

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

ОТЧЕТ
по учебной практике
Тема: «Минимальное остовное дерево»

Студент гр. 1384	_____	Бобков В.Д.
Студентка гр. 1384	_____	Усачева Д.В.
Студентка гр. 1384	_____	Пчелинцева К.Р.
Руководитель	_____	Фирсов М.А.

Санкт-Петербург
2023

ЗАДАНИЕ НА УЧЕБНУЮ ПРАКТИКУ

Студент Бобков В.Д. группы 1384

Студентка Усачева Д.В. группы 1384

Студентка Пчелинцева К.Р группы 1384

Тема практики: Минимальное остовное дерево

Задание на практику:

Командная итеративная разработка визуализатора алгоритма на Kotlin с графическим интерфейсом.

Алгоритм: ближайшего соседа [Prim].

Сроки прохождения практики: 30.06.2023 – 13.07.2023

Дата сдачи отчета: 12.07.2023

Дата защиты отчета: 12.07.2023

Студент	_____	Бобков В.Д.
Студентка	_____	Усачева Д.В.
Студентка	_____	Пчелинцева К.Р.
Руководитель	_____	Фирсов М.А.

АННОТАЦИЯ

Данный проект посвящён визуализации алгоритма Прима, используемого для решения задачи минимального остовного дерева, на ЯП Kotlin. В работе представлено краткое описание алгоритма, его ключевые шаги и основные принципы работы. Основной акцент делается на визуализации процесса построения минимального остовного дерева с использованием графических элементов.

Выполнение работы состоит из таких элементов как: создание спецификации и плана тестирования; написание кода, реализующего алгоритм; визуализация алгоритма; написание отчета. Входные данные могут задаваться тремя способами: граф считывается из файла, с консоли, либо в режиме реального времени создается самим пользователем с использованием программных инструментов.

SUMMARY

This project is a visualization of Prim's algorithm used to solve the minimum spanning tree problem in Kotlin. The paper presents a brief description of the algorithm, its key steps and basic principles of operation. The main focus is on visualizing the process of building a minimum spanning tree using graphic elements.

The execution of work consists of such elements as: creation of a specification and a test plan; writing code that implements the algorithm; algorithm visualization; writing a report. The input data can be specified in three ways: the graph is read from a file, from the console, or it is created in real time by the user using software tools.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ.....	5
1. ТРЕБОВАНИЯ К ПРОГРАММЕ	6
1.1. Исходные Требования к программе	6
1.1.1. Формальная постановка задачи	6
1.1.2. Описание интерфейса	6
1.1.3. Формат входных и выходных данных	8
1.2. Уточнения требований после сдачи 1-ой версии	9
1.3. Уточнения требований после сдачи 2-ой версии	9
2. ПЛАН РАЗРАБОТКИ И РАСПРЕДЕЛЕНИЕ РОЛЕЙ В БРИГАДЕ	10
2.1. План разработки.....	10
2.2. Распределение ролей в бригаде.....	10
3. ОСОБЕННОСТИ РЕАЛИЗАЦИИ.....	12
3.1. Структуры данных.....	12
3.2. Основные методы	14
4. ТЕСТИРОВАНИЕ	20
4.1. Тестирование основных функций продукта	20
4.2. Тестирование граничных условий	22
4.3. Тестирование интерфейса.....	25
4.4. Тестирование структуры данных.....	29
ЗАКЛЮЧЕНИЕ	33
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	34

ВВЕДЕНИЕ

Целью данного проекта является изучение языка Kotlin, GUI, а также разработка программы, решающую задачу нахождения минимального остовного дерева, с графическим интерфейсом. Этот интерфейс позволит пользователя взаимодействовать с программой.

Задачи проекта:

- Реализация алгоритма Прима.
- Визуализация графа: создание и отображение графа, на котором будет выполняться алгоритм Прима.
- Подсветка выбранных ребер: выделение выбранных ребер в остовном дереве разным цветом или толщиной, чтобы легче отследить процесс построения дерева.
- Возможность пользовательского ввода графа: добавление возможности ввода пользователем графа с помощью интерфейса приложения. Это позволит пользователям проверять алгоритм Прима на своих собственных данных.

1. ТРЕБОВАНИЯ К ПРОГРАММЕ

1.1. Исходные Требования к программе

1.1.1. Формальная постановка задачи

Имеется следующий неориентированный взвешенный граф. Назовем остовным деревом подграф, содержащий все вершины исходного графа, который является деревом. И задача состоит в том, чтобы найти такое остовное дерево, сумма рёбер которого минимальна и визуализировать процесс его нахождения. Псевдокод рассматриваемого алгоритма представлен на рисунке 1.

```
 $T \leftarrow \{\}$   
Для каждой вершины  $i \in V$   
   $d[i] \leftarrow \infty$   
   $p[i] \leftarrow nil$   
 $d[1] \leftarrow 0$   
  
 $Q \leftarrow V$   
 $v \leftarrow Extract.Min(Q)$   
  
Пока  $Q$  не пуста  
  Для каждой вершины  $u$  смежной с  $v$   
    Если  $u \in Q$  и  $w(v, u) < d[u]$   
       $d[u] \leftarrow w(v, u)$   
       $p[u] \leftarrow v$   
     $v \leftarrow Extract.Min(Q)$   
   $T \leftarrow T + (p[v], v)$ 
```

Рисунок 1 – Псевдокод алгоритма Прима

1.1.2. Описание интерфейса

Графический интерфейс, реализуемый для решения поставленной задачи, будет представлять собой окно, на котором изображены кнопки для перехода на следующий или предыдущий шаг алгоритма, а также на первый и последний шаги, неориентированный граф, полученный на данном шаге алгоритма, с текстовым пояснением происходящего.

Также подразумевается возможность взаимодействия с графическими элементами (переход на предыдущий или на следующий шаг алгоритма, задание начальных условий – добавление вершин и ребер при помощи мышки и клавиатуры). Эскиз интерфейса представлен на рисунке 2. Описание элементов эскиза: 1 – неориентированный граф, 2 – кнопки для перехода на следующий и предыдущий шаг алгоритма, а также на первый и последний шаги, 3 – текстовое пояснение происходящего на данном шаге, 4 – удаление элемента графа, 5 – добавление вершины, 6 – добавление ребра. Интерфейс должен быть ясным и удобным для пользователя.

Для создания графа в приложении необходимо создать некоторое количество вершин и ребер. Чтобы создать вершину пользователю необходимо нажать на кнопку добавления вершины и выбрать место для её добавления. Чтобы создать ребро пользователю необходимо нажать на кнопку добавления ребра, выбрать щелчком две вершины, между которыми будет нарисовано ребро, и ввести вес ребра». Если же пользователь хочет удалить вершину или ребро, ему необходимо нажать на кнопку удаления, а после мышкой выбрать необходимый элемент.

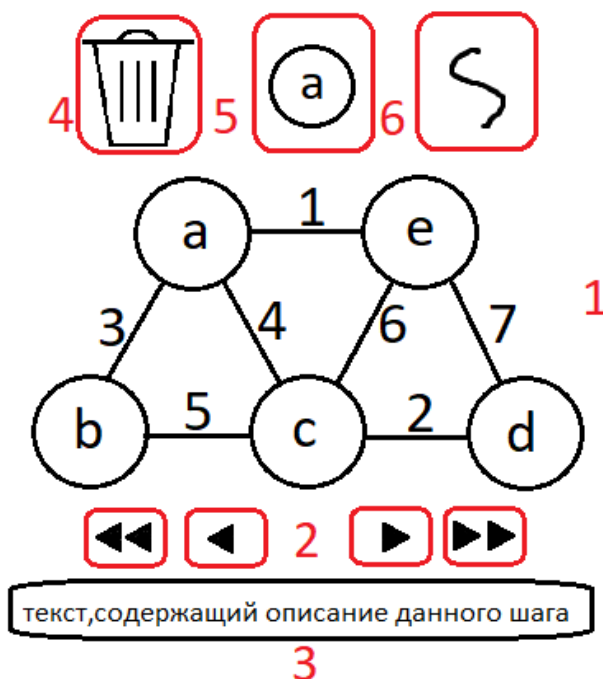


Рисунок 2 – Эскиз интерфейса

1.1.3. Формат входных и выходных данных

Для данной задачи должен быть предусмотрен ввод входных данных разными способами. Перед запуском приложения пользователю будут предложены варианты вида ввода графа: ввод из файла, ввод из консоли, создание графа в приложении. При выборе первых двух вариантов приложение запустится с заданным заранее графом, который пользователь может отредактировать перед началом работы алгоритма. Если же пользователь выберет третий вариант, то ему необходимо будет полностью создать неориентированный граф.

Задаётся неориентированный взвешенный граф $G = (V, E)$ в виде квадратной симметричной матрицы смежности, где $V(|V|=n)$ – это вершины графа; $E(|E|=m)$ – это ребра между вершинами графа.

Каждому ребру m_{ij} можно сопоставить критерий выгодности маршрута – вес ребра (натуральное число, формат числа int), $m_{ij}=-1$, если $i=j$ или ребро между вершинами i и j не существует.

Входные данные задаваемые из файла также представляют собой двумерный массив – матрицу смежности, заданную вышеописанным способом.

Выходные данные должны содержать пошаговую визуализацию работы алгоритма и текстовое описание проделанных действий. На каждом шаге уже выбранные вершины и ребра будут подсвечиваться красным цветом, а множество возможных ребер для следующего выбора – синим.

Текстовые сообщения будут иметь вид: «На шаге «номер шага» была добавлена вершина «имя вершины». Ребра, рассматриваемые на данном шаге, имеют веса: «последовательность весов ребер».» для промежуточных шагов, «Вершина «имя вершины» была выбрана в качестве начальной. Ребра, рассматриваемые на данном шаге, имеют веса «последовательность весов ребер»» для начального и «На шаге «номер шага» была добавлена вершина «имя вершины». Построение минимального остовного дерева окончено.» для конечного.

1.2. Уточнения требований после сдачи 1-ой версии

- [Описать формат матрицы смежности в файле/консоли.]
- До матрицы смежности в первой строке должна быть возможность задать имена вершин.
- Возможность задать стартовую вершину щелчком мыши.
- Корректная обработка ошибки отсутствия файла.
- Увеличить размер шрифта весов рёбер и чуть-чуть уменьшить размер вершин.

1.3. Уточнения требований после сдачи 2-ой версии

- Кнопки редактирования графа должны быть неактивны, когда ими нельзя пользоваться.
- Выводить в консоли предупреждение, если на вход была подана несимметричная матрица смежности.

2. ПЛАН РАЗРАБОТКИ И РАСПРЕДЕЛЕНИЕ РОЛЕЙ В БРИГАДЕ

2.1. План разработки

3 июля (понедельник) – первичное согласование спецификации и плана разработки.

5 июля (среда) – согласование скорректированных спецификации и плана разработки. Первичная сдача прототипа. Прототип должен содержать рабочий код рассматриваемого алгоритма и демонтировать интерфейс без реализации основных функций. Сдача плана тестирования.

7 июля (пятница) – сдача скорректированного прототипа и плана тестирования, первичная сдача 1-ой версии . 1-ая версия демонстрирует частичный функционал графического интерфейса – перемещение между шагами работы алгоритма. Ввод в данной версии алгоритма доступен только двумя способами – из консоли и из файла. Также не предусмотрено редактирование графа в приложении перед началом работы алгоритма.

10 июля (понедельник) – сдача скорректированной 1-ой версии и первичная сдача 2-ой версии. 2-ая версия содержит дополнение функционала – задание начальных условий через графический интерфейс и предварительное редактирование графа перед запуском алгоритма.

12 июля (среда) – сдача скорректированной 2-ой версии, сдача финальной версии с отчётом. Финальная версия содержит рабочий алгоритм с корректной пошаговой визуализацией и все необходимые дополнения функционала.

13 июля (четверг) – сдача финальной версии с отчётом со всеми требуемыми правками.

2.2. Распределение ролей в бригаде

Студент Бобков В.Д. группы 1384:

- Реализация первичного графического интерфейса;
- Реализация возможности задания начальных условий через графический интерфейс.

Студентка Усачева Д.В. группы 1384:

- Реализация перемещения между шагами работы алгоритма;
- Преобразование выходных данных работы алгоритма в удобный формат для последующей визуализации.

Студентка Пчелинцева К.Р группы 1384:

- Реализация алгоритма Прима;
- Тестирование программы и создание плана тестирования.

3. ОСОБЕННОСТИ РЕАЛИЗАЦИИ

3.1. Структуры данных

1. Класс *Graph* — открытый класс, содержащий поля и методы необходимые для работы с графом, представленным в виде матрицы смежности.

Данный класс имеет следующие поля:

- Поле *name_vertex* — изменяемый список имен вершин графа.
- Поле *data* — изменяемый двумерный список с весами.

2. Класс *Edge(Vertex1 : VertexVizualisation, Vertex2: VertexVizualisation, private val weight : Int) : Line()* реализует структуру вершины графа, принимает в качестве параметров конструктора две вершины: *Vertex1*, *Vertex2*, которым будущее ребро будет привязано, так же принимает целочисленное значение *weight*, обозначающее вес данного ребра.

Данный класс имеет следующие поля:

- Поле *weight* — вес ребра;
- Поле *line* — объект *Line* для визуализации самого ребра;
- Поле *line_label* — объект *Label* для визуализации веса ребра;
- Поле *edgegroup* — объект *Group* для того, чтоб совместить *line_label* и *line*;
- Поля *position_1*, *position_2* — объекты *Int*, номера вершин *Vertex1*, *Vertex2*;

3. Класс *VertexVizualisation (scene_size : Double, x : Double, y : Double, name_vertex: String,private var number: Int) : Circle()* является реализацией структуры вершины ребра, принимает для конструктора значение *scene_size* — высоту окна приложения для того, чтоб вершина на окне любого размера не была ни слишком мелкой, ни слишком большой, принимает координаты *x*, *y* для определения расположения вершины в окне приложения, значение *name_vertex*, нужное для определения имени будущей вершины, так же принимает *number* — номер вершины.

Данный класс имеет следующие поля:

- Поле *number* — номер вершины, объект *Int*;
 - Поле *name* — объект *Text*, имя вершины, которое будет отображено на графе;
 - Поле *data* — объект *Group*, нужен для объединения в одну группу поля *name* и *circle*;
 - Поле *circle* — объект *Circle*, нужен для визуализации вершины;
4. Класс *GraphEditor()*, нужный для того, чтоб редактировать граф в окне приложения, взаимодействовать с ним(передвигать вершины).

5. Класс *GraphVizualisation(scene_size: Double, val graph: Graph)* реализует визуализацию графа. Параметры, которые принимает конструктор: размер сцены для корректного размещения вершин и сам граф для последующей его визуализации.

Данный класс имеет следующие поля:

- Поле *vertexes* — массив объектов класса *VertexVizualisation* для визуализации вершин;
- Поле *radius* — радиус окружности для размещения вершин;
- Поле *edges* — массив объектов класса *Edge* для визуализации ребер;
- Поле *full_graph* — объект *Group* для визуализации графа, совмещает *vertexes* и *edges*;
- Поле *step* — номер шага работы алгоритма Прима, где шаг равный -1 — это шаг для редактирования или создания графа;

6. Класс *Vizualisation: Application()* реализует визуализацию самого приложения.

Данный класс имеет следующие поля:

- Поле *graph_visual* — объект класса *GraphVizualisation* для визуализации графа;

- Поле *full_group* – объект *Group* для визуализации приложения, совмещает *graph_visual*, *step_information* и два набора кнопок, объектов класса *HBox*;
- Поле *step_information* – объект класса *Label* для описания текущего состояния работы приложения.;
- Поле *graph_editor* – объект класса *GraphEditor()* для использования методов редактирования графа;

3.2. Основные методы

Методы класса *Graph*:

- Метод *fun default_name(quotient: Int): String* – задаёт имена вершин по умолчанию в количестве *quotient*.
- Метод *fun read_from_file(filename: String)* – для чтения матрицы из файла и записи её в поле класса *data*. В зависимости от выбранной опции имена для вершин задаются по умолчанию при помощи метода *default_name*, либо происходит считывание с консоли.
- Метод *fun read_from_console()* – для чтения матрицы с консоли. В зависимости от выбранной опции имена для вершин задаются по умолчанию при помощи метода *default_name*, либо происходит считывание с консоли. После чего происходит построчное считывание данных и запись их в поле класса *data*.
- Метод *fun reflect_matrix()* – для приведения нижнетреугольной матрицы к матрице смежности.
- Метод *fun check_correct_matrix(): Boolean* – вспомогательная функция для проверки правильной записи нижнетреугольной матрицы.
- Метод *fun check_symmetry()* – для проверки матрицы на симметричность.
- Метод *fun modify()* – для переопределения весов ребер (если в файле было -1, значит ребра нет).

- Метод *override fun toString(): String* - возвращающий строку для вывода информации о графе в консоль.
- Метод *fun set_start_vertex(new_start_vertex: Int)* – для переопределения стартовой вершины.
- Метод *fun PrimAlgorithm()* – реализация алгоритма Прима для графа, возвращает массив пар, соответствующих началу и концу ребра. Внутри метода запускается цикл, который выполняется до тех пор, пока количество добавленных вершин в остовное дерево меньше количества вершин в графе.
- Метод *fun correct_graph(): Boolean* – проверка корректности графа.
- Метод *fun iterated_PrimAlgorithm* – реализации итеративного алгоритма Прима. В данной функции происходит поиск смежной вершины с минимальным весом по отношению к текущей вершине. Происходит выбор ребра, добавляемого в остовное дерево.
- Метод *fun create_new_vertex(name: String)* – создание новой вершины.
- Метод *fun delete_vertex(number: Int)* – удаление вершины.
- Метод *fun get_matrix()* – геттер матрицы смежности для графа.
- Метод *fun get_names()* – геттер имен вершин графа.

Методы класса *Edge*:

- Конструктор *init*, который инициализирует объект класса *Edge*, инициализируется поле *line*: задается толщина, цвет, прикрепляются края *line* к двум вершинам *Vertex1*, *Vertex2*, инициализируется *line_label*, задается сама подпись, ее цвет, подпись и координаты (подпись будет находиться в середине *line*), затем поля *line* и *line_label* объединяются в группу *edgegroup*.
- Метод *group()* — геттер для *edgegroup*;
- Метод *get_weight()* — геттер для *weight*;
- Метод *get_line()* — геттер для *line*;
- Метод *get_label()* — геттер для *line_label*;
- Метод *get_positions()* — геттер для *positions1*, *positions2*;

- Метод *decrease_positions(index : Int)* — метод, нужный для того, чтоб уменьшить значение поля *position_”index”* на единицу;

Методы класса *VertexVizualisation*:

- Конструктор *init()*, который инициализирует объект класса *VertexVizualistaion*: инициализируется поле *circle*: задается цвет контура, толщина контура, цвет заливки, инициализируется поле *name*: задается сам текст, шрифт, размер шрифта и расположение текста(ровно посередине круга), затем поля *name* и *circle* объединяются в группу *data*;

- Метод *get_number()* — геттер для *number*;
- Метод *decrease_number()* — метод для уменьшения *number* на 1;
- Метод *group()* — геттер для *data*;
- Метод *get_circle()* — геттер для *circle*;
- Метод *get_name()* — геттер для *name*;

Методы класса *GraphEditor*:

- Метод *getClickedLine(x: Double, y: Double, graph_visual: GraphVizualisation): Edge?* принимает координаты щелчка *x* *y*, затем проходится по всем ребрам графа и возвращает нужное ребро, если щелчок был произведен не в области какой-либо вершины, возвращается *null*;

- Метод *getClickedCircle(x: Double, y: Double, graph_visual: GraphVizualisation): VertexVizualisation?* принимает координаты щелчка *x* *y*, затем проходится по всем вершинам графа и возвращает нужную вершину, если щелчок был произведен не в области какой-либо вершины, возвращается *null*;

- Метод *able_disable_buttons(button1 : Button, button2 : Button, button3: Button, button4 : Button, action : Boolean)* делает кнопки *button1-4* активными или неактивными, в зависимости от значения *action*;

- Метод *isInsideCircle(x: Double, y: Double, circle: Circle): Boolean* нужен для проверки того, находится ли точка с координатами *x,y* внутри круга *circle*;

- Метод *choose_new_name(graph_visual: GraphVizualisation): String* вызывает диалоговое окно, в котором пользователь выбирает имя будущей вершины, в случае, если пользователь откажется выбирать имя или не выберет ничего, имя вершины выберется по умолчанию, затем имя возвращается из данного метода;
- Метод *choose_weight() : Int* нужен для задания веса нового ребра, создается диалоговое окно, в котором и нужно указать целочисленное значение веса нового ребра, если формат ввода будет неверный, ребро будет иметь вес 0;
- Метод *createLine(startCircle: VertexVizualisation, endCircle: VertexVizualisation, graph_visual: GraphVizualisation, step_information: Label)* нужен для того, чтоб создать новое ребро, используя две вершины, к которым будет крепиться ребро;
- Метод *create_edge(stage: Stage, graph_visual: GraphVizualisation, step_information: Label, button1 : Button, button2 : Button, button3 : Button, button4 : Button)* нужен для того, чтоб создать новое ребро, считывается два щелчка левой кнопки мыши, затем определяются вершины, внутри которых эти щелчки, затем получив эти вершины, вызывается *createLine* и там уже создается ребро. Еще, пока действует этот метод, кнопки, отвечающие за передвижения по шагам алгоритма, становятся неактивны;
- Метод *create_vertex(stage: Stage, graph_visual: GraphVizualisation, step_information: Label, button1 : Button, button2 : Button, button3 : Button, button4 : Button)* нужен для создания новой вершины, определяется место щелчка левой кнопки мыши, затем вызывается метод *choose_name()*, затем создается вершина. Еще, пока действует этот метод, кнопки, отвечающие за передвижения по шагам алгоритма, становятся неактивны;
- Метод *delete_element(stage: Stage, graph_visual: GraphVizualisation, step_information: Label, button1 : Button, button2 : Button, button3 : Button, button4 : Button)* нужен для того, чтоб удалить элемент графа, по которому был произведен клик левой кнопки мыши. Еще, пока действует этот метод, кнопки, отвечающие за передвижения по шагам алгоритма, становятся неактивны;

Стоит отметить, что все методы для редактирования графа активны только тогда, когда граф находится на этапе выбора начальной вершины.

- Метод *action(stage: Stage, graph_visual: GraphVizualisation)* задает некоторые начальные действия для окна: передвижение вершин графа, выбор начальной вершины для алгоритма двойным щелчком;

Методы класса *GraphVizualisation* :

- Конструктор *init* – создает объект класса *GraphVizualisation*.

Добавляются объекты в поля *full_graph*, *edges*, *vertexes*. Задаётся положение вершин в окне.

- Метод *set_get_step* – задает значение для поля *step* и возвращает его;
- Метод *get_edges()* – геттер для *edges*;
- Метод *get_vertexes()* – геттер для *vertexes*;
- Метод *group()* – геттер для *full_graph*;
- Метод *get_step()* – геттер для *step*;
- Метод *next_step()* – увеличивает текущее значение поля *step*, не выходя за максимальное значение, и возвращает его;
- Метод *previous_step()* – уменьшает текущее значение поля *step*, не выходя за минимальное значение, и возвращает его;
- Метод *add_edge(new_edge : Edge)* – добавляет ребро *new_edge* в граф/ заменяет вес ребра. Обновляет данные о весе ребра в поле объекта *graph*, также обновляются данные в полях *edges* и *full_graph*;
- Метод *add_vertex(new_vertex: VertexVizualisation)* – добавляет вершину *new_vertex* в граф и обновляет данные в полях класса;
- Метод *delete_vertex(vertex_to_delete : VertexVizualisation)* – удаляет вершину *vertex_to_delete* и все инцидентные ей ребра из графа и обновляет данные в полях класса;
- Метод *delete_edge(edge : Edge)* – удаляет ребро *edge* из графа и обновляет данные в полях класса;

Методы класса *Vizualisation*:

- Метод *start(stage: Stage)* – запускает приложение. В нем задаются размер окна, кнопки и действия, выполняемые при нажатии на них. Задается положение кнопок в окне. Далее определяется способ считывания графа, после считывания происходит визуализация графа при помощи метода *draw_graph*;
- Метод *step_by_step_algorithm(step: Int, new_edge: Button, new_vertex: Button, delete: Button)* – обновляет вид графа в зависимости от шага алгоритма, на котором находится пользователь. Уже добавленные ребра и вершины выделяются красным цветом, рассматриваемые на шаге ребра выделяются синим. Также блокируются кнопки редактирования графа, если пользователь не находится на начальном шаге. При помощи метода *update_step_information* в окне обновляется о происходящем на текущем шаге;
- Метод *update_step_information(step: Int, considered_vertexes: MutableList<Int>, added_edges: MutableList<MutableList<Pair<Int, Int>>>)* – обновляет информацию в окне о происходящем на шаге *step*;
- Метод *draw_graph(stage: Stage, graph: Graph, buttons1: HBox, buttons2: HBox)* – рисует окно, заново инициализируя *graph_visual*, перемещает на первый шаг алгоритма. В методе задается положение пояснительной информации для пользователя. Инициализируется поле *full_group* – объект Group, совмещает *graph_visual*, *step_information* и два набора кнопок, объектов класса HBox. Также вызывается метод *action* класса *GraphEditor*.

4. ТЕСТИРОВАНИЕ

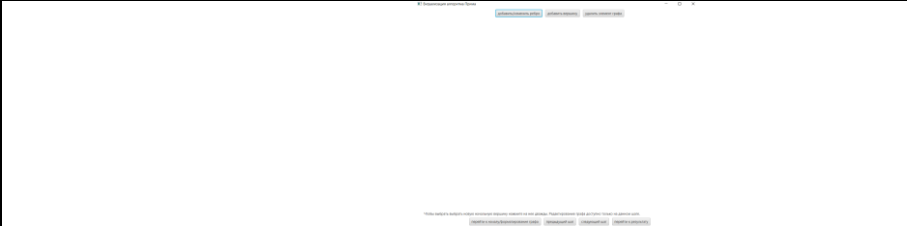
4.1. Тестирование основных функций продукта

Тест на граф с отрицательными весами ребер	
Входные данные:	Граф с несколькими вершинами и ребрами, имеющими отрицательные веса.
Ожидаемый результат:	Алгоритм должен вернуть минимальное остовное дерево, учитывая отрицательные веса ребер.
Полученный результат:	<p>Корректное выполнение работы.</p>
Тест на граф с одинаковыми весами ребер	
Входные данные:	Граф с несколькими вершинами и ребрами, имеющими одинаковые веса.
Ожидаемый результат:	Алгоритм должен вернуть минимальное остовное дерево, учитывая одинаковые веса ребер.
Полученный результат:	<p>Корректное выполнение работы.</p>

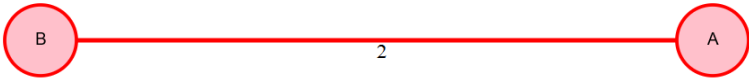
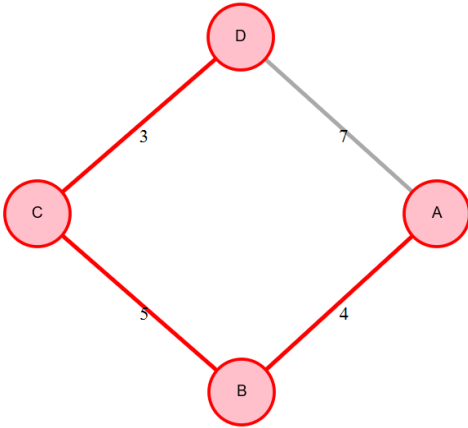
Тест на граф с большим количеством вершин и ребер	
Входные данные:	Граф с большим количеством вершин и ребер.
Ожидаемый результат:	Алгоритм должен вернуть минимальное остовное дерево, учитывая все вершины и связанные ребра в графе.
Полученный результат:	<p>Корректное выполнение работы.</p>
Тест на граф с множеством возможных минимальных остовных деревьев	
Входные данные:	Граф с несколькими вершинами и ребрами, где существует несколько различных минимальных остовных деревьев.
Ожидаемый результат:	Алгоритм должен вернуть одно из возможных минимальных остовных деревьев, не обязательно совпадающее с другими возможными решениями.
Полученный результат:	<p>Корректное выполнение работы.</p>

Тест на граф с большим количеством вершин и небольшим количеством ребер	
Входные данные:	Граф с большим количеством вершин и небольшим количеством ребер.
Ожидаемый результат:	Алгоритм должен вернуть минимальное остовное дерево, учитывая все вершины и связанные ребра в графе, даже при ограниченном количестве ребер.
Полученный результат:	 <p>Корректное выполнение работы.</p>

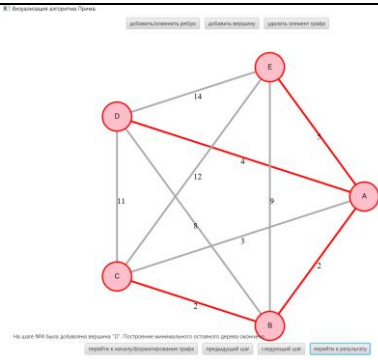
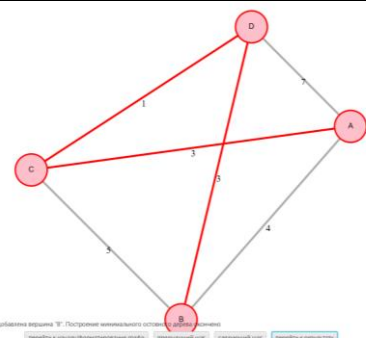
4.2. Тестирование граничных условий

Тест на пустой граф	
Входные данные:	<p>Пустой граф (без вершин и ребер).</p> <pre> Как вы хотите ввести граф : "1" - из файла(матрица смежности), "2" - из консоли(матрица смежности), "3" - в режиме реального времени самому нарисовать граф. 2 Матрица смежности должна быть представлена в формате : симметричная матрица с "-1" на главной диагонали, где "-1" означает отсутствие ребра между двумя вершинами. Сколько вершин должно быть в графе? 0 </pre>
Ожидаемый результат:	Алгоритм должен вернуть пустое минимальное остовное дерево.
Полученный результат:	 <p>Корректное выполнение работы.</p>
Тест на граф с одной вершиной	

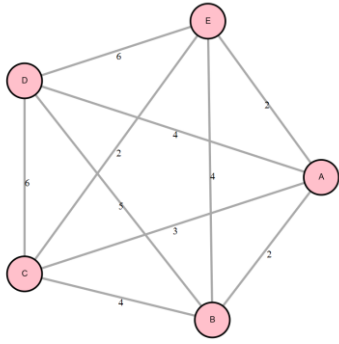

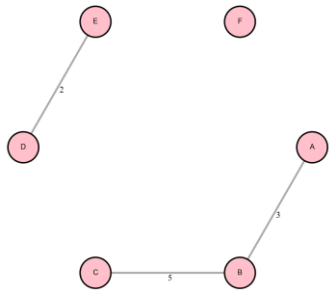
Входные данные:	<p>Граф с одной вершиной.</p> <pre> Как вы хотите ввести граф : "1" - из файла(матрица смежности), "2" - из консоли(матрица смежности), "3" - в режиме реального времени самому нарисовать граф. 2 Матрица смежности должна быть представлена в формате : симметричная матрица с "-1" на главной диагонали, где "-1" означает отсутствие ребра между двумя вершинами. Сколько вершин должно быть в графе? 1 </pre>
Ожидаемый результат:	Алгоритм должен вернуть пустое минимальное остовное дерево.
Полученный результат:	 <p>Созданный граф является несвязным или задано не более двух вершин.</p> <p>Корректное выполнение работы.</p>
Тест на граф без ребер	
Входные данные:	<p>Граф с несколькими вершинами, но без ребер.</p> <pre> Введите матрицу смежности: -1 -1 -1 -1 -1 -1 -1 -1 -1 </pre>
Ожидаемый результат:	Алгоритм должен вернуть пустое минимальное остовное дерево.
Полученный результат:	 <p>Созданный граф является несвязным или задано не более двух вершин.</p>

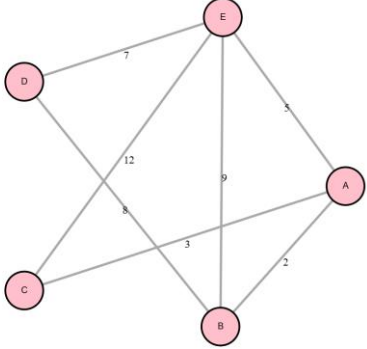
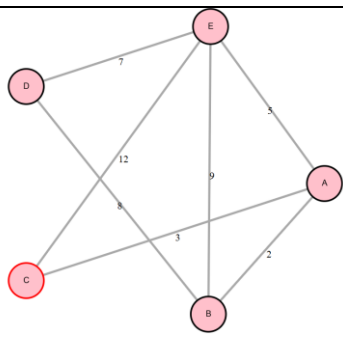
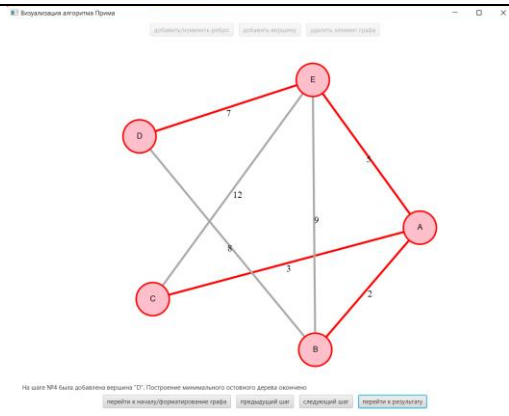
	Корректное выполнение работы.
Тест на граф с одним ребром	
Входные данные:	Граф с двумя вершинами и одним ребром.
Ожидаемый результат:	Алгоритм должен вернуть минимальное остовное дерево, состоящее из этого единственного ребра.
Полученный результат:	 <p>Корректное выполнение работы.</p>
Тест на граф с циклом	
Входные данные:	Граф с несколькими вершинами и циклическим путем.
Ожидаемый результат:	Алгоритм должен вернуть минимальное остовное дерево, исключив ребра, образующие цикл.
Полученный результат:	 <p>Корректное выполнение работы.</p>

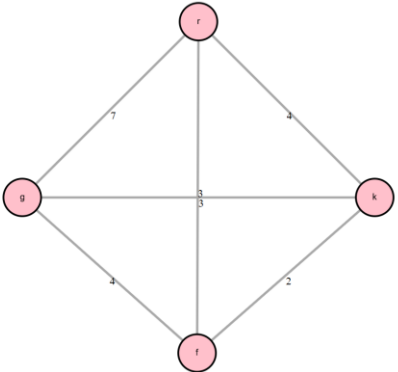
4.3. Тестирование интерфейса

Тестирование интерфейса	
Входные данные:	<p>Граф, представленный пользователю через интерфейс и введенный через файл.</p> <p>Как вы хотите ввести граф : "1" - из файла(матрица смежности), "2" - из консоли(матрица смежности), "3" - в режиме реального времени самому нарисовать граф.</p> <p>1</p> <p>Матрица смежности должна быть представлена в формате : симметричная матрица с "-1" на главной диагонали, где "-1" означает отсутствие ребра между двумя вершинами.</p> <p>Если вы желаете задать имена вершин в файле, введите "1". Введите "2", чтобы имена вершин были заданы автоматически.</p> <p>2</p>
Ожидаемый результат:	Алгоритм должен корректно обработать введенные пользователем данные и вернуть минимальное остовное дерево для данного графа.
Полученный результат:	 <p>Корректное выполнение работы.</p>
Входные данные:	<p>Граф, представленный пользователю через интерфейс и введенный через консоль.</p> <p>Как вы хотите ввести граф : "1" - из файла(матрица смежности), "2" - из консоли(матрица смежности), "3" - в режиме реального времени самому нарисовать граф.</p> <p>2</p> <p>Матрица смежности должна быть представлена в формате : симметричная матрица с "-1" на главной диагонали, где "-1" означает отсутствие ребра между двумя вершинами.</p> <p>Сколько вершин должно быть в графе?</p> <p>4</p> <p>Если вы желаете задать имена вершин, введите "1". Введите "2", чтобы имена вершин были заданы автоматически.</p> <p>2</p> <p>Введите матрицу смежности:</p> <pre>-1 4 3 7 4 -1 5 3 3 5 -1 1 7 3 1 -1</pre>
Ожидаемый результат:	Алгоритм должен корректно обработать введенные пользователем данные и вернуть минимальное остовное дерево для данного графа.
Полученный результат:	 <p>Корректное выполнение работы.</p>

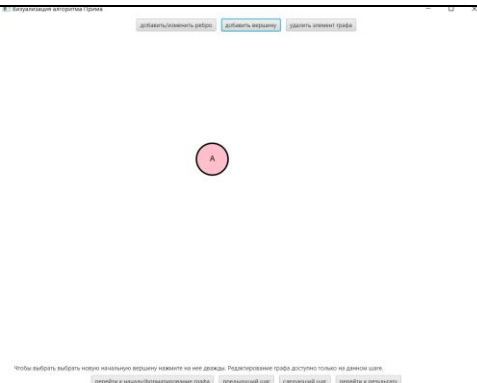
<p>Входные данные:</p>	<p>Граф, представленный пользователю через интерфейс и введенный через файл (нижнестреугольная матрица).</p> <div data-bbox="432 264 1350 584"> <p>Как вы хотите ввести граф : "1" - из файла(матрица смежности), "2" - из консоли(матрица смежности), "3" - в режиме реального времени самому нарисовать граф.</p> <p>1</p> <p>Матрица смежности должна быть представлена в формате : симметричная матрица с "-1" на главной диагонали, где "-1" означает отсутствие ребра между двумя вершинами.</p> <p>Если вы желаете задать имена вершин в файле, введите "1". Введите "2", чтобы имена вершин были заданы автоматически.</p> <p>2</p> </div>
<p>Ожидаемый результат:</p>	<p>Алгоритм должен корректно обработать введенные пользователем данные и вернуть минимальное остовное дерево для данного графа.</p>
<p>Полученный результат:</p>	<div data-bbox="692 712 1190 1178"> <p>На шаге КМ была добавлена вершина "D". Построение минимального остовного дерева окончено.</p> <p>перейти к началу/форматирование графа предыдущий шаг следующий шаг перейти к результату</p> </div> <p>Корректное выполнение работы.</p>
<p>Входные данные:</p>	<p>Граф, представленный пользователю через интерфейс и введенный через консоль (нижнестреугольная матрица)..</p> <div data-bbox="432 1368 1034 1809"> <p>Как вы хотите ввести граф :</p> <p>"1" - из файла(матрица смежности),</p> <p>"2" - из консоли(матрица смежности),</p> <p>"3" - из файла(нижнестреугольная матрица),</p> <p>"4" - из консоли(нижнестреугольная матрица),</p> <p>"5" - в режиме реального времени самому нарисовать граф.</p> <p>4</p> <p>Матрица должна быть представлена в формате : нижнестреугольная матрица с "-1" на главной диагонали, где "-1" означает отсутствие ребра между двумя вершинами.</p> <p>Сколько вершин должно быть в графе?</p> <p>5</p> <p>Если вы желаете задать имена вершин, введите "1". Введите "2", чтобы имена вершин были заданы автоматически.</p> <p>2</p> <p>Введите матрицу:</p> <pre>-1 2 -1 3 4 -1 4 5 6 -1 2 4 2 6 -1</pre> </div>
<p>Ожидаемый результат:</p>	<p>Алгоритм должен корректно обработать введенные пользователем данные и вернуть минимальное остовное дерево для данного графа.</p>

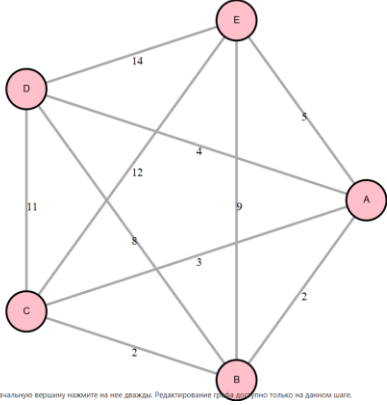
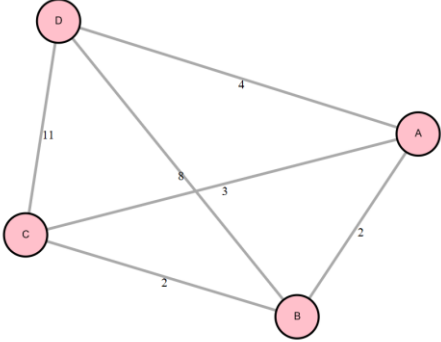
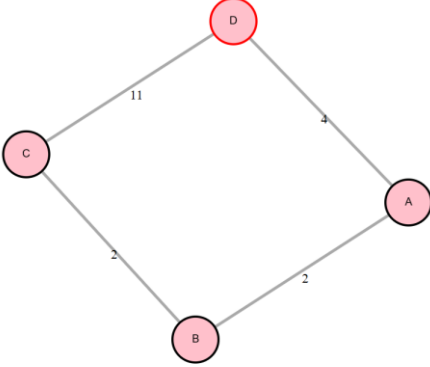
Полученный результат:	 <p>Корректное выполнение работы.</p>
Тест на случай отсутствия графа	
Входные данные:	Отсутствие графа
Ожидаемый результат:	Алгоритм должен обработать данную ситуацию и вернуть пустой результат.
Полученный результат:	 <p>Корректное выполнение работы.</p>
Тест на случай некорректных входных данных	
Входные данные:	<p>Граф, содержащий некорректные или неполные данные (несвязный граф)</p> 
Ожидаемый результат:	Алгоритм должен обработать данную ситуацию и вернуть сообщение об ошибке или пустой результат
Полученный результат:	<p>Созданный граф является несвязным или задано не более двух вершин.</p> <p>Корректное выполнение работы.</p>

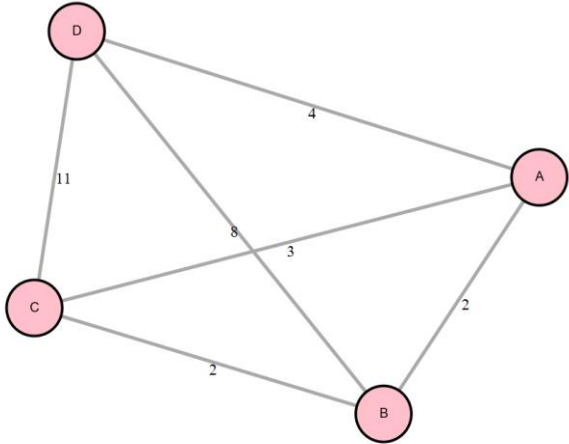
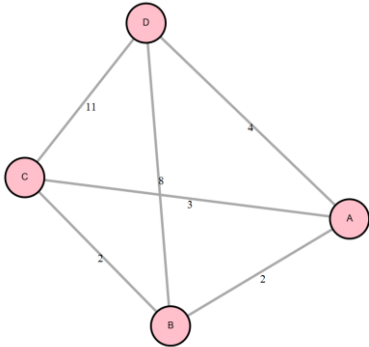
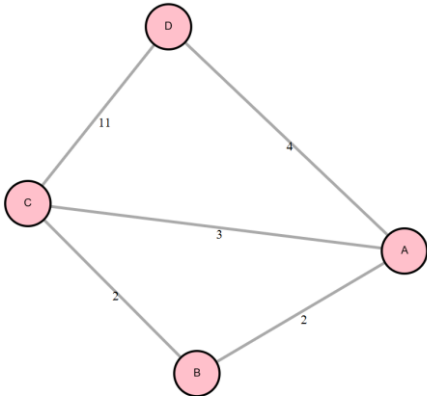
Тест на задание новой стартовой вершины двойным щелчком	
Входные данные:	<p>Граф с несколькими вершинами и ребрами.</p> 
Ожидаемый результат:	После двойного щелчка должна быть выделена новая стартовая вершина.
Полученный результат:	 <p>Корректное выполнение работы.</p>
Тест на блокировку кнопок	
Входные данные:	Граф с несколькими вершинами и ребрами.
Ожидаемый результат:	После получения результата должна будет произойти блокировка кнопок: «добавить/изменить ребро», «добавить вершину», «удалить элемент графа».
Полученный результат:	 <p>Корректное выполнение работы.</p>

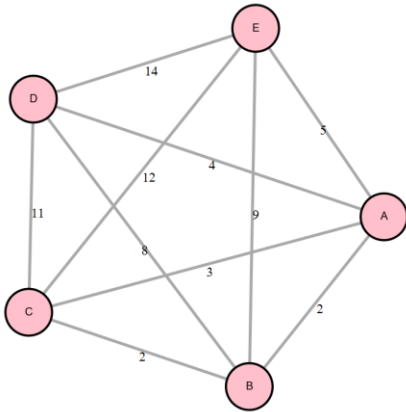
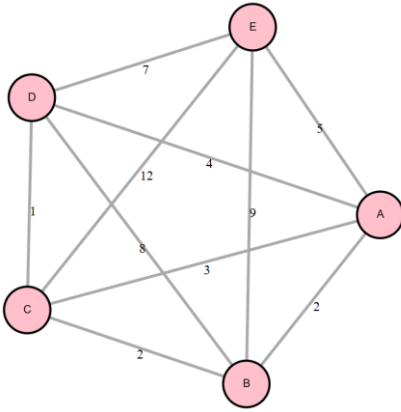
Тест на ввод имён вершин с консоли	
Входные данные:	<p>Граф с несколькими вершинами и ребрами. Имена вершин задаются с консоли.</p> <pre> Если вы желаете задать имена вершин, введите "1". Введите "2", чтобы имена вершин были заданы автоматически. 1 Введите имена вершин через пробел: k f g r </pre>
Ожидаемый результат:	Вершины будут иметь имена через консоль.
Полученный результат:	 <p>Корректное выполнение работы.</p>

4.4. Тестирование структуры данных

Тест на добавление вершины в граф	
Входные данные:	Пустой граф, команда добавления вершины.
Ожидаемый результат:	Вершина успешно добавлена в граф.
Полученный результат:	 <p>Корректное выполнение работы.</p>

Тест на удаление вершины из графа	
Входные данные:	<p>Граф с несколькими вершинами, команда удаления одной из вершин.</p>  <p><small>Для новой начальной вершины нажмите на нее дважды. Редактирование графа возможно только на данном шаге.</small></p>
Ожидаемый результат:	Вершина успешно удалена из графа и связанные с ней ребра также удалены.
Полученный результат:	 <p>Корректное выполнение работы.</p>
Тест на добавление ребра в граф	
Входные данные:	<p>Граф с несколькими вершинами, команда добавления ребра между двумя вершинами.</p> 

Ожидаемый результат:	Ребро успешно добавлено в граф.
Полученный результат:	 <p>Корректное выполнение работы.</p>
Тест на удаление ребра из графа	
Входные данные:	<p>Граф с несколькими вершинами и ребрами, команда удаления одного из ребер.</p> 
Ожидаемый результат:	Ребро успешно удалено из графа.
Полученный результат:	 <p>Корректное выполнение работы.</p>

Тест на изменение веса ребра	
Входные данные:	<p>Граф с некоторым количеством вершин и количеством ребер.</p> 
Ожидаемый результат:	Успешная замена веса ребра.
Полученный результат:	<p>Вес на ребрах DE, CD успешно изменен .</p>  <p>Корректное выполнение работы.</p>

ЗАКЛЮЧЕНИЕ

В результате выполнения данной работы было разработано приложение на языке Kotlin, которое реализует алгоритм Прима с использованием графического интерфейса, созданного с помощью библиотеки JavaFX. Приложение позволяет пользователю визуализировать минимальное остовное дерево графа и понять, как работает алгоритм Прима. Реализация на Kotlin и использование JavaFX позволяют создать легко читаемый и понятный код, а также обеспечивают гибкость и возможность дальнейшего расширения приложения.

Данная работа позволила развить навыки программирования на языке Kotlin и научиться использовать графическую библиотеку JavaFX для создания интерактивных приложений. В дальнейшем можно расширить приложение, например, улучшив пользовательский интерфейс для удобства использования.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Алгоритм Прима Вики конспекты. URL:
https://neerc.ifmo.ru/wiki/index.php?title=Алгоритм_Прима (дата обращения: 01.07.2023).
2. Kotlin docs. URL:
<https://kotlinlang.org/docs/home.html> (дата обращения: 01.07.2023).
3. JavaFX DOCUMENTATION. URL:
<https://openjfx.io/> (дата обращения: 01.07.2023).
4. Курс введение в Kotlin JVM. URL:
<https://stepik.org/course/5448/syllabus> (дата обращения: 30.06.2023).

ПРИЛОЖЕНИЕ А

КОД ПРОГРАММЫ

Файл Main.kt

```
package org.jetbrains.kotlin.Math

import javafx.application.Application

val scan = java.util.Scanner(System.`in`)
fun main() {
    Application.launch(Vizualisation::class.java)
}
```

Файл Edge.kt

```
package org.jetbrains.kotlin.Math

import javafx.scene.Group
import javafx.scene.control.Label
import javafx.scene.paint.Color
import javafx.scene.shape.Line
import javafx.scene.text.Font
/*
 * Класс, представляющий ребро графа.
 *
 * @property Vertex1 Первая вершина ребра [VertexVizualisation].
 * @property Vertex2 Вторая вершина ребра [VertexVizualisation].
 * @property weight Вес ребра.
 * @property line Линия, представляющая ребро.
 * @property edgegroup Группа, содержащая линию и метку ребра.
 * @property line_label Метка, отображающая вес ребра.
 * @property position_1 Позиция первой вершины в графе.
 * @property position_2 Позиция второй вершины в графе.
 */

class Edge(Vertex1 : VertexVizualisation, Vertex2:
VertexVizualisation,private val weight : Int) : Line(){
    private var line : Line
    private var edgegroup : Group
    private val line_label : Label
    private var position_1 : Int = Vertex1.get_number() - 1
    private var position_2 : Int = Vertex2.get_number() - 1
    /**
     * Инициализация ребра.
     *
     * Создает линию, метку ребра и привязывает их к вершинам графа.
     */
    init{
        val lineWidth = 4.0
        line = Line()
        line.stroke = Color.DARKGRAY
        line.strokeWidth = lineWidth
        line_label = Label("$weight")
        line_label.font = Font.font("Times New Roman",18.0)
        // Привязываем начало линии к окружности circle1
```

```

line.startXProperty().bind(Vertex1.get_circle().centerXProperty())

line.startYProperty().bind(Vertex1.get_circle().centerYProperty())
    line.endXProperty().bind(Vertex2.get_circle().centerXProperty())
    line.endYProperty().bind(Vertex2.get_circle().centerYProperty())
    line_label.textFill = Color.BLACK
    line_label.layoutX = (line.startX + line.endX) / 2.0
    line_label.layoutY = (line.startY + line.endY) / 2.0
    // Привязываем конец линии к окружности circle2
    edgegroup = Group(line, line_label)
}
/**
 * Возвращает группу, содержащую линию и метку ребра.
 */
fun group() : Group { return edgegroup }

/**
 * Возвращает вес ребра.
 */
fun get_weight() : Int { return weight}

/**
 * Возвращает линию, представляющую ребро.
 */
fun get_line() : Line { return line}

/**
 * Возвращает метку ребра.
 */
fun get_label() : Label { return line_label}

/**
 * Уменьшает позиции вершин ребра на указанное значение индекса.
 *
 * @param index Индекс вершины (1 или 2).
 */
fun decrease_positions(index : Int) {
    if (index == 1) position_1--
    else position_2--
}
/**
 * Возвращает позиции вершин ребра.
 *
 * @return Пара значений: первая вершина, вторая вершина.
 */
fun get_positions() : Pair<Int, Int> { return
Pair(position_1, position_2) }
}

```

Файл Graph.kt

```

package org.jetbrains.kotlin.Math

import java.io.File
import java.util.*
import kotlin.system.exitProcess

```

```

/**

```

```

* Ккласс для графа. Граф представлен матрицей смежности, если ребра
между вершинами нет, то в соответствующей ячейке будет значение
Int.MAXVALUE.
*
* @property start_vertex начальная вершина
* @property name_vertex список имен вершин
* @property data матрица смежности графа
*/
open class Graph(var start_vertex: Int = 0) {
    var name_vertex: MutableList<String> = mutableListOf()
    var data = mutableListOf<MutableList<Int>>()
    /**
     * Метод для генерации имени вершины по умолчанию.
     *
     * @param quotient порядок вершины для генерации имени вершины
     * @return сгенерированное имя вершины
     */
    fun default_name(quotient: Int): String {
        val base = 26
        val sb = StringBuilder()
        var coefficient = quotient
        while (coefficient > 0) {
            coefficient--
            val remainder = coefficient % base
            sb.append(('A'.code + remainder).toChar())
            coefficient /= base
        }
        return (sb.reverse().toString())
    }

    /**
     * Метод для чтения матрицы смежности из файла.
     *
     * @param filename имя файла, из которого будет производиться чтение
     */
    fun read_from_file(filename: String) {
        try {
            val file = File(filename)
            val scanner = Scanner(file)
            println("Если вы желаете задать имена вершин в файле, введите\n\"1\". Введите \"2\", чтобы имена вершин были заданы автоматически.")
            val flag = readln().toInt()
            if (flag == 1) this.name_vertex = scanner.nextLine().split(" ").toMutableList()
            while (scanner.hasNextLine()) {
                val line = scanner.nextLine()
                val row = line.split(" ").map { it.toInt() }
            }.toMutableList()
            data.add(row)
        }
        if (flag == 2) {
            for (i in 0 until data.size) {
                this.name_vertex.add(default_name(i + 1))
            }
        }
        modificate()
    } catch (e: Exception) {
    }
}

```

```

        println("Файл не найден: $filename")
    }
}

/**
 * Метод для чтения матрицы смежности из консоли.
 */
fun read_from_console() {
    val scan = Scanner(System.`in`)
    println("Сколько вершин должно быть в графе?")
    val size = readln().toInt()
    println("Если вы желаете задать имена вершин, введите \"1\". Введите \"2\", чтобы имена вершин были заданы автоматически.")
    val flag = readln().toInt()
    when (flag) {
        1 -> {
            println("Введите имена вершин через пробел: ")
            this.name_vertex = scan.nextLine().split(" ").toMutableList()
        }

        2 -> {
            for (i in 0 until size) {
                this.name_vertex.add(default_name(i + 1))
            }
        }
    }
    println("Введите матрицу: ")
    for (i in 0 until size) {
        val line = scan.nextLine()
        val row = line.split(" ").map { it.toInt() }.toMutableList()
        data.add(row)
    }
    modificate()
}

/**
 * Метод для отражения матрицы смежности (симметричность графа).
 */
fun reflect_matrix(){
    if (check_correct_matrix()) {
        for (i in 0 until data.size) {
            for (j in (i + 1) until data.size) {
                data[i].add(data[j][i])
            }
        }
    }
    else {
        println("Матрица введена неверно. Невозможно создать граф. Попробуйте снова.")
        exitProcess(0)
    }
}

/**
 * Метод для проверки корректности матрицы смежности.
 *
 * @return true, если матрица корректна, иначе false
 */

```

```

fun check_correct_matrix(): Boolean {
    var flag = true
    for (i in 0 until data.size) {
        if (data[i].size != (i + 1)) flag = false
    }
    return flag
}
/**
 * Метод для проверки симметричности матрицы смежности.
 */
fun check_symmetry(){
    var check = true
    for (i in 0 until data.size) {
        if (check) {
            for (j in 0 until data.size) {
                if (data[i][j] != data[j][i]) {
                    println("Матрица смежности несимметрична.")
                    check = false
                }
            }
        }
    }
}

// метод для переопределения весов ребер(если в файле было -1, значит
ребра нет)
fun modificate() {
    for (i in 0 until data.size) {
        for (j in 0 until data[i].size) {
            if (data[i][j] == -1 || i == j) data[i][j] =
Int.MAX_VALUE
        }
    }
}

// метод, который возвращает строку для вывода информации о графе в
консоль
override fun toString(): String {
    var string = ""
    data.forEach() {
        string += it.toString() + "\n"
    }
    return string
}
/**
 * Метод для установки вершины начала.
 *
 * @param new_start_vertex новая вершина начала
 */
fun set_start_vertex(new_start_vertex: Int) {
    this.start_vertex = new_start_vertex
}

/**
 * Реализация алгоритма Прима для графа, возвращающая массив пар,
соответствующих началу и концу ребра.
 */

```

```

    * @return пара, содержащая массив пар ребер и массив массивов пар
    ребер, а также список добавленных вершин
    */
    fun PrimAlgorithm(): Pair<Pair<MutableList<Pair<Int, Int>>,
    MutableList<MutableList<Pair<Int, Int>>>>, MutableList<Int>> {
        val edges_considered_at_the_step:
    MutableList<MutableList<Pair<Int, Int>>> = mutableListOf()
        val result_edges: MutableList<Pair<Int, Int>> = mutableListOf()
        val added_vertexes: MutableList<Int> = mutableListOf()
        added_vertexes.add(this.start_vertex)
        result_edges.add(Pair(0, 0))
        var index = 0
        while (added_vertexes.size < data.size) {
            edges_considered_at_the_step.add(mutableListOf())
            iterated_PrimAlgorithm(result_edges, added_vertexes,
edges_considered_at_the_step[index])
            index++
        }
        edges_considered_at_the_step.add(mutableListOf())
        return Pair(Pair(result_edges, edges_considered_at_the_step),
added_vertexes)
    }
    /**
    * Метод для проверки корректности графа.
    *
    * @return true, если граф корректен, иначе false
    */
    fun correct_graph(): Boolean {
        for (i in 0 until data.size) {
            var flag = false
            for (j in 0 until data[i].size) {
                if (data[i][j] != Int.MAX_VALUE) flag = true
            }
            if (!flag) {
                println("Созданный граф является несвязным или задано не
более двух вершин.")
                return false
            }
        }
        return true
    }
    /**
    * Итерационная реализация алгоритма Прима для графа.
    *
    * @param result_edges список ребер
    * @param added_vertexes список добавленных вершин
    * @param edges_considered_at_the_step список ребер, рассмотренных на
текущем шаге
    */
    fun iterated_PrimAlgorithm(result_edges: MutableList<Pair<Int, Int>>,
added_vertexes: MutableList<Int>,
edges_considered_at_the_step:
    MutableList<Pair<Int, Int>> ) {
        var min = Int.MAX_VALUE
        var new_edge = 0 to 0
        for (i in 0 until added_vertexes.size) {
            for (j in 0 until data[added_vertexes[i]].size) {

```



```

        if (j !in added_vertexes) {
            if (data[added_vertexes[i]][j] < Int.MAX_VALUE)
edges_considered_at_the_step.add(Pair(added_vertexes[i], j))
            if (data[added_vertexes[i]][j] < min) {
                min = data[added_vertexes[i]][j]
                new_edge = Pair(added_vertexes[i], j)
            }
        }
    }
    added_vertexes.add(new_edge.second)
    result_edges.add(new_edge)
}
/**
 * Метод для создания новой вершины в графе.
 *
 * @param name имя новой вершины
 */
fun create_new_vertex(name: String) {
    for (i in 0 until data.size) data[i].add(Int.MAX_VALUE)
    data.add(MutableList(data.size + 1) { Int.MAX_VALUE })
    name_vertex.add(name)
}
/**
 * Метод для удаления вершины из графа.
 *
 * @param number номер удаляемой вершины
 */
fun delete_vertex(number: Int) {
    data.removeAt(number - 1)
    for (string in data) {
        string.removeAt(number - 1)
    }
}
/**
 * Метод для получения матрицы смежности графа.
 *
 * @return матрица смежности графа
 */
fun get_matrix(): MutableList<MutableList<Int>> { return data }
/**
 * Метод для получения списка имен вершин графа.
 *
 * @return список имен вершин графа
 */
fun get_names(): MutableList<String> { return name_vertex }
}

```

Файл Vizualisation.kt

```

package org.jetbrains.kotlin.Math

import javafx.application.Application
import javafx.application.Platform
import javafx.scene.Group
import javafx.scene.Scene

```

```

import javafx.scene.control.*
import javafx.scene.layout.*
import javafx.scene.paint.Color
import javafx.stage.Stage

/**
 * Класс для визуализации графа и управления визуализацией.
 *
 * @property graph_visual объект класса GraphVizualisation для
визуализации графа
 * @property full_group группа, содержащая все элементы интерфейса
 * @property step_information информация о текущем шаге визуализации
 * @property graph_editor объект класса GraphEditor для редактирования
графа
 */
class Vizualisation : Application() {
    private lateinit var graph_visual : GraphVizualisation
    private lateinit var full_group : Group
    private lateinit var step_information : Label
    private val graph_editor = GraphEditor()

    /**
     * Метод, вызываемый при запуске приложения.
     *
     * @param stage текущее окно приложения
     */
    override fun start(stage: Stage) {
        val graph = Graph()
        val filename = "Graph1.txt"
        stage.title = "Визуализация алгоритма Прима"
        stage.width = 1000.0
        stage.height = 800.0
        val new_vertex = Button("добавить вершину")
        val new_edge = Button("добавить/изменить ребро")
        val delete = Button("удалить элемент графа")
        val next_step = Button("следующий шаг")
        val previous_step = Button("предыдущий шаг")
        val final = Button("перейти к результату")
        val first_step = Button("перейти к началу/форматирование графа")
        val movements = HBox(first_step, previous_step, next_step, final)
        val operations = HBox(new_edge, new_vertex, delete)
        operations.spacing = 10.0
        movements.spacing = 10.0
        new_vertex.setOnAction { graph_editor.create_vertex(stage,
graph_visual, step_information, final, first_step, previous_step, next_step)
        }

        new_edge.setOnAction { graph_editor.create_edge(stage,
graph_visual, step_information, final, first_step, previous_step, next_step)
        }

        delete.setOnAction { graph_editor.delete_element(stage,
graph_visual, step_information, final, first_step, previous_step, next_step)
        }

        next_step.setOnAction {
            if (graph_visual.get_step() == -1) {
                new_vertex.isDisable = true
                new_edge.isDisable = true
                delete.isDisable = true
            }
        }
    }
}

```

```

        if (graph_visual.graph.correct_graph())
step_by_step_algorithm(graph_visual.next_step(),new_edge,new_vertex,delete)
    }
    else
step_by_step_algorithm(graph_visual.next_step(),new_edge,new_vertex,delete)
    }
    previous_step.setOnAction {
step_by_step_algorithm(graph_visual.previous_step(),new_edge,new_vertex,delete) }
    final.setOnAction {
        if (graph_visual.get_step() == -1) {
            new_vertex.isDisable = true
            new_edge.isDisable = true
            delete.isDisable = true
            if (graph_visual.graph.correct_graph())
step_by_step_algorithm(graph_visual.set_get_step(graph_visual.graph.data.size - 1), new_edge, new_vertex,delete)
        }
        else
step_by_step_algorithm(graph_visual.set_get_step(graph_visual.graph.data.size - 1),new_edge,new_vertex,delete)
    }
    first_step.setOnAction {
        new_vertex.isDisable = false
        new_edge.isDisable = false
        delete.isDisable = false
        draw_graph(stage, graph, operations, movements)
    }
    println("Как вы хотите ввести граф : \n\"1\" - из файла(матрица смежности), \n\"2\" - из консоли(матрица смежности), \n\"3\" - из файла(нижнетреугольная матрица), \" +
        \"\n\"4\" - из консоли(нижнетреугольная матрица), \n\"5\" - в режиме реального времени самому нарисовать граф.")
    val variant = scan.nextInt()
    when (variant) {
        1 -> {
            println("Матрица смежности должна быть представлена в формате : симметричная матрица с \"-1\" на главной диагонали, где \"-1\" означает отсутствие ребра между двумя вершинами.")
            graph.read_from_file(filename)
            graph.check_symmetry()
        }
        2 -> {
            println("Матрица смежности должна быть представлена в формате : симметричная матрица с \"-1\" на главной диагонали, где \"-1\" означает отсутствие ребра между двумя вершинами.")
            graph.read_from_console()
            graph.check_symmetry()
        }
        3 -> {
            println("Матрица должна быть представлена в формате : нижнетреугольная матрица с \"-1\" на главной диагонали, где \"-1\" означает отсутствие ребра между двумя вершинами.")
            graph.read_from_file(filename)

```

```

        graph.reflect_matrix()
    }
    4 -> {
        println("Матрица должна быть представлена в формате :
нижнетреугольная матрица с \"-1\" на главной диагонали, где \"-1\"
означает отсутствие ребра между двумя вершинами.")
        graph.read_from_console()
        graph.reflect_matrix()
    }
}
draw_graph(stage, graph, operations, movements)
if (operations.width == 0.0) {
    Platform.runLater {
        val width1 = operations.width
        val width2 = movements.width
        val height2 = movements.height
        operations.layoutX = stage.width / 2.0 - width1 / 2.0
        operations.layoutY = 5.0
        movements.layoutX = stage.width / 2.0 - width2 / 2.0
        movements.layoutY = stage.height - 3.0 * height2
    }
}
stage.show()
}

/**
 * Метод для визуализации алгоритма Прима пошагово.
 *
 * @param step текущий шаг алгоритма
 * @param new_edge кнопка "добавить/изменить ребро"
 * @param new_vertex кнопка "добавить вершину"
 * @param delete кнопка "удалить элемент графа"
 */
fun step_by_step_algorithm(step: Int, new_edge: Button, new_vertex:
Button, delete: Button) {
    val step_data = graph_visual.graph.PrimAlgorithm()
    val result_edges = step_data.first.first
    val edges_considered_at_the_step = step_data.first.second
    val considered_vertexes = step_data.second
    for (edge_list in graph_visual.get_edges()) {
        for (edge in edge_list) {
            edge.get_line().stroke = Color.DARKGRAY
            if (step != -1) {
                if (Pair(edge.get_positions().first,
edge.get_positions().second) in
                    result_edges.subList(0, step + 1) ||
Pair(edge.get_positions().second,
                    edge.get_positions().first) in
result_edges.subList(0, step + 1)) {
                    edge.get_line().stroke = Color.RED
                } else if (Pair(edge.get_positions().first,
edge.get_positions().second) in
                    edges_considered_at_the_step[step] ||
Pair(edge.get_positions().second,
                    edge.get_positions().first) in
edges_considered_at_the_step[step]) {
                    edge.get_line().stroke = Color.BLUE
                }
            }
        }
    }
}

```

```

        }
    }
}
for (vertex in graph_visual.get_vertexes())
    vertex.get_circle().stroke = Color.BLACK
if (step == -1) {
    new_vertex.isDisable = false
    new_edge.isDisable = false
    delete.isDisable = false

graph_visual.get_vertexes()[graph_visual.graph.start_vertex].get_circle()
.stroke = Color.RED
}
for (i in 0 until step + 1)

graph_visual.get_vertexes()[considered_vertexes[i]].get_circle().stroke =
Color.RED
    update_step_information(step, considered_vertexes,
edges_considered_at_the_step)
}

/**
 * Метод обновляет информацию о текущем шаге алгоритма Прима и
отображает ее на экране.
 *
 * @param step текущий шаг алгоритма
 * @param consideredvertexes список рассматриваемых вершин на каждом
шаге
 * @param addededges список добавленных ребер на каждом шаге
 */
fun update_step_information(step: Int, considered_vertexes:
MutableList<Int>,
                        added_edges:
MutableList<MutableList<Pair<Int, Int>>>, ) {
    var weights = ""
    if (step >= 0) {
        for (edge in added_edges[step])
            weights += "
${graph_visual.graph.data[edge.first][edge.second]}"
    }
    if (step < graph_visual.graph.data.size - 1 && step > 0)
step_information.text = "На шаге №$step была " +
        "добавлена вершина
\"${graph_visual.get_vertexes()[considered_vertexes[step]].get_name().tex
t}\". Ребра, " +
        "рассматриваемые на данном шаге, имеют веса $weights"
    else if (step == 0) step_information.text = "Вершина \" " +
        "${graph_visual.get_vertexes()[considered_vertexes[0]].get_name().text}\"
была выбрана в качестве начальной. " +
        "Ребра, рассматриваемые на данном шаге, имеют веса
$weights"
    else if (step == graph_visual.graph.data.size - 1)
step_information.text = "На шаге №$step была добавлена " +
        "вершина
\"${graph_visual.get_vertexes()[considered_vertexes.last()].get_name().te

```

```

xt}\". Построение минимального " +
        "остовного дерева окончено"
        else step_information.text = "Чтобы выбрать новую начальную
вершину нажмите на нее дважды." +
        " Редактирование графа доступно только на данном шаге."
    }

    /**
     * Метод для визуализации окна.
     *
     * @param stage текущее окно приложения
     * @param graph граф, который нужно отрисовать
     * @param operations группа с кнопками операций над графом
     * @param movements группа с кнопками перемещения по визуализации
     */
    fun draw_graph(stage: Stage, graph: Graph, buttons1: HBox, buttons2:
HBox) {
        graph_visual = GraphVizualisation(stage.height, graph)
        step_information = Label("Чтобы выбрать новую начальную вершину
нажмите на нее дважды." +
        " Редактирование графа доступно только на данном
шаге.")
        step_information.layoutX = 30.0
        step_information.layoutY = stage.height - 100.0
        full_group = Group(graph_visual.group(), buttons1, buttons2,
step_information)
        val scene = Scene(full_group, 1000.0, 800.0)
        stage.setScene(scene)
        graph_editor.action(stage, graph_visual)
    }
}

```

Файл VertexVizualisation.kt

```

package org.jetbrains.kotlin.Math

import javafx.scene.Group
import javafx.scene.paint.Color
import javafx.scene.shape.Circle
import javafx.scene.text.Font
import javafx.scene.text.Text
/**
 * Класс, представляющий визуализацию вершины графа.
 *
 * Этот класс используется для создания и визуализации вершины графа на
графической сцене.
 * Визуализация вершины состоит из окружности и текстовой метки с
названием вершины.
 * Класс позволяет получить номер вершины, уменьшить его на единицу,
получить группу, содержащую окружность и текстовую метку,
 * а также получить окружность и метку вершины отдельно.
 *
 * @param scene_size Размер сцены, на которой будет располагаться
вершина.
 * @param x Координата x вершины на сцене.
 * @param y Координата y вершины на сцене.

```

```

* @param name_vertex Название вершины.
* @param number Номер вершины.
*
* @property name Текстовая метка с названием вершины.
* @property circle Окружность, представляющая вершину графа.
* @property data Группа, содержащая окружность и метку вершины.
*/class VertexVizualisation(scene_size : Double, x : Double, y : Double,
name_vertex: String,private var number: Int ) : Circle(){
    /**
     * Текстовая метка с названием вершины.
     */
    private var name: Text

    /**
     * Окружность, представляющая вершину графа.
     */
    private var circle: Circle

    /**
     * Группа, содержащая окружность и метку вершины.
     */
    private var data: Group

    /**
     * Инициализирует визуализацию вершины графа.
     *
     * Создает окружность и текстовую метку с названием вершины, задает
     им внешний вид и привязывает к окружности.
     */
    init {
        circle = Circle(x,y, scene_size / 25.0)
        circle.fill = Color.PINK
        val text = Text(name_vertex)
        text.font = Font.font("Arial",15.0)
        this.name = text
        circle.strokeWidth = 3.0
        circle.stroke = Color.BLACK
        text.fill = Color.BLACK
        text.x = circle.centerX - text.layoutBounds.width / 2.0
        text.y = circle.centerY + text.layoutBounds.height / 4.0
        data = Group(circle,text)
    }

    /**
     * Возвращает номер вершины.
     *
     * @return Номер вершины.
     */
    fun get_number(): Int {
        return number
    }

    /**
     * Уменьшает номер вершины на единицу.
     */
    fun decrease_number() {

```

```

        number--
    }

/**
 * Возвращает группу, содержащую окружность и метку вершины.
 *
 * @return Группа, содержащая окружность и метку вершины.
 */
    fun group() : Group { return data }

    /**
     * Возвращает окружность вершины.
     *
     * @return Окружность вершины.
     */
    fun get_circle() : Circle { return circle }

    /**
     * Возвращает текстовую метку вершины.
     *
     * @return Текстовая метка вершины.
     */
    fun get_name() : Text { return name }
}

```

Файл GraphEditor.kt

```

package org.jetbrains.kotlin.Math

import javafx.scene.control.Alert
import javafx.scene.control.Button
import javafx.scene.control.Label
import javafx.scene.control.TextInputDialog
import javafx.scene.input.MouseButton
import javafx.scene.paint.Color
import javafx.scene.shape.Circle
import javafx.scene.shape.Line
import javafx.stage.Stage

/**
 * Класс GraphEditor предоставляет функциональность для редактирования графа.
 *
 * @property create_vertex Метод для создания новой вершины в графе.
 * @property choose_new_name Метод для выбора имени новой вершины.
 * @property getClickedLine Метод для получения ребра, на которое было произведено нажатие.
 * @property getClickedCircle Метод для получения вершины, на которую было произведено нажатие.
 * @property create_edge Метод для создания нового ребра в графе.
 */
class GraphEditor() {

    /**
     * Метод create_vertex создает новую вершину в графе.
     *
     * @param stage сцена, на которой отображается граф
     * @param graph_visual визуализация графа
     * @param step_information информация о текущем шаге
     */
}

```



```

* @param button1 кнопка 1
* @param button2 кнопка 2
* @param button3 кнопка 3
* @param button4 кнопка 4
*/
fun create_vertex(stage: Stage, graph_visual: GraphVizualisation,
step_information: Label, button1 : Button, button2 : Button, button3 :
Button, button4 : Button) {
    if (graph_visual.get_step() == -1) {
        able_disable_buttons(button1, button2, button3, button4, true)
        step_information.text = "Выберете место для вставки вершины.
"
        stage.scene.setOnMouseClicked { event ->
            if (event.button == MouseButton.PRIMARY &&
getClickedCircle(event.x, event.y, graph_visual) == null) {
                val new_name = choose_new_name(graph_visual)
                val new_vertex = VertexVizualisation(
                    stage.height, event.x, event.y, new_name,
                    graph_visual.graph.data.size + 1
                )
                graph_visual.add_vertex(new_vertex)
                stage.scene.setOnMouseClicked(null)
                graph_visual.graph.name_vertex.add(new_name)

            }
        }
        able_disable_buttons(button1, button2, button3, button4, false)
        step_information.text = "Чтобы выбрать новую
начальную вершину нажмите на нее дважды." +
            " Редактирование графа доступно только на
данном шаге."
        action(stage, graph_visual)
    }
}

/**
* Метод choose_new_name позволяет выбрать имя для новой вершины.
*
* @param graph_visual визуализация графа
*
* @return имя для новой вершины
*/
fun choose_new_name(graph_visual: GraphVizualisation): String {
    val dialog = TextInputDialog()
    dialog.title = "Ввод имени для новой вершины."
    dialog.headerText = "Оставьте поле пустным или закройте окно,
если хотите, чтобы имя было задано автоматически."
    dialog.contentText = "Пожалуйста, введите имя для новой вершины:"
    val result = dialog.showAndWait()
    var name: String
    if (result.isPresent) {
        name = result.get()
        if (name == "") name =
graph_visual.graph.default_name(graph_visual.graph.name_vertex.size + 1)
    } else name =
graph_visual.graph.default_name(graph_visual.graph.name_vertex.size + 1)
    return name
}

```

```

    }

    /**
     * Метод getClickedLine возвращает ребро, на которое было произведено
     нажатие.
     *
     * @param x координата x нажатия
     * @param y координата y нажатия
     * @param graph_visual визуализация графа
     *
     * @return ребро, на которое было произведено нажатие или null, если
     ребра не было
     */
    fun getClickedLine(x: Double, y: Double, graph_visual:
    GraphVizualisation): Edge?{
        for(edgeLine in graph_visual.get_edges()){
            for (edge in edgeLine){
                if (edge.get_line().contains(x,y)) return edge
            }
        }
        return null
    }

    /**
     * Метод getClickedCircle возвращает вершину, на которую было
     произведено нажатие.
     *
     * @param x координата x нажатия
     * @param y координата y нажатия
     * @param graph_visual визуализация графа
     *
     * @return вершина, на которую было произведено нажатие или null,
     если вершины не было
     */
    fun getClickedCircle(x: Double, y: Double, graph_visual:
    GraphVizualisation): VertexVizualisation? {
        for (circle in graph_visual.get_vertexes()) {
            if (circle.get_circle().contains(x, y)) return circle
        }
        return null
    }

    /**
     * Метод create_edge создает новое ребро в графе.
     *
     * @param stage сцена, на которой отображается граф
     * @param graph_visual визуализация графа
     * @param step_information информация о текущем шаге
     * @param button1 кнопка 1
     * @param button2 кнопка 2
     * @param button3 кнопка 3
     * @param button4 кнопка 4
     */
    fun create_edge(stage: Stage, graph_visual: GraphVizualisation,
    step_information: Label, button1 : Button, button2 : Button, button3 :
    Button, button4 : Button) {
        if (graph_visual.get_step() == -1) {

```

```

        able_disable_buttons(button1,button2,button3,button4,true)
        step_information.text = "Чтобы создать ребро, необходимо
выбрать две вершины и последовательно нажать на них."
        var startCircle: VertexVizualisation? = null
        stage.scene.setOnMouseClicked { event ->
            if (event.button == MouseButton.PRIMARY &&
graph_visual.graph.data.size >= 2) {
                val clickedCircle = getClickedCircle(event.x,
event.y, graph_visual)
                if (clickedCircle != null) {
                    if (startCircle != null) {
                        createLine(startCircle!!, clickedCircle,
graph_visual, step_information)
                        startCircle = null
                        stage.scene.setOnMouseClicked(null)
                    }
                }
            }
        }

        able_disable_buttons(button1,button2,button3,button4,false)
        action(stage, graph_visual)
        } else startCircle = clickedCircle
    }
}

/**
 * Метод choose_weight() открывает диалоговое окно для ввода веса
нового ребра и возвращает введенное значение.
 * Если поле ввода оставлено пустым или окно закрыто, вес ребра будет
равен "0".
 */
@return введенное пользователем значение веса ребра
*/
fun choose_weight(): Int {
    val dialog = TextInputDialog()
    dialog.title = "Ввод веса для нового ребра."
    dialog.headerText = "Если вы оставите поле пустным или закроете
окно, вес ребра будет равен \"0\"."
    dialog.contentText = "Пожалуйста, введите целое число:"
    var weight = 0
    val result = dialog.showAndWait()
    if (result.isPresent) {
        val input = result.get()
        try {
            weight = input.toInt()
        } catch (e: NumberFormatException) {
            val alert = Alert(Alert.AlertType.ERROR)
            alert.title = "Ошибка!"
            alert.headerText = "Неверный формат числа."
            alert.contentText = "Пожалуйста, введите целое число."
            alert.showAndWait()
        }
    }
    return weight
}

/**

```

```

    * Метод createLine создает новое ребро между двумя заданными
    вершинами на графической сцене.
    * Вес ребра запрашивается с помощью метода choose_weight().
    *
    * @param startCircle начальная вершина ребра
    * @param endCircle конечная вершина ребра
    * @param graph_visual визуализация графа
    * @param step_information информация о текущем шаге
    */
    fun createLine(startCircle: VertexVizualisation, endCircle:
    VertexVizualisation, graph_visual: GraphVizualisation, step_information:
    Label) {
        val line = Line()
        line.startX = startCircle.get_circle().centerX
        line.startY = startCircle.get_circle().centerY
        line.endX = endCircle.get_circle().centerX
        line.endY = endCircle.get_circle().centerY
        line.stroke = Color.BLACK
        line.strokeWidth = 2.0
        val weight = choose_weight()
        if (startCircle.get_number() > endCircle.get_number()) {
            val new_edge = Edge(endCircle, startCircle, weight)
            graph_visual.add_edge(new_edge)
        } else {
            val new_edge = Edge(startCircle, endCircle, weight)
            graph_visual.add_edge(new_edge)
        }
        step_information.text = "Чтобы выбрать новую начальную вершину
        нажмите на нее дважды." +
            " Редактирование графа доступно только на данном шаге."
    }

    /**
    * Метод able_disable_buttons позволяет включать или отключать кнопки
    на графической сцене.
    *
    * @param button1 первая кнопка
    * @param button2 вторая кнопка
    * @param button3 третья кнопка
    * @param button4 четвертая кнопка
    * @param action значение true или false, определяющее, включить или
    отключить кнопки
    */
    fun able_disable_buttons(button1 : Button, button2 : Button, button3
    : Button, button4 : Button, action : Boolean){
        button1.isDisable = action
        button2.isDisable = action
        button3.isDisable = action
        button4.isDisable = action
    }

    /**
    * Метод delete_element позволяет пользователю удалить выбранный
    элемент (вершину или ребро) на графической сцене.
    *
    * @param stage сцена, на которой отображается граф
    * @param graph_visual визуализация графа

```

```

    * @param step_information информация о текущем шаге
    * @param button1 первая кнопка
    * @param button2 вторая кнопка
    * @param button3 третья кнопка
    * @param button4 четвертая кнопка
    */
    fun delete_element(stage: Stage, graph_visual: GraphVizualisation,
        step_information: Label, button1 : Button, button2 : Button, button3 :
        Button, button4 : Button) {
        if (graph_visual.get_step() == -1){
            able_disable_buttons(button1,button2,button3,button4,true)
            step_information.text = "Чтобы удалить элемент, нажмите на
него."

            stage.scene.setOnMouseClicked { event ->
                val clicked_circle : Circle?
                val clicked_line : Line?

                clicked_circle =
getClickedCircle(event.x,event.y,graph_visual)
                if (clicked_circle != null) {
                    graph_visual.delete_vertex(clicked_circle)
                }
                clicked_line =
getClickedLine(event.x,event.y,graph_visual)
                if (clicked_line != null){
                    graph_visual.delete_edge(clicked_line)
                }
            }

            able_disable_buttons(button1,button2,button3,button4,false)
            stage.scene.setOnMouseClicked(null)
            action(stage, graph_visual)
            step_information.text = "Чтобы выбрать новую начальную
вершину нажмите на нее дважды." +
                " Редактирование графа доступно только на данном
шаге."
        }
    }

    /**
    * Метод isInsideCircle проверяет, находятся ли указанные координаты
    внутри заданного круга.
    *
    * @param x координата по оси X
    * @param y координата по оси Y
    * @param circle круг для проверки
    * @return true, если координаты находятся внутри круга, иначе false
    */
    fun isInsideCircle(x: Double, y: Double, circle: Circle): Boolean {
        val distance = Math.sqrt(Math.pow(x - circle.centerX, 2.0) +
Math.pow(y - circle.centerY, 2.0))
        return distance <= circle.radius
    }

    /**

```

```

* Метод action позволяет пользователю выполнять действия с
элементами графа.
*
* @param stage сцена, на которой отображается граф
* @param graph_visual визуализация графа
*/
fun action(stage: Stage, graph_visual: GraphVizualisation) {
    stage.scene.setOnMouseClicked { event ->
        if (event.button == MouseButton.PRIMARY && event.clickCount
== 2 && graph_visual.get_step() == -1) {
            for (i in 0 until graph_visual.get_vertexes().size) {
                graph_visual.get_vertexes()[i].get_circle().stroke =
Color.BLACK
                if (isInsideCircle(event.sceneX, event.sceneY,
graph_visual.get_vertexes()[i].get_circle())) {
graph_visual.graph.set_start_vertex(graph_visual.get_vertexes()[i].get_nu
mber() - 1)

graph_visual.get_vertexes()[graph_visual.graph.start_vertex].get_circle()
.stroke = Color.RED
                }
            }
        }
        val deltaX = DoubleArray(graph_visual.get_vertexes().size)
        val deltaY = DoubleArray(graph_visual.get_vertexes().size)
        stage.scene.setOnMousePressed { event ->
            if (event.button == MouseButton.PRIMARY) {
                for (vertex in graph_visual.get_vertexes()) {
                    if (isInsideCircle(event.sceneX, event.sceneY,
vertex.get_circle())) {
                        deltaX[vertex.get_number() - 1] = event.sceneX -
vertex.get_circle().centerX
                        deltaY[vertex.get_number() - 1] = event.sceneY -
vertex.get_circle().centerY
                    }
                }
            }
        }
        stage.scene.setOnMouseDragged { event ->
            if (event.button == MouseButton.PRIMARY) {
                for (vertex in graph_visual.get_vertexes()) {
                    if (isInsideCircle(event.sceneX, event.sceneY,
vertex.get_circle())) {
                        vertex.get_circle().centerX = event.sceneX -
deltaX[vertex.get_number() - 1]
                        vertex.get_circle().centerY = event.sceneY -
deltaY[vertex.get_number() - 1]
                        // Обновляем положение метки в соответствии с
новым положением круга
                        vertex.get_name().x = vertex.get_circle().centerX
- vertex.get_name().layoutBounds.width / 2.0
                        vertex.get_name().y = vertex.get_circle().centerY
+ vertex.get_name().layoutBounds.height / 4.0
                        for (edge_list in graph_visual.get_edges()) {
                            for (edge in edge_list) {

```



```

        for (j in 0 until i + 1) {
            if (graph.data[i][j] != Int.MAX_VALUE) {
                edges[i].add(Edge(vertexes[j], vertexes[i],
graph.data[i][j]))
            }
        }
    }
    for (edge_array in edges) {
        for (edge in edge_array)
            full_graph.children.add(edge.group())
    }
    for (vertex in vertexes) {
        full_graph.children.add(vertex.group())
    }
}

/**
 * Устанавливает и возвращает текущий шаг при визуализации.
 *
 * @param step: Номер шага.
 * @return: Новый номер шага.
 */
fun set_get_step(step: Int): Int {
    this.step = step
    return this.step
}

/**
 * Возвращает список ребер графа.
 *
 * @return: Список ребер графа.
 */
fun get_edges() : MutableList<MutableList<Edge>> { return edges }

/**
 * Возвращает список вершин графа.
 *
 * @return: Список вершин графа.
 */
fun get_vertexes() : MutableList<VertexVizualisation> { return
vertexes }

/**
 * Возвращает объект Group, содержащий все ребра и вершины графа.
 *
 * @return: Объект Group.
 */
fun group() : Group { return full_graph }

/**
 * Возвращает текущий шаг при визуализации.
 *
 * @return: Номер текущего шага.
 */
fun get_step() : Int { return step }

/**

```



```

    * Переходит к следующему шагу при визуализации и возвращает новый
номер шага.
    *
    * @return: Новый номер шага.
    */
    fun next_step(): Int {
        if (step < graph.data.size - 1) step += 1
        else step = graph.data.size - 1
        return step
    }

    /**
    * Переходит к предыдущему шагу при визуализации и возвращает новый
номер шага.
    *
    * @return: Новый номер шага.
    */
    fun previous_step(): Int {
        if (step > -1) step -= 1
        else step = -1
        return step
    }

    /**
    * Добавляет новое ребро в граф.
    *
    * @param new_edge: Новое ребро для добавления.
    */
    fun add_edge(new_edge : Edge) {
        if (new_edge.get_positions().second !=
new_edge.get_positions().first) {

graph.data[new_edge.get_positions().first][new_edge.get_positions().secon
d] = new_edge.get_weight()

graph.data[new_edge.get_positions().second][new_edge.get_positions().firs
t] = new_edge.get_weight()
            for (edge in edges[new_edge.get_positions().second]) {
                if (edge.get_positions().first ==
new_edge.get_positions().first) {
                    edges[new_edge.get_positions().second].remove(edge)
                    full_graph.children.remove(edge.group())
                    break
                }
            }
            edges[new_edge.get_positions().second].add(new_edge)
            full_graph.children.add(0,new_edge.group())
        }
    }

    /**
    * Метод для добавления новой вершины в граф.
    *
    * @param new_vertex новая вершина, которую нужно добавить в граф
    */
    fun add_vertex(new_vertex: VertexVizualisation){
        graph.create_new_vertex(new_vertex.get_name().text)
    }

```

```

        vertexes.add(new_vertex)
        edges.add(mutableListOf<Edge>())
        full_graph.children.add(new_vertex.group())
    }

/**
 * Метод для удаления вершины из графа.
 *
 * @param vertex_to_delete вершина, которую нужно удалить из графа
 */
fun delete_vertex(vertex_to_delete : VertexVizualisation) {
    for (edge in edges[vertex_to_delete.get_number()-1]) {
        if (edge.get_positions().first ==
vertex_to_delete.get_number() - 1 || edge.get_positions().second ==
vertex_to_delete.get_number() - 1 ) {
            full_graph.children.remove(edge.group())
        }
    }
    edges.removeAt(vertex_to_delete.get_number() - 1)
    for (i in (vertex_to_delete.get_number() - 1)until edges.size) {
        var size = 0
        while (size < edges[i].size) {
            val edge = edges[i][size]
            if (edge.get_positions().first ==
(vertex_to_delete.get_number() - 1)) {
                full_graph.children.remove(edge.group())
                edges[i].remove(edge)
            }
            else size++
        }
    }
    graph.delete_vertex(vertex_to_delete.get_number())
    full_graph.children.remove(vertex_to_delete.group())
    vertexes.remove(vertex_to_delete)
    for (i in vertex_to_delete.get_number() - 1 until vertexes.size)
    {
        vertexes[i].decrease_number()
    }
    for (i in 0 until edges.size){
        for(edge in edges[i]){
            if (edge.get_positions().first >
vertex_to_delete.get_number() - 1)
                edge.decrease_positions(1)
            if (edge.get_positions().second >
vertex_to_delete.get_number() - 1)
                edge.decrease_positions(2)
        }
    }
}

/**
 * Метод для удаления ребра из графа.
 *
 * @param edge ребро, которое нужно удалить из графа
 */
fun delete_edge(edge : Edge) {

```

```

graph.data[edge.get_positions().first][edge.get_positions().second] =
Int.MAX_VALUE

graph.data[edge.get_positions().second][edge.get_positions().first] =
Int.MAX_VALUE
    edges[edge.get_positions().second].remove(edge)
    full_graph.children.remove(edge.group())
}

}

```