /\*\*  
 \*Метод для вывода сообщения в случае наудачного добавления ребра  
 \*/  
 **private void** exceptionChecker(**int** flag) {  
 **if**(flag == 1)  
 **JOptionPane**.showMessageDialog(**null**, "Пропущен ввод стартовой вершины", "Warning!", **JOptionPane**.PLAIN\_MESSAGE);  
 **else if**(flag == 2)  
 **JOptionPane**.showMessageDialog(**null**, "Пропущен ввод конечной вершины", "Warning!", **JOptionPane**.PLAIN\_MESSAGE);  
  
 }  
  
 /\*\*  
 \* Класс с методами и полями, необходимыми для скругления краёв у кнопок  
 \*/  
 **private static class RoundedBorder implements** Border {  
  
 **private int** radius;  
  
 RoundedBorder(**int** radius) {  
 **this**.radius = radius;  
 }  
 */\*  
 Возвращает вставки границы.  
 \*/* **public Insets** getBorderInsets(Component c) {  
 **return new** Insets(**this**.radius+1, **this**.radius+1, **this**.radius+2, **this**.radius);  
 }  
 */\*  
 Возвращает, является ли граница непрозрачной.  
 \*/* **public boolean** isBorderOpaque() {  
 **return true**;  
 }  
 */\*  
 Рисует границу для указанного компонента с указанным положением и размером.  
 \*/* **public void** paintBorder(Component c, Graphics g, **int** x, **int** y, **int** width, **int** height) {  
 g.drawRoundRect(x, y, width-1, height-1, radius, radius);  
 }  
 }  
}

**Исходный код класса GAdapter**

import Dijkstra.\*;

import com.mxgraph.layout.mxCircleLayout;

import com.mxgraph.model.\*;

import com.mxgraph.swing.mxGraphComponent;

import com.mxgraph.util.mxConstants;

import com.mxgraph.view.mxGraph;

import javax.swing.\*;

import java.awt.\*;

import java.io.\*;

import java.util.ArrayList;

class GAdapter {

/\*\*

\* dijkstra - объект класса Dijkstra

\* layout - круговой макет

\* graph - граф, с которым необходимо работать

\* parent - рродитель для всех остальных ячеек

\* graphComponent - компонента для вывода графа в окно

\* currV - текущая вершина

\* currE - текущее ребро

\* cell - текущая ячейка

\* checker - флаг для проверки наличия в графе начальной вершины

\* start - наальная вершина в графе

\*/

private Dijkstra dijkstra;

private mxCircleLayout layout;

private mxGraph graph;

private Object parent;

private mxGraphComponent graphComponent;

private Object currV = new mxCell();

private Object currE = new mxCell();

private Object cell = new mxCell();

private boolean checker = false;

private Object start;

/\*\*

\* Геттер для получения объект класса Dijkstra

\*/

Dijkstra getDijkstra() { return dijkstra; }

/\*\*

\* Геттер для получения объекта начальной вершины

\*/

Object getStart() { return start; }

/\*\*

\* Метод инициализации кругового макета

\*/

void initCircleLayout() {

layout = new mxCircleLayout(graph);

int radius = 150;

layout.setX0((GUI.getDimension().width / 2.0) - 1.33 \* radius);

layout.setY0((GUI.getDimension().height / 2.0) - 1.1 \* radius);

layout.setRadius(radius);

layout.setMoveCircle(true);

layout.execute(graph.getDefaultParent());

}

/\*\*

\* Метод инициализации графа

\*/

void initGraph() {

graph = new mxGraph();

parent = graph.getDefaultParent();

graphComponent = new mxGraphComponent(graph);

}

/\*\*

\* Геттер для получения компоненты графа

\*/

mxGraphComponent getGraphComponent() {

return graphComponent;

}

/\*\*

\* Метод для считывания из файла

\*/

void fileReader(JFileChooser jFileChooser, Container contentPane) {

int i = jFileChooser.showOpenDialog(contentPane);

File pathToFile = jFileChooser.getCurrentDirectory();

File file = jFileChooser.getSelectedFile();

JOptionPane jOptionPane = new JOptionPane();

if (i == jFileChooser.APPROVE\_OPTION && file.getName().endsWith("txt")) {

try {

graph.removeCells(graph.getChildCells(graph.getDefaultParent(), true, true));

GUI.getShowResultAlgoButton().setEnabled(false);

GUI.getAddVertexButton().setEnabled(true);

GUI.getAddEdgeButton().setEnabled(true);

checker = false;

File f = new File(pathToFile.toString(), file.getName());

BufferedReader br = new BufferedReader(new FileReader(f));

String strLine;

while ((strLine = br.readLine()) != null) {

String[] parts = strLine.split(" ", 3);

if (parts.length < 3) {

JOptionPane.showMessageDialog(null, "Некорректная строка: \"" + strLine +

"\"\nЗадавайте ребра в виде: \"начальная вершина\" \"конечная вершина\" \"вес\"", "Warning!", JOptionPane.PLAIN\_MESSAGE);

graph.removeCells(graph.getChildCells(graph.getDefaultParent(), true, true));

return;

}

Object From = null;

Object To = null;

for (Object v : graph.getChildVertices(graph.getDefaultParent())) {

if (parts[0].equals(((mxCell) v).getValue().toString()))

From = v;

if (parts[1].equals(((mxCell) v).getValue().toString()))

To = v;

}

if (From == null) {

From = graph.insertVertex(parent, null, parts[0], 100, 100, 45, 45, "shape=ellipse");

((mxCell) From).setId(parts[0]);

layout.execute(graph.getDefaultParent());

}

if (To == null) {

To = graph.insertVertex(parent, null, parts[1], 100, 100, 45, 45, "shape=ellipse");

((mxCell) To).setId(parts[1]);

layout.execute(graph.getDefaultParent());

}

if (!DoubleParser((mxCell) From, (mxCell) To, parts[2], strLine))

return;

}

GUI.getNextButton().setEnabled(true);

GUI.getExecuteButton().setEnabled(true);

GUI.getFileButton().setEnabled(true);

GUI.getLogsPane().setText("");

} catch (IOException error) {

JOptionPane.showMessageDialog(null, "Ошибка!", "Warning!", JOptionPane.PLAIN\_MESSAGE);

}

} else if (i == jFileChooser.CANCEL\_OPTION) {

} else {

jOptionPane.showMessageDialog(null, "<html><h2>Ошибка открытия файла!</h2><p>" + "<html><h2>Выберете файл с расширение \*.txt!</h2><p>", "Ошибка сохранения файла", JOptionPane.INFORMATION\_MESSAGE);

}

}

/\*\*

\* Метод для добавления вершины в граф

\*/

void addVertexButtonEventListener(String input) {

if (input != null) {

if (input.equals(""))

return;

for (Object v : graph.getChildVertices(graph.getDefaultParent())) {

if (input.equals(((mxCell) v).getValue())) {

JOptionPane.showMessageDialog(null, "Данная вершина уже есть в графе", "Warning!", JOptionPane.PLAIN\_MESSAGE);

return;

}

}

Object tmp = graph.insertVertex(parent, null, input, 100, 100, 45, 45, "shape=ellipse");

((mxCell) tmp).setId(input);

layout.execute(graph.getDefaultParent());

GUI.getNextButton().setEnabled(true);

GUI.getExecuteButton().setEnabled(true);

}

}

/\*\*

\* Метод для проверки введённого для реьра веса и добавления его в граф

\*/

boolean DoubleParser(mxCell From, mxCell To, String weight, String strLine) {

try {

if(Double.parseDouble(weight) < 0) {

JOptionPane.showMessageDialog(null, "Введено ребро отрицательного веса", "Warning!", JOptionPane.PLAIN\_MESSAGE);

return false;

}

if(isEnable(From, To) && isEnable(To, From))

graph.insertEdge(parent, null, Double.parseDouble(weight), From, To);

return true;

}

catch (NumberFormatException w) {

JOptionPane.showMessageDialog(null, "Неккоректная строка: \"" + strLine +

"\"\n<html><i>Совет: используйте \".\" для разделения целой и дробной частей.", "Warning!", JOptionPane.PLAIN\_MESSAGE);

graph.removeCells (graph.getChildCells (graph.getDefaultParent (), true, true));

return false;

}

}

/\*\*

\* Метод перехода к следующей итерации

\*/

boolean nextIteration() {

if(!checker) {

if(!checkStartVertex())

return false;

}

if (dijkstra.getStep() == Dijkstra.Steps.NEAREST\_NEIGHBOR\_SELECTION)

cell = currV;

else if (dijkstra.getStep() == Dijkstra.Steps.RELAXATION)

cell = currE;

Object result = nextStep(cell);

if (result instanceof Double) {

Object target = ((mxCell) cell).getTarget();

((mxCell) target).setValue("\n\n"+ ((mxCell) target).getId() + "\n\n(" + result + ")");

graph.setCellStyle("defaultVertex;shape=ellipse", new Object[]{target});

graph.setCellStyles(mxConstants.STYLE\_STROKECOLOR, "FA8072", new Object[]{cell});

}

else if(((mxCell) result).isEdge()) {

cell = result;

graph.setCellStyles(mxConstants.STYLE\_STROKECOLOR, "FA8072", new Object[]{cell});

}

else if(((mxCell) result).isVertex()) {

if(cell.equals(result))

graph.setCellStyle("defaultVertex;shape=ellipse;fillColor=#A9A9A9", new Object[]{result});

else

graph.setCellStyle("defaultVertex;shape=ellipse;fillColor=#FA8072", new Object[]{result});

cell = result;

}

if (!dijkstra.isNextStep()) {

GUI.getNextButton().setEnabled(false);

GUI.getAddEdgeButton().setEnabled(false);

GUI.getAddVertexButton().setEnabled(false);

GUI.getShowResultAlgoButton().setEnabled(true);

GUI.getSaveButton().setEnabled(true);

return false;

}

else {

GUI.getShowResultAlgoButton().setEnabled(true);

GUI.getAddEdgeButton().setEnabled(false);

GUI.getAddVertexButton().setEnabled(false);

return true;

}

}

/\*\*

\* Метод для проверки наличия в графе начальной вершины

\*/

private boolean checkStartVertex() {

Object[] tmp = graph.getChildVertices(graph.getDefaultParent());

ArrayList<String> vertices = new ArrayList<>();

for(Object v: tmp)

vertices.add((String)((mxCell) v).getValue());

String[] array = vertices.toArray(new String[vertices.size()]);

Object result = JOptionPane.showInputDialog(

null,

"Выбирете начальную вершину :",

"Выбор вершины",

JOptionPane.QUESTION\_MESSAGE, null,

array, array[0]);

if(result == null) return false;

for(Object v: tmp)

if((result.toString()).equals(((mxCell) v).getValue())) {

checker = true;

graph.setCellStyle("defaultVertex;shape=ellipse;fillColor=#A9A9A9", new Object[]{v});

start = v;

dijkstra = new Dijkstra(graph, v);

return true;

}

return false;

}

/\*\*

\* Метод для поиска вершины в графе

\*/

mxCell cellFounder(String result) {

for(Object v: graph.getChildVertices(graph.getDefaultParent())) {

if (result.equals(((mxCell) v).getValue())) {

return (mxCell) v;

}

}

return null;

}

/\*\*

\* Метод для выбора объекта на следующей итерации

\*/

private Object nextStep(Object cell) {

Object result = new mxCell();

switch (dijkstra.getStep()) {

case UNVISITED\_VERTEX\_SELECTION:

currV = dijkstra.selectUnvisitedVertex(GUI.getLogsPane());

result = currV;

break;

case NEAREST\_NEIGHBOR\_SELECTION:

currE = dijkstra.selectNearestNeighbor(cell, GUI.getLogsPane());

dijkstra.removeVertex(cell, currE);

result = currE;

break;

case RELAXATION:

result = dijkstra.relax(cell, GUI.getLogsPane());

break;

}

return result;

}

/\*\*

\* Метод для проверки на наличие между двумя вершинами ребра

\*/

private boolean isEnable(Object v1, Object v2) {

for(Object v: graph.getEdgesBetween(v1, v2))

if(((mxCell) v).getTarget().equals(v1)) {

JOptionPane.showMessageDialog(null, "Между вершинами " + ((mxCell) v1).getValue().toString()

+ " и " + ((mxCell) v2).getValue().toString() + " уже существует ребро!", "Warning!", JOptionPane.PLAIN\_MESSAGE);

return false;

}

return true;

}

}