МИНОБРНАУКИ РОССИИ
Санкт-Петербургский государственный
электротехнический университет
«ЛЭТИ» им. В.И. Ульянова (Ленина)
Кафедра МО ЭВМ

ОТЧЕТ

по учебной практике

Тема: «Поиск компонент сильной связности»

Студентка гр. 8304 _____ Николаева М. А.

Студентка гр. 8304 _____ Мельникова О. А.

Студент гр. 8304 _____ Щука А. А.

Руководитель _____ Фирсов М. А.

Санкт-Петербург

2020

ЗАДАНИЕ

НА УЧЕБНУЮ ПРАКТИКУ

Студентка Николаева М. А. группы 8304

Студентка Мельникова О. А. группы 8304

Студент Щука А. А. группы 8304

Тема практики: Поиск компонент сильной связности

Задание на практику:

Командная итеративная разработка визуализатора алгоритма на Java с графическим интерфейсом.

Алгоритм: алгоритм Косарайю.

Сроки прохождения практики: 29.06.2020 – 12.07.2020

Дата сдачи отчета: _.07.2020

Дата защиты отчета: _.07.2020

Студентка гр. 8304		Николаева М. А.
Студентка гр. 8304		Мельникова О. А.
Студент гр. 8304		Щука А. А.
Руководитель		Фирсов М. А.

АННОТАЦИЯ

Целью работы является получения навыков работы с такой парадигмой программирования, как объектно-ориентированное программирование. Для получения данных знаний выполняется один из вариантов мини-проекта. В процессе выполнения мини-проекта необходимо реализовать графический интерфейс к данной задаче, организовать ввод и вывод данных с его помощью, реализовать сам алгоритм, научиться работать в команде. В данной работе в качестве мини-проекта выступает поиск компонент сильной связности (визуализация алгоритма Косарайю). Также при разработке выполняется написание тестирования, для проверки корректности алгоритма.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Основная цель практики – реализация мини-проекта, который является визуализацией алгоритма. В данной работе это алгоритм Косарайю. Для выполнения этой цели были поставлены задачи: реализация алгоритма, разработка GUI к проекту, написание тестирования.

Ориентированный граф (орграф) называется сильно связным, если любые две его вершины s и t сильно связаны, то есть 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. Областью сильной связности называется множество вершин компоненты сильной связности.

Алгоритм Косарайю (в честь американского учёного индийского происхождения Самбасивы Рао Косарайю) — алгоритм поиска областей сильной связности в ориентированном графе. Чтобы найти области сильной связности, сначала выполняется поиск в глубину (DFS) на обращении исходного графа (то есть против дуг), вычисляя порядок выхода из вершин. Затем мы используем обращение этого порядка, чтобы выполнить поиск в глубину на исходном графе (в очередной раз берём вершину с максимальным номером, полученным при обратном проходе). Деревья в лесе DFS, которые выбираются в результате, представляют собой сильные компоненты связности.

1. ТРЕБОВАНИЯ К ПРОГРАММЕ

1.1 Исходные требования к программе

Пользователь с помощью графического интерфейса конструирует не взвешенный ориентированный граф. Пользователь может добавлять/удалять вершины графа. Помимо этого, пользователь может задавать ребра в графе, нажав на одну вершину и потянув указатель мыши к другой вершине. После запуска алгоритма программа визуализирует алгоритм, а также выводит текстовые данные, поясняющие ход выполнения алгоритма. Также входные данные могут быть получены из файла.

1.1.1 Требования к входным данным

Для корректной работы алгоритма требуется:

- множество вершин графа
- множество ребер графа

1.1.2 Требования к визуализации

Программа должна обладать простым и понятным интерфейсом. Визуализироваться должны все стадии алгоритма - обход графа в глубину, отображение всех вершин в порядке увеличения времени выхода (массив verticese), обход в глубину на инвертированном графе (каждый раз для обхода будем выбирать ещё не посещенную вершину с максимальным индексом в массиве verticese), при этом на каждой итерации должно быть отображение посещенных вершин - компоненты сильной связности. 0-я версия интерфейса (прототип) должна быть готова к 6 июля, выполнена 28 июня.

Программа имеет интерактивную область с графом с возможностью выбора вершин, ребер и запуска алгоритма. С правой стороны расположены функциональные кнопки, снизу расположено поле для вывода текстовой информации для пояснения алгоритма.

1.1.3 Требования к алгоритму и данным

Алгоритм получает на вход граф, заданный пользователем, и в процессе выполнения передает промежуточные состояния графа для визуализации.

1.1.4 Требования к выходным данным

Выходные данные: компоненты сильной связности в исходном графе.

1.2 Требования к программе после уточнения работы

Пользователь с помощью графического интерфейса конструирует не взвешенный ориентированный граф. Пользователь может добавлять/удалять вершины графа. Помимо этого, пользователь может задавать ребра в графе, нажав на одну вершину и потянув указатель мыши к другой вершине. После запуска алгоритма программа визуализирует алгоритм, а также выводит текстовые данные, поясняющие ход выполнения алгоритма. Во время визуализации пользователь может приостанавливать алгоритм, а также менять скорость визуализации. Входные данные могут быть получены из файла.

1.2.2 Требования к визуализации

Должна быть добавлена возможность вызова пояснения о том, как пользоваться программой с помощью ToolBar. Вершины должны добавляться нажатием на соответствующую кнопку. Ребра должны добавляться при перетягивании курсором от одной вершины к другой. Должна быть добавлена возможность выбора считывания из файла. Вершину/ребро можно выбрать и соответствующими кнопками удалить (при выборе должны подсвечиваться). Должна быть кнопка очистить, которая полностью удаляет весь граф. Для запуска алгоритма должна быть добавлена кнопка старт. Также должна быть кнопка стоп, при нажатии которой алгоритм сразу же завершает свою работу, граф возвращается в исходное состояние. Должна быть реализована

возможность остановить работу алгоритма (кнопка пауза) и продолжить с места остановки (кнопка старт). Также должна быть добавлена возможность уменьшать и увеличивать скорость демонстрации работы алгоритма. Изначально кнопки остановки, паузы и изменение скорости демонстрации заблокированы. После нажатия кнопки старт должны стать доступными заблокированные кнопки, а кнопка старт и все кнопки, отвечающие за изменение графа, заблокироваться. При нажатии кнопки паузы становится доступной кнопка старт, кнопка паузы блокируется. При нажатии кнопки стоп все кнопки, кроме остановки, паузы и изменения скорости демонстрации, становятся доступными.

Визуализация работы алгоритма представляет собой: первый поиск в глубину при выходе из вершины окрашивает ее в другой цвет. Затем вершины возвращают исходный цвет. Граф транспонируется, ребра ориентируются в противоположные стороны. Второй запуск поиска в глубину. Вершины также меняют цвет. При этом вершины каждой компоненты связности должны иметь свой цвет, соответствующий компоненте, в которую они входят. По ходу работы алгоритма должны выводиться пояснения, касающиеся первого и второго поиска в глубину, а также о транспонировании графа.

Итоговый прототип интерфейса представлен на рисунке 2.

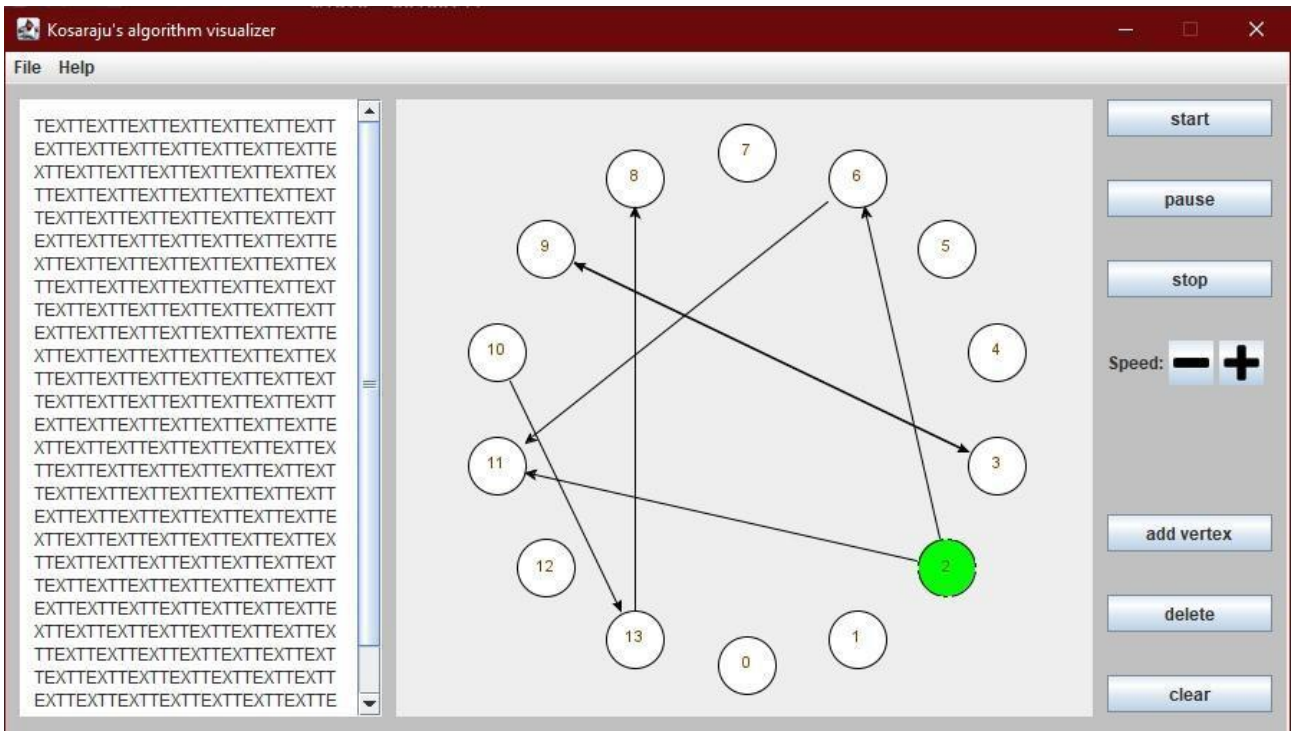

Рисунок 1 - Итоговый прототип интерфейса

1.2.3 Требования к алгоритму и данным

Алгоритм, для визуализации, должен при поиске компонент связности возвращать промежуточные значения, касающиеся первого и второго обходов в глубину (данные о запуске первого и второго поиска в глубину, информация о посещении вершин), а также о транспонировании графа.

По ходу работы алгоритма должны выводиться следующие пояснения: при первом запуске поиска в глубину сообщение о его запуске и данные о посещении вершин, затем сообщение о том, что все ребра были инвертированы и граф транспонирован, при втором запуске поиска в глубину сообщение о его запуске и данные о посещении вершин, информация о количестве найденных компонент сильной связности и сами компоненты.

Для ввода из файла первой строкой должно идти кол-во вершин в графе. Дальше ребра вида i j, где i и j в диапазоне от 0 до кол-ва вершин.

1.2.4 Требования к выходным данным

Выходными данными являются раскрашенные в свой цвет компоненты связности.

1.2.5 Требования после сдачи 1-ой версии

- При изменении размера окна и холста меняется и расстояние между вершинами, всё пространство холста должно использоваться.
- Текущая скорость воспроизведения должна выводиться под кнопками изменения скорости.
- Рёбра, не ставшие древесными при обходе, но просмотренные, должны выделяться ещё одним цветом.
- Рядом с вершинами при выполнении первого обхода следует выводить порядковые метки, которые будут учитываться при втором обходе.

1.2.6 Требования после сдачи 2-ой версии

- Сделать регулировку скорости доступной до старта алгоритма и во время паузы.
- Объединить кнопки Start и Pause (имя кнопки меняется по нажатию).
- Сделать либо свободное перетаскивание вершин по полю, либо возможность обменять вершины местами (в смысле расположения на холсте).

2. ПЛАН РАЗРАБОТКИ И РАСПРЕДЕЛЕНИЕ РОЛЕЙ В БРИГАДЕ

2.1 План разработки

1. Создание прототипа (к 2 июля 2020 года).
    1) Обсудить задание, распределить роли, выбрать необходимые средства разработки и структуры данных. Данный пункт задания необходимо выполнить до 1 июля 2020 года.
    2) Создать прототип GUI (0-я версия визуализации). Добавить возможность создавать граф и отображать его. Вершины должны добавляться нажатием на соответствующую кнопку. Ребра должны добавляться при перетягивании курсором от одной вершины к другой. Вершину/ребро

можно выбрать и соответствующими кнопками удалить (при выборе должны подсвечиваться). Данный пункт задания необходимо выполнить к 2 июля 2020 года.

2. Создание 1-ой версии программы (к 8 июля 2020 года).
    1) Реализовать структуры данных, необходимые для алгоритма. Данный пункт задания необходимо выполнить к 3 июля 2020 года.
    2) Реализовать алгоритм без использования GUI. Данный пункт задания необходимо выполнить к 4 июля 2020 года.
    3) Добавить JavaDoc комментарии для генерации документации к алгоритму и структуре данных. Данный пункт задания необходимо выполнить до 5 июля 2020 года.
    4) Связывание структур данных алгоритма и визуализации. Данный пункт задания необходимо выполнить к 6 июля 2020 года.
    5) Реализация основного GUI. (1-я версия). Должна быть реализована визуализация работы алгоритма. Добавить кнопку отчистки, которая полностью удаляет весь граф. Для запуска алгоритма должна быть добавлена кнопка старт. При ее нажатии все остальные кнопки в процессе работы алгоритма нажать нельзя. Также должна быть реализована возможность полной остановки работы алгоритма (кнопка стоп). Данный пункт задания необходимо выполнить к 7 июля 2020 года.
    6) Отладка ошибок. Данный пункт задания необходимо выполнить к 8 июля 2020 года.
3. Создание финальной версии (к 11 июля 2020 года).
    1) Улучшение GUI (2-я версия). Добавление возможности остановить работу алгоритма (кнопка пауза) с последующим продолжение работы с места остановки. Добавление возможности увеличивать и уменьшать скорость демонстрации работы алгоритма. Добавление возможности

получения пояснения об использовании программы с помощью ToolBar. Данный пункт задания необходимо выполнить к 9 июля 2020 года.

2) Добавление возможности считывания входных данных из файла. Данный пункт задания необходимо выполнить к 9 июля 2020 года.
3) Добавление полноценного тестирования всех модулей программы. Данный пункт задания необходимо выполнить до 11 июля 2020 года.

2.2 Распределение ролей в бригаде

- Николаева М. А.:
  - Расширение возможностей GUI.
  - Реализация ввода-вывода;
  - Проектирование, организация командной работы.
- Щука А. А.:
  - Создание основного GUI;
  - Объединение отдельных модулей программы;
  - Рефракторинг кода.
- Мельникова О. А.:
  - Создание алгоритма и структур данных;
  - Оформление пояснительной записки;
  - Тестирование программы;

3. ОСОБЕННОСТИ РЕАЛИЗАЦИИ

3.1 Использованные структуры данных

Описание реализованных структур данных указаны в таблице №1.

Таблица №1 - Описание классов и структур данных

Имя класса	Описание

Edge	Класс, описывающий ориентированное ребро, содержит информацию о вершинах, которые оно соединяет: 1. private Vertex sourceVertex; 2. private Vertex targetVertex; Методы: 1. public Edge(Vertex startVertex, Vertex targetVertex) - конструктор ориентированного ребра, принимающий начальную веришину и конечную. 2. public Vertex getSourceVertex() - возвращает начальную вершину. 3. public void setSourceVertex(Vertex sourceVertex) - устанавливает начальную вершину. 4. public Vertex getTargetVertex() - возвращает конечную вершину. 5. public void setTargetVertex(Vertex targetVertex) - устанавливает конечную вершину. 6. public String toString() - возвращает строковое представление ребра, которое содержит информацию о всех его полях.
Graph	Класс, реализующий граф. Содержит в себе следующие поля: 1. private List<Vertex> vertexList - список вершин графа. 2. private List<Edge> edgeList - список ребер графа. Методы: 1. public Graph() - конструктор для создания пустого графа. 2. public Graph(List<Vertex> vertexList, List<Edge> edgeList) - конструктор графа, принимающий список вершин vertexList и список ребер edgeList 3. public List<Vertex> getVertexList() - возвращает список вершин графа 4. public void setVertexList(List<Vertex> vertexList) - устанавливает список вершин 5. public List<Edge> getEdgeList() - возвращает список ориентированных ребер графа 6. public void setEdgeList(List<Edge> edgeList) - устанавливает список ребер 7. public void transpose() - транспонирует граф: ребра исходного графа ориентируются в противоположном направлении.
Vertex	Класс, представляющий собой вершину графа.Содержите в себе следующие поля: 1. private int id - номер вершины.

	2. private boolean isVisited - статус посещенности вершины. 3. private List<Vertex> adjacencyList - список смежных вершин. 4. private int componentId - номер компоненты сильной связности, к которой принадлежит вершина. Методы: 1. public Vertex(int id) - конструктор вершины, принимающий ее номер id. 2. public int getId() - Возвращает номер id вершины. 3. public void setId(int id) - устанавливает номер вершины id. 4. public boolean isVisited() - возвращает true, если вершина была посещена. 5. public void setVisited(boolean isVisited) - устанавливает значение статуса посещенности вершины. 6. public List<Vertex> getAdjacencyList() - возвращает список смежных вершин. 7. public void setAdjacencyList(List<Vertex> adjacencyList) - устанавливает список смежных вершин. 8. public int getComponentId() - возвращает номер компоненты сильной связности, к которой принадлежит вершина. 9. public void setComponentId(int componentId) - устанавливает номер компоненты сильной связности, к которой принадлежит вершина. 10. public void addNeighbour(Vertex vertex) - добавляет вершину в список смежных вершин. 11. public String toString() - возвращает строковое представление вершины, которое содержит информацию о всех ее полях.

3.2 Основные методы

Основные методы для работы были реализованы в классе Algorithm, который реализует поиск компонент связности.

Он содержит в себе такие переменные:

- MARK_EDGE - ребро помечено
- UNMARK_EDGE - ребро не помечено
- MARK_VISITED_VERTEX - вершина посещена
- MARK_FINISHED_VERTEX - вершина вышла из DFS
- UNMARK_VERTEX - вершина не помечена

- TRANSPOSE_GRAPH - граф транспонирован
- ALGORITHM_ENDED - алгоритм закончил работу
- ADD_TEXT - добавлен текст, поясняющий ход выполнения алгоритма
- MAX_DELAY - максимальная задержка анимации алгоритма
- MIN_DELAY - минимальная задержка анимации алгоритма
- DELTA_DELAY - шаг изменения задержки анимации алгоритма
- private final StringBuilder componentsString - строка для сохранения информации о компонентах сильной связности
- private Graph graph - граф, на котором будет реализован алгоритм
- private int count - количество компонент сильной связности в графе
- private final LinkedList<Vertex> orderList - список вершин, расположенных в порядке убывания времени выхода при первом обходе графа

Реализованные основные методы описаны в таблице №2.

Таблица №2 - Основные методы класса Algorithm

Объявление метода	Описание
public Algorithm(Graph graph)	Конструктор, принимающий граф, к которому будет применяться алгоритм.
protected Void doInBackground()	Перегруженный метод родительского класса. Метод выполняет алгоритм в новом потоке. Метод реализует алгоритм Косайрайю поиска компонент сильной связности в графе. Последовательно запускает первый обход графа в глубину, затем транспонирование графа и второй обход графа в глубину в порядке, определенном при первом обходе в глубину. После завершения работы алгоритма, граф возвращается в исходное состояние.
private void transposeGraph()	Вызывает метод транспонирования графа и посылает сигнал родительскому потоку об изменении состояния графа.
protected void done()	Переопределенный метод

	родительского класса, который будет вызван при завершении алгоритма. Посылает сигнал о том, что алгоритм завершен.
private void firstDFS(Vertex vertex)	Метод, реализующий обход графа в глубину, с сохранением порядка вершин по убыванию времени выхода в orderList, для последующего использования в алгоритме.
private void secondDFS(Vertex vertex)	Метод, реализующий обход графа в глубину. При обходе сохраняет в вершины vertex.setComponentId, входящие в одну компоненту сильной связности, соответствующий номер компоненты.
public void setRun(boolean run)	Устанавливает флаг статуса выполнения алгоритма. Если false, то алгоритм останавливается.
private synchronized void sleepOrWait()	Приостанавливает поток выполнения на delay мс, если isRun равен true. Если isRun равен false, то поток переходит в режим ожидания сигнала
public Graph getGraph()	Возвращает граф, к которому применяется алгоритм.
private void unVisit(Graph graph)	Метод, изменяющий статус посещенности вершин графа на "не посещённые". Отправляет сигнал об изменении состояния графа.
public synchronized void unSleep()	Устанавливает флаг статуса выполнения алгоритма. Вызывает метод для продолжения

	выполнения алгоритма.
public void increaseDelay()	Увеличивает задержку анимации алгоритма на заданную константу
public void decreaseDelay()	Уменьшает задержку анимации алгоритма на заданную константу

4. ТЕСТИРОВАНИЕ

4.1 Написание UNIT Tests

Написание Unit Test с помощью библиотеки JUnit. Unit тесты были написаны с целью покрыть основные моменты кода алгоритма для того, чтобы убедиться в корректности работы.

Были написаны тесты для всех реализованных структур данных. Для каждого класса были написаны тесты, покрывающие все его методы, в частности для проверки алгоритма были реализованы различные графы, включая пустой, граф без ребер, полный. Проверка осуществлялась с помощью функции Assertions.assertEquals.

4.2 Ручное тестирование программы

Ручное тестирование кода проводилось для выявления слабых мест программы, непокрытых Unit Test. Проводилось тестирование графического интерфейса. Пример ручного тестирования представлен на рисунке №2.

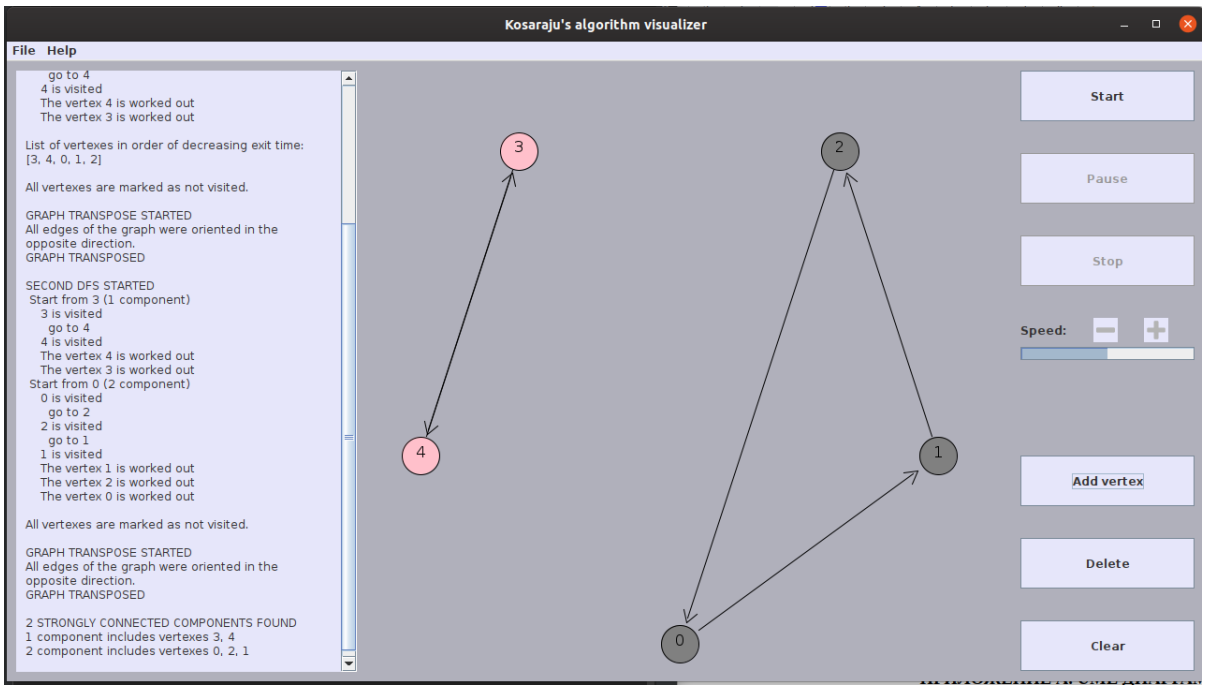

Рисунок №2 - Ручное тестирование

ЗАКЛЮЧЕНИЕ

Разработка поставленной задачи была выполнена в соответствии с планом. Было спроектировано и реализовано приложение для поиска компонент сильной связности (алгоритм Косарайю). Был реализован графический интерфейс, визуально отображающий результаты работы алгоритма и позволяющий управлять возможностями приложения. Основной алгоритм был покрыт Unit Test, а графический интерфейс был оттестирован вручную.

Таким образом разработка приложения была завершена успешно с полным выполнением плана.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. ГОСТ 7.32-2017 Система стандартов по информатизации, библиотечному и издательскому делу. Отчет о научно-исследовательской работе. Структура и правила оформления.
2. Учебный курс https://stepik.org/course/187/syllabus?auth=registration.

ПРИЛОЖЕНИЕ А. UML ДИАГРАММА

Смотреть файл Kosaraju.pdf

ПРИЛОЖЕНИЕ Б. ИСХОДНЫЙ КОД

Main.java

```
import logger.Logs;


import java.util.logging.Level;



public class Main {
  public static void main(String[] args) {
    Logs.setLogLevelForOutput(Level.INFO);
    Logs.writeToLog("Start program", Level.INFO);


    MainWindow mainWindow = new MainWindow();
    mainWindow.setVisible(true);
  }
}
```

Algorithm.java

```
package graph;
import GUI.Pair;
import logger.Logs;
import org.jetbrains.annotations.NotNull;


import javax.swing.*;
import java.util.LinkedList;
import java.util.concurrent.atomic.AtomicBoolean;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.logging.Level;


/**
* Класс, представляющий собой реализацию алгоритма поиска компонент
* сильной связности.
* <p>
* Наследуется от класса {@code SwingWorker}.
* <p>
* Содержит в себе:
*
* @value MARK_EDGE - ребро помечено
```

```
* @value UNMARK_EDGE - ребро не помечено
* @value MARK_VISITED_VERTEX - вершина посещена
* @value MARK_FINISHED_VERTEX - вершина вышла из DFS
* @value UNMARK_VERTEX - вершина не помечена
* @value TRANSPOSE_GRAPH - граф транспонирован
* @value ALGORITHM_ENDED - алгоритм закончил работу
* @value ADD_TEXT - добавлен текст, поясняющий ход выполнения алгоритма
* @value MAX_DELAY - максимальная задержка анимации алгоритма
* @value MIN_DELAY - минимальная задержка анимации алгоритма
* @value DELTA_DELAY - шаг изменения задержки анимации алгоритма
* @value SET_VERTEX_VALUE - установить новое значение вершины
* @value RESET_VERTEX_VALUES - сбросить все значения вершин
* @see SwingWorker
* <p>
* Поле для хранения графа {@code graph}, количество компонент сильной связности в графе {@code count},
* список вершин, расположенных в порядке убывания времени выхода при первом обходе графа {@code orderList} и
* строку с информацией о компонентах сильной связности {@code componentsString}.
* <p>
* Содержит методы для реализации алгоритма поиска компонент сильной связности Косарайю.
*/


public class Algorithm extends SwingWorker<Void, Void> {
  public static final String MARK_VISITED_EDGE = "markVisitedEdge";
  public static final String MARK_UNVISITED_EDGE = "markUnvisitedEdge";
  public static final String UNMARK_EDGE = "unmarkEdge";
  public static final String MARK_VISITED_VERTEX = "markVisitedVertex";
  public static final String MARK_FINISHED_VERTEX = "markFinishedVertex";
  public static final String UNMARK_VERTEX = "unmarkVertex";
  public static final String TRANSPOSE_GRAPH = "transposeGraph";
  public static final String ALGORITHM_ENDED = "algorithmEnded";
  public static final String ADD_TEXT = "addText";
  public static final String CLEAR_TEXT_PANE = "clearTextPane";
  public static final String SET_VERTEX_VALUE = "setVertexValue";
  public static final String RESET_VERTEX_VALUES = "resetVertexValues";


  public static final int MAX_DELAY = 2000;
  public static final int MIN_DELAY = 0;
  public static final int DELTA_DELAY = 100;


  private final AtomicBoolean isRun;
  private final AtomicInteger delay;


  /**
```

```
     * Строка для сохранения информации о компонентах сильной связности.
     */
    private final StringBuilder componentsString;
    /**
     * Граф, на котором будет реализован алгоритм.
     */
    private Graph graph;
    /**
     * Количество компонент сильной связности в графе {@code graph}.
     */
    private int count;
    /**
     * Список вершин, расположенных в порядке убывания времени выхода при первом обходе графа.
     */
    private final LinkedList<Vertex> orderList;


    /**
     * Конструктор, принимающий граф {@code graph}, к которому будет
     * применяться алгоритм.
     *
     * @param (graph) граф.
     */
    public Algorithm(@NotNull Graph graph) {
        this.componentsString = new StringBuilder();
        Logs.writeToLog("Called Algorithm constructor");
        this.isRun = new AtomicBoolean(true);
        this.delay = new AtomicInteger((MAX_DELAY + MIN_DELAY) / 2);
        this.graph = graph;
        this.orderList = new LinkedList<>();
    }


    public Algorithm(@NotNull Graph graph, int delay) {
        this(graph);
        this.delay.set(delay);
    }


    /**
     * Перегруженный метод родительского класса
     *
     * @see SwingWorker#doInBackground()
     * Метод выполняет алгоритм в новом потоке.
     * <p>
     * Метод реализует алгоритм Косайрайю поиска копонент сильной связности в графе.
     * Последовательно запускает первый обход графа в глубину, затем транспонирование графа и второй обход графа в
```

```
 * глубину в порядке, определенном при первом обходе в глубину.
 * После завершения работы алгоритма, граф возвращается в исходное состояние.
 */
@Override
protected Void doInBackground() {
    try {
        firePropertyChange(CLEAR_TEXT_PANE, null, null);
        Logs.writeToLog("Start working");
        firePropertyChange(ADD_TEXT, null, "START ALGORITHM" + System.lineSeparator());


        if (graph.getVertexList().size() == 0) {
            Logs.writeToLog("Error: graph is empty!");
            firePropertyChange(ADD_TEXT, null, "Error: graph is empty!" + System.lineSeparator());
            return null;
        }


        Logs.writeToLog("FIRST DFS STARTED");
        firePropertyChange(ADD_TEXT, null, "FIRST DFS STARTED" + System.lineSeparator());
        for (Vertex vertex : graph.getVertexList()) {
            if (!vertex.isVisited()) {
                sleepOrWait();
                Logs.writeToLog(" Start from " + vertex.getId());
                firePropertyChange(ADD_TEXT, null, " Start from " + vertex.getId() +
                        System.lineSeparator());
                firstDFS(vertex);
            }
        }


        sleepOrWait();
        Logs.writeToLog("List of vertexes in order of decreasing exit time: " + System.lineSeparator() +
                orderListToString());
        firePropertyChange(ADD_TEXT, null, System.lineSeparator() +
                "List of vertexes in order of decreasing exit time: " + System.lineSeparator() +
                orderListToString() + System.lineSeparator());


        unVisit(graph);
        transposeGraph();


        Logs.writeToLog("SECOND DFS STARTED");
        firePropertyChange(ADD_TEXT, null, System.lineSeparator() + "SECOND DFS STARTED" +
                System.lineSeparator());
        for (Vertex vertex : orderList) {
            if (!vertex.isVisited()) {
```

```
                sleepOrWait();
                Logs.writeToLog(" Start from " + vertex.getId() + " (" + (count + 1) + " component)");
                firePropertyChange(ADD_TEXT, null, " Start from " + vertex.getId() +
                        " (" + (count + 1) + " component)" + System.lineSeparator());


                componentsString.append(count + 1).append(" component includes vertexes ");
                secondDFS(vertex);
                componentsString.append(System.lineSeparator());
                ++count;
            }
        }
        unVisit(graph);
        transposeGraph();
        firePropertyChange(RESET_VERTEX_VALUES, null, null);


        return null;
    } catch (Exception ignored) {
        //ignored
    }


    return null;
}


/**
 * Вызывает метод транспонирования графа
 *
 * @throws InterruptedException если поток был прерван.
 * @see Graph#getTransposedGraph()
 * и посылает сигнал родительскому потоку об изменении состояния графа.
 */
private void transposeGraph() throws InterruptedException {
    sleepOrWait();
    Logs.writeToLog("GRAPH TRANSPOSE STARTED");
    firePropertyChange(ADD_TEXT, null, System.lineSeparator() +
            "GRAPH TRANSPOSE STARTED" + System.lineSeparator());


    sleepOrWait();
    graph = graph.getTransposedGraph();
    firePropertyChange(TRANSPOSE_GRAPH, null, null);
    Logs.writeToLog("All edges of the graph were oriented in the opposite direction." +
System.lineSeparator() +
            "GRAPH TRANSPOSED");
    firePropertyChange(ADD_TEXT, null,
            "All edges of the graph were oriented in the opposite direction." + System.lineSeparator() +
```

```
                "GRAPH TRANSPOSED" + System.lineSeparator());

        sleepOrWait();
    }

    /**
     * Переопределенный метод родительского класса, который будет вызван при завершении алгоритма.
     * Посылает сигнал о том, что алгоритм завершен.
     *
     * @see SwingWorker#done()
     */
    @Override
    protected void done() {
        try {
            get();
            Logs.writeToLog(count + " STRONGLY CONNECTED COMPONENTS FOUND ");
            firePropertyChange(ADD_TEXT, null, System.lineSeparator() + count +
                    " STRONGLY CONNECTED COMPONENTS FOUND " +
                    System.lineSeparator());
            Logs.writeToLog(componentsString.toString());
            firePropertyChange(ADD_TEXT, null, componentsString);
            firePropertyChange(ALGORITHM_ENDED, null, null);
        } catch (Exception e) {
            firePropertyChange(CLEAR_TEXT_PANE, null, null);
        }
    }

    /**
     * Метод, реализующий обход графа в глубину, с сохранением порядка вершин по убыванию времени выхода в
     * {@code orderList}, для последующего использования в алгоритме.
     *
     * @param vertex - вершина, к которой применяем обход в глубину.
     * @see Algorithm#doInBackground()
     */
    private void firstDFS(@NotNull Vertex vertex) throws InterruptedException {
        sleepOrWait();
        vertex.setVisited(true);
        firePropertyChange(MARK_VISITED_VERTEX, null, vertex);
        Logs.writeToLog(vertex.getId() + " is visited");
        firePropertyChange(ADD_TEXT, null, "    " + vertex.getId() + " is visited" +
                System.lineSeparator());

        for (Vertex neighbour : vertex.getAdjacencyList()) {
```

```
            if (!neighbour.isVisited()) {
                sleepOrWait();
                Logs.writeToLog("go to " + neighbour.getId());
                firePropertyChange(ADD_TEXT, null,
                        "      go to " + neighbour.getId() + System.lineSeparator());
                firePropertyChange(MARK_VISITED_EDGE, vertex, neighbour);
                firstDFS(neighbour);
            } else {
                firePropertyChange(MARK_UNVISITED_EDGE, vertex, neighbour);
            }
        }
        sleepOrWait();
        Logs.writeToLog("The vertex " + vertex.getId() +
                " is worked out ");
        firePropertyChange(ADD_TEXT, null, "    The vertex " + vertex.getId() +
                " is worked out " + System.lineSeparator());
        firePropertyChange(MARK_FINISHED_VERTEX, null, vertex);
        firePropertyChange(SET_VERTEX_VALUE, null,
                new Pair<>(vertex.getId(), orderList.size()));
        orderList.addFirst(vertex);
    }


    /**
     * Метод, реализующий обход графа в глубину.
     * При обходе сохраняет в вершины {@code vertex.setComponentId}, входящие в одну компоненту
сильной связности,
     * соответствующий номер компоненты.
     *
     * @param vertex - вершина, к которой применяем обход в глубину.
     * @throws InterruptedException если поток был прерван.
     */
    private void secondDFS(@NotNull Vertex vertex) throws InterruptedException {
        sleepOrWait();
        vertex.setVisited(true);
        vertex.setComponentId(count);
        Logs.writeToLog(vertex.getId() + " is visited ");
        firePropertyChange(ADD_TEXT, null, "    " + vertex.getId() + " is visited " +
                System.lineSeparator());
        firePropertyChange(MARK_VISITED_VERTEX, count, vertex);
        componentsString.append(vertex.getId());


        for (Vertex neighbour : vertex.getAdjacencyList()) {
            sleepOrWait();
            if (!neighbour.isVisited()) {
                firePropertyChange(MARK_VISITED_EDGE, vertex, neighbour);
```

```
            Logs.writeToLog("go to " + neighbour.getId());
            firePropertyChange(ADD_TEXT, null,
                    "      go to " + neighbour.getId() + System.lineSeparator());
            componentsString.append(", ");
            secondDFS(neighbour);
        } else {
            firePropertyChange(MARK_UNVISITED_EDGE, vertex, neighbour);
        }
    }
    Logs.writeToLog("The vertex " + vertex.getId() +
            " is worked out ");
    firePropertyChange(ADD_TEXT, null, "    The vertex " + vertex.getId() +
            " is worked out " + System.lineSeparator());
}


/**
 * Устанавливает флаг статуса выполнения алгоритма.
 * Если {@code false}, то алгоритм останавливается.
 *
 * @param run флаг статуса выполнения алгоритма.
 * @see Algorithm#sleepOrWait()
 */
public void setRun(boolean run) {
    Logs.writeToLog("isRun changed from " + this.isRun + " to " + run);
    isRun.set(run);
}


public void offDelay() {
    delay.set(0);
}


/**
 * Приостанавливает поток выполнения на {@code delay} мс, если {@code isRun == true}.
 * <p>
 * Если {@code isRun == false}, то поток переходит в режим ожидания сигнала
 *
 * @see Object#wait()
 * @see Algorithm#unSleep()
 */
private synchronized void sleepOrWait() throws InterruptedException {
    if (isRun.get()) {
        Logs.writeToLog("isRun - true");
        Thread.sleep(delay.get());
    }
```

```
    while (!isRun.get()) {
        Logs.writeToLog("isRun - false. Wait");
        wait();
    }
}

/**
 * Возвращает значение поля {@code graph}.
 *
 * @return граф, к которому применяется алгоритм.
 */
public Graph getGraph() {
    return graph;
}

/**
 * Устанавливает флаг статуса выполнения алгоритма {@code isRun = true}.
 * Вызывает метод {@code notifyAll()} для продолжения выполнения алгоритма.
 *
 * @see Object#notifyAll()
 */
public synchronized void unSleep() {
    Logs.writeToLog("isRun is set to true");
    isRun.set(true);
    notifyAll();
}

/**
 * Метод, изменяющий статус посещенности вершин графа на "не посещённые".
 * Отправляет сигнал об изменении состояния графа.
 *
 * @param graph граф, в котором меняется статус посещенности
 * @throws InterruptedException если поток был прерван
 */
private void unVisit(@NotNull Graph graph) throws InterruptedException {
    sleepOrWait();
    Logs.writeToLog("All vertexes are marked as not visited.");
    firePropertyChange(ADD_TEXT, null, System.lineSeparator() +
            "All vertexes are marked as not visited." + System.lineSeparator());

    for (Vertex vertex : graph.getVertexList()) {
        firePropertyChange(UNMARK_VERTEX, null, vertex);
        vertex.setVisited(false);
```

```
            for (Vertex neighbour : vertex.getAdjacencyList()) {
                firePropertyChange(UNMARK_EDGE, vertex, neighbour);
            }
        }
    }

    /**
     * Метод, возвращающий строковое предсставление {@code orderList}
     * в определенном формате.
     *
     * @return строковое представление {@code orderList}.
     */
    private @NotNull String orderListToString() {
        StringBuilder string = new StringBuilder("[");

        for (int i = 0; i < orderList.size(); ++i) {
            string.append(orderList.get(i).getId());
            if (i != orderList.size() - 1) {
                string.append(", ");
            }
        }
        string.append("]");

        Logs.writeToLog("orderList created and returned");
        return string.toString();
    }

    /**
     * Увеличивает задержку анимации алгоритма на заданную константу {@code DELTA_DELAY}.
     */
    public void increaseDelay() {
        synchronized (this.delay) {
            if (delay.get() <= MAX_DELAY - DELTA_DELAY) {
                Logs.writeToLog("Delay increased");
                delay.addAndGet(DELTA_DELAY);
            }
        }
    }

    /**
     * Метод, возвращающий количество компонент сильной связности {@code count}.
     *
```

```
     * @return количество компонент сильной связности.
     */
    public int getCount() {
        return count;
    }


    /**
     * Уменьшает задержку анимации алгоритма на заданную константу {@code DELTA_DELAY}.
     */
    public void decreaseDelay() {
        synchronized (this.delay) {
            if (delay.get() - DELTA_DELAY > MIN_DELAY) {
                Logs.writeToLog("Delay decreased");
                delay.addAndGet(-DELTA_DELAY);
            }
        }
    }


    public void setDelay(int delay) {
        if (delay >= MIN_DELAY && delay <= MAX_DELAY) {
            this.delay.set(delay);
        }
    }
}
```

Edge.java

```
package
graph;


        import logger.Logs;
        import org.jetbrains.annotations.NotNull;


        import java.util.Objects;


        /**
        * Класс, представляющий собой ориентированное ребро графа.
        * Содержит в себе информацию о начальной {@code sourceVertex} и конечной вершине
        {@code targetVertex}.
        * <p>
        * Класс предоставляет методы для получения значений полей класса:
        * {@code getSourceVertex}, {@code getTargetVertex}.
        * <p>
        * Класс предоставляет методы для установки значений полей класса:
        * {@code setSourceVertex}, {@code setTargetVertex}.
```

```
*/

public class Edge {
  /**
   * Начальная вершина.
   */
  private Vertex sourceVertex;

  /**
   * Конечная вершина.
   */
  private Vertex targetVertex;

  /**
   * Конструктор ориентированного ребра, принимающий начальную вершину {@code
startVertex} и
   * конечную вершину{@code targetVertex}.
   *
   * @param (startVertex,targetVertex) начальная и конечная вершина.
   */
  public Edge(@NotNull Vertex startVertex, @NotNull Vertex targetVertex) {
    Logs.writeToLog("Created edge from vertex " + startVertex.getId() + " to vertex " +
        targetVertex.getId());

    this.sourceVertex = startVertex;
    this.targetVertex = targetVertex;
  }

  /**
   * Возвращает начальную вершину {@code sourceVertex}.
   *
   * @return начальная вершина ребра.
   */
  public Vertex getSourceVertex() {
    return sourceVertex;
  }

  /**
   * Устанавливает начальную вершину {@code sourceVertex}.
   *
   * @param sourceVertex начальная вершина.
   */
  public void setSourceVertex(Vertex sourceVertex) {
```

```
        Logs.writeToLog("Source vertex of " + this.toString() + " changed to " + sourceVertex);

        this.sourceVertex = sourceVertex;
    }

    /**
     * Возвращает конечную вершину {@code targetVertex} .
     *
     * @return конечная вершина ребра.
     */
    public Vertex getTargetVertex() {
        return targetVertex;
    }

    /**
     * Устанавливает конечную вершину {@code targetVertex}.
     *
     * @param targetVertex конечная вершина.
     */
    public void setTargetVertex(Vertex targetVertex) {
        Logs.writeToLog("Target vertex of " + this.toString() + " changed to " + targetVertex);

        this.targetVertex = targetVertex;
    }

    /**
     * Возвращает строковое представление ребра {@code String}, которое содержит информацию о всех
     * его полях.
     *
     * @return строка, содержащая информацию о ребре.
     */
    @Override
    public String toString() {
        return "Edge{" +
                "source=" + sourceVertex +
                ", target=" + targetVertex +
                '}';
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
```

```
        if (o == null || getClass() != o.getClass()) return false;
        Edge edge = (Edge) o;
        return Objects.equals(sourceVertex, edge.sourceVertex) &&
                Objects.equals(targetVertex, edge.targetVertex);
    }

    @Override
    public int hashCode() {
        return Objects.hash(sourceVertex, targetVertex);
    }
}
```

Graph.java

```
package
graph;

import logger.Logs;
import org.jetbrains.annotations.NotNull;

import java.util.ArrayList;
import java.util.List;

/**
* Класс, представляющий собой граф.
* Содержит в себе список вершин {@code vertexList} и
* список ребер {@code edgeList}
* <p>
* Класс предоставляет методы для получения значений полей класса:
* {@code getVertexList}, {@code getEdgeList}.
* <p>
* Класс предоставляет методы для установки значений полей класса:
* {@code setVertexList}, {@code setEdgeList}.
* <p>
* Класс предоставляет метод для получения транспонированного графа {@code
getTransposedGraph}.
*/

public class Graph {
    /**
     * Список вершин графа.
     */
    private List<Vertex> vertexList;
    /**
     * Список ребер графа.
```

```
     */
    private List<Edge> edgeList;


    /**
     * Конструктор для создания пустого графа.
     */
    public Graph() {
        super();
    }


    /**
     * Конструктор графа, принимающий список вершин {@code vertexList} и
     * список ребер {@code edgeList}.
     *
     * @param vertexList список вершин.
     * @param edgeList   список ребер.
     */
    public Graph(@NotNull List<Vertex> vertexList, @NotNull List<Edge> edgeList) {
        super();
        Logs.writeToLog("Created graph with vertexes: " + vertexList.toString() +
                System.lineSeparator() + "edges:" + edgeList.toString());


        this.vertexList = vertexList;
        this.edgeList = edgeList;


        for (Edge edge : edgeList) {

vertexList.get(vertexList.indexOf(edge.getSourceVertex())).addNeighbour(edge.getTargetVertex())
;
        }
    }


    /**
     * Возвращает список вершин графа {@code vertexList}.
     *
     * @return список вершин графа.
     */
    public List<Vertex> getVertexList() {
        return vertexList;
    }


    /**
     * Устанавливает список вершин {@code vertexList}.
     *
```

```
     * @param vertexList список вершин.
     */
    public void setVertexList(@NotNull List<Vertex> vertexList) {
        Logs.writeToLog("Added list of vertexes to graph: " + vertexList.toString());

        this.vertexList = vertexList;
    }

    /**
     * Возвращает список ориентированных ребер графа {@code edgeList}.
     *
     * @return список ребер графа.
     */
    public List<Edge> getEdgeList() {
        return edgeList;
    }

    /**
     * Устанавливает список ребер {@code edgeList}.
     *
     * @param edgeList список ребер
     */
    public void setEdgeList(@NotNull List<Edge> edgeList) {
        Logs.writeToLog("Added list of edges to graph: " + edgeList.toString());

        this.edgeList = edgeList;
    }

    /**
     * Создает транспонированный граф (с тем же набором вершин и с теми же ребрами, что и у исходного,
     * но ориентация ребер этого графа противоположна ориентации ребер исходного графа)
     *
     * @return - транспонированный граф.
     */
    public Graph getTransposedGraph() {
        Logs.writeToLog("Created transposed graph with vertexes: " + vertexList.toString() +
                System.lineSeparator() + "edges:" + edgeList.toString());

        List<Vertex> newVertexList = new ArrayList<>();
        for (Vertex vertex : vertexList) {
            Vertex newVertex = new Vertex(vertex.getId());
            newVertex.setVisited(vertex.isVisited());
```

```
            newVertex.setComponentId(vertex.getComponentId());
            newVertexList.add(newVertex);
        }

        List<Edge> newEdgeList = new ArrayList<>();
        for (Edge edge : edgeList) {
            Vertex sourceVertex = null;
            Vertex targetVertex = null;

            for (Vertex vertex : newVertexList) {
                if (vertex.getId() == edge.getTargetVertex().getId()) {
                    sourceVertex = vertex;
                }
                if (vertex.getId() == edge.getSourceVertex().getId()) {
                    targetVertex = vertex;
                }
            }

            assert targetVertex != null;
            assert sourceVertex != null;
            Edge newEdge = new Edge(sourceVertex, targetVertex);
            newEdgeList.add(newEdge);
        }

        return new Graph(newVertexList, newEdgeList);
    }
}
```

Vertex.java

```
package
graph;

import logger.Logs;
import org.jetbrains.annotations.NotNull;

import java.util.ArrayList;
import java.util.List;
import java.util.Objects;

/**
* Класс, представляющий собой вершину графа.
* Содержит в себе информацию о номере {@code id} вершины,
* посещенности {@code isVisited}, о смежных вершинах {@code adjacencyList},
```

```
* и о номере компоненты сильной связности {@code componentId}, которой принадлежит вершина.
* <p>
* Класс предоставляет методы для получения значений полей класса:
* {@code getId}, {@code isVisited}, {@code getAdjacencyList}, {@code getComponentId}.
* <p>
* Класс предоставляет методы для установки значений полей класса:
* {@code setId}, {@code setVisited}, {@code setAdjacencyList}, {@code setComponentId}.
* <p>
* Класс предоставляет метод для добавления смежных вершин {@code addNeighbour}.
*/


public class Vertex {
  /**
   * Номер вершины.
   */
  private int id;


  /**
   * Статус посещенности вершины.
   */
  private boolean isVisited;


  /**
   * Список смежных вершин.
   */
  private List<Vertex> adjacencyList;


  /**
   * Номер компоненты сильной связности, к которой принадлежит вершина.
   */
  private int componentId;


  /**
   * Конструктор вершины, принимающий ее номер {@code id}.
   *
   * @param id номер вершины.
   */
  public Vertex(int id) {
     Logs.writeToLog("Created vertex number " + id);


     this.id = id;
     this.adjacencyList = new ArrayList<>();
```

```
}

/**
 * Возвращает {@code id} вершины.
 *
 * @return номер вершины.
 */
public int getId() {
    return id;
}

/**
 * Устанавливает номер вершины {@code id}.
 *
 * @param id номер вершины.
 */
public void setId(int id) {
    Logs.writeToLog("Vertex number changed from " + this.id + " to " + id);

    this.id = id;
}

/**
 * Возвращает {@code true}, если вершина была помечена.
 *
 * @return {@code true}, если вершина была помечена.
 */
public boolean isVisited() {
    return isVisited;
}

/**
 * Устанавливает значение статуса посещенности вершины {@code isVisited}.
 *
 * @param isVisited значение статуса посещенности вершины.
 */
public void setVisited(boolean isVisited) {
    Logs.writeToLog("Vertex " + this.id + " is visited. Status changes from " + this.isVisited +
            " to " + isVisited);

    this.isVisited = isVisited;
}
```

```
    /**
     * Возвращает список смежных вершин {@code adjacencyList}.
     *
     * @return список смежных вершин.
     */
    public List<Vertex> getAdjacencyList() {
        return adjacencyList;
    }

    /**
     * Устанавливает список смежных ребер {@code adjacencyList}.
     *
     * @param adjacencyList список смежных вершин.
     */
    public void setAdjacencyList(@NotNull List<Vertex> adjacencyList) {
        Logs.writeToLog("Vertex number " + this.id + " added adjacent vertices " +
adjacencyList.toString());

        this.adjacencyList = adjacencyList;
    }

    /**
     * Возвращает {@code componentId} вершины.
     *
     * @return номер компоненты сильной связности, к которой принадлежит вершина.
     */
    public int getComponentId() {
        return componentId;
    }

    /**
     * Устанавливает номер компоненты сильной связности {@code componentId}.
     *
     * @param componentId номер компоненты сильной связности.
     */
    public void setComponentId(int componentId) {
        Logs.writeToLog("Set component id " + componentId + " to vertex number " + this.id);

        this.componentId = componentId;
    }

    /**
```

```
    * Метод добавляет вершину {@code vertex} в список смежных вершин.
    *
    * @param vertex смежная вершина.
    */
    public void addNeighbour(@NotNull Vertex vertex) {
        Logs.writeToLog("Vertex number " + this.id + " added neighbour vertex " + vertex.id);
        this.adjacencyList.add(vertex);
    }


    /**
    * Возвращает строковое представление вершины {@code String}, которое содержит информацию о всех
    * ее полях.
    *
    * @return строка, содержащая информацию о вершине.
    */
    @Override
    public String toString() {
        StringBuilder stringBuilder = new StringBuilder();
        stringBuilder.append("[");
        for (int i = 0; i < adjacencyList.size(); ++i) {
            stringBuilder.append(adjacencyList.get(i).getId());


            if (i != adjacencyList.size() - 1) {
                stringBuilder.append(", ");
            }
        }
        stringBuilder.append("]");


        return "Vertex{" +
                "id=" + id +
                ", isVisited=" + isVisited +
                ", adjacencyList=" + stringBuilder +
                ", componentId=" + componentId +
                '}';
    }


    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Vertex vertex = (Vertex) o;
        return id == vertex.id &&
                isVisited == vertex.isVisited &&
```

```
                componentId == vertex.componentId;
        }

        @Override
        public int hashCode() {
            return Objects.hash(id, isVisited, componentId);
        }
    }
```

AlgorithmTests.java

```
package tests;

import graph.Algorithm;
import graph.Edge;
import graph.Graph;
import graph.Vertex;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;

import java.util.LinkedList;

public class AlgorithmTests {
    private Graph graph;
    private Algorithm algorithm;

    public void test() {
        LinkedList<Vertex> vertexList = new LinkedList<>();
        LinkedList<Edge> edgeList = new LinkedList<>();

        Vertex vertex1 = new Vertex(1);
        vertexList.add(vertex1);
        Vertex vertex2 = new Vertex(2);
        vertexList.add(vertex2);
        Vertex vertex3 = new Vertex(3);
        vertexList.add(vertex3);

        edgeList.add(new Edge(vertex1, vertex2));
        edgeList.add(new Edge(vertex2, vertex3));
        edgeList.add(new Edge(vertex3, vertex1));

        graph = new Graph(vertexList, edgeList);
        algorithm = new Algorithm(graph);
```

```
}

@Test
public void testAlgorithmAndGetGraph() {
    test();
    Assertions.assertSame(algorithm.getGraph(), graph);
}

@Test
public void testDoInBackGround1() {
    test();
    algorithm.offDelay();
    algorithm.run();
    Assertions.assertEquals(1, algorithm.getCount());
}

@Test
public void testDoInBackGround2() {
    LinkedList<Vertex> vertexList = new LinkedList<>();
    LinkedList<Edge> edgeList = new LinkedList<>();

    graph = new Graph(vertexList, edgeList);
    algorithm = new Algorithm(graph);

    Vertex vertex1 = new Vertex(1);
    vertexList.add(vertex1);
    Vertex vertex2 = new Vertex(2);
    vertexList.add(vertex2);
    Vertex vertex3 = new Vertex(3);
    vertexList.add(vertex3);

    algorithm.offDelay();
    algorithm.run();
    Assertions.assertEquals(3, algorithm.getCount());
}

@Test
public void testDoInBackGround3() {
    test();
    algorithm.getGraph().getEdgeList().clear();
    algorithm.getGraph().getVertexList().clear();
    algorithm.offDelay();
```

```
    algorithm.run();
    Assertions.assertEquals(0, algorithm.getCount());
}

public void test4() {
    LinkedList<Vertex> vertexList = new LinkedList<>();
    LinkedList<Edge> edgeList = new LinkedList<>();

    Vertex vertex1 = new Vertex(1);
    vertexList.add(vertex1);
    Vertex vertex2 = new Vertex(2);
    vertexList.add(vertex2);
    Vertex vertex3 = new Vertex(3);
    vertexList.add(vertex3);
    Vertex vertex4 = new Vertex(4);
    vertexList.add(vertex4);

    edgeList.add(new Edge(vertex1, vertex2));
    edgeList.add(new Edge(vertex2, vertex3));
    edgeList.add(new Edge(vertex3, vertex1));
    edgeList.add(new Edge(vertex3, vertex4));
    edgeList.add(new Edge(vertex4, vertex1));
    edgeList.add(new Edge(vertex2, vertex4));
    edgeList.add(new Edge(vertex4, vertex3));
    edgeList.add(new Edge(vertex1, vertex3));
    edgeList.add(new Edge(vertex1, vertex4));
    edgeList.add(new Edge(vertex2, vertex1));
    edgeList.add(new Edge(vertex3, vertex2));

    graph = new Graph(vertexList, edgeList);
    algorithm = new Algorithm(graph);
}

@Test
public void testDoInBackGround4() {
    test4();
    algorithm.offDelay();
    algorithm.run();
    Assertions.assertEquals(1, algorithm.getCount());
}

public void test5() {
    LinkedList<Vertex> vertexList = new LinkedList<>();
```

```
        LinkedList<Edge> edgeList = new LinkedList<>();

        Vertex vertex1 = new Vertex(1);
        vertexList.add(vertex1);
        Vertex vertex2 = new Vertex(2);
        vertexList.add(vertex2);
        Vertex vertex3 = new Vertex(3);
        vertexList.add(vertex3);
        Vertex vertex4 = new Vertex(4);
        vertexList.add(vertex4);
        Vertex vertex5 = new Vertex(5);
        vertexList.add(vertex5);
        Vertex vertex6 = new Vertex(6);
        vertexList.add(vertex6);

        edgeList.add(new Edge(vertex1, vertex2));
        edgeList.add(new Edge(vertex2, vertex3));
        edgeList.add(new Edge(vertex3, vertex1));

        edgeList.add(new Edge(vertex4, vertex5));
        edgeList.add(new Edge(vertex5, vertex4));

        graph = new Graph(vertexList, edgeList);
        algorithm = new Algorithm(graph);
    }

    @Test
    public void testDoInBackGround5() {
        test5();
        algorithm.offDelay();
        algorithm.run();
        Assertions.assertEquals(3, algorithm.getCount());
    }
}
```

EdgeTests.java

```
package tests;

import graph.Edge;
import graph.Vertex;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;
```

```
public class EdgeTests {
  @Test
  public Edge testEdge() {
     Vertex start = new Vertex(5);
     Vertex end = new Vertex(6);
     Edge edge = new Edge(start, end);
     Assertions.assertTrue(start.getId() == edge.getSourceVertex().getId() &&
          end.getId() == edge.getTargetVertex().getId());
     return edge;
  }

  @Test
  public void testSetGetSourceVertex() {
     Vertex v1 = new Vertex(5);
     Vertex v2 = new Vertex(6);
     Vertex source = new Vertex(3);
     Edge edge = new Edge(v1, v2);
     edge.setSourceVertex(source);

     Assertions.assertSame(source, edge.getSourceVertex());
  }

  @Test
  public void testSetGetTargetVertex() {
     Vertex v1 = new Vertex(5);
     Vertex v2 = new Vertex(6);
     Vertex vertex = new Vertex(3);
     Edge edge = new Edge(v1, v2);
     edge.setTargetVertex(vertex);

     Assertions.assertSame(vertex, edge.getTargetVertex());
  }
}
```

VertexTests.java

```
package tests;

import graph.Vertex;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
```

```
import java.util.LinkedList;

public class VertexTest {
  private LinkedList<Vertex> vertexList;
  private Vertex v2;

  @Test
  public void testVertex() {
     Vertex vertex = new Vertex(5);
     Assertions.assertEquals(5, vertex.getId());
  }

  @Test
  public void testSetGetId() {
     Vertex vertex = new Vertex(5);
     vertex.setId(1);
     Assertions.assertEquals(1, vertex.getId());
  }

  @Test
  public void testSetAndIsVisited() {
     Vertex vertex = new Vertex(5);
     vertex.setVisited(true);
     Assertions.assertTrue(vertex.isVisited());
  }

  @BeforeEach
  public void BTestSetGetAdjacencyList() {
     Vertex v1 = new Vertex(1);
     v2 = new Vertex(2);
     Vertex v3 = new Vertex(3);
     vertexList = new LinkedList<>();
     vertexList.add(v1);
     vertexList.add(v3);
     v2.setAdjacencyList(vertexList);
  }

  @Test
  public void testSetGetAdjacencyList() {
     Assertions.assertEquals(vertexList, v2.getAdjacencyList());
  }
```

```
    @Test
    public void testSetGetComponentId() {
        Vertex v = new Vertex(1);
        v.setComponentId(5);
        Assertions.assertEquals(5, v.getComponentId());
    }

    @Test
    public void testAddNeighbour() {
        Vertex v = new Vertex(1);
        v.addNeighbour(v);
        Assertions.assertEquals(v, v.getAdjacencyList().get(v.getAdjacencyList().size() - 1));
    }

    @Test
    public void testToString() {
        Vertex v = new Vertex(1);
        v.setVisited(true);
        v.setComponentId(5);
        Assertions.assertEquals("Vertex{id=1, isVisited=true," +
                " adjacencyList=[], componentId=5}", v.toString());
    }
}
```

GraphTests.java

```
package tests;

import graph.Edge;
import graph.Graph;
import graph.Vertex;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;

import java.util.LinkedList;

public class GraphTests {
    Vertex vertex1 = new Vertex(1);
    Vertex vertex2 = new Vertex(2);
    Vertex vertex3 = new Vertex(3);
    LinkedList<Vertex> vertexList = new LinkedList<>();
```

```
LinkedList<Edge> edgeList = new LinkedList<>();
Graph graph;
Graph graph1;


public void test() {
    vertexList.add(vertex1);
    vertexList.add(vertex2);
    vertexList.add(vertex3);


    edgeList.add(new Edge(vertex1, vertex2));
    edgeList.add(new Edge(vertex2, vertex3));
    edgeList.add(new Edge(vertex3, vertex1));


    graph = new Graph(vertexList, edgeList);
}


@Test
public void testGraph() {
    test();
    Assertions.assertEquals(vertexList, graph.getVertexList());
    Assertions.assertEquals(edgeList, graph.getEdgeList());
}


@Test
public void testSetGetVertexList() {
    test();
    Vertex vertex4 = new Vertex(4);
    vertexList.add(vertex4);
    graph.setVertexList(vertexList);


    Assertions.assertEquals(vertexList, graph.getVertexList());
}


@Test
public void testSetGetEdgeList() {
    test();
    edgeList.add(new Edge(vertex1, vertex3));
    graph.setEdgeList(edgeList);


    Assertions.assertEquals(edgeList, graph.getEdgeList());
}
```

```
    @Test
    public void testGetTransposedGraph() {
        test();
        graph1 = graph.getTransposedGraph();


        int i = 0;
        for (Edge edge1 : graph1.getEdgeList()) {
            Edge tmpEdge = new Edge(edge1.getTargetVertex(), edge1.getSourceVertex());
            for (Edge edge : graph.getEdgeList()) {
                if (tmpEdge.equals(edge)) {
                    i++;
                }
            }
        }


        Assertions.assertSame(i, graph.getEdgeList().size());
    }
}
```

Logs.java

```
package
logger;


import com.sun.tools.javac.Main;
import org.jetbrains.annotations.NotNull;


import java.sql.Time;
import java.util.logging.*;


/**
* Класс, реализующий логирование.
* Содержит в себе:
*
* @value logger - экземпляр класса Logger.
* @value handler - экзэмпляр класса Handler, экспортирующий сообщения логера на консоль.
* Класс, реализующий форматирвоание собщений {@code Forms}.
* <p>
* Класс предоставляет метод для получения значения уровня логирования, хранящегося в
поле класса {@code logger}:
* {@code getLevel}.
* <p>
* Класс предоставляет метод для установки уровня логирования, ниже которого сообщения
* не будут выводиться в консоль {@code setLevelOfOutput}, {@code setEdgeList}.
```

```
* <p>
* Класс предоставляет методы для записи сообщения в log {@code writeToLog}.
*/


public class Logs {
  private final static Logger logger = Logger.getLogger(Main.class.getName());
  private final static Handler handler = new ConsoleHandler();


  /**
   * Приватный конструктор, для исключения возможности создания
   * объекта класса.
   */
  private Logs() {
  }


  static {
     logger.setLevel(Level.ALL);
     logger.setUseParentHandlers(false);


     handler.setFormatter(new Forms());
     handler.setLevel(Level.ALL);
     logger.addHandler(handler);
  }


  /**
   * Метод для получения значения уровня, хранящегося у логера {@code logger}.
   *
   * @return возвращает значение уровня.
   */
  public static Level getLevel() {
     return logger.getLevel();
  }


  /**
   * Устанавливает уровень логирования, вниже которого логи не будут выводиться в
консоль.
   *
   * @param level уровень.
   */
  public static void setLogLevelForOutput(Level level) {
     handler.setLevel(level);
  }
```

```
    /**
     * Метод для записи сообщения в лог. Принимающий строку-сообщение {@code message}.
     *
     * @param message сообщение.
     */
    public static void writeToLog(String message) {
        logger.log(logger.getLevel(), Thread.currentThread().getStackTrace()[2].getClassName() + "::"
+
                Thread.currentThread().getStackTrace()[2].getMethodName() + "  \"" +
                message + "\"" + System.lineSeparator());
    }


    /**
     * Метод для записи сообщения в лог. Принимающий строку-сообщение {@code message}
и
     * уровень лога {@code level}.
     *
     * @param message сообщение.
     * @param level   уровень.
     */
    public static void writeToLog(String message, Level level) {
        logger.log(level, Thread.currentThread().getStackTrace()[2].getClassName() + "::" +
                Thread.currentThread().getStackTrace()[2].getMethodName() + " \"" +
                message + "\"" + System.lineSeparator());
    }


    /**
     * Класс, реализующий форматирование сообщений в логе.
     * <p>
     * Содержит в себе метод для форматирования {@code format}.
     */
    static class Forms extends Formatter {
        /**
         * Метод для форматирования сообщений.
         *
         * @return строка-сообщение в нужном формате.
         */
        @Override
        public String format(@NotNull LogRecord record) {
            return Time.from(record.getInstant()).toString() + System.lineSeparator() +
record.getLevel() + ": "
                    + record.getMessage() + System.lineSeparator();
        }
    }
}
```

Colors.java

```
package
GUI;

import java.awt.*;

/**
* Класс, используемый для удобного задания значений цветов в окне приложения, а также
* для задания цвета вершинам.
* <p>
* Солержит в себе:
*
* @value firstBackgroundColor - первый фоновый цвет
* @value secondBackgroundColor - второй фоновый цвет
* @value colorArray - массив различных цветов для окрашивания вершин
* <p>
* Имеет методы {@code getFirstBackgroundColor}, {@code getSecondBackgroundColor} для
получения первого фонового
* цвета и второго фонового цвета, а также метод {@code get} для получения цвета из массива
цветов
*/

public class Colors {
  private static final Color firstBackgroundColor = new Color(0XE6E6FA);
  private static final Color secondBackgroundColor = new Color(0xB0B0BB);

  private static final String[] colorArray = {
        "#FFC0CB", "#808080", "#004400", "#FF5C00",
        "#FF00FF", "#800080", "#FF0000", "#800000",
        "#FFFF00", "#808000", "#00FF00", "#008000",
        "#00FFFF", "#008080", "#0000FF", "#000080",
        "#8B4513", "#F08080", "#00FF7F", "#D2691E",
        "#DAA520", "#87CEEB", "#FFA07A", "#BDB76B",
        "#8A2BE2", "#1E90FF", "#111111", "#778899"
  };

  /**
   * Метод для получения цвета из массива цветов {@code colorArray}.
   *
   * @param index - индекс в массиве.
   * @return строковое представление цвета
   */
  public static String get(int index) {
     return colorArray[index];
  }
```

```
    /**
     * Метод, для получения значения {@code firstBackgroundColor}.
     *
     * @return первый фоновый цвет.
     */
    public static Color getFirstBackgroundColor() {
        return firstBackgroundColor;
    }

    /**
     * Метод, для получения значения {@code secondBackgroundColor}.
     *
     * @return второй фоновый цвет.
     */
    public static Color getSecondBackgroundColor() {
        return secondBackgroundColor;
    }

    /**
     * Метод, для получения размера массива цветов.
     *
     * @return размер массива цветов.
     */
    public static int size() {
        return colorArray.length;
    }

    /**
     * Приватный конструктор, для исключения возможности создания экземпляра класса вне его методов.
     */
    private Colors() {
    }
}
```

CommandPanel.java

```
package GUI;

import javax.imageio.ImageIO;
import javax.swing.*;
import java.awt.*;
import java.io.IOException;
```

```
/**
* Класс, представляющий собой коммандную панель.
* Наследуется от класса {@code JPanel}
*
* @value startButton - кнопка для запуска работы алгоритма и с помощью
* которой визуализацию можно поставить на паузу
* @value increaseSpeedButton - кнопка для увеличения скорости визуализаии
* @value decreaseSpeedButton - кнопка для уменьшения скорости визуализации
* @value addVertexButton - кнопка для добавления вершин
* @value deleteButton - кнопка для удаления выбранной вершины/ребра
* @value clearButton - кнопка для очищения окна
* @value stopButton - кнопка для остановки работы алгоритма
* @value progressBar - прогресс-бар, которой показывает текущий уровень скорости алгоритма
* @see JPanel
* <p>
* Содержит в себе экземпляры класса {@code JButton} и {@code JProgressBar}:
* @see JButton
* @see JProgressBar
*/


public class CommandPanel extends JPanel {
  private final JButton startPauseButton;
  private final JButton increaseSpeedButton;
  private final JButton decreaseSpeedButton;
  private final JButton addVertexButton;
  private final JButton deleteButton;
  private final JButton clearButton;
  private final JButton stopButton;
  private final JProgressBar progressBar;


  /**
   * Конструктор панели, который инициализирует переменные,
   * устанавливает свойства и размещает компоненты на панели.
   */
  CommandPanel() {
    double height = Toolkit.getDefaultToolkit().getScreenSize().getHeight();
    setBackground(Colors.getSecondBackgroundColor());
    setLayout(new BoxLayout(this, BoxLayout.PAGE_AXIS));
    setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));


    ImageIcon iconPlus = null;
    ImageIcon iconMinus = null;
    try {
```

```
    Image imagePlus = ImageIO.read(getClass().getResourceAsStream("/images/plus.png"));
    Image imageMinus = ImageIO.read(getClass().getResourceAsStream("/images/minus.png"));

    iconPlus = new ImageIcon(imagePlus.getScaledInstance(20,
            20, java.awt.Image.SCALE_SMOOTH));

    iconMinus = new ImageIcon(imageMinus.getScaledInstance(20,
            20, java.awt.Image.SCALE_SMOOTH));

} catch (IOException e) {
    e.printStackTrace();
}

startPauseButton = new JButton("Start");
stopButton = new JButton("Stop");
addVertexButton = new JButton("Add vertex");
deleteButton = new JButton("Delete");
clearButton = new JButton("Clear");
JLabel speedLabel = new JLabel("Speed:");
increaseSpeedButton = new JButton(iconPlus);
decreaseSpeedButton = new JButton(iconMinus);
progressBar = new JProgressBar();

startPauseButton.setAlignmentX(CENTER_ALIGNMENT);
stopButton.setAlignmentX(CENTER_ALIGNMENT);
addVertexButton.setAlignmentX(CENTER_ALIGNMENT);
deleteButton.setAlignmentX(CENTER_ALIGNMENT);
clearButton.setAlignmentX(CENTER_ALIGNMENT);

stopButton.setEnabled(false);

startPauseButton.setPreferredSize(new Dimension(0, (int) height / 20));
stopButton.setPreferredSize(new Dimension(0, (int) height / 20));
addVertexButton.setPreferredSize(new Dimension(0, (int) height / 20));
deleteButton.setPreferredSize(new Dimension(0, (int) height / 20));
clearButton.setPreferredSize(new Dimension(0, (int) height / 20));

startPauseButton.setMaximumSize(new Dimension(Integer.MAX_VALUE, (int) height / 15));
stopButton.setMaximumSize(new Dimension(Integer.MAX_VALUE, (int) height / 15));
addVertexButton.setMaximumSize(new Dimension(Integer.MAX_VALUE, (int) height / 15));
deleteButton.setMaximumSize(new Dimension(Integer.MAX_VALUE, (int) height / 15));
clearButton.setMaximumSize(new Dimension(Integer.MAX_VALUE, (int) height / 15));
```

```
        stopButton.setBackground(Colors.getFirstBackgroundColor());
        startPauseButton.setBackground(Colors.getFirstBackgroundColor());
        increaseSpeedButton.setBackground(Colors.getFirstBackgroundColor());
        decreaseSpeedButton.setBackground(Colors.getFirstBackgroundColor());
        deleteButton.setBackground(Colors.getFirstBackgroundColor());
        addVertexButton.setBackground(Colors.getFirstBackgroundColor());
        clearButton.setBackground(Colors.getFirstBackgroundColor());

        JPanel speedPanel = new JPanel();
        speedPanel.setLayout(new BoxLayout(speedPanel, BoxLayout.LINE_AXIS));

        increaseSpeedButton.setBorder(BorderFactory.createEmptyBorder(3, 3, 3, 3));
        decreaseSpeedButton.setBorder(BorderFactory.createEmptyBorder(3, 3, 3, 3));

        speedLabel.setBackground(Colors.getSecondBackgroundColor());
        speedPanel.setBackground(Colors.getSecondBackgroundColor());

        speedPanel.add(speedLabel);
        speedPanel.add(Box.createHorizontalGlue());
        speedPanel.add(decreaseSpeedButton);
        speedPanel.add(Box.createHorizontalGlue());
        speedPanel.add(increaseSpeedButton);
        speedPanel.add(Box.createHorizontalGlue());

        add(startPauseButton);
        add(Box.createVerticalGlue());
        add(stopButton);
        add(Box.createVerticalGlue());
        add(speedPanel);
        add(Box.createVerticalStrut(5));
        add(progressBar);
        add(Box.createVerticalGlue());
        add(Box.createVerticalGlue());
        add(Box.createVerticalGlue());
        add(addVertexButton);
        add(Box.createVerticalGlue());
        add(deleteButton);
        add(Box.createVerticalGlue());
        add(clearButton);
    }
```

```
    /**
     * Возвращает {@code addVertexButton}.
     *
     * @return кнопка для добавления вершин.
     */
    public JButton getAddVertexButton() {
        return addVertexButton;
    }

    /**
     * Возвращает {@code stopButton}.
     *
     * @return кнопка для остановки работы алгоритма.
     */
    public JButton getStopButton() {
        return stopButton;
    }

    /**
     * Возвращает {@code deleteButton}.
     *
     * @return кнопка для удаления выбранной вершины/ребра.
     */
    public JButton getDeleteButton() {
        return deleteButton;
    }

    /**
     * Возвращает {@code clearButton}.
     *
     * @return кнопка для очищения окна.
     */
    public JButton getClearButton() {
        return clearButton;
    }

    /**
     * Возвращает {@code startButton}.
     *
     * @return кнопка для запуска работы алгоритма и с помощью которой визуализацию можно
поставить на паузу.
     */
    public JButton getStartPauseButton() {
        return startPauseButton;
```

```
    }

    /**
     * Возвращает {@code decreaseSpeedButton}.
     *
     * @return кнопка для уменьшения скорости визуализации.
     */
    public JButton getDecreaseSpeedButton() {
        return decreaseSpeedButton;
    }

    /**
     * Возвращает {@code increaseSpeedButton}.
     *
     * @return кнопка для увеличения скорости визуализаии.
     */
    public JButton getIncreaseSpeedButton() {
        return increaseSpeedButton;
    }

    /**
     * Возвращает {@code progressBar}.
     *
     * @return прогресс-бар, которой показывает текущий уровень скорости алгоритма.
     */
    public JProgressBar getProgressBar() {
        return progressBar;
    }
}
```

Graph.java

```
package
GUI;

import com.mxgraph.model.mxCell;
import com.mxgraph.model.mxICell;
import com.mxgraph.util.mxConstants;
import com.mxgraph.view.mxGraph;

import java.util.*;

/**
* Класс, представляющий собой граф.
```

```
* Наследуется от {@code mxGraph}.
* <p>
* Содержит в себе:
*
* @value savedCellStyles - словарь для сохранения исходных стилей вершин
* @value savedEdges - словарь для сохранения исходных ребер
* @value cells - словарь вершин
* @value removedCells - множество удаленных вершин
* <p>
* Содержит в себе счетчик вершин {@code count}
* @see mxGraph
*/


public class Graph extends mxGraph {
  private final Map<Integer, String> savedCellStyles;
  private final Map<Object, String> savedEdgesStyles;
  private final Map<Integer, Object> cells;
  private final SortedSet<Integer> removedCells;
  private boolean isTransposed;


  /**
   * Счетчик вершин.
   */
  private int count;


  /**
   * Конструктор графа.
   * Инициализирует переменные, устанавливает свойства и задает стиль вершин и рёбер по умолчанию.
   */
  public Graph() {
    isTransposed = false;
    savedCellStyles = new HashMap<>();
    removedCells = new TreeSet<>();
    savedEdgesStyles = new HashMap<>();
    cells = new HashMap<>();
    count = 0;


    setAllowDanglingEdges(false);
    setMultigraph(false);
    setCellsEditable(false);
    setCellsResizable(false);


    Map<String, Object> vertexStyle = getStylesheet().getDefaultVertexStyle();
```

```
        Map<String, Object> edgeStyle = getStylesheet().getDefaultEdgeStyle();


        vertexStyle.put(mxConstants.STYLE_SHAPE, mxConstants.SHAPE_ELLIPSE);
        vertexStyle.put(mxConstants.STYLE_FONTSIZE, 16);
        vertexStyle.put(mxConstants.STYLE_FONTCOLOR, "#000000");
        vertexStyle.put(mxConstants.STYLE_STROKECOLOR, "#000000");
        vertexStyle.put(mxConstants.STYLE_FILLCOLOR, "#FFFFFF");


        mxConstants.DEFAULT_MARKERSIZE = 15;
        edgeStyle.put(mxConstants.STYLE_STROKECOLOR, "#000000");
        edgeStyle.put(mxConstants.STYLE_ENDARROW, mxConstants.ARROW_OPEN);
    }


    /**
     * Метод для добавления вершины в граф.
     * Добавление происходит следующим образом: если множество удаленных вершин пусто,
     * то добавляется новая вершина со значением {@code id} count, иначе добавляется первая среди удаленных.
     */
    public void insertVertex() {
        if (removedCells.isEmpty()) {
            cells.put(count, insertVertex(getDefaultParent(), null,
                    count, 0, 0, 40, 40));
            ++count;
        } else {
            int index = removedCells.first();
            removedCells.remove(index);
            cells.put(index, insertVertex(getDefaultParent(), null,
                    index, 0, 0, 40, 40));
        }
    }


    /**
     * Метод для удаления вершины по индексу {@code index}.
     *
     * @param index - индекс вершины в графе.
     */
    public void deleteVertex(int index) {
        removeCells(new Object[]{cells.get(index)});
        cells.remove(index);
        removedCells.add(index);
    }


    /**
```

```
     * Метод для возвращения массива всех вершин в графе.
     *
     * @return массив всех вершин графа.
     */
    public Object[] getAllVertex() {
        return cells.values().toArray();
    }


    /**
     * Метод, транспонирующий исходный граф.
     * Изменяет направление всех ребер графа на противоположное.
     */
    public void transpose() {
        for (Object vertex : getAllVertex()) {
            for (Object edge : getOutgoingEdges(vertex)) {
                mxCell mxEdge = ((mxCell) edge);


                mxICell target = mxEdge.getTarget();
                mxEdge.setTarget(mxEdge.getSource());
                mxEdge.setSource(target);
            }
        }


        isTransposed = !isTransposed;
    }


    /**
     * Метод для раскрашивания вершины с индексом {@code id} в цвет {@code color}
     *
     * @param id    - индекс вершины
     * @param color - строковое представление цвета для окрашивания вершины.
     */
    public void paintVertex(int id, String color) {
        ((mxCell) cells.get(id)).setStyle(mxConstants.STYLE_FILLCOLOR + "=" + color);
    }


    /**
     * Метод для раскрашивания ребра с индексом начальной вершины {@code idSource} и
конечной
     * вершины {@code idTarget}. Ребро окрашивается в цвет {@code color}.
     *
     * @param idSource - индекс начальной вершины ребра
     * @param idTarget - индекс конечной вершины ребра
     * @param color    - строковое представление цвета для окрашивания ребра
```

```
 */
public void paintEdge(int idSource, int idTarget, String color) {
    Object[] edges = getEdgesBetween(cells.get(idSource), cells.get(idTarget), true);
    for (Object edge : edges) {
        ((mxCell) edge).setStyle(mxConstants.STYLE_STROKECOLOR + "=" + color);
    }
}

/**
 * Метод для раскрашивания всех вершин в цвет, заданный по умолчанию.
 */
public void paintDefault() {
    for (Object cell : cells.values()) {
        for (Map.Entry<String, Object> entry : getStylesheet().getDefaultVertexStyle().entrySet()) {
            ((mxCell) cell).setStyle(entry.getKey() + "=" + entry.getValue());
        }
    }
}

/**
 * Метод для отчистки графа. Удаляет все вершины графа.
 */
public void clear() {
    removeCells(getAllVertex());
    count = 0;
    removedCells.clear();
    cells.clear();
}

/**
 * Метод, реализующий сохранение графа. В том числе: ребра графа и стиль вершин.
 */
public void save() {
    savedEdgesStyles.clear();
    savedCellStyles.clear();

    for (Object cell : cells.values()) {
        for (Object edge : getOutgoingEdges(cell)) {
            savedEdgesStyles.put(edge, ((mxCell) edge).getStyle());
        }
        savedCellStyles.put((Integer) ((mxCell) cell).getValue(), ((mxCell) cell).getStyle());
    }
}
```

```
/**
 * Метод для восстановления исходного графа.
 *
 * @see Graph#save()
 */
public void load() {
    if (isTransposed) {
        transpose();
    }

    for (int key : savedCellStyles.keySet()) {
        ((mxCell) cells.get(key)).setStyle(savedCellStyles.get(key));
    }

    for (Object cell : cells.values()) {
        for (Object edge : getOutgoingEdges(cell)) {
            ((mxCell) edge).setStyle(savedEdgesStyles.get(edge));
        }
    }
}

/**
 * Метод для создания графа по известному количеству вершин {@code count} и множеству
 * рёбер {@code edgeSet}.
 *
 * @param count   - количество вершин
 * @param edgeSet - множество рёбер
 */
public void createGraph(int count, Set<Pair<Integer, Integer>> edgeSet) {
    clear();
    for (int i = 0; i < count; ++i) {
        insertVertex();
    }

    for (Pair<Integer, Integer> elem : edgeSet) {
        insertEdge(getDefaultParent(), null, null, cells.get(elem.first),
                cells.get(elem.second));
    }
}

/**
 * Метод для обновления значений вершин.
```

```
     *
     * @param id    - номер вершины
     * @param order - порядковый номер вершины
     */
    public void setVertexValue(int id, int order) {
        ((mxCell) cells.get(id)).setValue(id + " (" + order + ")");
    }


    /**
     * Метод для установки значений вершин по умолчанию.
     */
    public void resetVertexValues() {
        for (Integer key : cells.keySet()) {
            ((mxCell) cells.get(key)).setValue(key);
        }
    }
}
```

MainWindow.java

```
package
GUI;


import com.mxgraph.layout.mxCircleLayout;
import com.mxgraph.model.mxCell;
import com.mxgraph.swing.mxGraphComponent;
import com.mxgraph.swing.util.mxSwingConstants;
import com.mxgraph.util.mxConstants;
import com.mxgraph.util.mxEvent;
import graph.Algorithm;
import graph.Edge;
import graph.Vertex;
import org.jetbrains.annotations.Contract;
import org.jetbrains.annotations.NotNull;


import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.ComponentAdapter;
import java.awt.event.ComponentEvent;
import java.beans.PropertyChangeEvent;
import java.beans.PropertyChangeListener;
import java.io.FileReader;
import java.io.IOException;
```

```
import java.util.*;


/**
* Класс основного окна. Наследуется от JFrame
*
* @see JFrame
* <p>
* Содержит в себе: алгоритм {@code algorithm}, коммандную панель {@code commandPanel},
граф {@code graph},
* меню {@code menuBar}, текстовое поле {@code scrollTextPane}, {@code layout},
* {@code graphComponent}, длину {@code height} и ширину {@code width}.
* <p>
* Имеет метод инициализации {@code init}, а также методы инициализации меню {@code
initMenuBar}, текстового
* поля {@code initScrollTextPane}, графа {@code initGraph}, коммандной панели {@code
initCommandPanel} и
* алгоритма {@code initAlgorithm}.
* Содержит метод для установки состояния кнопкам при остановке {@code
setButtonsStateWhenStop}
*/
public class MainWindow extends JFrame {
  private boolean isRun;
  /**
   * Алгоритм.
   */
  private Algorithm algorithm;
  /**
   * Коммандная панель.
   */
  private CommandPanel commandPanel;
  /**
   * Граф.
   */
  private Graph graph;
  /**
   * Меню.
   */
  private MenuBar menuBar;
  /**
   * Прокручивающееся текстовое поле.
   */
  private ScrollTextPane scrollTextPane;
  /**
   * Поле, значением которого будет true в случае, когда выполнение поставлено на паузу.
   */
  private boolean isPaused;
  /**
```

```
     * Высота окна.
     */
    private int height;
    /**
     * Ширина окна.
     */
    private int width;
    private mxCircleLayout layout;
    private mxGraphComponent graphComponent;


    /**
     * Конструктор. Вызывает метод инициализации {@code init}.
     */
    public MainWindow() {
        init();
    }


    /**
     * Метод инициализации. Инициализирует поля класса, устанавливает минимальный размер
окна.
     * Вызывает методы инициализации графа, коммандной панели, меню и текстового поля.
     * Устанавливает меню, добавляет текстовое поле, команндную панель.
     */
    private void init() {
        setTitle("Kosaraju's algorithm visualizer");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);


        Dimension dimension = Toolkit.getDefaultToolkit().getScreenSize();
        width = dimension.width;
        height = dimension.height;


        isRun = false;


        setBounds(width / 6, height / 6, 2 * width / 3, 2 * height / 3);
        setMinimumSize(new Dimension(2 * width / 3, height / 2));
        setLayout(new BoxLayout(getContentPane(), BoxLayout.X_AXIS));


        menuBar = new MenuBar();
        commandPanel = new CommandPanel();
        scrollTextPane = new ScrollTextPane();
        graph = new Graph();
        graphComponent = new mxGraphComponent(graph);
```

```
        initGraph();
        initCommandPanel();
        initMenuBar();
        initScrollTextPane();


        setJMenuBar(menuBar);
        add(scrollTextPane);
        add(graphComponent);
        add(commandPanel);
    }


    /**
     * Метод инициализации меню. Добавляет слушателей для кнопок Open, About, Help.
     */
    private void initMenuBar() {
        menuBar.getOpen().addActionListener(e -> {
            JFileChooser fileChooser = new JFileChooser();
            int result = fileChooser.showOpenDialog(this);


            if (result == JFileChooser.APPROVE_OPTION) {
                try (FileReader fileReader = new FileReader(fileChooser.getSelectedFile())) {
                    Scanner scanner = new Scanner(fileReader);


                    Set<Pair<Integer, Integer>> edgeSet = new HashSet<>();


                    int count = scanner.nextInt();


                    if (count > Colors.size() || count < 0) {
                        throw new IndexOutOfBoundsException("Incorrect count of vertexes");
                    }


                    while (scanner.hasNext()) {
                        int source = scanner.nextInt();
                        if (source >= Colors.size() || source < 0) {
                            throw new IndexOutOfBoundsException("Incorrect source vertex: " + source);
                        }
                        int target = scanner.nextInt();
                        if (target >= Colors.size() || target < 0) {
                            throw new IndexOutOfBoundsException("Incorrect target vertex: " + target);
                        }
                        edgeSet.add(new Pair<>(source, target));
                    }
```

```
                graph.createGraph(count, edgeSet);
                executeGraph();
            } catch (Exception exception) {
                exception.printStackTrace();
                scrollTextPane.getTextArea().setText("Can't read data from file");
            }
        }
    });

    menuBar.getAbout().addActionListener(e -> {
        Scanner scanner = new Scanner(getClass().getResourceAsStream("/HTML/about.html"));
        StringBuilder sb = new StringBuilder();
        while (scanner.hasNextLine()) {
            sb.append(scanner.nextLine());
        }
        JOptionPane.showMessageDialog(this,
                sb.toString(), "About", JOptionPane.PLAIN_MESSAGE);
    });

    menuBar.getHelp().addActionListener(e -> {
        JEditorPane editorPane = null;
        try {
            editorPane = new JEditorPane(getClass().getResource("/HTML/help.html"));
        } catch (IOException ioException) {
            ioException.printStackTrace();
        }

        assert editorPane != null;
        editorPane.setEditable(false);
        editorPane.setMaximumSize(new Dimension(width / 2, height / 2));

        JScrollPane scrollPane = new JScrollPane(editorPane);
        scrollPane.setPreferredSize(new Dimension(width / 2, 2 * height / 3));
        editorPane.setBackground(new Color(0xEEEEEE));
        JOptionPane.showMessageDialog(this,
                scrollPane, "Help", JOptionPane.PLAIN_MESSAGE);
    });

    menuBar.getSort().addActionListener(e -> layout.execute(graph.getDefaultParent()));
}
```

```
    /**
     * Метод инициализации текстового поля. Задает свойства поля.
     */
    private void initScrollTextPane() {
        scrollTextPane.getTextArea().setFocusable(false);
        scrollTextPane.setMaximumSize(new Dimension(width / 5, height));
        scrollTextPane.setMinimumSize(new Dimension(width / 5, height));
        scrollTextPane.setPreferredSize(new Dimension(width / 5, height));
    }


    /**
     * Метод инициализации графа. Добавляет слушателей. Задает свойства graphComponent (цвет, размер, границы).
     */
    private void initGraph() {
        layout = new mxCircleLayout(graph);


        addComponentListener(new ComponentAdapter() {
            @Override
            public void componentResized(ComponentEvent e) {
                int radius = Math.min(graphComponent.getHeight(), graphComponent.getWidth()) / 2 - 40;
                layout.setRadius(radius);
                layout.setX0(graphComponent.getWidth() / 2.0 - radius - 30);
                layout.setY0(graphComponent.getHeight() / 2.0 - radius - 30);
                executeGraph();
            }
        });


        graphComponent.getViewport().setBackground(Colors.getSecondBackgroundColor());
        graphComponent.setBackground(Colors.getSecondBackgroundColor());
        graphComponent.setBorder(BorderFactory.createEmptyBorder(10, 0, 10, 0));
        mxSwingConstants.VERTEX_SELECTION_COLOR = Colors.getSecondBackgroundColor();


        graph.getSelectionModel().addListener(mxEvent.CHANGE, (o, mxEventObject) -> {
            ArrayList<mxCell> list = (ArrayList<mxCell>) mxEventObject.getProperty("added");
            if (list != null) {
                for (mxCell elem : list) {
                    elem.setStyle(mxConstants.STYLE_FILLCOLOR + "=#FFFFFF");
                }
            }


            list = (ArrayList<mxCell>) mxEventObject.getProperties().get("removed");
            if (list != null) {
                for (mxCell elem : list) {
```

```
                elem.setStyle(mxConstants.STYLE_FILLCOLOR + "=#DAA520");
            }
        }


        graph.refresh();
        graph.repaint();
    });


    graphComponent.setMaximumSize(new Dimension(width, height));
}


/**
 * Метод, инициализирующий коммандную панель. Задает свойства коммандной панели. Добавляет слушателей для
 * кнопки увеличения/уменьшения скорости, остановки, добавления вершины, отчистки графа, паузы, удаления, старта.
 */
private void initCommandPanel() {
    commandPanel.getProgressBar().setMinimum(0);
    commandPanel.getProgressBar().setMaximum(Algorithm.MAX_DELAY + Algorithm.DELTA_DELAY);
    commandPanel.getProgressBar().setValue((Algorithm.MAX_DELAY +
            Algorithm.MIN_DELAY + Algorithm.DELTA_DELAY) / 2);


    commandPanel.getIncreaseSpeedButton().addActionListener(e -> {
        if (isRun || isPaused) {
            algorithm.decreaseDelay();
        }
        int value = commandPanel.getProgressBar().getValue();
        if (value + Algorithm.DELTA_DELAY <= Algorithm.MAX_DELAY) {
            commandPanel.getProgressBar().setValue(value + Algorithm.DELTA_DELAY);
        }
    });


    commandPanel.getDecreaseSpeedButton().addActionListener(e -> {
        if (isRun || isPaused) {
            algorithm.increaseDelay();
        }
        int value = commandPanel.getProgressBar().getValue();
        if (value - Algorithm.DELTA_DELAY > Algorithm.MIN_DELAY) {
            commandPanel.getProgressBar().setValue(value - Algorithm.DELTA_DELAY);
        }
    });
```

```
        commandPanel.getStopButton().addActionListener(e -> {
            scrollTextPane.getTextArea().setText("");
            for (PropertyChangeListener listener :
algorithm.getPropertyChangeSupport().getPropertyChangeListeners()) {
                algorithm.removePropertyChangeListener(listener);
            }
            algorithm.cancel(true);
            graphComponent.setEnabled(true);
            graph.resetVertexValues();
            isPaused = false;
            isRun = false;
            graph.load();
            setButtonsStateWhenStop(true);
            executeGraph();
        });


        commandPanel.getAddVertexButton().addActionListener(e -> {
            graph.insertVertex();
            executeGraph();
        });


        commandPanel.getClearButton().addActionListener(e -> {
            graph.clear();
            scrollTextPane.getTextArea().setText("");
            executeGraph();
        });


        commandPanel.getDeleteButton().addActionListener(e -> {
            Object cell = graph.getSelectionCell();
            if (graph.getModel().isVertex(cell)) {
                graph.deleteVertex((Integer) graph.getModel().getValue(cell));
            } else if (graph.getModel().isEdge(cell)) {
                graph.removeCells();
            }
            executeGraph();
        });


        commandPanel.getStartPauseButton().addActionListener(e -> {
            if (!isRun) {
                commandPanel.getStartPauseButton().setText("Pause");
                isRun = true;
                if (isPaused) {
                    algorithm.setDelay(Algorithm.MAX_DELAY -
commandPanel.getProgressBar().getValue() + 50);
                    algorithm.unSleep();
```

```
                graphComponent.setEnabled(false);
                isPaused = false;
            } else {
                graph.paintDefault();
                graph.save();
                graphComponent.setEnabled(false);
                graph.setSelectionCells(new Object[]{});
                algorithm = new Algorithm(createGraph(),
                        Algorithm.MAX_DELAY - commandPanel.getProgressBar().getValue() + 50);
                initAlgorithm();
                algorithm.execute();
            }
            setButtonsStateWhenStop(false);
        } else {
            commandPanel.getStartPauseButton().setText("Start");
            isRun = false;
            graphComponent.setEnabled(true);
            algorithm.setRun(false);
            isPaused = true;
        }
    });


    commandPanel.setMaximumSize(new Dimension(width / 7, height));
}


/**
 * Метод, инициализирующий алгоритм. Добавляет слушателей для следующих событий:
 * транспонирвоание графа, окончание работы алгоритма, добавление текста, помечания
отработанной вершины, ребра,
 * отчистки текстового поля.
 */
private void initAlgorithm() {
    algorithm.addPropertyChangeListener(evt -> {
        if (evt.getPropertyName().equals(Algorithm.TRANSPOSE_GRAPH)) {
            graph.transpose();
            executeGraph();
        }
    });


    algorithm.addPropertyChangeListener(evt -> {
        if (evt.getPropertyName().equals(Algorithm.ALGORITHM_ENDED)) {
            for (Vertex vertex : algorithm.getGraph().getVertexList()) {
                graph.paintVertex(vertex.getId(), Colors.get(vertex.getComponentId()));
            }
            isPaused = false;
```

```
                isRun = false;
                setButtonsStateWhenStop(true);
                graphComponent.setEnabled(true);
                executeGraph();
            }
        });

        algorithm.addPropertyChangeListener(evt -> {
            if (evt.getPropertyName().equals(Algorithm.ADD_TEXT)) {
                scrollTextPane.getTextArea().append(evt.getNewValue().toString());
            }
        });

        algorithm.addPropertyChangeListener(evt -> {
            if (evt.getPropertyName().equals(Algorithm.MARK_VISITED_VERTEX)) {
                if (evt.getOldValue() != null) {
                    graph.paintVertex(((Vertex) evt.getNewValue()).getId(), Colors.get((Integer)
evt.getOldValue()));
                } else {
                    graph.paintVertex(((Vertex) evt.getNewValue()).getId(), "#DAA520");
                }
            }
            executeGraph();
        });

        algorithm.addPropertyChangeListener(evt -> {
            if (evt.getPropertyName().equals(Algorithm.MARK_FINISHED_VERTEX)) {
                graph.paintVertex(((Vertex) evt.getNewValue()).getId(), "#B8860B");
            }
            executeGraph();
        });

        algorithm.addPropertyChangeListener(evt -> {
            if (evt.getPropertyName().equals(Algorithm.UNMARK_VERTEX)) {
                graph.paintVertex(((Vertex) evt.getNewValue()).getId(), "#FFFFFF");
            }
            executeGraph();
        });

        algorithm.addPropertyChangeListener(evt -> {
            if (evt.getPropertyName().equals(Algorithm.MARK_VISITED_EDGE)) {
                graph.paintEdge(((Vertex) evt.getOldValue()).getId(),
                        ((Vertex) evt.getNewValue()).getId(), "#B8860B");
            }
```

```
            executeGraph();
        });

        algorithm.addPropertyChangeListener(evt -> {
            if (evt.getPropertyName().equals(Algorithm.MARK_UNVISITED_EDGE)) {
                graph.paintEdge(((Vertex) evt.getOldValue()).getId(),
                        ((Vertex) evt.getNewValue()).getId(), "#FFDEAD");
            }
            executeGraph();
        });

        algorithm.addPropertyChangeListener(evt -> {
            if (evt.getPropertyName().equals(Algorithm.UNMARK_EDGE)) {
                graph.paintEdge(((Vertex) evt.getOldValue()).getId(),
                        ((Vertex) evt.getNewValue()).getId(), "#000000");
            }
            executeGraph();
        });

        algorithm.addPropertyChangeListener(evt -> {
            if (evt.getPropertyName().equals(Algorithm.SET_VERTEX_VALUE)) {
                Pair<Integer, Integer> pair = (Pair<Integer, Integer>) evt.getNewValue();
                graph.setVertexValue(pair.first, pair.second);
            }
            executeGraph();
        });

        algorithm.addPropertyChangeListener(evt -> {
            if (evt.getPropertyName().equals(Algorithm.RESET_VERTEX_VALUES)) {
                graph.resetVertexValues();
            }
            executeGraph();
        });

        algorithm.addPropertyChangeListener(evt -> {
            if (evt.getPropertyName().equals(Algorithm.CLEAR_TEXT_PANE)) {
                scrollTextPane.getTextArea().setText("");
            }
        });
    }

    /**
     * Метод, устанавливающий статус кнопкам во время остановки.
```

```
     *
     * @param isStop {@code true}, если работа алгоритма остановлена.
     */
    private void setButtonsStateWhenStop(boolean isStop) {
        commandPanel.getStopButton().setEnabled(!isStop);

        commandPanel.getAddVertexButton().setEnabled(isStop);
        commandPanel.getClearButton().setEnabled(isStop);
        commandPanel.getDeleteButton().setEnabled(isStop);

        if (isStop) {
            commandPanel.getStartPauseButton().setText("Start");
        } else {
            commandPanel.getStartPauseButton().setText("Pause");
        }
    }

    /**
     * Метод, перерисовывающий граф.
     */
    private void executeGraph() {
        graph.refresh();
        graph.repaint();
    }

    /**
     * Метод создания алгоритмического графа на основе визуализированного.
     *
     * @return алгоритмический граф.
     */
    @Contract(" -> new")
    private graph.@NotNull Graph createGraph() {
        Map<Integer, Vertex> vertexMap = new HashMap<>();
        LinkedList<Vertex> vertexList = new LinkedList<>();
        LinkedList<Edge> edgeList = new LinkedList<>();

        for (Object cell : graph.getAllVertex()) {
            int source = (Integer) ((mxCell) cell).getValue();
            Vertex vertex = new Vertex(source);
            vertexList.add(vertex);
            vertexMap.put(source, vertex);
        }
```

```
        for (Object cell : graph.getAllVertex()) {
            int source = (Integer) ((mxCell) cell).getValue();

            for (Object edge : graph.getOutgoingEdges(cell)) {
                int target = (Integer) ((mxCell) edge).getTarget().getValue();
                edgeList.add(new Edge(vertexMap.get(source), vertexMap.get(target)));
            }
        }

        return new graph.Graph(vertexList, edgeList);
    }
}
```

MenuBar.java

```
package GUI;

import org.jetbrains.annotations.NotNull;

import javax.swing.*;
import java.awt.event.ActionEvent;

/**
* Класс, представляющий собой строку меню.
* Наследуется от {@code JMenuBar}.
*
* Содержит в себе экземпляры класса {@code JMenuItem}:
* @value open - кнопка для выбора файла для ввода графа из файла.
* @value exit - кнопка для выхода из приложения.
* @value help - кнопка для вызова справки.
* @value about - кнопка для вызова информации о программе.
*
* Содержит методы для создания меню File {@code createFileMenu} и меню Help {@code
createHelpMenu}.
* А также методы, возвращающие значение полей:
* {@code getHelp}, {@code getAbout}, (@code getOpen}
*/
public class MenuBar extends JMenuBar {
    private final JMenuItem open;
    private final JMenuItem exit;
    private final JMenuItem sort;
    private final JMenuItem help;
    private final JMenuItem about;
```

```
    /**
     * Конструктор, устанавливающий фоновый цвет строке меню, инициализирующий поля класса и
     * и добавляющий два экземпляра класса JMenu {@code add()} на строку меню.
     */
    public MenuBar() {
        setBackground(Colors.getFirstBackgroundColor());


        sort = new JMenuItem("Sort graph");
        open = new JMenuItem("Open");
        exit = new JMenuItem(new ExitAction());
        help = new JMenuItem("Help");
        about = new JMenuItem("About");


        add(createFileMenu());
        add(createHelpMenu());
    }




    /**
     * Метод, создающий {@code file} экземпляр класса JMenu. Добавляет в него {@code open}, {@code exit}.
     * @see JMenu
     *
     * @return меню File
     */
    private @NotNull JMenu createFileMenu() {
        JMenu file = new JMenu("File");


        file.add(open);
        file.add(sort);
        file.addSeparator();
        file.add(exit);


        return file;
    }


    /**
     * Метод, создающий {@code help} экземпляр класса JMenu. Добавляет в него {@code help}, {@code about}.
     * @see JMenu
     *
     * @return меню Help.
     */
```

```
private @NotNull JMenu createHelpMenu() {
    JMenu help = new JMenu("Help");


    help.add(this.help);
    help.add(about);


    return help;
}


/**
 * Метод, возвращающий значение поля {@code open}.
 *
 * @return кнопка open.
 */
public JMenuItem getOpen() {
    return open;
}


/**
 * Метод, возвращающий значение поля {@code help}.
 *
 * @return кнопка kelp.
 */
public JMenuItem getHelp() {
    return help;
}


/**
 * Метод, возвращающий значение поля {@code about}.
 *
 * @return кнопка about.
 */
public JMenuItem getAbout() {
    return about;
}


/**
 * Класс-наследник AbstractAction, используется при инициализации кнопки {@code exit}.
 * @see AbstractAction
 */
private static class ExitAction extends AbstractAction {
    ExitAction() {
        putValue(NAME, "Exit");
```

```
            }

            public void actionPerformed(ActionEvent e) {
                System.exit(0);
            }
        }

        public JMenuItem getSort() {
            return sort;
        }
    }
```

Pair.java

```
package GUI;
import java.util.Objects;

/**
* Класс-контейнер, представляющий собой пару объектов.
*/

public class Pair<T, U> {
  public final T first;
  public final U second;

  /**
   * Конструктор, принимающий два объекта {@code first} и {@code second}.
   */
  public Pair(T first, U second) {
     this.first = first;
     this.second = second;
  }

  /**
   * Перегруженный метод {@code equals}.
   *
   * @see Object#equals(Object)
   */
  @Override
  public boolean equals(Object o) {
     if (this == o) return true;
     if (o == null || getClass() != o.getClass()) return false;
```

```
        Pair<?, ?> pair = (Pair<?, ?>) o;
        return Objects.equals(first, pair.first) &&
                Objects.equals(second, pair.second);
    }

    /**
     * Перегруженный метод {@code hashCode}.
     *
     * @see Object#hashCode()
     */
    @Override
    public int hashCode() {
        return Objects.hash(first, second);
    }
}
```

ScrollTextPane.java

```
package GUI;

import javax.swing.*;

/**
* Класс, представляющий собой прокручиваемую область,
* на которую помещается {@code textArea}.
* Наследуется от класса {@code JScrollPane}.
*
* @value textArea - текстовая панель
* @see JScrollPane
* <p>
* Содержит в себе экземпляр класса {@code JTextArea}:
* @see JTextArea
*/

public class ScrollTextPane extends JScrollPane {
    private final JTextArea textArea;

    /**
     * Конструктор панели, который инициализирует переменные,
     * устанавливает свойства и размещает компоненты на панели.
     */
    ScrollTextPane() {
        textArea = new JTextArea();
        textArea.setLineWrap(true);
        textArea.setWrapStyleWord(true);
```

```
        setBackground(Colors.getSecondBackgroundColor());
        setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));
        textArea.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));


        setViewportView(textArea);
        textArea.setBackground(Colors.getFirstBackgroundColor());
    }


    /**
     * Возвращает значение {@code textArea}.
     *
     * @return текстовая панель.
     */
    public JTextArea getTextArea() {
        return textArea;
    }
}
```