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

ОТЧЕТ

по учебной практике

ТЕМА: «АЛГОРИТМ ФЛОЙДА-УОРШЕЛЛА»

Студентка гр. 8382 _____ Кузина А.М.

Студентка гр. 8382 _____ Кулачкова М.К.

Студентка гр. 8382 _____ Рочева А.К.

Руководитель _____ Ефремов М.А.

Санкт-Петербург
2020

ЗАДАНИЕ

НА УЧЕБНУЮ ПРАКТИКУ

Студентка Кузина А.М. группы 8382

Студентка Кулачкова М.К. группы 8382

Студентка Рочева А.К. группы 8382

Тема практики: алгоритм 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: 01.07.2020

Дата защиты отчета: 12.07.2020

Студентка	_____	Кузина А.М.
Студентка	_____	Кулачкова М.К.
Студентка	_____	Рочева А.К.
Руководитель	_____	Ефремов М.А.

АННОТАЦИЯ

Целью учебной практики является разработка приложения для визуализации алгоритма Флойда-Уоршелла. Приложение создается на языке Java и должно обладать графическим интерфейсом. Пользователю должна быть предоставлена возможность отрисовки используемых структур данных (графа и соответствующей матрицы смежности), а также пошагового выполнения алгоритма с пояснениями. Приложение должно быть понятным и удобным для использования.

Задание выполняется командой из трех человек, за которыми закреплены определенные роли. Выполнение работы и составление отчета осуществляются поэтапно.

SUMMARY

The purpose of training practice is to create an application which would visualize the Floyd-Warshall algorithm. The application should be written in Java programming language and must implement a graphical user interface. The user must be provided with possibilities to view data structures in use (the graph and the respective adjacency matrix) and the step-by-step execution of the algorithm with commentaries. The application must be transparent and handy.

The task is fulfilled by a team of three members, each of them assigned with certain obligations. Implementation of the task and report composition should be gradual.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Целью учебной практики является создание приложения, визуализирующего работу алгоритма Флойда-Уоршелла, предназначенного для нахождения кратчайших расстояний между всеми вершинами взвешенного ориентированного графа. Приложение должно быть написано на языке Java и снабжено понятным и удобным в использовании графическим интерфейсом. Пользователю должна быть предоставлена возможность ввести исходные данные в самой программе с клавиатуры или загрузить их из файла. Результат работы алгоритма также должен выводиться на экран и по требованию сохраняться в файл. Должна быть предоставлена возможность как моментального отображения результата, так и визуализации пошагового выполнения алгоритма.

Задание выполняется командой из трех человек, за каждым из которых закреплены определенные обязанности – реализация графического интерфейса, логики алгоритма, проведение тестирование и сборка проекта. Готовая программа должна корректно собираться из исходников в один исполняемый jar-архив. В ходе сборки должны выполняться модульные тесты и завершаться успехом. Также на момент завершения практики должен быть составлен подробный отчет, содержащий моделирование программы, описание алгоритмов и структур данных, план тестирования, исходный код и др.

1. ТРЕБОВАНИЯ К ПРОГРАММЕ

1.1. Требования к вводу исходных данных

Исходными данными для реализуемого приложения является граф, в котором будет осуществляться поиск путей. Граф задается списком ребер в формате $v_i v_j w_{ij}$, где v_i, v_j – смежные вершины, w_{ij} – вес (длина) ребра между ними. Необходимо предоставить пользователю возможность ввода исходных данных как с клавиатуры в самой программе, так и из текстового файла.

1.2. Требования к выводу результата

Результат выполнения алгоритма должен выводиться на экран в виде таблицы, а также сохраняться в текстовый файл по требованию пользователя.

1.3. Требования к визуализации

Необходимо реализовать удобный и понятный пользователю графический интерфейс. Должна быть предоставлена возможность отрисовки заданного графа, выполнение алгоритма по требованию пользователя необходимо осуществлять моментально с выводом результата или пошагово. При пошаговом выполнении алгоритма каждый этап должен быть снабжен пояснениями.

На рисунке 1 изображена диаграмма прецедентов проекта, описывающая функционал программы.

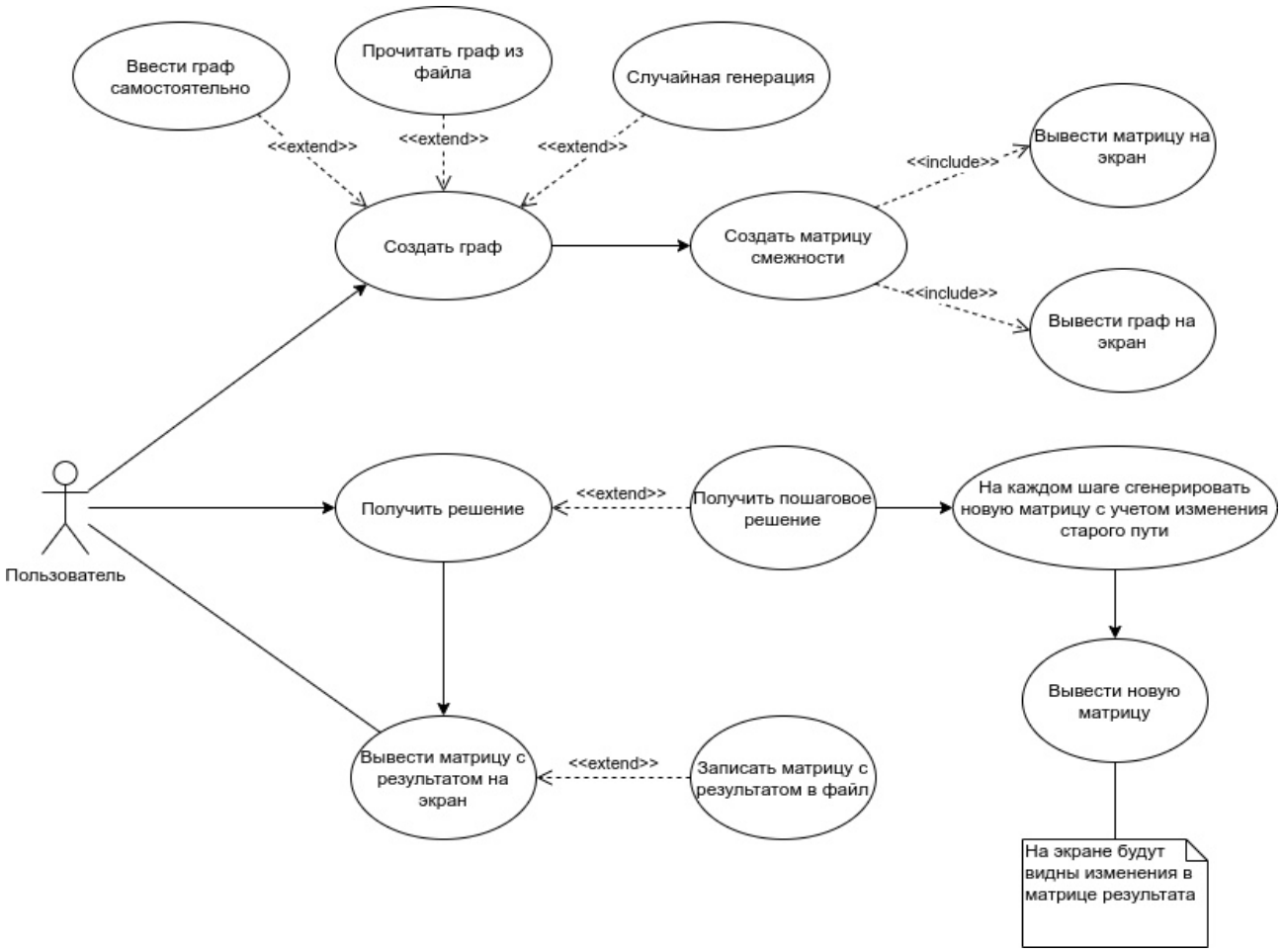

Рисунок 1 - Диаграмма прецедентов

2. ПЛАН РАЗРАБОТКИ И РАСПРЕДЕЛЕНИЕ РОЛЕЙ В БРИГАДЕ

2.1. План разработки

К 02.07.2020 должны быть распределены роли между членами бригады, составлена диаграмма прецедентов программы, а также создана директория с исходным кодом и скриптом сборки.

К 04.07.2020 должны быть размещены все элементы интерфейса, составлены UML-диаграмма классов программы с пояснениями, а также UML-диаграмма состояний программы.

К 06.07.2020 необходимо сделать случайную генерацию изначальных графов с проверкой корректности вводимых данных, решение алгоритма при нажатии на кнопку графического интерфейса с отображением конечного результата работы алгоритма, а также добавить в отчет описание алгоритма и план тестирования.

К 08.07.2020 должна быть добавлена возможность визуализации пошагового выполнения алгоритма, должны быть сделаны тесты для созданных структур данных и функций алгоритма согласно плану тестирования, в отчет добавлено описание алгоритма пошагового отображения работы алгоритма.

К 10.07.2020 проект должен быть полностью готов, программа должна корректно собираться, в ходе сборки должны выполняться и успешно завершаться модульные тесты.

2.2. Распределение ролей в бригаде

Кузина А.М. отвечает за разработку графического интерфейса.

Кулачкова М.К. отвечает за реализацию логики алгоритма.

Рочева А.К. отвечает за тестирование и сборку приложения.

3. ОСОБЕННОСТИ РЕАЛИЗАЦИИ

3.1. Структуры данных

Алгоритм Флойда-Уоршелла, реализуемый в программе, предназначен для обработки графа. Хранение графа осуществляется при помощи класса **Graph**. Граф включает в себя массив вершин, которые представляют собой объекты класса **Vertex**, матрицу смежности – двойной массив связок между вершинами, которые хранятся в виде объектов класса **Connection**, а также три целочисленные переменные для хранения состояния графа. Класс **Vertex** состоит из поля с именем вершины, которое не может быть изменено в ходе работы приложения, и метода, возвращающего имя вершины. Класс **Connection** хранит вес ребра, соединяющего вершины, или -1, если такого ребра нет, длину кратчайшего пути между вершинами и строку, содержащую сам путь. Класс также содержит методы, возвращающие значения приватных полей, метод, обновляющий кратчайший путь и его длину, и метод, изменяющий вес ребра.

Класс **Graph** содержит методы, реализующие работу алгоритма, методы, изменяющие текущий граф, а также методы, возвращающие информацию о графе.

Основным классом программы является класс **App**. В нем создается графический интерфейс. В графическом интерфейсе хранится экземпляр контроллера **GraphController**, который осуществляет взаимодействие между интерфейсом и графом.

На рисунке 2 изображена UML-диаграмма классов программы. Она будет дополнена в ходе дальнейшей работы над проектом.

Исходный код класса Vertex представлен в приложении Б, исходный код класса Connection представлен в приложении В.

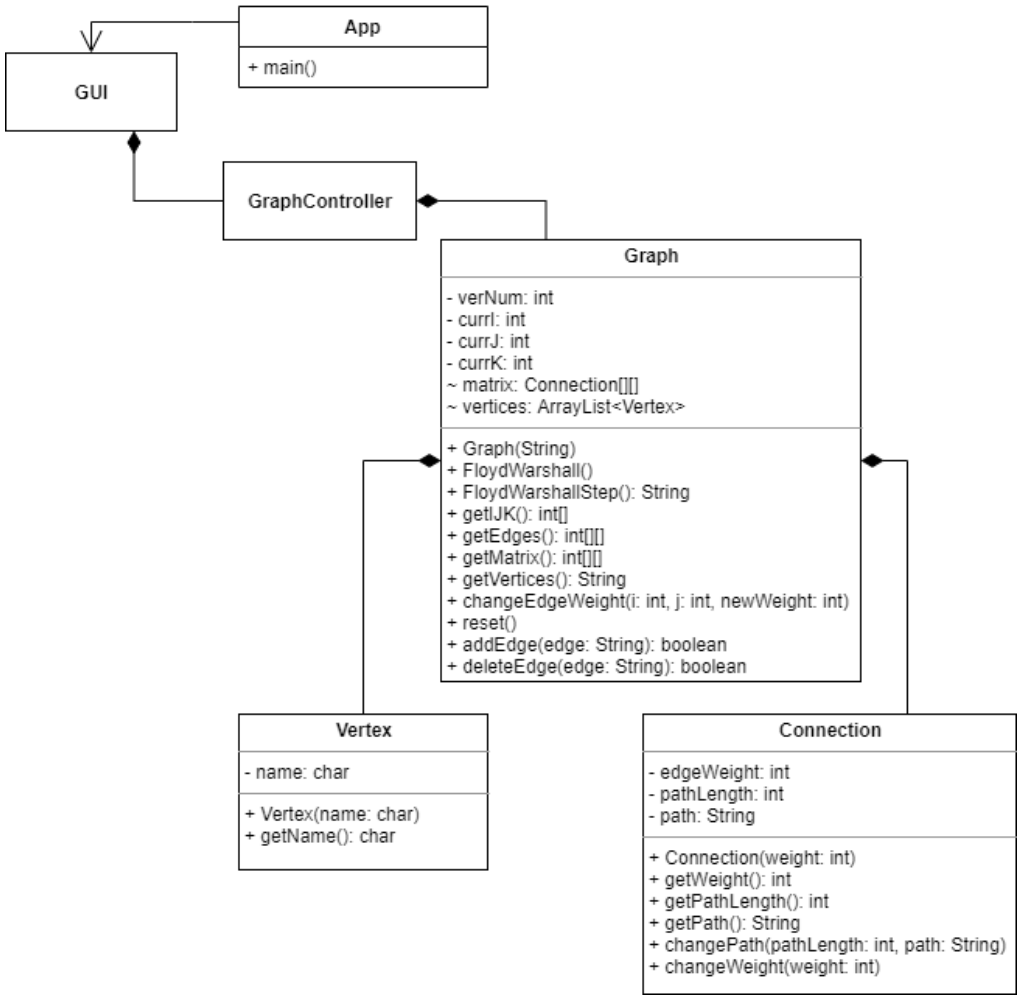

Рисунок 2 - Диаграмма классов модели

3.2. Описание алгоритма

Алгоритм Флойда-Уоршелла реализован в методе *FloydWarshall* класса **Graph**. Исходный код алгоритма представлен в приложении А. На каждом шаге метод изменяет матрицу, которая содержит длины кратчайших путей между всеми вершинами.

Алгоритм содержит три цикла, в которых обходятся все вершины графа. В двух внутренних циклах рассматриваются ячейки матрицы кратчайших путей, и текущий кратчайший путь из одной вершины в другую сравнивается с путем, проходящим через вершину, рассматриваемую во внешнем цикле, т.е. суммой путей из начальной вершины во внешнюю и из внешней в конечную. Если путь через внешнюю вершину короче текущего пути, в матрице кратчайших путей изменяется кратчайший путь между вершинами.

3.3. Алгоритм пошагового отображения

Пошаговое выполнение алгоритма представляет собой, по сути, одну итерацию основного алгоритма. Для его реализации было решено хранить переменные, использующиеся для обхода графа в циклах алгоритма, в качестве полей класса. Переменные предварительно проверяются на невыход за границы графа.

Далее выполняется шаг алгоритма, т. е. текущий кратчайший путь из одной вершины в другую сравнивается с путем, проходящим через внешнюю вершину. Если путь через внешнюю вершину короче текущего пути, в матрице кратчайших путей изменяется кратчайший путь между вершинами.

Значения переменных увеличиваются так, как это осуществлялось бы при выполнении всего алгоритма сразу.

3.3. Основные методы

На рисунке 3 представлена диаграмма состояний программы (рисунок находится в diagrams/states).

3.5. Интерфейс

Взаимодействие пользователя с программой осуществляется при помощи графического интерфейса (рисунок 4). При запуске программы открывается главное окно программы, состоящее из четырех панелей: панель для текстового вывода, которая содержит текстовое представление графа или информацию о его состоянии, панель для отображения исходной матрицы смежности графа, панель для отображения результирующей матрицы, содержащей текущие кратчайшие пути между вершинами, и панель управляющих кнопок.

Нажатием на кнопки пользователь может считать граф из файла, ввести граф с клавиатуры – при этом открывается новое окно для ввода, сгенерировать случайный граф с заданным числом вершин, выполнить алгоритм полностью, выполнить один шаг алгоритма, отрисовать исходный граф и граф в текущем

состоянии, сохранить текущее состояние графа в файл, добавить и удалить ребро в текущем графе, сбросить результат работы алгоритма.

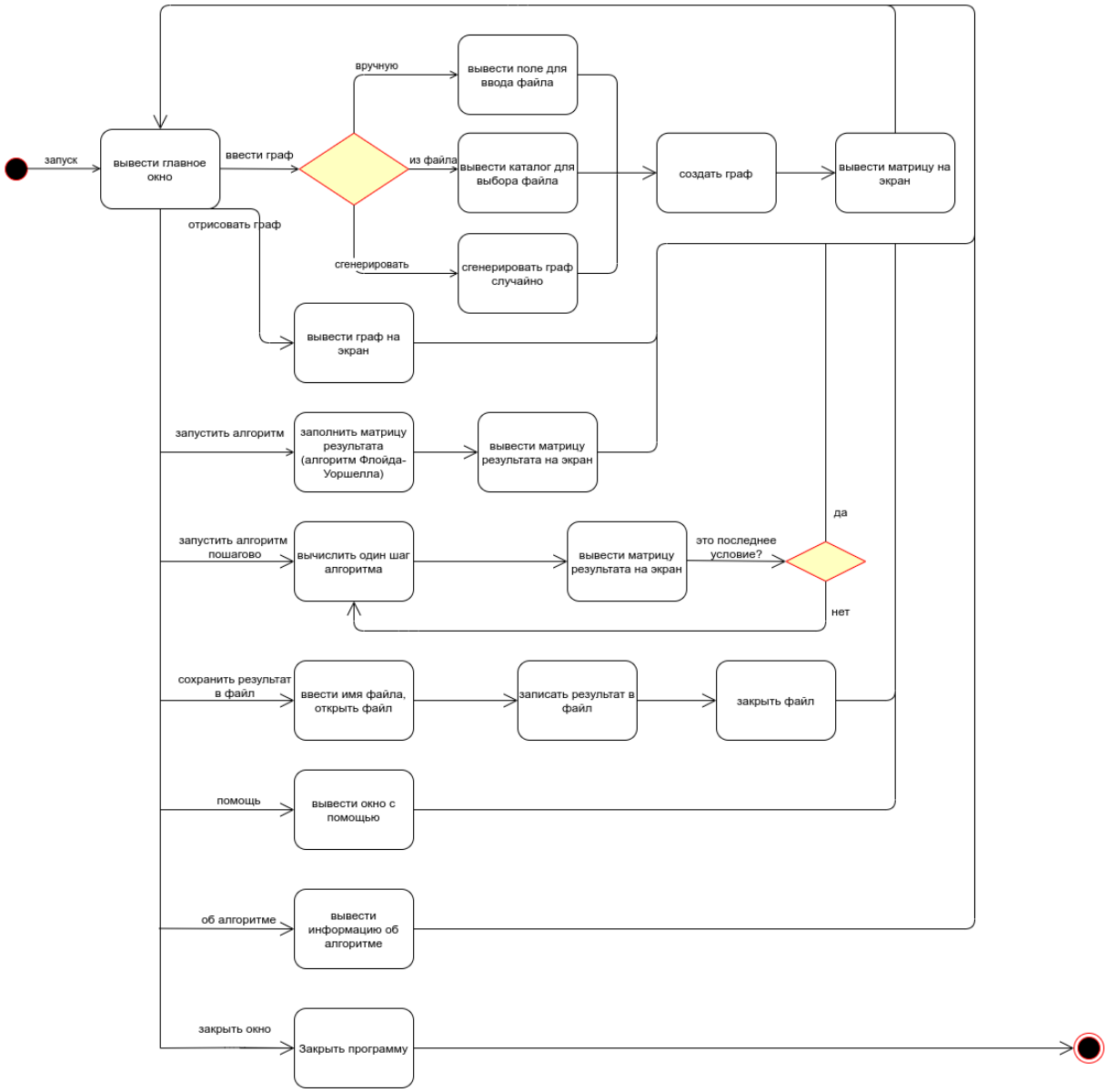

Рисунок 3 - Диаграмма состояний

При создании нового графа отрисовывается его матрица смежности. В матрице смежности пользователь может изменить веса существующих ребер. При изменении графа матрица смежности также перерисовывается.

При запуске алгоритма или шага алгоритма отрисовывается матрица результата, содержащая текущие кратчайшие пути между вершинами графа. Поля матрицы результата изменить нельзя.

При нажатии на кнопку «Отрисовать граф» открывается новое окно, содержащее графическое представление графа (рисунок 5). Вершины в этом окне можно перетаскивать. При запуске пошагового выполнения алгоритма вершины, рассматриваемые на данном шаге, будут окрашены разными цветами: вершина, из которой происходит поиск пути, будет оранжевой, вершина, в которую строится путь, – красной, вершина, через которую строится новый путь, – зеленой. Неактивные вершины будут желтыми. При отрисовке графа после завершения работы алгоритма кратчайшие пути из одной вершины в другие вершины будут изображены при нажатии на вершину.

Для реализации интерфейса использовались библиотеки swing и awt. С их помощью создаются окна, показываемые пользователю, происходит отслеживание нажатий кнопок, создание и размещение всех элементов интерфейса: текстовых полей, кнопок, таблиц и т.д. С их же помощью осуществляется запрет на использование основного окна интерфейса при открытом окне ввода графа или при выводе информационного сообщения пользователю.

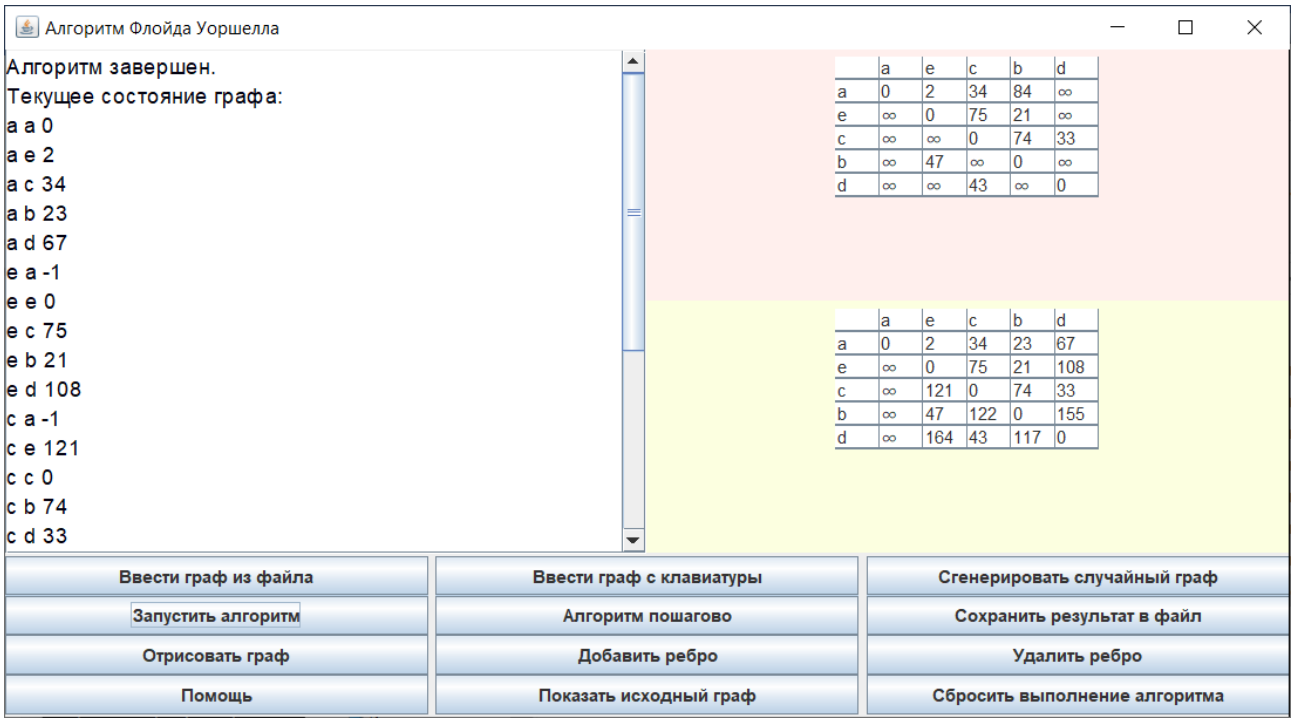

Рисунок 4 - Главное окно программы

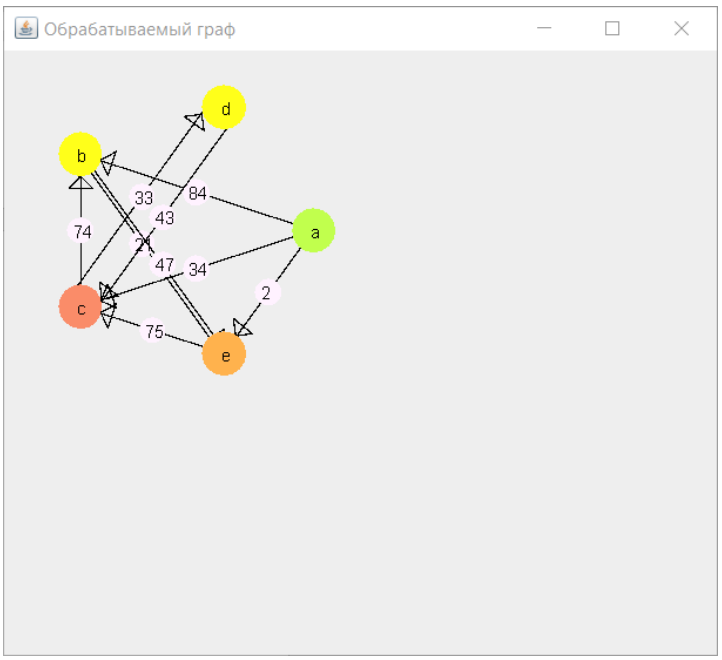

Рисунок 5 - Окно отрисовки графа

3.6. Механизм визуализации

При отрисовке графа вершины изначально располагаются на определенной окружности. Это нужно для того, чтобы не перекрывали друг друга, т.е. ребра могут пересекаться, но не могут накладываться. При этом, после изначальной отрисовки пользователь может передвигать вершины по своему усмотрению.

Вначале вычисляется радиус окружности и шаг между вершинами. Затем итеративно вычисляется координата для следующей вершины – координаты центра окружности + модификатор для координат. Модификатор может представлять собой также отрицательное число, так как является синусом или косинусом угла наклона от центра до данной вершины.

Когда для всех вершин распределены координаты, на визуальное поле наносятся: ориентированные ребра с весами и стрелками, вершины, выделенные цветом, имена вершин.

При обработке перетаскивания вершин используются события мыши: нажатие, удерживание и отпускание. При нажатии происходит проверка: лежит ли по координатам мыши вершина. Если по координатам мыши находится

вершина, то она запоминается и затем происходит движение мыши с удержанием нажатия – вершина передвигается за курсором, постоянно изменяя свои координаты. При прекращении удержания вершины – отпускания кнопки мыши – вершина заканчивает следовать за курсором и перестает отслеживаться.

После завершения алгоритма при открытии окна отрисовки изображаются только вершины. Для отображения кратчайших путей осуществляется обработка щелчка мыши. Если по координатам мыши находится вершина, отрисовываются все ребра, выходящие из этой вершины, а ребра, выходящие из других вершин, делаются невидимыми.

3.7. Полная диаграмма классов проекта

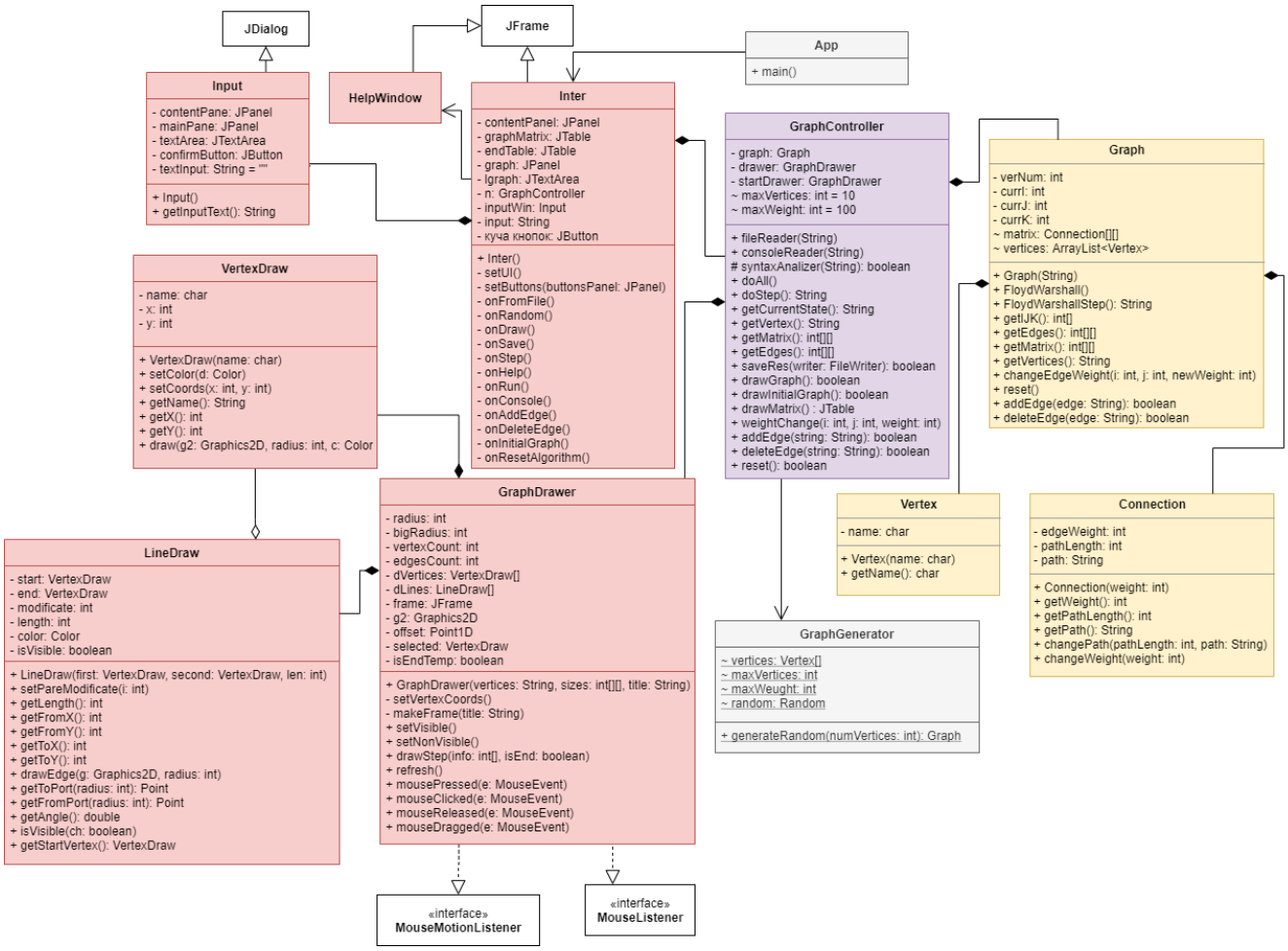

Рисунок 6 - Полная диаграмма классов проекта

На рисунке 6 представлена полная диаграмма классов проекта. На ней желтым цветом выделены классы модели, красным – классы, реализующие интерфейс и визуализацию, фиолетовым – класс **GraphController**, осуществляющий взаимодействие между интерфейсом, визуализацией и моделью. Класс **GraphGenerator** осуществляет генерацию случайного графа. Класс **App** является основным классом программы.

4. ТЕСТИРОВАНИЕ

4.1. План тестирования программы

1. **Вступление**

   Объектом тестирования является программа для визуализации работы алгоритма Флойда-Уоршелла по поиску кратчайших путей в графе.

2. **Функционал, который будет протестирован**

   - Случайная генерация графа
   - Создание графа по введенным данным
   - Создание матрицы смежности по введенным данным
   - Работа алгоритма
   - Работа контроллера графа

3. **Подход к тестированию**

   Уровень тестирования: модульное

   Специальные средства тестирования: тестирование будет проходить с помощью фреймворка автоматического тестирования JUnit.

4. **Критерии успешности тестирования**

   Программа считается законченной, если все разработанные тесты выполняются без ошибок

5. **Критерии прекращения тестирования**

   Программа возвращается на доработку, если хотя бы один из тестов обнаружил ошибку. После исправления ошибки программа снова передается на тестирование.

4.2. Тестовые случаи

1. Случайная генерация графа

На вход подается количество вершин в будущем графе. Тестируется метод generateRandom(int numVertices) класса GraphGenerator. На выходе ожидается объект класса Graph с количеством вершин, равным числу вершин на входе, и с случайно сгенерированным количеством ребер (каждая вершина должна быть

инцидентна минимум одному ребру и максимум (numVertices-1) числу ребер, ребро из каждой вершины должно вести в отличную от нее вершину).

Что подается на вход	Тестируемый метод	Что ожидается на выходе
Число, меньшее двух	public Graph generateRandom(int numVertices)	null
Число, большее значения в константе maxVertices	public Graph generateRandom(int numVertices)	null
Двойка	public Graph generateRandom(int numVertices)	Граф с двумя вершинами и одним ребром
Другое число	public Graph generateRandom(int numVertices)	Граф с полученным на вход количеством вершин и со случайным набором ребер. Ребра должны удовлетворять ранее описанным условиям

Пример тестовых данных:

Что подается на вход	Тестируемый метод	Что ожидается на выходе
1	public Graph generateRandom(int numVertices)	null
11	public Graph generateRandom(int numVertices)	null
4	public Graph generateRandom(int numVertices)	Граф с четырьмя вершинами

2. Создание графа по введенным данным

Тестируются методы класса Graph (проверяется правильность структуры). На вход подается строка, содержащая описания ребер. Тестируются работа конструктора класса Graph (косвенно), метода getMatrix() и getVertices(). На выходе ожидается двумерный массив типа int, представляющий матрицу смежности построенного графа (при тестировании метода getMatrix()) и строка с именами вершин (при тестировании метода getVertices()). Размер матрицы

смежности должен соответствовать количеству вершин в графе (|M| = V*V), значения элементов на главной диагонали должно равняться нулю, в матрице не должно быть элементов со значениями меньше минуса единицы.

Создается граф, строки передаются на вход конструктору графа. Затем вызываются методы getVertices() или getMatrix().

Что подается на вход	Тестируемый метод	Что ожидается на выходе
Непустая строка, соответствующая всем требованиям (v_1 v_2 $w_{v1, v2}$ \n)	public String getVertices()	Строка, состоящая из имен вершин графа (без пробелов). Имена не повторяются.

Что подается на вход	Тестируемый метод	Что ожидается на выходе
Непустая строка, соответствующая всем требованиям (v_1 v_2 $w_{v1, v2}$ \n)	public int[][] getMatrix()	Двумерный массив типа int, отвечающий всем требованиям

Пример тестовых данных:

Что подается на вход	Тестируемый метод	Что ожидается на выходе
"a b 1\nc d 2\nd a 3"	public String getVertices()	`"abcd"`

Что подается на вход	Тестируемый метод	Что ожидается на выходе
"a b 1\nc d 2\nd a 3"	public int[][] getMatrix()	{0, 1, -1, -1 -1, 0, -1, -1 -1, -1, 0, 2 3, -1, -1, 0}

Так же тестируется метод getName() класса Vertex. На вход конструктору объекта подается имя вершины типа char, на выходе метода ожидается это же имя.

Что подается на вход	Тестируемый метод	Что ожидается на выходе
Символ	public char getName()	Этот же символ

Пример тестовых данных:

Что подается на вход	Тестируемый метод	Что ожидается на выходе

'a'	public char getName()	'a'

3. Создание матрицы смежности по введенным данным

Тестируются методы класса Connection. Для тестирования методов класса Connection создается объект этого же класса с параметрами конструктора равными весу ребра из одной вершины в другую и кратчайшим путем между ними типа String (изначально кратчайший путь из вершины *a* и в вершину *b* равен *ab*). Тестируются методы getWeight(), getPathLength(), getPath() и changePath(int pathLength, String path). На вход методу changePath() подается новый кратчайший путь и новая длина. Этот метод будет тестироваться совместно с остальными методами (сначала вызывается метод changePath, затем по очереди снова проводятся все тесты предыдущих методов).

Что подается на вход	Тестируемый метод	Что ожидается на выходе
Вес ребра, путь	public int getWeight()	Вес ребра
Вес ребра, путь	public int getPathLength()	Длина пути
Вес ребра, путь	public String getPath()	Путь

Пример тестовых данных:

Что подается на вход	Тестируемый метод	Что ожидается на выходе
10, "ab"	public int getWeight()	10
10, "ab"	public int getPathLength()	2
10, "ab"	public String getPath()	"ab"

После вызова метода changePath(int pathLength, String path) (на входе указаны аргументы этого метода)

Что подается на вход	Тестируемый метод	Что ожидается на выходе
Новая длина пути и новый путь	public int getWeight()	Вес ребра (не должен измениться)
Новая длина пути и новый путь	public int getPathLength()	Новая длина пути
Новая длина пути и новый путь	public String getPath()	Новый путь

Пример тестовых данных:

Что подается на вход	Тестируемый метод	Что ожидается на выходе
7, "acb"	public int getWeight()	10
7, "acb"	public int getPathLength()	3
7, "acb"	public String getPath()	"acb"

4. Работа алгоритма

Тестируется метод FloydWarshall() класса Graph. Входными данными является поле класса Connection[][] matrix. Алгоритм работает напрямую с этой матрицей, изменяя значения пути в ее элементах (в объектах класса Connection)

Что подается на вход	Тестируемый метод	Что ожидается на выходе
Матрица со всеми положительными элементами (кроме элементов на главной диагонали)	public void FloydWarshall()	Длина пути всех элементов матрицы (кроме элементов на главной диагонали) должна остаться положительной

Примет тестовых данных:

Что подается на вход	Тестируемый метод	Что ожидается на выходе
{0, 1, -1, -1 -1, 0, -1, -1 -1, -1, 0, 2 3, -1, -1, 0}	public void FloydWarshall()	{0, 1, -1, -1 -1, 0, -1, -1 5, 6, 0, 2 3, 4, -1, 0}

5. Работа контроллера графа

Тестируется метод syntaxAnalizer() класса GraphController. Входными данными является строка, которую следует проверить, выходными — булевое значение true или false. Вид корректной строки: $v_1\ v_2\ w_{v1,\,v2}$ \n … $v_i\ v_j\ w_{i1,\,j2}$ \n. Так же именно здесь проверяется количество вершин (оно не должно быть больше допустимого значения).

Что подается на вход	Тестируемый метод	Что ожидается на выходе

Пустая строка	protected boolean syntaxAnalizer(String input)	false
Некорректная строка	protected boolean syntaxAnalizer(String input)	false
Корректная строка	protected boolean syntaxAnalizer(String input)	true

Пример тестовых данных

Что подается на вход	Тестируемый метод	Что ожидается на выходе
""	protected boolean syntaxAnalizer(String input)	false
"acb vc 0"	protected boolean syntaxAnalizer(String input)	false
"a c 4"	protected boolean syntaxAnalizer(String input)	true

ЗАКЛЮЧЕНИЕ

В результате выполнения работы было создано приложение, визуализирующее работу алгоритма Флойда-Уоршелла, предназначенного для нахождения кратчайших расстояний между всеми вершинами взвешенного ориентированного графа. В программе присутствует возможность ввода графа из файла или вручную, так же добавлена возможность его случайной генерации. Для взаимодействия пользователя с программой разработан набор кнопок. Пользователь может добавлять и удалять ребра, отрисовывать граф в отдельном окне, запускать алгоритм (сразу или пошагово). Результат работы алгоритма можно записать в файл.

Разработка приложения выполнялась в команде, создание графического интерфейса происходило с помощью библиотеки Swing, тестирование — с помощью JUnit. Так же были созданы диаграмма классов, диаграмма состояний и диаграмма прецендентов.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Библиотека Swing // URL: http://java-online.ru/libs-swing.xhtml
2. JUnit URL: https://junit.org/junit4/
3. Apache Maven Project URL: https://maven.apache.org/

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД КЛАССА ГРАФА

```
package project.orange;

import java.util.ArrayList;

public class Graph {
    private int verNum;
    private int currI;
    private int currJ;
    private int currK;
    Connection[][] matrix;
    ArrayList<Vertex> vertices;

    public Graph(String str) {
        currI = currJ = currK = 0;
        vertices = new ArrayList<Vertex>();
        String[] edges = str.split("\n");
        Connection[][] tmp = new Connection[edges.length * 2][];
        for (int i = 0; i < tmp.length; i++) {
            tmp[i] = new Connection[edges.length * 2];
        }
        for (String edge: edges) {
            Vertex ver = new Vertex(edge.charAt(0));
            if (!vertices.contains(ver)) {
                vertices.add(ver);
            }
            int ind1 = vertices.indexOf(ver);
            ver = new Vertex(edge.charAt(2));
            if (!vertices.contains(ver)) {
                vertices.add(ver);
            }
            int ind2 = vertices.indexOf(ver);
            tmp[ind1][ind2] = new Connection(Integer.valueOf(edge.substring(4)),
"" + edge.charAt(0) + edge.charAt(2));
        }
        matrix = new Connection[vertices.size()][];
        for (int n = 0; n < matrix.length; n++) {
            matrix[n] = new Connection[vertices.size()];
            for (int m = 0; m < matrix[n].length; m++) {
                if (m == n) {
                    matrix[n][m]      =      new      Connection(0,      ""      +
vertices.get(m).getName());
                } else if (tmp[n][m] == null) {
                    matrix[n][m] = new Connection(-1, "");
                } else {
                    matrix[n][m] = tmp[n][m];
                }
            }
        }
        verNum = vertices.size();
    }

    public void reset() {
        currI = currJ = currK = 0;
        for (int n = 0; n < matrix.length; n++) {
            for (int m = 0; m < matrix.length; m++) {
                String path = "";
                if (n == m) {
                    path += vertices.get(m).getName();
                } else if (matrix[n][m].getWeight() > 0) {
```

```
                    path = path + vertices.get(n).getName() +
vertices.get(m).getName();
                }
                matrix[n][m].changePath(matrix[n][m].getWeight(), path);
            }
        }
    }

    public void FloydWarshall() {
        for (currK = 0; currK < verNum; currK++) {
            for (currI = 0; currI < verNum; currI++) {
                for (currJ = 0; currJ < verNum; currJ++) {

                    if (matrix[currI][currK].getPathLength() < 0 ||
matrix[currK][currJ].getPathLength() < 0) {
                        continue;
                    }

                    if (matrix[currI][currJ].getPathLength() < 0 ||
matrix[currI][currJ].getPathLength() >
                            (matrix[currI][currK].getPathLength() +
matrix[currK][currJ].getPathLength())) {

matrix[currI][currJ].changePath(matrix[currI][currK].getPathLength() +
matrix[currK][currJ].getPathLength(),
                                matrix[currI][currK].getPath() +
matrix[currK][currJ].getPath().substring(1));
                    }
                }
            }
        }
    }

    public String FloydWarshallStep() {
        String res;
        if (currI >= (verNum - 1) && currJ >= (verNum - 1) && currK >= (verNum -
1)) {
            res = "Алгоритм был завершен.";
            return res;
        }
        if (matrix[currI][currK].getPathLength() < 0 ||
matrix[currK][currJ].getPathLength() < 0) {
            res = "Нельзя построить новый путь из вершины " +
vertices.get(currI).getName() + " в вершину " + vertices.get(currJ).getName() +
"\nчерез вершину " + vertices.get(currK).getName();
        } else if (matrix[currI][currJ].getPathLength() < 0 ||
matrix[currI][currJ].getPathLength() >
                (matrix[currI][currK].getPathLength() +
matrix[currK][currJ].getPathLength())) {
            res = "Путь из вершины " + vertices.get(currI).getName() + " в вершину
"
                    + vertices.get(currJ).getName() + " был изменен.\nСтарый
путь: " + matrix[currI][currJ].getPath() + "; \n";
            matrix[currI][currJ].changePath(matrix[currI][currK].getPathLength()
+ matrix[currK][currJ].getPathLength(),
                    matrix[currI][currK].getPath() +
matrix[currK][currJ].getPath().substring(1));
            res = res + "Новый путь: " + matrix[currI][currJ].getPath() + "; длина
нового пути: "
                    + matrix[currI][currJ].getPathLength();
        } else {
            res = "Путь не был изменен.\nПуть из вершины " +
vertices.get(currI).getName() + " в вершину " + vertices.get(currJ).getName() +
```

```
": " + matrix[currI][currJ].getPath() + "; длина пути: " +
matrix[currI][currJ].getPathLength();
        }

        if (currJ < (verNum - 1)) {
            currJ++;
        } else if (currI < (verNum - 1)) {
            currJ = 0;
            currI++;
        } else if (currK < (verNum - 1)) {
            currJ = 0;
            currI = 0;
            currK++;
        }
        return res;
    }

    public int[] getIJK(){
        int[] ew = new int[]{currI, currJ, currK};
        return ew;
    }

    public String getVertices() {
        if (vertices.size() == 0) {
            return null;
        }
        String str = new String("");
        for (Vertex ver : vertices) {
            str = str + ver.getName();
        }
        return str;
    }

    public int[][] getMatrix() {
        if (matrix.length == 0) {
            return null;
        }
        int[][] paths = new int[matrix.length][];
        for (int n = 0; n < matrix.length; n++) {
            paths[n] = new int[matrix.length];
            for (int m = 0; m < matrix[n].length; m++) {
                if (m == n) {
                    paths[n][m] = 0;
                } else if (matrix[n][m] == null) {
                    paths[n][m] = -1;
                } else {
                    paths[n][m] = matrix[n][m].getPathLength();
                }
            }
        }
        return paths;
    }

    public int[][] getEdges() {
        if (matrix.length == 0) {
            return null;
        }
        int[][] edges = new int[matrix.length][];
        for (int n = 0; n < matrix.length; n++) {
            edges[n] = new int[matrix.length];
            for (int m = 0; m < matrix[n].length; m++) {
                if (m == n) {
                    edges[n][m] = 0;
```

```
                } else if (matrix[n][m] == null) {
                    edges[n][m] = -1;
                } else {
                    edges[n][m] = matrix[n][m].getWeight();
                }
            }
        }
        return edges;
    }

    public void changeEdgeWeight(int i, int j, int newWeight) {
        if (i != j) {
            currI = currJ = currK = 0;
            matrix[i][j].changeWeight(newWeight);
            matrix[i][j].changePath(newWeight, "" + vertices.get(i).getName() +
vertices.get(j).getName());
        }
    }

    public boolean addEdge(String edge) {
        if (edge == null) { return false; }
        Vertex ver = new Vertex(edge.charAt(0));
        if (!vertices.contains(ver)) {
            vertices.add(ver);
        }
        int ind1 = vertices.indexOf(ver);
        ver = new Vertex(edge.charAt(2));
        if (!vertices.contains(ver)) {
            vertices.add(ver);
        }
        int ind2 = vertices.indexOf(ver);

        if (vertices.size() > matrix.length) {
            Connection[][] tmp = new Connection[vertices.size()][];
            for (int n = 0; n < vertices.size(); n++) {
                tmp[n] = new Connection[vertices.size()];
                for (int m = 0; m < vertices.size(); m++) {
                    if (m < matrix.length && n < matrix.length) {
                        tmp[n][m] = matrix[n][m];
                    } else if (m == n) {
                        tmp[n][m]     =     new     Connection(0,     ""     +
vertices.get(m).getName());
                    } else {
                        tmp[n][m] = new Connection(-1, "");
                    }
                }
            }
            matrix = tmp;
            verNum = vertices.size();
        }
        currI = currJ = currK = 0;
        matrix[ind1][ind2] = new Connection(Integer.valueOf(edge.substring(4)),
"" + edge.charAt(0) + edge.charAt(2));
        return true;
    }

    public boolean deleteEdge(String edge) {
        if (edge == null) { return false; }
        Vertex ver = new Vertex(edge.charAt(0));
        if (!vertices.contains(ver)) {
            return false;
        }
        int ind1 = vertices.indexOf(ver);
```

```
            ver = new Vertex(edge.charAt(2));
            if (!vertices.contains(ver)) {
                return false;
            }
            int ind2 = vertices.indexOf(ver);

            currI = currJ = currK = 0;
            matrix[ind1][ind2] = new Connection(-1, "");
            return true;
        }
    }
```

ПРИЛОЖЕНИЕ Б

ИСХОДНЫЙ КОД КЛАССА ВЕРШИНЫ

```
package project.orange;

import java.util.Objects;

public class Vertex {
    private final char name;

    public Vertex(char name) {
        this.name = name;
    }
    public char getName() {
        return name;
    }

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

    @Override
    public int hashCode() {
        return Objects.hash(name);
    }
}
```

ПРИЛОЖЕНИЕ В

ИСХОДНЫЙ КОД КЛАССА CONNECTION

```
package project.orange;

public class Connection {
    private int edgeWeight;
    private int pathLength;
    private String path;

    public Connection(int weight, String path) {
        pathLength = edgeWeight = weight;
        this.path = path;
    }

    public int getWeight() {
        return edgeWeight;
    }

    public int getPathLength() {
        return pathLength;
    }

    public String getPath() {
        return path;
    }

    public void changePath(int pathLength, String path) {
        this.pathLength = pathLength;
        this.path = path;
    }

    public void changeWeight(int weight) {
        edgeWeight = weight;
    }
}
```