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

ОТЧЕТ

по учебной практике

Тема: Визуализация алгоритма Краскала

Студент гр. 1384 _____ Корякин А.И.

Студентка гр. 1381 _____ Манцева Т.К.

Студент гр. 1383 _____ Малых А.А.

Руководитель _____ Токарев А.П.

Санкт-Петербург
2023

ЗАДАНИЕ

НА УЧЕБНУЮ ПРАКТИКУ

Студент Корякин А.И. группы 1384
Студентка Манцева Т.К. группы 1381
Студент Малых А.А. группы 1383

Тема практики: Визуализация алгоритма Краскала

Задание на практику:

Командная итеративная разработка приложения визуализатора алгоритма Краскала (построение минимального остовного дерева) на Kotlin с графическим интерфейсом.

Алгоритм: Алгоритм Краскала

Сроки прохождения практики: 30.06.2023 – 13.07.2023

Дата сдачи отчета: 12.07.2023

Дата защиты отчета: 12.07.2023

Студент		Корякин А.И.
Студентка		Манцева Т.К.
Студент		Малых А.А.
Руководитель		Токарев А.П.

АННОТАЦИЯ

Целью практики является получение опыта командной разработки приложения и закрепление навыков программирования на языке Kotlin. В ходе прохождения практики выполняется мини-проект, заключающийся в создании приложения на языке Kotlin с графическим интерфейсом, которое осуществляет визуализацию алгоритма Краскала (построение минимального остовного дерева). Разработка ведётся итеративно. Сперва разрабатываются две промежуточные версии приложения, затем финальная, учитывающая ошибки, допущенные при разработке первых итераций. В процессе работы используется командный репозиторий, проводится тестирование написанного кода, задачи декомпозируются и распределяются среди членов бригады.

SUMMARY

The purpose of the practice is to gain experience in team development of an application and to solidify Kotlin programming skills. During the practice, a mini-project is performed, which consists in creating a Kotlin application with a graphical interface that visualizes the Kruskal algorithm (building a minimum spanning tree). Development is carried out iteratively. At first, two intermediate versions of the application are developed, then the final one, which takes into account the errors made during the development of the first iterations. In the process of work, a team repository is used, the written code is tested, tasks are decomposed and distributed among the team members.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цель практики - получение опыта командной разработки приложения и закрепление навыков программирования на языке Kotlin. В ходе выполнения практики необходимо составить спецификацию приложения и план разработки с распределением ролей. Далее необходимо разработать прототип приложения, который демонстрирует интерфейс, но не реализует основную логику. Затем реализуется первая рабочая версия приложения, в которой должны функционировать все основные элементы. Могут иметь место мелкие недоработки, которые необходимо исправить ко второй версии. Вторая версия — финальная, учтены все недоработки и проведено тестирование.

1. ТРЕБОВАНИЯ К ПРОГРАММЕ

1.1. Исходные требования к программе

1.1.1. Требования к вводу исходных данных

Программа должна иметь возможность вводить граф из файла и внутри самого приложения.

Вводимый файл должен хранить список ребер графа. На каждой строке файла одно ребро. Ребро записывается в формате: два символа латинского алфавита (две вершины, инцидентные данному ребру) и целое число (вес ребра), разделенные пробелами.

Ввод данных внутри приложения должен осуществляться путем создания графа с помощью инструментов добавления и удаления вершин и ребер.

1.1.2. Требования к визуализации

Программа должна обладать графическим интерфейсом.

В окне приложения находится визуальное представление графа, на котором выполняется алгоритм. Каждый шаг алгоритма сопровождается текстовыми пояснениями, располагаемыми в выделенной части окна.

Ребра графа, вошедшие в МОД должны быть выделены цветом.

Графический интерфейс должен позволять пользователю:

- добавить вершины/ребра;
- удалить вершины/ребра;
- очистить экран;
- указать файл, из которого следует считать граф;
- перейти к следующему шагу алгоритма;
- перейти к предыдущему шагу алгоритма;

- перейти к последнему шагу алгоритма;
- начать визуализацию заново;

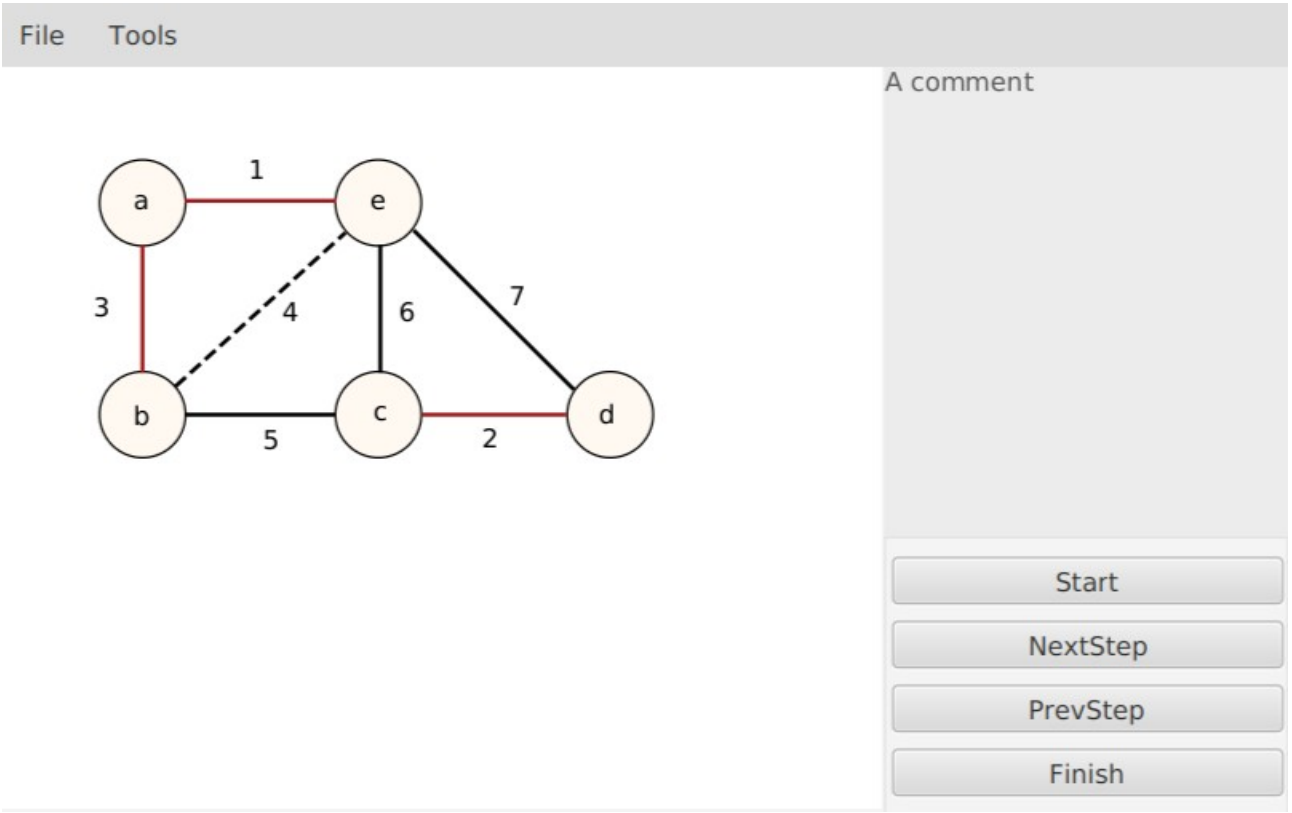

Рисунок 1 - Эскиз интерфейса

1.1.3. Требования к выходным данным

По окончании выполнения алгоритма программа должна представить измененное изображение графа. В новом представлении выделенные цветом ребра будут образовывать минимальное остовное дерево - результат работы алгоритма Краскала. Также должно быть представлено завершающее текстовое пояснение с весом МОД.

1.2. Уточнение требований после сдачи первой версии

1.2.1. Требования к вводу исходных данных

Без изменений.

1.2.2. Требования к визуализации

В дополнение к исходным требованиям.

При вводе файла необходимо обеспечить интеграцию с файловой системой. Пользователь иметь возможность выбрать файл с помощью файлового менеджера.

Требуется отображать вес ребра вместе с прямоугольным фоном, чтобы исключить возможность перекрыть надпись другим ребром, снизив тем самым её видимость.

1.2.3. Требования к выходным данным

В дополнение к исходным требованиям.

Требуется выводить сообщение с текущим весом МОД на каждом шаге алгоритма.

2. ПЛАН РАЗРАБОТКИ И РАСПРЕДЕЛЕНИЕ РОЛЕЙ В БРИГАДЕ

1. Составление спецификации. Исполнители: Корякин А.И, Манцева Т.К., Малых А.А. Срок сдачи: 04.07
2. Составление плана разработки. Исполнитель: Малых А.А. Срок сдачи: 04.07
3. Алгоритма Краскала.
    a. Реализация алгоритма. Корякин А.И. Срок сдачи 05.07
    b. Тестирование алгоритма. Исполнитель: Малых А.А. 06.07
4. Графический интерфейс.
    a. Реализация прототипа. Исполнитель: Манцева Т.К. Срок сдачи 07.07
    b. Реализация ввода графа из файла. Исполнитель: Корякин А.И. Срок сдачи 08.07
    c. Реализация рабочей версии графического интерфейса. Исполнитель: Манцева Т.К. Срок сдачи: 09.07
    d. Возможность редактирования графа внутри графического интерфейса. Исполнитель: Малых А.А. Срок сдачи 10.07
    e. Тестирование графического интерфейса. Исполнитель: Малых А.А. Срок сдачи 12.07
5. Подготовка отчёта. Исполнители: Корякин А.И., Манцева Т. К., Малых А.А. Срок сдачи: 12.07.

3. ОСОБЕННОСТИ РЕАЛИЗАЦИИ

3.1. Структуры данных

Kruskal - класс, отвечающий за логику самого алгоритма и хранения структур данных для представления графа. Также отвечает за изменение графа, которое реализуется с помощью методов добавления и удаления ребра или вершины, каждый такой метод возвращает сообщение с результатом того, что произошло(добавлена, ошибка и.т.д)

Граф хранится в классе Kruskal в виде двух структур данных.

Вершина графа представлена только своим своего именем, для которого используется тип Char. Для хранения всех вершин используется HashMap. Ключами являются имена вершин, в качестве значений используется ArrayList, в котором содержатся все смежные ей вершины.

Для представления ребер используется класс Edge. Имеет поля node1: Char, node2: Char - обозначающие инцидентные ребру вершины, weight: Int - вес вершины. Метод добавления ребер, находящийся в классе Kruskal гарантирует, что код node1 меньше кода node2. Ребра хранятся в ArrayList.

Для метки ребер в ходе алгоритма используется класс перечисление EdgeState. У ребра может быть 3 состояния: NOT_SEEN - не рассмотрена, INCLUDED - включен в МОД, DISCARDED - образовывал цикл, поэтому отброшен.

Также для работы алгоритма нужно хранить компоненты связности графа. Для этого использовался ArrayList, в котором имена всех вершин, образующих одну компоненту связности, хранятся в виде строки.

Для отображения графа в окне приложения созданы следующие классы.

Класс DrawEdge представляет изображение ребра взвешенного графа. В нём содержится поле edge: Edge - само ребро, поля с графическими объектами библиотеки JavaFX: weightText: Text - текст для веса ребра, background:

Rectangle - прямоугольный фон для текста, line: Line - линия, изображающая ребро.

Аналогичную задачу выполняет класс DrawVertex. Поля: symbol: Char - название вершины, x, y: Double - координаты вершины на рабочей области графического интерфейса, circle: Circle - изображение безымянной вершины, text: Text - текст для названия вершины.

Перечисление AlgorithmState задаёт возможные состояния алгоритма: начало, конец, выполнение, добавление вершины, удаление элемента графа.

Для обеспечения связи между графическим интерфейсом и логикой приложения создан класс KruskalWrapper. В нём имеется ссылка на граф kruskal: Kruskal для получения результатов работы алгоритма Краскала на любом шаге. Также хранится поле totalStepsNumber: Int - количество шагов, за которое был выполнен алгоритм, поле stepNumber: Int - текущий шаг алгоритма, контролируемый пользователем при взаимодействии с GUI. Отображение графа представлено двумя списками: drawableEdges: ArrayList<DrawEdge>, drawableVertices: ArrayList<DrawVertex>.

3.2. Основные методы

Класс Kruskal:

- Метод addEdge добавляет в список рёбер новое ребро. Если вершины с именем вершины инцидентной добавленному ребру нет, то создается новая вершина с таким именем. Если добавляемое ребро уже присутствует в графе, то ничего не происходит.

- Метод delEdge удаляет из списка ребер указанное ребро. Также в HashMap обе вершины, инцидентные указанному ребру, взаимно удаляются из списка смежных вершин. Если указанного ребра нет, то ничего не происходит.

- Метод addNode добавляет в HashMap новую вершину. Если вершина с таким именем уже есть, то ничего не происходит.

- Метод delNode удаляет из HashMap указанную вершину. Также вместе с этим из списка ребер удаляются все ребра инцидентные данной вершине. Соответственно, также в HashMap все вершины смежные с указанной удаляют из своих списков смежности данную. Если указанной вершины нет, то ничего не происходит.

- Метод createGraph предназначен создавать граф из данных в файле. Принимает список строк, в которых должна храниться информация о каждом ребре. Вершины определяет из данных ребер, то есть непосредственно вершины указывать не нужно. При какой-либо ошибке в данных восстанавливается граф, который был до этого.

- Метод getGraphByStep вовзвращает список ребер, который был на конкретном номере шаге.

- Метод doAlgorithm отвечает за сам алгоритм Краскала, возвращает список ребер с метками, по которым можно определить МОД.

- Метод getComp возвращает компоненту связности к которой принадлежит указанная вершина. Приватный метод, используется в doAlgorithm

- Метод uniteComps объединяет указанные компоненты связности графа. Приватный метод, используется в doAlgorithm.

- Метод isGraphConnected определяет является ли граф связным.

Использует приватный метод bfc, реализующий обход в глубину.

- Метод getWeight возвращает вес МОД на конкретном шаге алгоритма, по умолчанию возвращается вес МОД после последнего шага.

- Метод startAgain предназначен для очищения меток и структур данных, которые использовались при предыдущей работе алгоритма.

Класс KruskalWrapper:

- Метод setVerticesCoordinates добавляет на экран вершины графа и располагает их вдоль окружности. Используется при вводе графа из файла.

- Метод setEdgesCoordinates, вызывая метод setOneEdgeCoordinates для каждого ребра, добавляет их на экран согласно координатам вершин, установленными прежде методом setVerticesCoordinates.

- Метод doSteps определяет состояние алгоритма по номеру шага. Получает результат работы алгоритма на текущем шаге и меняет внешний вид ребер в соответствии с их состоянием (вошло в МОД, откинуто, не просмотрено).

- Метод findTree с помощью методов startAgain, doAlgorithm класса Kruskal заново запускает алгоритм Краскала, получает информацию о количестве шагов, создает новое изображение ребер, вызывая метод setEdgesCoordinates.

- Метод changeEdgesAfterDrag принимает вершину, положение которой в окне приложения было изменено. Меняет положение на экране инцидентных этой вершин ребер.

Класс MainController, который отвечает за управление поведением GUI:

- Метод makeDraggableVertices - определяет поведение вершин на экране при перетаскивании их с помощью мыши. Позволяет перемещать вершины по окну приложения Устанавливает границы области графического интерфейса, в которой может находиться граф.

- Метод drawGraph - отрисовывает граф. Вызывает makeDraggableVertice.

4. ТЕСТИРОВАНИЕ

4.1. Тестирование алгоритма Краскала

Для модульного тестирования кода алгоритма Краскала была использована библиотека JUnit5. Тестами были покрыты методы класса Kruskal.

Класс KruskalTest содержит несколько внутренних классов, каждому из которых соответствует набор тестовых случаев для определённого тестируемого метода. Процент покрытия кода класса Kruskal тестами составил около восьмидесяти трех процентов.

4.2. Тестирование графического интерфейса

Тестирование графического интерфейса проводилось в ручном режиме. Написаны наборы тестовых случаев, направленные на проверку:

- Корректности обработки входных данных, в том числе и невалидных.
- Отображения графа в ходе визуализации. Корректное наложение объектов друг на друга, без влияния на дальнейшее отображение и работу программы.
- Реализации операций редактирования графа. Корректность отображения после изменения графа. Соответствие данных в модели и в GUI.
- Перехода алгоритма из одного режима работы в другой. GUI должен обеспечивать блокировку некоторых операций, что не совместимы с текущим режимом работы. Также требуется обеспечивать возврат из режима с сохранением ожидаемого поведения приложения.
- Устойчивость программы к попыткам использовать уязвимости в структуре окна приложения. Например: перемещение вершины за пределы предназначенного для неё поля.

ЗАКЛЮЧЕНИЕ

В ходе прохождения практики разработано приложение на языке Kotlin с графическим интерфейсом, реализованном при помощи библиотеке JavaFX. Проведено тестирование кода алгоритма и графического интерфейса. Все недоработки были учтены и исправлены к финальной версии приложения. Получен опыт командной разработки с использованием соответствующих инструмент. Закреплены навыки программирования на языке Kotlin.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Документация JavaFX 8 // Oracle Help Center, URL:https://docs.oracle.com/javase/8/javafx/api/toc.htm (дата обращения: 12.07.2023).

2. Документация Kotlin: тестирование с JUnit // Test code using JUnit in JVM – tutorial, URL: https://kotlinlang.org/docs/jvm-test-using-junit.html (дата обращения 12.07.2023)

3. Реализация алгоритма Краскала // Kruskal's Minimum Spanning Tree (MST) Algorithm, URL: https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/ (Дата обращения 12.07.2023)

ПРИЛОЖЕНИЕ А

Исходный код

Файл src/main/kotlin/ru.etu.visualkruskal/Main.kt:

```
package ru.etu.visualkruskal

import javafx.application.Application
import javafx.fxml.FXMLLoader
import javafx.scene.Scene
import javafx.stage.Stage

class Main : Application() {
    override fun start(stage: Stage) {
        val fxmlLoader = FXMLLoader(Main::class.java.getResource("main-
view.fxml"))

        val graph = Kruskal()
        val graphWrapper  = KruskalWrapper(graph)

        val scene = Scene(fxmlLoader.load(), 800.0, 600.0)
        val controller = fxmlLoader.getController<MainController>()
        controller.setGraphWrapper(graphWrapper)

        stage.title = "Kruskal's algorithm"
        stage.scene = scene
        stage.show()
    }
}

fun main() {
    Application.launch(Main::class.java)
}
```

Файл src/main/kotlin/ru.etu.visualkruskal/Edge.kt:

```
package ru.etu.visualkruskal

class Edge(var node1: Char, var node2: Char, var weight: Int, var state:
EdgeState = EdgeState.NOT_SEEN) {
    fun printEdge() {
        if (state == EdgeState.INCLUDED)
            print("$node1 <-> $node2: weight = $weight\n")
    }

    override fun toString(): String {
        return when (state) {
            EdgeState.DISCARDED -> "Edge $node1 <-> $node2 is $
{state.toString().lowercase()} due to cycle."
            else -> "Edge $node1 <-> $node2 is $
{state.toString().lowercase()}."
        }
    }
}
```

Файл src/main/kotlin/ru.etu.visualkruskal/EdgeState.kt:

```
package ru.etu.visualkruskal

enum class EdgeState {
    NOT_SEEN, INCLUDED, DISCARDED
}
```

Файл src/main/kotlin/ru.etu.visualkruskal/Kruskal.kt:

```
package ru.etu.visualkruskal

import kotlin.collections.ArrayList

class Kruskal {
    private var edges = ArrayList<Edge>()
    private var components = ArrayList<String>()
    private var resEdges = ArrayList<Edge>()
    private var nodes = HashMap<Char, ArrayList<Char>>()

    fun addEdge(node1: Char, node2: Char, weight: Int): String{
        if(node1 in 'a'..'z' && node2 in 'a'..'z' && node1 != node2) {
            val connectedNodes1 = nodes[node1]
            val connectedNodes2 = nodes[node2]
            if(connectedNodes1 != null && connectedNodes2 != null){
                if(node1 in connectedNodes2)
                    return "Such edge already exists"
                else{
                    connectedNodes1.add(node2)
                    connectedNodes2.add(node1)
                }
            }
            if(connectedNodes1 == null && connectedNodes2 != null){
                connectedNodes2.add(node1)
                nodes[node1] = ArrayList()
                nodes[node1]!!.add(node2)
            }
            if(connectedNodes1 != null && connectedNodes2 == null){
                connectedNodes1.add(node2)
                nodes[node2] = ArrayList()
                nodes[node2]!!.add(node1)
            }
            if(connectedNodes1 == null && connectedNodes2 == null){
                nodes[node1] = ArrayList()
                nodes[node2] = ArrayList()
                nodes[node1]!!.add(node2)
                nodes[node2]!!.add(node1)
            }

            if (node1 < node2)
                edges.add(Edge(node1, node2, weight))
            else
                edges.add(Edge(node2, node1, weight))
            return "Edge added"
        }
        if(node1 == node2) return "Nodes must be different"
        return "Node name must be a latin letter"
```

```
}

fun delEdge(node1: Char, node2: Char): String{
    val tempNode1: Char
    val tempNode2: Char
    if(node1 > node2){
        tempNode1 = node2
        tempNode2 = node1
    }
    else{
        tempNode1 = node1
        tempNode2 = node2
    }
    for(edge in edges){
        if(tempNode1 == edge.node1 && tempNode2 == edge.node2){
            edges.remove(edge)
            nodes[tempNode1]!!.remove(tempNode2)
            nodes[tempNode2]!!.remove(tempNode1)
            return "Edge removed"
        }
    }
    return "There is no such edge in the graph"
}

fun addNode(nodeName: Char): String{
    if(nodeName in 'a'..'z') {
        val connectedNodes = nodes[nodeName]
        if(connectedNodes == null) {
            nodes[nodeName] = ArrayList<Char>()
        }
        else return "Such node already exists"
        return "Node added"
    }
    return "Node name must be a latin letter"
}

fun delNode(nodeName: Char): String{
    val connectedNodes = nodes[nodeName]
    if(connectedNodes != null) {
        for(node in connectedNodes) {
            nodes[node]!!.remove(nodeName)
        }

        val connectedEdges = ArrayList<Edge>()
        for (edge in edges) {
            if (edge.node1 == nodeName || edge.node2 == nodeName) {
                connectedEdges.add(edge)
            }
        }

        for (edge in connectedEdges) {
            edges.remove(edge)
        }

        nodes.remove(nodeName)
        return "Node deleted"
    }
```

```
        return "There is no such node"
    }

    fun createGraph(listOfEdges: List<String>): Boolean{
        val oldEdges = edges
        val oldNodes = nodes
        edges = ArrayList()
        nodes = HashMap()
        var tempList: List<String>
        for(edge in listOfEdges){
            tempList = edge.split(" ")
            if(tempList.size == 3) {
                val node1: Char = tempList[0].first()
                val node2: Char = tempList[1].first()
                val weight: Int = tempList[2].toInt()
                if(addEdge(node1, node2, weight) != "Edge added") {
                    clearGraph()
                    edges = oldEdges
                    nodes = oldNodes
                    return false
                }
            }
            else{
                clearGraph()
                edges = oldEdges
                nodes = oldNodes
                return false
            }
        }
        return true
    }

    fun getGraphByStep(step: Int): ArrayList<Edge>{
        val newEdges = ArrayList<Edge>()
        for(i in 0 until step){
            newEdges.add(edges[i])
        }
        for(i in step until edges.size){
            newEdges.add(Edge(edges[i].node1, edges[i].node2,
edges[i].weight, EdgeState.NOT_SEEN))
        }
        return newEdges
    }

    fun doAlgorithm(): ArrayList<Edge>{
        if(!isGraphConnected()) return edges
        edges.sortBy { it.weight }

        for(key in 'a'..'z'){
            val value = nodes[key]
            if(value != null){
                components.add(key.toString())
            }
        }

        var comp1: String
        var comp2: String
```

```
    for(i in 0..edges.size){
        comp1 = getComp(edges[i].node1)
        comp2 = getComp(edges[i].node2)
        if(comp1 != comp2 || comp1.isEmpty()) {
            edges[i].state = EdgeState.INCLUDED
            resEdges.add(edges[i])
            uniteComps(comp1, comp2)
            if(resEdges.size == nodes.size - 1)
                break
        }
        else{
            edges[i].state = EdgeState.DISCARDED
        }
    }
    return edges
}

fun getNodes(): HashMap<Char, ArrayList<Char>>{
    return nodes
}

fun getEdges(): ArrayList<Edge>{
    return edges
}

private fun getComp(node: Char): String{
    for(comp in components) {
        if (node in comp)
            return comp
    }
    return ""
}

private fun uniteComps(comp1: String, comp2: String){
    components.remove(comp2)
    components.remove(comp1)
    components.add(comp1+comp2)
}

fun clearGraph(){
    edges.clear()
    nodes.clear()
    components.clear()
    resEdges.clear()
}

fun isGraphConnected(): Boolean{
    if(nodes.size == 0) return false
    val visitedNodes = ArrayList<Char>()
    val firstNode = nodes.keys.elementAt(0)

    return bfc(firstNode, visitedNodes) == nodes.size
}

private fun bfc(node: Char, visitedNodes: ArrayList<Char>): Int{
    var countOfVisited = 1
    visitedNodes.add(node)
```

```
        val connectedNodes = nodes[node]!!
        for(tempNode in connectedNodes){
            if(tempNode !in visitedNodes)
                countOfVisited += bfc(tempNode, visitedNodes)
        }
        return countOfVisited
    }

    fun startAgain(){
        components.clear()
        resEdges.clear()
        for(edge in edges){
            edge.state = EdgeState.NOT_SEEN
        }
    }

    fun getWeight(step: Int = edges.size): Int{
        var weight = 0
        for(i in 0 until  step){
            if(edges[i].state == EdgeState.INCLUDED)
                weight += edges[i].weight
        }
        return weight
    }
}
```

Файл src/main/kotlin/ru.etu.visualkruskal/AlgorithmState.kt:

```
package ru.etu.visualkruskal

enum class AlgorithmState {
    START, IN_PROGRESS, FINISHED, ADD_VERTEX, DELETION
}
```

Файл src/main/kotlin/ru.etu.visualkruskal/DrawEdge.kt:

```
package ru.etu.visualkruskal

import javafx.scene.paint.Color
import javafx.scene.shape.Line
import javafx.scene.shape.Rectangle
import javafx.scene.text.Font
import javafx.scene.text.Text

const val weightSize = 20.0
const val symbolWidth = weightSize / 1.628
const val symbolHeight = weightSize * 1.628
const val backgroundEndWidth = 5.0 // Width of extra space on the right
part of the rectangular

const val edgeBorderWidth = 3.0
const val weightAlignment = 3.0
const val backgroundAlignmentX = 0.0
const val backgroundAlignmentY = 21.0

const val includedColour = "#42AAFF"
```

```
const val discardedDashedStyle = " -fx-stroke-dash-array: 5;"
const val discardedColour = "#000000"
const val unseenColour = "#000000"
const val weightColour = "#000000"
const val backgroundColour = "#98FB98"


class DrawEdge(private var edge: Edge) {
    private val line = Line()
    private val weightText = Text()
    private val background = Rectangle()

    init {
        line.strokeWidth = edgeBorderWidth
        line.stroke = Color.web(unseenColour)
        weightText.text = edge.weight.toString()
        weightText.font = Font(weightSize)
        weightText.fill = Color.web(weightColour)
        background.fill = Color.web(backgroundColour)
        background.height = symbolHeight
        background.width = symbolWidth * weightText.text.length +
backgroundEndWidth
    }

    fun getEdge(): Edge = edge
    fun setEdge(edge: Edge) {
        this.edge = edge
    }

    fun getWeightText(): Text = weightText
    fun getLine(): Line = line
    fun changeAppearance() {
        line.style = null
        when (edge.state) {
            EdgeState.NOT_SEEN -> line.stroke = Color.web(unseenColour)
            EdgeState.DISCARDED -> {
                line.style = discardedDashedStyle
                line.stroke = Color.web(discardedColour)
            }

            EdgeState.INCLUDED -> line.stroke = Color.web(includedColour)
        }
    }

    fun getRectangular(): Rectangle {
        return background
    }
}
```

Файл src/main/kotlin/ru.etu.visualkruskal/DrawVertex.kt:

```
package ru.etu.visualkruskal

import javafx.scene.shape.Circle
import javafx.scene.text.Font
import javafx.scene.text.Text
```

```
const val circleRadius = 23.0
const val borderColour = "#000000"
const val vertexBorderWidth = 2.0
const val circleFillColour = "#fff9f2"
const val nameAlignment = 4.0

class DrawVertex(private val symbol: Char, x: Double, y: Double) {
    private val circle = Circle()
    private val text = javafx.scene.text.Text()

    init {
        text.text = symbol.toString()
        text.font = Font.font(13.0)
        text.x = x - nameAlignment
        text.y = y + nameAlignment

        circle.centerX = x
        circle.centerY = y
        circle.radius = circleRadius
        circle.fill = javafx.scene.paint.Color.web(circleFillColour)
        circle.stroke = javafx.scene.paint.Color.web(borderColour)
        circle.strokeWidth = vertexBorderWidth

    }

    fun getName():Char = symbol
    fun getCircle():Circle = circle
    fun getText():Text = text

}
```

Файл src/main/kotlin/ru.etu.visualkruskal/KruskalWrapper.kt:

```
package ru.etu.visualkruskal

import javafx.scene.shape.Line
import kotlin.math.cos
import kotlin.math.sin

const val graphCenterX = 290.0
const val graphCenterY = 265.0
const val drawVerticesRadius = 231.0
const val deletionColour = "#A91D11"

class KruskalWrapper(private val kruskal: Kruskal, var state:
AlgorithmState = AlgorithmState.START) {
    private var stepNumber = 0
    private val drawableVertices = ArrayList<DrawVertex>()
    private val drawableEdges = ArrayList<DrawEdge>()
    private var totalStepsNumber = 0

    fun getGraphFromFile(inputList: List<String>): Boolean {
        if (kruskal.createGraph(inputList)) {
            drawableEdges.clear()
            drawableVertices.clear()
```

```
            setVerticesCoordinates()
            setEdgesCoordinates()
            return true
        }
        return false
    }

    private fun setVerticesCoordinates() {
        val nodes = kruskal.getNodes()
        val angle = 360.0 / nodes.size

        for (i in 0 until nodes.size) {
            val x = graphCenterX + drawVerticesRadius *
cos(Math.toRadians(angle * i))
            val y = graphCenterY + drawVerticesRadius *
sin(Math.toRadians(angle * i))
            drawableVertices.add(DrawVertex(nodes.keys.elementAt(i), x,
y))
        }
    }

    private fun setEdgesCoordinates() {
        val sortedEdges = kruskal.getEdges()
        sortedEdges.sortBy { it.weight }
        for (i in sortedEdges.indices) {
            drawableEdges.add(DrawEdge(sortedEdges[i]))
            setOneEdgeCoordinates(drawableEdges[i])
        }
    }

    private fun setOneEdgeCoordinates(drawEdge: DrawEdge) {
        val firstVertex = drawableVertices.find { it.getName() ==
drawEdge.getEdge().node1 }
        val secondVertex = drawableVertices.find { it.getName() ==
drawEdge.getEdge().node2 }

        val x1 = firstVertex!!.getCircle().centerX
        val y1 = firstVertex.getCircle().centerY

        val x2 = secondVertex!!.getCircle().centerX
        val y2 = secondVertex.getCircle().centerY

        changeEdgeStart(drawEdge.getLine(), x1, y1)
        changeEdgeEnd(drawEdge.getLine(), x2, y2)
        changeWeightText(drawEdge)
    }

    private fun changeEdgeStart(line: Line, x: Double, y: Double) {
        line.startX = x
        line.startY = y
    }

    private fun changeEdgeEnd(line: Line, x: Double, y: Double) {
        line.endX = x
        line.endY = y
    }
```

```
    private fun changeWeightText(edge: DrawEdge) {
        edge.getWeightText().x =
            weightAlignment + edge.getLine().startX +
((edge.getLine().endX - edge.getLine().startX) / 2)
        edge.getWeightText().y =
            -weightAlignment + edge.getLine().startY +
((edge.getLine().endY - edge.getLine().startY) / 2)
        edge.getRectangular().x = edge.getWeightText().x -
backgroundAlignmentX
        edge.getRectangular().y = edge.getWeightText().y -
backgroundAlignmentY
    }

    private fun inputEdgeValidation(inputList: ArrayList<String>):
Boolean {
        if (inputList.size != 3) return false
        val weight = inputList.last().toIntOrNull()
        return weight != null && inputList[0].length == 1 &&
inputList[1].length == 1
    }

    fun getVacantVertexOrNull(): Char? {
        for (symbol in 'a'..'z')
            if (!kruskal.getNodes().containsKey(symbol)) return symbol
        return null
    }

    fun addInputEdge(inputList: ArrayList<String>): String {
        var message = "Input data is not valid!"

        if (inputEdgeValidation(inputList)) {
            val node1 = inputList[0].first()
            val node2 = inputList[1].first()
            val weight = inputList[2].toInt()

            // Vertices must be in the graph
            if (!(node1 in kruskal.getNodes() && node2 in
kruskal.getNodes()))
                return message

            message = kruskal.addEdge(node1, node2, weight)
            if (message == "Edge added") {
                drawableEdges.clear()
                setEdgesCoordinates()
            }
        }
        return message
    }

    fun addInputVertex(newVertex: DrawVertex) {
        kruskal.addNode(newVertex.getName())
        drawableVertices.add(newVertex)
    }

    fun deleteDrawObject(edge: DrawEdge) {
        kruskal.delEdge(edge.getEdge().node1, edge.getEdge().node2)
        drawableEdges.remove(edge)
```

```
    }

    fun deleteDrawObject(vertex: DrawVertex) {
        val symbol = vertex.getName()
        kruskal.delNode(symbol)
        drawableEdges.removeAll { it.getEdge().node1 == symbol ||
it.getEdge().node2 == symbol }
        drawableVertices.remove(vertex)
    }

    fun clearDrawGraph() {
        kruskal.clearGraph()
        drawableEdges.clear()
        drawableVertices.clear()
    }

    fun changeStrokeColour() {
        if (state == AlgorithmState.DELETION) {
            drawableEdges.forEach { it.getLine().stroke =
javafx.scene.paint.Color.web(deletionColour) }
            drawableVertices.forEach { it.getCircle().stroke =
javafx.scene.paint.Color.web(deletionColour) }
        } else {
            drawableEdges.forEach { it.getLine().stroke =
javafx.scene.paint.Color.web(unseenColour) }
            drawableVertices.forEach { it.getCircle().stroke =
javafx.scene.paint.Color.web(borderColour) }
        }
    }


    private fun doSteps() {
        state = when (stepNumber) {
            0 -> AlgorithmState.START
            totalStepsNumber -> AlgorithmState.FINISHED
            else -> AlgorithmState.IN_PROGRESS
        }
        if (kruskal.getEdges().size > 0) {
            val newEdges = kruskal.getGraphByStep(stepNumber)
            for (i in drawableEdges.indices) {
                drawableEdges[i].setEdge(newEdges[i])
                drawableEdges[i].changeAppearance()
            }
        }
    }

    fun stepBack(): String {
        if (stepNumber < 1) return "Wrong step number"
        stepNumber -= 1
        doSteps()
        return if (stepNumber > 0) {
            "${drawableEdges[stepNumber - 1].getEdge()}\n" + "MST weight
is ${kruskal.getWeight(stepNumber)}."
        } else {
            "Zero step. Graph can be edited"
        }
```

```
    }

    fun stepForward(): String {
        if (stepNumber > totalStepsNumber) return "Wrong step number"
        stepNumber += 1
        doSteps()
        return if (stepNumber < totalStepsNumber) {
            "${drawableEdges[stepNumber - 1].getEdge()}\n" + "MST weight
is ${kruskal.getWeight(stepNumber)}."
        } else {
            "The last step. ${drawableEdges[stepNumber - 1].getEdge()}\n"
+ "MST weight is ${kruskal.getWeight()}."
        }
    }

    fun initialGraphState(): String {
        stepNumber = 0
        doSteps()
        return "Zero step. Graph can be edited"
    }

    fun finalGraphState(): String {
        stepNumber = totalStepsNumber
        doSteps()
        return "The last step. ${drawableEdges[stepNumber -
1].getEdge()}\n" + "MST weight is ${kruskal.getWeight()}."
    }

    fun changeEdgesAfterDrag(drawVertex: DrawVertex) {
        for (i in drawableEdges) {
            if (drawVertex.getName() == i.getEdge().node1) {
                changeEdgeStart(i.getLine(),
drawVertex.getCircle().centerX, drawVertex.getCircle().centerY)
                changeWeightText(i)
            }
            if (drawVertex.getName() == i.getEdge().node2) {
                changeEdgeEnd(i.getLine(),
drawVertex.getCircle().centerX, drawVertex.getCircle().centerY)
                changeWeightText(i)
            }
        }
    }

    fun findTree() {
        drawableEdges.clear()
        setEdgesCoordinates()
        kruskal.startAgain()
        kruskal.doAlgorithm()

        val tempEdges = kruskal.getEdges()
        for (i in tempEdges.indices) {
            if ((i == tempEdges.size - 1)) {
                totalStepsNumber = i + 1
                break
            }
            if ((tempEdges[i].state == EdgeState.INCLUDED) and
(tempEdges[i + 1].state == EdgeState.NOT_SEEN)) {
```

```
                totalStepsNumber = i + 1
                break
            }
        }

    }

    fun oneVertexCheck(): Boolean = kruskal.getNodes().size == 1
    fun getAlgState(): AlgorithmState = this.state
    fun getDrawVertices(): ArrayList<DrawVertex> = drawableVertices
    fun getDrawEdges(): ArrayList<DrawEdge> = drawableEdges
    fun getConnectivity(): Boolean = kruskal.isGraphConnected()

}
```

Файл src/main/kotlin/ru.etu.visualkruskal/MainController.kt:

```
package ru.etu.visualkruskal

import javafx.fxml.FXML
import javafx.scene.control.*
import javafx.scene.input.KeyCode
import javafx.scene.layout.GridPane
import javafx.scene.layout.Pane
import javafx.scene.text.Text
import javafx.stage.FileChooser
import java.io.File

const val dragBorder = 40

class MainController {

    @FXML
    private lateinit var graphPane: Pane

    @FXML
    private lateinit var commentText: Text

    @FXML
    private lateinit var toolsMenu: MenuButton

    @FXML
    private lateinit var startButton: Button

    @FXML
    private lateinit var nextStepButton: Button

    @FXML
    private lateinit var prevStepButton: Button

    @FXML
    private lateinit var finishButton: Button

    @FXML
    private lateinit var fileButton: Button
```

```
    private lateinit var graphWrapper: KruskalWrapper

    private var isRedacted = true

    @FXML
    private fun onFileButtonClick() {
        val fileChooser = FileChooser()

fileChooser.extensionFilters.add(FileChooser.ExtensionFilter("TEXT files
(*.txt)", "*.txt"))
        fileChooser.title = "Choose a .txt file for graph reading"
        fileChooser.initialDirectory =
File(System.getProperty("user.home"))
        val result = fileChooser.showOpenDialog(graphPane.scene.window)
        graphPane.children.clear()
        if (result != null) {
            if (result.length() != 0L && result.canRead()) {
                if (graphWrapper.getGraphFromFile(result.readLines()))
commentText.text = "Graph read from file"
                else commentText.text = "Incorrect data in file"
            } else commentText.text = "File must not be empty"
        }
        drawGraph()
    }


    @FXML
    private fun onClickAddEdgeDialog() {
        val edgeDialog: Dialog<ArrayList<String>> = Dialog()
        edgeDialog.headerText = "Enter edge. Example: a b 10"
        val dialogGrid = GridPane()
        dialogGrid.hgap = 10.0
        dialogGrid.vgap = 10.0

        val inputButtonType = ButtonType("OK",
ButtonBar.ButtonData.OK_DONE)
        edgeDialog.dialogPane.buttonTypes.addAll(inputButtonType,
ButtonType.CANCEL)

        val firstEdgeField = TextField()
        val secondEdgeField = TextField()
        val edgeWeightField = TextField()

        firstEdgeField.setMaxSize(30.0, 10.0)
        secondEdgeField.setMaxSize(30.0, 10.0)
        edgeWeightField.setMaxSize(70.0, 10.0)

        dialogGrid.add(Label("First"), 0, 0)
        dialogGrid.add(firstEdgeField, 1, 0)
        dialogGrid.add(Label("Second"), 2, 0)
        dialogGrid.add(secondEdgeField, 3, 0)
        dialogGrid.add(Label("Weight"), 4, 0)
        dialogGrid.add(edgeWeightField, 5, 0)

        edgeDialog.dialogPane.content = dialogGrid

        edgeDialog.setResultConverter { dialogButton ->
```

```
            if (dialogButton === inputButtonType) {
                return@setResultConverter arrayListOf<String>(
                    firstEdgeField.text, secondEdgeField.text,
edgeWeightField.text
                )
            }
            null
        }
        val result = edgeDialog.showAndWait()
        if (result.isPresent) {
            val arr = result.get()
            commentText.text = graphWrapper.addInputEdge(arr)
            if (commentText.text == "Edge added") {
                graphPane.children.clear()
                commentText.text = "Edge ${arr[0]} <-> ${arr[1]} added"
                drawGraph()
            }
        }
    }

    @FXML
    private fun onStartButtonClick() {
        graphPane.children.clear()
        drawGraph()
        commentText.text = graphWrapper.initialGraphState()
        changeButtonsState()
        isRedacted = true
    }

    @FXML
    private fun onStepPrevButtonClick() {
        commentText.text = graphWrapper.stepBack()
        changeButtonsState()
        if (graphWrapper.getAlgState() == AlgorithmState.START)
isRedacted = true
    }

    @FXML
    private fun onStepNextButtonClick() {
        if (isRedacted) {
            if (graphWrapper.getConnectivity() && !
graphWrapper.oneVertexCheck()) {
                graphPane.children.clear()
                graphWrapper.findTree()
                commentText.text = graphWrapper.stepForward()
                changeButtonsState()
                isRedacted = false
                drawGraph()
            } else {
                commentText.text = "Graph is disconnected or there is
only one vertex in the graph"
            }
        } else {
            commentText.text = graphWrapper.stepForward()
            changeButtonsState()
        }
    }
```

```
    @FXML
    private fun onFinishButtonClick() {
        if (isRedacted) {
            if (graphWrapper.getConnectivity() && !
graphWrapper.oneVertexCheck()) {
                graphPane.children.clear()
                graphWrapper.findTree()
                commentText.text = graphWrapper.finalGraphState()
                changeButtonsState()
                isRedacted = false
                drawGraph()
            } else {
                commentText.text = "Graph is disconnected or there is
only one vertex in the graph"
            }
        } else {
            commentText.text = graphWrapper.finalGraphState()
            changeButtonsState()
        }
    }


    @FXML
    private fun onAddVertexButtonClick() {
        val symbol = graphWrapper.getVacantVertexOrNull()
        if (symbol != null) {
            val newVertex = DrawVertex(symbol, graphCenterX,
graphCenterY)

            commentText.text = "Click to set the vertex. Press ESC to
cancel."
            graphWrapper.state = AlgorithmState.ADD_VERTEX
            graphPane.children.add(newVertex.getCircle())
            graphPane.children.add(newVertex.getText())
            changeButtonsState() // Disable buttons

            // Pressed ESC - cancel adding a vertex
            toolsMenu.setOnKeyPressed { keyEvent ->
                if (keyEvent.code == KeyCode.ESCAPE) {
                    graphWrapper.state = AlgorithmState.START
                    toolsMenu.setOnKeyPressed {}
                    graphPane.setOnMouseClicked {}
                    graphPane.setOnMouseMoved {}
                    graphPane.children.remove(newVertex.getCircle())
                    graphPane.children.remove(newVertex.getText())
                    commentText.text = "Zero step. Graph can be edited"
                    changeButtonsState()
                }
            }

            graphPane.setOnMouseMoved {
                newVertex.getCircle().centerX = it.x
                newVertex.getCircle().centerY = it.y
                newVertex.getText().x = it.x - nameAlignment
                newVertex.getText().y = it.y + nameAlignment
            }
```

```
            // Click to setting a vertex
            graphPane.setOnMouseClicked {
                graphWrapper.state = AlgorithmState.START
                graphPane.children.clear()
                toolsMenu.setOnKeyPressed {}
                graphPane.setOnMouseClicked {}
                graphPane.setOnMouseMoved {}
                graphWrapper.addInputVertex(newVertex)
                commentText.text = "The vertex $symbol has been set"
                changeButtonsState()
                drawGraph()
            }

        } else commentText.text = "The maximum number of vertices has
been reached"
    }

    private fun deleteHandlerReset() {
        graphWrapper.state = AlgorithmState.START
        graphWrapper.changeStrokeColour() // Restore stroke colour
        toolsMenu.setOnKeyPressed {}
        graphWrapper.getDrawEdges().forEach
{ it.getLine().setOnMouseClicked {} }
        graphWrapper.getDrawVertices().forEach
{ it.getCircle().setOnMouseClicked {} }
        graphPane.children.clear()
        changeButtonsState()
        drawGraph()
    }

    @FXML
    private fun onDeleteButtonClick() {
        commentText.text = "Click on a vertex or edge to delete. Press
ESC to cancel."
        graphWrapper.state = AlgorithmState.DELETION
        graphWrapper.changeStrokeColour() // Set deletion colour
        changeButtonsState()              // Disable buttons

        // Pressed ESC - cancel deletion
        toolsMenu.setOnKeyPressed { keyEvent ->
            if (keyEvent.code == KeyCode.ESCAPE) {
                commentText.text = "Zero step. Graph can be edited"
                deleteHandlerReset()
                changeButtonsState()
            }
        }

        // Deletion for edges
        graphWrapper.getDrawEdges().forEach { edge ->
            edge.getLine().setOnMouseClicked {
                commentText.text = "Edge ${edge.getEdge().node1} <-> $
{edge.getEdge().node2} has been deleted"
                graphWrapper.deleteDrawObject(edge)
                deleteHandlerReset()
            }
        }
```

```
        // Deletion for vertices
        graphWrapper.getDrawVertices().forEach { vertex ->
            vertex.getCircle().setOnMouseClicked {
                commentText.text = "Vertex ${vertex.getName()} has been
deleted"
                graphWrapper.deleteDrawObject(vertex)
                deleteHandlerReset()
            }
        }
    }

    @FXML
    private fun onClickClearGraph() {
        graphPane.children.clear()
        graphWrapper.clearDrawGraph()
    }

    fun setGraphWrapper(graphWrapper: KruskalWrapper) {
        this.graphWrapper = graphWrapper
    }

    private fun drawGraph() {
        graphWrapper.getDrawEdges().forEach
{ graphPane.children.add(it.getLine()) }
        graphWrapper.getDrawEdges().forEach
{ graphPane.children.add(it.getRectangular()) }
        graphWrapper.getDrawEdges().forEach
{ graphPane.children.add(it.getWeightText()) }
        graphWrapper.getDrawVertices().forEach
{ graphPane.children.add(it.getCircle()) }
        graphWrapper.getDrawVertices().forEach
{ graphPane.children.add(it.getText()) }
        makeDraggableVertices()
    }

    private fun changeButtonsState() {
        when (graphWrapper.getAlgState()) {
            AlgorithmState.START -> {
                startButton.isDisable = false
                nextStepButton.isDisable = false
                prevStepButton.isDisable = true
                finishButton.isDisable = false
                toolsMenu.isDisable = false
                fileButton.isDisable = false
                toolsMenu.items.forEach { it.isDisable = false }
            }

            AlgorithmState.IN_PROGRESS -> {
                nextStepButton.isDisable = false
                prevStepButton.isDisable = false
                finishButton.isDisable = false
                toolsMenu.isDisable = true
                fileButton.isDisable = true
            }

            AlgorithmState.FINISHED -> {
```

```
                nextStepButton.isDisable = true
                prevStepButton.isDisable = false
                finishButton.isDisable = true
                toolsMenu.isDisable = true
                fileButton.isDisable = true
            }

            AlgorithmState.ADD_VERTEX, AlgorithmState.DELETION -> {
                startButton.isDisable = true
                nextStepButton.isDisable = true
                prevStepButton.isDisable = true
                finishButton.isDisable = true
                fileButton.isDisable = true
                toolsMenu.items.forEach { it.isDisable = true }
            }
        }
    }

    private fun makeDraggableVertices() {
        for (i in graphWrapper.getDrawVertices()) {
            i.getCircle().setOnMouseDragged {
                if ((it.x < graphPane.width - dragBorder) && (it.x >
dragBorder / 2) && (it.y < graphPane.height - dragBorder) && (it.y >
dragBorder)) {
                    i.getCircle().centerX = it.x
                    i.getCircle().centerY = it.y
                    i.getText().x = i.getCircle().centerX - nameAlignment
                    i.getText().y = i.getCircle().centerY + nameAlignment
                    graphWrapper.changeEdgesAfterDrag(i)
                    i.getCircle().toFront()
                    i.getText().toFront()
                }
            }
        }
    }

}
```

Файл src/test/kotlin/ru.etu.visualkruskal/KruskalTest.kt:

```
package ru.etu.visualkruskal
import org.junit.jupiter.api.Assertions.*
import org.junit.jupiter.api.BeforeEach
import org.junit.jupiter.api.Nested
import org.junit.jupiter.api.Test

internal class KruskalTest {

    @Nested
    inner class AddEdgeTest {
        private lateinit var solver: Kruskal

        @BeforeEach
        fun setUp() {
            solver = Kruskal()
```

```
}

@Test
fun `Regular case`() {
    // Act
    val res: String = solver.addEdge('a', 'z', 23)
    val edges: List<Edge> = solver.getEdges()
    val nodes = solver.getNodes()

    // message check
    assertEquals("Edge added", res)

    // nodes-map check
    assertEquals(2, nodes.size)
    assertArrayEquals(arrayOf('z'), nodes['a']!!.toArray())
    assertArrayEquals(arrayOf('a'), nodes['z']!!.toArray())

    // Edges list check
    assertEquals(1, edges.size)
    assertEquals(23, edges.first().weight)
    assertEquals('a', edges.first().node1)
    assertEquals('z', edges.first().node2)
    assertEquals(EdgeState.NOT_SEEN, edges.first().state)
}

@Test
fun `Reverse order of symbols`() {
    // Act
    val res: String = solver.addEdge('v', 'f', -3)
    val edges: List<Edge> = solver.getEdges()
    val nodes = solver.getNodes()

    // message check
    assertEquals("Edge added", res)

    // nodes-map check
    assertEquals(2, nodes.size)
    assertArrayEquals(arrayOf('v'), nodes['f']!!.toArray())
    assertArrayEquals(arrayOf('f'), nodes['v']!!.toArray())

    // Edges list check
    assertEquals(1, edges.size)
    assertEquals(-3, edges.first().weight)
    assertEquals('f', edges.first().node1)
    assertEquals('v', edges.first().node2)
    assertEquals(EdgeState.NOT_SEEN, edges.first().state)
}

@Test
fun `Same symbols`() {
    // Act
    val res: String = solver.addEdge('d', 'd', 0)
    val edges: List<Edge> = solver.getEdges()
    val nodes = solver.getNodes()

    // message check
    assertEquals("Nodes must be different", res)
```

```
        // nodes-map check
        assertTrue(nodes.isEmpty())

        // Edges list check
        assertTrue(edges.isEmpty())
    }

    @Test
    fun `Incorrect symbols`() {
        // Act
        val res: String = solver.addEdge('A', 'd', 0)
        val edges: List<Edge> = solver.getEdges()
        val nodes = solver.getNodes()

        // message check
        assertEquals("Node name must be a latin letter", res)

        // nodes-map check
        assertTrue(nodes.isEmpty())

        // Edges list check
        assertTrue(edges.isEmpty())
    }

    @Test
    fun `Duplicate edge`() {
        // Arrange
        solver.addEdge('f', 'd', 3)

        // Act
        val res: String = solver.addEdge('d', 'f', 55)
        val edges: List<Edge> = solver.getEdges()
        val nodes = solver.getNodes()

        // message check
        assertEquals("Such edge already exists", res)

        // nodes-map check
        assertEquals(2, nodes.size)
        assertArrayEquals(arrayOf('f'), nodes['d']!!.toArray())
        assertArrayEquals(arrayOf('d'), nodes['f']!!.toArray())

        // Edges list check
        assertEquals(1, edges.size)
        assertEquals(3, edges.first().weight)
        assertEquals('d', edges.first().node1)
        assertEquals('f', edges.first().node2)
        assertEquals(EdgeState.NOT_SEEN, edges.first().state)

    }

    @Test
    fun `Two edges with the same vertex `() {
        // Arrange
        solver.addEdge('a', 'f', 3)
```

```
        // Act
        val res: String = solver.addEdge('f', 'b', 55)
        val edges: List<Edge> = solver.getEdges()
        val nodes = solver.getNodes()

        // message check
        assertEquals("Edge added", res)

        // nodes-map check
        assertEquals(3, nodes.size)
        assertArrayEquals(arrayOf('a', 'b'), nodes['f']!!.toArray())
        assertArrayEquals(arrayOf('f'), nodes['a']!!.toArray())
        assertArrayEquals(arrayOf('f'), nodes['b']!!.toArray())

        // Edges list check
        assertEquals(2, edges.size)

        assertEquals(3, edges.first().weight)
        assertEquals('a', edges.first().node1)
        assertEquals('f', edges.first().node2)
        assertEquals(EdgeState.NOT_SEEN, edges.first().state)

        assertEquals(55, edges.last().weight)
        assertEquals('b', edges.last().node1)
        assertEquals('f', edges.last().node2)
        assertEquals(EdgeState.NOT_SEEN, edges.last().state)

    }
}

@Nested
inner class DeleteEdgeTest {
    private lateinit var solver: Kruskal

    @BeforeEach
    fun setUp() {
        solver = Kruskal()
    }

    @Test
    fun `One edge`() {
        // Arrange
        solver.addEdge('a', 'b', 43)

        // Act
        val res: String = solver.delEdge('a', 'b')
        val edges: List<Edge> = solver.getEdges()
        val nodes = solver.getNodes()

        // message check
        assertEquals("Edge removed", res)

        // nodes-map check
        assertEquals(2, nodes.size)
        assertTrue(nodes['a']!!.isEmpty())
        assertTrue(nodes['b']!!.isEmpty())
```

```
        // Edges list check
        assertTrue(edges.isEmpty())
    }

    @Test
    fun `Path of vertex`() {
        // Arrange
        solver.addEdge('a', 'b', -5)
        solver.addEdge('b', 'd', 1)
        solver.addEdge('a', 'c', 4)

        // Act
        val res: String = solver.delEdge('a', 'b')
        val edges: List<Edge> = solver.getEdges()
        val nodes = solver.getNodes()

        // message check
        assertEquals("Edge removed", res)

        // nodes-map check
        assertEquals(4, nodes.size)
        assertArrayEquals(arrayOf('c'), nodes['a']!!.toArray())
        assertArrayEquals(arrayOf('d'), nodes['b']!!.toArray())
        assertArrayEquals(arrayOf('b'), nodes['d']!!.toArray())
        assertArrayEquals(arrayOf('a'), nodes['c']!!.toArray())

        // Edges list check
        assertEquals(2, edges.size)

    }

    @Test
    fun `Reverse order of symbols`() {
        // Arrange
        solver.addEdge('a', 'b', -5)
        solver.addEdge('b', 'd', 1)
        solver.addEdge('a', 'c', 4)

        // Act
        val res: String = solver.delEdge('b', 'a')
        val edges: List<Edge> = solver.getEdges()
        val nodes = solver.getNodes()

        // message check
        assertEquals("Edge removed", res)

        // nodes-map check
        assertEquals(4, nodes.size)
        assertArrayEquals(arrayOf('c'), nodes['a']!!.toArray())
        assertArrayEquals(arrayOf('d'), nodes['b']!!.toArray())
        assertArrayEquals(arrayOf('d'), nodes['b']!!.toArray())
        assertArrayEquals(arrayOf('c'), nodes['a']!!.toArray())

        // Edges list check
        assertEquals(2, edges.size)
    }
```

```
    @Test
    fun `From the empty edges-list`() {
        // Act
        val res: String = solver.delEdge('b', 'a')
        val edges: List<Edge> = solver.getEdges()
        val nodes = solver.getNodes()

        // message check
        assertEquals("There is no such edge in the graph", res)

        // nodes-map check
        assertTrue(nodes.isEmpty())

        // Edges list check
        assertTrue(edges.isEmpty())
    }

    @Test
    fun `Non-existent edge`() {
        // Arrange
        solver.addEdge('a', 'b', -5)
        solver.addEdge('b', 'd', 1)
        solver.addEdge('a', 'c', 4)

        // Act
        val res: String = solver.delEdge('a', 'd')
        val edges: List<Edge> = solver.getEdges()
        val nodes = solver.getNodes()

        // message check
        assertEquals("There is no such edge in the graph", res)

        // nodes-map check
        assertEquals(4, nodes.size)
        assertArrayEquals(arrayOf('b', 'c'), nodes['a']!!.toArray())
        assertArrayEquals(arrayOf('a', 'd'), nodes['b']!!.toArray())
        assertArrayEquals(arrayOf('b'), nodes['d']!!.toArray())
        assertArrayEquals(arrayOf('a'), nodes['c']!!.toArray())

        // Edges list check
        assertEquals(3, edges.size)
    }
}

@Nested
inner class AddNodeTest {
    private lateinit var solver: Kruskal

    @BeforeEach
    fun setUp() {
        solver = Kruskal()
    }

    @Test
    fun `Regular case`() {
        // Arrange
```

```
    solver.addNode('b')

    // Act
    val res: String = solver.addNode('a')
    val edges: List<Edge> = solver.getEdges()
    val nodes = solver.getNodes()

    // message check
    assertEquals("Node added", res)

    // nodes-map check
    assertEquals(2, nodes.size)
    assertTrue(nodes['a']!!.isEmpty())
    assertTrue(nodes['b']!!.isEmpty())

    // Edges list check
    assertTrue(edges.isEmpty())
}

@Test
fun `First node`() {
    // Act
    val res: String = solver.addNode('a')
    val edges: List<Edge> = solver.getEdges()
    val nodes = solver.getNodes()

    // message check
    assertEquals("Node added", res)

    // nodes-map check
    assertEquals(1, nodes.size)
    assertTrue(nodes['a']!!.isEmpty())

    // Edges list check
    assertTrue(edges.isEmpty())
}


@Test
fun `Duplicate node`() {
    // Arrange
    solver.addNode('a')
    solver.addNode('b')

    // Act
    val res: String = solver.addNode('a')
    val edges: List<Edge> = solver.getEdges()
    val nodes = solver.getNodes()

    // message check
    assertEquals("Such node already exists", res)

    // nodes-map check
    assertEquals(2, nodes.size)
    assertTrue(nodes['a']!!.isEmpty())
    assertTrue(nodes['b']!!.isEmpty())
```

```
        // Edges list check
        assertTrue(edges.isEmpty())
    }

    @Test
    fun `Incorrect symbol`() {
        // Arrange
        solver.addNode('a')
        solver.addNode('b')

        // Act
        val res: String = solver.addNode('F')
        val edges: List<Edge> = solver.getEdges()
        val nodes = solver.getNodes()

        // message check
        assertEquals("Node name must be a latin letter", res)

        // nodes-map check
        assertEquals(2, nodes.size)
        assertTrue(nodes['a']!!.isEmpty())
        assertTrue(nodes['b']!!.isEmpty())

        // Edges list check
        assertTrue(edges.isEmpty())
    }
}

@Nested
inner class DeleteNodeTest {
    private lateinit var solver: Kruskal

    @BeforeEach
    fun setUp() {
        solver = Kruskal()
    }

    @Test
    fun `Isolated nodes`() {
        // Arrange
        solver.addNode('a')
        solver.addNode('b')
        solver.addNode('c')

        // Act
        val res: String = solver.delNode('b')
        val edges: List<Edge> = solver.getEdges()
        val nodes = solver.getNodes()

        // message check
        assertEquals("Node deleted", res)

        // nodes-map check
        assertEquals(2, nodes.size)
        assertTrue(nodes['a']!!.isEmpty())
        assertTrue(nodes['c']!!.isEmpty())
```

```
    // Edges list check
    assertTrue(edges.isEmpty())
}


@Test
fun `One node`() {
    // Arrange
    solver.addNode('a')

    // Act
    val res: String = solver.delNode('a')
    val edges: List<Edge> = solver.getEdges()
    val nodes = solver.getNodes()

    // message check
    assertEquals("Node deleted", res)

    // nodes-map check
    assertTrue(nodes.isEmpty())

    // Edges list check
    assertTrue(edges.isEmpty())
}

@Test
fun `Incorrect symbol`() {
    // Arrange
    solver.addNode('a')
    solver.addNode('b')
    solver.addNode('c')

    // Act
    val res: String = solver.delNode('Q')
    val edges: List<Edge> = solver.getEdges()
    val nodes = solver.getNodes()

    // message check
    assertEquals("There is no such node", res)

    // nodes-map check
    assertEquals(3, nodes.size)
    assertTrue(nodes['a']!!.isEmpty())
    assertTrue(nodes['b']!!.isEmpty())
    assertTrue(nodes['c']!!.isEmpty())

    // Edges list check
    assertTrue(edges.isEmpty())
}

@Test
fun `Non-existent node`() {
    // Arrange
    solver.addNode('a')
    solver.addNode('b')
    solver.addNode('c')
```

```
        // Act
        val res: String = solver.delNode('g')
        val edges: List<Edge> = solver.getEdges()
        val nodes = solver.getNodes()

        // message check
        assertEquals("There is no such node", res)

        // nodes-map check
        assertEquals(3, nodes.size)
        assertTrue(nodes['a']!!.isEmpty())
        assertTrue(nodes['b']!!.isEmpty())
        assertTrue(nodes['c']!!.isEmpty())

        // Edges list check
        assertTrue(edges.isEmpty())
    }

    @Test
    fun `Terminal node`() {
        // Arrange
        solver.addEdge('a', 'b', 54)
        solver.addEdge('b', 'c', 12)

        // Act
        val res: String = solver.delNode('a')
        val edges: List<Edge> = solver.getEdges()
        val nodes = solver.getNodes()

        // message check
        assertEquals("Node deleted", res)

        // nodes-map check
        assertEquals(2, nodes.size)
        assertArrayEquals(arrayOf('c'), nodes['b']!!.toArray())
        assertArrayEquals(arrayOf('b'), nodes['c']!!.toArray())

        // Edges list check
        assertEquals(1, edges.size)
        assertEquals('b', edges.first().node1)
        assertEquals('c', edges.first().node2)
        assertEquals(12, edges.first().weight)
    }

    @Test
    fun `Two adjacent vertices`() {
        // Arrange
        solver.addEdge('a', 'b', 54)
        solver.addEdge('b', 'c', 12)

        // Act
        val res: String = solver.delNode('b')
        val edges: List<Edge> = solver.getEdges()
        val nodes = solver.getNodes()

        // message check
        assertEquals("Node deleted", res)
```

```
        // nodes-map check
        assertEquals(2, nodes.size)
        assertTrue(nodes['a']!!.isEmpty())
        assertTrue(nodes['c']!!.isEmpty())

        // Edges list check
        assertTrue(edges.isEmpty())
    }
}

@Nested
inner class IsGraphConnectedTest {
    private lateinit var solver: Kruskal

    @BeforeEach
    fun setUp() {
        solver = Kruskal()
    }

    @Test
    fun `One node`() {
        // Arrange
        solver.addNode('a')

        // Act
        val res: Boolean = solver.isGraphConnected()

        // Assert
        assertTrue(res)
    }

    @Test
    fun `One edge`() {
        // Arrange
        solver.addEdge('a', 'b', 3)

        // Act
        val res: Boolean = solver.isGraphConnected()

        // Assert
        assertTrue(res)
    }

    @Test
    fun `Isolated nodes`() {
        // Arrange
        solver.addNode('a')
        solver.addNode('b')
        solver.addNode('c')

        // Act
        val res: Boolean = solver.isGraphConnected()

        // Assert
        assertFalse(res)
    }
```

```
    @Test
    fun `Ordinary connected graph `() {
        // Arrange
        solver.addEdge('a', 'b', 4)
        solver.addEdge('b', 'c', -54)
        solver.addEdge('c', 'd', 5)
        solver.addEdge('d', 'b', 43)
        solver.addEdge('a', 'f', 1)
        solver.addEdge('f', 'v', 4)

        // Act
        val res: Boolean = solver.isGraphConnected()

        // Assert
        assertTrue(res)
    }

    @Test
    fun `Ordinary disconnected graph `() {
        // Arrange
        solver.addEdge('a', 'b', 4)
        solver.addEdge('b', 'c', -54)
        solver.addEdge('c', 'd', 5)
        solver.addEdge('d', 'b', 43)
        solver.addEdge('a', 'f', 1)
        solver.addEdge('f', 'v', 4)
        solver.addEdge('y', 'z', 4)
        solver.addEdge('z', 'l', 4)
        solver.addEdge('l', 'm', 987)

        // Act
        val res: Boolean = solver.isGraphConnected()

        // Assert
        assertFalse(res)
    }
}

@Nested
inner class DoAlgorithmTest {
    private lateinit var solver: Kruskal

    @BeforeEach
    fun setUp() {
        solver = Kruskal()
    }

    @Test
    fun `One edge`() {
        // Arrange
        solver.addEdge('a', 'b', 100)

        // Act
        val res: ArrayList<Edge> = solver.doAlgorithm()

        // Assert
```

```
            assertEquals(EdgeState.INCLUDED, res.first().state)
            assertEquals(100, solver.getWeight())
        }

        @Test
        fun `Tree with repeated weights`() {
            // Arrange
            val inputEdge: List<String> = arrayListOf(
                "a b 5", "a d 1", "d e 1", "d f 3", "d g 2", "a c 2", "c
i 2", "c h 4"
            )
            solver.createGraph(inputEdge)

            // Act
            val res: ArrayList<Edge> = solver.doAlgorithm()

            // Assert
            res.forEach { assertEquals(EdgeState.INCLUDED, it.state) }
            assertEquals(20, solver.getWeight())
        }

        @Test
        fun `Joining two large connected components`() {
            // Arrange
            val inputEdge: List<String> = arrayListOf(
                "a b 1", "a f 3", "b c 2", "c d 5", "d e -4", "e f 1"
            )
            val expectedStates = arrayListOf(
                EdgeState.INCLUDED,
                EdgeState.INCLUDED,
                EdgeState.INCLUDED,
                EdgeState.INCLUDED,
                EdgeState.INCLUDED,
                EdgeState.NOT_SEEN
            )

            solver.createGraph(inputEdge)

            // Act
            val res: ArrayList<Edge> = solver.doAlgorithm()

            // Assert
            assertEquals(3, solver.getWeight())
            for (i in expectedStates.indices)
assertEquals(expectedStates[i], res[i].state)

        }

        @Test
        fun `Check skipped edges`() {
            // Arrange
            val inputEdge: List<String> = arrayListOf(
                "d e 7", "b e 6", "b d 5", "c d 4", "b c 3", "a c 2", "a
b 1"
            )
            val expectedStates = arrayListOf(
                EdgeState.INCLUDED,
```

```
                EdgeState.INCLUDED,
                EdgeState.DISCARDED,
                EdgeState.INCLUDED,
                EdgeState.DISCARDED,
                EdgeState.INCLUDED,
                EdgeState.NOT_SEEN
            )

            solver.createGraph(inputEdge)

            // Act
            val res: ArrayList<Edge> = solver.doAlgorithm()

            // Assert
            assertEquals(13, solver.getWeight())
            for (i in expectedStates.indices)
assertEquals(expectedStates[i], res[i].state)

        }

        @Test
        fun `Disconnected graph`() {
            // Arrange
            val inputEdge: List<String> = arrayListOf(
                "a b 2", "b c 20", "b d 100", "e f -70"
            )

            solver.createGraph(inputEdge)

            // Act
            val res: ArrayList<Edge> = solver.doAlgorithm()

            // Assert
            assertEquals(0, solver.getWeight())
            res.forEach { assertEquals(EdgeState.NOT_SEEN, it.state) }

        }

    }


}
```