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

ОТЧЕТ

по учебной практике

Тема: Визуализатор алгоритма

Студент гр. 9304 _____ Силкин В.А.

Студент гр. 9304 _____ Боблаков Д.С.

Студент гр. 9304 _____ Атаманов С.Д.

Руководитель _____ Фиалковский М.С.

Санкт-Петербург

2021

ЗАДАНИЕ
НА УЧЕБНУЮ ПРАКТИКУ

Студент Силкин В.А. группы 9304

Студент Боблаков Д.С. группы 9304

Студент Атаманов С.Д. группы 9304

Тема практики: Визуализатор алгоритма

Задание на практику:

Командная итеративная разработка визуализатора алгоритма на Kotlin с графическим интерфейсом.

Алгоритм: жадный алгоритм построения минимального остовного дерева (алгоритм Краскала).

Сроки прохождения практики: 01.07.2020 – 14.07.2021

Дата сдачи отчета: 14.07.2020

Дата защиты отчета: 14.07.2020

Студент _____ Силкин В.А.

Студент _____ Боблаков Д.С.

Студент _____ Атаманов С.Д.

Руководитель _____ Фиалковский М.С.

АННОТАЦИЯ

Выполнение в бригаде визуализатора алгоритма на языке, использующем JVM (java virtual machine). В этом случае основным языком программирования был выбран Kotlin, использующим для визуализации фреймворк TornadoFX. Алгоритм для визуализации – алгоритм построения минимального остовного дерева (алгоритм Краскала). В ходе написания программы необходимо тестировать код, написанный в течении текущего этапа практики, а также составлять документацию к нему.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цель практики – за счёт работы в команде, написать визуализатор алгоритма Краскала. Алгоритм Краскала преобразует граф в минимальное остовное дерево, рассматривая минимальные рёбра, которые ещё не были рассмотрены, добавляет их в граф, если они не образуют цикл, и заканчивает свою работу, когда в графе остаётся одна компонента связности. Минимальное остовное дерево может быть использовано для оптимизации работы других алгоритмах на графе, а также для связи всех вершин графа минимальным количиством рёбер (например, это может понадобиться для связи городов с помощью дорог).

1. ТРЕБОВАНИЯ К ПРОГРАММЕ

1.1. Исходные Требования к программе

Используемая архитектура – MVC.

Интерфейс программы должен содержать следующие опции: выход из программы и сворачивание окна, вывод окна помощи, ввод данных о графе из текстового документа, поле для графического представления графа, кнопки запуска, предыдущего шага, следующего шага, быстрого завершения алгоритма, очистки поля, сохранения графа в текстовый документ.

1.1.1. Ввод данных о графе из текстового документа

Документ для ввода данных всегда выбирает пользователь. Формат документа, который хранит данные о графе - .txt. Если не удается визуализировать граф из документа - вывести ошибку, понятную пользователю (неправильный формат данных в документе, не удалось открыть файл на запись, и т.д.).

Формат данных в текстовом файле - первая строка содержит названия вершин через пробел. Остальные строки записывает рёбра графа в виде - <Значение 1-ой вершины> <Значение 2-ой вершины> <Значение ребра>. Пример записи графа в текстовом документе:

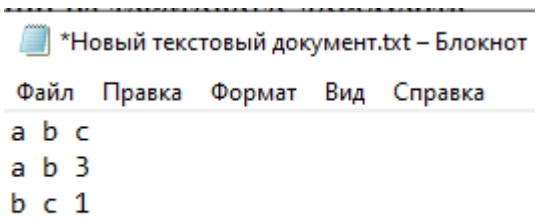

Рисунок 1 - Пример записи графа из 3 вершин и 2 рёбер в .txt документе

Значение ребра графа может быть только целочисленным или числом с плавающей точкой. Если нет - вывести ошибку при старте работы алгоритма.

1.1.2. Реализация графического отображения графа

Граф отображается в виде кругов или прямоугольников с данными в центре фигуры. Стандартный цвет рёбер – синий или чёрный. Около середины ребра отображается его значение. Граф неориентированный, потому направление рёбер графически отображать не нужно.

1.1.3 Визуализация работы алгоритма.

Шаг визуализации – Выделить добавленное ребро (красным цветом) или удалённое (синим цветом).

После старта алгоритма пользователь может перейти на предыдущий (если шаг не первый) или следующий шаг визуализации. При быстром завершении алгоритма, выводится только последний шаг визуализации. Все кнопки шагов активны только до последнего шага алгортима. После достижения последнего шага, алгоритм завершается, все красные рёбра отбрасываются, зелёные рёбра перекрашиваются в основной цвет.

1.1.4 Работа остальных кнопок интерфейса

Очистка поля сбрасывает все текущие настройки графа. Активна только вне работы алгоритма.

Сохранение графа в текстовый документ - если была вызвана до работы алгоритма, сохраняет граф, необработанный алгоритмом, если после - сохраняет обработанный граф. Нельзя вызвать во время работы алгоритма.

1.2. Уточнение требований после 1-го этапа

В интерфейс нужно добавить кнопки добавления и удаления рёбер и вершин графа. Программа должна корректно обрабатывать подаваемый граф, и уметь сохранять результат в файл.

2. ПЛАН РАЗРАБОТКИ И РАСПРЕДЕЛЕНИЕ РОЛЕЙ В БРИГАДЕ

2.1. План разработки

2.1.1. План разработки алгоритма

1) Реализован собственно сам алгоритм, возвращающий минимальное остовное дерево. – До второго этапа

2) К данному функционалу написать юнит-тесты.

3) Отладить алгоритм на основе результатов юнит тестов.

4) Реализовать логирование.

5) Реализовать просмотр работы алгоритма по шагам.

2.1.2. План разработки UI

1) Разработка окна GUI, расположение указанных кнопок. – До первого этапа

2) Добавление отображения и загрузки графа из файла

3) Возможность вызвать алгоритм из GUI – До второго этапа

4) Добавление сохранения графа, удаление графа внутри программы

5) Реализация кнопок пошагового выполнения алгоритма

6) Отладка алгоритма на тестовых данных

Окно UI выглядит следующим образом:

Рисунок 2 – Прототип UI

Поле графа находится в центре, остальные кнопки находятся справа сверху и слева сверху.

Открыть граф из файла можно с помощью File->Open. Выведется следующее окно:

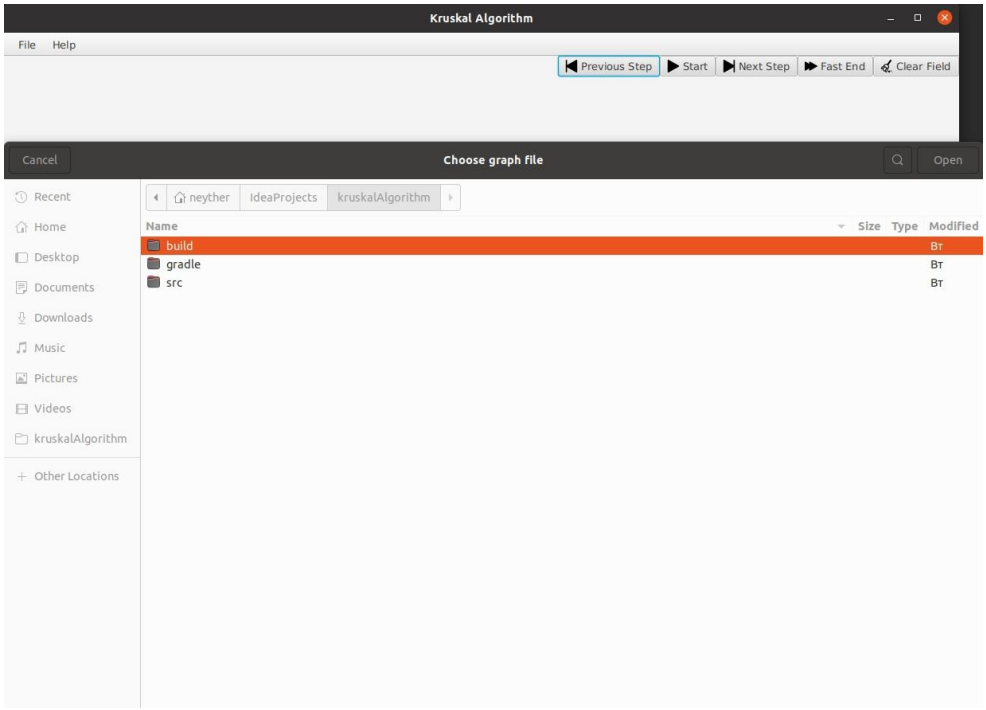

Рисунок 3 – Выбор документа, хранящего граф

После того как был выбран документ с графом, выводится его визуализация:

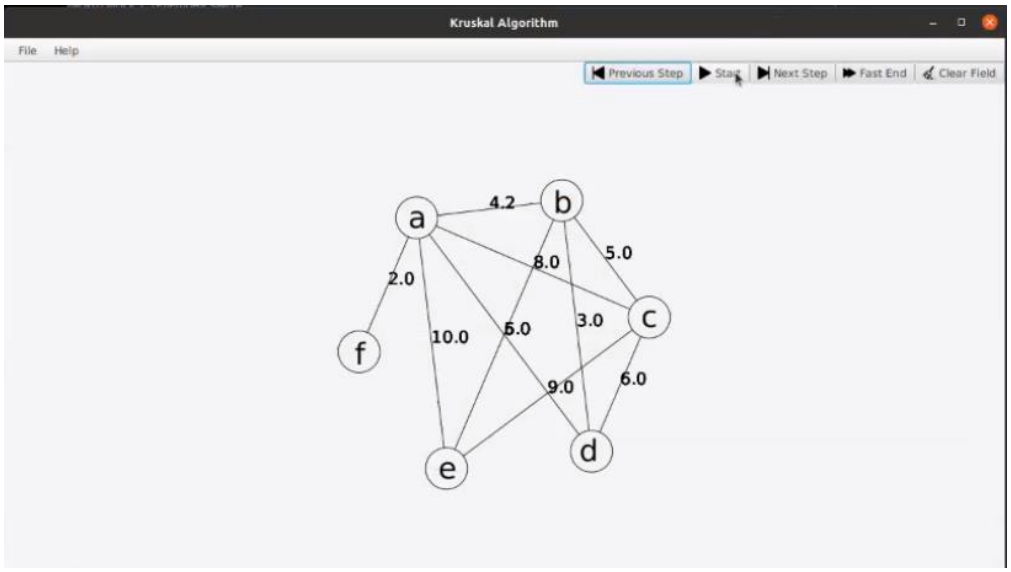

Рисунок 4 – Отображение графа с 6 вершинами

Рёбра выводятся чёрным цветом, с их весом в середине ребра. При нажатии кнопки Start, можно запустить алгоритм, становятся активными кнопки Previous Step, Next Step и Fast End, но неактивными становятся все остальные кнопки.

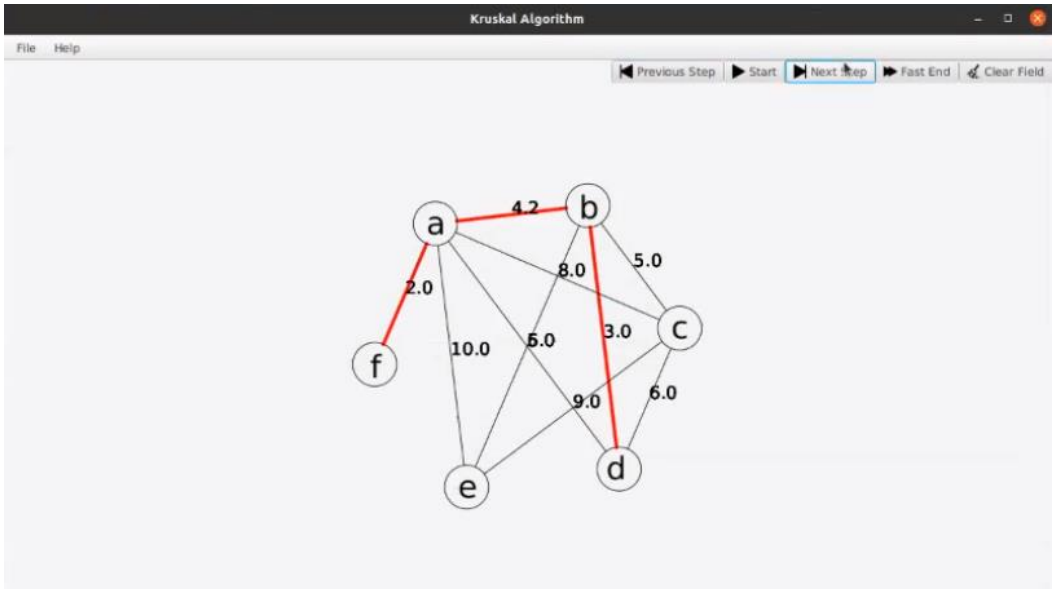

Рисунок 5 – Пошаговое добавление рёбер в граф

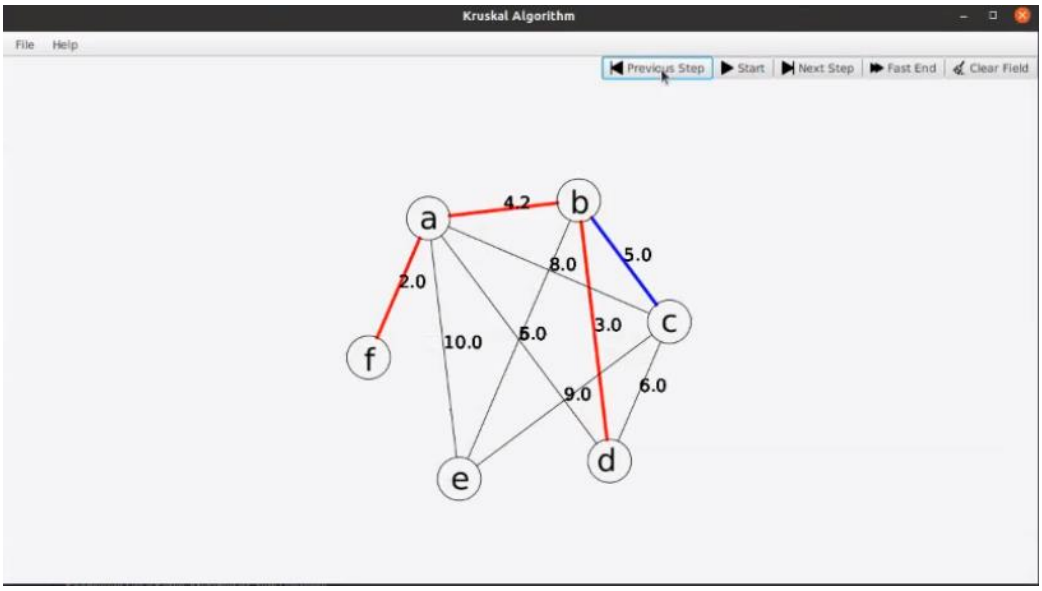

Рисунок 6 – Шаг назад

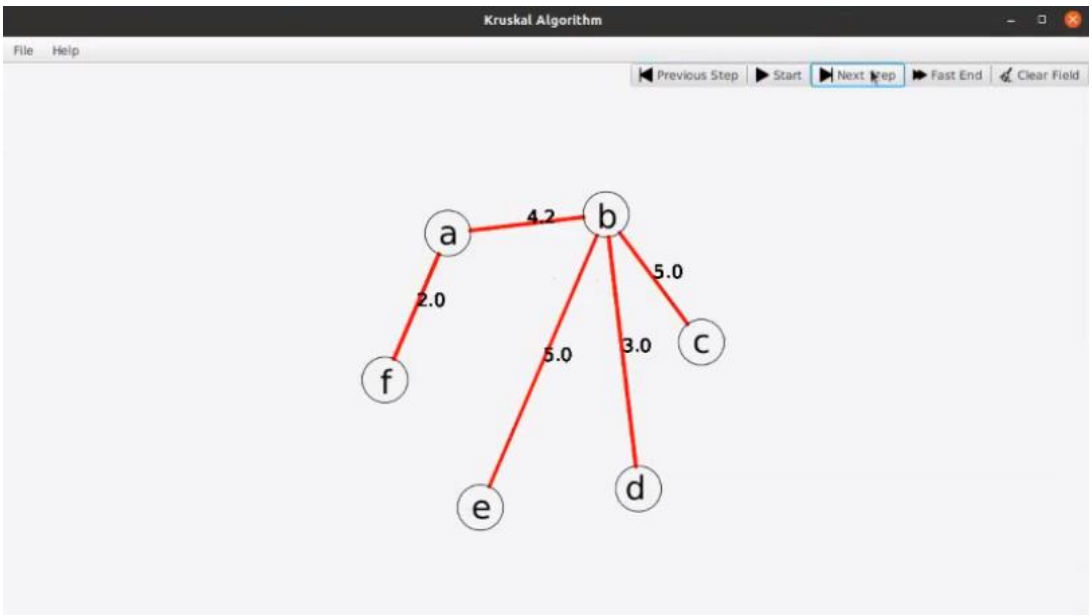

Рисунок 7 – Конечный шаг алгоритма

Рисунки 5-7 показывают работу алгоритма в пошаговом виде. При нажатии кнопки Fast End, алгоритм сразу переходит на конечный шаг алгоритма. При нажатии кнопки Next Step, добавляется новое красное ребро на основе работы алгоритма, а при шаге назад, удаляется последнее добавленное ребро, и выделяется синим.

2.2. Распределение ролей в бригаде

Роли в бригаде распределены следующим образом:

Боблаков Д.С. – ответственный за реализацию алгоритма на языке программирования kotlin (алгоритмист), тестировщик.

Атаманов С.Д. – реализация интерфейса на kotlin с помощью фреймворка TornadoFX (фронтенд).

Силкин В.А. – лидер, ответственный за документацию, тестировщик.

3. ОСОБЕННОСТИ РЕАЛИЗАЦИИ

3.1. Структуры данных

Node – класс, обозначающий вершину графа, имеет методы добавления рёбер, связанных с этой вершиной, удаления этих рёбер.

Edge – класс ребра графа, который хранит в себе вершины, которые он связывает и вес ребра.

Граф в KruskalWork хранится в HashMap<String, Node>, где String – имя вершины, а Node – объект этой же вершины. Рёбра графа хранятся в Vector<Edge>, т.е. вектор объектов Edge.

3.2. Основные методы и классы

KruskalController – класс, ответственный за логику передачи данных алгоритму. Оформляет граф в вектор.

KruskalWork – логика алгоритма, реализует сам алгоритм Краскала с помощью метода doKruskal.

UML диаграмма классов:

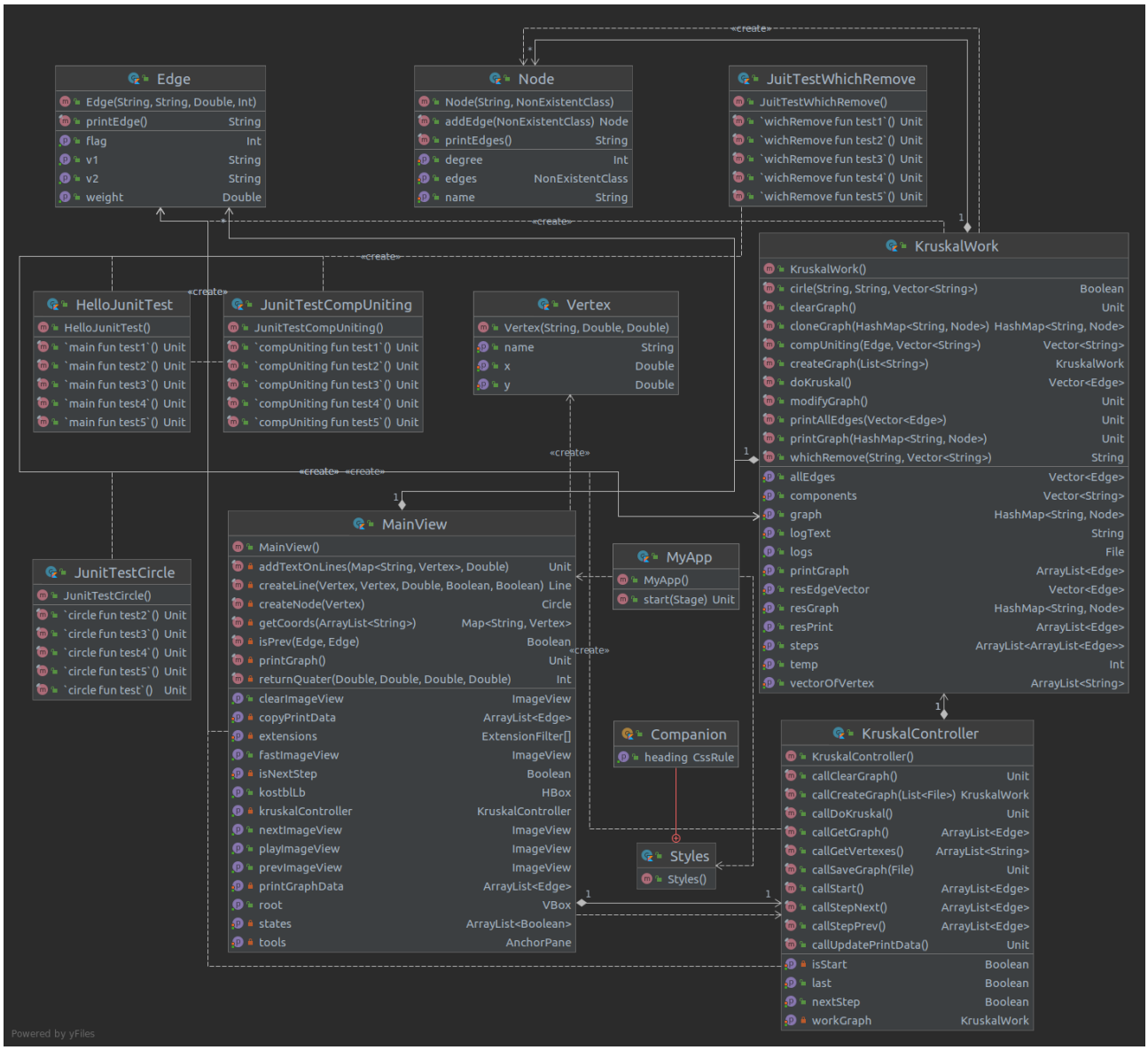

Рисунок 8 – UML-диаграмма

4. ТЕСТИРОВАНИЕ

4.1. Тестирование графического интерфейса

Графический интерфейс тестируется вручную – проверка работы всех кнопок, проверка исключений, описанных в пункте 1, проверка корректности отображения графа, а также инструментов его редактирования.

4.2. Тестирование кода алгоритма

Тестирование будет осуществляться с помощью Unit-тестов для каждой функции. Для readGraph будут применяться тесты следующего вида: некорректный/пустой вес рёбер, подача на вход несвязного графа, обычный граф для работы алгоритма. Для тестирования findMinSpanningTree будут применяться тесты вида: обычный граф для работы алгоритма, граф, имеющий более одной компоненты связности, несвязный граф, граф, имеющий рёбра с отрицательным весом, полносвязный граф, в котором все рёбра имеют одинаковый вес. Функция switchStep будет тестироваться вместе с тестированием графического интерфейса.

ЗАКЛЮЧЕНИЕ

Была проведена работа по реализации приложения, которое визуализирует граф, и алгоритм поиска минимального остовного дерева на нём. Было проведено планирование работы, а также реализация графической и алгоритмической части приложения, сама работа была распределена по членам бригады.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. https://kotlinlang.org/docs/gradle.html
2. https://codeforces.com/blog/entry/20079
3. https://stepik.org/course/5448

ПРИЛОЖЕНИЕ А
НАЗВАНИЕ ПРИЛОЖЕНИЯ

Edge.kt:

```
package com.example

class Edge(val v1: String, val  v2: String, val weight: Double, val flag: Int){

    fun printEdge(){
        println("${this.v1} <-> ${this.v2} = ${this.weight}")
    }
}
```

kruskalController.kt:

```
package com.example

import tornadofx.Controller
import java.io.File

class KruskalController : Controller(){
    private val workGraph = KruskalWork()
    fun callCreateGraph(path: List<File>) : KruskalWork{
        val fileGraph = File(path[0].toString())
        val rightLine = ArrayList<String>()

        for(item in fileGraph.readLines()[0]){
            if(!item.isLetter() && item != ' ')
                throw Exception("Wrong vertexName")
        }

       // for(item in fileGraph.readLines())
        //    rightLine.add(getStringWithoutSpaces(item))
        return workGraph.createGraph(fileGraph.readLines())
    }

    fun callSaveGraph(saveFile: File){
        var writeString = String()
        for(item in workGraph.vectorOfVertex){
            if(item == workGraph.vectorOfVertex[workGraph.vectorOfVertex.size-1])
                writeString += "$item\n"
            else
                writeString += "$item "
        }

        for(item in workGraph.printGraph){
            writeString += "${item.v1} ${item.v2} ${item.weight.toString()}\n"
        }

        saveFile.writeText(writeString)
    }

    fun callGetGraph() : ArrayList<Edge>{
```

```
        return workGraph.printGraph
    }

    fun callGetVertexes() : ArrayList<String>{
        return workGraph.vectorOfVertex
    }

    fun callDoKruskal(){
        workGraph.doKruskal()
    }

    fun callClearGraph(){
        workGraph.clearGraph()
    }

    fun callUpdatePrintData(){
        workGraph.modifyGraph()
    }

    fun getStringWithoutSpaces(changeString: String) : String{
        var outputString = String()
        for(item in changeString) {
            outputString += if (item != changeString[changeString.length - 1])
                "$item "
            else
                "$item"
        }
        return outputString
    }
}
```

kruskalWork.kt:

```
package com.example

import Node
import tornadofx.property
import java.util.*
import kotlin.collections.ArrayList
import kotlin.collections.HashMap

class KruskalWork {

    var vectorOfVertex: ArrayList<String> = ArrayList<String>()
    var graph = HashMap<String, Node>()
    var resGraph: HashMap<String, Node> = graph.clone() as HashMap<String, Node>
    var printGraph = ArrayList<Edge>()
    var allEdges = Vector<Edge>()
    var resEdgeVector = Vector<Edge>()



    fun createGraph(list: List<String>) : KruskalWork {
        for (item in list[0].split(" ")) {
```

```
            vectorOfVertex.add(item)
        }
        for (name in vectorOfVertex) {
            val node = Node(name)
            val nodeRes = Node(name)
            resGraph[name] = nodeRes
            graph[name] = node
        }

        var from: String
        var to: String
        var weight: Double

        for (i in 1 until list.size) {
            val parts = list[i].split(" ")
            from = parts[0]
            to = parts[1]
            weight = parts[2].toDouble()
            val edge1 = Edge(from, to, weight, 0)

            /*** СОЗДАЕМ РЕБРА */
            val edge2 = Edge(to, from, weight, 0)
            allEdges.addElement(edge1)

            graph[from]?.addEdge(edge1)
            /*** Добавляем ребра в граф */
            graph[to]?.addEdge(edge2)
            printGraph.add(
                Edge(
                    list[i].split(" ")[0].toString(),
                    list[i].split(" ")[1].toString(),
                    (list[i].split(" ")[2].toDouble()), 0
                )
            )
        }
        return this
    }

    fun modifyGraph(){
        printGraph.clear()
        for(item in resEdgeVector){
            printGraph.add(Edge(item.v1, item.v2, item.weight, 1))
        }
    }
    fun doKruskal(){
        allEdges.sortBy { it -> it.weight }

        val countVertex = vectorOfVertex.count()
        var countOfAddedNodes = 0
        for (edge in allEdges){
            if (!checkCycle(resGraph,edge)){
                resEdgeVector.addElement(edge)
                val edge1 = Edge(edge.v1,edge.v2,edge.weight, 0)
```

```
                val edge2 = Edge(edge.v2, edge.v1, edge.weight, 0)
                val froms = edge.v1
                val tos = edge.v2
                resGraph[froms]?.addEdge(edge1)
                resGraph[tos]?.addEdge(edge2)
                countOfAddedNodes++
            }
            if (countOfAddedNodes == countVertex-1){
                break
            }
        }

        var resLength =0.0
        for (elem in resEdgeVector){
            resLength += elem.weight
        }
        printGraph(resGraph)
        return
    }

    fun printGraph(graph: HashMap<String, Node>/*, a:Int*/){
//    println("Printed graph $a")
        for (elem in graph){
            print("${elem.key} : \n")
            elem.value.printEdges()
            println("Degree = ${elem.value.degree}")
            print("\n")
        }
    }

    fun <String, Node> clone(original: HashMap<String, Node>): MutableMap<String, Node> {
        val copy: MutableMap<String, Node> = HashMap()
        copy.putAll(original)
        return copy
    }

    fun printMutableGraph(graph: MutableMap<String, Node>){
        for (elem in graph){
            elem.value.printEdges()
        }
    }

    fun printAllEdges(edges: Vector<Edge>){
        println("All/Result Edges:")
        for (elem in edges){
            elem.printEdge()
        }
    }

    fun isExsistOneDegree(graph: HashMap<String, Node>):Boolean{
        var isExsist = false
        for (elem in graph){
            if (elem.value.degree == 1){
```

```
                return true
            }
        }

        return isExsist

    }

    fun isCycle(graph: HashMap<String, Node>) : Boolean{
        //  var status = false // не цикл
/*//
//    for (elem in graph){        //делаем все вершины не просмотренными
//      elem.value.clearStatus()
//    }
//
//    for (elem in graph){
//        dfs(elem.value, status)
//        if (status == true){
//            return true
//        }
//    }*/

        while (isExsistOneDegree(graph)){
            for (elem in graph){
                if (elem.value.degree == 1){
                    elem.value.removeNode(graph)
                }
            }
        }
        var cycle = true
        for (elem in graph){
            cycle = elem.value.degree>0
            //  printGraph(graph)
        }
        // println("HEY GRAPH - CYCLE? $cycle")
        return cycle
    }

    fun checkCycle(graph: HashMap<String, Node>, edge: Edge):Boolean{
        val from = edge.v1
        val to = edge.v2

        var status = false
        // var tmpGraph = clone(graph) as HashMap<String, Node>

        // printGraph(tmpGraph)


        var edge1 = Edge(edge.v1, edge.v2, edge.weight, 0)
        var edge2 = Edge(edge.v2, edge.v1, edge.weight, 0)
        graph[from]?.addEdge(edge1)
        graph[to]?.addEdge(edge2)
```

```
        status = isCycle(graph)
        //printGraph(tmpGraph,9)
        //remove edges
        graph[from]?.removeEdge(edge1)
        graph[to]?.removeEdge(edge2)

        return status

    }

    fun clearGraph(){
        vectorOfVertex.clear()
        graph.clear()
        resGraph.clear()
        printGraph.clear()
        allEdges.clear()
        resEdgeVector.clear()
    }
}
```

main.kt:

```
package com.example

import tornadofx.launch

fun main() {
    launch<MyApp>()
}
```

MyApp.kt:

```
package com.example

import com.example.view.MainView
import javafx.stage.Stage
import tornadofx.App

class MyApp: App(MainView::class, Styles::class){
    override fun start(stage: Stage){
        with(stage){
            width = 1200.0
            height = 675.0
        }
        super.start(stage)
    }
}
```

Node.kt:

```
import com.example.Edge
import java.util.*
import kotlin.collections.HashMap

class Node(var name: String= "", var edges:Vector<Edge> = Vector<Edge>()){

    init {
```

```
        // println("Created Node with name $name\n")
    }

    var status: Boolean = false
    var degree = 0

    fun clearStatus(){
        this.status=false
    }

    fun printName(){
        println("PRINTED: ${this.name}")
    }

    fun printEdges(){
        for (elem in this.edges){
            elem.printEdge()
        }
    }

    fun addEdge(edge: Edge): Node {
        this.edges.addElement(edge)
        this.degree++
        return this
    }
    fun removeEdge(edge: Edge): Node{
        this.edges.remove(edge)
        this.degree--
        return this
    }
    fun removeNode(graph: HashMap<String, Node>): Node{                 //  Делает узел пустым
        reduceNode(this.edges.firstElement(), graph)
        this.removeEdge(this.edges.firstElement())
        return this
    }

    fun reduceNode(edge: Edge, graph: HashMap<String, Node>): Node{       //  Уменьшает на 1
степень, удаляя ребро перевернутое
        //  var value = graph[edge.v2]
        var neibEdge = Edge(edge.v2, edge.v1, edge.weight, 0)
        graph[edge.v2]?.removeEdge(neibEdge)
        return this
    }
}
```

Styles.kt:

```
package com.example

import javafx.scene.text.FontWeight
import tornadofx.Stylesheet
import tornadofx.box
import tornadofx.cssclass
import tornadofx.px
```

```
class Styles : Stylesheet() {
    companion object {
        val heading by cssclass()
    }

    init {
        label and heading {
            padding = box(10.px)
            fontSize = 20.px
            fontWeight = FontWeight.BOLD
        }
    }
}
```

Vertex.kt:

```
package com.example

class Vertex(var name: String, var x: Double, var y: Double) {
}
```