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

ОТЧЕТ

по учебной практике

Тема: Алгоритм Борувки

Студент гр. 8303 _____ Абибулаев Э.Э.

Студент гр. 8303 _____ Рудько Д.Ю.

Студент гр. 8303 _____ Парфентьев Л.М.

Руководитель _____ Ефремов М.А.

Санкт-Петербург

2020

ЗАДАНИЕ

НА УЧЕБНУЮ ПРАКТИКУ

Студент Абибулаев Э.Э. группы 8303

Студент Рудько Д.Ю. группы 8303

Студент Парфентьев Л.М. группы 8303

Тема практики: Алгоритм Борувки

Задание на практику:

Командная итеративная разработка визуализатора алгоритма на Java с графическим интерфейсом.

Алгоритм: Борувки.

Сроки прохождения практики: 29.06.2020 – 12.07.2020

Дата сдачи отчета: 12.07.2020

Дата защиты отчета: 12.07.2020

Студент гр. 8303	_____	Абибулаев Э.Э.
Студент гр. 8303	_____	Рудько Д.Ю.
Студент гр. 8303	_____	Парфентьев Л.М.
Руководитель	_____	Ефремов М.А.

АННОТАЦИЯ

Целью данной учебной практики является разработка графического приложения визуализации алгоритма Борувки поиска минимального остовного дерева в графе. Приложение пишется на языке Java с использованием фреймворка JavaFX.

Приложение разрабатывается бригадой из трех человек за несколько итераций.

SUMMARY

The purpose of this educational practice is to develop a graphical application for visualizing the Boruvka algorithm for finding the minimum spanning tree in a graph. The application is written in Java using the JavaFX framework.

The application is developed by a team of three people for several iterations.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Целью данной практической работы является разработка графического приложения, выполняющего визуализацию работы алгоритма Борувки нахождения минимального остовного дерева графа. Реализован функционал загрузки и сохранения графа в файл на диске, а также создания или редактирования графа в самом приложении. При визуализации реализовано отображение последовательных шагов алгоритма, а также переход к ранее отображённым шагам.

Разработка осуществляется на языке Java с использованием фреймворка JavaFX командой из трёх человек. В команде распределяются обязанности – разработка алгоритма, разработка интерфейса и визуализации, сборка и тестирование программы.

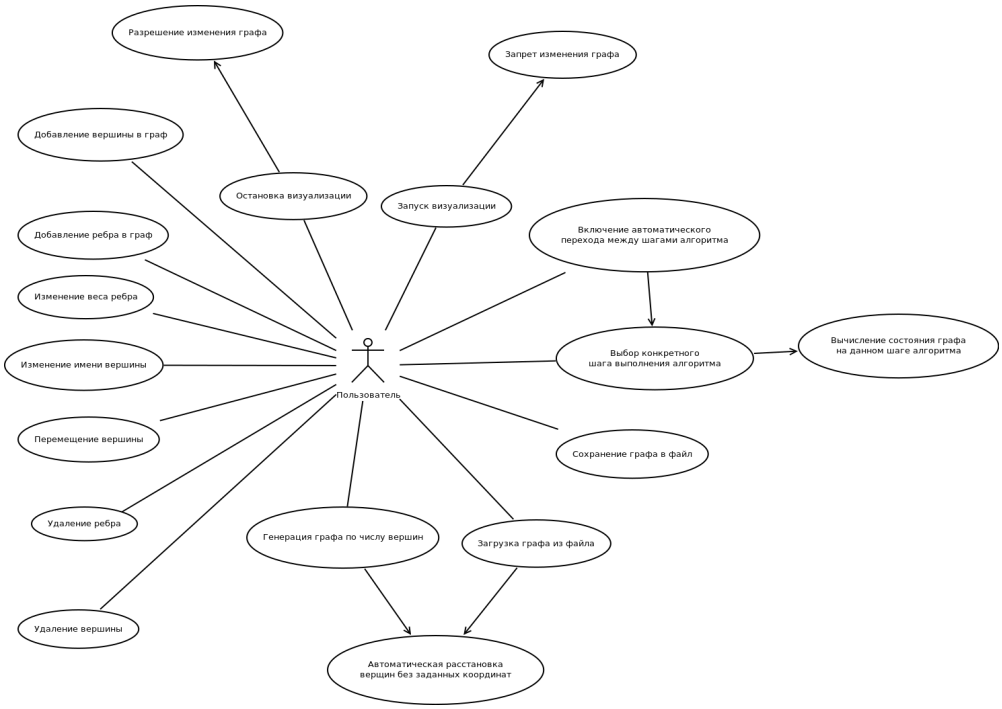
1. ТРЕБОВАНИЯ К ПРОГРАММЕ

1.1. Начальные требования

- Программа представляет собой визуализацию алгоритма Борувки нахождения наименьшего остовного дерева.
- Граф, на котором выполняется алгоритм, можно загружать из файла, а также создавать или модифицировать в самой программе.
- Визуализация пошаговая: на каждом шаге может происходить следующее:
    - Выбор очередного ребра;
    - Объединение компонент связности.
- Граф можно загрузить из файла и сохранить в файл.
    - При загрузке у части вершин могут быть указаны координаты. У остальных вершин координаты должны выставляться автоматически.
    - При сохранении графа координаты вершин записываются в файл.
- Перед запуском алгоритма граф можно изменять.
    - У вершин можно менять текст.
    - У ребер можно менять веса.
    - Можно добавлять новые ребра и вершины.
    - Можно удалять ребра и вершины.
- После запуска алгоритма граф становится неизменяемым. При выходе из режима визуализации граф снова становится изменяемым.

- При визуализации алгоритма отображается дополнительная информация.
    - Вершины и рёбра из разных компонент связности раскрашены разными цветами. Цвета выбираются случайным образом при запуске алгоритма.
    - При отображении каждого шага, на котором выбирается новое ребро, это ребро подсвечивается.
    - Не выбранные ребра внутри компонент связности (т.е. такие ребра, которые точно не будут выбраны на последующих шагах), будут отображаться специальным образом (тонкими светлыми линиями).

Диаграмма сценариев использования:



2. ПЛАН РАЗРАБОТКИ И РАСПРЕДЕЛЕНИЕ РОЛЕЙ В КОМАНДЕ

2.1. План разработки

- 2 июля: распределение ролей в бригаде; UML-диаграмма сценариев использования.
- 4 июля: графический интерфейс (не рабочий), проектирование классов программы, проектирование поведения программы.
- 6 июля: случайная генерация входных данных, обычная реализация алгоритма (до конца без промежуточных результатов) с отображением результата, план тестирования.
- 8 июля: прототип визуализации, тестирование, пошаговая реализация алгоритма.
- 10 июля: подготовка итогового “релиза”, завершение отчета.

2.2. Распределение ролей в бригаде

- Абибулаев Э.Э.: разработка визуализации и графического интерфейса.
- Рудько Д.Ю.: реализация алгоритма.
- Парфентьев Л.М.: тестирование и сборка приложения.

3. ОСОБЕННОСТИ РЕАЛИЗАЦИИ

3.1. Структура проекта

- Класс, реализующий граф – `Graph`. Вспомогательные классы `Graph.Edge` и `Graph.Node` представляют рёбра и узлы графа.
- Класс, содержащий реализацию алгоритма – `BoruvkaAlgorithm`. Этот класс получает граф в конструкторе.
- Промежуточные состояния, которые отображаются при визуализации, являются объектами `BoruvkaSnapshot`. Эти объекты содержат всю необходимую информацию, такую как тип шага и состав компонент связности.
- Промежуточные состояния могут соответствовать шагам различного типа. Шаги представлены абстрактным классом `BoruvkaStep`, а также его наследниками.
- Из объекта алгоритма промежуточные состяния запрашиваются как из итератора – методом `next`. Метод `hasNext` проверяет, что остались ещё не отданные состояния.
- Алгоритм может вычислять все состояния сразу (при вызове конструктора или первом вызове `next`), а может вычислять их по мере необходимости,
- Компоненты сзязности, образующиеся при выполнении алгоритма, представлены классом `Group`.
- Класс отображения – `ConcreteGraphView`. Он получает обновления графа через интерфейс `Subscriber`. Отображение отвечает за отрисовку графа и шагов алгоритма на экране. Промежуточные

сосотяния алгоритма отображение получает через метод setSnapshot, вызываемый контроллером.

- Класс контроллера – `GraphViewController`. Он взаимодействует с отображением через интерфейс `GraphView`; он также содержит ссылки на граф и на алгоритм (если он запущен). Контроллер реализует все операции, выполняемые из графического интерфейса над графом или алгоритмом. Также контроллер хранит предыдущие промежуточные состояния алгоритма для навигации.
- Генерация графа выполняется несколькими классами. Они используются остальной частью программы через фасад `GraphGenerationFacade`.
- Ввод-вывод графов выполняется классом `GraphIO`.

Диаграммы классов

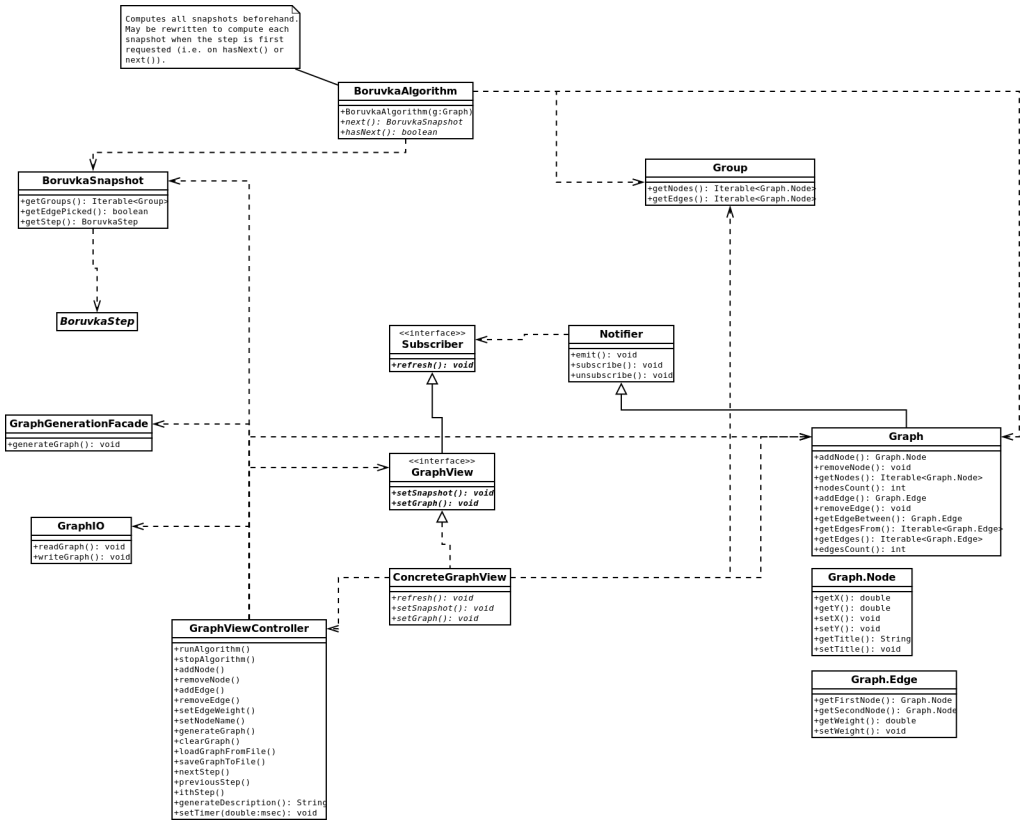


Рисунок 1: Основная диаграмма классов

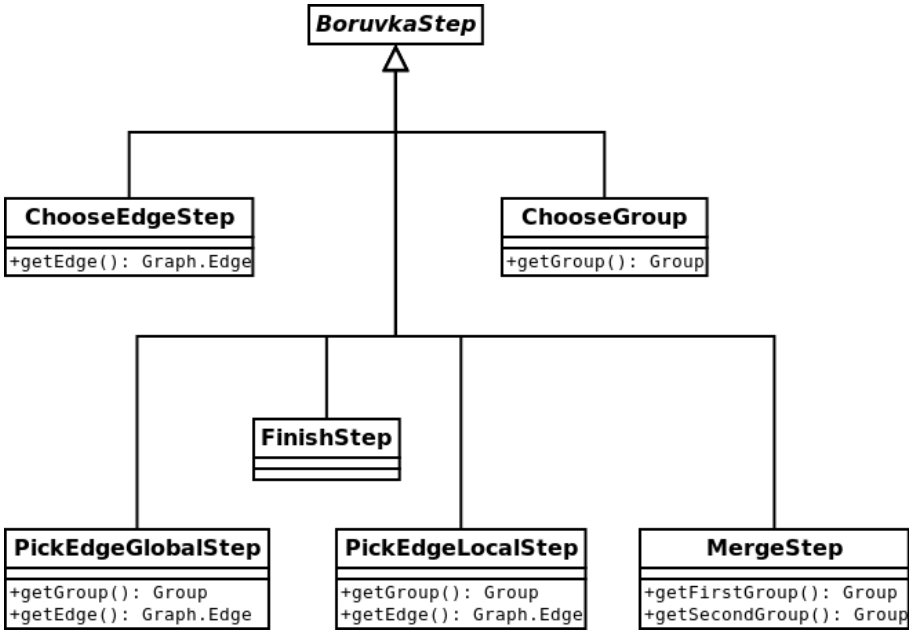


Рисунок 2: Диаграмма классов шагов алгоритма

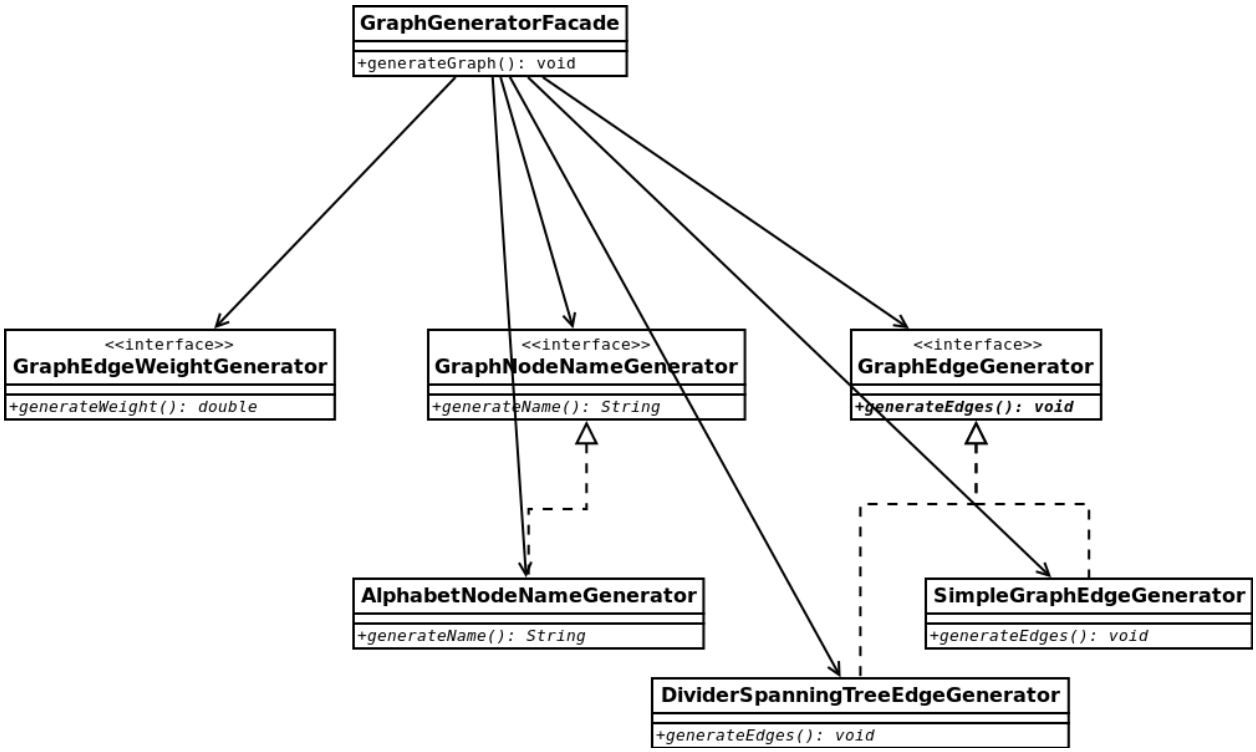


Рисунок 3: Диаграмма классов генерации графов

3.2 Описание алгоритма

Алгоритм Борувки работает следующим образом:

1. Изначально, граф, который в итоге станет минимальным каркасом исходного графа, содержит все его вершины, и не содержит ни одного ребра. Таким образом, каждая вершина составляет отдельную компоненту связности.

2. На каждом шаге, из исходного графа для каждой компоненты связности выбирается самое дешёвое ребро, ведущее в другую компоненту. Затем, эти рёбра добавляются в каркас.

3. Алгоритм завершает работу когда ни для одной компоненты связности не нашлось подходящего ребра.

3.3 Диаграммы состояний

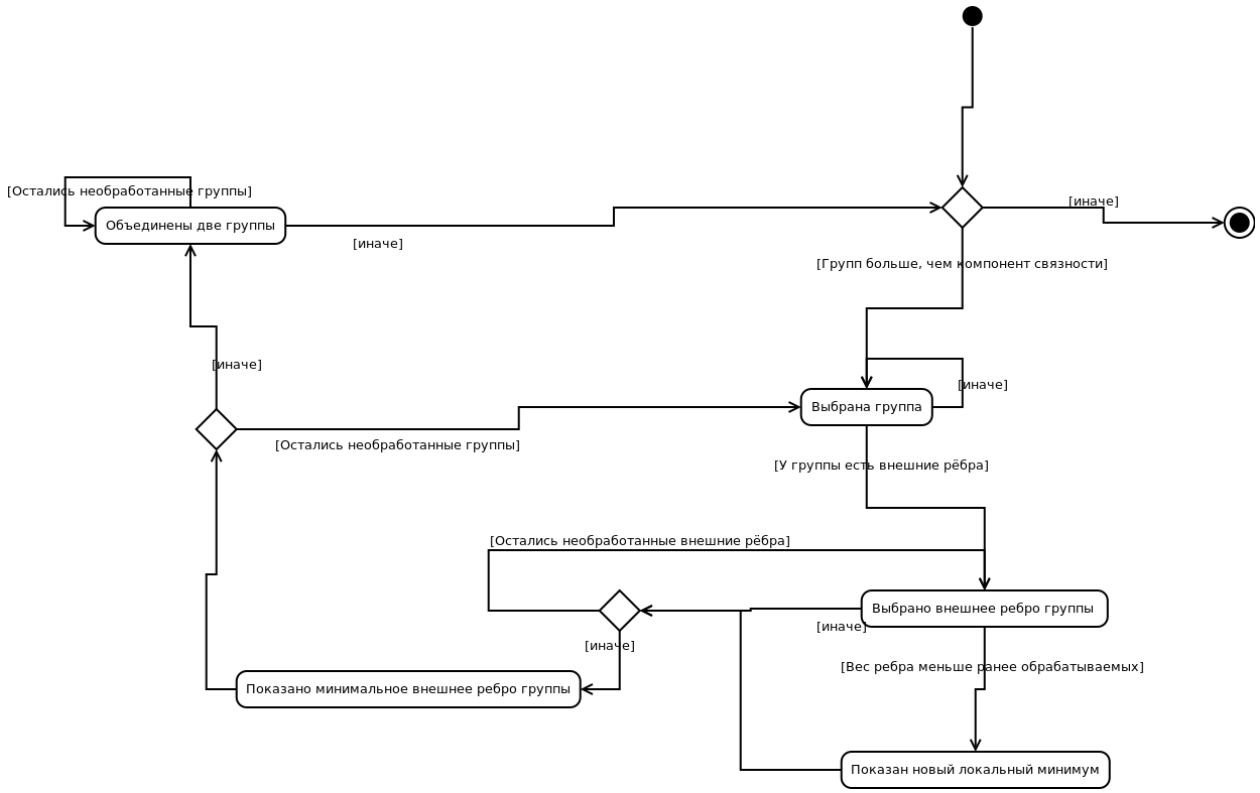


Рисунок 4: Диаграмма состояний визуализации

На рисунке 4 показана диаграмма состояний визуализации алгоритма. Состояния соответствуют подклассам `BoruvkaStep` с рисунка 2.

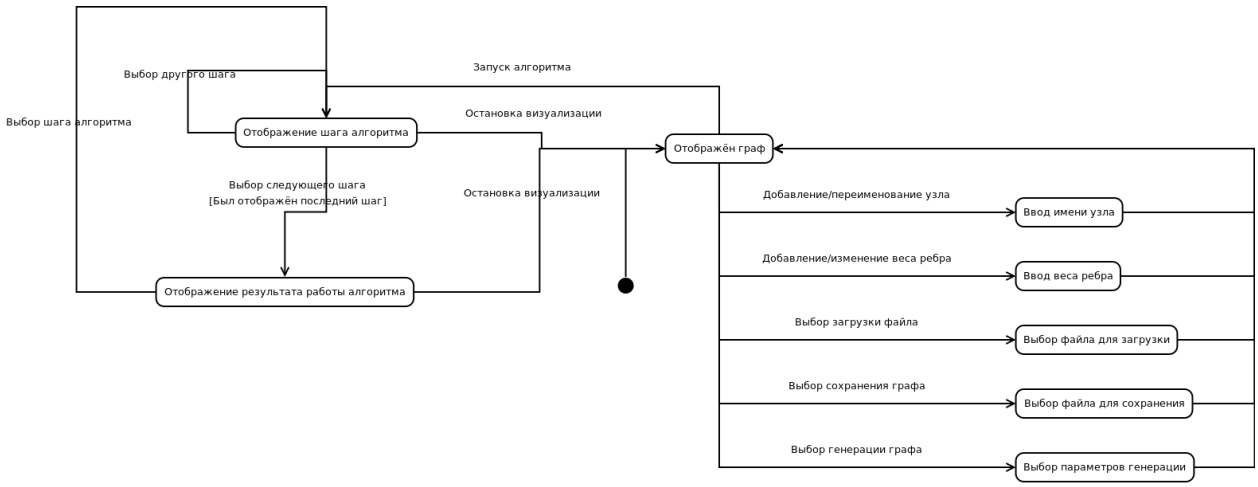


Рисунок 5: Диаграмма состояний пользовательского интерфейса

3.4. Алгоритм пошагового отображения

- Объект алгоритма умеет по очереди отдавать объекты, представляющие из себя промежуточные шаги его выполнения (как итератор).
- Контроллер запрашивает очередной шаг только когда его нужно показать, то есть когда пользователь запросил следующий шаг, и этот шаг не был получен ранее).
- Контроллер сохраняет у себя промежуточные шаги алгоритма, и может их использовать заново, например, если пользователь запросил предыдущий шаг или перезапуск визуализации на том же графе.
- Алгоритм можно реализовать так, чтобы он вычислял промежуточные шаги (в том числе итоговый ответ) только при необходимости. Пока что (8.07) это реализовано не было, поэтому алгоритм запускается до конца при запуске визуализации, и отдаёт сохранённые шаги.

4. ТЕСТИРОВАНИЕ

4.1. План тестирования

- Тестирование операций графа:
  - Базовый функционал
  - Добавление ребра в несуществующую вершину, запрос рёбер несуществующей вершины и т. п. должны выдавать исключение.
  - Удаление вершины должно вызывать удаление связанных с ней рёбер.
  - При перечислении рёбер через getEdges не должно быть дубликатов
- Тестирование алгоритма:
  - Базовый функционал
  - Корректная обработка неоднозначных ситуаций (два ребра с одинаковым весом: поведение будет явно задано в спецификации)
  - Корректная обработка несвязных графов (требует верного определения условия завершения)

Таблица 1: Запланированные тестовые случаи

Тестируемая функция	Входные данные	Ожидаемый результат
`Graph.addNode`, `Graph.removeNode`	Вершина создаётся и дважды удаляется.	Добавление вершины должно сработать, равно как и первое её удаление. При повторном удалении выбрасывается `NoSuchElementException`.
`Graph.nodesCount`	Создаётся случайное число вершин, несколько удаляются.	`Graph.nodesCount` возвращает число добавленных и не удалённых вершин.
`Graph.getNodes`	Создаётся случайное число вершин, несколько удаляются.	`Graph.getNodes` возвращает каждую добавленную, но не удалённую вершину по одному

		разу
`Graph.addEdge`	Ребро создаётся между парой существующих вершин, между существующей и удалённой, между двумя удалёнными.	Первый случай должен сработать, остальные – вызвать `NoSuchElementException`.
`Graph.removeEdge`	Ребро создаётся между парой существующих вершин, а затем дважды удаляется.	Добавление должно сработать, первое удаление тоже. Второе удаление должно вызвать `NoSuchElementException`.
`Graph.edgesCount`	Создаётся случайное число рёбер, несколько удаляются.	`Graph.edgesCount` возвращает число добавленных, но не удалённых рёбер.
`Graph.getEdges`	Создаётся случайное число рёбер, несколько удаляются.	`Graph.getEdges` возвращает каждое созданное, но не удалённое ребро по одному разу.
`Graph.removeNode`	Создаётся пара вершин и ребро между ними. Затем одна из вершин удаляется.	При удалении вершины ребро также должно быть удалено.
`Graph.getEdgeBetween`	Создаётся пара вершин, между ними добавляется ребро.	Это ребро должно быть получено при вызове `Graph.getEdgeBetween` вне зависимости от порядка, в котором указаны вершины.
`Graph.getEdgeBetween`	Запрашивается ребро между вершинами, между которыми его нет.	Должно быть получено значение `null`.
`Graph.getEdgesFrom`	Создаётся случайный граф, запрашиваются рёбра, идущие из какого-либо узла.	Должны быть получены все рёбра, идущие из вершины (для получения настоящего списка используется `Graph.getEdgeBetween`).
`Graph.DEdge.firstNode`, `Graph.Dedge.secondNode`, `Graph.getEdgeBetween`	Создаётся случайный граф.	Каждое ребро `e` в графе `g` можно получить через `g.getEdgeBetween(e.firstNode(), e.secondNode())` и `g.getEdgeBetween(e.secondNode(), e.firstNode())`
(класс `Boruvka`)	Случайный связный граф.	Результат является каркасом графа: 1. Это дерево (по количеству

		вершин и количеству рёбер в ответе) 2. Это связный граф (из случайной вершины по рёбрам из ответа достижима каждая другая вершина)
(класс `Boruvka`)	Небольшой случайный связный граф.	Результат является минимальным каркасом графа. Для проверки перебираются все подходящие наборы рёбер подходящего размера.
(класс `Boruvka`)	Граф с произвольным числом вершин и без рёбер.	Результатом должно быть пустое множество рёбер.
(класс `Boruvka`)	Случайный несвязный граф, состоящий из известного числа единиц связности известных размеров.	Результат является лесом, содержащим каркас каждой единицы связности.

ЗАКЛЮЧЕНИЕ

В результате выполнения практической работы была разработана программа с графическим интерфейсом для визуализации алгоритма Борувки. Программа соответствует начальным требованиям.

Во время разработки были получены навыки работы с языком Java, а также фреймворком построения графических интерфейсов JavaFX, библиотекой для тестирования JUnit, а также системой сборки Maven.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Эккель Б. Философия Java. 4-е полное изд. – Спб.:Питер, 2015. 1168 с.

2. Прохоренок Н.А. JavaFX. БХВ-Петербург, 2020. 768 с.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД

Имя файла: App.java

```
package summer_practice_2020.purple;

import javafx.application.Application;
import javafx.application.Platform;
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

import java.io.IOException;
import java.net.URL;


public class App extends Application {
    @Override
    public void start(Stage s) throws IOException {
        Parent root = FXMLLoader.load(getClass().getResource("ui.fxml"));
        Scene sc = new Scene(root);
        URL css_path = this.getClass().getResource("style/light/main.css");
        if (css_path == null) {
            System.err.println("Критическая ошибка: CSS файл не найден!
Программа будет завершена.");
            return;
        } else {
            sc.getStylesheets().add(css_path.toString());
        }

        s.setTitle("Визуализация работы алгоритма Борувки");
        s.setScene(sc);
        s.show();
    }

    @FXML
    protected void onQuitClicked(ActionEvent e) {
        Platform.exit();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Имя файла: Boruvka.java

```
package summer_practice_2020.purple;

import java.util.*;

public class Boruvka{
```

```
    private IGraph g;
    private HashMap<IGraph.Node, String> visitedMap = new HashMap<IGraph.Node,
String>();
    private int amountCompanent = 1;
    private Iterable<IGraph.Node> nodes;
    private Set<IGraph.Edge> blockedEdges = new HashSet<IGraph.Edge>();
    private List<IGraph.Edge> list = new ArrayList<IGraph.Edge>();
    private Set<IGraph.Edge> SnapShot = new HashSet<IGraph.Edge>();
    private List<BoruvkaSnapshot> blist = new ArrayList<BoruvkaSnapshot>();
    private Group nullGroup = new Group();
    private int step = 0;
    private  Group cloneGroupfirst, cloneGroupsecond;
    private IGraph.Edge currentMinEdge = null;

    private Queue<Group> allGroups = new ArrayDeque<>();

    public Boruvka(IGraph g) {
        this.g = g;
    }

    private void dfs(IGraph.Node v){
        visitedMap.put(v, "visited");
        for(IGraph.Edge now: g.getEdgesFrom(v)) {
            if (v.equals(now.firstNode())) {
                if (visitedMap.get(now.secondNode()).equals("not_visited")) {
                    dfs(now.secondNode());
                }
            }
            else {
                if (visitedMap.get(now.firstNode()).equals("not_visited")) {
                    dfs(now.firstNode());
                }
            }
        }
    }

    private int component(){
        int result = 0;
        for(IGraph.Node it: g.getNodes()){
            visitedMap.put(it, "not_visited");
        }

        for(IGraph.Node it:g.getNodes()){
            if(visitedMap.get(it).equals("not_visited")){
                dfs(it);
                result++;
            }
        }
        return result;
    }

    private boolean hasNext_step() {
        if (SnapShot.size() < g.nodesCount() - amountCompanent) {
            return true;
        } else {
            return false;
        }
    }

    private void next_step(){

        Group nowGroup = allGroups.remove();
```

```
        double min = 2000000;
        IGraph.Edge minEdge = null;

        Set<IGraph.Node> nowNodes = nowGroup.getNodesGroup();
        Iterable<IGraph.Edge> nowEdges = g.getEdges();
        for(IGraph.Node n: nowNodes){
            for(IGraph.Edge e: nowEdges){
                if(nowGroup.HasEdge(e) && !blockedEdges.contains(e)){
                    if(!nowGroup.getNodesGroup().contains(e.firstNode()) | !
nowGroup.getNodesGroup().contains(e.secondNode())) {
                        list.add(e);
                        if(e.getWeight() < min) {
                            min = e.getWeight();
                            minEdge = e;
                        }
                    }
                }
            }
        }
        currentMinEdge = minEdge;


        if(nowNodes.size() == 1){
            nullGroup.addNode(nowNodes.iterator().next());
        }

        if(minEdge != null) {
            blockedEdges.add(minEdge);
            boolean flag = true;
            List<Group> newlist = new ArrayList<Group>();
            newlist.addAll(allGroups);
            for (Group now : newlist) {
                if (flag && now.HasEdge(minEdge)) {
                    cloneGroupfirst = nowGroup.clone();
                    cloneGroupsecond = now.clone();
                    now.merge(nowGroup);
                    SnapShot.add(minEdge);
                }
            }
            allGroups.add(nowGroup);

        }
    }

    public void boruvka() {

        nodes = g.getNodes();

        amountCompanent = component();

        for (IGraph.Node n : nodes) {
            Group now = new Group();
            now.addNode(n);
            allGroups.add(now);
        }

        int mark = 1;
        while (hasNext_step() && !allGroups.isEmpty()) {
            next_step();
            if(list.size() > 0 && currentMinEdge != null && cloneGroupfirst !=
null && cloneGroupsecond != null) {
                List<IGraph.Edge> cEdges = new ArrayList<IGraph.Edge>();
                cEdges.addAll(list);
```

```
                IGraph.Edge cEdge = currentMinEdge;
                blist.add(new BoruvkaSnapshot(allGroups, g.getEdges(),
cloneGroupfirst.clone(), cloneGroupsecond.clone(), cEdges, cEdge));
            }
        }

    }

    public Iterable<IGraph.Edge> getSnapShotSet(){
        return SnapShot;
    }

    public Set<IGraph.Edge> resultEdgeSet(){
        return SnapShot;
    }

    public boolean hasNext() {
        if(step < blist.size()){
            return true;
        }
        else {
            return false;
        }
    }

    public void setStep(int st){
        step = st;
    }

    public BoruvkaSnapshot next() {
        step++;
        return blist.get(step-1);
    }
}
```

Имя файла: BoruvkaSnapshot.java

```
package summer_practice_2020.purple;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

public class BoruvkaSnapshot {
      private final List<Group> groups;
      private final Set<IGraph.Edge> pickedEdges;
      private final Group currentGroup;
      private final Group nextGroup;
      private final Set<IGraph.Edge> availEdges;
      private final IGraph.Edge selectedEdge;

      public Group getCurrentGroup() {
            return currentGroup;
      }

      public Group getNextGroup() {
            return nextGroup;
      }

      public IGraph.Edge getSelectedEdge() {
            return selectedEdge;
```

```
	}

	public Iterable<Group> getGroups() {
		return groups;
	}

	public boolean getEdgePicked(IGraph.Edge e) {
		return pickedEdges.contains(e);
	}

	public boolean getEdgeAvailable(IGraph.Edge e) {
		return availEdges.contains(e);
	}

	public BoruvkaSnapshot(Iterable<Group> groups,
	                       Iterable<IGraph.Edge> edges,
	                       Group currentGroup,
	                       Group nextGroup,
	                       Iterable<IGraph.Edge> availEdges,
	                       IGraph.Edge selectedEdge) {
		List<Group> groupsCopy = new ArrayList<>();
		Group currentGroupCopy = currentGroup.clone();
		Group nextGroupCopy = nextGroup.clone();
		Set<IGraph.Edge> edgesCopy = new HashSet<>();
		Set<IGraph.Edge> availEdgesCopy = new HashSet<>();

		for (Group g : groups) {
			Group gc = g.clone();
			groupsCopy.add(gc);
		}

		edges.forEach(edgesCopy::add);
		availEdges.forEach(availEdgesCopy::add);

		this.groups = groupsCopy;
		this.pickedEdges = edgesCopy;
		this.currentGroup = currentGroupCopy;
		this.availEdges = availEdgesCopy;
		this.selectedEdge = selectedEdge;
		this.nextGroup = nextGroupCopy;
	}
}
```

Имя файла: Controller.java

```
package summer_practice_2020.purple;

import javafx.animation.Animation;
import javafx.animation.KeyFrame;
import javafx.animation.Timeline;
import javafx.collections.FXCollections;
import javafx.fxml.FXML;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.canvas.Canvas;
import javafx.scene.control.*;
import javafx.scene.input.KeyCode;
import javafx.scene.input.MouseButton;
import javafx.scene.input.MouseEvent;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.Pane;
import javafx.scene.layout.VBox;
```

```
import javafx.stage.FileChooser;
import javafx.stage.Modality;
import javafx.stage.Stage;
import javafx.stage.StageStyle;
import javafx.util.Duration;
import summer_practice_2020.purple.graphgen.GraphGeneratorFacade;
import summer_practice_2020.purple.rendering.Edge;
import summer_practice_2020.purple.rendering.Node;
import summer_practice_2020.purple.rendering.Renderer;
import summer_practice_2020.purple.rendering.WorkStep;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.util.LinkedList;
import java.util.List;

public class Controller {
    Graph graphToWork;
    Boruvka algorithm;
    Renderer renderer;
    List<WorkStep> stepList;
    int index;

    boolean isGraphBlocked;
    boolean nodeMoveMode;
    boolean autoShow;
    boolean addEdgeMode;

    Node selectedNode;
    Node nodeForEdge;
    Edge selectedEdge;

    @FXML
    private MenuItem generateGraph;
    @FXML
    private MenuItem importGraph;
    @FXML
    private MenuItem exportGraph;
    @FXML
    private Pane canvas_container;
    @FXML
    private Button previous;
    @FXML
    private Button play_pause;
    @FXML
    private Button stop;
    @FXML
    private Button next;
    @FXML
    private ListView<String> list;
    @FXML
    private Canvas canvas;
    @FXML
    private Slider speed_control;

    @FXML
    private void initialize() {

        this.next.setDisable(true);
        this.previous.setDisable(true);
        this.stop.setDisable(true);
```

```
        this.speed_control.setMin(0);
        this.speed_control.setMax(10);
        this.speed_control.setBlockIncrement(0.5);


        this.isGraphBlocked = false;
        this.nodeMoveMode = false;
        this.addEdgeMode = false;
        this.autoShow = false;

        this.graphToWork = new Graph();
        this.canvas.widthProperty().bind(canvas_container.widthProperty());
        this.canvas.heightProperty().bind(canvas_container.heightProperty());
        this.renderer = new Renderer(this.canvas);


        this.renderer.setGraph(this.graphToWork);

        Timeline timeline = new Timeline(
                new KeyFrame(Duration.millis(0), ae -> {
                    System.out.println("TimerClock " + ((1000 * 10) -
this.speed_control.getValue() * 1000));
                    this.next.fire();
                }));

        timeline.setCycleCount(99999);


        generateGraph.setOnAction(e -> this.generateGraph());

        this.list.setCellFactory(param -> new ListCell<>() {
            @Override
            protected void updateItem(String item, boolean empty) {
                super.updateItem(item, empty);
                if (empty || item == null) {
                    setGraphic(null);
                    setText(null);
                } else {
                    setMinWidth(param.getWidth());
                    setMaxWidth(param.getWidth());
                    setPrefWidth(param.getWidth());
                    setWrapText(true);
                    setText(item);
                }
            }
        });

        importGraph.setOnAction(e -> {
            FileChooser fileChooser = new FileChooser();
            fileChooser.setTitle("Выберите файл графа");
            File file = fileChooser.showOpenDialog(new Stage());
            if (file == null) return;
            try {
                FileInputStream stream = new FileInputStream(file);
                this.graphToWork = new Graph();
                GraphIO.readGraph(stream, this.graphToWork);
            } catch (FileNotFoundException fileNotFoundException) {
                Alert importError = new Alert(Alert.AlertType.ERROR);
                importError.setContentText("Ошибка при импортировании графа");
                importError.showAndWait();
            }
        });
```

```
        exportGraph.setOnAction(e -> {
            FileChooser fileChooser = new FileChooser();
            fileChooser.setTitle("Выберите директорию для сохранения графа");
            fileChooser.setInitialFileName("Graph");
            File newGraphFile = fileChooser.showSaveDialog(new Stage());
            if (newGraphFile == null) return;
            try {
                FileOutputStream stream = new FileOutputStream(newGraphFile);
                GraphIO.writeGraph(stream, this.graphToWork);
            } catch (FileNotFoundException fileNotFoundException) {
                Alert importError = new Alert(Alert.AlertType.ERROR);
                importError.setContentText("Ошибка при экспортировании графа");
                importError.showAndWait();
            }
        });

        speed_control.setOnTouchReleased(e -> timeline.setRate((1000 * 10) -
this.speed_control.getValue() * 1000));

        play_pause.addEventHandler(MouseEvent.MOUSE_CLICKED, e -> {
            if (timeline.getStatus() == Animation.Status.STOPPED) {
                this.isGraphBlocked = true;
                this.algorithm = new Boruvka(this.graphToWork);
                this.algorithm.boruvka();
                this.stepList = new LinkedList<>();
                this.index = 0;

                stop.setDisable(false);

                if (this.algorithm.hasNext()) {
                    next.setDisable(false);
                    timeline.setRate((1000 * 10) - this.speed_control.getValue()
* 1000);
                    timeline.play();
                }
            } else {
                timeline.pause();
            }
        });

        next.setOnMouseClicked(e -> {
            if (this.algorithm.hasNext()) {
                this.stepList.add(new WorkStep(this.algorithm.next()));
                previous.setDisable(false);

this.renderer.addToEdgeSet(this.stepList.get(this.index).getEdge());

this.list.getItems().add(this.stepList.get(this.index).getDescription());
                this.index += 1;
                this.renderer.drawGraph();
            } else {
                this.list.getItems().add("Конец работы алгоритма");
                next.setDisable(true);
                timeline.stop();
            }
        });


        previous.setOnMouseClicked(e -> {

        });

        stop.addEventHandler(MouseEvent.MOUSE_CLICKED, e -> {
```

```
            this.renderer.setEdgeSet(null);
            this.renderer.clear();
            this.renderer.drawGraph();
            list.setItems(FXCollections.observableArrayList());
            this.isGraphBlocked = false;
            timeline.stop();
        });

        canvas_container.setOnMouseMoved(e -> this.selectedNode =
renderer.isNodePosition(e.getX(), e.getY()));

        canvas_container.setOnMouseDragged(e -> {
            if (this.selectedNode != null) {
                this.nodeMoveMode = true;
                this.selectedNode.getNode().setPosX(e.getX());
                this.selectedNode.getNode().setPosY(e.getY());
                this.renderer.drawGraph();
            }
        });

        canvas_container.setOnMouseDragReleased(e -> this.nodeMoveMode = false);

        canvas_container.setOnMouseClicked(e -> {
            ContextMenu menu = new ContextMenu();
            MenuItem deleteNode = new MenuItem("Удалить вершину");
            MenuItem addEdge = new MenuItem("Добавить ребро");
            MenuItem rename = new MenuItem("Переименовать вершину");
            TextArea newName = new TextArea();

            menu.getItems().addAll(deleteNode, addEdge, rename);
            newName.getOnKeyPressed();

            this.selectedNode = renderer.isNodePosition(e.getX(), e.getY());

            deleteNode.setOnAction(g -> {
                this.graphToWork.removeNode(this.selectedNode.getNode());
                this.renderer.drawGraph();
            });

            addEdge.setOnAction(g -> {
                this.nodeForEdge = this.selectedNode;
                this.addEdgeMode = true;
            });

            rename.setOnAction(g -> this.editPole(e));

            if (e.getButton() == MouseButton.PRIMARY && !this.isGraphBlocked) {
                if (!addEdgeMode) {
                    if (this.selectedNode == null) {
                        this.selectedEdge =
this.renderer.isEdgePosition(e.getX(), e.getY());
                        if (this.selectedEdge != null) {
                            this.editPole(e);
                        } else {
                            IGraph.Node addedNode = this.graphToWork.addNode();
                            addedNode.setTitle("name");
                            addedNode.setPosX(e.getX());
                            addedNode.setPosY(e.getY());
                            renderer.drawGraph();
                        }
                    }
                } else if (this.selectedNode != null) {
                    if (!this.selectedNode.equals(this.nodeForEdge)) {
```

```
                        IGraph.Edge edge =
this.graphToWork.addEdge(this.selectedNode.getNode(),
this.nodeForEdge.getNode());
                        edge.setWeight(0);
                        this.addEdgeMode = false;
                        this.renderer.drawGraph();
                    }
                }
            } else if (e.getButton() == MouseButton.SECONDARY) {
                if (this.selectedNode != null) {
                    menu.show(canvas_container, e.getScreenX(), e.getScreenY());
                } else {
                    this.selectedEdge = this.renderer.isEdgePosition(e.getX(),
e.getY());
                    if (this.selectedEdge != null) {

this.graphToWork.removeEdge(this.selectedEdge.getEdge());
                        this.renderer.drawGraph();
                    }
                }

            }
        });
    }

    private void editPole(MouseEvent e) {
        Stage editStage = new Stage();
        Pane root = new Pane();
        TextField textField = this.selectedNode == null ?
                new
TextField(Long.toString(Math.round(this.selectedEdge.getWeight())))
                : new TextField(this.selectedNode.getTitle());
        textField.selectAll();
        root.getChildren().addAll(textField);
        Scene editScene = new Scene(root);
        editStage.setScene(editScene);
        editStage.setX(e.getScreenX());
        editStage.setY(e.getScreenY());
        editStage.setAlwaysOnTop(true);
        editStage.initModality(Modality.APPLICATION_MODAL);
        editStage.initStyle(StageStyle.UNDECORATED);
        editStage.show();

        textField.setOnKeyPressed(f -> {
            if (f.getCode() == KeyCode.ENTER) {
                if (this.selectedNode != null) {
                    this.selectedNode.getNode().setTitle(textField.getText());
                } else {

this.selectedEdge.getEdge().setWeight(Double.parseDouble(textField.getText()));
                }
                this.renderer.drawGraph();
                editStage.close();
            }
        });
    }


    @FXML
    private void generateGraph() {
        Stage dialog = new Stage();
        dialog.setTitle("Параметры генерации");
        VBox root = new VBox();
```

```
        TextField nodesCount = new TextField();
        nodesCount.setPromptText("Введите количество вершин");
        CheckBox connected = new CheckBox();
        connected.setAllowIndeterminate(false);

        GridPane gridPane = new GridPane();
        gridPane.setHgap(10);
        gridPane.setVgap(10);
        gridPane.setPadding(new Insets(20, 150, 10, 10));

        gridPane.add(new Label("Количество вершин:"), 0, 0);
        gridPane.add(nodesCount, 1, 0);
        gridPane.add(new Label("Соединить вершины:"), 0, 1);
        gridPane.add(connected, 1, 1);

        Button submit = new Button("Принять");
        submit.setDefaultButton(true);

        root.getChildren().addAll(gridPane, submit);

        Scene scene = new Scene(root);

        dialog.setScene(scene);

        dialog.show();

        submit.setOnAction(e -> {
            this.graphToWork = new Graph();
            this.renderer.setGraph(this.graphToWork);
            new GraphGeneratorFacade().generateGraph(this.graphToWork,
                    Integer.parseInt(nodesCount.getText()),
connected.isSelected());
            this.renderer.drawGraph();
            dialog.close();
        });
    }

}
```

Имя файла: Graph.java

```
package summer_practice_2020.purple;

import java.util.HashSet;
import java.util.NoSuchElementException;
import java.util.Set;

public class Graph implements IGraph  {

    private final Set<Node> nodes = new HashSet<>();
    private final Set<Edge> edges = new HashSet<>();

    private static class DNode implements IGraph.Node {
        private String title = "";
        private double posX = -1;
        private double posY = -1;

        @Override
        public String getTitle() {
            return title;
        }
```

```
    @Override
    public void setTitle(String title) {
        this.title = title;
    }

    @Override
    public void setPosX(double posX) {
        this.posX = posX;
    }

    @Override
    public void setPosY(double posY) {
        this.posY = posY;
    }

    @Override
    public double getPosX() {
        return this.posX;
    }

    @Override
    public double getPosY() {
        return posY;
    }
}

private static class DEdge implements IGraph.Edge {
    private double weight = 0.0;
    private final Node a;
    private final Node b;

    public DEdge(Node a, Node b) {
        this.a = a;
        this.b = b;
    }

    @Override
    public void setWeight(double w) {
        weight = w;
    }

    @Override
    public double getWeight() {
        return weight;
    }

    @Override
    public Node firstNode() {
        return a;
    }

    @Override
    public Node secondNode() {
        return b;
    }
}


@Override
public Node addNode() {
    Node n = new DNode();
    nodes.add(n);
```

```
        return n;
    }

    @Override
    public void removeNode(Node node) {
        if(nodes.contains(node)) {
            nodes.remove(node);
            for (Edge e : getEdgesFrom(node)) {
                removeEdge(e);
            }
        }
        else
            throw new NoSuchElementException();
    }

    @Override
    public Iterable<Node> getNodes() {
        return nodes;
    }

    @Override
    public int nodesCount() {
        return nodes.size();
    }

    @Override
    public Edge addEdge(Node a, Node b) {
        if(a == b){
            throw new IllegalArgumentException();
        }else{
            if(getEdgeBetween(a, b) != null | getEdgeBetween(b, a) != null){
                throw new IllegalArgumentException();
            }else{
                if(!nodes.contains(a) || !nodes.contains(b)){
                    throw new NoSuchElementException();
                }else{
                    Edge e = new DEdge(a, b);
                    edges.add(e);
                    return e;
                }
            }
        }
    }

    @Override
    public void removeEdge(Edge edge) {
        if(edges.contains(edge))
            edges.remove(edge);
        else
            throw new NoSuchElementException();
    }

    @Override
    public Edge getEdgeBetween(Node a, Node b) {

        for (Edge e: edges){
            if( (e.firstNode() == a && e.secondNode() == b) || (e.firstNode() ==
b && e.secondNode() == a) ){
                return e;
            }
        }
        return null;
    }
```

```
    @Override
    public Iterable<Edge> getEdgesFrom(Node node) {
        Set<Edge> fromNode = new HashSet<>();
        for(Edge e: edges){
            if(e.firstNode() == node || e.secondNode() == node){
                fromNode.add(e);
            }
        }
        return fromNode;
    }

    @Override
    public Iterable<Edge> getEdges() {
        return edges;
    }

    @Override
    public int edgesCount() {
        return edges.size();
    }

}
```

Имя файла: GraphFormatException.java

```
package summer_practice_2020.purple;

public class GraphFormatException extends RuntimeException {

      public GraphFormatException() {
            super();
      }

      public GraphFormatException(String message, Throwable cause) {
            super(message, cause);
      }

      public GraphFormatException(String message) {
            super(message);
      }

      public GraphFormatException(Throwable cause) {
            super(cause);
      }

}
```

Имя файла: GraphIO.java

```
package summer_practice_2020.purple;

import java.io.InputStream;
import java.io.OutputStream;
import java.io.PrintStream;
import java.util.HashMap;
import java.util.Locale;
import java.util.Map;
import java.util.Scanner;
```

```
public class GraphIO {
	public static void writeGraph(OutputStream s, IGraph g) {
		PrintStream ps = new PrintStream(s);
		Map<IGraph.Node, Integer> nodes = new HashMap<>();

		int i = 0;
		for (IGraph.Node n : g.getNodes()) {
			ps.printf("node %d %s%n", i, n.getTitle());
			ps.printf("pos %d %g %g%n", i, n.getPosX(), n.getPosY());

			nodes.put(n, i);
			i++;
		}

		for (IGraph.Edge e : g.getEdges()) {
			ps.printf("edge %d %d %g%n",
					nodes.get(e.firstNode()),
					nodes.get(e.secondNode()),
					e.getWeight());
		}
	}

	private static IGraph.Node getNodeChecked(Map<String, IGraph.Node> map,
			String name) {
		IGraph.Node n = map.get(name);
		if (n == null) {
			throw new GraphFormatException("unknown node :" + name);
		}
		return n;
	}

	public static void readGraph(InputStream s, IGraph g) {
		Map<String, IGraph.Node> nodes = new HashMap<>();

		@SuppressWarnings("resource")
		Scanner sc = new Scanner(s);
		sc.useLocale(Locale.ROOT);  // required to use `.' as decimal point

		while (sc.hasNext()) {
			String key = sc.next();
			switch (key) {
			case "node": {
				String name = sc.next();
				IGraph.Node n = nodes.compute(name, (x, prev) -> {
					if (prev != null) {
						throw new GraphFormatException(
								"duplicate node: " + name);
					}
					return g.addNode();
				});
				sc.skip("\\s*");

				String title = sc.nextLine();
				n.setTitle(title);
				break;
			}

			case "edge": {
				IGraph.Node a = getNodeChecked(nodes, sc.next());
				IGraph.Node b = getNodeChecked(nodes, sc.next());
				IGraph.Edge e = g.addEdge(a, b);
				e.setWeight(sc.nextDouble());
				break;
```

```
                }

                case "pos": {
                        IGraph.Node n = getNodeChecked(nodes, sc.next());
                        n.setPosX(sc.nextDouble());
                        n.setPosY(sc.nextDouble());
                        break;
                }

                default:
                        throw new GraphFormatException("unknown key :" + key);
                }
        }
    }
}
```

Имя файла: Group.java

```
package summer_practice_2020.purple;

import java.util.HashSet;
import java.util.Set;

public class Group {
    private final Set<IGraph.Node> nodes = new HashSet<>();

    public Group() {
    }

    public Group(Iterable<IGraph.Node> nodes) {
        nodes.forEach(this.nodes::add);
    }

    // copy ctor
    public Group(Group grp) {
        grp.nodes.forEach(this.nodes::add);
    }

    public Group clone() {
        return new Group(this);
    }

    public boolean hasEdge(IGraph.Edge e) {
        return nodes.contains(e.firstNode())
                && nodes.contains(e.secondNode());
    }

    public boolean HasEdge(IGraph.Edge e) {
        return nodes.contains(e.firstNode())
                || nodes.contains(e.secondNode());
    }

    public void addNode(IGraph.Node node) {
        nodes.add(node);
    }

    public Iterable<IGraph.Node> getNodes() {
        return nodes;
    }

    public void merge(Group g) {
        g.nodes.forEach(this.nodes::add);
```

```
            g.nodes.clear();
      }

      public boolean isEmpty() {
            return nodes.isEmpty();
      }

      public Set<IGraph.Node> getNodesGroup(){
            return nodes;
      }
}
```

Имя файла: IGraph.java

```
package summer_practice_2020.purple;

    public interface IGraph {
        interface Edge {
            void setWeight(double w);
            double getWeight();
            Node firstNode();
            Node secondNode();
        }

        interface Node {
            void setTitle(String t);
            String getTitle();
            void setPosX(double posX);
            void setPosY(double posY);
            double getPosX();
            double getPosY();
        }

        Node addNode();
        void removeNode(Node node);

        Iterable<Node> getNodes();
        int nodesCount();

        Edge addEdge(Node a, Node b);
        void removeEdge(Edge edge);

        Edge getEdgeBetween(Node a, Node b);
        Iterable<Edge> getEdgesFrom(Node node);
        Iterable<Edge> getEdges();
        int edgesCount();
    }
```

Имя файла: Main.java

```
package summer_practice_2020.purple;

// See:
// https://stackoverflow.com/questions/52653836/maven-shade-javafx-runtime-
components-are-missing
public class Main {
    public static void main(String[] args) {
```

```
        App.main(args);
    }
}
```

Имя файла: AlphabetNodeNameGenerator.java

```
package summer_practice_2020.purple.graphgen;

public class AlphabetNodeNameGenerator implements GraphNodeNameGenerator {
	private final StringBuilder next = new StringBuilder("A");

	@Override
	public String generateName() {
		String prev = next.toString();

		for (int i = next.length(); i-- > 0;) {
			char c = next.charAt(i);
			char c1 = (c == 'Z') ? 'A' : (char)(c+1);
			String s = String.valueOf(c1);
			next.replace(i, i+1, s);

			if (c != 'Z') {
				return prev;
			}
		}

		next.insert(0, 'A');
		return prev;
	}

}
```

Имя файла: DividerSpanningTreeEdgeGenerator.java

```
package summer_practice_2020.purple.graphgen;

import summer_practice_2020.purple.IGraph;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;

public class DividerSpanningTreeEdgeGenerator implements GraphEdgeGenerator {
	private final Random rng = new Random();

	private void generateList(IGraph g, List<IGraph.Node> s) {
		if (s.size() < 2) {
			return;
		}

		List<IGraph.Node> left =  new ArrayList<>();
		left.add(s.remove(s.size() - 1));
		List<IGraph.Node> right = new ArrayList<>();
		right.add(s.remove(s.size() - 1));

		for (IGraph.Node n : s) {
			List<IGraph.Node> dest = rng.nextBoolean() ? left : right;
			dest.add(n);
		}
```

```
            int leftIdx = rng.nextInt(left.size());
            int rightIdx = rng.nextInt(right.size());

            g.addEdge(left.get(leftIdx), right.get(rightIdx));
            generateList(g, left);
            generateList(g, right);
      }

      @Override
      public void generateEdgesOnNodes(IGraph g, Iterable<IGraph.Node> nodes) {
            List<IGraph.Node> list = new ArrayList<>();
            nodes.forEach(list::add);
            generateList(g, list);
      }

}
```

Имя файла: GraphEdgeGenerator.java

```
package summer_practice_2020.purple.graphgen;

import summer_practice_2020.purple.IGraph;

// NOTE: nodes are pre-generated
@FunctionalInterface
public interface GraphEdgeGenerator {
      void generateEdgesOnNodes(IGraph g, Iterable<IGraph.Node> nodes);

      default void generateEdges(IGraph g) {
            generateEdgesOnNodes(g, g.getNodes());
      }
}
```

Имя файла: GraphEdgeWeightGenerator.java

```
package summer_practice_2020.purple.graphgen;

@FunctionalInterface
public interface GraphEdgeWeightGenerator {
      double generateWeight();
}
```

Имя файла: GraphGenerator.java

```
package summer_practice_2020.purple.graphgen;

import summer_practice_2020.purple.IGraph;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class GraphGenerator {
      private final int nodesCount;
      private final GraphEdgeGenerator edgeGen;
      private final GraphEdgeWeightGenerator weightGen;
      private final GraphNodeNameGenerator nameGen;

      public GraphGenerator(int nodesCount,
```

```
            GraphEdgeGenerator edgeGen,
            GraphEdgeWeightGenerator weightGen,
            GraphNodeNameGenerator nameGen) {
      this.nodesCount = nodesCount;
      this.edgeGen = edgeGen;
      this.weightGen = weightGen;
      this.nameGen = nameGen;
}

private IGraph.Node genNode(IGraph g) {
      IGraph.Node n = g.addNode();
      n.setTitle(nameGen.generateName());
      return n;
}

private void genNodes(IGraph g) {
      for (int i = 0; i < nodesCount; i++) {
            genNode(g);
      }
}

private List<List<IGraph.Node>> genNodesInComponents(
            IGraph g, Iterable<Integer> counts) {
      List<List<IGraph.Node>> lists = new ArrayList<>();
      for (int count : counts) {
            List<IGraph.Node> nodes = new ArrayList<>();
            for (int i = 0; i < count; i++) {
                  nodes.add(genNode(g));
            }
            lists.add(nodes);
      }
      return lists;
}

private void genWeights(IGraph g) {
      for (IGraph.Edge e : g.getEdges()) {
            e.setWeight(weightGen.generateWeight());
      }
}

public void generateGraph(IGraph g) {
      genNodes(g);
      edgeGen.generateEdges(g);
      genWeights(g);
}

public void generateGraphComponents(IGraph g, Iterable<Integer> counts) {
      List<List<IGraph.Node>> nodeLists = genNodesInComponents(g, counts);
      for (List<IGraph.Node> nodes : nodeLists) {
            edgeGen.generateEdgesOnNodes(g, nodes);
      }
      genWeights(g);
}

// NOTE: ignores the generator's GraphEdgeGenerator
public void generateGraphComponents(IGraph g, Iterable<Integer> counts,
            Iterable<GraphEdgeGenerator> gens) {
      Iterator<GraphEdgeGenerator> genIter = gens.iterator();
      List<List<IGraph.Node>> nodeLists = genNodesInComponents(g, counts);
      for (List<IGraph.Node> nodes : nodeLists) {
            genIter.next().generateEdgesOnNodes(g, nodes);
      }
      genWeights(g);
```

```
	}
}
```

Имя файла: GraphGeneratorFacade.java

```
package summer_practice_2020.purple.graphgen;

import summer_practice_2020.purple.IGraph;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;

public class GraphGeneratorFacade {
	private final Random rng = new Random();

	private GraphEdgeWeightGenerator makeDefaultWeightGenerator() {
		return () -> rng.nextDouble() * 24 + 1;
	}

	private GraphNodeNameGenerator makeDefaultNameGenerator() {
		return new AlphabetNodeNameGenerator();
	}

	private GraphEdgeGenerator makeUnconnectedGenerator() {
		return new SimpleGraphEdgeGenerator(0.3);
	}

	private GraphEdgeGenerator makeConnectedGenerator() {
		GraphEdgeGenerator g1 = new DividerSpanningTreeEdgeGenerator();
		GraphEdgeGenerator g2 = new SimpleGraphEdgeGenerator(0.2);
		return (g, ns) -> {
			g1.generateEdgesOnNodes(g, ns);
			g2.generateEdgesOnNodes(g, ns);
		};
	}
	private void generateGraphWithEdgeGen(IGraph g,
			GraphEdgeGenerator edgeGen, int nodesCount) {
		GraphGenerator gen = new GraphGenerator(
				nodesCount, edgeGen,
				makeDefaultWeightGenerator(),
				makeDefaultNameGenerator());
		gen.generateGraph(g);
	}

	public void generateGraph(IGraph g, int nodesCount, boolean connected) {
		GraphEdgeGenerator edgeGen = connected ? makeConnectedGenerator()
				: makeUnconnectedGenerator();
		generateGraphWithEdgeGen(g, edgeGen, nodesCount);
	}

	private GraphEdgeGenerator makeUnconnectedNEdgesGenerator(int edgesCount)
{
		return new ShuffleGraphEdgeGenerator(edgesCount);
	}


	private GraphEdgeGenerator makeConnectedNEdgesGenerator(int nodesCount,
			int edgesCount) {
		GraphEdgeGenerator g1 = new DividerSpanningTreeEdgeGenerator();
		GraphEdgeGenerator g2 = new ShuffleGraphEdgeGenerator(
				edgesCount - nodesCount + 1);
```

```
            return (g, ns) -> {
                  g1.generateEdgesOnNodes(g, ns);
                  g2.generateEdgesOnNodes(g, ns);
            };
      }

      public void generateGraphByNEdges(IGraph g,
                  int nodesCount, int edgesCount, boolean connected) {
            // TODO: check that edgesCount >= nodesCount - 1 if connected
            GraphEdgeGenerator edgeGen = connected ?
makeConnectedNEdgesGenerator(nodesCount, edgesCount)
                        : makeUnconnectedNEdgesGenerator(edgesCount);
            generateGraphWithEdgeGen(g, edgeGen, nodesCount);
      }

      public void randomlyDistribute(List<Integer> ints, int total) {
            int n = ints.size();
            for (int i = 0; i < total; i++) {
                  int idx = rng.nextInt(n);
                  ints.set(idx, ints.get(idx) + 1);
            }
      }

      public void randomlyDistributeWithUpperBound(List<Integer> ints,
                  int total, List<Integer> bounds) {
            List<Integer> availIndexes = new ArrayList<>();
            for (int i = 0; i < ints.size(); i++) {
                  if (ints.get(i) < bounds.get(i)) {
                        availIndexes.add(i);
                  }
            }
            for (int i = 0; i < total; i++) {
                  int idx = rng.nextInt(availIndexes.size());
                  int realIdx = availIndexes.get(idx);
                  Integer newVal = ints.get(realIdx) + 1;
                  ints.set(realIdx, newVal);
                  if (newVal == bounds.get(realIdx)) {
                        availIndexes.remove(idx);
                  }
            }
      }

      private List<Integer> distributeList(int n, int total) {
            List<Integer> res = new ArrayList<>();
            for (int i = 0; i < n; i++) {
                  res.add(1);
            }
            randomlyDistribute(res, total - n);
            return res;
      }

      public void generateComponents(IGraph g, int nodesCount, int compsCount) {
            GraphEdgeGenerator edgeGen = makeConnectedGenerator();

            GraphGenerator gen = new GraphGenerator(
                        nodesCount, edgeGen,
                        makeDefaultWeightGenerator(),
                        makeDefaultNameGenerator());

            List<Integer> counts = distributeList(compsCount, nodesCount);

            gen.generateGraphComponents(g, counts);
      }
```

```
	private List<GraphEdgeGenerator> distributeEdgeGenerators(
				List<Integer> nodeCounts, int edgesCount, int compsCount) {
		List<Integer> genCounts = new ArrayList<>();
		int total = edgesCount;

		for (int nc : nodeCounts) {
			genCounts.add(nc - 1);
			total -= nc - 1;
		}

		List<Integer> maxEdgeCounts = new ArrayList<>();
		nodeCounts.forEach(nc -> maxEdgeCounts.add(nc * (nc-1) / 2));

		randomlyDistributeWithUpperBound(genCounts, total, maxEdgeCounts);

		List<GraphEdgeGenerator> gens = new ArrayList<>();
		for (int i = 0; i < compsCount; i++) {
			gens.add(makeConnectedNEdgesGenerator(
					nodeCounts.get(i), genCounts.get(i)));
		}

		return gens;
	}

	public void generateComponentsWithNEdges(IGraph g,
				int nodesCount, int edgesCount, int compsCount) {

		GraphGenerator gen = new GraphGenerator(
					nodesCount, null,
					makeDefaultWeightGenerator(),
					makeDefaultNameGenerator());

		List<Integer> counts = distributeList(compsCount, nodesCount);

		List<GraphEdgeGenerator> gens =
					distributeEdgeGenerators(counts, edgesCount,
compsCount);

		gen.generateGraphComponents(g, counts, gens);
	}
}
```

Имя файла: GraphNodeNameGenerator.java

```
package summer_practice_2020.purple.graphgen;

@FunctionalInterface
public interface GraphNodeNameGenerator {
	String generateName();
}
```

Имя файла: ShuffleGraphEdgeGenerator.java

```
package summer_practice_2020.purple.graphgen;

import summer_practice_2020.purple.IGraph;

import java.util.ArrayList;
import java.util.Collections;
```

```
import java.util.List;
import java.util.Random;

public class ShuffleGraphEdgeGenerator implements GraphEdgeGenerator {

	private final int edgesCount;
	private final Random rng = new Random();

	public ShuffleGraphEdgeGenerator(int edgesCount) {
		this.edgesCount = edgesCount;
	}

	private class EdgeCandidate {
		private final IGraph.Node a;
		private final IGraph.Node b;

		public EdgeCandidate(IGraph.Node a, IGraph.Node b) {
			this.a = a;
			this.b = b;
		}

		public IGraph.Node getFirstNode() {
			return a;
		}

		public IGraph.Node getSecondNode() {
			return b;
		}
	}

	@Override
	public void generateEdgesOnNodes(IGraph g, Iterable<IGraph.Node> ns) {
		List<IGraph.Node> nodes = new ArrayList<>();
		ns.forEach(nodes::add);

		List<EdgeCandidate> possibleEdges = new ArrayList<>();
		for (int i = 0; i < nodes.size(); i++) {
			IGraph.Node ni = nodes.get(i);
			for (int j = i+1; j < nodes.size(); j++) {
				IGraph.Node nj = nodes.get(j);
				if (g.getEdgeBetween(ni, nj) != null) {
					continue;
				}

				possibleEdges.add(new EdgeCandidate(ni, nj));
			}
		}

		Collections.shuffle(possibleEdges, rng);

		int realEdgesCount = Math.min(edgesCount, possibleEdges.size());

		for (int i = 0; i < realEdgesCount; i++) {
			EdgeCandidate e = possibleEdges.get(i);
			g.addEdge(e.getFirstNode(), e.getSecondNode());
		}
	}

}
```

Имя файла: SimpleGraphEdgeGenerator.java

```
package summer_practice_2020.purple.graphgen;

import summer_practice_2020.purple.IGraph;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;

public class SimpleGraphEdgeGenerator implements GraphEdgeGenerator {

	private final double edgeProb;
	private final Random rng = new Random();

	public SimpleGraphEdgeGenerator(double edgeProb) {
		this.edgeProb = edgeProb;
	}

	@Override
	public void generateEdgesOnNodes(IGraph g, Iterable<IGraph.Node> ns) {
		List<IGraph.Node> nodes = new ArrayList<>();
		ns.forEach(nodes::add);

		for (int i = 0; i < nodes.size(); i++) {
			for (int j = i+1; j < nodes.size(); j++) {
				IGraph.Node ni = nodes.get(i);
				IGraph.Node nj = nodes.get(j);

				if (ni == nj || g.getEdgeBetween(ni, nj) != null) {
					continue;
				}

				if (rng.nextDouble() < edgeProb) {
					g.addEdge(ni, nj);
				}
			}
		}
	}

}
```

Имя файла: Edge.java

```
package summer_practice_2020.purple.rendering;

import summer_practice_2020.purple.IGraph;

public class Edge {
    private final IGraph.Edge edge;
    private final Node node1;
    private final Node node2;

    public Edge(IGraph.Edge edge, Node node1, Node node2) {
        this.edge = edge;
        this.node1 = node1;
        this.node2 = node2;
    }

    public IGraph.Edge getEdge(){
        return this.edge;
    }

    public Node getNode1() {
```

```
        return node1;
    }

    public Node getNode2() {
        return node2;
    }

    public double getWeight() {
        return this.edge.getWeight();
    }
}
```

Имя файла: EdgeList.java

```
package summer_practice_2020.purple.rendering;

import summer_practice_2020.purple.IGraph;

public class EdgeList {
    private Edge[] edgeList;
    private int size;
    private int index = 0;

    public EdgeList(int size) {
        this.size = size;
        edgeList = new Edge[this.size];
    }

    private void extend() {
        this.size += 10;
        Edge[] tmp = new Edge[size];
        if (this.index >= 0) {
            System.arraycopy(this.edgeList, 0, tmp, 0, this.index);
        }
        this.edgeList = tmp;
    }

    public void addEdge(IGraph.Edge edge, Node[] nodeList) {
        if (this.index == this.size) {
            this.extend();
        }
        Node node1 = null;
        Node node2 = null;
        for (int i = 0; (node1 == null || node2 == null) && i < nodeList.length;
i++) {
            if (edge.firstNode().equals(nodeList[i].getNode())) {
                node1 = nodeList[i];
            } else if (edge.secondNode().equals(nodeList[i].getNode())) {
                node2 = nodeList[i];
            }
        }
        this.edgeList[this.index++] = new Edge(edge, node1, node2);
        if (node2 != null) {
            if (node1 != null) {
                node2.setColor(node1.getColor());
            }
        }
    }

    public Edge[] getEdgeArray() {
        Edge[] tmp = new Edge[this.index];
        System.arraycopy(this.edgeList, 0, tmp, 0, this.index);
```

```
        return tmp;
    }
}
```

Имя файла: Node.java

```
package summer_practice_2020.purple.rendering;

import javafx.scene.paint.Color;
import summer_practice_2020.purple.IGraph;

public class Node {
    private final IGraph.Node node;
    private final double posx;
    private final double posy;
    private double radius;
    private Color color;

    public Node(IGraph.Node node, double posx, double posy, Color color) {
        this.node = node;
        this.posx = posx;
        this.posy = posy;
        this.color = color;
        this.radius = (node.getTitle().length() + 1) * 6;
    }

    public String getTitle() {
        return this.node.getTitle();
    }

    public IGraph.Node getNode() {
        return this.node;
    }

    public void setColor(Color color) {
        this.color = color;
    }

    public double getPosx() {
        return this.posx;
    }

    public double getPosy() {
        return this.posy;
    }

    public double getRadius() { return this.radius; }

    public Color getColor() {
        return this.color;
    }

    public void updateRadius(){
        this.radius =  (node.getTitle().length() + 1) * 6;
    }
}
```

Имя файла: NodeList.java

```
package summer_practice_2020.purple.rendering;
```

```
import javafx.scene.paint.Color;
import summer_practice_2020.purple.IGraph;

public class NodeList {
    private Node[] nodeList;
    private int size;
    private int index = 0;

    public NodeList(int size) {
        this.size = size;
        nodeList = new Node[this.size];
    }

    private void extend() {
        this.size += 10;
        Node[] tmp = new Node[size];
        if (this.index >= 0) {
            System.arraycopy(this.nodeList, 0, tmp, 0, this.index);
        }
        this.nodeList = tmp;
    }

    public void addNode(IGraph.Node node, double posx, double posy, Color color)
{
        if (this.index == this.size) {
            this.extend();
        }
        this.nodeList[this.index++] = new Node(node, posx, posy, color);
    }

    public Node[] getNodeArray() {
        Node[] tmp = new Node[this.index];
        if (this.index >= 0) {
            System.arraycopy(this.nodeList, 0, tmp, 0, this.index);
        }
        return tmp;
    }
}
```

Имя файла: Renderer.java

```
package summer_practice_2020.purple.rendering;

import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.paint.Color;
import summer_practice_2020.purple.Graph;
import summer_practice_2020.purple.IGraph;

import java.util.HashSet;
import java.util.Random;
import java.util.Set;


public class Renderer {
    Canvas workingCanvas;
    GraphicsContext graphicsContext;
    Graph graph;
    Set<IGraph.Edge> edgeSet;
    Node[] nodes;
    Edge[] edges;
```

```
    public Renderer(Canvas canvas) {
        this.workingCanvas = canvas;
        this.graphicsContext = this.workingCanvas.getGraphicsContext2D();
    }

    public void clear() {
        graphicsContext.clearRect(0, 0, workingCanvas.getWidth(),
workingCanvas.getHeight());
    }

    public void setGraph(Graph graph) {
        this.setEdgeSet(new HashSet<>());
        this.graph = graph;
    }

    public void setEdgeSet(Set<IGraph.Edge> edgeSet) {
        this.edgeSet = edgeSet;
    }

    public void addToEdgeSet(IGraph.Edge edge) {
        if (edge == null) {
            System.out.println("edge null");
            System.exit(-1);
        } else if (this.edgeSet == null) {
            System.out.println("Edgeset null");
            System.exit(-2);
        }
        this.edgeSet.add(edge);
    }

    public void drawGraph() {

        NodeList nodeList = new NodeList(graph.nodesCount());
        EdgeList edgeList = new EdgeList(graph.edgesCount());
        double angle = 90.0;
        double angleStep = 360.0 / graph.nodesCount();
        double posx;
        double posy;
        Random r = new Random();
        r.setSeed(System.currentTimeMillis());

        for (IGraph.Node node : this.graph.getNodes()) {
            if (node.getPosX() == -1) {
                double tmpMin = Math.min(this.workingCanvas.getWidth(),
this.workingCanvas.getHeight());
                tmpMin *= 0.45;
                posx = this.workingCanvas.getWidth() / 2 + tmpMin * Math.cos(2 *
Math.PI / 360 * angle);
                posy = this.workingCanvas.getHeight() / 2 - tmpMin * Math.sin(2
* Math.PI / 360 * angle);
                angle += angleStep;
                node.setPosX(posx);
                node.setPosY(posy);
            }
            nodeList.addNode(node, node.getPosX(), node.getPosY(),
Color.rgb(255, 255, 255));
        }

        for (IGraph.Edge edge : this.graph.getEdges()) {
            edgeList.addEdge(edge, nodeList.getNodeArray());
        }
```

```
        this.edges = edgeList.getEdgeArray();
        this.nodes = nodeList.getNodeArray();

        clear();

        for (int i = 0; i < edgeList.getEdgeArray().length; i++) {
            drawEdge(edges[i]);
        }

        for (int i = 0; i < nodeList.getNodeArray().length; i++) {
            this.nodes[i].updateRadius();
            drawNode(this.nodes[i]);
        }
    }


    public void drawNode(Node node) {
        this.graphicsContext.setFill(node.getColor());
        this.graphicsContext.setLineWidth(1);
        this.graphicsContext.setStroke(Color.rgb(0, 0, 0));
        this.graphicsContext.fillOval(node.getPosx() - node.getRadius(),
node.getPosy() - node.getRadius(),
                node.getRadius() * 2, node.getRadius() * 2);
        this.graphicsContext.strokeOval(node.getPosx() - node.getRadius(),
node.getPosy() - node.getRadius(),
                node.getRadius() * 2, node.getRadius() * 2);
        this.graphicsContext.strokeText(node.getTitle(), node.getPosx() -
node.getRadius() / 6.0, node.getPosy() + 3);
    }

    public void drawEdge(Edge edge) {
        Node node1 = edge.getNode1();
        Node node2 = edge.getNode2();
        if (this.edgeSet != null && this.edgeSet.contains(edge.getEdge())) {
            this.graphicsContext.setLineWidth(7);
        } else if (this.edgeSet == null) {
            this.graphicsContext.setLineWidth(3);
        } else {
            this.graphicsContext.setLineWidth(1);
        }

        double middlePosX = (node2.getPosx() - node1.getPosx()) / 2;
        double middlePosY = (node2.getPosy() - node1.getPosy()) / 2;
        int approximatedValueOfWeight = (int) edge.getWeight();

        this.graphicsContext.setStroke(Color.rgb(0, 0, 0));

        this.graphicsContext.strokeLine(node1.getPosx(), node1.getPosy(),
node2.getPosx(), node2.getPosy());
        this.graphicsContext.setLineWidth(1);
        this.graphicsContext.setFill(Color.rgb(255, 255, 255));
        final int i = (String.valueOf(approximatedValueOfWeight).length() + 1) *
6;
        this.graphicsContext.fillRect(node1.getPosx() + middlePosX - i,
node1.getPosy() + middlePosY - 10,
                (String.valueOf(Math.round(edge.getWeight())).length() + 2) * 6,
15);
        this.graphicsContext.strokeRect(node1.getPosx() + middlePosX - i,
node1.getPosy() + middlePosY - 10,
                (String.valueOf(Math.round(edge.getWeight())).length() + 2) * 6,
15);

this.graphicsContext.strokeText(Integer.toString(approximatedValueOfWeight),
```

```
                node1.getPosx() + middlePosX -
String.valueOf(approximatedValueOfWeight).length() * 6,
                node1.getPosy() + middlePosY + 3);
    }

    public Edge isEdgePosition(double posx, double posy) {
        double edgePosX;
        double edgePosY;
        Node node1;
        Node node2;
        double middlePosX;
        double middlePosY;
        int approximatedValueofWeight;
        if (graph != null && this.edges != null) {
            for (Edge edge : this.edges) {
                node1 = edge.getNode1();
                node2 = edge.getNode2();
                middlePosX = (node2.getPosx() - node1.getPosx()) / 2;
                middlePosY = (node2.getPosy() - node1.getPosy()) / 2;
                approximatedValueofWeight = (int) edge.getWeight();
                edgePosX = node1.getPosx() + middlePosX -
                        (String.valueOf(approximatedValueofWeight).length() + 1)
* 6;
                edgePosY = node1.getPosy() + middlePosY - 10;
                if (posx >= edgePosX && posx < edgePosX +
(String.valueOf(Math.round(edge.getWeight())).length() + 2) * 6) {
                    if (posy >= edgePosY && posy < edgePosY + 15) {
                        return edge;
                    }
                }
            }
        }
        return null;
    }

    public Node isNodePosition(double posx, double posy) {
        double nodePosX;
        double nodePosy;
        if (graph != null && this.nodes != null) {
            for (Node node : this.nodes) {
                nodePosX = node.getPosx();
                nodePosy = node.getPosy();
                if (Math.pow(Math.abs(posx - nodePosX), 2) +
Math.pow(Math.abs(posy - nodePosy), 2) <= Math.pow(node.getRadius(), 2)) {
                    return node;
                }
            }
        }
        return null;
    }
}
```

Имя файла: WorkStep.java

```
package summer_practice_2020.purple.rendering;

import summer_practice_2020.purple.BoruvkaSnapshot;
import summer_practice_2020.purple.IGraph;

public class WorkStep {
    private final IGraph.Edge edge;
    private final String description;
```

```
    public WorkStep(BoruvkaSnapshot boruvkaSnapshot) {
        this.edge = boruvkaSnapshot.getSelectedEdge();
        this.description = "Выбрано ребро между " + edge.firstNode().getTitle()
                + " и " + edge.secondNode().getTitle();

    }

    public IGraph.Edge getEdge() {
        return this.edge;
    }

    public String getDescription() {
        return this.description;
    }
}
```

Имя файла: BoruvkaTest.java

```
package summer_practice_2020.purple;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertFalse;
import static org.junit.jupiter.api.Assertions.assertTrue;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.HashSet;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.NoSuchElementException;
import java.util.Queue;
import java.util.Random;
import java.util.Set;

import org.junit.jupiter.api.RepeatedTest;
import org.junit.jupiter.api.Test;

import summer_practice_2020.purple.IGraph.Edge;
import summer_practice_2020.purple.IGraph.Node;
import summer_practice_2020.purple.graphgen.AlphabetNodeNameGenerator;
import summer_practice_2020.purple.graphgen.DividerSpanningTreeEdgeGenerator;
import summer_practice_2020.purple.graphgen.GraphEdgeGenerator;
import summer_practice_2020.purple.graphgen.GraphEdgeWeightGenerator;
import summer_practice_2020.purple.graphgen.GraphGeneratorFacade;
import summer_practice_2020.purple.graphgen.GraphNodeNameGenerator;
import summer_practice_2020.purple.graphgen.SimpleGraphEdgeGenerator;

class BoruvkaTest {

      private static final Random rng = new Random();

      private static int randomInt(int min, int max) {
            return rng.nextInt(max - min + 1) + min;
      }

      private static IGraph createEmptyGraph() {
            return new Graph();
      }
```

```
	private static void generateConnectedGraph(IGraph g, int nodesCount) {
		GraphNodeNameGenerator ngen = new AlphabetNodeNameGenerator();
		GraphEdgeWeightGenerator wgen = () -> rng.nextInt(100);
		GraphEdgeGenerator gen1 = new DividerSpanningTreeEdgeGenerator();
		GraphEdgeGenerator gen2 = new SimpleGraphEdgeGenerator(0.8);

		for (int i = 0; i < nodesCount; i++) {
			g.addNode();
		}
		gen1.generateEdges(g);
		gen2.generateEdges(g);
		for (Node n : g.getNodes()) {
			n.setTitle(ngen.generateName());
		}
		for (Edge e : g.getEdges()) {
			e.setWeight(wgen.generateWeight());
		}
	}

	private static Node someNode(IGraph g) {
		return g.getNodes().iterator().next();
	}

	private static Set<Node> reachableNodes(IGraph g, Set<Edge> edges, Node
start) {
		Set<Node> nodes = new HashSet<>();
		nodes.add(start);

		Queue<Node> q = new LinkedList<>();
		q.add(start);

		while (!q.isEmpty()) {
			Node n = q.remove();
			for (Edge e : edges) {
				Node d;
				if (e.firstNode() == n) {
					d = e.secondNode();
				} else if (e.secondNode() == n) {
					d = e.firstNode();
				} else {
					continue;
				}
				if (nodes.add(d)) {
					q.add(d);
				}
			}
		}

		return nodes;
	}

	@Test
	void testGraphUnmodified() {
		final int nNodes = 50;
		IGraph g = createEmptyGraph();
		generateConnectedGraph(g, nNodes);

		List<Double> weights = new ArrayList<>();
		for (Edge e : g.getEdges()) {
			weights.add(e.getWeight());
		}

		Boruvka b = new Boruvka(g);
```

```
            b.boruvka();

            int i = 0;
            for (Edge e : g.getEdges()) {
                  assertEquals(weights.get(i++), e.getWeight());
            }
      }

      @RepeatedTest(5)
      void testResultIsSpanningTree() {
            final int nNodes = randomInt(50, 100);
            IGraph g = createEmptyGraph();
            generateConnectedGraph(g, nNodes);

            Boruvka b = new Boruvka(g);
            b.boruvka();
            Set<Edge> edges = b.resultEdgeSet();

            assertEquals(nNodes - 1, edges.size());

            Set<Node> nodes = reachableNodes(g, edges, someNode(g));
            assertEquals(nNodes, nodes.size());
      }

      private static double totalWeight(Iterable<Edge> edges) {
            double sum = 0;
            for (Edge e : edges) {
                  sum += e.getWeight();
            }
            return sum;
      }

      // needed to test the algorithm
      private static class CombinationsIterator<T> implements Iterator<Set<T>> {

            private List<Integer> next = new ArrayList<>();
            private final List<T> elts;
            private final int size;

            public CombinationsIterator(List<T> elts, int size) {
                  if (size < 0 || size > elts.size()) {
                        throw new IllegalArgumentException(
                                    "requested size negative or greater than
number of elements");
                  }

                  this.elts = elts;
                  this.size = size;

                  for (int i = 0; i < size; i++) {
                        next.add(i);
                  }
            }

            private Set<T> makeCurrentSet() {
                  Set<T> s = new HashSet<>();
                  for (int i = 0; i < size; i++) {
                        s.add(elts.get(next.get(i)));
                  }
                  return s;
            }

            private boolean advance() {
```

```
            int i;
            for (i = 0; i < size; i++) {
                  if (next.get(size - i - 1) < elts.size() - i - 1) {
                        break;
                  }
            }
            if (i == size) {
                  return false;
            }

            int pos = size - i - 1;
            int low = next.get(pos)+1;
            next.set(pos, low);

            for (int j = 0; j+pos < size; j++) {
                  next.set(pos+j, low+j);
            }

            return true;
      }

      @Override
      public Set<T> next() {
            if (!hasNext()) {
                  throw new NoSuchElementException();
            }
            Set<T> s = makeCurrentSet();
            if (!advance()) {
                  next = null;
            }
            return s;
      }

      @Override
      public boolean hasNext() {
            return next != null;
      }

}

@Test
void testCombinationsIterator() {
      final List<Integer> answer = new ArrayList<>();
      final int n = 5;
      final int k = 3;
      Collections.addAll(answer,
                  7, 11, 19, 13, 21, 25, 14, 22, 26, 28);

      List<Integer> elts = new ArrayList<>();
      for (int i = 0, next = 1; i < n; i++, next *= 2) {
            elts.add(next);
      }

      Iterator<Set<Integer>> iter =
                  new CombinationsIterator<>(elts, k);

      // trick to get iterator from stream
      int[] res = new int[10];
      int count = 0;

      while (iter.hasNext()) {
            Set<Integer> s = iter.next();
            int sum = s.stream().mapToInt(Integer::intValue).sum();
```

```
            if (res.length == count) res = Arrays.copyOf(res, count * 2);
            res[count++] = sum;
        }
        res = Arrays.copyOfRange(res, 0, count);

        assertEquals(answer.size(), res.length);

        for (int i = 0; i < res.length; i++) {
            assertEquals(answer.get(i), res[i]);
        }
    }

    @RepeatedTest(5)
    void testResultIsOptimal() {
        final int nNodes = randomInt(5, 8);
        IGraph g = createEmptyGraph();
        generateConnectedGraph(g, nNodes);
        Node start = someNode(g);

        Boruvka b = new Boruvka(g);
        b.boruvka();
        Set<Edge> edges = b.resultEdgeSet();

        assertEquals(nNodes-1, edges.size());

        double answerWeight = totalWeight(edges);

        List<Edge> allEdges = new ArrayList<>();
        g.getEdges().forEach(allEdges::add);

        Iterator<Set<Edge>> iter =
                    new CombinationsIterator<>(allEdges, nNodes - 1);

        while (iter.hasNext()) {
            Set<Edge> tryEdges = iter.next();

            Set<Node> nodes = reachableNodes(g, tryEdges, start);
            if (nodes.size() != nNodes) {
                continue;
            }

            double weight = totalWeight(tryEdges);
            assertFalse(weight < answerWeight);
        }
    }


    @Test
    void testNoEdges() {
        final int nNodes = 100;
        IGraph g = createEmptyGraph();
        for (int i = 0; i < nNodes; i++) {
            g.addNode();
        }

        Boruvka b = new Boruvka(g);
        b.boruvka();

        assertEquals(nNodes, g.nodesCount());
        assertEquals(0, b.resultEdgeSet().size());
    }

    private void generateComponentGraph(IGraph g, int nNodes, int nComps) {
```

```
            GraphGeneratorFacade gen = new GraphGeneratorFacade();
            gen.generateComponents(g, nNodes, nComps);
      }

      private Set<IGraph.Edge> nodeSetEdges(IGraph g, Set<IGraph.Node> nodes) {
            Set<IGraph.Edge> edges = new HashSet<>();
            for (IGraph.Node n : nodes) {
                  for (IGraph.Edge e : g.getEdgesFrom(n)) {
                        edges.add(e);
                  }
            }
            return edges;
      }

      @RepeatedTest(10)
      void testComponentsResultIsSpanningTree() {
            final int nNodes = randomInt(50, 400);
            final int nComps = randomInt(5, 50);
            IGraph g = createEmptyGraph();
            generateComponentGraph(g, nNodes, nComps);

            Boruvka b = new Boruvka(g);
            b.boruvka();
            Set<Edge> edges = b.resultEdgeSet();

            Set<Edge> realEdges = new HashSet<>();
            g.getEdges().forEach(realEdges::add);

            Set<Node> nodes = new HashSet<>();
            List<Set<Edge>> edgesSets = new ArrayList<>();
            for (Node n : g.getNodes()) {
                  if (!nodes.contains(n)) {
                        Set<Node> comp = reachableNodes(g, edges, n);
                        Set<Node> realComp = reachableNodes(g, realEdges, n);

                        assertEquals(realComp, comp);

                        for (Node m : comp) {
                              assertTrue(nodes.add(m));
                        }
                        Set<Edge> compEdges = nodeSetEdges(g, comp);
                        edgesSets.add(compEdges);
                  }
            }

            assertEquals(nComps, edgesSets.size());

            int totalEdges = edgesSets.stream().mapToInt(Set::size).sum();
            assertEquals(g.edgesCount(), totalEdges);

            // Test that there are no common edges
            for (int i = 0; i < edgesSets.size(); i++) {
                  for (int j = i+1; j < edgesSets.size(); j++) {
                        for (Edge e : edgesSets.get(j)) {
                              assertFalse(edgesSets.get(i).remove(e));
                        }
                  }
            }
      }

}
```

Имя файла: GraphIOTest.java

```
package summer_practice_2020.purple;

import org.junit.jupiter.api.Test;

import java.io.*;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import static org.junit.jupiter.api.Assertions.*;

class GraphIOTest {

	private static IGraph createEmptyGraph() {
		return new Graph();
	}

	@Test
	void testWriteGraph() {
		IGraph g = createEmptyGraph();
		IGraph.Node n1 = g.addNode();
		IGraph.Node n2 = g.addNode();
		IGraph.Node n3 = g.addNode();
		n1.setTitle("A");
		n2.setTitle("B");
		n3.setTitle("C");
		g.addEdge(n1, n2).setWeight(1);
		g.addEdge(n1, n3).setWeight(2);
		g.addEdge(n3, n2).setWeight(3);

		ByteArrayOutputStream s = new ByteArrayOutputStream();
		GraphIO.writeGraph(s, g);

		InputStream is = new ByteArrayInputStream(s.toByteArray());
		BufferedReader r = new BufferedReader(new InputStreamReader(is));
		List<String> lines = new ArrayList<>();
		r.lines().forEach(lines::add);

		int nodes = 0;
		int edges = 0;
		int pos = 0;
		for (String l : lines) {
			if (l.startsWith("node ")) {
				nodes++;
			} else if (l.startsWith("edge ")) {
				edges++;
			} else if (l.startsWith("pos ")) {
				pos++;
			}
		}

		assertEquals(3, nodes);
		assertEquals(nodes, pos);
		assertEquals(3, edges);
	}

	@Test
	void testReadGraph1() {
		final String input =
```

```
                        "node a A\nnode b B\nnode c C\nedge a b 0.75\nedge a c
1.5\n";
            ByteArrayInputStream s = new ByteArrayInputStream(input.getBytes());

            IGraph g = createEmptyGraph();
            GraphIO.readGraph(s, g);

            assertEquals(3, g.nodesCount());
            assertEquals(2, g.edgesCount());

            Map<String, IGraph.Node> nodes = new HashMap<>();
            g.getNodes().forEach(n -> nodes.put(n.getTitle(), n));

            IGraph.Node nA = nodes.get("A");
            assertNotNull(nA);
            IGraph.Node nB = nodes.get("B");
            assertNotNull(nB);
            IGraph.Node nC = nodes.get("C");
            assertNotNull(nC);

            assertEquals(0.75, g.getEdgeBetween(nB, nA).getWeight());
            assertEquals(1.5, g.getEdgeBetween(nA, nC).getWeight());
            assertNull(g.getEdgeBetween(nB, nC));
      }

      // TODO: tests with invalid input

}
```

Имя файла: GraphTest.java

```
package summer_practice_2020.purple;

import org.junit.jupiter.api.Test;
import summer_practice_2020.purple.IGraph.Edge;
import summer_practice_2020.purple.IGraph.Node;

import java.util.*;

import static org.junit.jupiter.api.Assertions.*;

class GraphTest {

      private static final Random rng = new Random();

      private static IGraph createEmptyGraph() {
            return new Graph();
      }

      private static int randomInt(int min, int max) {
            return rng.nextInt(max - min + 1) + min;
      }

      @Test
      void testAddRemoveNode() {
            IGraph g = createEmptyGraph();
            Node n = g.addNode();
            g.removeNode(n);
            assertThrows(NoSuchElementException.class,
                        () -> g.removeNode(n));
      }
```

```
@Test
void testGetNodes() {
	IGraph g = createEmptyGraph();
	final int nNodes = randomInt(100, 200);
	Set<Node> nodes = new HashSet<>();

	for (int i = 0; i < nNodes; i++) {
		Node n = g.addNode();
		if (rng.nextBoolean()) {
			g.removeNode(n);
		} else {
			assertTrue(nodes.add(n));
		}
	}

	assertEquals(nodes.size(), g.nodesCount());

	for (Node n : g.getNodes()) {
		assertTrue(nodes.remove(n));
	}

	assertTrue(nodes.isEmpty());
}

@Test
void testAddEdge() {
	IGraph g = createEmptyGraph();
	Node n1 = g.addNode();
	Node n2 = g.addNode();
	Node n3 = g.addNode();
	Node n4 = g.addNode();
	Node n5 = g.addNode();
	Node n6 = g.addNode();

	g.removeNode(n3);
	g.removeNode(n5);
	g.removeNode(n6);

	g.addEdge(n1, n2);
	assertThrows(IllegalArgumentException.class,
			() -> g.addEdge(n1, n2));
	assertThrows(IllegalArgumentException.class,
			() -> g.addEdge(n2, n1));
	assertThrows(IllegalArgumentException.class,
			() -> g.addEdge(n3, n3));
	assertThrows(NoSuchElementException.class,
			() -> g.addEdge(n3, n4));
	assertThrows(NoSuchElementException.class,
			() -> g.addEdge(n5, n6));
}

@Test
void testRemoveEdge() {
	IGraph g = createEmptyGraph();
	Node n1 = g.addNode();
	Node n2 = g.addNode();

	Edge e = g.addEdge(n1, n2);
	g.removeEdge(e);
	assertThrows(NoSuchElementException.class,
			() -> g.removeEdge(e));
}
```

```
@Test
void testGetEdges() {
	IGraph g = createEmptyGraph();

	List<Node> nodes = new ArrayList<>();
	final int nNodes = randomInt(100, 200);
	for (int i = 0; i < nNodes; i++) {
		nodes.add(g.addNode());
	}

	Set<Edge> edges = new HashSet<>();
	for (int i = 0; i < nNodes; i++) {
		for (int j = i+1; j < nNodes; j++) {
			if (rng.nextBoolean()) {
				continue;
			}

			Edge e = g.addEdge(nodes.get(i), nodes.get(j));
			if (rng.nextBoolean()) {
				g.removeEdge(e);
			} else {
				assertTrue(edges.add(e));
			}
		}
	}

	assertEquals(edges.size(), g.edgesCount());

	for (Edge e : g.getEdges()) {
		assertTrue(edges.remove(e));
	}

	assertTrue(edges.isEmpty());
}

private static <T> boolean presentInIterable(T x, Iterable<T> it) {
	for (T y : it) {
		if (y == x) {
			return true;
		}
	}
	return false;
}

@Test
void testRemoveNodeRemovesEdges() {
	IGraph g = createEmptyGraph();
	Node n1 = g.addNode();
	Node n2 = g.addNode();
	Node n3 = g.addNode();
	Node n4 = g.addNode();

	Edge e12 = g.addEdge(n1, n2);
	Edge e34 = g.addEdge(n3, n4);

	g.removeNode(n1);
	assertFalse(presentInIterable(e12, g.getEdges()));
	assertTrue(presentInIterable(e34, g.getEdges()));
	assertEquals(1, g.edgesCount());

	g.removeNode(n4);
	assertFalse(presentInIterable(e12, g.getEdges()));
	assertFalse(presentInIterable(e34, g.getEdges()));
```

```
	assertEquals(0, g.edgesCount());
}

@Test
void testGetEdgeBetween() {
	IGraph g = createEmptyGraph();
	Node n1 = g.addNode();
	Node n2 = g.addNode();
	Node n3 = g.addNode();
	Edge e = g.addEdge(n1, n2);

	assertEquals(e, g.getEdgeBetween(n1, n2));
	assertEquals(e, g.getEdgeBetween(n2, n1));

	assertNull(g.getEdgeBetween(n1, n3));
	assertNull(g.getEdgeBetween(n3, n1));
	assertNull(g.getEdgeBetween(n2, n3));
	assertNull(g.getEdgeBetween(n3, n2));

	assertNull(g.getEdgeBetween(n1, n1));
	assertNull(g.getEdgeBetween(n2, n2));
	assertNull(g.getEdgeBetween(n3, n3));
}

private static void fillRandomGraph(IGraph g, int nNodes) {
	List<Node> nodes = new ArrayList<>();
	for (int i = 0; i < nNodes; i++) {
		nodes.add(g.addNode());
	}

	for (int i = 0; i < nNodes; i++) {
		for (int j = i+1; j < nNodes; j++) {
			if (rng.nextBoolean()) {
				g.addEdge(nodes.get(i), nodes.get(j));
			}
		}
	}
}

@Test
void testEdgeNodeRefs() {
	IGraph g = createEmptyGraph();
	int nNodes = randomInt(50, 100);
	fillRandomGraph(g, nNodes);

	for (Edge e : g.getEdges()) {
		Node f = e.firstNode();
		Node s = e.secondNode();
		assertEquals(e, g.getEdgeBetween(f, s));
		assertEquals(e, g.getEdgeBetween(s, f));
	}
}

@Test
void testGetEdgesFrom() {
	IGraph g = createEmptyGraph();
	int nNodes = randomInt(50, 100);
	fillRandomGraph(g, nNodes);

	for (Node n : g.getNodes()) {
		Set<Edge> realEdges = new HashSet<>();
		for (Node m : g.getNodes()) {
			Edge e = g.getEdgeBetween(n, m);
```

```
                        if (e != null) {
                              assertTrue(realEdges.add(e));
                        }
                  }

                  for (Edge e : g.getEdgesFrom(n)) {
                        assertTrue(realEdges.remove(e));
                  }

                  assertTrue(realEdges.isEmpty());
            }
      }

}
```

Имя файла: GraphGenerationTest.java

```
package summer_practice_2020.purple.graphgen;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertTrue;

import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Queue;
import java.util.Random;
import java.util.Set;

import org.junit.jupiter.api.RepeatedTest;
import org.junit.jupiter.api.Test;

import summer_practice_2020.purple.Graph;
import summer_practice_2020.purple.IGraph;

class GraphGenerationTest {

      static final Random rng = new Random();

      private static int randomInt(int min, int max) {
            return rng.nextInt(max - min + 1) + min;
      }

      private static IGraph createEmptyGraph() {
            return new Graph();
      }

      private Set<IGraph.Node> reachableNodes(IGraph g, IGraph.Node start) {
            Set<IGraph.Node> s = new HashSet<>();
            s.add(start);
            Queue<IGraph.Node> q = new ArrayDeque<>();
            q.add(start);

            while (!q.isEmpty()) {
                  IGraph.Node i = q.remove();
                  for (IGraph.Edge e : g.getEdgesFrom(i)) {
                        IGraph.Node fn = e.firstNode();
                        IGraph.Node sn = e.secondNode();
                        IGraph.Node o = (fn == i) ? sn : fn;
                        if (s.add(o)) {
                              q.add(o);
```

```
                }
            }
        }

        return s;
    }

    @Test
    void testDividerSpanningTree() {
        final int count = 25;

        IGraph g = createEmptyGraph();
        IGraph.Node n = g.addNode();
        for (int i = 0; i < count-1; i++) {
            g.addNode();
        }

        GraphEdgeGenerator gen = new DividerSpanningTreeEdgeGenerator();
        gen.generateEdges(g);

        assertEquals(count, g.nodesCount());
        assertEquals(count - 1, g.edgesCount());

        Set<IGraph.Node> s = reachableNodes(g, n);
        assertEquals(count, s.size());
    }

    @Test
    void testNoDuplicateEdgesInShuffleGenerator() {
        final int nNodes = 10;

        IGraph g = createEmptyGraph();
        for (int i = 0; i < nNodes; i++) {
            g.addNode();
        }

        // should only generate nNodes*(nNodes-1)/2
        GraphEdgeGenerator gen =
                    new ShuffleGraphEdgeGenerator(nNodes*(nNodes-1));

        gen.generateEdges(g);

        assertEquals(nNodes*(nNodes-1)/2, g.edgesCount());
    }

    @Test
    void testTwoSubgraphs() {
        final int size1 = 20;
        final int size2 = 20;

        IGraph g = createEmptyGraph();

        List<IGraph.Node> nodes1 = new ArrayList<>();
        for (int i = 0; i < size1; i++) {
            nodes1.add(g.addNode());
        }

        List<IGraph.Node> nodes2 = new ArrayList<>();
        for (int i = 0; i < size2; i++) {
            nodes2.add(g.addNode());
        }

        GraphEdgeGenerator gen = new DividerSpanningTreeEdgeGenerator();
```

```
            gen.generateEdgesOnNodes(g, nodes1);
            gen.generateEdgesOnNodes(g, nodes2);

            assertEquals(size1 + size2 - 2, g.edgesCount());

            Set<IGraph.Node> s1 = reachableNodes(g, nodes1.get(0));
            assertEquals(size1, s1.size());

            Set<IGraph.Node> s2 = reachableNodes(g, nodes2.get(0));
            assertEquals(size2, s2.size());
      }

      private List<Set<IGraph.Node>> getNodeSets(IGraph g) {
            Set<IGraph.Node> visitedNodes = new HashSet<>();
            List<Set<IGraph.Node>> nodeSets = new ArrayList<>();

            for (IGraph.Node n : g.getNodes()) {
                  if (visitedNodes.contains(n)) {
                        continue;
                  }

                  Set<IGraph.Node> nodes = reachableNodes(g, n);
                  nodeSets.add(nodes);

                  for (IGraph.Node m : nodes) {
                        assertTrue(visitedNodes.add(m));
                  }
            }

            return nodeSets;
      }

      @RepeatedTest(10)
      void testSubgraphsFromFacade() {
            final int nodesCount = randomInt(50, 400);
            final int compsCount = randomInt(5, 50);

            IGraph g = createEmptyGraph();
            GraphGeneratorFacade gen = new GraphGeneratorFacade();
            gen.generateComponents(g, nodesCount, compsCount);

            assertEquals(nodesCount, g.nodesCount());

            List<Set<IGraph.Node>> nodeSets = getNodeSets(g);
            assertEquals(compsCount, nodeSets.size());
            assertEquals(nodesCount,
nodeSets.stream().mapToInt(Set::size).sum());
      }

      private Set<IGraph.Edge> nodeSetEdges(IGraph g, Set<IGraph.Node> nodes) {
            Set<IGraph.Edge> edges = new HashSet<>();
            for (IGraph.Node n : nodes) {
                  for (IGraph.Edge e : g.getEdgesFrom(n)) {
                        edges.add(e);
                  }
            }
            return edges;
      }

      @RepeatedTest(10)
      void testComponentsWithNEdges() {
            final int nodesCount = randomInt(100, 200);
            final int compsCount = randomInt(3, 10);
```

```
            final int moreEdges = randomInt(1, 200);

            final int edgesCount = nodesCount - compsCount + moreEdges;

            IGraph g = createEmptyGraph();
            GraphGeneratorFacade gen = new GraphGeneratorFacade();
            gen.generateComponentsWithNEdges(g, nodesCount, edgesCount,
compsCount);

            assertEquals(nodesCount, g.nodesCount());
            assertEquals(edgesCount, g.edgesCount());

            List<Set<IGraph.Node>> nodeSets = getNodeSets(g);
            assertEquals(compsCount, nodeSets.size());

            for (Set<IGraph.Node> nodes : nodeSets) {
                  Set<IGraph.Edge> edges = nodeSetEdges(g, nodes);
                  int size = edges.size();
                  assertTrue(size <= nodes.size() - 1 + moreEdges);
            }
      }

      @Test
      void testNoDuplicateEdgesInSimpleGenerator() {
            final int nNodes = 10;

            IGraph g = createEmptyGraph();
            for (int i = 0; i < nNodes; i++) {
                  g.addNode();
            }

            // should only generate nNodes*(nNodes-1)/2
            GraphEdgeGenerator gen = new SimpleGraphEdgeGenerator(1.0);

            gen.generateEdges(g);

            assertEquals(nNodes*(nNodes-1)/2, g.edgesCount());
      }

      @Test
      void testAlphabetNameGenerator() {
            final int alpha_size = 26;

            GraphNodeNameGenerator gen = new AlphabetNodeNameGenerator();
            assertEquals("A", gen.generateName());

            final int skip = 15;
            for (int i = 0; i < skip; i++) {
                  gen.generateName();
            }

            assertEquals(String.valueOf((char)('A'+1+skip)),
gen.generateName());

            final int skip2 = alpha_size-2-skip;
            for (int i = 0; i < skip2; i++) {
                  gen.generateName();
            }

            assertEquals("AA", gen.generateName());

            for (int i = 0; i < alpha_size-1; i++) {
                  gen.generateName();
```

```
            }

            assertEquals("BA", gen.generateName());

            final int skip3 = alpha_size * (alpha_size-1) - 2;
            for (int i = 0; i < skip3; i++) {
                  gen.generateName();
            }

            assertEquals("ZZ", gen.generateName());
            assertEquals("AAA", gen.generateName());
      }

}
```