

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

ОТЧЕТ
по учебной практике
Тема: Визуализация алгоритмов на графах на языке Java

Студент гр. 7303	_____	Ковалёв К.А.
Студент гр. 7303	_____	Романенко М.В.
Студент гр. 7303	_____	Державин Д.П.
Руководитель	_____	Размочаева Н.В.

Санкт-Петербург
2019

ЗАДАНИЕ НА УЧЕБНУЮ ПРАКТИКУ

Студент Ковалёв К.А. группы 7303

Студент Романенко М.В. группы 7303

Студент Державин Д.П. группы 7303

Тема практики: Визуализация алгоритмов на графах на языке Java

Задание на практику:

Командная итеративная разработка визуализатора алгоритма на языке Java с графическим интерфейсом.

Алгоритм: Косарайю

Сроки прохождения практики: 01.07.2019 – 14.07.2019

Дата сдачи отчета: 10.07.2019

Дата защиты отчета: 10.07.2019

Студент	_____	Ковалёв К.А.
Студент	_____	Романенко М.В.
Студент	_____	Державин Д.П.
Руководитель	_____	Размочасева Н.В.

АННОТАЦИЯ

В ходе выполнения задания учебной практики была реализована программа, осуществляющая поиск компонент сильной связности в ориентированном графе с помощью алгоритма Косарайю. Программа разработана в среде IntelliJ IDEA. Язык, используемый для написания программы – Java. В проекте используется Swing — библиотека для создания графического интерфейса для программ на языке Java [5]. Разработанная программа детально показывает этапы работы алгоритма.

SUMMARY

During the educational practice there has been created the program that searches strongly connected components using Kosaraju's algorithm. The program has been developed in IDE IntelliJ IDEA. The language of program is Java. There has been used Swing in a project. It is a library for writing GUI in Java. The developed program shows in detail the stages of the algorithm Kosaraju.

СОДЕРЖАНИЕ

Введение	5
1. Постановка задачи	6
1.1. Теоретические сведения	6
1.2. Реализуемый алгоритм	6
2. Спецификация программы	8
2.1. Требования к входным данным программы	8
2.2. Описание интерфейса	8
3. План разработки и распределение ролей в бригаде	11
3.1. План разработки	11
3.2. Распределение ролей в бригаде	11
4. Особенности реализации	12
4.1. Архитектура программы	12
5. Процесс тестирования	13
5.1. Тестирование графического интерфейса	13
5.2. Тестирование кода алгоритма	18
Заключение	24
Список используемых источников	25
Приложение А. Исходный код – только в электронном виде	26

ВВЕДЕНИЕ

Целью данной учебной практики является развитие следующих навыков:

- программирование на языке Java;
- работа над проектом в команде;
- использование системы контроля версий.

Таким образом, данный отчет посвящен выполнению учебного проекта, выданного в соответствии с указанными требованиями. Основой для изучения языка Java, стал курс на образовательной платформе Stepik[1], и руководство Гербера Шилдта[5].

Формулировка задания: требуется разработать программу на языке Java, визуализирующую алгоритм поиска компонент сильной связанности в ориентированном графе.

1. ПОСТАНОВКА ЗАДАЧИ

1.2. Теоретические сведения

Дан ориентированный или неориентированный граф. Граф называется сильно связным, если любые две его вершины сильно связаны. Две вершины s и t любого графа сильно связны, если существует ориентированный путь из s в t и ориентированный путь из t в s . Компонентами сильной связности орграфа называются его максимальные по включению сильно связные подграфы.

Поиск в глубину (англ. depth-first search, DFS) – это рекурсивный алгоритм обхода вершин графа. Алгоритм поиска: перебираем все исходящие из рассматриваемой вершины рёбра. Если ребро ведёт в вершину, которая не была рассмотрена ранее, то запускаем алгоритм от этой нерассмотренной вершины, а после возвращаемся и продолжаем перебирать рёбра. Возврат происходит в том случае, если в рассматриваемой вершине не осталось рёбер, которые ведут в нерассмотренную вершину. Если после завершения алгоритма не все вершины были рассмотрены, то необходимо запустить алгоритм от одной из нерассмотренных вершин.

1.2. Реализуемый алгоритм

Алгоритм Косарайю содержит три этапа: поиск в глубину с фиксированием времени выхода вершин, транспонирование рёбер графа и непосредственно сам поиск компонент связности [4].

- *Первый этап.* Поиск в глубину с фиксированием времени выхода вершин. Обозначить все вершины графа не посещёнными и завести два стека: для DFS и списка времени выхода вершин графа. Применить к графу поиск в глубину. Если из текущей вершины нельзя попасть в новую, положить её в список времени выхода вершин графа. Если стек для DFS пуст, проверить, существует ли не посещённая вершина. Существует - положить её в стек для

DFS и применить к графу действия первого этапа, иначе перейти к следующему этапу.

- *Второй этап.* Транспонирование рёбер графа. Необходимо сменить ориентацию каждого ребра графа.

- *Третий этап.* Нахождение компонент сильной связности. Завести стек для DFS. Пока список времени выхода вершин графа не пуст, снять вершину со списка времени выхода вершин графа и поиском в глубину искать очередную вершину и добавлять в компоненту сильной связности. Если стек для DFS пуст, снять со списка времени выхода вершин графа вершину и начать искать новую компоненту сильной связности.

2. СПЕЦИФИКАЦИЯ ПРОГРАММЫ

2.1. Требования к входным данным программы

Доступно 2 способа ввода:

- Вручную. Пользуясь кнопками панели инструментов и/или Меню → Действие можно построить граф вручную.
- Ввод из файла. Пункт меню Файл → Сохранить, либо по комбинации клавиш **ctrl+O**. Граф хранится в виде списка инцидентности. Каждый список содержит вершину, её координаты, если они даны, и связанные с этой вершиной вершины.

2.2. Описание интерфейса

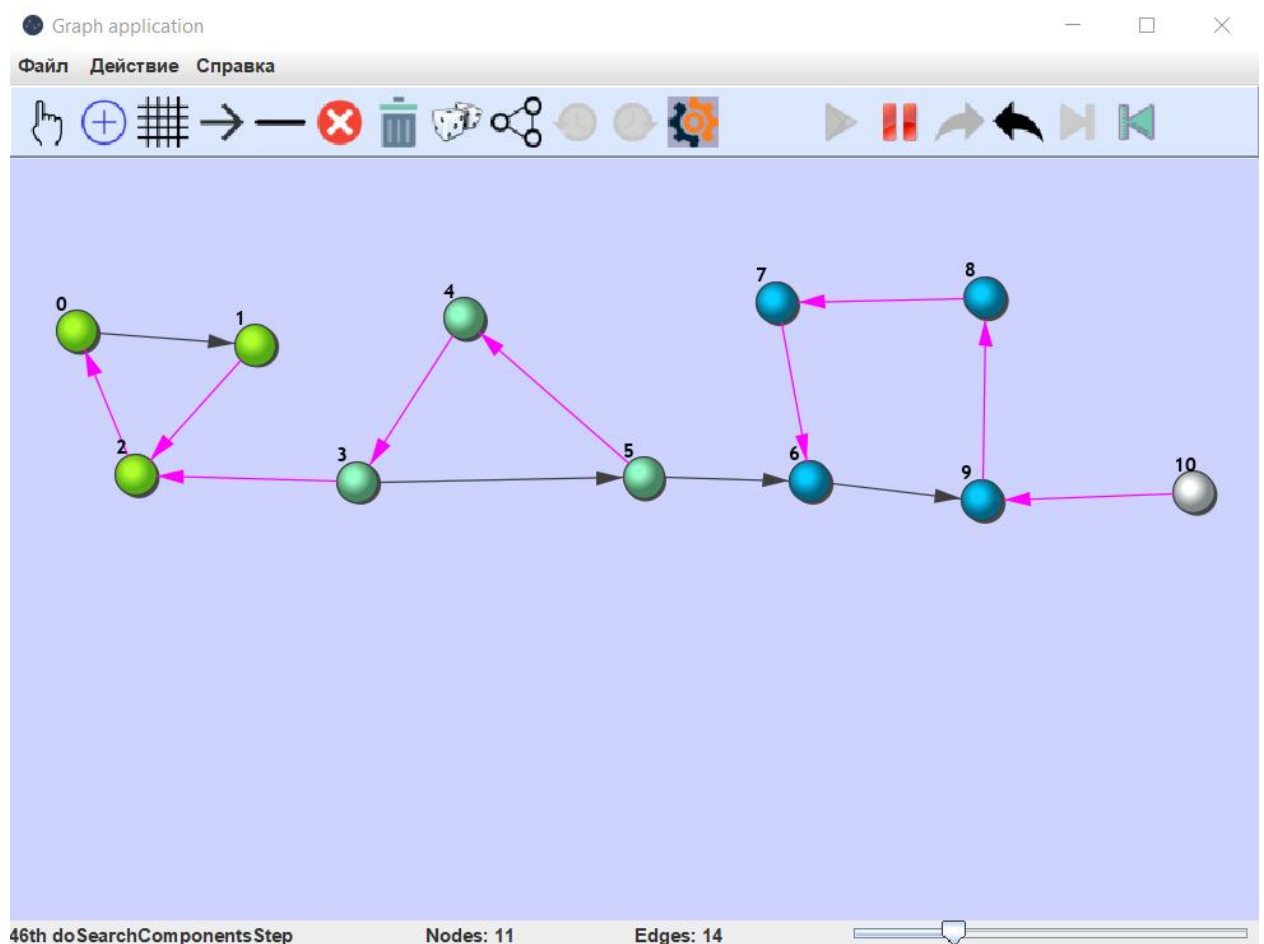




















Рисунок 1. Схема графического интерфейса пользователя

В нижней части окна находится статус бар с текстовым отображением выполняемого в данный момент действия, количеством вершин и ребер текущего графа и слайдер для изменения скорости работы алгоритма.

На панели инструментов сверху располагаются кнопки для создания и редактирования графа, а также кнопки для управления состоянием выполнения алгоритма.

1.  – перемещение вершин мышью.
2.  – добавить вершину;
3.  – соединить все вершины неориентированными ребрами;
4.  – добавить ориентированное ребро;
5.  – добавить неориентированное ребро;
6.  – удалить ребро/рёбра и вершины;
7.  – очистить графическую панель;
8.  – сгенерировать случайный граф;
9.  – создать граф в виде триангуляционной сети;
10.  – отменить последнее совершенное действие, также доступно по комбинации клавиш `ctrl+Z`;
11.  – вернуть последнее отмененное действие, также доступно по комбинации клавиш `ctrl+Y`;
12.  – начать выполнение выбранного алгоритма с выбора вершины;
13.  – запустить выполнение алгоритма;

14.  – приостановить выполнение алгоритма;
15.  – шаг вперед для выполняемого алгоритма;
16.  – шаг назад для выполняемого алгоритма;
17.  – переместиться к концу выполнения алгоритма;
18.  – переместиться к началу выполнения алгоритма;

Также для пользователя реализованы следующие возможности взаимодействия с программой:

- Перемещение сцены мышью с зажатым колёсиком;
- Изменение масштаба сцены по кручению колёсика мыши;
- Регулирование скорости работы алгоритма;
- Сохранение графа в файл. Пункт меню Файл → Сохранить, либо по комбинации клавиш ctrl+S;

3. РАЗРАБОТКИ И РЕСПРЕДЕЛЕНИЕ РОЛЕЙ В БРИГАДЕ

3.1. План разработки

Таблица 1 – план разработки

№	Задача	Планируемый срок сдачи	Срок сдачи
1	Добавить спецификацию	02.07.2019	02.07.2019
2	Представить прототип	02.07.2019	02.07.2019
3	Представить 1 версию	08.07.2019	05.07.2019
4	Представить 2 версию	10.07.2019	08.07.2019
5	Представить финальную версию	12.07.2019	10.07.2019

3.2. Распределение ролей в бригаде

Державин Д.П. Реализация пользовательского интерфейса и алгоритма Косарайю на языке Java, тестирование программы.

Ковалёв К.А. Разработка и реализация архитектуры хранения данных, визуализация алгоритма.

Романенко М.В. Разработка и реализация ввода графа из файла и его сохранение в файл, дизайн пользовательского интерфейса.

4. ОСОБЕННОСТИ РЕАЛИЗАЦИИ

4.1. Архитектура программы

Как говорил мой дед, «Разделяй и властвуй». Именно поэтому программа спроектирована таким образом, чтобы классы обработчики программы не имели доступа визуальной составляющей (Но не везде). Классы программы связывались таким образом, чтобы каждый из них занимался исключительно своей задачей и передавал информацию о своем состоянии другим. Данная архитектура позволила свободно добавлять в код новые классы для визуализации алгоритмов.

Для отображения графа на графическую панель был создан класс DrawGraph, который получает информацию о вершинах и ребрах графа, берет из этих данных классы «обертки», и передает им управление на отрисовку всех составляющих графа.

Класс Graph хранит в себе список вершин и список ребер. Данные списки необходимы для предоставления всех компонент визуализатору.

Для быстрого доступа к ребрам, которые связаны с конкретной вершиной, класс Node содержит в себе список экземпляров класса Edge, который в свою очередь хранит в себе пару экземпляров Node. Благодаря такой архитектуре хранения данных графа можно быстро итерировать по любым вершинам и ребрам графа.

Для того, чтобы граф не занимался обработкой команд, поступающих от GUI программы, был разработан класс GraphEventManager. Данный класс принимает информацию от кнопок графической панели, а также события нажатия клавиши и обрабатывает их в зависимости от текущего состояния графа(режим добавления вершины, удаления ребра и т.д.). Класс GraphEventManager, после обработки полученных данных, передает графу команды на добавление, удаления вершин и\или ребер.

Гвоздем данной программы являются классы алгоритмов. Созданный абстрактный класс Algorithm содержит основные методы для работы любого алгоритма. Для добавления нового (в последний раз пишу это слово)

алгоритма создается наследник класса `Algorithm`, где переопределяются методы базового класса.

Для хранения экземпляров алгоритмов был добавлен отдельный класс со статическим списком доступных для визуализации алгоритмов.

Менеджеров много не бывает, поэтому в проект был добавлен класс `AlgorithmEventManager`, который, как и его предшественник, получает данные от GUI, обрабатывает их и передает результат текущему алгоритму, которые тот исполняет и возвращает свой статус как ответ. По результатам работы алгоритма обновляются элементы GUI и перерисовывается граф.

5. ПРОЦЕСС ТЕСТИРОВАНИЯ

5.1. Тестирование графического интерфейса

Запускаем программный модуль.

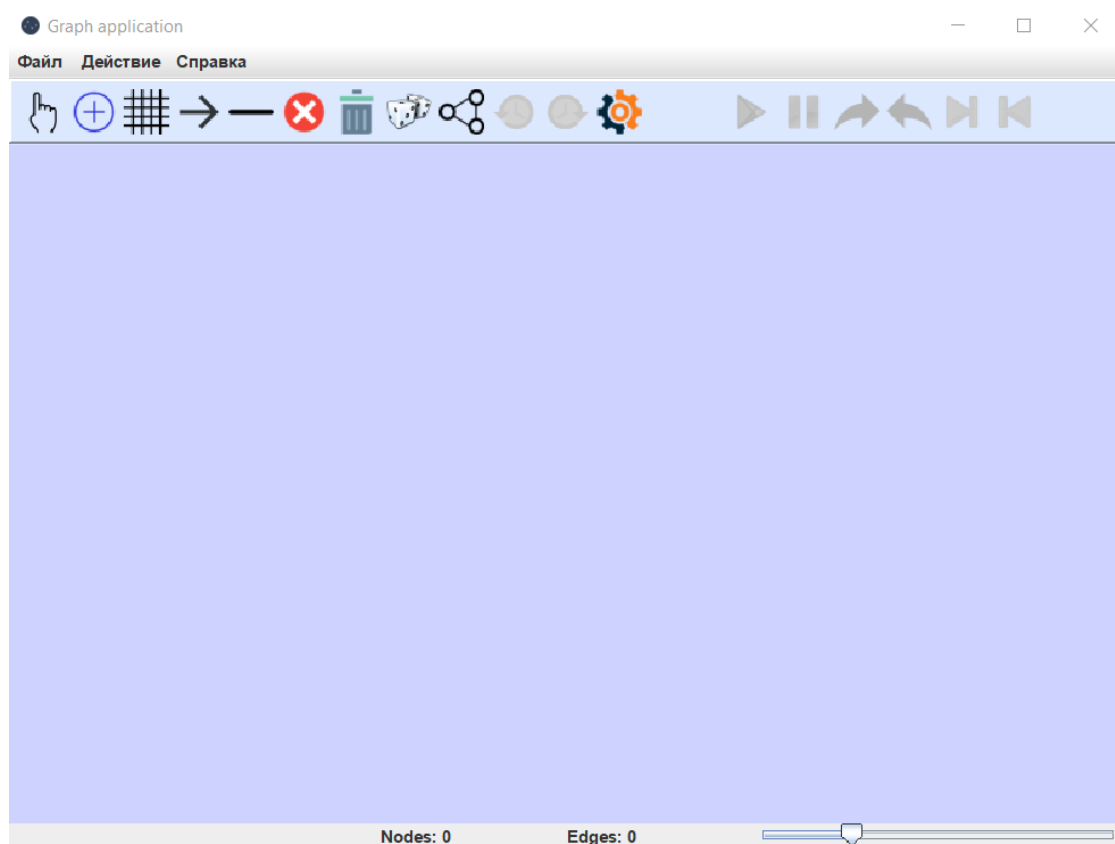


Рисунок 2 – Внешний вид интерфейса программы после запуска

С помощью панели инструментов добавим немного вершин, а после соединим их всех друг с другом при помощи соответствующей кнопки.

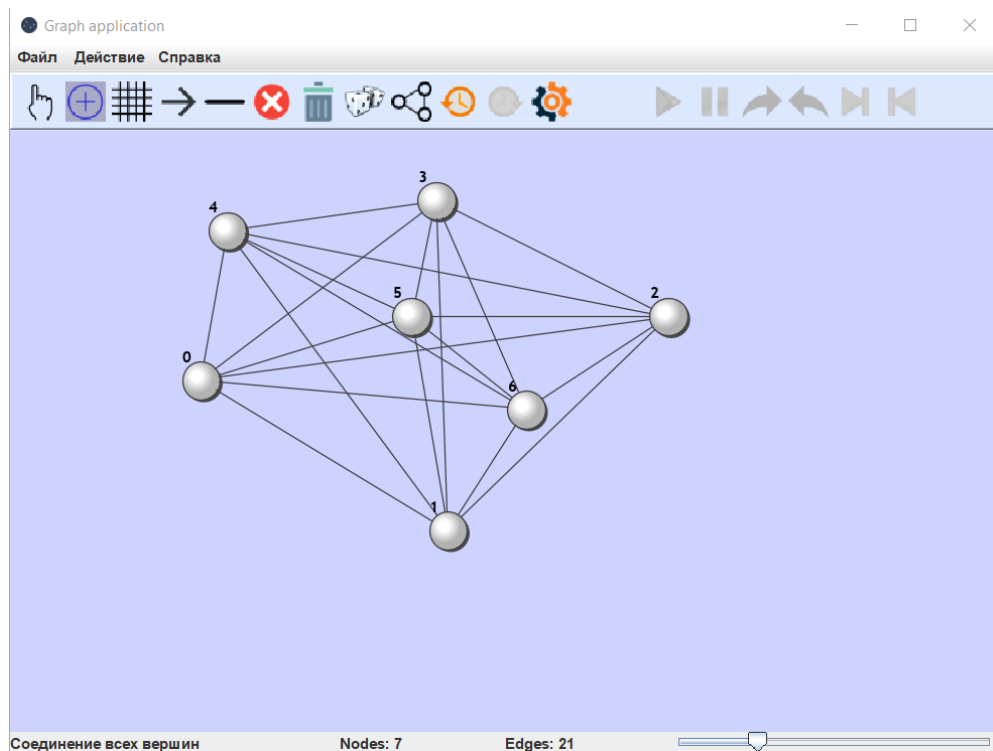


Рисунок 3 – Внешний вид созданного графа

После удалим некоторые ребра.

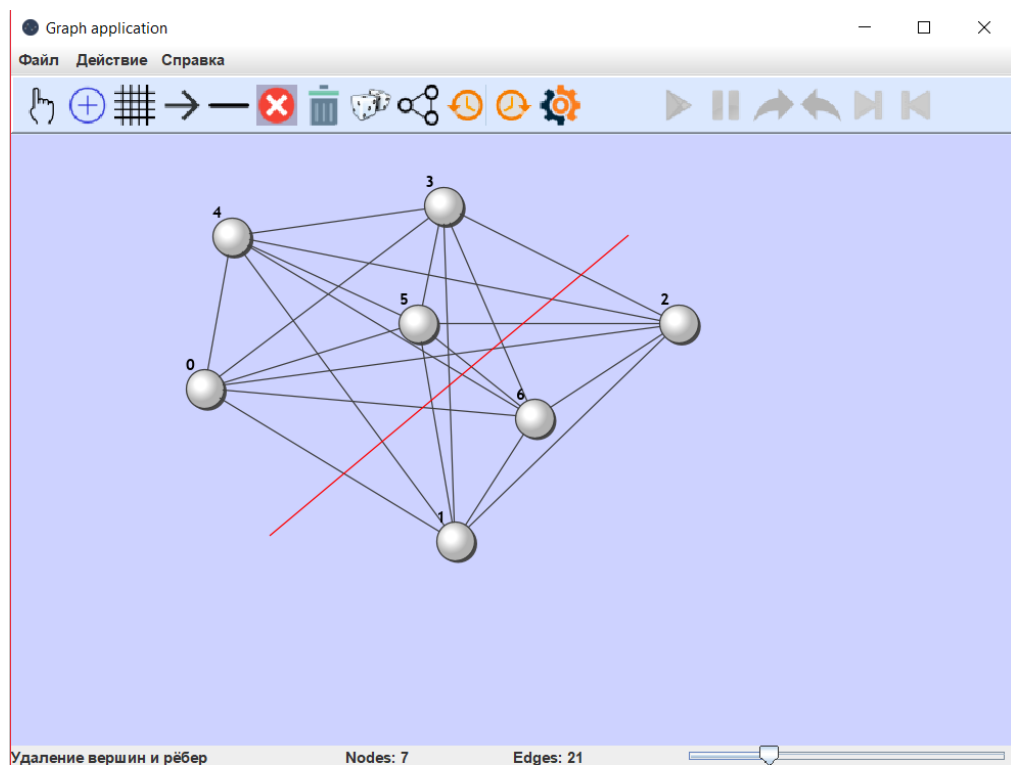


Рисунок 4 – Процесс удаления ребер

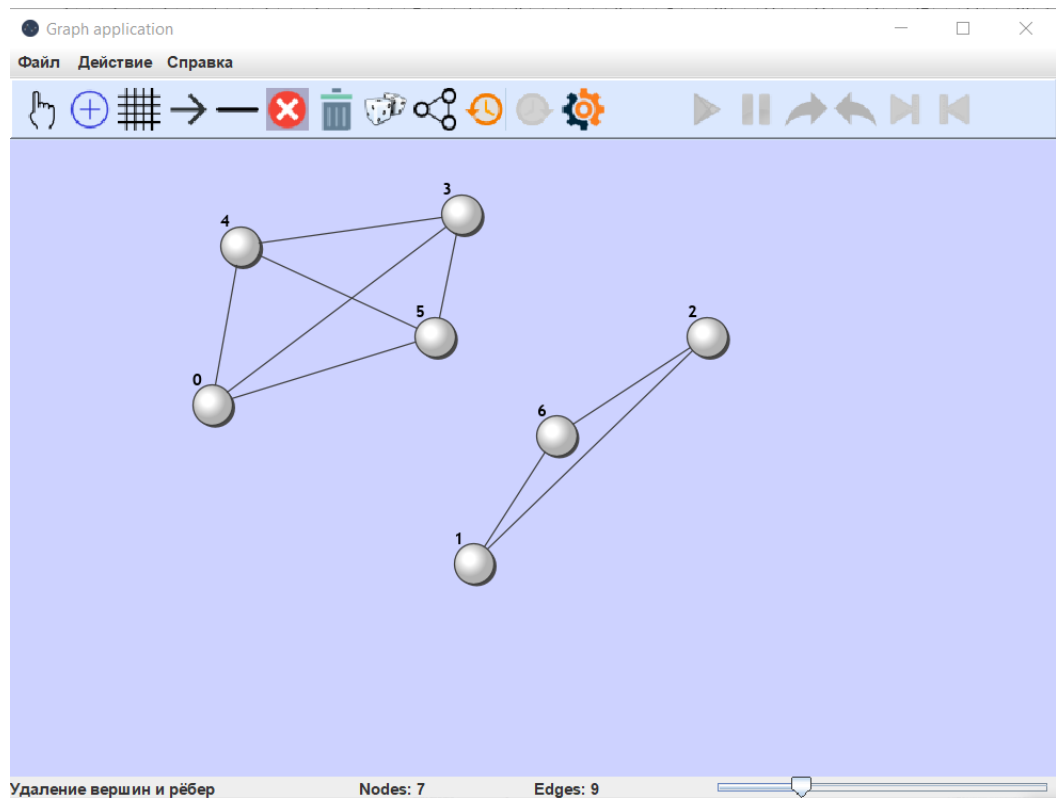


Рисунок 5 – После удаления ребер

Очистим сцену, добавим новые вершины и создадим триангуляционную сеть. После удалим некоторые ребра, что бы получилось 4 отдельных графа.

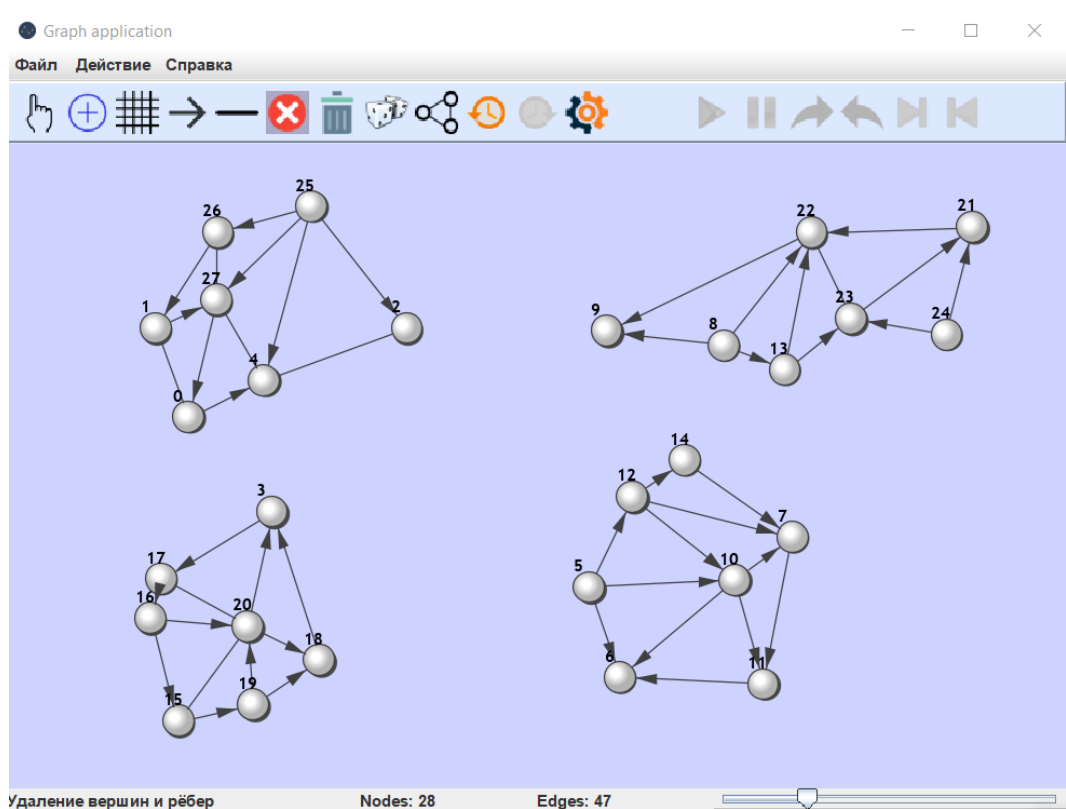


Рисунок 6 – Новый граф

Запустим для него алгоритм Косарайю. Выберем вершину и нажмем пуск. В процессе выполнения можно изменять скорость, приостанавливать и выполнять пошагово.

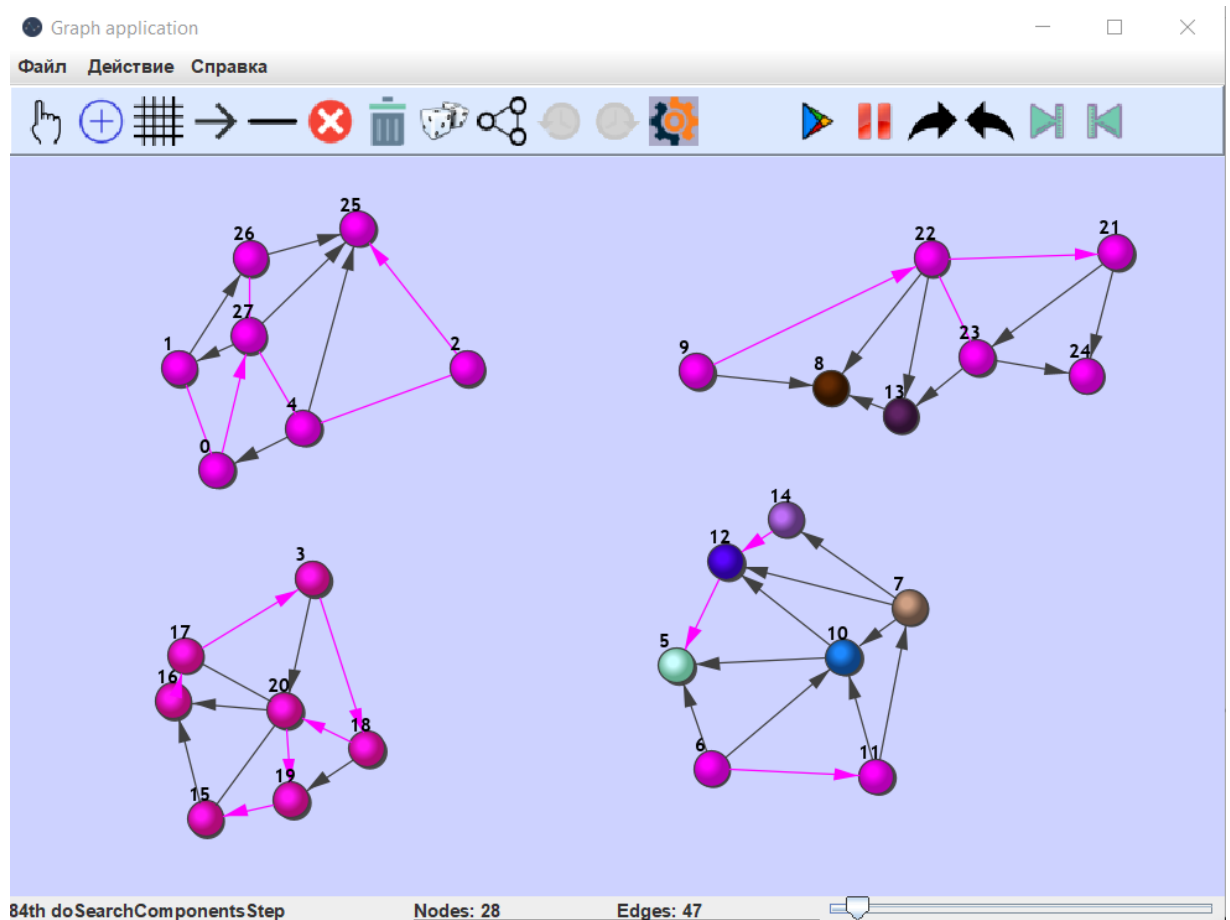


Рисунок 7 – Пауза при выполнении алгоритма

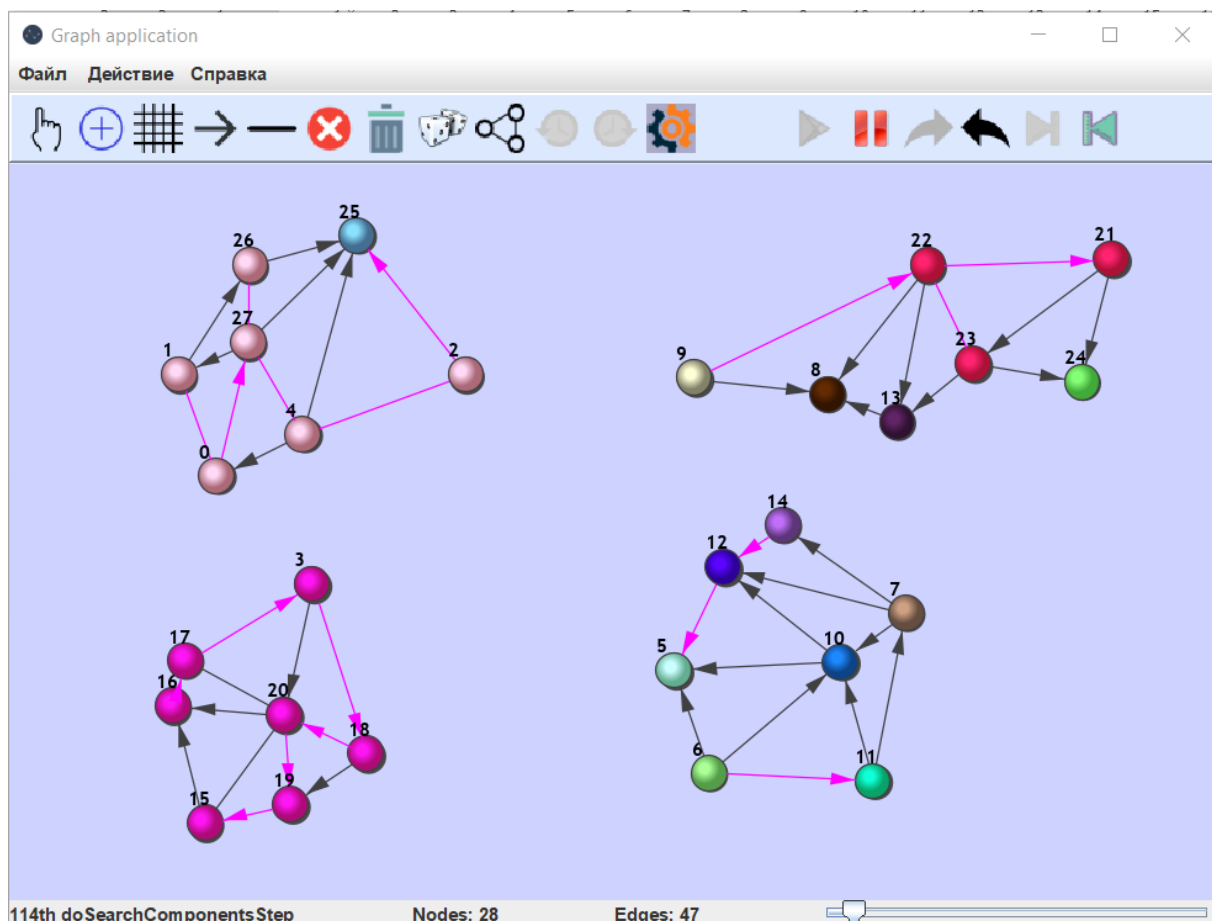


Рисунок 8 – Результат работы алгоритма

Тестирование графического интерфейса пройдено успешно.

5.2. Тестирование алгоритма

Для тестирования алгоритма был написан класс `AlgorithmKosarajuTest`, который симулировал графическую панель и кнопки для работы с алгоритмом. В каждом тесте программа считывала граф из файла и запускала алгоритм Косарайю. Для проверки результата работы алгоритма был написан метод, который проверял, раскрашены ли вершины, принадлежащие одной компоненте сильной связности, в один цвет и отличаются ли цвета каждой компоненты.

Первый тест – тестирование графа, в котором на этапе DFS стек не посещённых вершин несколько раз становится пустым. Программа должна

находить четыре компоненты связности. На рис. 9 приведён соответствующий граф и результат работы алгоритма.

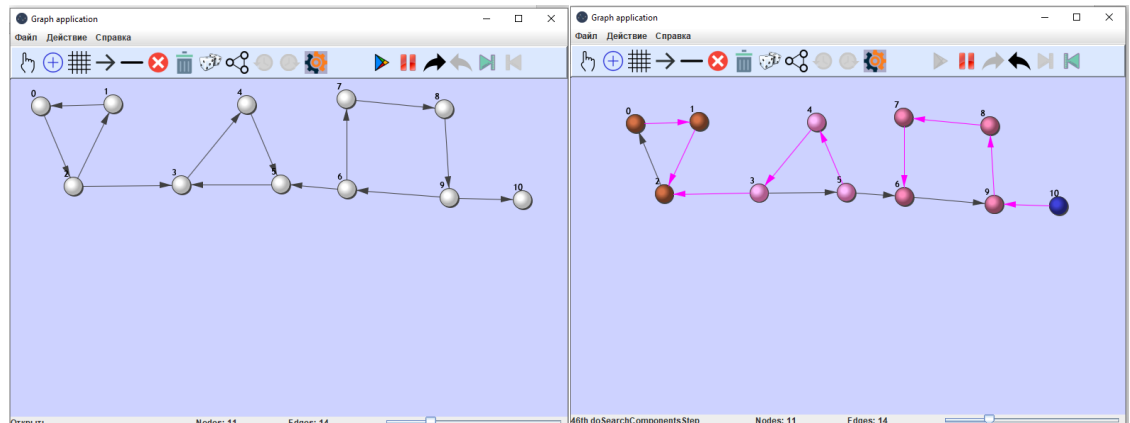


Рисунок 9 – Тестирование графа, стек которого несколько раз становится пустым на этапе DFS

Второй тест – тестирование графа с одной вершиной. Программа должна находить одну компоненту связности. На рис. 10 приведён соответствующий граф и результат работы алгоритма.

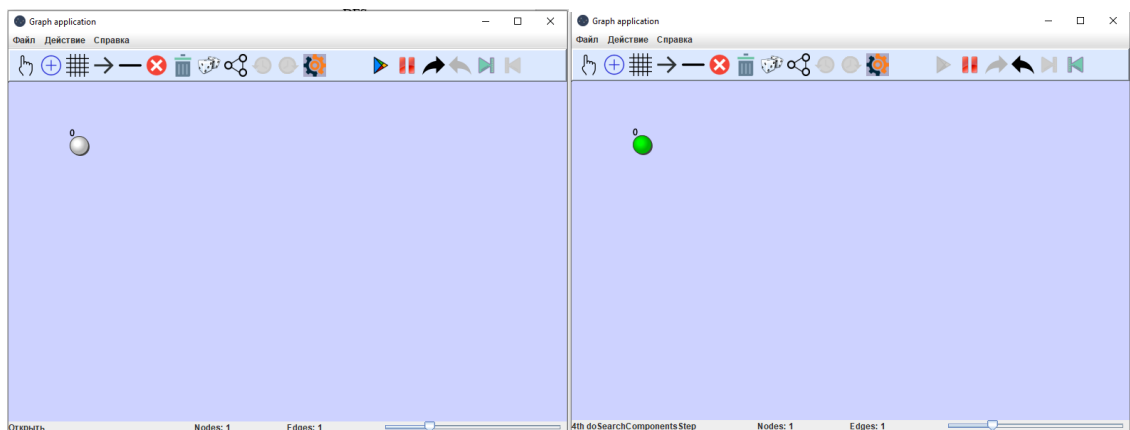


Рисунок 10 – Тестирование графа, с одной вершиной

Третий тест – тестирование графа, состоящего из нескольких одиночных вершин. Программа должна находить пять компонент связности. На рис. 11 приведён соответствующий граф и результат работы алгоритма.

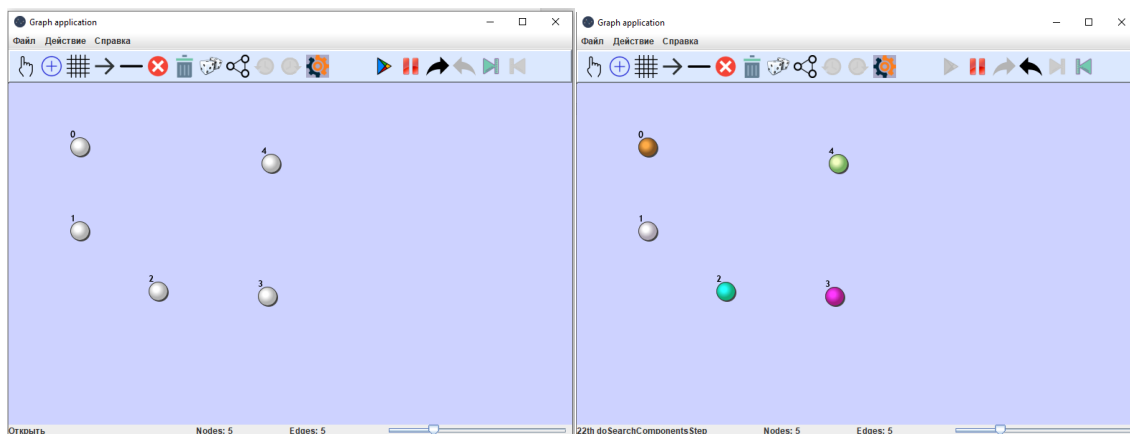


Рисунок 11 – Тестирование графа, состоящего из нескольких
одиночных вершин

Четвёртый тест – тестирование графа, состоящего из двух не связанных компонент связности. Программа должна находить две компоненты связности. На рис. 12 приведён соответствующий граф и результат работы алгоритма.

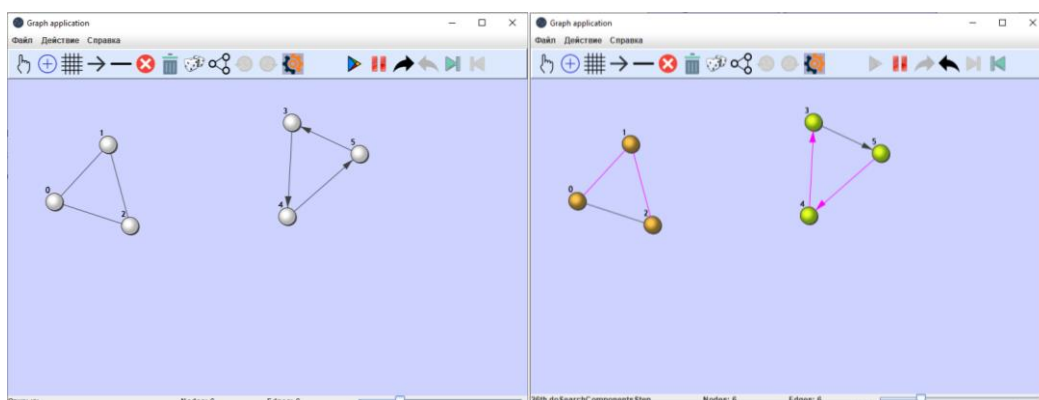


Рисунок 12 – Тестирование графа, состоящего из двух не связанных
компонент связности

Пятый тест – тестирование графа, компоненты связности которого соединены ориентированными рёбрами. Компоненты и соединяющие их рёбра не должны образовывать цикл. Программа должна находить три компоненты связности. На рис. 13 приведён соответствующий граф и результат работы алгоритма.

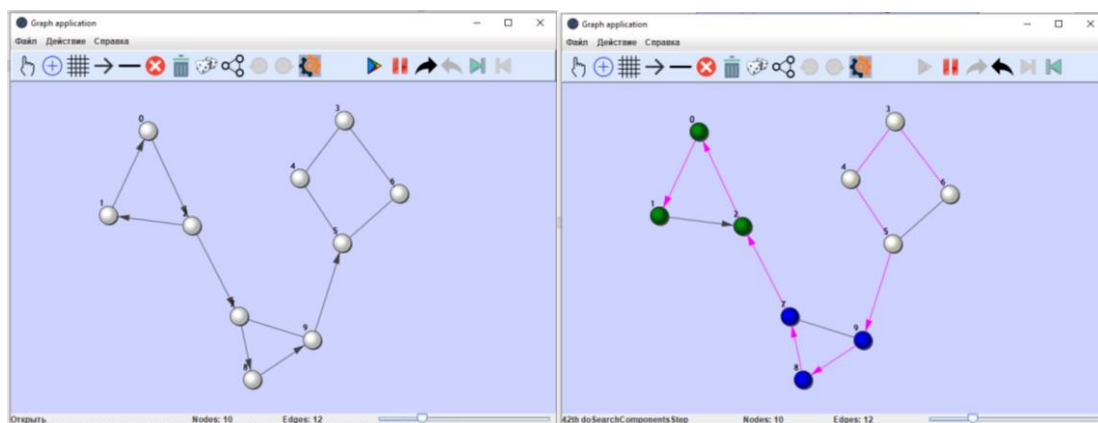


Рисунок 13 – Тестирование графа, компоненты связности которого соединены ориентированными рёбрами

Шестой тест – тестирование графа, компоненты связности которого соединены двумя ориентированными рёбрами. Компоненты и соединяющие их рёбра не должны образовывать цикл. Программа должна находить две компоненты связности. На рис. 14 приведён соответствующий граф и результат работы алгоритма.

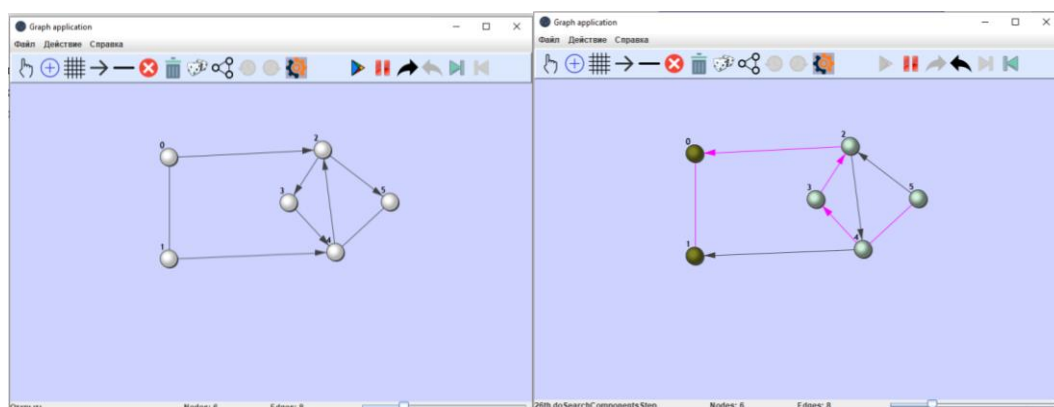


Рисунок 14 – тестирование графа, компоненты связности которого соединены двумя ориентированными рёбрами

Седьмой тест – тестирование графа, все вершины которого связаны между собой. Программа должна находить одну компоненты связности. На рис. 15 приведён соответствующий граф и результат работы алгоритма.

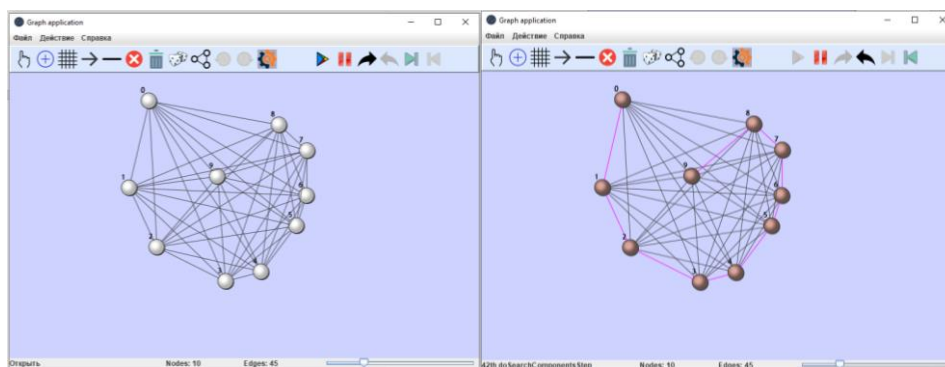


Рисунок 15 – Тестирование графа, все вершины которого связаны между собой

Восьмой тест – тестирование графа, который на первый взгляд может показаться состоящим из трёх компонент сильной связности. Компоненты сильной связности и рёбра, соединяющие их, не должны образовывать цикл. Программа должна находить одну компоненту связности. На рис. 16 приведён соответствующий граф и результат работы алгоритма.

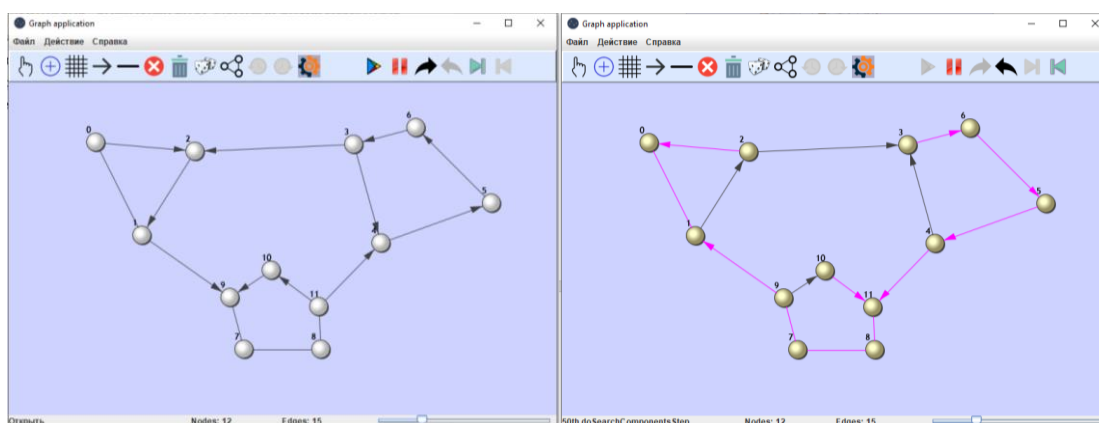


Рисунок 16 – Тестирование графа, который на первый взгляд может показаться состоящим из трёх компонент сильной связности

Девятый тест – тестирование графа, вершины и рёбра которого образуют незамкнутую цепь. Программа должна находить пять компонент связности. На рис. 17 приведён соответствующий граф и результат работы алгоритма.

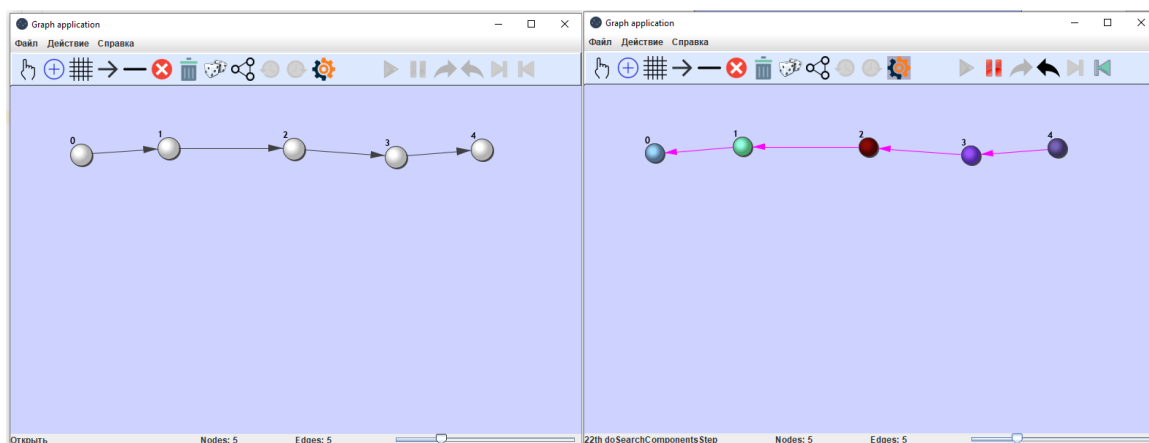


Рисунок 17 – Тестирование графа, вершины и рёбра которого образуют незамкнутую цепь

Девятый тест – тестирование графа, вершины и рёбра которого образуют замкнутую цепь. Программа должна находить одну компоненту связности. На рис. 18 приведён соответствующий граф и результат работы алгоритма.

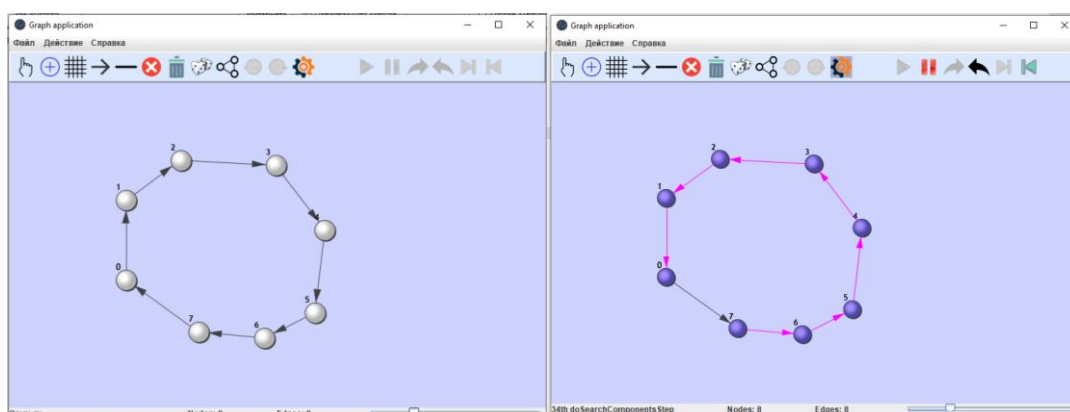


Рисунок 18 – Тестирование графа, вершины и рёбра которого образуют замкнутую цепь

ЗАКЛЮЧЕНИЕ

В результате выполнения учебной практики была разработана и протестирована программа, реализующая алгоритм поиска компонент сильной связанности в графе и наглядно демонстрирующая принцип его работы.

В ходе работы было проведено тестирование с целью выявления возможных ошибок. По результатам было выяснено, что алгоритм работает корректно.

Выполнение данного проекта позволило приобрести навыки, необходимые для будущей профессии, тесно связанной с ИТ. Это навыки:

- разработки в среде программирования Java;
- работы в команде;
- использования известной системы контроля версий GitHub;
- использования библиотеки Swing для реализации графического интерфейса.

Таким образом, цели практики успешно достигнуты, и по окончании разработки получен корректно работающий визуализатор алгоритма

Косарайю на языке программирования Java.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Java. Базовый курс. // Stepik. URL: <https://stepik.org/course/Java-Базовый-курс-187/syllabus> .
2. Брюс Эккель. Философия Java. СПб.: Питер, 2009.
3. Дейтел Х.М., Дейтел П.Дж. Как программировать на Java. Кн. 1: Основы программирования. / пер. с англ. А.В. Козлова. 4-е изд. М.: Бином, 2003;
4. Седжвик Р., Уэйн К. Алгоритмы на Java, 4-е изд.: Пер. с англ. – М.: ООО “И.Д.Вильямс”, 2013. – 848 с.
5. Шилдт Г. - Java 8. Полное руководство. 9-е изд.: Пер. с англ. – М.: ООО “И.Д.Вильямс”, 2015. – 1376 с.

ПРИЛОЖЕНИЕ А

```
import javax.swing.*;

class Main {

    public Main() {

        Application application = new Application();

    }

    public static void main(String[] args) {

        SwingUtilities.invokeLater(new Runnable() {

            public void run() {

                new Main();

            }

        } );

    }

}
```

```
import IO.InputReader;
import IO.OutputWriter;
import Managers.AlgorithmEventManager;
import Managers.GraphEventManager;
import Widgets.ButtonState;
import Widgets.GraphicsPanel;
import Widgets.Button;
import Snapshots.GraphCaretaker;
import ViewCompomemnts.DrawGraph;
import algorithmComponents.AlgorithmButtons;
import algorithmComponents.Algorithms;
import graphComponents.GraphStates;
import graphComponents.RandomGraphCreator;
```

```

import graphComponents.Triangulator;

import java.io.File;
import java.io.IOException;
import java.awt.*;
import java.awt.event.*;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Random;
import javax.swing.*;
import static java.awt.Cursor.getPredefinedCursor;

class Application implements ActionListener {
    public JFrame frame;
    private static GraphicsPanel graphicsPanel;
    private JMenuBar menuBar;
    private JPanel statusBar;
    JPanel toolBar;
    private HashMap<String, Button> buttonHashMap = new HashMap<String,
Button>();
    private HashMap<String, Button> singleActiveButtonHashMap = new
HashMap<>();

    public Application() {
        Image iconOfApp = new ImageIcon("resources/img/Иконка
приложения.png").getImage();

        frame = new JFrame("Graph application");
        frame.setIconImage(iconOfApp);

```

```
frame.setMinimumSize(new Dimension(800, 600));  
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
frame.setLocationRelativeTo(null);
```

```
menuBar = new JMenuBar();
```

```
JMenu menuFile = new JMenu("Файл");  
JMenu menuAction = new JMenu("Действие");  
JMenu menuHelp = new JMenu("Справка");
```

```
menuFile.setMnemonic(KeyEvent.VK_F);
```

```
menuBar.add(menuFile);  
menuBar.add(menuAction);  
menuBar.add(menuHelp);
```

```
JMenuItem itemOpenFile = new JMenuItem("Открыть", KeyEvent.VK_O);  
JMenuItem itemSaveGraph = new JMenuItem("Сохранить граф",  
KeyEvent.VK_S);
```

```
itemOpenFile.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_O,  
InputEvent.CTRL_DOWN_MASK));  
itemSaveGraph.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_S,  
InputEvent.CTRL_DOWN_MASK));
```

```
menuFile.add(itemOpenFile);
```

```
menuFile.add(itemSaveGraph);
```

```
String[] actioncommands = {"Добавить вершины", "Добавить  
ориентированное ребро", "Добавить неориентированное ребро",  
"Удалить вершины и рёбра", "Очистить  
полотно", "Перемещение", "Соединить все вершины", "Создать случайный  
граф"};
```

```
String[] actionIcons =  
{ "resources/img/Добавить(small).png", "resources/img/Направленное  
ребро(small).png", "resources/img/Ненаправленное ребро(small).png",  
  
"resources/img/Удалить(small).png", "resources/img/Очистить(small).png", "resou  
rces/img/Перемещение(small).png", "resources/img/Соединить все(small).png",  
"resources/img/Создать случайный граф(small).png"};
```

```
for(int i = 0; i < actioncommands.length; i++){  
    JMenuItem action = new JMenuItem(actioncommands[i], new  
    ImageIcon(actionIcons[i]));  
    action.addActionListener(this);  
    menuAction.add(action);  
}
```

```
JMenu itemSubMenuAlgorithm = new JMenu("Алгоритм");  
itemSubMenuAlgorithm.setIcon(new  
ImageIcon("resources/img/Алгоритм(small).png"));
```

```
JMenuItem itemKosaraju = new JMenuItem("Косарайю");  
JMenuItem itemDFS = new JMenuItem("Поиск в глубину");
```

```
JMenuItem StepBack = new JMenuItem("Отмена",KeyEvent.VK_Z);  
StepBack.setIcon(new ImageIcon("resources/img/Отмена(small).png"));  
StepBack.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_Z,  
InputEvent.CTRL_DOWN_MASK));
```

```
JMenuItem StepUp = new JMenuItem("Отмена отмены",KeyEvent.VK_Y);  
StepUp.setIcon(new ImageIcon("resources/img/Up(small).png"));  
StepUp.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_Y,  
InputEvent.CTRL_DOWN_MASK));
```

```
itemSubMenuAlgorithm.add(itemKosaraju);  
itemSubMenuAlgorithm.add(itemDFS);
```

```
menuAction.add(StepBack);  
menuAction.add(StepUp);
```

```
StepBack.addActionListener(this);  
StepUp.addActionListener(this);  
itemDFS.addActionListener(this);  
itemKosaraju.addActionListener(this);  
itemOpenFile.addActionListener(this);  
itemSaveGraph.addActionListener(this);
```

```
menuAction.add(itemSubMenuAlgorithm);
```

```
ImageIcon iconQuestion = new ImageIcon("resources/img/Вопрос.png");  
JMenuItem itemAboutProgram = new JMenuItem("О программе",  
iconQuestion);
```

```
menuHelp.add(itemAboutProgram);  
itemAboutProgram.addActionListener(this);
```

```
graphicsPanel = new GraphicsPanel();
```

```
statusBar = new JPanel(new GridLayout(1,3,0,0));
```

```
BoundedRangeModel model = new DefaultBoundedRangeModel(500, 0, 0,  
2000);
```

```
JSlider slider = new JSlider(model);
```

```
slider.addChangeListener(e -> Algorithms.setDelay(slider.getValue()));
```

```
JLabel labelAction = new JLabel();
```

```
JLabel labelNodes = new JLabel("Nodes: " + 0);
```

```
JLabel labelEdges = new JLabel("Edges: " + 0);
```

```
JPanel panelNodesEdges= new JPanel(new GridLayout(1, 2, 0, 0));
```

```
panelNodesEdges.add(labelNodes);
```

```
panelNodesEdges.add(labelEdges);
```

```
graphicsPanel.setPanelNodesEdges(panelNodesEdges);
```

```
statusBar.add(labelAction);
```

```
statusBar.add(panelNodesEdges);
```

```
statusBar.add(slider);
```

```
toolBar = createToolBar();
```

```
frame.add(graphicsPanel);
```

```
frame.add(toolBar, BorderLayout.NORTH);
```

```

frame.add(statusBar, BorderLayout.SOUTH);
frame.setJMenuBar(menuBar);
frame.setVisible(true);

Algorithms.init();
GraphEventManager.getInstance().setGraphicsPanel(graphicsPanel);
AlgorithmEventManager.getInstance().setGraphicsPanel(graphicsPanel);
AlgorithmEventManager.getInstance().setDisplay(labelAction);
AlgorithmEventManager.getInstance().setAlgorithmButtons(new
AlgorithmButtons(buttonHashMap.get("Запустить алгоритм"),
    buttonHashMap.get("Остановить алгоритм"),
    buttonHashMap.get("Сделать шаг вперед"),
    buttonHashMap.get("Сделать шаг назад"),
    buttonHashMap.get("Результат"),
    buttonHashMap.get("Начало"))));
}

public void actionPerformed(ActionEvent ae) {
    String command = ae.getActionCommand();

    JLabel labelAction = (JLabel)statusBar.getComponents()[0];

    if (GraphEventManager.getInstance().getState() ==
GraphStates.ALGORITHM) {
        if (!command.equals("Запустить алгоритм") &&
            !command.equals("Остановить алгоритм") &&
            !command.equals("Сделать шаг вперед") &&
            !command.equals("Сделать шаг назад") &&
            !command.equals("Результат") &&

```



```

        !command.equals("Начало")) {
            AlgorithmEventManager.getInstance().reset();
        }
    }

    if (!command.equals("Добавить ориентированное ребро") &&
        !command.equals("Добавить неориентированное ребро")) {
        GraphEventManager.getInstance().resetConnectData();
    }

    switch(command) {
        case "Открыть":
            labelAction.setText("Открыть");
            InputReader newOne = new InputReader();

            if(newOne.FileOpen) {
                graphicsPanel.setGraph(new DrawGraph(newOne.initFromData()));

                AlgorithmEventManager.getInstance().getAlgorithm().setGraph(GraphEventManager.getInstance().getGraph());
            }

            break;
        case "Сохранить граф":
            OutputWriter saveOne = new OutputWriter();
            saveOne.saveGraph(GraphEventManager.getInstance().getGraph());
            break;
        case "Добавить вершины":
            labelAction.setText("Добавление вершин");
            graphicsPanel.setGraphState(GraphStates.CREATE_NODE);
    }

```

```

        break;
    case "Добавить ориентированное ребро":
        labelAction.setText("Добавление ориентированных рёбер");
        graphicsPanel.setGraphState(GraphStates.CONNECT_NODE);
        GraphEventManager.getInstance().setNodeConnectionType(true);
        break;
    case "Добавить неориентированное ребро":
        labelAction.setText("Добавление неориентированных рёбер");
        graphicsPanel.setGraphState(GraphStates.CONNECT_NODE);
        GraphEventManager.getInstance().setNodeConnectionType(false);
        break;
    case "Перемещение":
        labelAction.setText("Перемещение");
        graphicsPanel.setGraphState(GraphStates.MOVE_NODE);
        break;
    case "Очистить полотно":
        labelAction.setText("Очищение полотна");

    GraphCaretaker.push(GraphEventManager.getInstance().getGraph().save());
    GraphEventManager.getInstance().removeGraph();
    break;
    case "Удалить вершины и рёбра":
        labelAction.setText("Удаление вершин и рёбер");
        graphicsPanel.setGraphState(GraphStates.DELETE_NODE);
        break;
    case "Соединить все вершины":
        labelAction.setText("Соединение всех вершин");

    GraphCaretaker.push(GraphEventManager.getInstance().getGraph().save());
    GraphEventManager.getInstance().connectAllVertices();

```

```

        break;
    case "Создать случайный граф":

GraphCaretaker.push(GraphEventManager.getInstance().getGraph().save());

        graphicsPanel.setGraph(RandomGraphCreator.create(
            1 + new Random(System.currentTimeMillis()).nextInt(16),
            0.25D, graphicsPanel.getWidth(),
            graphicsPanel.getHeight(),
            true));

AlgorithmEventManager.getInstance().getAlgorithm().setGraph(GraphEventManager.getInstance().getGraph());

        break;
    case "Триангулировать":

GraphCaretaker.push(GraphEventManager.getInstance().getGraph().save());

        Triangulator.triangulate(GraphEventManager.getInstance().getGraph());
        break;
    case "Алгоритм":

        GraphEventManager.getInstance().setState(GraphStates.ALGORITHM);
        AlgorithmEventManager.getInstance().reset();

AlgorithmEventManager.getInstance().getAlgorithm().setGraph(GraphEventManager.getInstance().getGraph());

        break;
    case "Запустить алгоритм":
    case "Остановить алгоритм":
    case "Результат":
        if (GraphEventManager.getInstance().getState() ==
GraphStates.ALGORITHM &&

```

```

        AlgorithmEventManager.getInstance().isInitialized()) {
            AlgorithmEventManager.getInstance().sendCommand(command);
        }
        break;
    case "Сделать шаг вперед":
    case "Сделать шаг назад":
    case "Начало":
        if (GraphEventManager.getInstance().getState() ==
GraphStates.ALGORITHM &&
            AlgorithmEventManager.getInstance().isInitialized()) {
            AlgorithmEventManager.getInstance().sendCommand("Остановить
алгоритм");
            AlgorithmEventManager.getInstance().sendCommand(command);
        }
        break;
    case "Поиск в глубину":
        labelAction.setText("DFS");
        Algorithms.selectAlgorithmByName("DFS");
        AlgorithmEventManager.getInstance().reset();

    AlgorithmEventManager.getInstance().getAlgorithm().setGraph(GraphEventManager.getInstance().getGraph());
        break;
    case "Косарайю":
        labelAction.setText("Kosaraju");
        Algorithms.selectAlgorithmByName("Kosaraju");
        AlgorithmEventManager.getInstance().reset();

    AlgorithmEventManager.getInstance().getAlgorithm().setGraph(GraphEventManager.getInstance().getGraph());

```

```

        break;
    case "О программе":
        labelAction.setText("О программе");
        openHelp();
        break;
    case "Отмена":
        if (!AlgorithmEventManager.getInstance().isInitialized()) {

GraphEventManager.getInstance().getGraph().restore(GraphCaretaker.pop());

        }
        break;
    case "Отмена отмены":
        if (!AlgorithmEventManager.getInstance().isInitialized()) {

GraphEventManager.getInstance().getGraph().restore(GraphCaretaker.poll());

        }
        break;
    default:
        labelAction.setText("");
        graphicsPanel.setGraphState(GraphStates.NOTHING);
    }

    ButtonClicked(command);

    graphicsPanel.updatePanelNodesEdges();
    graphicsPanel.updateUI();
}

public void ButtonClicked(String command) {
    if (command.equals("Алгоритм")) {

```

```

        buttonHashMap.get("Запустить алгоритм").setEnabled(true);
        buttonHashMap.get("Остановить алгоритм").setEnabled(true);
        buttonHashMap.get("Сделать шаг вперед").setEnabled(true);
        buttonHashMap.get("Сделать шаг назад").setEnabled(false);
        buttonHashMap.get("Отмена").setEnabled(false);
        buttonHashMap.get("Отмена отмены").setEnabled(false);
        buttonHashMap.get("Результат").setEnabled(true);
        buttonHashMap.get("Начало").setEnabled(false);
    }
    else if (GraphEventManager.getInstance().getState() !=
GraphStates.ALGORITHM) {
        buttonHashMap.get("Запустить алгоритм").setEnabled(false);
        buttonHashMap.get("Остановить алгоритм").setEnabled(false);
        buttonHashMap.get("Сделать шаг вперед").setEnabled(false);
        buttonHashMap.get("Сделать шаг назад").setEnabled(false);
        buttonHashMap.get("Результат").setEnabled(false);
        buttonHashMap.get("Начало").setEnabled(false);

        if (!GraphCaretaker.isEmpty()) {
            buttonHashMap.get("Отмена").setEnabled(true);
        }

        if (!GraphCaretaker.isFull()) {
            buttonHashMap.get("Отмена отмены").setEnabled(true);
        }
    }

    if (singleActiveButtonHashMap.containsKey(command)) {
        for (Button button : singleActiveButtonHashMap.values()) {
            button.setState(ButtonState.INACTIVE);
        }
    }

```

```

        button.changeState();
    }

singleActiveButtonHashMap.get(command).setState(ButtonState.ACTIVE);
    singleActiveButtonHashMap.get(command).changeState();
}

if (command.equals("Запустить алгоритм")) &&
!AlgorithmEventManager.getInstance().isInitialized()) {
    buttonHashMap.get("Запустить
алгоритм").setState(ButtonState.INACTIVE);
}
else if (command.equals("Запустить алгоритм")) {
    buttonHashMap.get("Запустить
алгоритм").setState(ButtonState.ACTIVE);
    buttonHashMap.get("Запустить алгоритм").changeState();
}
else {
    buttonHashMap.get("Запустить
алгоритм").setState(ButtonState.INACTIVE);
    buttonHashMap.get("Запустить алгоритм").changeState();
}
}

public JPanel createToolBar(){
    String[] icons = {"resources/img/Перемещение.png",
        "resources/img/Добавить.png",
        "resources/img/Соединить все.png",

```

```
"resources/img/Направленное ребро.png",  
"resources/img/Ненаправленное ребро.png",  
"resources/img/Удалить.png",  
"resources/img/Очистить.png",  
"resources/img/Создать случайный граф.png",  
"resources/img/Триангуляция.png",  
"resources/img/Отмена.png",  
"resources/img/Up.png",  
"resources/img/Алгоритм.png",  
"resources/img/Запустить алгоритм.png",  
"resources/img/Остановить алгоритм.png",  
"resources/img/Сделать шаг вперед.png",  
"resources/img/Сделать шаг назад.png",  
"resources/img/Конец.png",  
"resources/img/Начало.png"  
};
```

```
String[] commands = {  
    "Перемещение",  
    "Добавить вершины",  
    "Соединить все вершины",  
    "Добавить ориентированное ребро",  
    "Добавить неориентированное ребро",  
    "Удалить вершины и рёбра",  
    "Очистить полотно",  
    "Создать случайный граф",  
    "Триангулировать",  
    "Отмена",  
    "Отмена отмены",  
    "Алгоритм",
```



```
"Запустить алгоритм",  
"Остановить алгоритм",  
"Сделать шаг вперед",  
"Сделать шаг назад",  
"Результат",  
"Начало"  
};
```

```
ArrayList<String> singleActiveCommands = new ArrayList<>();  
singleActiveCommands.add("Перемещение");  
singleActiveCommands.add("Добавить вершины");  
singleActiveCommands.add("Добавить ориентированное ребро");  
singleActiveCommands.add("Добавить неориентированное ребро");  
singleActiveCommands.add("Удалить вершины и рёбра");  
singleActiveCommands.add("Алгоритм");
```

```
JPanel toolBar = new JPanel(new FlowLayout(FlowLayout.LEFT));  
toolBar.setBackground(new Color(219, 232, 254));
```

```
for(int i = 0; i < icons.length; i++){  
    Button button = new Button();  
    button.setIcon(new ImageIcon(icons[i]));  
    button.setActionCommand(commands[i]);  
    button.setPreferredSize(new Dimension(32, 32));  
    button.addActionListener(this);  
  
    button.setBackground(new Color(219, 232, 254));  
    button.setFocusPainted(false);  
  
    if(commands[i].equals("Запустить алгоритм")){
```

```

        for(int j = 0; j < 8; j++) {
            toolBar.add(new JSeparator(SwingConstants.VERTICAL));
        }
    }

    button.setToolTipText(commands[i]);
    button.setCursor(getPredefinedCursor(Cursor.HAND_CURSOR));
    button.setBorderPainted(false);

    if (commands[i].equals("Запустить алгоритм") ||
        commands[i].equals("Остановить алгоритм") ||
        commands[i].equals("Сделать шаг вперед") ||
        commands[i].equals("Сделать шаг назад") ||
        commands[i].equals("Отмена") ||
        commands[i].equals("Отмена отмены") ||
        commands[i].equals("Результат") ||
        commands[i].equals("Начало")) {
        button.setEnabled(false);
    }

    toolBar.add(button);
    buttonHashMap.put(commands[i], button);
}

for (String buttonCommand : buttonHashMap.keySet()) {
    if (singleActiveCommands.contains(buttonCommand)) {
        singleActiveButtonHashMap.put(buttonCommand,
buttonHashMap.get(buttonCommand));
    }
}

```

```

    GraphCaretaker.button1 = buttonHashMap.get("Отмена");
    GraphCaretaker.button2 = buttonHashMap.get("Отмена отмены");

    toolBar.setBorder(BorderFactory.createRaisedBevelBorder());

    return toolBar;
}

public void openHelp() {
    File htmlFile = new File("resources/html/Help.html");
    try {
        Desktop.getDesktop().browse(htmlFile.toURI());
    } catch(IOException exception) {
        System.out.println("IO");
    }
}

}

package IO;

import ViewCompomemnts.DrawNode;
import graphComponents.Edge;
import graphComponents.Graph;
import graphComponents.Node;

import javax.swing.*;
import java.awt.geom.Point2D;
import java.io.*;
import java.util.ArrayList;
import java.util.Random;

```

```
enum DataMode {
    DEFAULT,
    RANDOM,
    COORDS
}
```

```
public class InputReader {
    private ArrayList<String> lines;
    private DataMode mode;
    public boolean FileOpen = false;

    public InputReader(String fileName) {
        lines = new ArrayList<String>(50);

        File file = new File(fileName);

        if (file != null) {
            FileOpen = true;
        }

        try(BufferedReader reader = new BufferedReader(new
InputStreamReader(new FileInputStream(file)))){
            String line;
            while((line = reader.readLine()) != null){
                if (!line.equals(""))lines.add(line);
            }
        }catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```

    }

}

public InputReader(){
    lines = new ArrayList<String>(50);
    JFileChooser fileopen = new JFileChooser(new
File("./resources/GraphExamples"));
    int ret = fileopen.showOpenDialog(null);
    if (ret == JFileChooser.APPROVE_OPTION) FileOpen = true;
    if (FileOpen) {
        File file = fileopen.getSelectedFile();
        try(BufferedReader reader = new BufferedReader(new
InputStreamReader(new FileInputStream(file)))){
            String line;
            while((line = reader.readLine()) != null){
                if (!line.equals(""))lines.add(line);
            }
        }catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

public Graph initFromData(){
    StringBuilder name;
    Graph graph = new Graph();
    int radius = 150;

```

```

int indent = 0;
switch (lines.get(0).replaceAll(" ", "")){
    case "Location:default": mode = DataMode.DEFAULT; break;
    case "Location:random" : mode = DataMode.RANDOM; break;
    case "Location:bycoordinates": mode = DataMode.COORDS; break;
    default: mode = DataMode.DEFAULT; indent = 1; break;
}
for(int i = 1 - indent; i < lines.size(); i++){
    String line = lines.get(i);
    int k = 0;
    for (; k < line.length(); k++){
        if (line.charAt(k) == ' ' || line.charAt(k) == '(') break;
    }
    name = new StringBuilder(line.substring(0,k));
    Point2D.Double point = new Point2D.Double();
    line = line.replaceAll(" ", "");
    switch (mode){
        case DEFAULT:
            point = new Point2D.Double(350+ radius*Math.cos(Math.toRadians(i
* 360 / (lines.size() - (1-indent)))),
            250 + radius*Math.sin(Math.toRadians(i*360/(lines.size()-(1-
indent))))); break;
        case RANDOM:
            Random rand = new Random();
            point = new Point2D.Double( 50 + rand.nextInt(700), 50 +
rand.nextInt(400)); break;
        case COORDS:
            int
x
=
Integer.parseInt(line.substring(line.indexOf('(')+1,line.indexOf(';')));

```

```

        int                                     y                                     =
Integer.parseInt(line.substring(line.indexOf(';')+1,line.indexOf(')')));
        point = new Point2D.Double(x,y);
    }
    graph.add(new Node(new DrawNode(point,name.toString())));
}
for (int i = 1- indent; i< lines.size(); i++){
    String line = lines.get(i);
    int ind = 0;
    if(mode == DataMode.COORDS){
        line = line.substring(line.indexOf(')') + 1);
        ind = 1;
    }
    line = line.trim();
    String[] neighbours = line.split("\\s+");
    Node SourceNode = graph.getNodes().get(i-(1-indent));
    Node DestNode = SourceNode;
    for(int j = 1 - ind; j < neighbours.length; j++){
        for (Node node :graph.getNodes()){
            if(node.getView().getName().equals(neighbours[j])){
                DestNode = node; break;
            }
        }
    }

    graph.add(new Edge(SourceNode, DestNode,true));
}
}

return graph;
}

```

```
}
```

```
package IO;
```

```
import graphComponents.Edge;  
import graphComponents.Graph;  
import graphComponents.Node;
```

```
import javax.swing.*;  
import java.io.File;  
import java.io.FileWriter;  
import java.io.IOException;  
import java.util.ArrayList;
```

```
public class OutputWriter {  
    public void saveGraph(Graph graph){  
        ArrayList<Node> nodes = graph.getNodes();  
        StringBuilder[] res = new StringBuilder[nodes.size()+1];  
        for (int i = 0; i < res.length; i++) {  
            res[i] = new StringBuilder("");  
        }  
        res[0] = new StringBuilder("Location: by coordinates");  
        for(int i = 0; i<nodes.size();i++){  
            Node node = nodes.get(i);  
            res[i+1].append(node.getView().getName() + "("  
(int)node.getView().getPosition().getX() + ";"  
            + (int)node.getView().getPosition().getY() + ") ");  
            ArrayList<Edge> edges = node.getEdges();  
            for(Edge edge : edges){
```



```

Node destNode = edge.getSmartNeighbour(node);
if (destNode != null){
    res[i+1].append(destNode.getView().getName()+ " ");
}
}
}

```

```

JFileChooser savefile = new JFileChooser(new File("./GraphExamples"));
int ret = savefile.showSaveDialog(null);
if(ret == JFileChooser.APPROVE_OPTION){
    File file = savefile.getSelectedFile();
    try(FileWriter writer = new FileWriter(file, false)) {
        for (StringBuilder string: res){
            writer.append(string.toString() + System.lineSeparator());

        }
    }
    catch (IOException ex){
        System.out.println(ex.getMessage());
    }
}
}
}

```

```

package Managers;

```

```

import Widgets.ButtonState;
import Widgets.GraphicsPanel;
import algorithmComponents.Algorithm;

```

```

import algorithmComponents.AlgorithmButtons;

import javax.swing.*;

public class AlgorithmEventManager {
    private static final int BASIC_DO_STEP_TIMER_DELAY = 200;
    private static final AlgorithmEventManager instance = new
AlgorithmEventManager();

    private Algorithm currentAlgorithm;
    private AlgorithmButtons algorithmButtons;

    private GraphicsPanel graphicsPanel;
    private JLabel display;
    private Timer doStepTimer = new Timer(BASIC_DO_STEP_TIMER_DELAY,
e->sendCommand("Сделать шаг вперед"));

    public static AlgorithmEventManager getInstance() {
        return instance;
    }

    public boolean isInitialized() {
        return currentAlgorithm.isInitialized();
    }

    public void sendCommand(String command) {
        Algorithm.AlgorithmState algorithmState = null;

        switch (command) {
            case "Запустить алгоритм":
                doStepTimer.start();

```

```

        break;
    case "Остановить алгоритм":
        doStepTimer.stop();
        break;
    case "Сделать шаг вперед":
        algorithmState = doForwardStep();
        break;
    case "Сделать шаг назад":
        algorithmState = doBackwardStep();
        break;
    case "Результат":
        algorithmState = jumpToFinal();
        break;
    case "Начало":
        algorithmState = jumpToStart();
        break;
}

if (algorithmState != null) {
    repaint(algorithmState.stepsColorDataBase);
    updateButtons(algorithmState.initialized, algorithmState.step);
    sendMessage(algorithmState.stepsColorDataBase.stepName);
}
}

private void repaint(Algorithm.StepsColorDataBase currentStepColorData) {
    currentStepColorData.resetColors();
    graphicsPanel.repaint();
}

```

```

private void updateButtons(boolean initialized, int step) {
    if (step == 2) {
        algorithmButtons.buttons.get(0).setState(ButtonState.INACTIVE);
        algorithmButtons.buttons.get(0).changeState();
        algorithmButtons.buttons.get(0).setEnabled(false);
        algorithmButtons.buttons.get(2).setEnabled(false);
        algorithmButtons.buttons.get(4).setEnabled(false);
    }
    else {
        algorithmButtons.buttons.get(0).setEnabled(true);
        algorithmButtons.buttons.get(2).setEnabled(true);
        algorithmButtons.buttons.get(4).setEnabled(true);
    }

    if (step == 0) {
        algorithmButtons.buttons.get(3).setEnabled(false);
        algorithmButtons.buttons.get(5).setEnabled(false);
    }
    else {
        algorithmButtons.buttons.get(3).setEnabled(true);
        algorithmButtons.buttons.get(5).setEnabled(true);
    }

    if (!initialized) {
        algorithmButtons.buttons.get(3).setEnabled(false);
        algorithmButtons.buttons.get(5).setEnabled(false);
    }
}

private void sendMessage(String message) {

```

```

        display.setText(message);
    }

    private Algorithm.AlgorithmState doForwardStep() {
        currentAlgorithm.doForwardStep();
        return currentAlgorithm.getState();
    }

    private Algorithm.AlgorithmState doBackwardStep() {
        currentAlgorithm.doBackwardStep();
        return currentAlgorithm.getState();
    }

    private Algorithm.AlgorithmState jumpToFinal() {
        Algorithm.AlgorithmState algorithmState;

        do {
            currentAlgorithm.doForwardStep();
            algorithmState = currentAlgorithm.getState();
        } while (algorithmState.step != 2);

        return algorithmState;
    }

    private Algorithm.AlgorithmState jumpToStart() {
        Algorithm.AlgorithmState algorithmState;

        do {
            currentAlgorithm.doBackwardStep();
            algorithmState = currentAlgorithm.getState();
        } while (algorithmState.step != 0);
    }

```

```

        return algorithmState;
    }

    public void reset() {
        doStepTimer.stop();
        graphicsPanel.setGraphBasicColor();
        updateButtons(false, 1);
        currentAlgorithm.reset();
    }

    public void setAlgorithm(Algorithm algorithm) {
        currentAlgorithm = algorithm;
        currentAlgorithm.setGraph(GraphEventManager.getInstance().getGraph());
    }

    public void setGraphicsPanel(GraphicsPanel graphicsPanel) {
        this.graphicsPanel = graphicsPanel;
    }

    public void setDisplay(JLabel display) {
        this.display = display;
    }

    public void setAlgorithmButtons(AlgorithmButtons algorithmButtons) {
        this.algorithmButtons = algorithmButtons;
    }

    public void setDelay(int delay) {
        doStepTimer.setDelay(delay);
    }

```

```

    public Algorithm getAlgorithm() {
        return currentAlgorithm;
    }
}

package Managers;

import Snapshots.GraphCaretaker;
import ViewCompomemnts.DrawNode;
import Widgets.GraphicsPanel;
import graphComponents.Edge;
import graphComponents.Graph;
import graphComponents.GraphStates;
import graphComponents.Node;

import javax.swing.*;
import java.awt.*;
import java.awt.event.MouseWheelEvent;
import java.awt.event.MouseEvent;
import java.awt.geom.Line2D;
import java.awt.geom.Point2D;
import java.util.ArrayList;
import java.util.Arrays;

public class GraphEventManager {
    private Graph graph;
    private GraphStates graphState;
    private DraggData draggData;
    private ConnectData connectData;

```

```

private DeleteData deleteData;

private Point oldDragPoint;


private GraphicsPanel graphicsPanel;


private static final GraphEventManager instance = new GraphEventManager();


public static GraphEventManager getInstance() {
    return instance;
}


public void setGraph(Graph graph) {
    this.graph = graph;
}


public GraphStates getState() {
    return graphState;
}


public int getNodesCount() {
    return graph.getNodes().size();
}


public int getEdgesCount() {
    return graph.getEdges().size();
}


public Graph getGraph() {
    return graph;
}

```



```

public void setState(GraphStates state) {
    switch (state) {
        case CREATE_NODE:
        case MOVE_NODE:
        case NOTHING:
        case DELETE_NODE:
        case ALGORITHM:
            connectData.clear();
            break;
        case CONNECT_NODE:
            break;
    }

    graphState = state;
}

public void removeGraph() {
    graph.clear();
}

public void connectAllVertices() {
    graph.connectAllVertices();
}

public boolean isCutting() {
    return deleteData.cutting;
}

public void setNodeConnectionType(boolean isDirected) {
    ConnectData.IS_DIRECTED_CONNECTION = isDirected;
}

```

```
}
```

```
public void setGraphicsPanel(GraphicsPanel graphicsPanel) {  
    this.graphicsPanel = graphicsPanel;  
}
```

```
public Node getNodeOnPos(Point position, ArrayList<Node> nodes) {  
    for (Node node : nodes) {  
        if (node.getView().getBoundingRect().contains(position)) {  
            return node;  
        }  
    }  
  
    return null;  
}
```

```
public ArrayList<Point> getScissors() {  
    ArrayList<Point> pair = new ArrayList<>();  
    pair.add(deleteData.firstPoint);  
    pair.add(deleteData.secondPoint);  
  
    return pair;  
}
```

```
public void resetConnectData() {  
    connectData.clear();  
}
```

```
private GraphEventManager() {  
    graph = null;  
    graphState = GraphStates.NOTHING;
```

```

    draggData = new DraggData();
    connectData = new ConnectData();
    deleteData = new DeleteData();
    graphicsPanel = null;
}

private int getNewNodeName() {
    int newNodeName = 0;

    ArrayList<Node> nodes = graph.getNodes();
    Integer[] mas = new Integer[nodes.size()];

    for(int i = 0; i< nodes.size(); i++ ){
        mas[i] = Integer.parseInt(nodes.get(i).getView().getName());
    }

    Arrays.sort(mas);

    for (int i : mas){
        if(newNodeName != i) {
            break;
        }

        newNodeName++;
    }

    return newNodeName;
}

public void mousePressed(MouseEvent mouseEvent) {
    oldDragPoint = mouseEvent.getPoint();

```

```

    if (SwingUtilities.isMiddleMouseButton(mouseEvent)) {
        return;
    }

    switch (graphState) {
        case CREATE_NODE:
            if (mouseEvent.getButton() == MouseEvent.BUTTON1) {
                GraphCaretaker.push(graph.save());

                graph.add(new Node(new DrawNode(new
Point2D.Double(mouseEvent.getX(), mouseEvent.getY()),
                String.valueOf(getNewNodeName()))));
            }

            break;
        case CONNECT_NODE:
            if (mouseEvent.getButton() == MouseEvent.BUTTON3) {
                connectData.clear();
                break;
            }
            else if (mouseEvent.getButton() == MouseEvent.BUTTON1) {
                connectData.addNodeOnMousePos(mouseEvent.getPoint(),
graph.getNodes());

                if (connectData.edgeCreated) {
                    Edge newEdge = connectData.getEdgeIfExists();

                    if (newEdge != null) {
                        GraphCaretaker.push(graph.save());
                        graph.add(newEdge);
                    }
                }
            }
        }
    }

```

```

        }
    }
}
break;
case MOVE_NODE:
    if (mouseEvent.getButton() == MouseEvent.BUTTON1) {
        draggData.grabNodeOnMousePos(mouseEvent.getPoint(),
graph.getNodes());

        if (draggData.isDragg) {
            GraphCaretaker.push(graph.save());
        }
    }
    break;
case NOTHING:
    break;
case DELETE_NODE:
    deleteData.setFirstPoint(mouseEvent.getPoint());
    deleteData.secondPoint = mouseEvent.getPoint();

    for (Node node : graph.getNodes()) {
        if
(node.getView().getBoundingRect().contains(mouseEvent.getPoint())) {
            GraphCaretaker.push(graph.save());
            graph.remove(node);
            graph.removeAll(node.getEdges());
            node.destroy();

            break;
        }
    }

```

```

    }
    break;
case ALGORITHM:
    if (!AlgorithmEventManager.getInstance().isInitialized()) {
        Node    selectedNode    =    getNodeOnPos(mouseEvent.getPoint(),
graph.getNodes());

        if (selectedNode != null) {

AlgorithmEventManager.getInstance().getAlgorithm().initialize(selectedNode);
        }
    }
    break;
}
}

```

```

public void mouseWheelMoved(MouseWheelEvent e, Point2D.Double center) {
    double scale = 1D;
    center.x = e.getPoint().x;
    center.y = e.getPoint().y;

    if (e.getPreciseWheelRotation() < 0) {
        scale += 0.1;
    } else {
        scale -= 0.1;
    }

    DrawNode.scale *= scale;

    if (DrawNode.scale > 4) {

```

```

        DrawNode.scale /= scale;
        scale = 1;
    }
    if (DrawNode.scale < 0.5) {
        DrawNode.scale /= scale;
        scale = 1;
    }

    if (scale != 1) {
        for (Node node : graph.getNodes()) {
            Point2D.Double nodePoint2D = node.getView().getPosition();

            if (center.y > nodePoint2D.y) {
                nodePoint2D.y = center.y - Math.abs(center.y - nodePoint2D.y) * scale;
            } else if (center.y < nodePoint2D.y) {
                nodePoint2D.y = center.y + Math.abs(center.y - nodePoint2D.y) *
scale;
            }

            if (center.x > nodePoint2D.x) {
                nodePoint2D.x = center.x - Math.abs(center.x - nodePoint2D.x) * scale;
            } else if (center.x < nodePoint2D.x) {
                nodePoint2D.x = center.x + Math.abs(center.x - nodePoint2D.x) *
scale;
            }
        }
    }
}

public void mouseReleased(MouseEvent mouseEvent) {
    if (SwingUtilities.isMiddleMouseButton(mouseEvent)) {

```

```

        return;
    }

    switch (graphState) {
        case CREATE_NODE:
            break;
        case CONNECT_NODE:
            break;
        case MOVE_NODE:
            draggData.clear();
            break;
        case NOTHING:
            break;
        case DELETE_NODE:
            deleteData.setSecondPoint(mouseEvent.getPoint());
            ArrayList<Edge> edgesToDelete = new ArrayList<>();

            for (Edge edge : graph.getEdges()) {
                if
(Line2D.linesIntersect(edge.getNodes().get(0).getView().getPosition().x,
edge.getNodes().get(0).getView().getPosition().y,
                        edge.getNodes().get(1).getView().getPosition().x,
edge.getNodes().get(1).getView().getPosition().y,
                        deleteData.firstPoint.x, deleteData.firstPoint.y,
                        deleteData.secondPoint.x, deleteData.secondPoint.y)) {
                    edgesToDelete.add(edge);
                }
            }

            if (!edgesToDelete.isEmpty()) {

```



```

        GraphCaretaker.push(graph.save());
    }

    for (Edge edge : edgesToDelete) {
        edge.destroy();
    }

    graph.removeAll(edgesToDelete);
    break;
}
}

public void mouseDragged(MouseEvent mouseEvent) {
    if (SwingUtilities.isMiddleMouseButton(mouseEvent)) {
        double xMotion = mouseEvent.getX() - oldDragPoint.getX();
        double yMotion = mouseEvent.getY() - oldDragPoint.getY();

        for(Node node : graph.getNodes()) {
            Point2D.Double oldPoint2D = node.getView().getPosition();
            node.getView().moveTo(new
Point2D.Double(oldPoint2D.getX()+xMotion, oldPoint2D.getY()+yMotion));
        }

        oldDragPoint = mouseEvent.getPoint();
    }
    else {
        switch (graphState) {
            case CREATE_NODE:
                break;
            case CONNECT_NODE:

```

```

        break;
    case MOVE_NODE:
        if (draggData.getIsDragg()) {
            draggData.moveNodeIfGrabed(mouseEvent.getPoint());
        }

        break;
    case NOTHING:
        break;
    case DELETE_NODE:
        deleteData.secondPoint = mouseEvent.getPoint();
        break;
    }
}
}
}

```

```

private static class DraggData {
    private boolean isDragg;
    private Node node;

    public boolean getIsDragg() {return isDragg;}

    public DraggData() {
        clear();
    }

    public void grabNodeOnMousePos(Point mousePosition, ArrayList<Node>
nodes) {
        for (Node node : nodes) {
            if (node.getView().getBoundingRect().contains(mousePosition)) {

```

```

        isDragg = true;
        this.node = node;
        break;
    }
}

}

public void moveNodeIfGrabed(Point mousePosition) {
    if (isDragg) {
        node.getView().moveTo(new Point2D.Double(mousePosition.getX(),
mousePosition.getY()));
    }
}

public void clear() {
    isDragg = false;
    node = null;
}
}

private static class DeleteData {
    private Point firstPoint;
    private Point secondPoint;
    private boolean cutting;

    public DeleteData() {
        firstPoint = null;
        secondPoint = null;
        cutting = false;
    }
}

```

```

public void setFirstPoint(Point point) {
    firstPoint = point;
    cutting = true;
}

public void setSecondPoint(Point point) {
    secondPoint = point;
    cutting = false;
}
}

public static class ConnectData {
    private Node firstNode;
    private Node secondNode;
    private boolean edgeCreated;
    public static boolean IS_DIRECTED_CONNECTION = false;

    public ConnectData() {
        firstNode = null;
        secondNode = null;
        edgeCreated = false;
    }

    private void addNode(Node node) {
        if (secondNode != null) {
            return;
        }
        else if (firstNode == null) {
            node.getView().setColor(DrawNode.SELECTED_COLOR);

```

```

        firstNode = node;
    }
    else {
        if (node != firstNode) {
            node.getView().setColor(DrawNode.SELECTED_COLOR);
            secondNode = node;
            edgeCreated = true;
        }
    }
}

```

```

private void addNodeOnMousePos(Point mousePosition, ArrayList<Node>
nodes) {
    for (Node node : nodes) {
        if (node.getView().getBoundingRect().contains(mousePosition)) {
            addNode(node);
            break;
        }
    }
}

```

```

private Edge getEdgeIfNotExists() {
    if (!firstNode.getNeighbours().contains(secondNode)) {
        Edge    newEdge    =    new    Edge(firstNode,    secondNode,
IS_DIRECTED_CONNECTION);

```

```

        clear();

```

```

        return newEdge;

```

```

    } else {

```

```

        secondNode.getView().setColor(DrawNode.BASIC_COLOR);
        secondNode = null;
        edgeCreated = false;

        return null;
    }
}

public void clear() {
    if (firstNode != null)
        firstNode.getView().setColor(DrawNode.BASIC_COLOR);
    if (secondNode != null)
        secondNode.getView().setColor(DrawNode.BASIC_COLOR);

    firstNode = null;
    secondNode = null;
    edgeCreated = false;
}
}

package Shapes;

import java.awt.*;

@FunctionalInterface
public interface Drawable {
    void draw(Graphics2D g);
}

```

```

package Shapes;

import java.awt.geom.Rectangle2D;

@FunctionalInterface
public interface Movable {
    Rectangle2D.Double getBoundingRect();
}

package Snapshots;

import Managers.GraphEventManager;
import Widgets.Button;

import java.util.ArrayDeque;

public class GraphCaretaker {
    private static ArrayDeque<Snapshot> backDeque = new ArrayDeque<>();
    private static ArrayDeque<Snapshot> frontDeque = new ArrayDeque<>();
    public static final int STACK_MAX_DEPTH = 50;
    public static Button button1;
    public static Button button2;

    public static void push(Snapshot snapshot) {
        backDeque.add(snapshot);
        frontDeque.clear();

        if (backDeque.size() == STACK_MAX_DEPTH) {
            backDeque.removeFirst();
        }
    }
}

```

```

        button1.setEnabled(true);
        button2.setEnabled(false);
    }

    public static boolean isEmpty() {
        return backDeque.isEmpty();
    }

    public static boolean isFull() {
        return frontDeque.isEmpty();
    }

    public static Snapshot pop() {
        if (backDeque.size() == 1) {
            button1.setEnabled(false);
        }

        button2.setEnabled(true);

        if (!backDeque.isEmpty()) {
            Snapshot snapshot = backDeque.removeLast();
            frontDeque.add(GraphEventManager.getInstance().getGraph().save());
            return snapshot;
        }
        else {
            return null;
        }
    }

    public static Snapshot poll() {

```



```

        if (frontDeque.size() == 1) {
            button2.setEnabled(false);
        }

        button1.setEnabled(true);

        if (!frontDeque.isEmpty()) {
            Snapshot snapshot = frontDeque.removeLast();
            backDeque.add(GraphEventManager.getInstance().getGraph().save());
            return snapshot;
        }
        else {
            return null;
        }
    }
}

```

```

package Snapshots;

```

```

import graphComponents.Edge;
import graphComponents.Node;

```

```

import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Set;

```

```

public class Snapshot {
    private ArrayList<Node> singleNodes = new ArrayList<>();
    private ArrayList<Edge> edges = new ArrayList<>();

```

```

public Snapshot(ArrayList<Node> graphNodes, ArrayList<Edge> graphEdges) {
    HashMap<Node, Node> checkedNodes = new HashMap<>();

    for (Node node : graphNodes) {
        if (node.getEdges().isEmpty()) {
            singleNodes.add(new Node(node));
        }
    }

    for (Edge edge : graphEdges) {
        Edge newEdge = new Edge(edge, checkedNodes);
        checkedNodes.put(edge.getNodes().get(0), newEdge.getNodes().get(0));
        checkedNodes.put(edge.getNodes().get(1), newEdge.getNodes().get(1));

        edges.add(newEdge);
    }
}

public ArrayList<Node> getSingleNodes() {
    return singleNodes;
}

public ArrayList<Edge> getEdges() {
    return edges;
}
}

package ViewCompomemnts;

import java.awt.*;
import java.awt.geom.Point2D;

```

```

public class DrawDirectedEdge extends DrawEdge {
    private static final double POLYGON_ANGLE = (Math.PI / 12);
    private static final double POLYGON_SIZE = 20D;

    public DrawDirectedEdge(DrawNode sourceNode, DrawNode destNode) {
        super(sourceNode, destNode);
    }

    public DrawDirectedEdge(DrawEdge other, DrawNode sourceNode, DrawNode
destNode) {
        super(other, sourceNode, destNode);
    }

    @Override
    public void draw(Graphics2D g) {
        super.draw(g);

        Point2D.Double[] polygonPoints = new Point2D.Double[3];
        Point2D.Double vector = new Point2D.Double(sourcePosition.x -
destPosition.x, sourcePosition.y - destPosition.y);
        double vectorLength = Math.sqrt(Math.pow(vector.x, 2) + Math.pow(vector.y,
2));
        Point2D.Double unitVector = new Point2D.Double(vector.x / vectorLength,
vector.y / vectorLength);
        unitVector.x *= POLYGON_SIZE * DrawNode.scale;
        unitVector.y *= POLYGON_SIZE * DrawNode.scale;

        polygonPoints[0] = (Point2D.Double) destPosition.clone();
    }
}

```

```

        polygonPoints[1] = new Point2D.Double(unitVector.x *
Math.cos(POLYGON_ANGLE) - unitVector.y * Math.sin(POLYGON_ANGLE) +
destPosition.x,
        unitVector.x * Math.sin(POLYGON_ANGLE) + unitVector.y *
Math.cos(POLYGON_ANGLE) + destPosition.y);
        polygonPoints[2] = new Point2D.Double(unitVector.x * Math.cos(-
POLYGON_ANGLE) - unitVector.y * Math.sin(-POLYGON_ANGLE) +
destPosition.x,
        unitVector.x * Math.sin(-POLYGON_ANGLE) + unitVector.y *
Math.cos(-POLYGON_ANGLE) + destPosition.y);

        int xs[] = { (int)polygonPoints[0].x, (int)polygonPoints[1].x,
(int)polygonPoints[2].x };
        int ys[] = { (int)polygonPoints[0].y, (int)polygonPoints[1].y,
(int)polygonPoints[2].y };

        g.fillPolygon(xs, ys, 3);
    }
}

package ViewCompomemnts;

import Shapes.Drawable;

import java.awt.*;
import java.awt.geom.Point2D;

public class DrawEdge implements Drawable {
    protected Color color;
    private DrawNode sourceNode;
    private DrawNode destNode;

```

```

protected Point2D.Double sourcePosition;
protected Point2D.Double destPosition;

public static final Color BASIC_COLOR = Color.darkGray;
public static final Color VISITED_COLOR = Color.magenta;

public DrawEdge(DrawNode sourceNode, DrawNode destNode, Color color) {
    this.sourceNode = sourceNode;
    this.destNode = destNode;
    this.color = color;
    updateOffsetVector();
}

public DrawEdge(DrawEdge other, DrawNode sourceNode, DrawNode
destNode) {
    this.color = other.color;
    this.sourceNode = sourceNode;
    this.destNode = destNode;
    updateOffsetVector();
}

public Color getColor() {
    return color;
}

public DrawEdge(DrawNode sourceNode, DrawNode destNode) {
    this(sourceNode, destNode, BASIC_COLOR);
}

private Point2D.Double createOffsetVector(DrawNode sourceNode, DrawNode
destNode) {

```

```

        Point2D.Double vector = new Point2D.Double(destNode.getPosition().x -
sourceNode.getPosition().x,
            destNode.getPosition().y - sourceNode.getPosition().y);
        double vectorLength = Math.sqrt(Math.pow(vector.x, 2) + Math.pow(vector.y,
2));
        vector.x /= vectorLength;
        vector.y /= vectorLength;
        vector.x *= DrawNode.BASIC_RADIUS * DrawNode.scale;
        vector.y *= DrawNode.BASIC_RADIUS * DrawNode.scale;

        return vector;
    }

    private void updateOffsetVector() {
        Point2D.Double offsetVector = createOffsetVector(sourceNode, destNode);

        this.sourcePosition = new Point2D.Double(sourceNode.getPosition().x +
offsetVector.x,
            sourceNode.getPosition().y + offsetVector.y);
        this.destPosition = new Point2D.Double(destNode.getPosition().x -
offsetVector.x,
            destNode.getPosition().y - offsetVector.y);
    }

    public void setColor(Color color) {
        this.color = color;
    }

    public void transpose() {
        DrawNode tmpNode = sourceNode;

```

```

        sourceNode = destNode;
        destNode = tmpNode;
    }

```

```

@Override
public void draw(Graphics2D g) {
    updateOffsetVector();
    g.setColor(color);
    g.drawLine((int)sourcePosition.x, (int)sourcePosition.y, (int)destPosition.x,
(int)destPosition.y);
}
}

```

```

package ViewCompomemnts;

```

```

import Managers.GraphEventManager;
import Shapes.Drawable;
import graphComponents.Edge;
import graphComponents.Graph;
import graphComponents.Node;

```

```

import java.awt.*;

```

```

public class DrawGraph implements Drawable {
    private Graph graph;

    public DrawGraph() {
        this.graph = new Graph();
        GraphEventManager.getInstance().setGraph(graph);
    }

    public DrawGraph(Graph graph){

```

```

    this.graph = graph;
    GraphEventManager.getInstance().setGraph(graph);
}

public void resetColors() {
    for (Node node : graph.getNodes()) {
        node.getView().setColor(DrawNode.BASIC_COLOR);
    }

    for (Edge edge : graph.getEdges()) {
        edge.getView().setColor(DrawEdge.BASIC_COLOR);
    }
}

@Override
public void draw(Graphics2D g) {
    for (Node node : graph.getNodes()) {
        node.getView().draw(g);
    }

    for (Edge edge : graph.getEdges()) {
        edge.getView().draw(g);
    }
}

public void print(Graphics2D g) {
    for (Node node : graph.getNodes()) {
        node.getView().print(g);
    }
}

```



```

}

package ViewCompomemnts;

import Shapes.Drawable;
import Shapes.Movable;

import java.awt.*;
import java.awt.geom.*;

public class DrawNode implements Drawable, Movable {
    private Point2D.Double position;
    private Color color;
    private int radius;
    private String name;
    public static double scale = 1D;
    public static final Color BASIC_COLOR = Color.white;
    public static final Color SELECTED_COLOR = Color.green;
    public static final Color VISITED_COLOR = Color.magenta;
    public static final int BASIC_RADIUS = 15;
    public static final String DEFAULT_NAME = "";

    public DrawNode(Point2D.Double position, Color color, int radius, String name)
    {
        this.position = position;
        this.color = color;
        this.radius = radius;
        this.name = name;
    }

    public DrawNode(DrawNode other) {

```

```

        this.position      =      new      Point2D.Double(other.getPosition().x,
other.getPosition().y);
        this.color = other.color;
        this.radius = other.radius;
        this.name = other.name;
    }

    public DrawNode(Point2D.Double position, Color color) {
        this(position, color, BASIC_RADIUS, DEFAULT_NAME);
    }

    public DrawNode(Point2D.Double position) {
        this(position, BASIC_COLOR, BASIC_RADIUS, DEFAULT_NAME);
    }

    public DrawNode(Point2D.Double position, String name) {
        this(position, BASIC_COLOR, BASIC_RADIUS, name);
    }

    public void moveTo(Point2D.Double newPosition) {
        position = newPosition;
    }

    public void setColor(Color color) {
        this.color = color;
    }

    public Color getColor() {
        return color;
    }

```

```

public Point2D.Double getPosition() {
    return position;
}

public String getName(){ return name; }

@Override
public void draw(Graphics2D g) {
    g.setColor(new Color(73, 73, 73));
    g.fillOval((int)(position.x - radius*scale + 2*scale), (int)(position.y -
radius*scale + 2*scale), (int)(2 * radius*scale ), (int)(2 * radius*scale));

    Point2D center = new Point2D.Float((int)(position.x -4*scale ) ,
(int)(position.y -4*scale ));
    float[] dist = {0.3f, 0.9f};
    Color[] colors = {color.brighter(), color.darker()};
    RadialGradientPaint p =
        new RadialGradientPaint(center, (int)(radius*scale),
            dist, colors,
            MultipleGradientPaint.CycleMethod.NO_CYCLE);
    g.setPaint(p);

    g.fillOval((int)(position.x - radius*scale), (int)(position.y - radius*scale),
(int)(2 * radius*scale), (int)(2 * radius*scale));

    g.setColor(Color.darkGray);
    g.drawOval((int)(position.x - radius*scale), (int)(position.y - radius*scale),
(int)(2 * radius*scale), (int)(2 * radius*scale));
}

public void print(Graphics2D g) {

```

```

        g.setFont(new Font("Trebuchet MS", Font.BOLD, 12));
        g.setColor(Color.black);
        g.drawString(name,(int)(position.x - radius * scale),(int)(position.y - radius *
scale));
    }

```

```

@Override
public Rectangle2D.Double getBoundingRect() {
    return new Rectangle2D.Double(position.x - radius*scale, position.y -
radius*scale, 2 * radius*scale, 2 * radius*scale);
}
}

```

```

package Widgets;

```

```

import javax.swing.JButton;
import java.awt.Color;

```

```

public class Button extends JButton{
    private ButtonState state;

    public void changeState() {
        if (state == ButtonState.ACTIVE) {
            this.setBackground(new Color(170, 170, 199));
        } else {
            this.setBackground(new Color(219, 232, 254));
        }
    }
}

```

```

public void setState(ButtonState state) {

```

```

        this.state = state;
    }

    ButtonState getState() {
        return state;
    }
}

package Widgets;

public enum ButtonState {
    ACTIVE,
    INACTIVE
}

package Widgets;

import Managers.GraphEventManager;
import ViewCompomemnts.DrawGraph;
import graphComponents.GraphStates;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.ArrayList;
import java.awt.geom.*;

public class GraphicsPanel extends JPanel {
    private static final long serialVersionUID = 1L;
    private DrawGraph graph;
    private JPanel panelNodesEdges;

```

```

public GraphicsPanel() {
    graph = new DrawGraph();
    setBackground(new Color(205, 210, 255));

    addMouseMotionListener(new MouseMotionAdapter() {
        @Override
        public void mouseDragged(MouseEvent mouseEvent) {
            super.mouseDragged(mouseEvent);
            GraphEventManager.getInstance().mouseDragged(mouseEvent);

            if (SwingUtilities.isMiddleMouseButton(mouseEvent)) {
                setCursor(Cursor.getPredefinedCursor(Cursor.MOVE_CURSOR));
            }

            repaint();
        }
    });

    addMouseListener(new MouseAdapter() {
        @Override
        public void mousePressed(MouseEvent mouseEvent) {
            super.mousePressed(mouseEvent);
            GraphEventManager.getInstance().mousePressed(mouseEvent);

            if (GraphEventManager.getInstance().getState() ==
GraphStates.MOVE_NODE) {
                setCursor(Cursor.getPredefinedCursor(Cursor.MOVE_CURSOR));
            }

            updatePanelNodesEdges();
        }
    });
}

```

```

        repaint();
    }

    @Override
    public void mouseReleased(MouseEvent mouseEvent) {
        super.mouseReleased(mouseEvent);
        GraphEventManager.getInstance().mouseReleased(mouseEvent);
        setCursor(Cursor.getPredefinedCursor(Cursor.DEFAULT_CURSOR));

        updatePanelNodesEdges();

        repaint();
    }
});

addMouseWheelListener(new MouseAdapter() {
    @Override
    public void mouseWheelMoved(MouseWheelEvent e) {
        Point2D.Double center = new Point2D.Double(getWidth()/2.0,
getHeight()/2.0);

        GraphEventManager.getInstance().mouseWheelMoved(e, center);

        repaint();
    }
});
}

public void setGraphState(GraphStates state) {
    GraphEventManager.getInstance().setState(state);
}

```

```

    }

    public void setGraph(DrawGraph graph){
        this.graph = graph;
        this.updatePanelNodesEdges();
        repaint();
    }

    public void setGraphBasicColor() {
        graph.resetColors();
    }

    @Override
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        Graphics2D g2 = (Graphics2D) g;
        g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
RenderingHints.VALUE_ANTIALIAS_ON);
        g2.setFont(new Font("Times New Roman", Font.BOLD + Font.ITALIC, 15));

        graph.draw(g2);
        graph.print(g2);

        if (GraphEventManager.getInstance().isCutting()) {
            ArrayList<Point> pair = GraphEventManager.getInstance().getScissors();

            g.setColor(Color.red);
            g.drawLine(pair.get(0).x, pair.get(0).y, pair.get(1).x, pair.get(1).y);
        }
    }
}

```



```

public void updatePanelNodesEdges() {
    JLabel labelNodes = (JLabel)panelNodesEdges.getComponents()[0];
    JLabel labelEdges = (JLabel)panelNodesEdges.getComponents()[1];

    labelNodes.setText("Nodes:                " +
GraphEventManager.getInstance().getNodesCount());
    labelEdges.setText("Edges:                " +
GraphEventManager.getInstance().getEdgesCount());
}

public void setPanelNodesEdges(JPanel panelNodesEdges) {
    this.panelNodesEdges = panelNodesEdges;
}
}

```

```

package algorithmComponents;

```

```

import graphComponents.Edge;
import graphComponents.Graph;
import graphComponents.Node;

```

```

import javax.swing.*;
import java.awt.*;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.concurrent.Callable;

```

```

public abstract class Algorithm {
    protected boolean initialized = false;
    protected boolean finished = false;
    protected int currentStep = 0;

```

```

protected Graph graph = null;
protected ArrayList<StepsColorDataBase> stepsColorDataBase = new
ArrayList<>();
protected int stepsColorDataBaseIterator = 0;

public abstract void initialize(Node node);
public abstract void initialize(Edge edge);

public boolean isInitialized() {
    return initialized;
}

public abstract void doRun();
public abstract void doForwardStep();
public abstract void doBackwardStep();

public void setGraph(Graph graph) {
    this.graph = graph;
}

public AlgorithmState getState() {
    int step = 0;

    if (stepsColorDataBaseIterator == 0) {
        step = 0;
    }
    else if (finished && (stepsColorDataBaseIterator ==
stepsColorDataBase.size() - 1)) {
        step = 2;
    }
}

```

```

        else {
            step = 1;
        }

        return new AlgorithmState(initialized, step,
stepsColorDataBase.get(stepsColorDataBaseIterator));
    }

    public void reset() {
        stepsColorDataBaseIterator = 0;
        stepsColorDataBase.clear();

        initialized = false;
        finished = false;
        currentStep = 0;
    }

    public class AlgorithmState {
        public boolean initialized;
        public int step; //0 - first; 1 - middle; 2 - last
        public StepsColorDataBase stepsColorDataBase;

        public AlgorithmState(boolean initialized, int step, StepsColorDataBase
stepsColorDataBase) {
            this.initialized = initialized;
            this.step = step;
            this.stepsColorDataBase = stepsColorDataBase;
        }
    }

    public class StepsColorDataBase {

```

```

protected HashMap<Node, Color> nodeColorHashMap = new HashMap<>();
protected HashMap<Edge, Color> edgeColorHashMap = new HashMap<>();
public String stepName = "";

public StepsColorDataBase(Graph graph, String stepName) {
    for (Node node : graph.getNodes()) {
        nodeColorHashMap.put(node, node.getView().getColor());
    }

    for (Edge edge : graph.getEdges()) {
        edgeColorHashMap.put(edge, edge.getView().getColor());
    }

    this.stepName = stepName;
}

public void resetColors() {
    for (Node node : nodeColorHashMap.keySet()) {
        node.getView().setColor(nodeColorHashMap.get(node));
    }

    for (Edge edge : edgeColorHashMap.keySet()) {
        edge.getView().setColor(edgeColorHashMap.get(edge));
    }
}

}

package algorithmComponents;

```

```

import Widgets.Button;

import java.util.ArrayList;

public class AlgorithmButtons {
    public ArrayList<Button> buttons = new ArrayList<>();

    public AlgorithmButtons(Button b1, Button b2, Button b3, Button b4, Button b5,
Button b6) {
        buttons.add(b1);
        buttons.add(b2);
        buttons.add(b3);
        buttons.add(b4);
        buttons.add(b5);
        buttons.add(b6);
    }
}

package algorithmComponents;

import Managers.AlgorithmEventManager;

import java.util.ArrayList;
import java.util.HashMap;

public class Algorithms {
    private static HashMap<String, Algorithm> algorithmHashMap = new
HashMap<>();
    private static int delay = 500;

    public Algorithms() {}
}

```

```

public static void init() {
    algorithmHashMap.put("DFS", new DFSAlgorithm());
    algorithmHashMap.put("Kosaraju", new KosarajuAlgorithm());

    AlgorithmEventManager.getInstance().setAlgorithm(algorithmHashMap.get("DFS
"));
}

public static void selectAlgorithmByName(String name) {

    AlgorithmEventManager.getInstance().setAlgorithm(algorithmHashMap.get(name
));
    AlgorithmEventManager.getInstance().setDelay(delay);
}

public static void setDelay(int _delay) {
    delay = _delay;
    AlgorithmEventManager.getInstance().setDelay(delay);
}
}

package algorithmComponents;

import ViewCompomemnts.DrawEdge;
import ViewCompomemnts.DrawNode;
import graphComponents.Edge;
import graphComponents.Node;

```

```

import java.awt.*;
import java.util.*;
import java.util.concurrent.Callable;

public class DFSAlgorithm extends Algorithm {
    private static final int INIT_STEPS_COUNT = 1;

    private Node startNode;
    private Node currentNode;

    private Stack<Node> stack = new Stack<>();
    private HashMap<Node, AlgorithmNodeStructure> nodes = new HashMap<>();
    private ArrayList<Edge> edges = new ArrayList<>();

    public DFSAlgorithm() {
        this.startNode = null;
        this.currentNode = null;
    }

    private void checkInitialization() {
        if (currentStep == INIT_STEPS_COUNT) {
            initialized = true;
            createStructures();
            doRun();
        }
    }

    private void createStructures() {
        for (Node node : graph.getNodes()) {
            nodes.put(node, new AlgorithmNodeStructure());
        }
    }
}

```

```

    }
}

@Override
public void initialize(Node node) {
    switch (currentStep) {
        case 0:
            this.startNode = node;
            break;
    }

    currentStep++;
    checkInitialization();
}

@Override
public void initialize(Edge edge) {
    currentStep++;
    checkInitialization();
}

@Override
public void doRun() {
    currentNode = startNode;
    nodes.get(currentNode).setVisited(true);
    stack.push(startNode);
    updatePictures();
    stepsColorDataBase.add(new StepsColorDataBase(graph, "first DFS step"));
}

```



```

@Override
public void doForwardStep() {
    if (stepsColorDataBaseIterator != (stepsColorDataBase.size() - 1)) {
        stepsColorDataBaseIterator++;
        stepsColorDataBase.get(stepsColorDataBaseIterator).resetColors();
        return;
    }

    if (finished) {
        return;
    }

    doDFSStep();
    updatePictures();
    stepsColorDataBase.add(new StepsColorDataBase(graph,
stepsColorDataBaseIterator + 1 + "th DFS step"));
    stepsColorDataBaseIterator++;
}

private void doDFSStep() {
    currentNode = stack.peek();
    nodes.get(currentNode).setVisited(true);

    ArrayList<Node> neighbours = currentNode.getSmartNeighbours();

    for (Node neighbour : neighbours) {
        if (!nodes.get(neighbour).isVisited) {
            edges.add(currentNode.getEdge(neighbour));
            stack.push(neighbour);
        }
    }
    return;
}

```

```

    }
}

stack.pop();

if (stack.isEmpty()) {
    finished = true;
}
}

@Override
public void doBackwardStep() {
    if (stepsColorDataBaseIterator != 0) {
        stepsColorDataBaseIterator--;
        stepsColorDataBase.get(stepsColorDataBaseIterator).resetColors();
    }
}

private void updatePictures() {
    for (Node node : nodes.keySet()) {
        if (nodes.get(node).isVisited) {
            node.getView().setColor(DrawNode.VISITED_COLOR);
        }
    }

    for (Edge edge : edges) {
        edge.getView().setColor(DrawEdge.VISITED_COLOR);
    }

    if (!stack.isEmpty()) {

```

```

        stack.peek().getView().setColor(DrawNode.SELECTED_COLOR);
    }
}

```

```

@Override
public void reset() {
    super.reset();

    stack.clear();
    nodes.clear();
    edges.clear();
}

```

```

private class AlgorithmNodeStructure {
    private boolean isVisited;

    public AlgorithmNodeStructure() {
        this.isVisited = false;
    }

    public void setVisited(boolean isVisited) {
        this.isVisited = isVisited;
    }
}

```

```

private class AlgorithmEdgeStructure {
    private boolean isVisited;

    public AlgorithmEdgeStructure() {
        this.isVisited = false;
    }
}

```

```

    }

    public void setVisited(boolean isVisited) {
        this.isVisited = isVisited;
    }
}

}

package algorithmComponents;

import ViewCompomemnts.DrawEdge;
import ViewCompomemnts.DrawNode;
import graphComponents.Edge;
import graphComponents.Graph;
import graphComponents.Node;

import java.util.*;
import java.util.concurrent.Callable;
import java.awt.Color;

public class KosarajuAlgorithm extends Algorithm {
    private static final int INIT_STEPS_COUNT = 1;

    private Node startNode;
    private Node currentNode;

    private int UnVisitedCounter;
    private boolean edgesTransposed = false;
    private Color colorOfComponent;
    private String stageOfAlgorithm = "doDFSstep";
    private Random random = new Random(System.currentTimeMillis());

```

```

private HashMap<Node, NodeWrapper> nodes = new HashMap<>();
private ArrayList<Edge> edges = new ArrayList<>();

private Stack<Node> stack = new Stack<>();
private Stack<Node> timeOutList = new Stack<>();

public KosarajuAlgorithm() {
    this.startNode = null;
    this.currentNode = null;
    this.colorOfComponent = new Color(random.nextInt(256),
random.nextInt(256), random.nextInt(256));
}

private void wrappNodes() {
    for (Node node : graph.getNodes()) {
        nodes.put(node, new NodeWrapper());
    }
}

@Override
public void initialize(Node node) {
    switch (currentStep) {
        case 0:
            this.startNode = node;
            break;
    }

    currentStep++;
    checkInitialization();
}

```

```

private void checkInitialization() {
    if (currentStep == INIT_STEPS_COUNT) {
        initialized = true;
        wrappNodes();
        UnVisitedCounter = nodes.size();
        doRun();
    }
}

@Override
public void initialize(Edge edge) {

}

@Override
public void doRun() {
    currentNode = startNode;
    nodes.get(currentNode).setVisited(true);
    stack.push(startNode);
    updatePictures();
    stepsColorDataBase.add(new KosarajuStepsColorDataBase(graph, "first DFS
step"));
    UnVisitedCounter--;
}

@Override
public void doForwardStep() {
    if (stepsColorDataBaseIterator != (stepsColorDataBase.size() - 1)) {
        stepsColorDataBaseIterator++;
    }
}

```

```

stepsColorDataBase.get(stepsColorDataBaseIterator).resetColors();

    if
(((KosarajuStepsColorDataBase)stepsColorDataBase.get(stepsColorDataBaseIterat
or)).transposed) {
        transpose();
    }

    return;
}

if (finished) {
    return;
}

String currentStageOfAlgorithm = stageOfAlgorithm;

switch (stageOfAlgorithm) {
    case "doDFSstep":
        doDFSstep();
        break;
    case "doTransposeStep":
        doTransposeStep();
        doInitializationSearchComponentsStep();
        break;
    case "doInitializationSearchComponentsStep":
        break;
    case "doSearchComponentsStep":
        doSearchComponentsStep();
        break;
}

```

```

    }

    System.out.println(stageOfAlgorithm);
    updatePictures();
    stepsColorDataBase.add(new KosarajuStepsColorDataBase(graph,
        currentStageOfAlgorithm.equals("doTransposeStep"),
        stepsColorDataBaseIterator + 1 + "th " + currentStageOfAlgorithm));
    stepsColorDataBaseIterator++;
}

@Override
public void doBackwardStep() {
    if (stepsColorDataBaseIterator != 0) {
        if (stageOfAlgorithm.equals("doDFSstep")) {
            UnVisitedCounter++;
        }

        if
        (((KosarajuStepsColorDataBase)stepsColorDataBase.get(stepsColorDataBaseIterat
        or)).transposed) {
            transpose();
        }

        System.out.println(stepsColorDataBaseIterator);
        stepsColorDataBaseIterator--;
        stepsColorDataBase.get(stepsColorDataBaseIterator).resetColors();
    }
}

private void doInitializationSearchComponentsStep() {

```



```

    wrappNodes();

    currentNode = timeOutList.pop();
    nodes.get(currentNode).setVisited(true);
    stack.push(currentNode);

    nodes.get(currentNode).color = colorOfComponent;
    stageOfAlgorithm = "doSearchComponentsStep";

    updatePictures();
}

private void doDFSstep() {
    if (stack.isEmpty()) {
        if (UnVisitedCounter > 0) {
            for (Node node : graph.getNodes()) {
                if (!nodes.get(node).isVisited) {
                    currentNode = node;
                    stack.push(currentNode);
                    return;
                }
            }
        }

        stageOfAlgorithm = "doTransposeStep";
        return;
    }

    currentNode = stack.peek();
    nodes.get(currentNode).setVisited(true);

```

```

for (Node neighbour: currentNode.getSmartNeighbours()) {
    if (!nodes.get(neighbour).isVisited) {
        UnVisitedCounter--;
        edges.add(currentNode.getEdge(neighbour));
        stack.push(neighbour);
        return;
    }
}

timeOutList.push(currentNode);
stack.pop();
}

private void updatePictures() {
    if (stageOfAlgorithm.equals("doSearchComponentsStep")) {
        for (Node node : nodes.keySet()) {
            NodeWrapper nodeWrapper = nodes.get(node);
            node.getView().setColor(nodeWrapper.color);
        }
    }
    else {
        for (Node node : nodes.keySet()) {
            if (nodes.get(node).isVisited) {
                node.getView().setColor(DrawNode.VISITED_COLOR);
            }
        }

        for (Edge edge : edges) {
            edge.getView().setColor(DrawEdge.VISITED_COLOR);
        }
    }
}

```

```

    }

    if (!stack.isEmpty()) {
        stack.peek().getView().setColor(DrawNode.SELECTED_COLOR);
    }
}

private void transpose() {
    for(Edge edge: graph.getEdges()) {
        edge.transpose();
    }

    edgesTransposed = !edgesTransposed;
}

private void doTransposeStep() {
    transpose();

    updatePictures();
    stageOfAlgorithm = "doInitializationSearchComponentsStep";
}

private void doSearchComponentsStep() {
    if (timeOutList.isEmpty()) {
        System.out.println("FINISH!!!");
        finished = true;
        return;
    }

    if (stack.empty()) {

```

```
        colorOfComponent      =      new      Color(random.nextInt(256),
random.nextInt(256), random.nextInt(256));
```

```
        currentNode = timeOutList.peek();
```

```
        while(nodes.get(currentNode).isVisited) {
            timeOutList.pop();
```

```
            if (timeOutList.isEmpty()) {
                return;
            }

```

```
            currentNode = timeOutList.peek();
        }
```

```
        nodes.get(currentNode).setVisited(true);
        nodes.get(currentNode).color = colorOfComponent;
        stack.push(currentNode);
        return;
    }
```

```
currentNode = stack.peek();
```

```
for (Node neighbour: currentNode.getSmartNeighbours()) {
    if (!nodes.get(neighbour).isVisited) {
        currentNode = neighbour;
        nodes.get(currentNode).setVisited(true);
        nodes.get(currentNode).color = colorOfComponent;
        stack.push(currentNode);
        return;
    }
}
```

```

    }

    stack.pop();
    updatePictures();
}

@Override
public void reset() {
    super.reset();

    if (edgesTransposed) {
        transpose();
    }

    stack.clear();
    edges.clear();
    nodes.clear();
    timeOutList.clear();
    stageOfAlgorithm = "doDFSstep";
}

private class NodeWrapper {
    private boolean isVisited;
    public Color color = Color.MAGENTA;

    public NodeWrapper() {
        this.isVisited = false;
    }

    public void setVisited(boolean isVisited) {
        this.isVisited = isVisited;
    }
}

```

```
    }  
}
```

```
public class KosarajuStepsColorDataBase extends StepsColorDataBase {  
    public boolean transposed = false;  
  
    public KosarajuStepsColorDataBase(Graph graph, boolean transposed, String  
stepNmae) {  
        super(graph, stepNmae);  
        this.transposed = transposed;  
    }  
  
    public KosarajuStepsColorDataBase(Graph graph, String stepName) {  
        this(graph, false, stepName);  
    }  
  
    @Override  
    public void resetColors() {  
        super.resetColors();  
    }  
}  
}
```

```
package graphComponents;
```

```
import ViewCompomemnts.DrawDirectedEdge;  
import ViewCompomemnts.DrawEdge;
```

```
import java.util.ArrayList;  
import java.util.HashMap;
```

```

import java.util.HashSet;

public class Edge {
    private Node sourceNode;
    private Node destNode;
    private boolean isDirected = false;
    private DrawEdge edgeView;

    public Edge(Node sourceNode, Node destNode, DrawEdge edgeView) {
        this.sourceNode = sourceNode;
        this.destNode = destNode;
        this.edgeView = edgeView;
    }

    public Edge(Edge other, HashMap<Node, Node> checkedNodes) {
        if (!checkedNodes.keySet().contains(other.sourceNode)) {
            this.sourceNode = new Node(other.sourceNode);
        }
        else {
            this.sourceNode = checkedNodes.get(other.sourceNode);
        }

        if (!checkedNodes.keySet().contains(other.destNode)) {
            this.destNode = new Node(other.destNode);
        }
        else {
            this.destNode = checkedNodes.get(other.destNode);
        }

        this.isDirected = other.isDirected;
    }

```

```

        this.edgeView = this.isDirected ? new DrawDirectedEdge(other.edgeView,
this.sourceNode.getView(), this.destNode.getView()) :
            new DrawEdge(other.edgeView, this.sourceNode.getView(),
this.destNode.getView());
    }

```

```

    public Edge(Node sourceNode, Node destNode) {
        this(sourceNode, destNode, new DrawEdge(sourceNode.getView(),
destNode.getView()));
    }

```

```

    public Edge(Node sourceNode, Node destNode, boolean isDirected) {
        this(sourceNode, destNode, isDirected ? new
DrawDirectedEdge(sourceNode.getView(), destNode.getView()) :
            new DrawEdge(sourceNode.getView(),
destNode.getView()));
        this.isDirected = isDirected;
    }

```

```

    public Node getNeighbour(Node node) {
        if (node != sourceNode && node != destNode) {
            return null;
        }
        else {
            return (node == sourceNode) ? destNode : sourceNode;
        }
    }

```

```

    public Node getSmartNeighbour(Node node) {
        if (!isDirected) {

```



```

        return getNeighbour(node);
    }

    if (node != sourceNode && node != destNode) {
        return null;
    }
    else {
        return (node == sourceNode) ? destNode : null;
    }
}

public void destroy() {
    sourceNode.removeEdge(this);
    destNode.removeEdge(this);
}

public boolean isDirected() {
    return isDirected;
}

public ArrayList<Node> getNodes() {
    ArrayList<Node> pair = new ArrayList<>();
    pair.add(sourceNode);
    pair.add(destNode);

    return pair;
}

public DrawEdge getView() {
    return edgeView;
}

```

```

    public void transpose() {
        Node tmpNode = sourceNode;
        sourceNode = destNode;
        destNode = tmpNode;

        edgeView.transpose();
    }
}

package graphComponents;

import Snapshots.Snapshot;

import java.util.ArrayList;

public class Graph {
    private ArrayList<Edge> edges = new ArrayList<>();
    private ArrayList<Node> nodes = new ArrayList<>();

    public Graph() {}

    public void add(Node node) {
        nodes.add(node);
    }

    public void add(Edge edge) {
        ArrayList<Node> pair = edge.getNodes();

```

```

Edge e1 = pair.get(0).getEdge(pair.get(1));

if (e1 != null && edge.isDirected() && e1.isDirected()) {
    if (pair.get(0) == e1.getNodes().get(1)) {
        edges.remove(e1);
        e1.destroy();

        Edge newEdge = new Edge(pair.get(0), pair.get(1), false);
        pair.get(0).addEdge(newEdge);
        pair.get(1).addEdge(newEdge);
        edges.add(newEdge);
    }
}

if (e1 == null) {
    pair.get(0).addEdge(edge);
    pair.get(1).addEdge(edge);
    edges.add(edge);
}

}

public void remove(Node node) {
    nodes.remove(node);
}

public void remove(Edge edge) {
    edges.remove(edge);
}

public void removeAll(ArrayList<Edge> edges) {
    this.edges.removeAll(edges);
}

```

```

    }

    public void removeAllEdges() {
        edges.clear();
    }

    public void clear() {
        nodes.clear();
        edges.clear();
    }

    public void connectAllVertices() {
        for (int i = 0; i < nodes.size() - 1; i++) {
            for (int j = i + 1; j < nodes.size(); j++) {
                if (!nodes.get(i).getNeighbours().contains(nodes.get(j))) {
                    Edge edge = new Edge(nodes.get(i), nodes.get(j));

                    nodes.get(i).addEdge(edge);
                    nodes.get(j).addEdge(edge);
                    edges.add(edge);
                }
            }
        }
    }

    public Snapshot save() {
        return new Snapshot(nodes, edges);
    }

    public void restore(Snapshot snapshot) {
        if (snapshot == null) {

```

```

        return;
    }

    nodes.clear();
    edges.clear();

    nodes.addAll(snapshot.getSingleNodes());
    edges.addAll(snapshot.getEdges());

    for (Edge edge : edges) {
        ArrayList<Node> pair = edge.getNodes();

        if (!nodes.contains(pair.get(0))) {
            nodes.add(pair.get(0));
        }

        if (!nodes.contains(pair.get(1))) {
            nodes.add(pair.get(1));
        }

        pair.get(0).addEdge(edge);
        pair.get(1).addEdge(edge);
    }
}

public ArrayList<Node> getNodes() {
    return nodes;
}

public ArrayList<Edge> getEdges() {
    return edges;
}

```

```

    }
}

package graphComponents;

public enum GraphStates {
    CREATE_NODE,
    CONNECT_NODE,
    MOVE_NODE,
    NOTHING,
    DELETE_NODE,
    ALGORITHM
}

package graphComponents;

import ViewCompomemnts.DrawNode;

import java.util.ArrayList;

public class Node {
    private ArrayList<Edge> edges = new ArrayList<>();
    private DrawNode nodeView;

    public Node(DrawNode nodeView) {
        this.nodeView = nodeView;
    }

    public Node(Node other) {
        this.nodeView = new DrawNode(other.getView());
    }
}

```

```

public DrawNode getView() {
    return nodeView;
}

public void addEdge(Edge edge) {
    edges.add(edge);
}

public ArrayList<Edge> getEdges() {
    return edges;
}

public Edge getEdge(Node neighbour) {
    for (Edge edge : edges) {
        if (edge.getNeighbour(this) == neighbour) {
            return edge;
        }
    }

    return null;
}

public void destroy() {
    for (Edge edge : edges) {
        Node neighbour = edge.getNeighbour(this);
        neighbour.getEdges().remove(edge);
    }

    edges.clear();
}

```

```

public void removeEdge(Edge edge) {
    edges.remove(edge);
}

public ArrayList<Node> getNeighbours() {
    ArrayList<Node> nbs = new ArrayList<>();

    for (Edge edge : edges) {
        nbs.add(edge.getNeighbour(this));
    }

    return nbs;
}

public ArrayList<Node> getSmartNeighbours() {
    ArrayList<Node> nbs = new ArrayList<>();

    for (Edge edge : edges) {
        Node neighbour = edge.getSmartNeighbour(this);

        if (neighbour != null) {
            nbs.add(edge.getSmartNeighbour(this));
        }
    }

    return nbs;
}

}

package graphComponents;

```



```

import ViewCompomemts.DrawGraph;
import ViewCompomemts.DrawNode;

import java.awt.geom.Point2D;
import java.util.ArrayList;
import java.util.Random;

public class RandomGraphCreator {
    public static DrawGraph create(int nodesCount, double edgeFrequency, int width,
int height, boolean isCircular) {
        Graph graph = new Graph();
        Random random = new Random(System.currentTimeMillis());

        double angle = (2 * Math.PI) / nodesCount;
        double radius = ((3 * nodesCount) * DrawNode.BASIC_RADIUS) / Math.PI;
        Point2D.Double center = new Point2D.Double(width / 2F, height / 2F);

        for (int i = 0; i < nodesCount; i++) {
            Point2D.Double position = isCircular ? new Point2D.Double(center.x +
radius * Math.cos(angle * i), center.y + radius * Math.sin(angle * i)) :
                new Point2D.Double(Math.random() * width, Math.random() *
height);

            graph.add(new Node(new DrawNode(position, String.valueOf(i))));
        }

        ArrayList<Node> graphNodes = graph.getNodes();

        for (int i = 0; i < nodesCount - 1; i++) {
            for (int j = i + 1; j < nodesCount; j++) {

```

```

        if (Math.random() <= edgeFrequency) {
            Edge newEdge = new Edge(graphNodes.get(i), graphNodes.get(j),
random.nextBoolean());
            graph.add(newEdge);
        }
    }
}

return new DrawGraph(graph);
}
}

```

```

package graphComponents;

```

```

import java.awt.geom.Point2D;
import java.util.ArrayList;
import java.util.Comparator;
import java.util.Stack;

```

```

public class Triangulator {
    public static void triangulate(Graph graph) {
        if (graph.getNodes().size() < 3) {
            return;
        }

        for (Edge edge : graph.getEdges()) {
            ArrayList<Node> pair = edge.getNodes();
            pair.get(0).removeEdge(edge);
            pair.get(1).removeEdge(edge);
        }
    }
}

```

```

graph.removeAllEdges();
ArrayList<Node> nodes = graph.getNodes();
Stack<Edge> stack = new Stack<>();
int count = 2;

nodes.sort((o1, o2) -> {
    double firstLength = Math.sqrt(Math.pow(o1.getView().getPosition().x, 2) +
        Math.pow(o1.getView().getPosition().y, 2));
    double secondLength = Math.sqrt(Math.pow(o2.getView().getPosition().x,
2) +
        Math.pow(o2.getView().getPosition().y, 2));

    if (firstLength < secondLength) {
        return -1;
    }
    else if (firstLength > secondLength ) {
        return 1;
    }

    return 0;
});

stack.push(new Edge(nodes.get(0), nodes.get(1), true));
graph.add(stack.peek());

while (!stack.isEmpty()) {
    Edge currentEdge = stack.pop();
    Node firstNode = currentEdge.getNodes().get(0);
    Node secondNode = currentEdge.getNodes().get(1);

```

```

        Point2D.Double          currentEdgecenter          =          new
Point2D.Double((firstNode.getView().getPosition().x          +
secondNode.getView().getPosition().x) / 2F,
                (firstNode.getView().getPosition().y          +
secondNode.getView().getPosition().y) / 2F );

        nodes.sort((o1, o2) -> {
            double firstLength = Math.sqrt(Math.pow(o1.getView().getPosition().x -
currentEdgecenter.x, 2) +
                Math.pow(o1.getView().getPosition().y - currentEdgecenter.y, 2));
            double          secondLength          =
Math.sqrt(Math.pow(o2.getView().getPosition().x - currentEdgecenter.x, 2) +
                Math.pow(o2.getView().getPosition().y - currentEdgecenter.y, 2));

            if (firstLength < secondLength) {
                return 1;
            }
            else if (firstLength > secondLength) {
                return -1;
            }

            return 0;
        });

        for (Node bidder : nodes) {
            if (bidder == firstNode || bidder == secondNode) {
                continue;
            }

```

```

        if (firstNode.getNeighbours().contains(bidder) &&
secondNode.getNeighbours().contains(bidder)) {
            continue;
        }

        Point2D.Double A = firstNode.getView().getPosition();
        Point2D.Double B = secondNode.getView().getPosition();
        Point2D.Double C = bidder.getView().getPosition();

        double D = 2 * (A.x * (B.y - C.y) + B.x * (C.y - A.y) + C.x * (A.y - B.y));
        double X = ((Math.pow(A.x, 2) + Math.pow(A.y, 2)) * (B.y - C.y) +
            (Math.pow(B.x, 2) + Math.pow(B.y, 2)) * (C.y - A.y) +
            (Math.pow(C.x, 2) + Math.pow(C.y, 2)) * (A.y - B.y)) / D;
        double Y = ((Math.pow(A.x, 2) + Math.pow(A.y, 2)) * (C.x - B.x) +
            (Math.pow(B.x, 2) + Math.pow(B.y, 2)) * (A.x - C.x) +
            (Math.pow(C.x, 2) + Math.pow(C.y, 2)) * (B.x - A.x)) / D;

        Point2D.Double triangleCircleCenter = new Point2D.Double(X, Y);
        double triangleCircleRadius =
Math.sqrt(Math.pow(triangleCircleCenter.x - bidder.getView().getPosition().x, 2) +
            Math.pow(triangleCircleCenter.y - bidder.getView().getPosition().y,
2));

        boolean spyDetected = false;

        for (Node spy : nodes) {
            if (spy == firstNode || spy == secondNode || spy == bidder) {
                continue;
            }

```

```

        double distance = Math.sqrt(Math.pow(triangleCircleCenter.x -
spy.getView().getPosition().x, 2) +
        Math.pow(triangleCircleCenter.y - spy.getView().getPosition().y,
2));

        if (distance <= triangleCircleRadius) {
            spyDetected = true;
            break;
        }
    }

    if (!spyDetected) {
        Edge firstEdge = new Edge(firstNode, bidder, true);
        Edge secondEdge = new Edge(secondNode, bidder, true);
        graph.add(firstEdge);
        graph.add(secondEdge);
        stack.push(firstEdge);
        stack.push(secondEdge);
        break;
    }
}
}
}
}
}

```

```

package algorithmComponents;

```

```

import IO.InputReader;
import Managers.AlgorithmEventManager;
import Managers.GraphEventManager;

```

```

import ViewCompomemts.DrawGraph;
import Widgets.Button;
import graphComponents.GraphStates;
import graphComponents.Node;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import Widgets.GraphicsPanel;

import javax.swing.*;
import java.awt.*;
import java.io.*;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;

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

class KosarajuAlgorithmTest {
    private static GraphicsPanel graphicsPanel = new GraphicsPanel();
    private HashMap<String, Button> buttonHashMap = new HashMap<String,
Button>();
    private JLabel labelAction = new JLabel();
    private String dirKosaraju = "resources/testing/testing_Kosaraju/";

    @BeforeEach
    void simulate_a_frame() {
        String commands[] = {"Запустить алгоритм",
            "Остановить алгоритм",
            "Сделать шаг вперед",
            "Сделать шаг назад",

```

```

        "Отмена",
        "Отмена отмены",
        "Результат",
        "Начало"};

for(int i = 0; i < commands.length; i++ ) {
    Button button = new Button();
    buttonHashMap.put(commands[i], button);
}

Algorithms.init();
GraphEventManager.getInstance().setGraphicsPanel(graphicsPanel);
AlgorithmEventManager.getInstance().setGraphicsPanel(graphicsPanel);
AlgorithmEventManager.getInstance().setDisplay(labelAction);
AlgorithmEventManager.getInstance().setAlgorithmButtons(new
AlgorithmButtons(buttonHashMap.get("Запустить алгоритм"),
    buttonHashMap.get("Остановить алгоритм"),
    buttonHashMap.get("Сделать шаг вперед"),
    buttonHashMap.get("Сделать шаг назад"),
    buttonHashMap.get("Результат"),
    buttonHashMap.get("Начало"))));
buttonHashMap.get("Запустить алгоритм").setEnabled(true);
buttonHashMap.get("Остановить алгоритм").setEnabled(true);
buttonHashMap.get("Сделать шаг вперед").setEnabled(true);
buttonHashMap.get("Сделать шаг назад").setEnabled(false);
buttonHashMap.get("Отмена").setEnabled(false);
buttonHashMap.get("Отмена отмены").setEnabled(false);
buttonHashMap.get("Результат").setEnabled(true);
buttonHashMap.get("Начало").setEnabled(false);
}

```



```

void tuneKosarajuAlgorithm() {
    labelAction.setText("Kosaraju");
    Algorithms.init();
    Algorithms.selectAlgorithmByName("Kosaraju");
    GraphEventManager.getInstance().setState(GraphStates.ALGORITHM);
    AlgorithmEventManager.getInstance().reset();

    AlgorithmEventManager.getInstance().getAlgorithm().setGraph(GraphEventManager.getInstance().getGraph());
    Node selectedNode =
    GraphEventManager.getInstance().getGraph().getNodes().get(0);

    AlgorithmEventManager.getInstance().getAlgorithm().initialize(selectedNode);
    AlgorithmEventManager.getInstance().sendCommand("Результат");
}

ArrayList<ArrayList<String>> parse(String answerFileName) {
    File answerFile = new File(answerFileName);
    String line = null;

    try(BufferedReader reader = new BufferedReader(new
InputStreamReader(new FileInputStream(answerFile)))){
        line = reader.readLine();
    }catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

```

```

String[] strComponents = line.split("\\|");

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

for(String strComponent: strComponents) {
    String[] strNamesOfNodes = strComponent.split(",");
    ArrayList<String> namesOfNodes = new ArrayList<>();
    for(String strNameOfNode: strNamesOfNodes) {
        namesOfNodes.add(strNameOfNode);
    }
    namesOfnodesInComponents.add(namesOfNodes);
}

return namesOfnodesInComponents;
}

Node    getNodeFromUnChecked(HashSet<Node>    unChecked,    String
nameOfNode) {
    for(Node node:unChecked) {
        if (node.getView().getName().equals(nameOfNode)) {
            return node;
        }
    }
    return null;
}

void    checkDrawNodes(ArrayList<ArrayList<String>>
namesOfNodesInComponents) {
    HashSet<Node> unChecked = new HashSet<>();
    HashSet<Color> colorsOfComponents = new HashSet<>();

```

```

for(Node node :GraphEventManager.getInstance().getGraph().getNodes()) {
    unchecked.add(node);
}

while(!unchecked.isEmpty() && !namesOfNodesInComponents.isEmpty()) {
    ArrayList<String>          namesOfNodesInComponent          =
namesOfNodesInComponents.remove(namesOfNodesInComponents.size()-1);

    String                    nameOfNode                        =
namesOfNodesInComponent.remove(namesOfNodesInComponent.size()-1);

    Node memberNode = getNodeFromUnChecked(unchecked, nameOfNode);

    if (memberNode == null) {
        fail();
    }

    unchecked.remove(memberNode);

    Color componentColor = memberNode.getView().getColor();

    if (!colorsOfComponents.isEmpty()) {
        if (colorsOfComponents.contains(componentColor)) {
            fail();
        }
    }

    colorsOfComponents.add(componentColor);

    if (namesOfNodesInComponent.size() > 1) {

```

```

        while (!namesOfNodesInComponent.isEmpty() &&
!unChecked.isEmpty()) {
            nameOfNode =
namesOfNodesInComponent.remove(namesOfNodesInComponent.size() - 1);
            memberNode = getNodeFromUnChecked(unChecked, nameOfNode);

            if (memberNode == null) {
                fail();
            }

            if (componentColor != memberNode.getView().getColor()) {
                fail();
            }

            unChecked.remove(memberNode);
        }
    }
}

@Test
void lots_of_empty_stack_in_DFS_1() {
    InputReader newOne = new InputReader(dirKosaraju+"test_1/dataset_1.xml");

    if (!newOne.FileOpen) {
        fail();
    }

    DrawGraph drawGraph = new DrawGraph(newOne.initFromData());

    tuneKosarajuAlgorithm();

```

```

        checkDrawNodes(parse(dirKosaraju+"test_1/answer_1.txt"));
    }

    @Test
    void one_node_2() {
        InputReader          newOne          =          new
InputReader(dirKosaraju+"test_2/dataset_2.xml");

        if (!newOne.FileOpen) {
            fail();
        }

        DrawGraph drawGraph = new DrawGraph(newOne.initFromData());

        tuneKosarajuAlgorithm();
        checkDrawNodes(parse(dirKosaraju+"test_2/answer_2.txt"));
    }

    @Test
    void lots_of_disconennected_nodes_3() {
        InputReader          newOne          =          new
InputReader(dirKosaraju+"test_3/dataset_3.xml");

        if (!newOne.FileOpen) {
            fail();
        }

        DrawGraph drawGraph = new DrawGraph(newOne.initFromData());

        tuneKosarajuAlgorithm();
        checkDrawNodes(parse(dirKosaraju+"test_3/answer_3.txt"));
    }

```

```

}

@Test
void two_disconenected_components_4() {
    InputReader newOne = new
InputReader(dirKosaraju+"test_4/dataset_4.xml");

    if (!newOne.FileOpen) {
        fail();
    }

    DrawGraph drawGraph = new DrawGraph(newOne.initFromData());

    tuneKosarajuAlgorithm();
    checkDrawNodes(parse(dirKosaraju+"test_4/answer_4.txt"));
}

@Test
void three_conenected_components_5() {
    InputReader newOne = new
InputReader(dirKosaraju+"test_5/dataset_5.xml");

    if (!newOne.FileOpen) {
        fail();
    }

    DrawGraph drawGraph = new DrawGraph(newOne.initFromData());

    tuneKosarajuAlgorithm();
    checkDrawNodes(parse(dirKosaraju+"test_5/answer_5.txt"));
}

```

```

@Test
void two_components_connected_by_two_parallel_edges_6() {
    InputReader newOne = new
InputReader(dirKosaraju+"test_6/dataset_6.xml");

    if (!newOne.FileOpen) {
        fail();
    }

    DrawGraph drawGraph = new DrawGraph(newOne.initFromData());

    tuneKosarajuAlgorithm();
    checkDrawNodes(parse(dirKosaraju+"test_6/answer_6.txt"));
}

@Test
void one_big_connected_component_7() {
    InputReader newOne = new
InputReader(dirKosaraju+"test_7/dataset_7.xml");

    if (!newOne.FileOpen) {
        fail();
    }

    DrawGraph drawGraph = new DrawGraph(newOne.initFromData());

    tuneKosarajuAlgorithm();
    checkDrawNodes(parse(dirKosaraju+"test_7/answer_7.txt"));
}

```

```

@Test
void one_component_consisting_of_components_8() {
    InputReader newOne = new
    InputReader(dirKosaraju+"test_8/dataset_8.xml");

    if (!newOne.FileOpen) {
        fail();
    }

    DrawGraph drawGraph = new DrawGraph(newOne.initFromData());

    tuneKosarajuAlgorithm();
    checkDrawNodes(parse(dirKosaraju+"test_8/answer_8.txt"));
}

```

```

@Test
void chain_of_components_each_consisting_of_one_node_9() {
    InputReader newOne = new
    InputReader(dirKosaraju+"test_9/dataset_9.xml");

    if (!newOne.FileOpen) {
        fail();
    }

    DrawGraph drawGraph = new DrawGraph(newOne.initFromData());

    tuneKosarajuAlgorithm();
    checkDrawNodes(parse(dirKosaraju+"test_9/answer_9.txt"));
}

```

```

@Test

```



```

void closed_chain_of_nodes_it_is_one_component_10() {
    InputReader          newOne          =          new
InputReader(dirKosaraju+"test_10/dataset_10.xml");

    if (!newOne.FileOpen) {
        fail();
    }

    DrawGraph drawGraph = new DrawGraph(newOne.initFromData());

    tuneKosarajuAlgorithm();
    checkDrawNodes(parse(dirKosaraju+"test_10/answer_10.txt"));
}
}

```