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

ОТЧЕТ

по учебной практике

Тема: Командная визуализация алгоритма Борувки

Студент гр. 8382 _____ Мирончик П.Д.

Студент гр. 8382 _____ Синельников М.Р.

Студент гр. 8382 _____ Янкин Д.О.

Руководитель _____ Ефремов М.А.

Санкт-Петербург

2020

ЗАДАНИЕ
НА УЧЕБНУЮ ПРАКТИКУ

Студент Мирончик П.Д. группы 8382

Студент Синельников М.Р. группы 8382

Студент Янкин Д.О. группы 8382

Тема практики: Командная визуализация алгоритма Борувки

Задание на практику:

Командная итеративная разработка визуализатора алгоритма на Kotlin с графическим интерфейсом.

Алгоритм: Борувки.

Сроки прохождения практики: 29.06.2020 – 12.07.2020

Дата сдачи отчета: 10.07.2020

Дата защиты отчета: 10.07.2020

Студент	_____	Мирончик П.Д.
Студент	_____	Синельников М.Р.
Студент	_____	Янкин Д.О.
Руководитель	_____	Ефремов М.А.

АННОТАЦИЯ

Целью учебной практики является разработка приложения для визуализации алгоритма Борувки. Пользователю предоставляются инструменты для рисования графа и возможность пошагового просмотра работы реализуемого алгоритма. Приложение должно обладать доступным и удобным интерфейсом.

Разработка ведется на языке Kotlin с использованием JavaFX для создания графического интерфейса. Разработка ведется командой из трех человек, за которыми закреплены определенные роли. Выполнение работы осуществляется поэтапно.

SUMMARY

Academia practice aim is to develop application for visualizing Boruvka’s algorithm. The user is provided with tools of drawing graph and has opportunity to see all steps of realized algorithm. The application should provide accessible and intuitive interface.

Development is done on Kotlin programming language using JavaFX framework for creating graphical interface. Development team consists of three people, each of them responsible for certain task. Development is done step by step.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Целью учебной практики является реализация приложения для визуализации алгоритма Борувки для нахождения минимального остовного дерева в графе. Пользователю должна быть предоставлена возможность задания графа посредством взаимодействия с графическими элементами. Результат работы алгоритма должен иметь графическое отображение. Должна быть предоставлена возможность просмотра итогового результата алгоритма и просмотра хода его исполнения по шагам.

Работа выполняется командой из трех человек, за каждым из которых закреплены ответственности: за разработку графического интерфейса, за разработку логики алгоритма, за проведение тестирования и сборку проекта.

Необходимо провести модульное тестирование программы. Разработанная программа должна собираться в jar-архив.

1. ТРЕБОВАНИЯ К ПРОГРАММЕ

1.1. Исходные Требования к программе

1.1.1. Требования к вводу исходных данных

Для задания графа должны быть реализованы несколько возможностей: задание графа рисованием посредством взаимодействия с графическими элементами: интерактивным добавлением вершин, ребер, назначением весов; загрузка графа из файла с описанием вершин и ребер; случайная генерация графа по параметрам, задаваемых пользователем: количество вершин, количество ребер.

1.1.2. Требования к визуализации

Пользователю должно быть доступно графическое изображение графа, интерактивное взаимодействие с графом (перемещение, добавление или удаление элементов), выполнение алгоритма по команде, просмотр состояний графа на каждом шаге алгоритма и просмотр итогового результата.

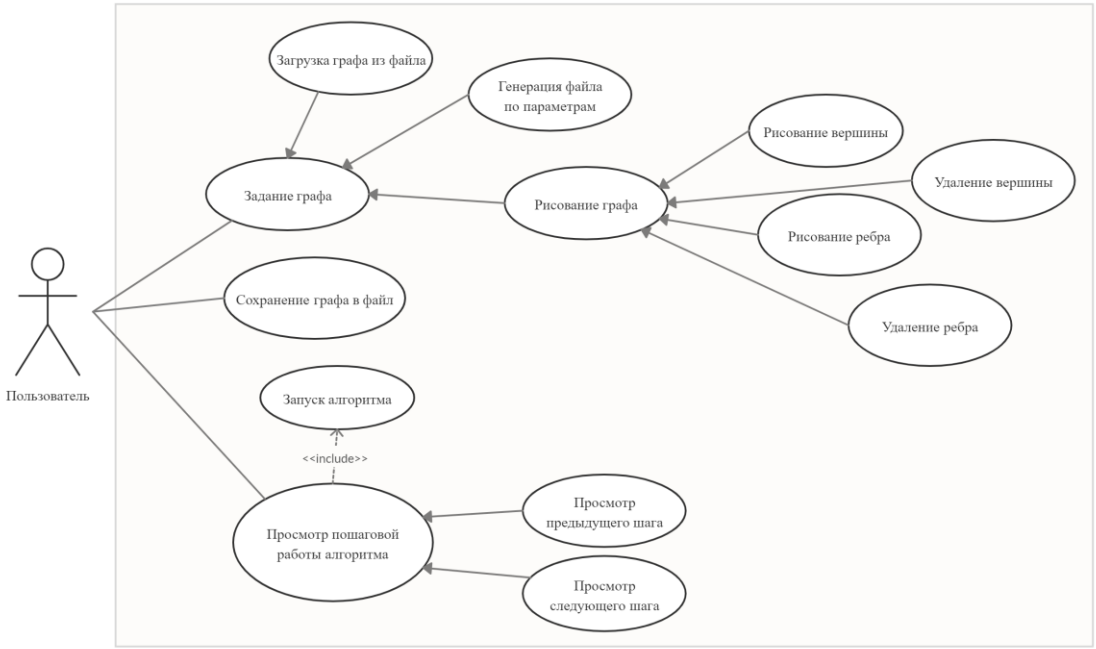

Рисунок 1. Диаграмма прецедентов разрабатываемой программы

2. ПЛАН РАЗРАБОТКИ И РАСПРЕДЕЛЕНИЕ РОЛЕЙ В БРИГАДЕ

2.1. План разработки

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

К 04.07.2020 должны быть размещены все элементы интерфейса с использованием заглушек, составлена UML-диаграмма классов программы с пояснением принятых решений.

К 06.07.2020 необходимо реализовать случайную генерацию изначальных графов по параметрам, задаваемых пользователем, с проверкой корректности входных данных; запуск алгоритма из графического интерфейса с отображением конечного результата работы алгоритма; добавить в отчет описание алгоритма, составить UML-диаграмму состояний и план тестирования.

К 08.07.2020 должна быть добавлена возможность визуализации как пошагового выполнения алгоритма, так и просмотра итогового результата; должны быть проведены тесты для созданных структур данных и функций алгоритма согласно плану тестирования; в отчет добавлено описание алгоритма пошагового отображения работы алгоритма.

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

2.2. Распределение ролей в бригаде

Мирончик П.Д. отвечает за разработку графического интерфейса.

Янкин Д.О. отвечает за реализацию логики алгоритма.

Синельников М.Р. отвечает за тестирование и сборку приложения.

3. ОСОБЕННОСТИ РЕАЛИЗАЦИИ

3.1. Описание структуры приложения

Корневым рабочим классом является **TrioWorkspace**. В нем расположены основные элементы интерфейса: **AppBar** (верхняя панель с кнопками добавления вершин и ребер) и **TreeView**. **TrioWorkspace** также создает модель дерева, которое будет использовано в приложении, и **TreeViewController**, после чего проводит инициализацию **TreeView**, добавляя в него ссылку на контроллер и модель дерева.

TreeView – виджет, отвечающий за отображение графа. К данному элементу в инициализированном состоянии привязаны **TreeViewController** и **TreeModel**. При инициализации **TreeView** подписывается на изменения привязанной **TreeModel** (добавления/удаления вершин и ребер из дерева) с учетом уже добавленных вершин и ребер.

При получении события добавления вершины **TreeView** создает **NodeView**, привязывая к нему добавляемую вершину **TreeModel**, и добавляет созданный **NodeView** в список дочерних виджетов, после чего **NodeView** отображается на экране. При удалении вершины **NodeView** отвязывается от **NodeModel** и удаляется из родительского **виджета**.

Работа с ребрами ведется несколько иначе. Поскольку ребро это прямая линия, необходимо решить проблему создания нескольких ребер между одинаковыми вершинами. Для этого создан виджет **EdgesView**. В отличие от **NodeView**, являющимся **One2One** структурой, **EdgesView** – **One2Many**. Это означает, что к одному виджету может быть привязано несколько моделей **EdgeModel**. Так при добавлении ребра **edge** в модель производятся следующие операции:

1. Производится поиск виджета ребра между вершинами **EdgeModel.firstNode** и **EdgeModel.secondModel**. Если такой виджет не найден, создается новый **EdgesView**.

2. К полученному на 1 шаге виджету привязывается **edge**.

Для пользователя несколько ребер между вершинами выглядят как линия между вершинами, в центре которой находятся несколько прямоугольников с весами соответствующих привязанных ребер.

Рис. 2. Отображение нескольких ребер между одинаковыми вершинами

Управление приложением производится при помощи класса **TreeViewController**. Этот класс агрегирует все события: нажатия мыши, клавиатуры, и, основываясь на них меняет состояние приложения. **TreeViewController** реализован в виде машины состояний. Так, начальному состоянию соответствует класс **StateIdle**, состоянию, когда нажата кнопка “Добавить вершину” – **StateAddNode** и т.д.

TreeViewController не обрабатывает действия пользователя своими силами, он лишь делегирует события текущему состоянию. Такая реализация позволяет избавиться от большого количества проверок: например, при нажатии на поле (**TreeView**) возможны разные действия – добавление вершины, выделение вершины, на которую произошло нажатие, очистка списка выделенных вершин, если нажатие произведено на точку, которая не покрывается вершиной, и т.д.

После идентификации действия состояние **TreeViewController**-а зачастую выполняет определенные операции. Это может быть выделение вершины или ребра (для этого в **TreeViewController** хранится набор выделенных вершин и ребер), редактирование значения вершины или ребра (для этого используется

класс **SimpleEditor**), удаления выделенных вершин и ребер (для этого используется ссылка tree соответствующего объекта **TreeViewController**).

TreeViewController не хранит в себе информацию о виджетах (кроме ссылки на **TreeView**) и не влияет на состояние виджетов напрямую (кроме задания значений вершин и ребер при редактировании а также задания координат вершины при добавлении ее в дерево), однако хранит вспомогательную информацию, такую как выделенные вершины, активность кнопок. Виджеты, которые должны менять свое состояние в зависимости от этих параметров (например выделенные ребро и вершина отличаются от невыделенных цветом), подписываются на изменения параметров **TreeViewController**-а и определяют особенности отображения самостоятельно.

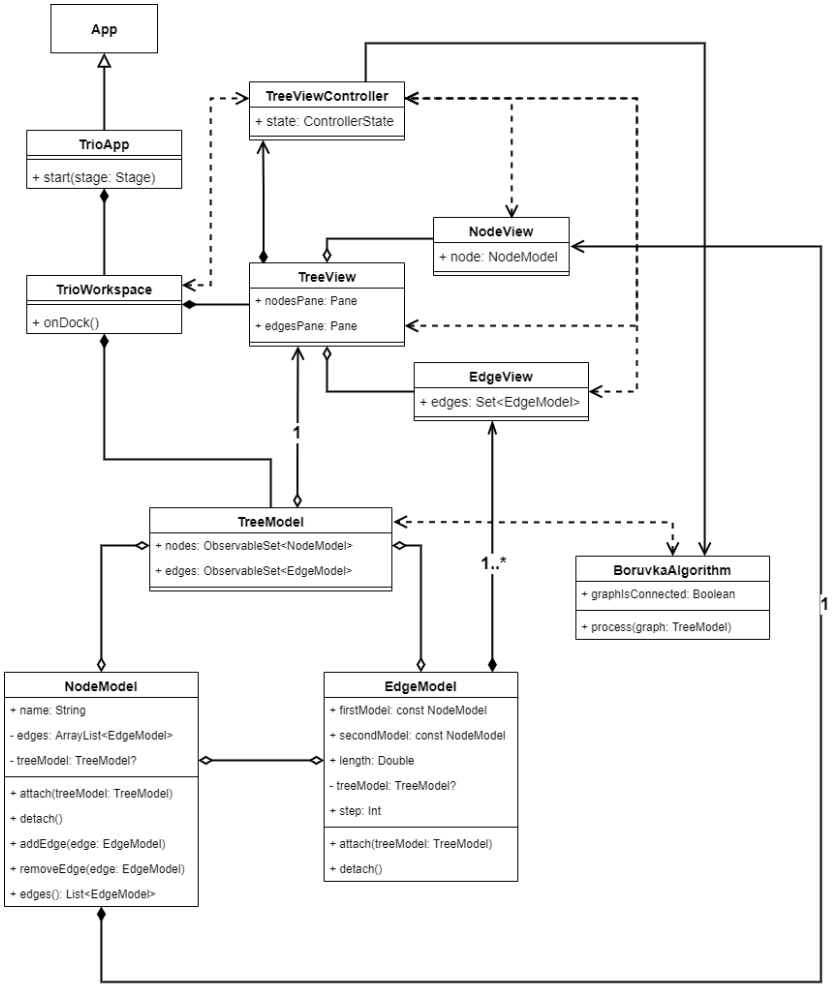

Рисунок 3. Диаграмма классов разрабатываемой программы

3.2. Описание алгоритма

При инициализации алгоритма каждая вершина графа заносится в отдельное множество – компоненту. На каждом шаге алгоритма для каждой компоненты находится ребро минимального веса, ведущее в другую компоненту, и происходит слияние с найденной компонентой. Процесс продолжается до тех пор, пока такие ребра существуют.

Для удобства формирования компонент для следующей итерации алгоритма заводится массив назначений, определяющий какая компонента текущей итерации в какую компоненту следующей итерации должна отобразиться – при инициализации каждая компонента отображается в саму себя. По мере обработки компонент текущей итерации назначения меняются. Когда одна из компонент A находит ребро, ведущую в другую компоненту B, значение в массиве назначений меняется для компоненты B так, чтобы на следующей итерации она объединилась с множеством A.

Для удобства поиска минимального ребра все ребра всех вершин компоненты объединяются в одно множества. Все внутренние ребра компонент удаляются.

3.3. Пошаговое отображение алгоритма

В классе модели ребра **EdgeModel** присутствует поле **step**, в котором хранится номер шага алгоритма, на котором это ребро выбирается в минимальное остовное дерево. При инициализации оно равно -1 (означает, что ребро не входит в остовное дерево). Значение поля **step** задается в процессе выбора ребер в ходе алгоритма. Когда пользователь нажимает на кнопки «вперед» и «назад» в режиме отображения алгоритма, меняется значение поля **currentStep** в **TreeViewController**. При изменении этого поля из множества выделенных ребер удаляются все ребра, чьи значения **step** меньше текущего **currentStep**, и добавляются все ребра, чьи значения **step** больше или равны текущему **currentStep.**

3.4. Состояния программы

На рисунке 4 представлена диаграмма состояний и переходов между ними разрабатываемой программы.

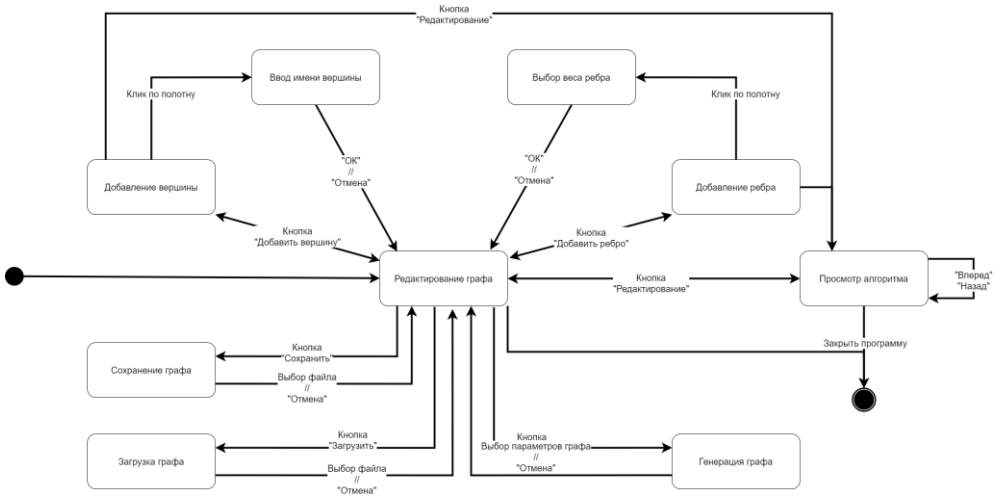

Рисунок 4. Диаграмма состояний разрабатываемой программы

3.5. Описание интерфейса

Большую часть сцены приложения занимает прокручиваемая панель для визуального отображения графа.

При запуске приложение находится в состоянии редактирования графа. В левой части верхней панели присутствуют кнопки для добавления вершин и ребер. В правой части панели есть кнопки для сохранения графа файл или загрузки из файла, а также генерации случайного графа.

Переход из состояния редактирования в состояние отображения алгоритма и обратно осуществляется при помощи кнопки «Редактирование». В состоянии редактирования доступны кнопки перемещения по шагам алгоритма: начальное состояние, шаг алгоритма назад, шаг алгоритма вперед, конечное состояние.

Посредством интерактивного взаимодействия так же удалять вершины и ребра, перемещать вершины по панели, менять вес ребер.

Вид графического интерфейса приложения представлен на рисунке 5.

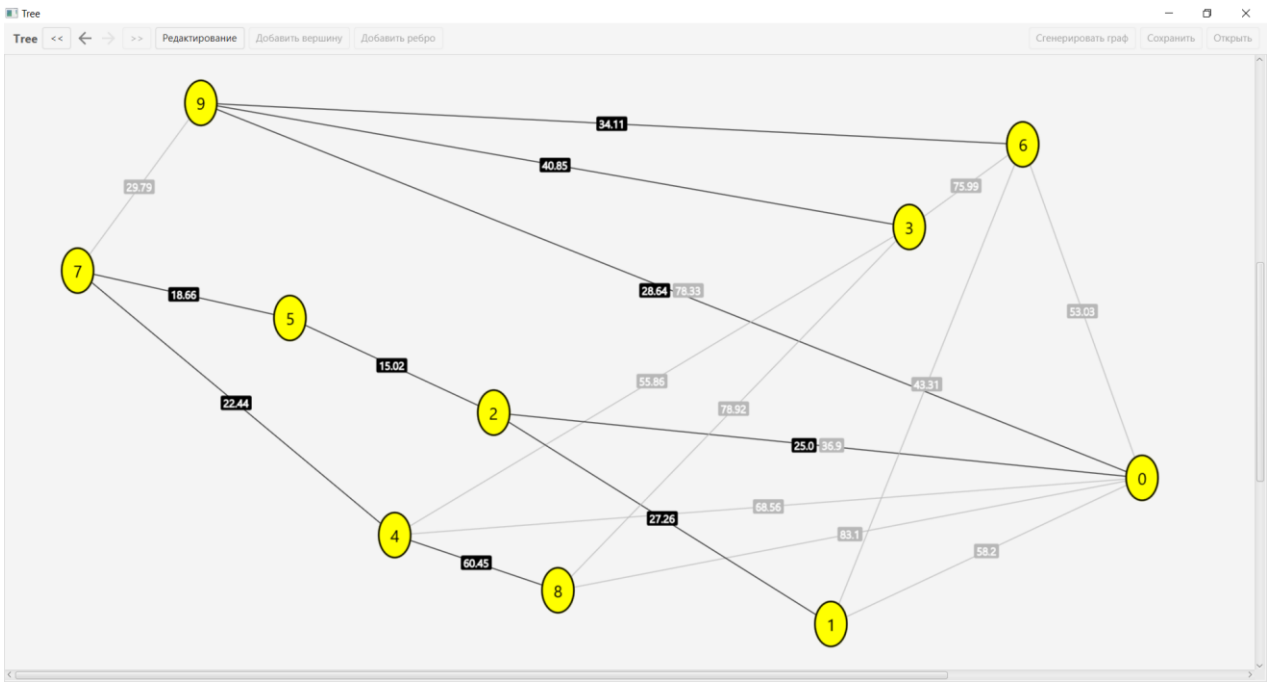

Рисунок 5. Интерфейс программы

4. ТЕСТИРОВАНИЕ

4.1. План тестирования

4.1.1. Вступление

Документ является тест-планом проекта “Алгоритм Борувки” – визуализатора алгоритма построения минимального остовного дерева. Описывается, что планируется протестировать, какие тестовые техники и подходы планируется применить.

4.1.2. Функционал, который будет протестирован

- Запуск программы;
- Ввод графа через файл;
- Ввод графа посредством GUI;
- Случайная генерация графа;
- Сохранение текущего состояния графа;
- Корректный результат работы алгоритма.

4.1.3. Тестовые единицы

Запуск программы:

- Запуск программы без сохранений;
- Запуск программы с сохранениями.

Запуск через файл: проверка на корректное состояние полей структур.

Ввод графа посредством GUI:

- Добавление вершины;
- Редактирование вершины;
- Удаление вершины;
- Добавление ребра;
- Редактирование ребра;
- Удаление ребра.

Для каждого пункта также необходима проверка полей классов **NodeModel**, **EdgeModel**, **TreeModel**, а также проверка на корректное добавление и удаление виджетов с поля.

Случайная генерация графа: проверка на корректное состояние полей классов **NodeModel**, **EdgeModel**, **TreeModel**, а также на корректное добавление виджетов на поле

4.1.4. Подход к тестированию

Тестирование будет вестись посредством фреймворка JUnit, с помощью которого удобно проверять работоспособность небольших кусков проекта.

4.2. Тестовые случаи

Класс **NodeModel:**

Тестируемый метод	Что подаётся на вход	Что ожидается на выходе
isInTree	*node*, добавляемый в дерево	node.isInTree == true
isInTree	*node*, убираемый из дерева	node.isInTree == false
attach	*node: NodeModel* *tree: TreeModel*	node.treeModel == tree
detach	*node: NodeModel*	node.treeModel != tree
addEdge	*node: NodeModel* *edge: EdgeModel*	node.edges().last() == edge
removeEdge	*node: NodeModel* *edge: EdgeMode*	node.edges().size == 0
removeEdge	*node: NodeModel* *edge: EdgeModel*	Node.edges().last() != edge

Класс **EdgeModel**:

Тестируемый метод	Что подаётся на вход	Что ожидается на выходе
isInTree	*edge*, добавленный в дерево	edge.isInTree == true
isInTree	*edge*, убранный из дерева	edge.isInTree == false
attach	*edge: EdgeModel* *tree: TreeModel*	edge.treeModel == tree
detach	*edge: EdgeModel*	edge.treeModel != tree
getLength	*edge: EdgeModel,* *len: Double*	edge.length == len

getFirstModel	*node1: NodeModel,* *node2: NodeModel,* *edge: EdgeModel*	edge.FirstModel == node1
getSecondModel	*node1: NodeModel,* *node2: NodeModel,* *edge: EdgeModel*	edge.FirstModel == node2

Класс **TreeModel**:

Тестируемый метод	Что подаётся на вход	Что ожидается на выходе
addNode	*node: NodeModel,* *tree: TreeModel*	tree.nodes.last() == node
removeNode	*node1: NodeModel,* *tree: TreeModel*	tree.nodes.size == 0
removeNode	*node1: NodeModel,* *tree: TreeModel*	tree.nodes.last() != node1
addEdge	*edge: EdgeModel,* *tree: TreeModel*	tree.edges.last() == edge
removeEdge	*edge1: EdgeModel,* *tree: TreeModel*	tree.edges.size == 0
removeEdge	*edge1: EdgeModel,* *tree: TreeMode*	tree.edges.last() != edge1
findNodeByName	*node: NodeModel,* *tree: TreeModel,* *name: String*	tree.findNodeByName(name) == node

Класс **BoruvkaAlgorithm**:

Тестируемый метод	Что подаётся на вход	Что ожидается на выходе
process	*boruvka:BoruvkaAlgorithm*	результат по каждому тесту, совпадающий со значением из выходного файла

ЗАКЛЮЧЕНИЕ

В результате выполнения работы был разработан визуализатор алгоритма Борувки для нахождения минимального остовного дерева графа. Реализованная программа полностью соответствует предъявленным требованиям. В ней присутствует возможность задания графа посредством взаимодействия с графическими элементами, с помощью случайной генерации или же заданием из файла. С графическими элементами графа можно взаимодействовать: добавлять и удалять вершины, перетаскивать их по экрану; добавлять и удалять ребра, менять их вес. В программе предусмотрена возможность просмотра пошагового выполнения алгоритма, для этого в интерфейсе созданы кнопки для перехода на следующий или предыдущий шаг алгоритма, а также сразу на конечный или начальный.

Разработка приложения выполнялась в команде. В ходе нее были практические навыки создания графического интерфейса при помощи фреймворка TornadoFX (оберткой над JavaFX), также были реализованы тесты с использованием JUnit. Был получен опыт составления диаграмм и планирования разработки программ.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Официальный сайт TornadoFX // https://tornadofx.io/ (дата обращения: 10.07.2020).
2. Официальный сайт JavaFX // https://www.oracle.com/technetwork/java/javase/overview/index.html (дата обращения: 10.07.2020).
3. Официальный сайт Gradle // https://docs.gradle.org/current/userguide/userguide.html (дата обращения: 10.07.2020).
4. JUnit :: или как полюбить валидатор JavaRush // https://javarush.ru/groups/posts/605-junit (дата обращения: 10.07.2020).

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД КЛАССА ВЕРШИНЫ

```
package model

import javafx.beans.property.ReadOnlyObjectProperty
import tornadofx.*
import ui.views.NodeView

class NodeModel(
    defaultName: String = ""
) {
    private val edges = arrayListOf<EdgeModel>()

    val nameProperty = stringProperty(defaultName)
    var name: String by nameProperty

    private val treeModelProperty = objectProperty<TreeModel?>()
    var treeModel: TreeModel? by treeModelProperty
        private set

    val inTreeProperty = treeModelProperty.booleanBinding { it != null }
    val isInTree: Boolean by inTreeProperty

    fun treeModelProperty(): ReadOnlyObjectProperty<TreeModel?> =
treeModelProperty

    val viewProperty = objectProperty<NodeView?>()
    var view by viewProperty

    fun attach(treeModel: TreeModel) {
        assert(!isInTree)
        this.treeModel = treeModel
    }

    fun detach() {
        assert(isInTree)
        assert(edges.isEmpty())
        this.treeModel = null
    }

    fun addEdge(edge: EdgeModel) {
        assert(isInTree)
        edges.add(edge)
    }

    fun removeEdge(edge: EdgeModel) {
        assert(isInTree)
```

```
        edges.remove(edge)
    }

    fun edges(): List<EdgeModel> = edges

    override fun toString(): String {
        return "NodeModel(name='$name')"
    }
}
```

ПРИЛОЖЕНИЕ Б

ИСХОДНЫЙ КОД КЛАССА РЕБРА

```
package model

import javafx.beans.property.ReadOnlyObjectProperty
import tornadofx.*
import ui.views.EdgesView

class EdgeModel(
    val firstModel: NodeModel,
    val secondModel: NodeModel,
    defaultLength: Double = 0.0
) {
    private val treeModelProperty = objectProperty<TreeModel?>()
    var treeModel: TreeModel? by treeModelProperty
        private set

    val lengthProperty = doubleProperty(defaultLength)
    var length: Double by lengthProperty

    val inTreeProperty = treeModelProperty.booleanBinding { it != null }
    val isInTree: Boolean by inTreeProperty

    val stepProperty = intProperty(-1)
    var step: Int by stepProperty

    val viewProperty = objectProperty<EdgesView>()
    var view by viewProperty

    fun otherNode(model: NodeModel) = if (model == firstModel)
secondModel else firstModel

    fun treeModelProperty(): ReadOnlyObjectProperty<TreeModel?> =
treeModelProperty

    fun attach(treeModel: TreeModel) {
        assert(!isInTree)
        this.treeModel = treeModel
        firstModel.addEdge(this)
        secondModel.addEdge(this)
    }

    fun detach() {
        assert(isInTree)
        firstModel.removeEdge(this)
        secondModel.removeEdge(this)
        treeModel = null
```

```
    }

    override fun toString(): String {
        return "EdgeModel(length=$length, first=$firstModel,
second=$secondModel)"
    }
}
```

ПРИЛОЖЕНИЕ В
ИСХОДНЫЙ КОД КЛАССА ГРАФА

```
package model

import javafx.collections.SetChangeListener
import tornadofx.*

class TreeModel {
    val nodes = observableSetOf<NodeModel>().also { nodes ->
        nodes.addListener(SetChangeListener { change ->
            if (change.wasAdded()) {
                change.elementAdded.attach(this)
            }
            else {
                val node = change.elementRemoved
                while (node.edges().isNotEmpty())
                    edges.remove(node.edges().first())
                node.detach()
            }
        })
    }
    val edges = observableSetOf<EdgeModel>().also { edges ->
        edges.addListener(SetChangeListener { change ->
            if (change.wasAdded()) {
                change.elementAdded.attach(this)
            } else {
                change.elementRemoved.detach()
            }
        })
    }

    @Deprecated("Work directly with TreeModel.nodes",
ReplaceWith("nodes.add(node)"))
    fun addNode(node: NodeModel) {
        nodes.add(node)
    }

    @Deprecated("Work directly with TreeModel.nodes",
ReplaceWith("nodes.remove(node)"))
    fun removeNode(node: NodeModel) {
        nodes.remove(node)
    }

    @Deprecated("Work directly with TreeModel.edges",
ReplaceWith("edges.add(edge)"))
    fun addEdge(edge: EdgeModel) {
        edges.add(edge)
```

```
    }

    @Deprecated("Work directly with TreeModel.edges",
ReplaceWith("edges.remove(edge)"))
    fun removeEdge(edge: EdgeModel) {
        edges.remove(edge)
    }

    fun findNodeByName(name: String): NodeModel? {
        return nodes.find { it.name == name }
    }
}
```

ПРИЛОЖЕНИЕ Г

ИСХОДНЫЙ КОД АЛГОРИТМА

```
package model

import java.util.*
import kotlin.collections.ArrayList
import kotlin.math.max
import kotlin.math.min

typealias BoruvkaComponents = ArrayList<Pair<ArrayList<NodeModel>,
LinkedList<EdgeModel>>>

class BoruvkaAlgorithm {
    var graphIsConnected = false
        private set

    fun process(graph: TreeModel) {
        for (edge in graph.edges) {
            edge.step = -1
        }
        graphIsConnected = true

        // Массив компонент графа, которые должны слиться в одну
        var components = graphToBoruvkaComponents(graph)

        var stepCounter = 0
        while (components.size > 1) {
            // nextComponents отвечает за формирование components на
следующую итерацию
            val nextComponents = BoruvkaComponents()
            // assignment какое множество с каким объединять
            val assignment = Array(components.size) { i -> i }


            for (i in components.indices) {
                // Если здесь в какой-то момент окажется пустой edgeList,
то граф не связный
                val edgeList = components[i].second
                // Компонента, не связанная с остальным графом, не
отобразится в следующем nextComponents
                // Таким образом, условие завершения
while(components.size > 1) в любом случае выполнится
                if (edgeList.isEmpty()) {
                    graphIsConnected = false
                    continue
                }

                // Нахождение минимального ребра для текущей компоненты
```

```
                val minEdge = findMinEdge(edgeList)!!
                // Оно сразу же удаляется из списка
                edgeList.remove(minEdge)
                // Не находилась ли ранее связть между этими компонентами
                var connectionIsNew = true

                // Вершина, принадлежащая текущему множеству
                val nodeFromCurrentGroup =
determineNodeOfEdgeByArray(minEdge, components[i].first)!!
                // Вершина с другого конца ребра
                val nodeFromOtherGroup =
determineNodeOfEdgeByOtherNode(minEdge, nodeFromCurrentGroup)!!


                // Если никакое ребро из предыдущих множеств не ведет в
текущее множество
                // (В nextComponents еще не существует той компоненты, в
которую должна отобразиться по assignment)
                if (assignment[i] >= nextComponents.size) {
                    // Индекс компоненты, в которую ведет текущее ребро
                    val index: Int = getGroupByNode(components,
nodeFromOtherGroup)

                    // Если для "парной" компоненты уже создана
компонента следующего шага, слияние
                    if (assignment[index] < nextComponents.size) {
                        assignment[i] = assignment[index]

nextComponents[assignment[i]].first.addAll(components[i].first)

nextComponents[assignment[i]].second.addAll(components[i].second)
                    }
                    // Если для "парной" компоненты еще не существует
отображения на следующем шаге
                    // Текущая компонента создает новую компоненту
следующего шага
                    else {
                        nextComponents.add(components[i])
                        assignment[i] = nextComponents.size - 1
                    }
                }
                // Иначе текущее множество объединяется с уже
существующим
                else {
                    // Индекс компоненты, в которую ведет текущее ребро
                    val index = getGroupByNode(components,
nodeFromOtherGroup)

                    // Если текущая компонента и так слиявается с той, в
которую ведет ребро,
                    // То это ребро не превносит ничего нового.
```

```
Соответствующая пометка
                    if (assignment[index] == assignment[i]) {
                        connectionIsNew = false
                    }
                    // Если текущая компонента должна слиться с одной из
предыдущих,
                    // Но при этом найденное ребро требует объединения с
ДРУГОЙ из числа предыдущих
                    // Объединение двух предыдущих в одно общее множество
                    else if (assignment[index] < nextComponents.size) {
                        val minInd: Int = min(assignment[i],
assignment[index])
                        val maxInd: Int = max(assignment[i],
assignment[index])


nextComponents[minInd].first.addAll(nextComponents[maxInd].first)

nextComponents[minInd].second.addAll(nextComponents[maxInd].second)
                        nextComponents.removeAt(maxInd)
                        redefineAssigment(assignment, maxInd, minInd)
                    }


nextComponents[assignment[i]].first.addAll(components[i].first)

nextComponents[assignment[i]].second.addAll(components[i].second)
                }

                // Переопределение assignment для того множества, в
которое ведет minEdge
                if (minEdge.step < 0 && connectionIsNew) {
                    for (j in i + 1 until components.size) {
                        if
(components[j].first.contains(nodeFromOtherGroup)) {
                            assignment[j] = assignment[i]
                            break
                        }
                    }
                    minEdge.step = stepCounter
                }
            }

            // Из каждого множества удаляются внутренние ребра
            removeInternalEdges(nextComponents)

            // Переход к компоненте следующей итерации
            components = nextComponents
            stepCounter++
        }
        return
```

```
    }

    private fun graphToBoruvkaComponents(graph: TreeModel):
BoruvkaComponents {
        val components = BoruvkaComponents()
        val nodes = graph.nodes
        for (node in nodes) {
            val nodeArray = ArrayList<NodeModel>()
            nodeArray.add(node)
            val edgeList = LinkedList<EdgeModel>(node.edges())
            components.add(Pair(nodeArray, edgeList))
        }
        return components
    }

    private fun findMinEdge(edgeList: List<EdgeModel>): EdgeModel? {
        if (edgeList.isEmpty()) return null

        var minEdge = edgeList.first()
        for (edge in edgeList) {
            if (edge.length < minEdge.length) {
                minEdge = edge
            }
        }
        return minEdge
    }

    private fun determineNodeOfEdgeByArray(edge: EdgeModel, nodeArray:
ArrayList<NodeModel>): NodeModel? {
        if (nodeArray.contains(edge.firstModel)) return edge.firstModel
        if (nodeArray.contains(edge.secondModel)) return edge.secondModel
        return null
    }

    private fun determineNodeOfEdgeByOtherNode(edge: EdgeModel,
nodeModel: NodeModel): NodeModel? {
        if (edge.firstModel.name == nodeModel.name) return
edge.secondModel
        if (edge.secondModel.name == nodeModel.name) return
edge.firstModel
        return null
    }

    private fun getGroupByNode(components: BoruvkaComponents, nodeModel:
NodeModel): Int {
        for (i in components.indices) {
            if (components[i].first.contains(nodeModel)) {
                return i
            }
        }
        return -1
```

```
    }

    private fun removeInternalEdges(components: BoruvkaComponents) {
        for (component in components) {
            val vertexList = component.first
            val edgeList = component.second
            val edgesToRemove = LinkedList<EdgeModel>()
            for (edge in edgeList) {
                if (vertexList.contains(edge.firstModel) &&
vertexList.contains(edge.secondModel)) {
                    edgesToRemove.add(edge)
                }
            }
            edgeList.removeAll(edgesToRemove)
        }
    }

    private fun redefineAssigment(assignment: Array<Int>, oldInd: Int,
newInd: Int) {
        for (i in assignment.indices) {
            if (assignment[i] == oldInd) {
                assignment[i] = newInd
            } else if (assignment[i] > oldInd) {
                assignment[i]--
            }
        }
    }
}
```