

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

ОТЧЕТ
по учебной практике
Тема: Визуализация алгоритма Флойда-Уоршелла

Студент гр. 8304	_____	Рыжиков А.В.
Студент гр. 8304	_____	Мешков М.А.
Студент гр. 8382	_____	Колногоров Д.Г.
Руководитель	_____	Ефремов М.А.

Санкт-Петербург

2020

ЗАДАНИЕ НА УЧЕБНУЮ ПРАКТИКУ

Студент Рыжиков А.В. группы 8304

Студент Мешков М.А. группы 8304

Студент Колногоров Д.Г. группы 8382

Тема практики: Визуализация алгоритма Флойда-Уоршелла

Задание на практику:

Командная итеративная разработка визуализатора алгоритма(ов) на Java с графическим интерфейсом.

Алгоритм: Флойда-Уоршелла.

Сроки прохождения практики: 29.06.2020 – 12.07.2020

Дата сдачи отчета: 10.07.2020

Дата защиты отчета: 12.07.2020

Студент	_____	Рыжиков А.В.
Студент	_____	Мешков М.А.
Студент	_____	Колногоров Д.Г.
Руководитель	_____	Ефремов М.А.

АННОТАЦИЯ

Целью практической работы является итеративная разработка программы для визуализации работы алгоритма Флойда-Уоршелла. Пользователю программы должна быть представлена возможность самостоятельно задать входные данные для алгоритма с помощью графического интерфейса, а также возможность пошагового исполнения алгоритма с просмотром промежуточных состояний алгоритма (матрицы смежности). Разработка осуществляется с использованием языка программирования Java командой из трех человек.

SUMMARY

The purpose of practical work is iterative development of the program for visualization of Floyd-Warshell algorithm. The user of the program should be provided with the possibility to set the input data for the algorithm using the GUI, as well as the possibility of step-by-step execution of the algorithm with viewing the intermediate states of the algorithm (adjacency matrix). The development is performed using Java programming language by a team of three people.

СОДЕРЖАНИЕ

	Введение	5
1.	Требования к программе	6
1.1.	Требования к вводу исходных данных	6
1.2.	Требования к визуализации	7
2.	План разработки и распределение ролей в бригаде	8
2.1.	План разработки	8
2.2.	Распределение ролей в бригаде	9
3.	Особенности реализации	10
3.1.	Реализуемые классы	10
3.2.	Основные методы	11
4.	Тестирование	12
4.1	План тестирования	12
	Заключение	14
	Список использованных источников	15
	Приложение А. Исходный код алгоритма	16

ВВЕДЕНИЕ

Целью практической работы является итеративная разработка программы для визуализации работы алгоритма Флойда-Уоршелла для нахождения кратчайших расстояний между всеми вершинами взвешенного ориентированного графа. Пользователю программы должна быть представлена возможность самостоятельно задать входные данные для алгоритма с помощью графического интерфейса, а также возможность пошагового исполнения алгоритма с просмотром промежуточных состояний алгоритма (матрицы смежности). В конечной версии программы должна быть предусмотрена возможность сохранения и загрузки исходных данных для алгоритма из файла.

Разработка осуществляется с использованием языка программирования Java командой из трех человек. Каждому участнику команды наанчается роль и выполняемые им задачи (разработка интерфейса, логики алгоритма и тестирование программы). В результате выполнения работы должна быть представлена программа, без ошибок собирающаяся из исходных файлов и протестированная на корректность.

1. ТРЕБОВАНИЯ К ПРОГРАММЕ

1.1. Требования к вводу исходных данных.

Должны быть реализованы два способа ввода исходных данных для алгоритма (граф):

- Чтение графа из файла с проверкой формата данных на корректность (в случае некорректного формата данных пользователю должна выводиться соответствующая информация).
- Создание графа в пользовательском интерфейсе (возможно ограничение на максимальное количество вершин в графе).

1.2. Требования к визуализации.

Пользовательский интерфейс должен состоять из двух частей:

- Окно для создания и модификации графа. Пользователю предоставляется возможность создавать и удалять вершины и рёбра, а также задавать вес каждого ребра.
- Окно для просмотра процесса выполнения алгоритма. Пользователю предоставляется возможность выполнения одного или нескольких шагов алгоритма и просмотра внутреннего состояния алгоритма.

Также должен быть реализован функционал выбора файла, из которого загружается граф или в который сохраняется созданный пользователем граф.

На рисунке 1 представлен макет интерфейса программы.

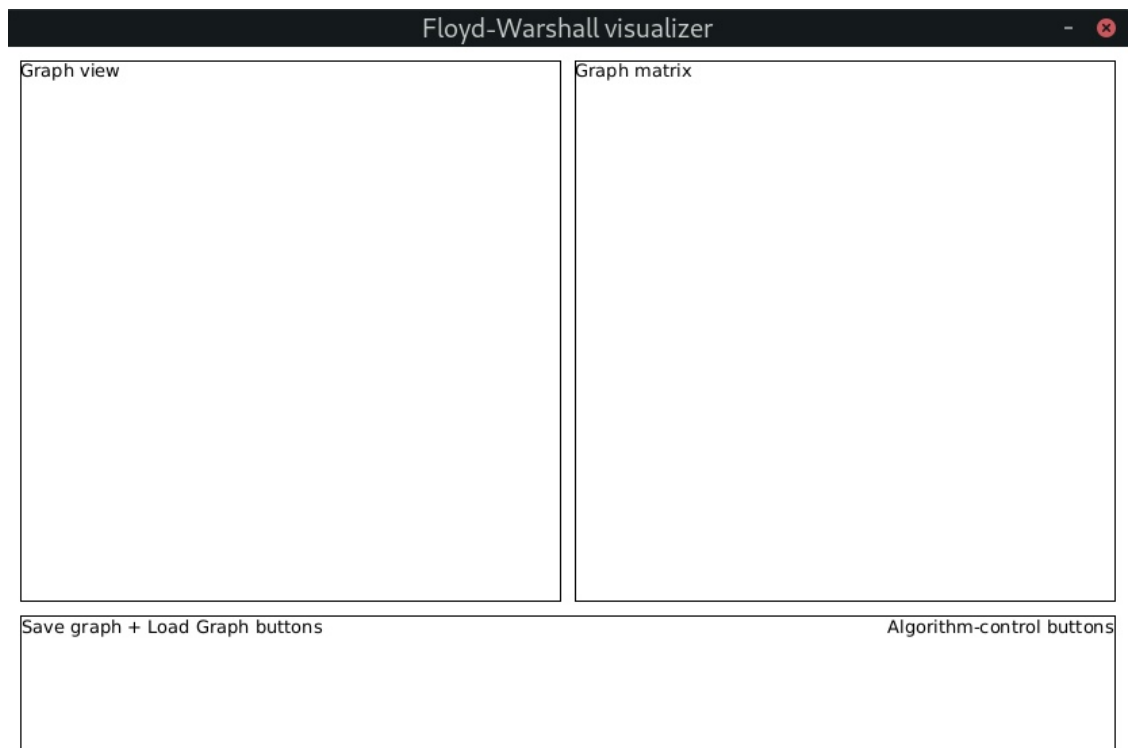


Рисунок 1 - Макет интерфейса программы

На рисунке 2 представлен финальный интерфейс программы.

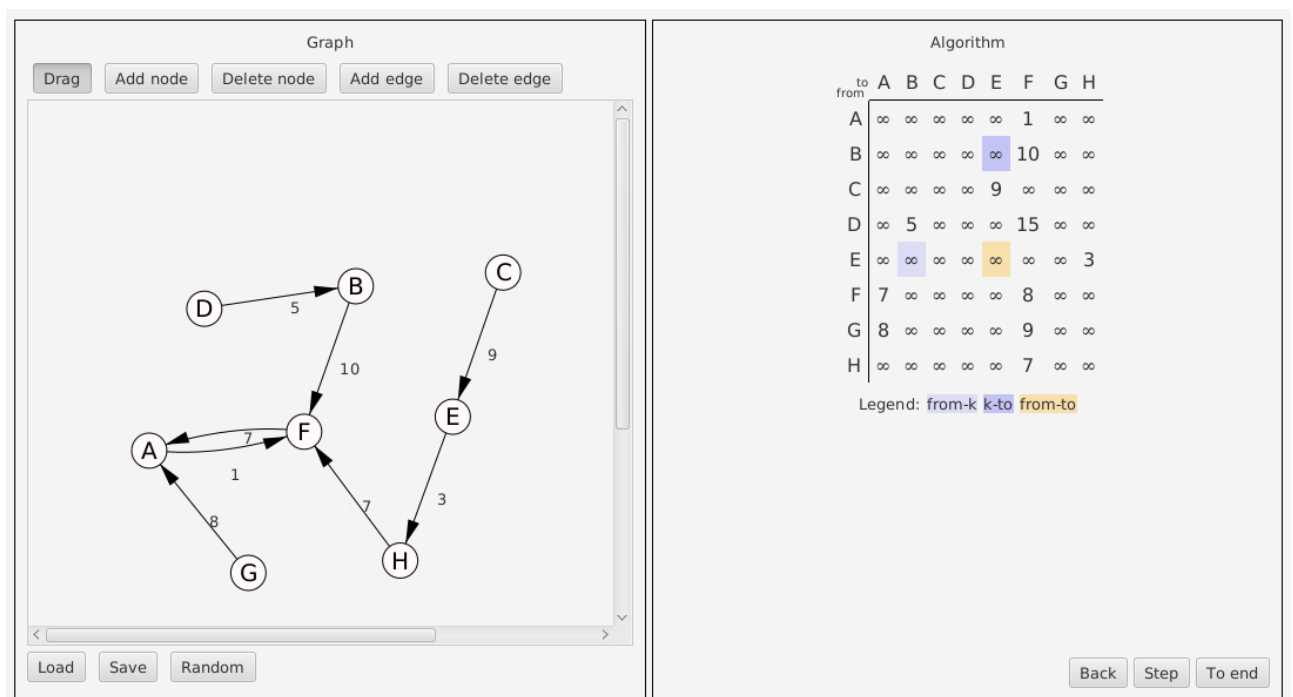


Рисунок 2 - Финальный интерфейс программы

2. ПЛАН РАЗРАБОТКИ И РАСПРЕДЕЛЕНИЕ РОЛЕЙ В БРИГАДЕ

2.1. План разработки

- Составление спецификации
- Распределение ролей
- Разработка прототипа пользовательского интерфейса
- Возможность построения графа в пользовательском интерфейсе
- Просмотр начального и конечного состояний алгоритма
- Чтение/сохранение графа в файл
- Возможность пошагового исполнения алгоритма
- Возможность возврата алгоритма к предыдущим состояниям
- Тестирование корректности работы программы
- Реализация дополнительного функционала

На рисунке 3 представлена диаграмма последовательности.

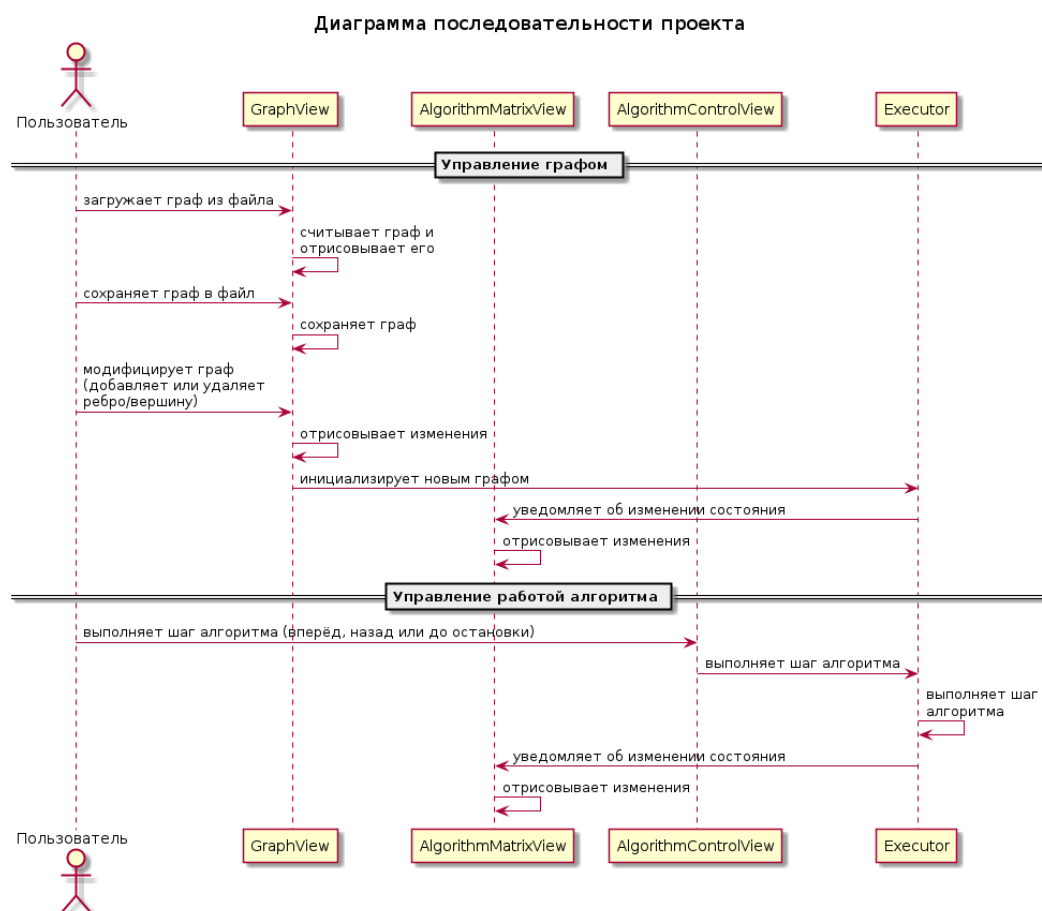


Рисунок 3 - Диаграмма последовательности

На рисунке 4 представлена диаграмма прецедентов.

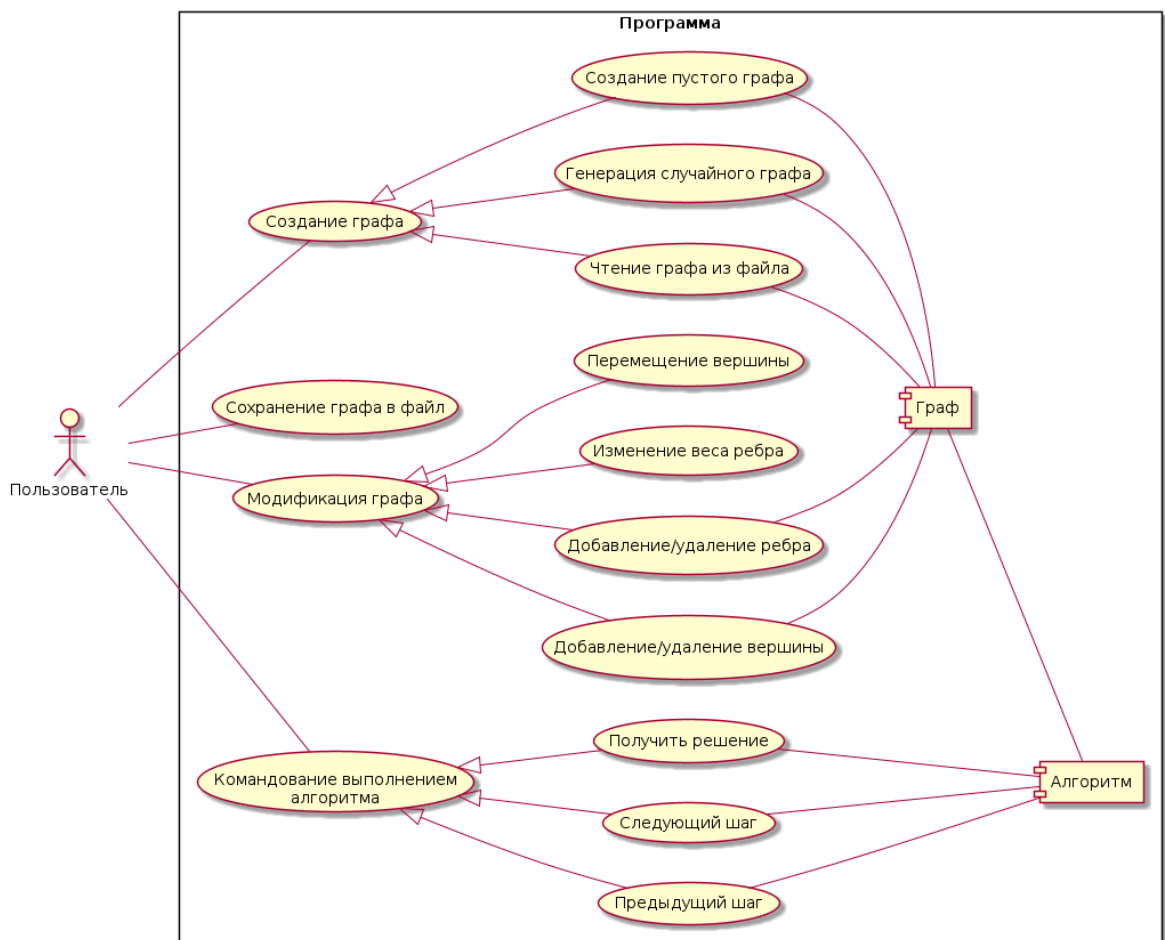


Рисунок 4 - Диаграмма прецедентов

2.2. Распределение ролей в бригаде

Рыжиков Александр - пользовательский интерфейс (ввод и отрисовка графа).

Мешков Максим - пользовательский интерфейс (визуализация работы алгоритма).

Колногоров Даниил - реализация алгоритма, тестирование.

3. ОСОБЕННОСТИ РЕАЛИЗАЦИИ

3.1. Реализуемые классы

GraphView реализует компоненту интерфейса, отвечающую за взаимодействие пользователя с графом: создание произвольного графа, загрузка графа из файла, генерирование случайного графа. При очередном изменении графа пользователем алгоритм заново инициализируется новым графом.

AlgorithmMatrixView реализует компоненту интерфейса, отвечающую за отображение используемой алгоритмом матрицы смежности. При выполнении шага алгоритма *AlgorithmMatrixView* получает уведомления об изменении матрицы алгоритма и обновляет свое содержимое.

AlgorithmControlView реализует компоненту интерфейса, отвечающую за управление работой алгоритма. Состоит из нескольких кнопок, при нажатии на которые происходит выполнение/отмена одного или нескольких шагов алгоритма.

ExecutorInterface определяет интерфейс взаимодействия с алгоритмом. Определяет несколько групп методов: методы получения текущего состояния матрицы алгоритма, методы выполнения/отмены произвольного количества шагов, метод установки графа, с которым работает алгоритм, и метод установки наблюдателя, который будет уведомлён об изменении состояния алгоритма.

Executor реализует *ExecutorInterface*.

Gui создает экземпляры классов, представляющих компоненты пользовательского интерфейса, и располагает их в окне программы.

Main является точкой входа в программу. Создает экземпляр *Gui* и устанавливает в него экземпляр *Executor*, а затем передает управление *Gui*.

GravitySimulation реализует симуляцию гравитации для вершин графа.

На рисунке 5 представлена диаграмма классов.

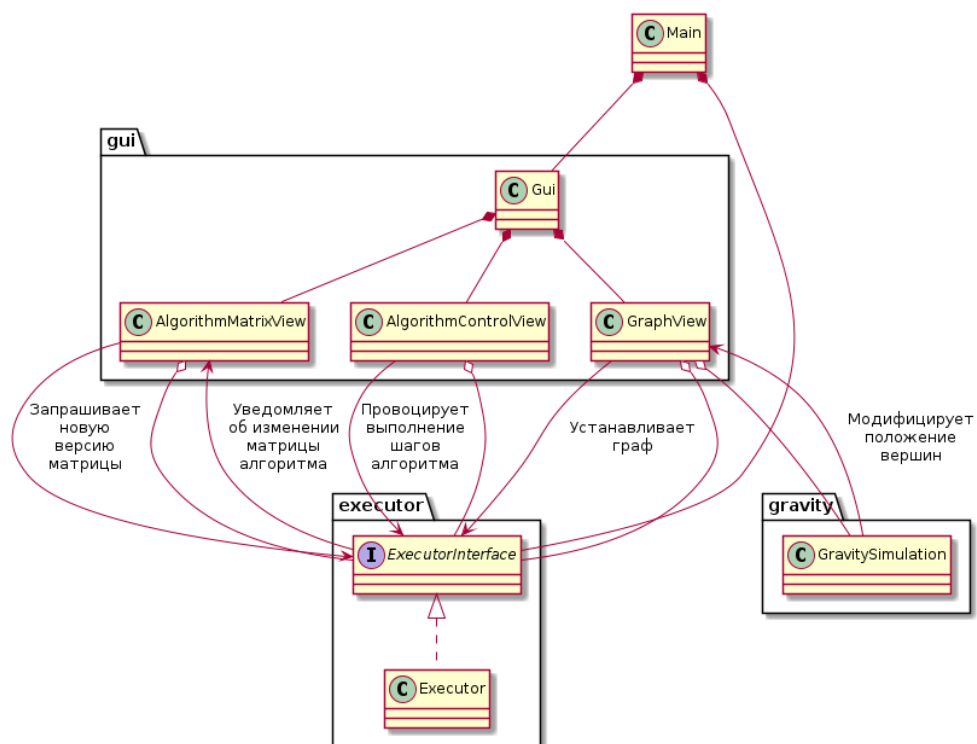


Рисунок 5 - Диаграмма классов

3.2. Основные методы

На рисунке 6 представлена диаграмма состояний программы.

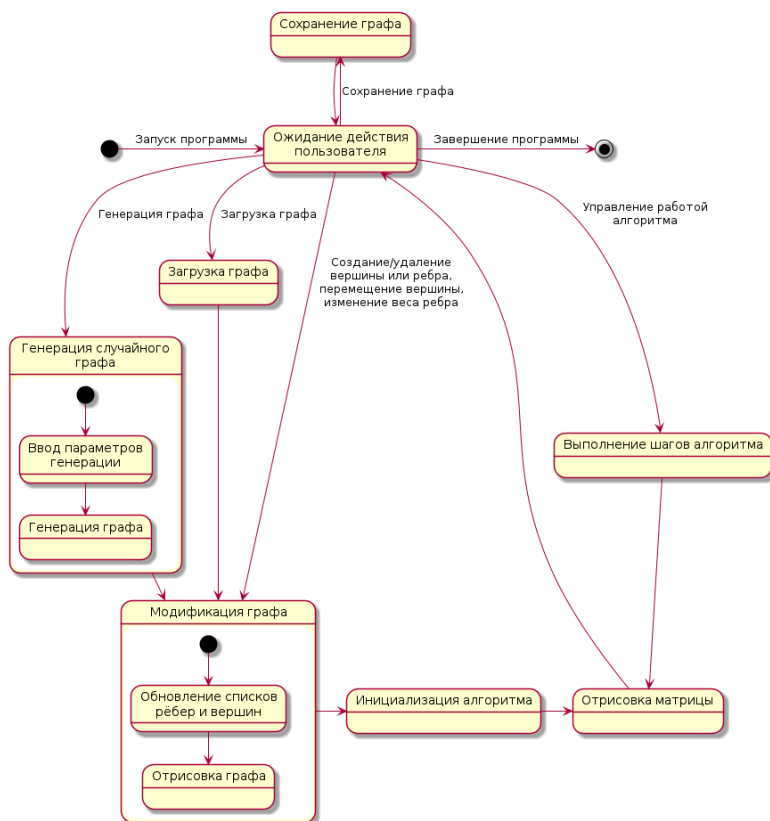


Рисунок 6 - Диаграмма состояний

4. ТЕСТИРОВАНИЕ

4.1. План тестирования программы

В процессе разработки программы планируется проведения тестирования корректности работы программы с использованием библиотеки для модульного тестирования JUnit. Основной объект тестирования - реализуемый алгоритм Флойда-Уоршелла. В таблице 1 представлен план тестирования класса *Executor*, реализующего алгоритм.

Таблица 1 - План тестирования

Тестируемый функционал	Исходные данные	Ожидаемый результат
Конструктор класса	Пустой граф	После инициализации экземпляра класса метод <i>isFinished</i> должен возвращать значение <i>True</i> , вызов метода <i>step</i> не должен влиять на состояние алгоритма.
Методы <i>setGraph</i> и <i>getPathLength</i>	Непустой граф (список рёбер и количество вершин)	Матрица смежности, совпадающая с матрицей смежности передаваемого графа
Методы <i>step</i> , <i>toEnd</i> и <i>getPathLength</i>	Непустой граф (список рёбер и количество вершин)	Матрица, в ячейке (i, j) которой находится значение минимального расстояния от вершины i до вершины j , а при отсутствии пути — значение <i>null</i> .
Методы <i>step</i> , <i>nextCell</i> и <i>prevCell</i>	Непустой граф (список рёбер и количество вершин)	Алгоритм перемещается по таблице смежности в правильном порядке: слева-направо и сверху-вниз. После посещения нижней правой ячейки матрицы алгоритм должен возвращаться к верхней левой ячейке.
Возможность возврата алгоритма к более ранним состояниям	Непустой граф (список рёбер и количество вершин)	После вызова метода <i>step</i> с положительным аргументом n , не превышающим количество оставшихся шагов до завершения алгоритма, и последующим вызовом метода <i>step</i> с аргументом $-n$ состояние матрицы смежности не должно измениться.
Реализуемый классом паттерн проектирования «Наблюдатель»	Несколько экземпляров класса, реализующего <i>ExecutorInterface</i>	После вызов метода <i>step</i> все подписчики должны быть уведомлены об изменении состояния алгоритма.

Дополнительно проверяется корректность сборки проекта из исходных файлов в jar-архив. Программа считается корректно работающей при успешном прохождении ею тестов и корректной сборке проекта.

ЗАКЛЮЧЕНИЕ

В результате выполнения практической работы была разработана программа для визуализации работы алгоритма Флойда-Уоршелла. Программа полностью соответствует поставленным в начале разработки требованиям: у пользователя есть возможность задать входные данные для алгоритма с помощью графического интерфейса, а также возможность пошагового исполнения алгоритма с просмотром его промежуточных состояний. Разработка осуществлялась при активном участии всех членов команды с предварительным распределением ролей в проекте.

Во время разработки были получены практические знания о языке программирования Java, фреймворке для разработки пользовательских интерфейсов JavaFX и фреймворке для юнит-тестирования JUnit. Также были получены навыки проектирования программ с использованием представления внутренних процессов программы в виде диаграмм и навыки командной разработки.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Java. Библиотека профессионала. Том 1. Основы. Хорстманн Кей С. 864 с.
2. Прохоренок Н. А. JavaFX. СПб.: БХВ-Петербург, 2020. 768 с.
3. Gradle in Action. Benjamin Muschko. 480 с.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД АЛГОРИТМА

```
package floydwarshall.executor;

import java.util.ArrayList;
import java.util.Stack;

public class Executor implements ExecutorInterface {
    private int k;
    private int from;
    private int to;
    private int verticesAmount;
    private Integer[][] matrix;

    private Stack<Integer> history;

    private ArrayList<ExecutorObserver> observers;

    public Executor() {
        matrix = new Integer[0][0];
        verticesAmount = 0;
        k = 0;
        observers = new ArrayList<ExecutorObserver>();
    }

    public void setGraph(int verticesAmount, ArrayList<Edge> edges) {
        k = 0;
        from = 0;
        to = 0;
        this.verticesAmount = verticesAmount;
        history = new Stack<Integer>();

        matrix = new Integer[verticesAmount][verticesAmount];

        // fill matrix with infinities (represented by null)
        for (int i = 0; i < verticesAmount; i++) {
            for (int j = 0; j < verticesAmount; j++) {
                matrix[i][j] = null;
            }
        }
    }
}
```



```

    }

    // fill edge weights
    for (Edge edge : edges) {
        if (edge.from != edge.to) {
            matrix[edge.from][edge.to] = edge.weight;
        }
    }

    notifyObservers();
}

// NOTE: returns null if the path has not yet been found
public Integer getPathLength(PathEnds pathEnds) {
    Integer length = matrix[pathEnds.from][pathEnds.to];
    return length;
}

public int getVerticesAmount() {
    return verticesAmount;
}

public void step(int amount) {
    if (amount > 0) {
        stepForward(amount);
    } else {
        stepBackward(amount);
    }

    notifyObservers();
};

public int getK() {
    return k;
}

public int getFrom() {
    return from;
}

public int getTo() {

```

```

        return to;
    }

    public void toEnd() {
        while (!isFinished()) {
            step(1000);
        }
    };

    public boolean isFinished() {
        return k == verticesAmount;
    };

    public void addObserver(ExecutorObserver observer) {
        observers.add(observer);
    };

    private void stepForward(int amount) {
        for (; amount > 0 && !isFinished(); amount--) {
            history.push(matrix[from][to]);

            // update value
            if (matrix[from][k] != null && matrix[k][to] != null) {
                if (matrix[from][to] != null) {
                    matrix[from][to] = Math.min(matrix[from][to],
                                                    matrix[from][k] + matrix[k]
[to]);
                } else {
                    matrix[from][to] = matrix[from][k] + matrix[k][to];
                }
            }
            nextCell();
        }
    }

    private void stepBackward(int amount) {
        if (history == null) {
            return;
        }

        amount = -amount;

```

```

        for (; (amount > 0) && !history.isEmpty(); amount--) {
            prevCell();

            // restore previous value
            matrix[from][to] = history.pop();
        }
    }

    private void nextCell() {
        if (++to == verticesAmount) {
            to = 0;
            if (++from == verticesAmount) {
                from = 0;
                k++;
            }
        }
    }

    private void prevCell() {
        if (--to == -1) {
            to = verticesAmount - 1;
            if (--from == -1) {
                from = verticesAmount - 1;
                k--;
            }
        }
    }

    private void notifyObservers() {
        for (ExecutorObserver observer : observers) {
            observer.stateChanged();
        }
    }
}

```