Санкт-Петербургский политехнический университет Петра Великого

Институт компьютерных наук и технологий

Высшая школа интеллектуальных систем и суперкомпьютерных технологий

**КУРСОВОЙ ПРОЕКТ**

**Дисциплина:** Алгоритмы и структуры данных

**Тема:** Разработка GUI приложения для алгоритма нахождения реберной связности невзвешенного графа.

Выполнила

студентка гр. №3530903/80003 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Астудина А.П.

(подпись)

Руководитель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Глухих М.И.

(подпись)

«\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2019 г.

Санкт-Петербург   
2019

Оглавление

[Техническое задание 2](#_Toc27233200)

[Метод решения 4](#_Toc27233201)

[Листинг 6](#_Toc27233202)

[**Main.java** and **EdgeConnectivity.java** 6](#_Toc27233203)

[**DrawGraph.java** 9](#_Toc27233204)

[**Graph.java** 12](#_Toc27233205)

[**GraphException.java** 19](#_Toc27233206)

[**IncorrectGraphEntryException.java** 19](#_Toc27233207)

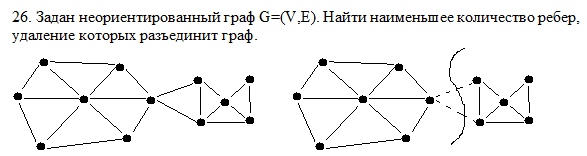
[**InputFileIsEmptyException.java** 19](#_Toc27233208)

[**NoInputFileException.java** 19](#_Toc27233209)

[Тестирование 20](#_Toc27233210)

[Скриншоты программы: 21](#_Toc27233211)

# Техническое задание

Задан неориентированный граф G=(V,E). Найти наименьшее количество ребер, удаление которых разъединит граф. 

GitHub репозиторий проекта: <https://github.com/astudina/EdgeConnectivity>

# Метод решения

В проекте с некоторыми отступлениями использована концепция MVC (Model-View-Controller) для структурирования кода и отделения логики от визуализации.

Main.java – осуществляет запуск приложения.

EdgeConnectivity.java - графическое представление главного окна, меню.

DrawGraph.java – отрисовка графа.

Graph.java – осуществляет обработку событий (чтение входного файла) и содержит полную реализацию алгоритма.

Порядок работы программы:

При запуске программы в меню главного окна выбирается входной файл (Меню => Ввести граф).

Ввод графа необычен: пользователь сам задает его внешний вид, указывая расположение вершин (подробнее в разделе «Скриншоты программы»).

В случае некорректного ввода (пустой входной файл, отсутствие файла ввода, несоблюдение условий ввода файла) появляются предупреждения. За ошибки отвечают классы IncorrectGraphEntryException.java, InputFileIsEmptyException.java и NoInputFileException.java, наследуемые от GraphException.java (который в свою очередь наследуется от Exception).

После корректного ввода графа пользователь видит в левой части окна введенный им граф, а в правой части тот же граф, с выделенной красным цветом реберной связностью. В случае, если граф изначально несвязен и реберная связность = 0, пользователь получает две одинаковые картинки.

Теперь немного подробнее о самом алгоритме (все далее упомянутые методы находятся в классе Graph.java):

Для нахождения реберной связности нужно перебрать все пары вершин source (сток) и drain (исток), найти количество непересекающихся путей из source в drain и выбрать минимум. Для нахождения количества непересекающихся путей из source в drain воспользуемся алгоритмом нахождения максимального потока ([алгоритм Эдмондса-Карпа](http://neerc.ifmo.ru/wiki/index.php?title=%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%AD%D0%B4%D0%BC%D0%BE%D0%BD%D0%B4%D1%81%D0%B0-%D0%9A%D0%B0%D1%80%D0%BF%D0%B0), метод *int EdmondsKarp()).* Но нам недостаточно просто найти количество ребер, нам нужно знать конкретные ребра, чтобы быть в состоянии их отобразить. Для этого нам и понадобится нахождение минимального разреза (разрез с минимально возможной пропускной способностью) с помощью метода *boolean[][] minCut().* По лемме о максимальном потоке и минимальном разрезе: *Если f(S,T)=c(S,T)f(S,T)=c(S,T), то поток f — максимален, а разрез ⟨S,T⟩⟨S,T⟩ — минимален.*

Также следует упомянуть о наличии методов *boolean bfs()* и *boolean* *dfs().* Первый (поиск в глубину) нужен для нахождения кратчайшего пути из источника в сток в методе *int EdmondsKarp()*, в то время как второй (поиск в ширину) вызывается в *boolean[][] minCut().*

Для нахождения соседей конкретной вершинки используется метод *ArrayList<Integer> neighbours().*

# Листинг

## **Main.java** import java.awt.\*; public class Main { public static void main(String[] args) { EventQueue.*invokeLater*(EdgeConnectivity::new); } } **EdgeConnectivity.java**

import javax.swing.\*;  
import javax.swing.border.TitledBorder;  
import java.awt.\*;  
import java.io.File;  
  
class EdgeConnectivity extends JFrame {  
 private Graph graph = null;  
 private Box box1, box2;  
  
 EdgeConnectivity() {  
 super("Курсовой проект");  
  
 //создаем окно  
 setDefaultCloseOperation(JFrame.*EXIT\_ON\_CLOSE*);  
 setLayout(new GridLayout(1, 2));  
  
 //создаем меню  
 JMenuBar menuBar = new JMenuBar();  
 JMenu menu = new JMenu("Меню");  
  
 JMenuItem help = new JMenuItem("Справочная информация");  
 menu.add(help).addActionListener(e -> JOptionPane.*showMessageDialog*(EdgeConnectivity.this,  
 "Граф - абстрактный математический объект, представляющий собой" + "\n" +  
 "множество вeршин графа и набор ребер, то есть соединений между" + "\n" +  
 "парами вершин. " + "\n" +  
 "Реберная связность графа - наименьшее количество" + "\n" +  
 "ребер, удаление которых приводит к несвязному или тривиальному " + "\n" +  
 "графу." + "\n" + "Например, для несвязного графа рёберная связность равна нулю. " + "\n" +  
 "Для связного графа с единственным мостом рёберная связность " + "\n" +  
 "равна единице."));  
  
 JMenuItem helpEnter = new JMenuItem("Как осуществить ввод?");  
 menu.add(helpEnter).addActionListener(e -> JOptionPane.*showMessageDialog*(EdgeConnectivity.this,  
 "Условия для задания графа:" + "\n" +  
 "1) Первая строка - MxN, где M - количество строк, N - количество символов в строке " + "\n" +  
 "(строго \"\*\" или целые числа)" + "\n" +  
 "2) Прописываем внешний вид графа, не забывая о заданных нами M и N" + "\n" +  
 "3) Записываем связи между вершинами" + "\n" +  
 "Пример:" + "\n" + "4x15" + "\n" + "\*\*1\*\*\*\*\*\*\*2\*\*\*\*" + "\n" + "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" + "\n"  
 + "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" + "\n" + "\*\*4\*\*\*\*\*\*\*3\*\*\*\*" + "\n" + "1 2" + "\n" + "1 4" + "\n"  
 + "1 3" + "\n" + "2 3" + "\n" + "2 4" + "\n" + "3 4"));  
  
 JMenuItem enterGraph = new JMenuItem("Ввести граф");  
 enterGraph.addActionListener(e -> {  
 JFileChooser fc = new JFileChooser();  
 int show = fc.showDialog(null, "Открыть файл");  
 if (show == JFileChooser.*APPROVE\_OPTION*) {  
 File file = fc.getSelectedFile();  
 try {  
 graph = new Graph(file);  
 } catch (NoInputFileException | InputFileIsEmptyException ex) {  
 JOptionPane.*showMessageDialog*(  
 box1.getParent(),  
 ex.getMessage(),  
 "Предупреждение",  
 JOptionPane.*WARNING\_MESSAGE* );  
 } catch (IncorrectGraphEntryException ex) {  
 JOptionPane.*showMessageDialog*(  
 box1.getParent(),  
 ex.getMessage(),  
 "Ошибка",  
 JOptionPane.*ERROR\_MESSAGE* );  
 }  
 box1.removeAll();  
 box1.add(new DrawGraph(graph, false));  
 box1.revalidate();  
 box1.repaint();  
 box2.removeAll();  
 box2.add(new DrawGraph(graph, true));  
 box2.revalidate();  
 box2.repaint();  
 }  
  
 });  
  
 menu.add(enterGraph);  
 menuBar.add(menu);  
 setJMenuBar(menuBar);  
  
 box1 = Box.*createVerticalBox*();  
 TitledBorder titledBorder1 = new TitledBorder("Введенный граф");  
 box1.setBorder(titledBorder1);  
 box1.add(new DrawGraph());  
 add(box1);  
  
 box2 = Box.*createVerticalBox*();  
 TitledBorder titledBorder2 = new TitledBorder("Реберная связность");  
 box2.setBorder(titledBorder2);  
 box2.add(new DrawGraph());  
 add(box2);  
  
 pack();  
 setLocationRelativeTo(null); // центрируем окно  
 setVisible(true);  
 }  
}

## **DrawGraph.java**

import java.awt.Color;  
import java.awt.Dimension;  
import java.awt.FontMetrics;  
import java.awt.Graphics;  
import javax.swing.JPanel;  
  
public class DrawGraph extends JPanel {  
 private int width, height;  
 private boolean answer;  
 private Graph graph;  
  
 DrawGraph() {  
 setPreferredSize(new Dimension(400, 300));  
 width = 30;  
 height = 30;  
 graph = null;  
 }  
  
 DrawGraph(Graph graph, boolean answer) {  
 this();  
 this.answer = answer;  
 this.graph = graph;  
 }  
  
 public void paint(Graphics g) {  
 if (graph == null)  
 return;  
  
 width = this.getWidth() / graph.matrix[0].length;  
 height = this.getHeight() / graph.matrix.length;  
 width = height = Math.*min*(height, width);  
  
 // нашли реберную связность и выделили её красным  
 boolean[][] edgeconnectivity = null;  
 if (answer) {  
 edgeconnectivity = graph.edgeConnectivity();  
 g.setColor(Color.*red*);  
 for (int i = 0; i < graph.numberOfStrings; i++)  
 for (int j = 0; j < graph.numberOfStrings; j++)  
 if (edgeconnectivity[i][j]) {  
 g.drawLine(  
 width \* graph.columnNumber.get(i) + width / 2,  
 height \* graph.lineNumber.get(i) + height / 2,  
 width \* graph.columnNumber.get(j) + width / 2,  
 height \* graph.lineNumber.get(j) + height / 2  
 );  
 }  
 }  
  
 // остальные ребра  
 g.setColor(Color.*black*);  
 for (int i = 0; i < graph.numberOfStrings; ++i)  
 for (int j = 0; j < graph.numberOfStrings; ++j)  
 if (graph.sourceNetwork[i][j] > 0) {  
 assert edgeconnectivity != null;  
 if (!answer || !edgeconnectivity[i][j])  
 g.drawLine(  
 width \* graph.columnNumber.get(i) + width / 2,  
 height \* graph.lineNumber.get(i) + height / 2,  
 width \* graph.columnNumber.get(j) + width / 2,  
 height \* graph.lineNumber.get(j) + height / 2  
 );  
 }  
 // вырисовываем и подписываем вершинки  
 for (int i = 0; i < graph.numberOfStrings; ++i) {  
 FontMetrics f = g.getFontMetrics(); // достать размер  
 String number = Integer.*toString*(graph.vertexList.get(i));  
 int nodeWidth = Math.*max*(width, f.stringWidth(number) + width / 2);  
 int nodeHeight = Math.*max*(height, f.getHeight());  
 int x = width \* graph.columnNumber.get(i) + width / 2;  
 int y = height \* graph.lineNumber.get(i) + height / 2;  
 g.setColor(Color.*white*);  
 g.fillOval(  
 x - nodeWidth / 2,  
 y - nodeHeight / 2,  
 nodeWidth,  
 nodeHeight  
 );  
 g.setColor(Color.*black*);  
 g.drawOval(  
 x - nodeWidth / 2,  
 y - nodeHeight / 2,  
 nodeWidth,  
 nodeHeight  
 );  
 g.drawString(  
 number,  
 x - f.stringWidth(number) / 2,  
 y + f.getHeight() / 2  
 );  
 }  
 }  
}

# **Graph.java**

import java.io.File;  
import java.io.FileNotFoundException;  
import java.util.ArrayList;  
import java.util.LinkedList;  
import java.util.Scanner;  
  
public class Graph {  
 int numberOfStrings;  
 int[][] matrix, sourceNetwork;  
 private int[][] residualNetwork;  
 ArrayList<Integer> columnNumber, lineNumber, vertexList;  
  
 Graph(File file) throws NoInputFileException, InputFileIsEmptyException, IncorrectGraphEntryException {  
 try {  
 Scanner sc = new Scanner(file);  
 // если файл ввода пуст  
 if (!sc.hasNextLine())  
 throw new InputFileIsEmptyException();  
 String string = sc.nextLine();  
 String[] substr;  
 // numberOfStrings и objectsInString задаются строго через "x"  
 if (!string.contains("x")) {  
 throw new IncorrectGraphEntryException();  
 } else {  
 substr = string.split("x");  
 }  
 if (substr.length != 2)  
 // substr[0] - это кол-во строк, substr[1] - кол-во символов в строках  
 throw new IncorrectGraphEntryException();  
 try {  
 numberOfStrings = Integer.*parseInt*(substr[0]); // преобразовали строку в число (задаем только интами!!!)  
 } catch (NumberFormatException ex) {  
 throw new IncorrectGraphEntryException();  
 }  
 int objectsInString; // objectsInString - кол-во объектов в строке (звездочки и числа)  
 try {  
 objectsInString = Integer.*parseInt*(substr[1]);  
 } catch (NumberFormatException ex) {  
 throw new IncorrectGraphEntryException();  
 }  
 // для хранения массива используем матрицу  
 this.matrix = new int[numberOfStrings][objectsInString];  
 for (int i = 0; i < numberOfStrings; i++) {  
 if (!sc.hasNextLine())  
 throw new IncorrectGraphEntryException();  
 string = sc.nextLine();  
 int elements = 0; // elements - это сумма всех звёздочек и чисел в строке  
 StringBuilder temp = new StringBuilder();  
 for (char c : string.toCharArray()) {  
 // обрабатываем ситуацию, когда последним в строке стоит какое-то число (именно число, не цифра)  
 // тогда мы вынуждены считывать до конца, и это будет номер вершинки  
 if (elements > objectsInString - 1)  
 if (Character.*isDigit*(c)) { // если является цифрой  
 elements--;  
 temp.append(c);  
 this.matrix[i][elements] = Integer.*parseInt*(temp.toString());  
 elements++;  
 } else  
 temp = new StringBuilder();  
 // ситуация, когда последний элемент - цифра или звездочка (то есть все окей)  
 if (elements == objectsInString - 1) {  
 if (c == '\*')  
 this.matrix[i][elements] = 0;  
 else if (Character.*isDigit*(c)) {  
 temp.append(c);  
 this.matrix[i][elements] = Integer.*parseInt*(temp.toString());  
 elements++;  
 temp = new StringBuilder();  
 }  
 }  
 if (c == '\*') {  
 if (temp.length() > 0) {  
 this.matrix[i][elements] = Integer.*parseInt*(temp.toString());  
 elements++;  
 if (elements > objectsInString)  
 throw new IncorrectGraphEntryException();  
 temp = new StringBuilder();  
 }  
 this.matrix[i][elements] = 0;  
 elements++;  
 if (elements > objectsInString)  
 throw new IncorrectGraphEntryException();  
 } else if (Character.*isDigit*(c))  
 temp.append(c);  
 else  
 throw new IncorrectGraphEntryException();  
 }  
 // в случае, если ввели граф не учитывая количество символов в строке  
 if (elements < objectsInString || elements > objectsInString) throw new IncorrectGraphEntryException();  
 }  
 columnNumber = new ArrayList<>();// номер столбца в матрице для отрисовки  
 lineNumber = new ArrayList<>();// номер строки  
 vertexList = new ArrayList<>();// список вершинок в том порядке, как они встречаются в матрице для отрисовки  
 for (int i = 0; i < numberOfStrings; i++) // собсвенно говоря, заполняем матрицу для отрисовки  
 for (int j = 0; j < objectsInString; j++)  
 if (this.matrix[i][j] > 0) {  
 columnNumber.add(j);  
 lineNumber.add(i);  
 vertexList.add(this.matrix[i][j]);  
 }  
 numberOfStrings = vertexList.size();  
 sourceNetwork = new int[numberOfStrings][numberOfStrings]; //sourceNetwork - наша матрица смежности  
 for (int i = 0; i < numberOfStrings; i++)  
 for (int j = 0; j < numberOfStrings; j++)  
 sourceNetwork[i][j] = 0; // занулили на всякий случай  
 while (sc.hasNextLine()) {  
 if (!sc.hasNextInt())  
 throw new IncorrectGraphEntryException();  
 int element = sc.nextInt();  
 if (!vertexList.contains(element))  
 throw new IncorrectGraphEntryException();  
 int firstIndex = vertexList.indexOf(element); // путь из вершинки "а" в "б"  
 if (!sc.hasNextInt())  
 throw new IncorrectGraphEntryException();  
 element = sc.nextInt();  
 if (!vertexList.contains(element))  
 throw new IncorrectGraphEntryException();  
 int secondIndex = vertexList.indexOf(element); // из "б" в "а"  
 sourceNetwork[firstIndex][secondIndex] = sourceNetwork[secondIndex][firstIndex] = 1; // наличие связи  
 // должны быть равны, так как граф невзвешенный (по этой же причине = 1) (  
 }  
 } catch (FileNotFoundException ex) {  
 throw new NoInputFileException();  
 }  
 }  
  
 // находим соседей конкретной вершинки  
 ArrayList<Integer> neighbours(int firstVertex) {  
 ArrayList<Integer> neighbours = new ArrayList<>();  
 for (int secondVertex = 0; secondVertex < numberOfStrings; secondVertex++)  
 if (sourceNetwork[firstVertex][secondVertex] != 0) // если есть связь  
 neighbours.add(secondVertex);  
 return neighbours;  
 }  
  
  
 // поиск в ширину  
 boolean bfs(int source, int drain, int[] parent) { // source - исток, drain - сток  
 boolean[] used = new boolean[numberOfStrings]; // для того, чтобы отмечать, пройдена ли вершинка  
 for (int firstVertex = 0; firstVertex < numberOfStrings; firstVertex++)  
 used[firstVertex] = false;  
 LinkedList<Integer> queue = new LinkedList<>(); // создали очередь вершин  
 queue.add(source); // вначале очередь состоит из одной вершины  
 used[source] = true; // отметили её как посещенную  
 parent[source] = -1; // без родителя  
 while (!queue.isEmpty()) {  
 int secondVertex = queue.poll(); // извлекли первую в очереди вершинку  
 for (int firstVertex = 0; firstVertex < numberOfStrings; firstVertex++) {  
 // теперь для всех дуг из вершины secondVertex, для которых firstVertex не посещена:  
 if (!used[firstVertex] && residualNetwork[secondVertex][firstVertex] > 0) {  
 queue.add(firstVertex); // добавили вершинку в конец очереди  
 used[firstVertex] = true; // отметили как посещенную  
 parent[firstVertex] = secondVertex; // родитель secondVertex - это firstVertex  
 }  
 }  
 }  
 return used[drain]; // если firstVertex == drain, то все, выходим из циклов, мы нашли кратчайший путь  
 }  
  
  
 // поиск в глубину (ищем кратчайший путь для разреза)  
 void dfs(int firstVertex, boolean[] used) {  
 used[firstVertex] = true; // отметили вершину как пройденную  
 for (int secondVertex = 0; secondVertex < numberOfStrings; secondVertex++)  
 if (residualNetwork[firstVertex][secondVertex] > 0 && !used[secondVertex]) // если вершина не помечена и смежна с текущей  
 dfs(secondVertex, used);  
 }  
  
 int EdmondsKarp(int source, int drain) {  
 // это мы создаем нашу остаточную сеть, которая изначально совпадает с исходной (с sourceNetwork)  
 residualNetwork = new int[numberOfStrings][numberOfStrings];  
 // обнуляем все потоки  
 for (int firstVertex = 0; firstVertex < numberOfStrings; firstVertex++)  
 System.*arraycopy*(sourceNetwork[firstVertex], 0, residualNetwork[firstVertex], 0, numberOfStrings);  
 int[] parent = new int[numberOfStrings];  
 int flow = 0;  
 // в остаточной сети находим кратчайший путь из источника в сток. Если такого пути нет, останавливаемся  
 while (bfs(source, drain, parent)) {  
 // пускаем максимально возможный поток через найденный путь и ищем ребро с min. пропускной способностью  
 int karpFlow = Integer.*MAX\_VALUE*;  
 for (int firstVertex = drain; firstVertex != source; firstVertex = parent[firstVertex]) {  
 int secondVertex = parent[firstVertex];  
 // в остаточной сети находим кратчайший путь из источника в сток  
 karpFlow = Math.*min*(karpFlow, residualNetwork[secondVertex][firstVertex]);  
 }  
 // Для каждого ребра на найденном пути увеличиваем поток на min, а в противоположном ему — уменьшаем на min  
 // min - это как раз-таки karpFlow  
 for (int firstVertex = drain; firstVertex != source; firstVertex = parent[firstVertex]) {  
 int secondVertex = parent[firstVertex];  
 residualNetwork[secondVertex][firstVertex] -= karpFlow;  
 residualNetwork[firstVertex][secondVertex] += karpFlow;  
 }  
 // Модифицируем остаточную сеть. Для всех рёбер на найденном пути, а также для противоположных им рёбер,  
 // вычисляем новую пропускную способность. Если она стала ненулевой, добавляем ребро к остаточной сети,  
 // а если обнулилась, стираем его (и так каждый раз)  
 flow += karpFlow;  
 }  
 return flow;  
 }  
  
 // минимальный разрез, он же максимальное значение потока  
 // булевская чтобы определить, каким цветом закрашивать  
 boolean[][] minCut(int source) {  
 boolean[][] mincut = new boolean[numberOfStrings][numberOfStrings];  
 for (int firstVertex = 0; firstVertex < numberOfStrings; firstVertex++)  
 for (int secondVertex = 0; secondVertex < numberOfStrings; secondVertex++)  
 mincut[firstVertex][secondVertex] = false;  
 boolean[] used = new boolean[numberOfStrings];  
 dfs(source, used);  
 for (int firstVertex = 0; firstVertex < numberOfStrings; firstVertex++)  
 for (int secondVertex : neighbours(firstVertex))  
 if (sourceNetwork[firstVertex][secondVertex] > 0 && used[firstVertex] && !used[secondVertex])  
 mincut[firstVertex][secondVertex] = mincut[secondVertex][firstVertex] = true;  
 return mincut; // список ребер в минимальном разрезе  
 }  
  
 /\* !рёберная связность графа равна минимуму от наименьшего числа рёбер, разделяющих две вершины source и drain,  
 взятому среди всевозможных пар! \*/  
 boolean[][] edgeConnectivity() {  
 int answer = Integer.*MAX\_VALUE*;  
 for (int source = 0; source < numberOfStrings; source++)  
 for (int drain = source + 1; drain < numberOfStrings; drain++) {  
 int flow = EdmondsKarp(source, drain);  
 answer = Math.*min*(answer, flow);  
 }  
 for (int source = 0; source < numberOfStrings; source++)  
 for (int drain = source + 1; drain < numberOfStrings; drain++) {  
 int flow = EdmondsKarp(source, drain); // т.е величина максимального потока из истока в сток)  
 if (flow == answer)  
 return minCut(source);  
 }  
 return null;  
 }  
}

## **GraphException.java**

class GraphException extends Exception {  
 GraphException(String message) {  
 super(message);  
 }  
}

## **IncorrectGraphEntryException.java**

class IncorrectGraphEntryException extends GraphException {  
 IncorrectGraphEntryException() {  
 super("Неверный ввод графа");  
 }  
}

## **InputFileIsEmptyException.java**

class InputFileIsEmptyException extends GraphException {  
 InputFileIsEmptyException() {  
 super("Файл ввода пуст");  
 }  
}

## **NoInputFileException.java**

class NoInputFileException extends GraphException {  
 NoInputFileException() {  
 super("Нет файла ввода");  
 }  
}

# Тестирование

Класс GraphTest.kt тестирует класс Graph и его метод minCut() (на котлине дабы тестирование было компактнее)

import org.junit.jupiter.api.Test;  
import java.io.File  
  
class GraphTest {  
 // реберная связность равна 2  
 private val file\_1 = File("example\_5.txt")  
 // реберная связность равна нулю  
 private val file\_2 = File("zeroConnectivity.txt")  
 // реберная связность равна одному (т.е задан граф с мостом)  
 private val file\_3 = File("example.txt")  
  
 private val graph\_1 = Graph(file\_1)  
 private val graph\_2 = Graph(file\_2)  
 private val graph\_3 = Graph(file\_3);  
  
 private val mincut\_1 = Array(graph\_1.numberOfStrings) **{**BooleanArray(graph\_1.numberOfStrings)**}** private val mincut\_2 = Array(graph\_2.numberOfStrings) **{**BooleanArray(graph\_2.numberOfStrings)**}** private val mincut\_3 = Array(graph\_3.numberOfStrings) **{**BooleanArray(graph\_3.numberOfStrings)**}** private var valcut\_1 = Array(graph\_1.numberOfStrings) **{**BooleanArray(graph\_1.numberOfStrings)**}** private var valcut\_2 = Array(graph\_2.numberOfStrings) **{**BooleanArray(graph\_2.numberOfStrings)**}** private var valcut\_3 = Array(graph\_3.numberOfStrings) **{**BooleanArray(graph\_3.numberOfStrings)**}** @Test  
 fun test1(){  
 mincut\_1[0][2] = true  
 mincut\_1[2][0] = true  
 mincut\_1[0][1] = true  
 mincut\_1[1][0] = true  
  
 graph\_1.EdmondsKarp(0, 3)  
 valcut\_1 = graph\_1.minCut(0)  
 if (mincut\_1[0][2] == graph\_1.minCut(0)[0][2] && mincut\_1[2][0] == graph\_1.minCut(0)[2][0] &&  
 mincut\_1[0][1] == graph\_1.minCut(0)[0][1])  
 *print*("ok")  
 }  
  
 @Test  
 fun test2(){  
 graph\_2.EdmondsKarp(0, 4)  
 valcut\_2 = graph\_2.minCut(0)  
 for(i in mincut\_2.*indices*) {  
 for (j in mincut\_2[i].*indices*) {  
 if (mincut\_2[i][j] == graph\_2.minCut(0)[i][j])  
 *print*("ok")  
 else {  
 *print*("none")  
 break  
 }  
 }  
 }  
 }  
  
 @Test  
 fun test3(){  
 mincut\_3[2][4] = true  
 mincut\_3[4][2] = true  
 graph\_3.EdmondsKarp(0,4)  
 valcut\_3 = graph\_3.minCut(0)  
 if (mincut\_3[2][4] == graph\_3.minCut(0)[2][4] && mincut\_3[4][2] == graph\_3.minCut(0)[4][2])  
 *print*("ok")  
 }  
}

# Скриншоты программы:

1. Внешний вид, меню:

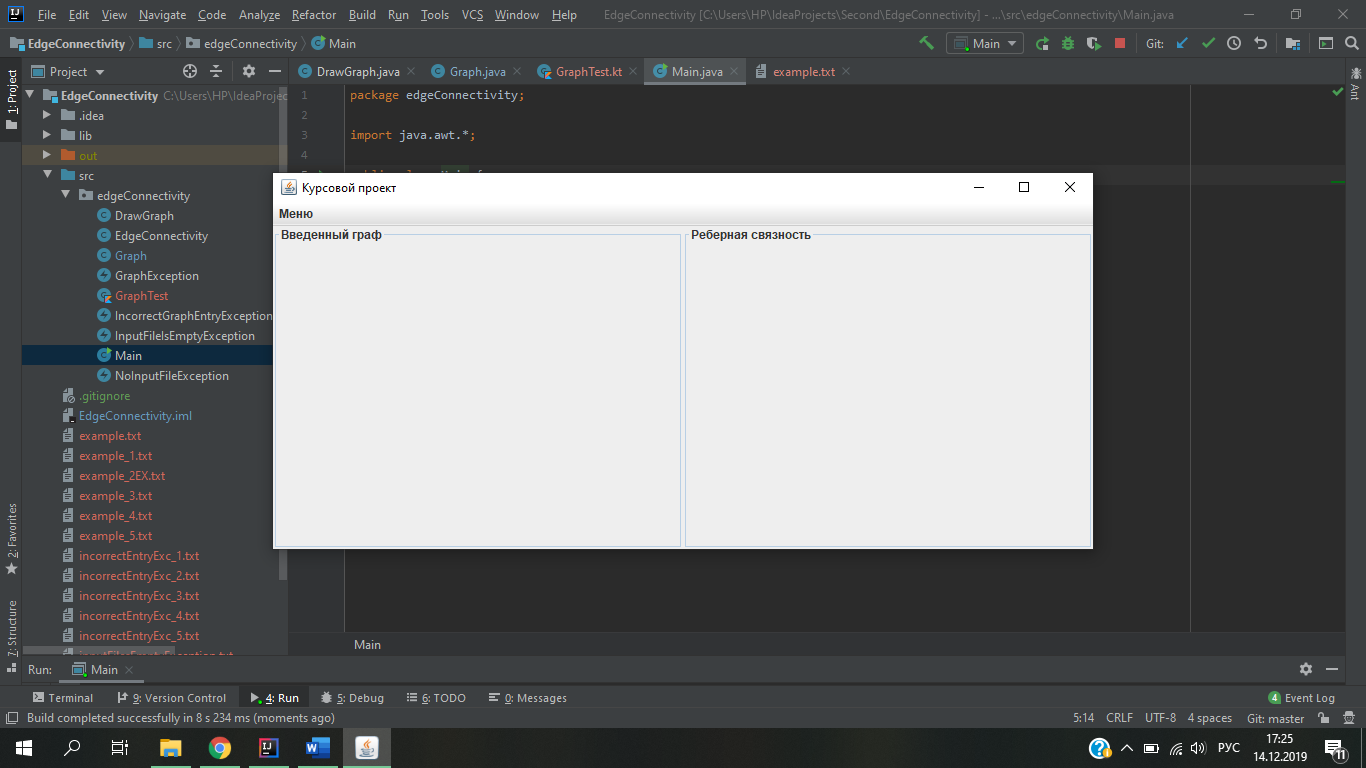
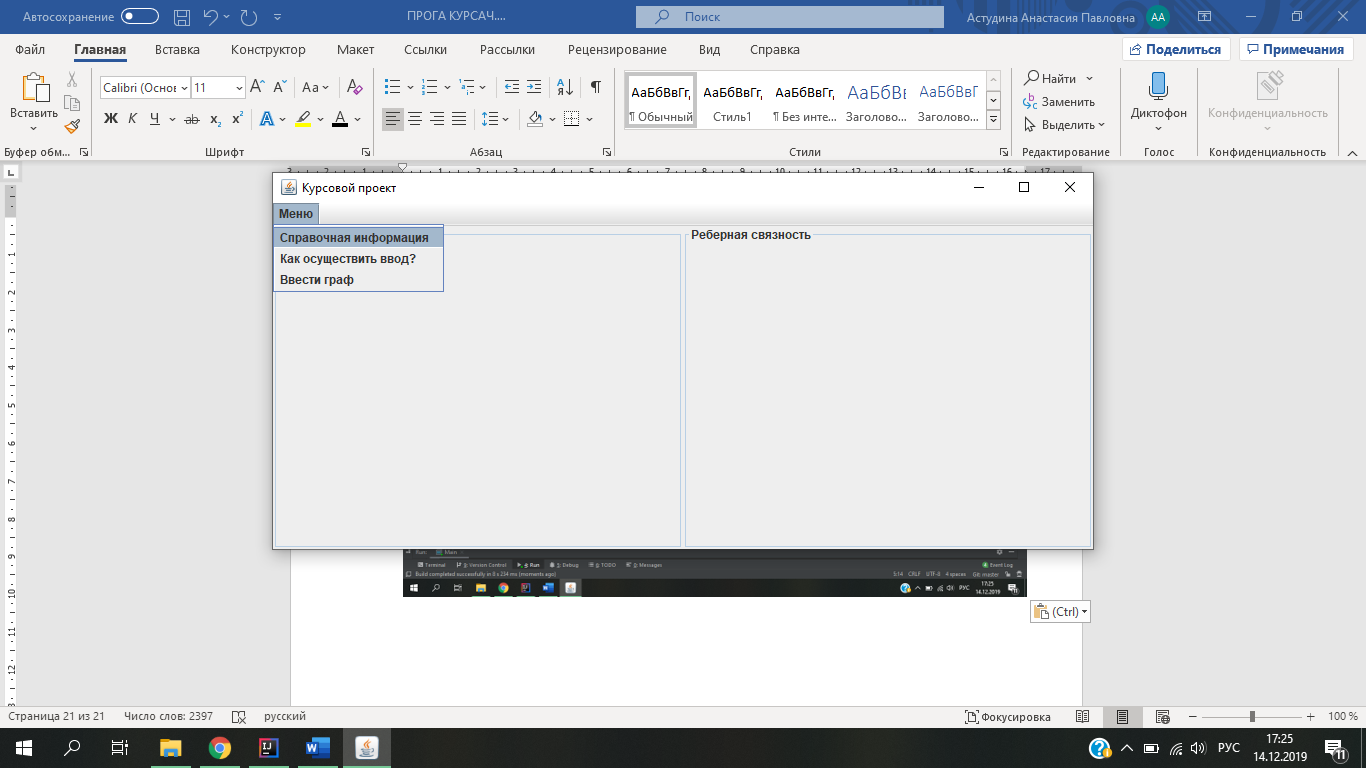
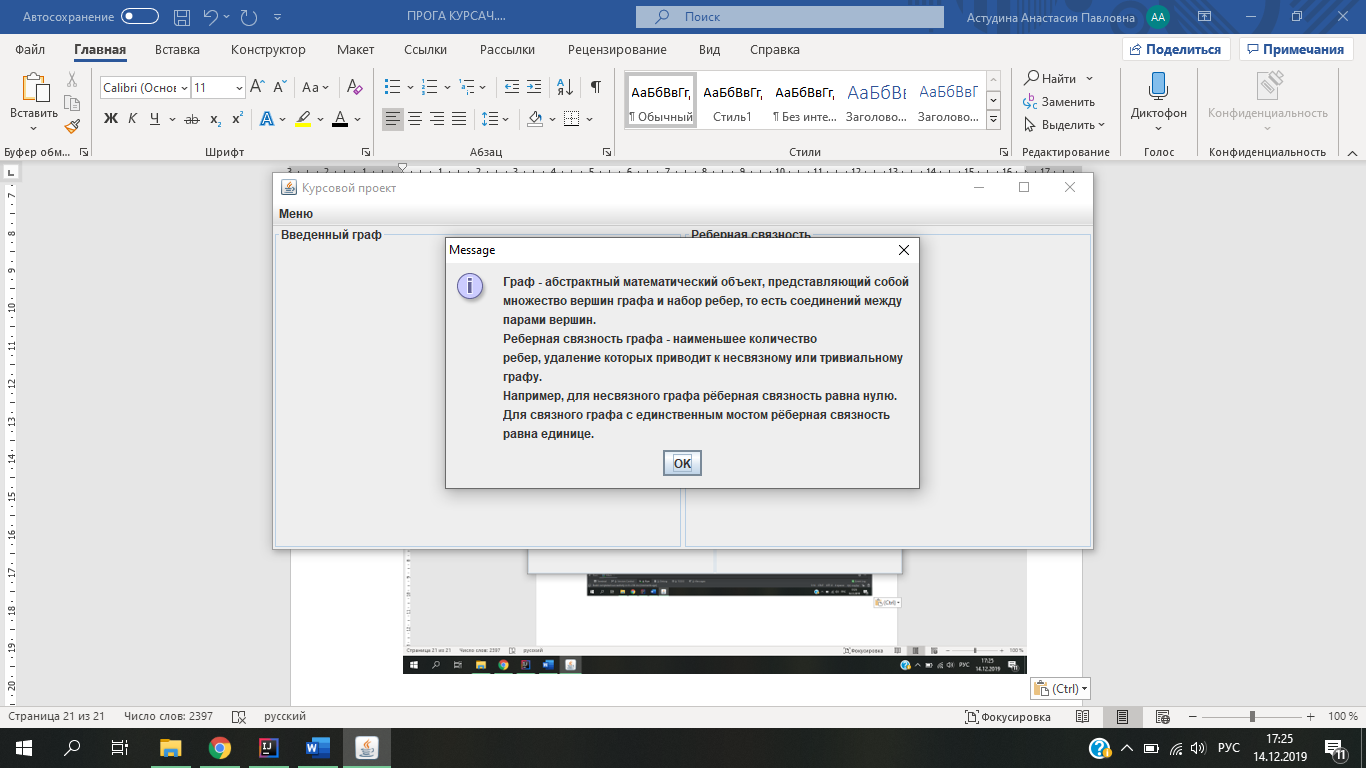
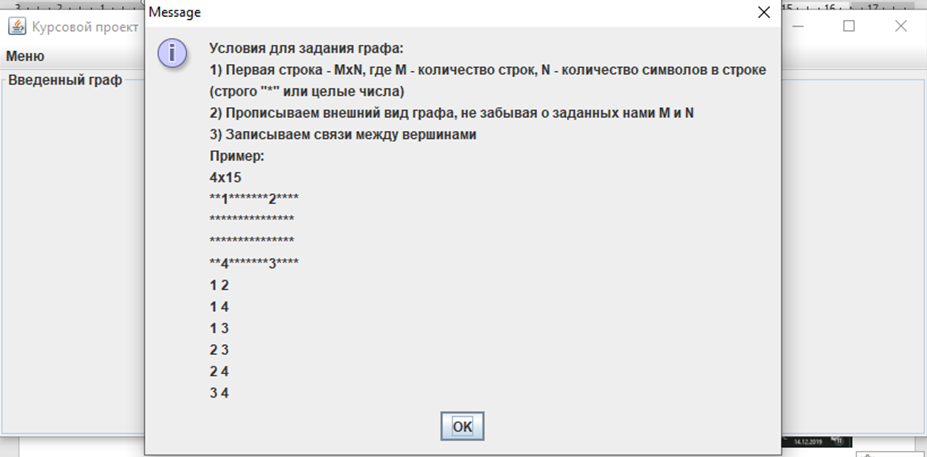
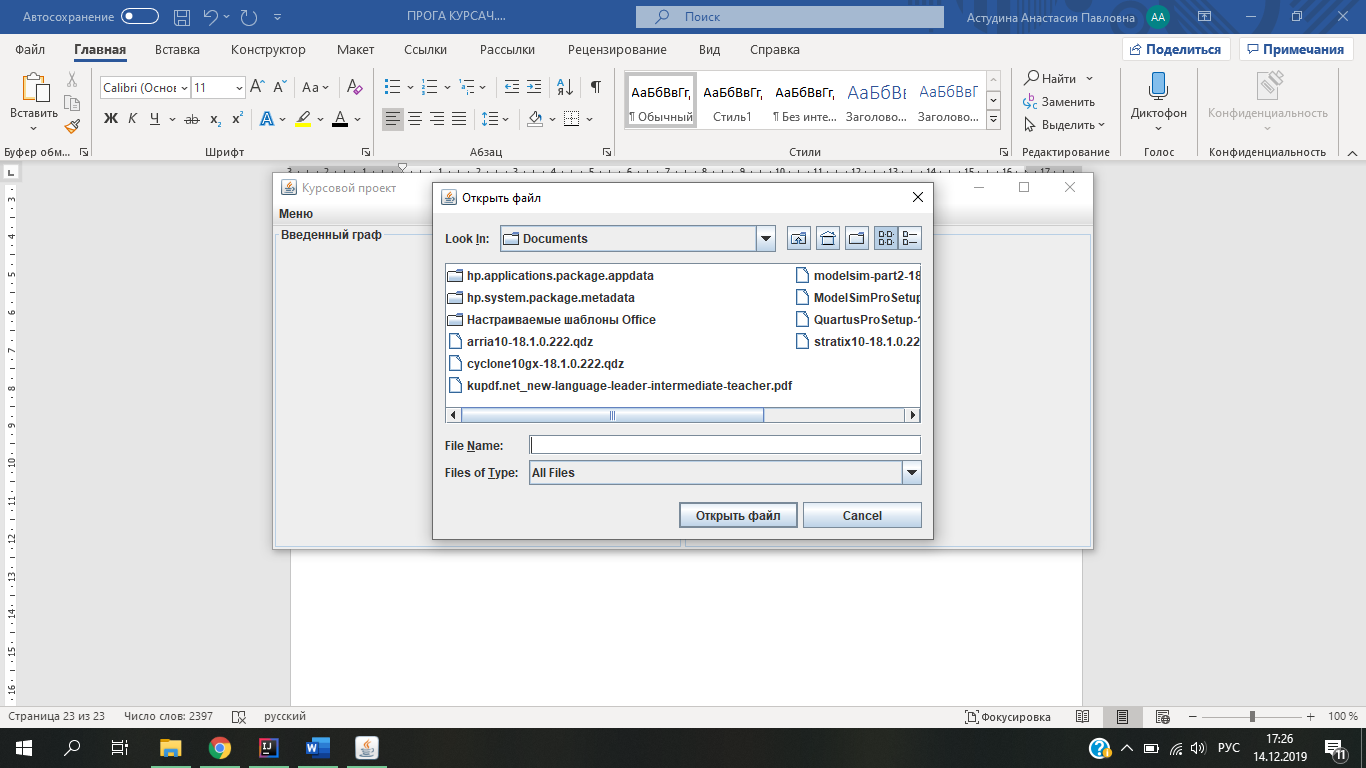
Рис. 1.1: Начальное окно

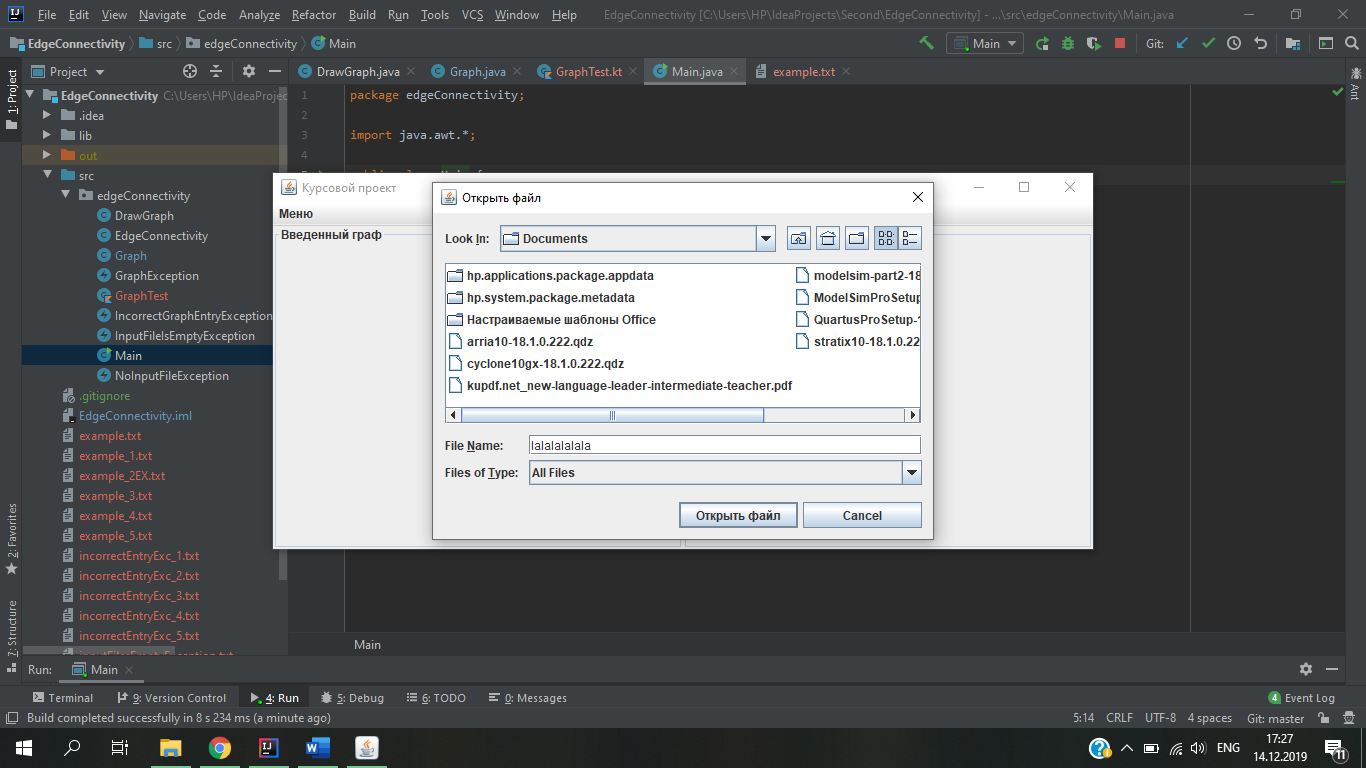
Рис. 1.2: Меню программы

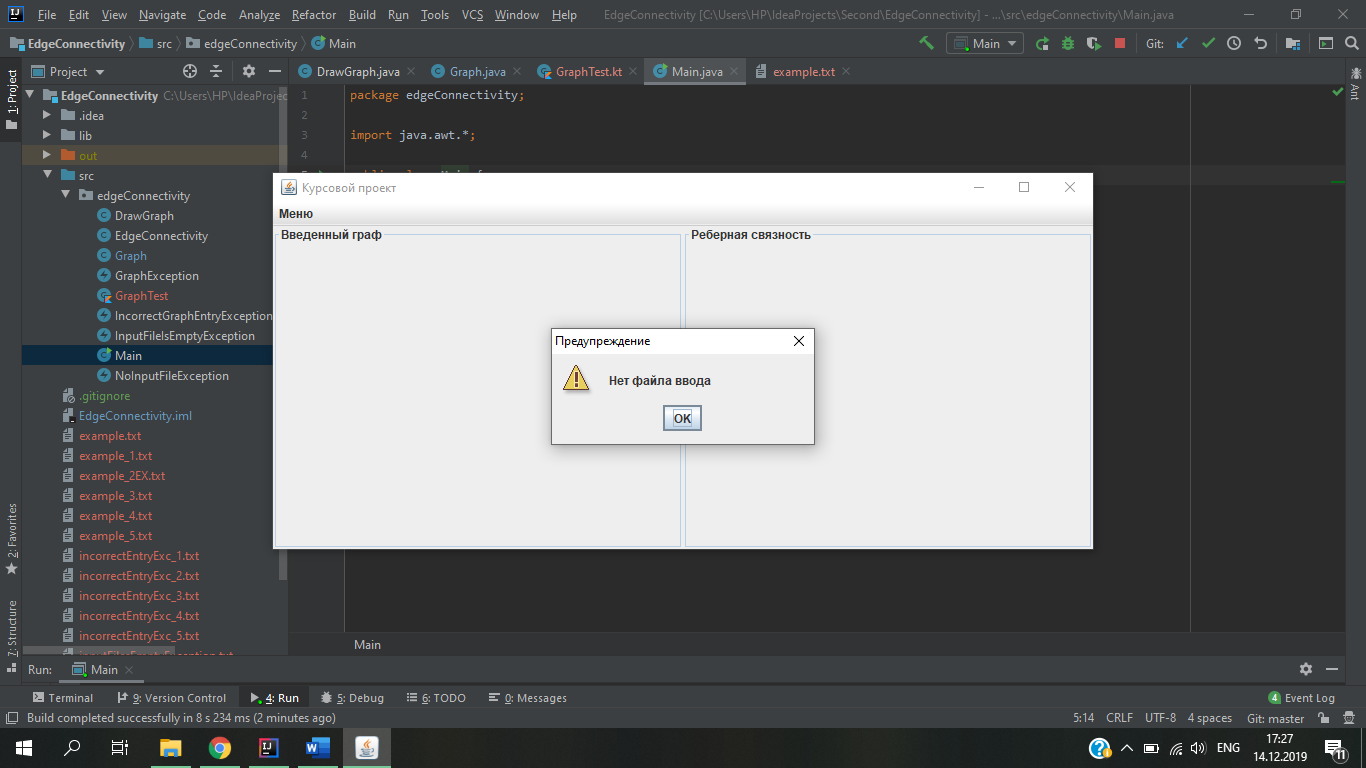
  
Рис 1.3: Справочная информация

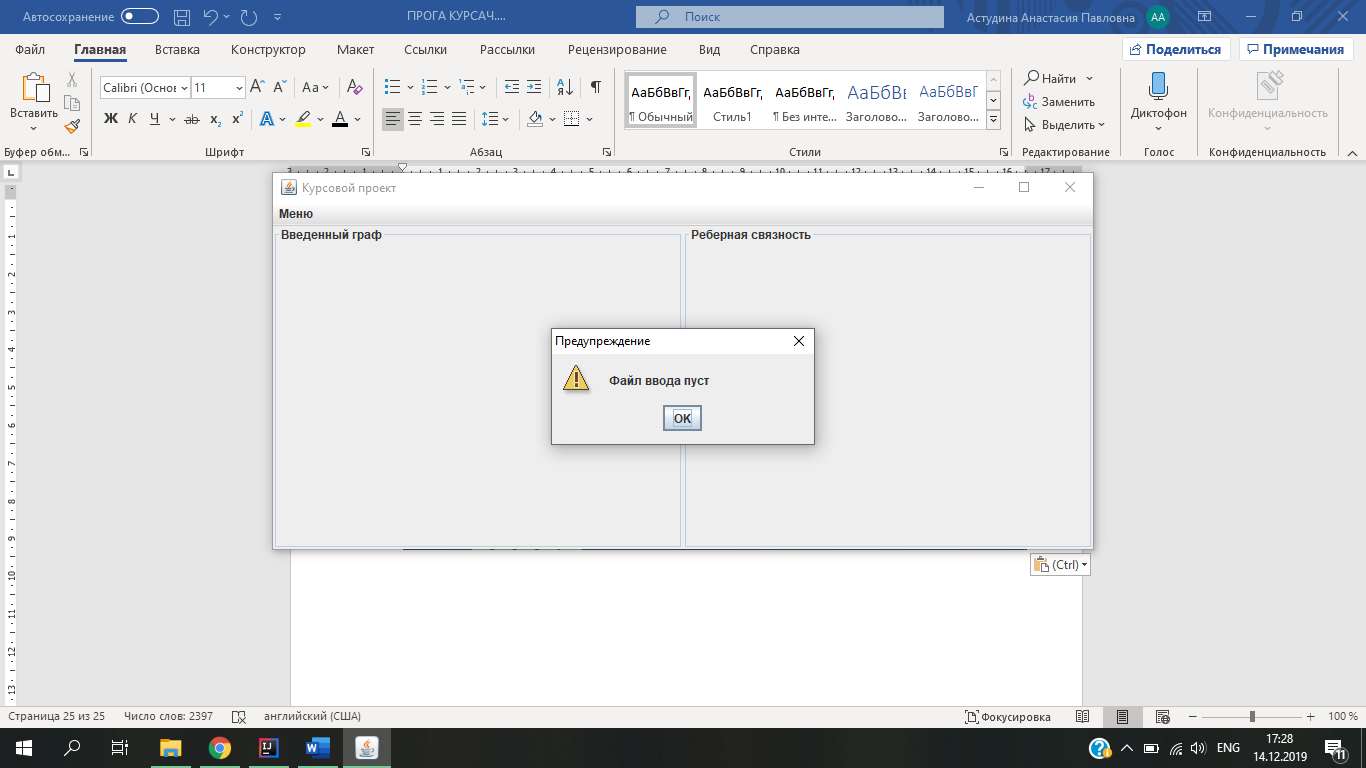
  
Рис. 1.4: Описание способа задания графа

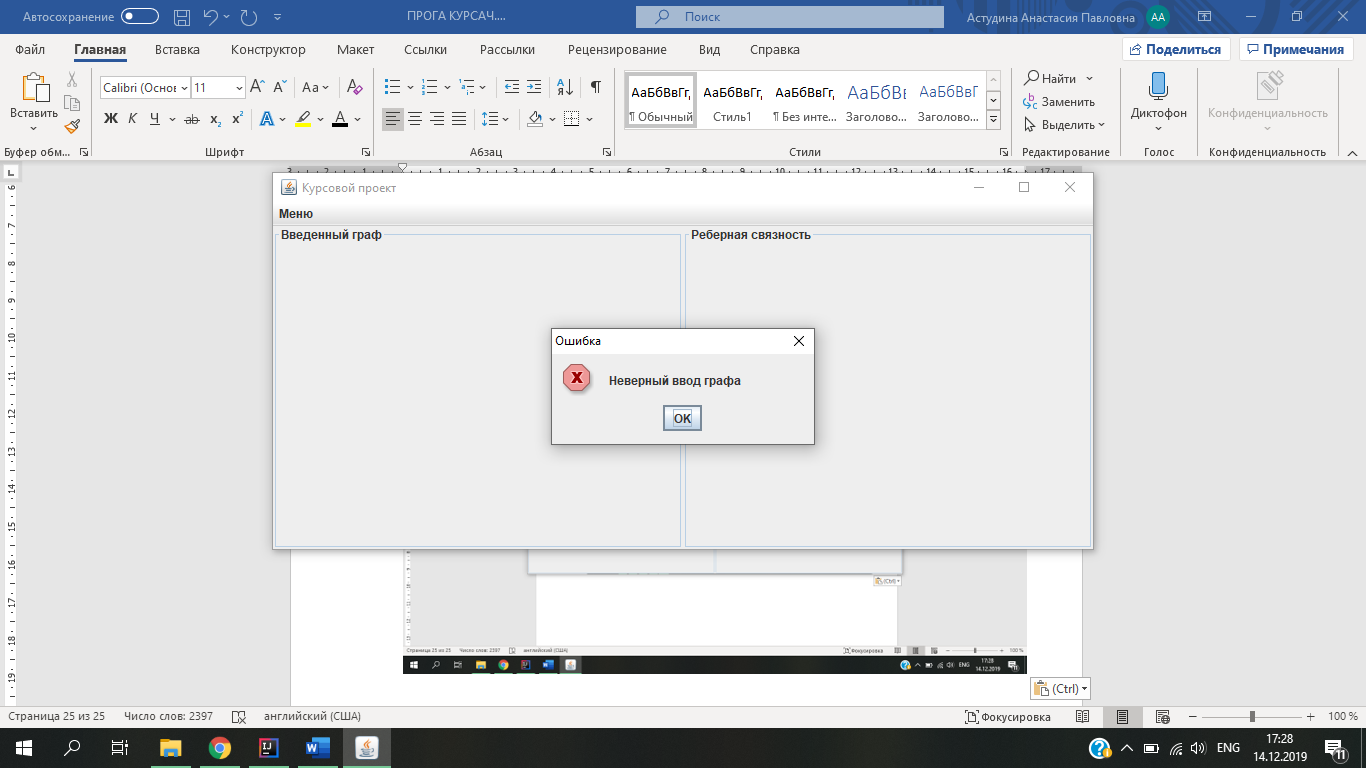
  
Рис. 1.5: Выбор файла ввода

1. Обработка ошибок ввода:

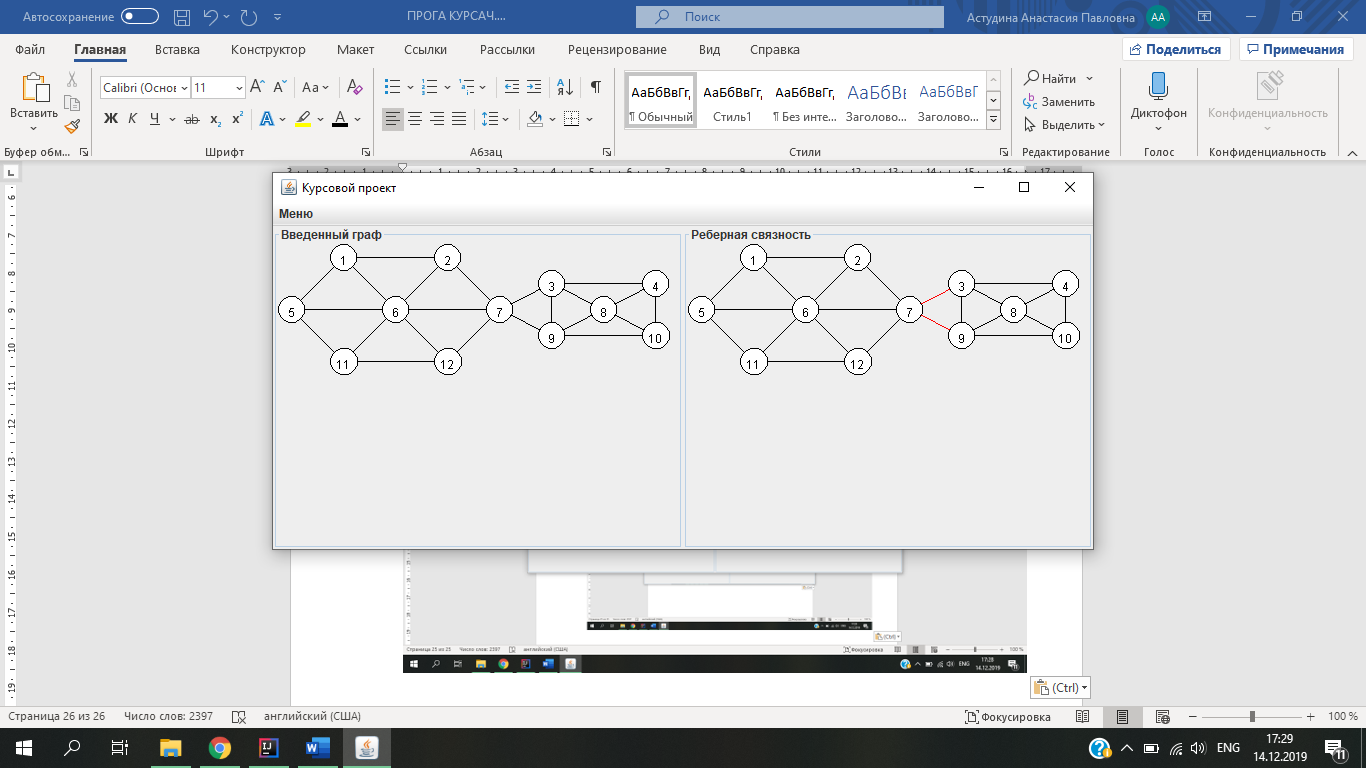
  
Рис. 2.1: Введем имя несуществующего файла

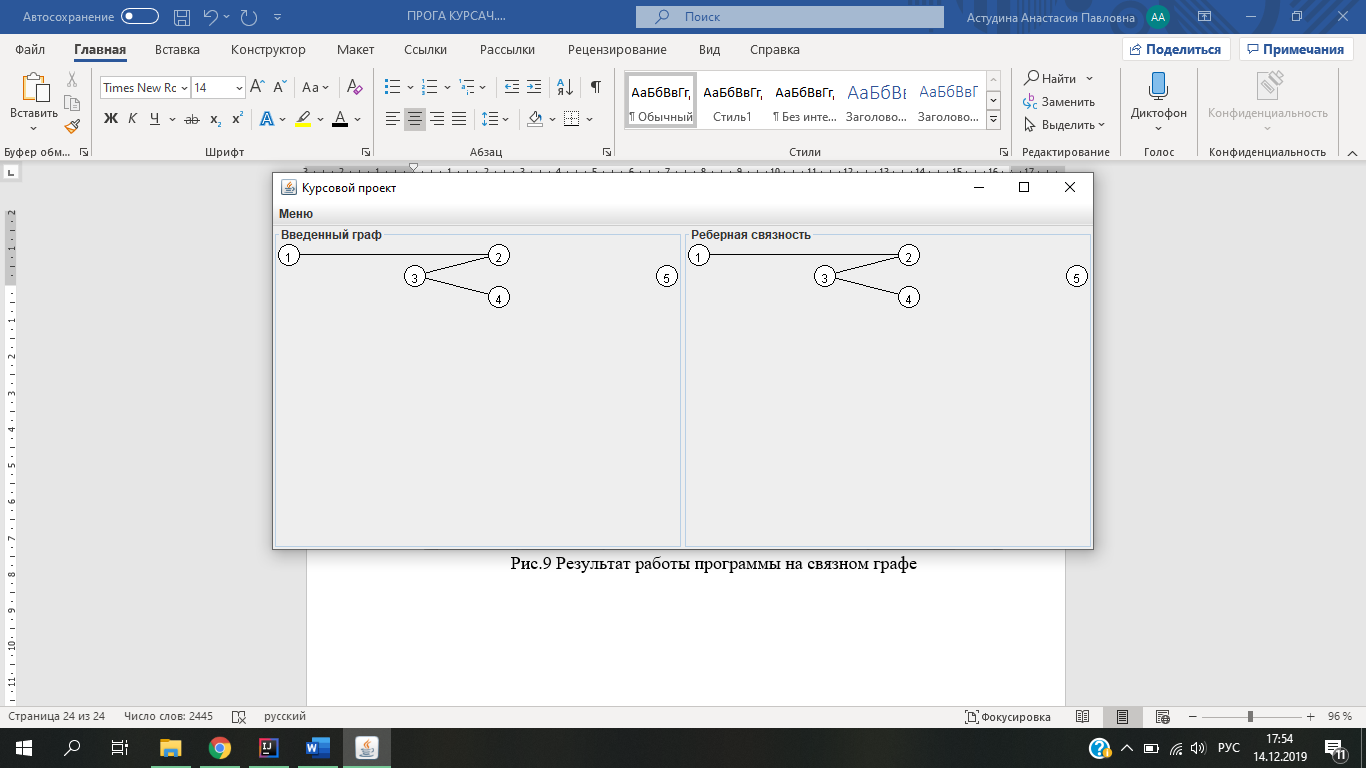
  
Рис. 2.2: Обработка NoInputFileException

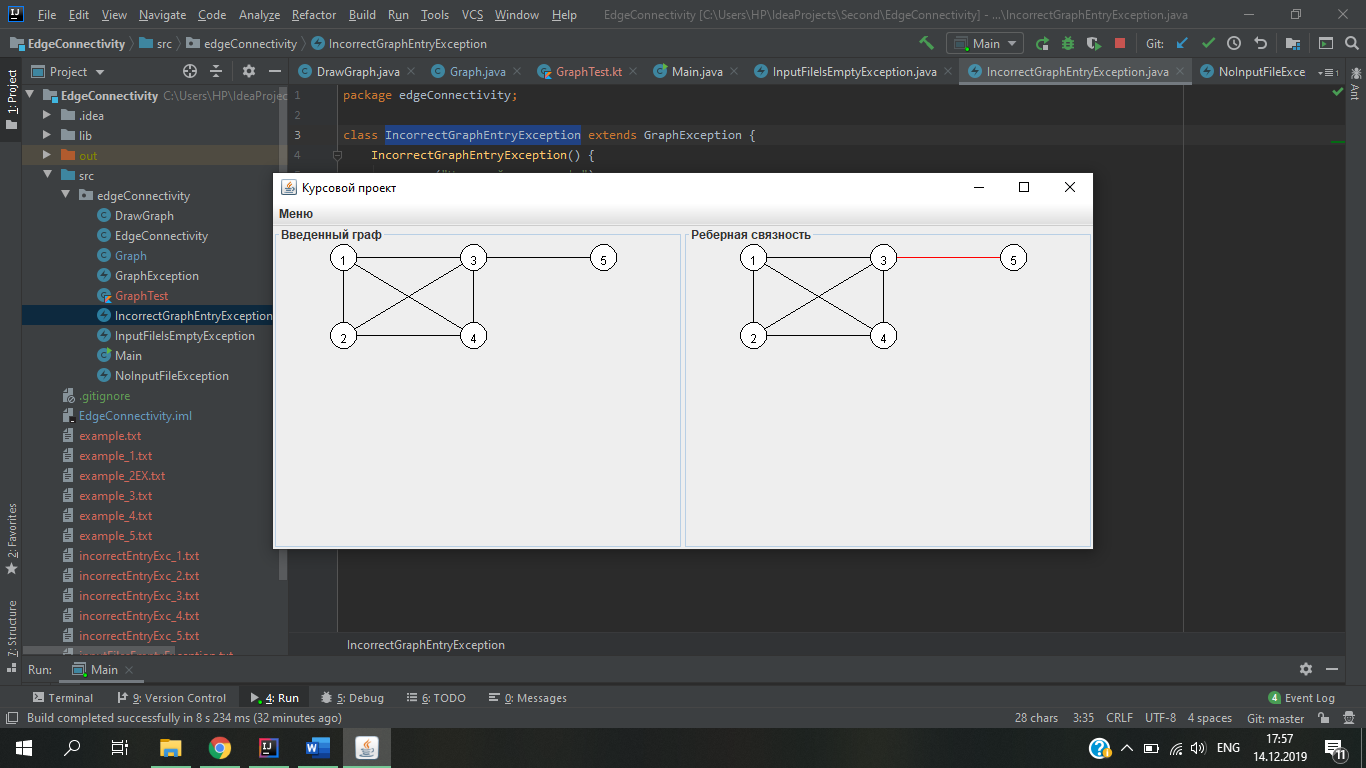
  
Рис. 2.3: Обработка InputFileIsEmptyException

  
Рис. 2.4: Обработка IncorrectGraphEntryException

1. Работа приложения в случае корректно заданного файла ввода:

  
Рис.3.1: Результат работы программы на связном графе (example\_2EX.txt)

  
Рис.3.2: Работа программы на несвязном графе, т.е. реберная связность равна нулю (zeroConnectivity.txt)

  
Рис. 3.3: Граф с мостом