Санкт-Петербургский государственный
Электротехнический университет «ЛЭТИ»

Кафедра МО ЭВМ

Отчет по курсовой работе
по дисциплине «Построение и анализ алгоритмов»

Тема: «Алгоритмы на графах. Алгоритм поиска Минимального Остовного Дерева (МОД) методом Борувки»

Выполнил: Ефремов М. А.
Группа: 2304

Преподаватель: Балтрашевич В.Э.

Санкт-Петербург
2014

1) Постановка задачи:

Пусть сгенерирован граф G(V, E), где V – сгенерированное множество вершин с заданным наперед количеством n, E – сгенерированное множество ребер с заданным наперед количеством m с весами от minRand до maxRand. Требуется методом Борувки найти МОД, последовательность входящих в него ребер.

2) Постановка и реализация алгоритма:

Особенность этого алгоритма в скорости его выполнения – O(m*log(n))! В отличии, например, от жадного алгоритма стоимостью O(m*log(m)). В алгоритме Борувки мы перебираем m ребер и за каждый такт основного цикла уменьшаем количество компонент связности минимум в два раза! Для графов 100-500 задача решается, в среднем, за 3 шага.

Для начала, поясним алгоритм рандомизации графа:

1. Сначала генерируем массив вершин с заданного количества.
2. Потом, генерируем $\frac{n(n-1)}{2}$ ребер (как для полного графа), а потом случайным образом убираем их до нужного количества m.

Таким простым способом генерируется исходный граф. Алгоритм Борувки же заключается в следующем:

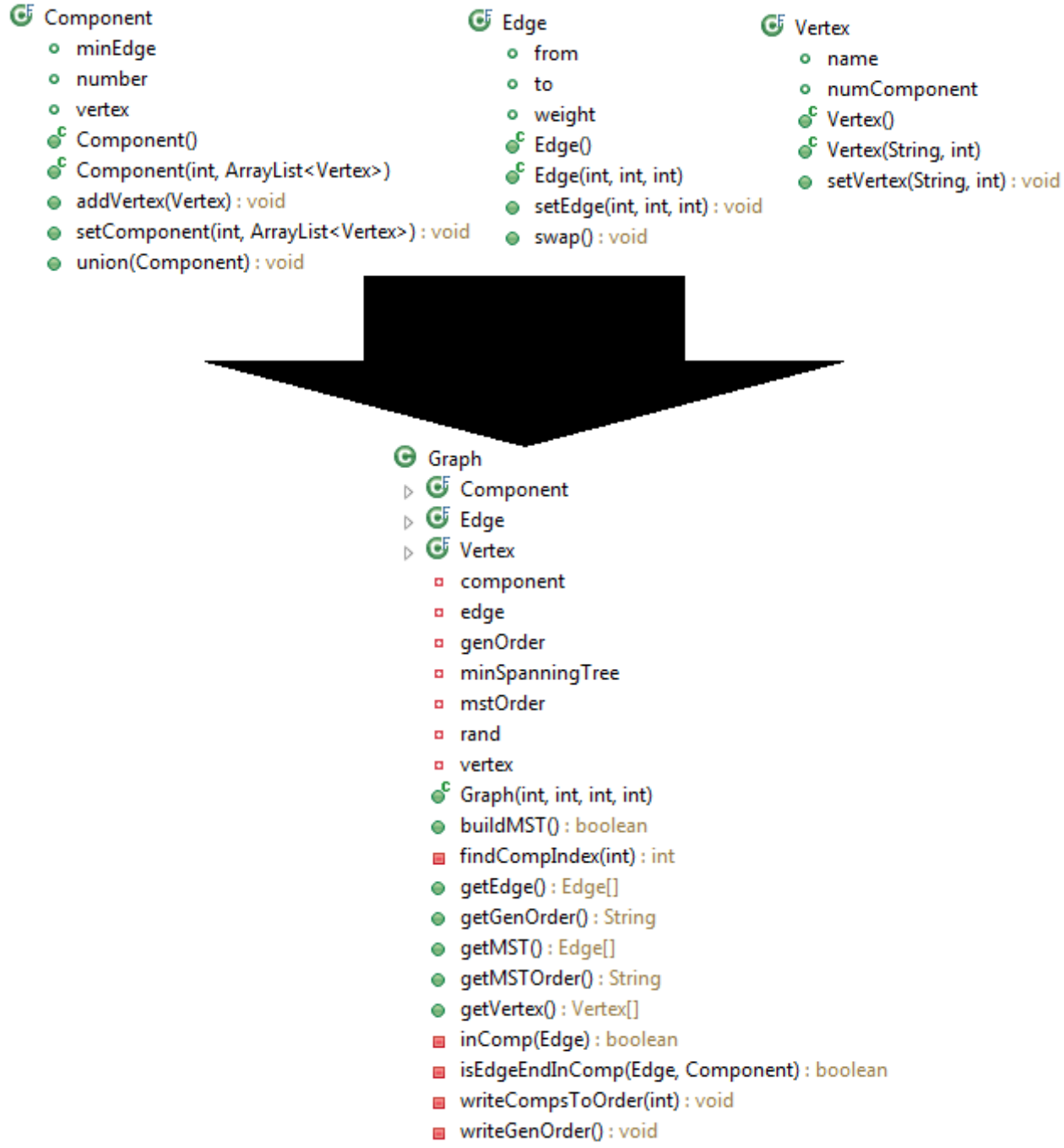
1. Делаем в начале алгоритма каждую вершину отдельным множеством.
2. После, в цикле, проходимся по всем ребрам. Внутри, проверяем наши множества: если текущее ребро входит одной вершиной в данное множество, то, сравнив с минимальным выходящим из текущего множества ребром, обмениваем, если оно меньше минимального.
3. Пройдясь по всем ребрам, мы получили список из минимальных ребер, выходящих из одних множеств в другие. Объединяем эти множества между собой по этим ребрам.
4. Продолжаем с п.2 до тех пор, пока не останется одного множества.

К слову, в случае, если в п.3 у какого-либо множества нету выходящего ребра, то это является отдельной компонентой связности. Как следствие, граф не связный и задача нахождения МОД для него некорректна.

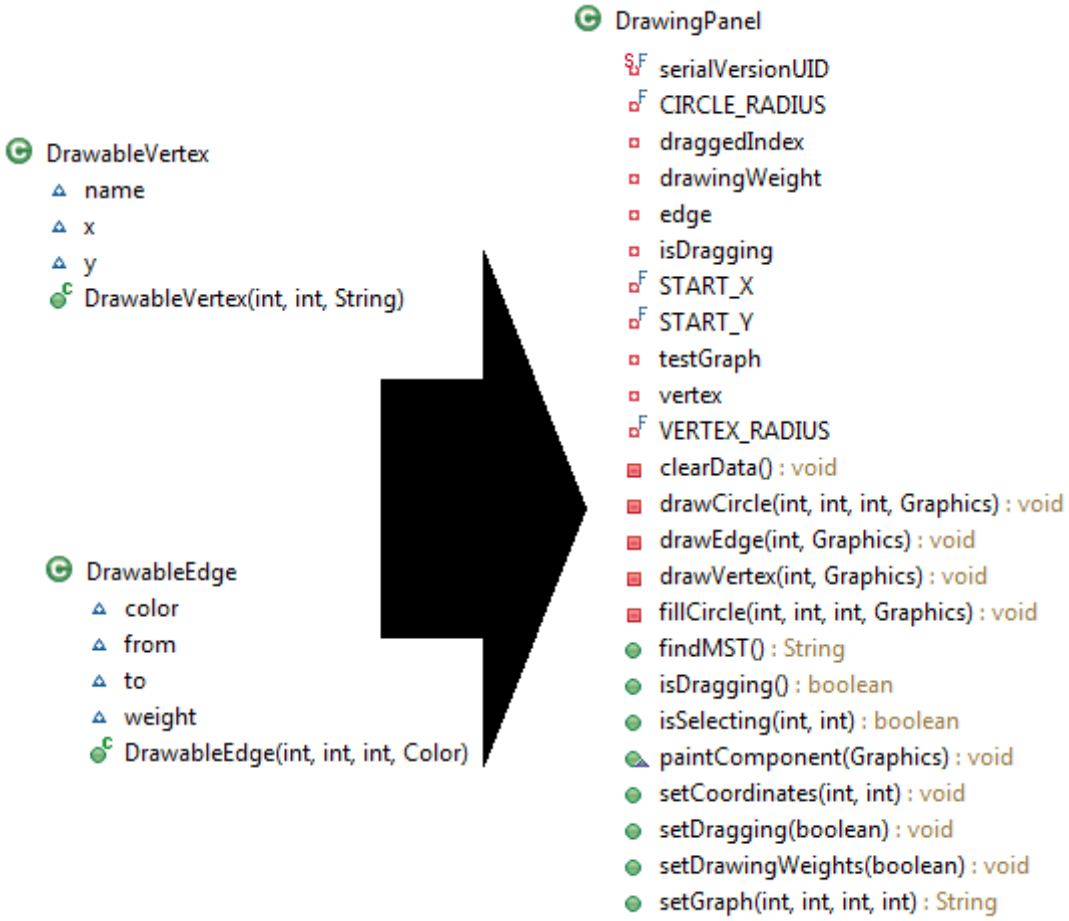
3) Структуры данных:

Ниже приведены классы, их поля, методы и возвращаемые значения. Назначения функций и полей подписаны в тексте программы

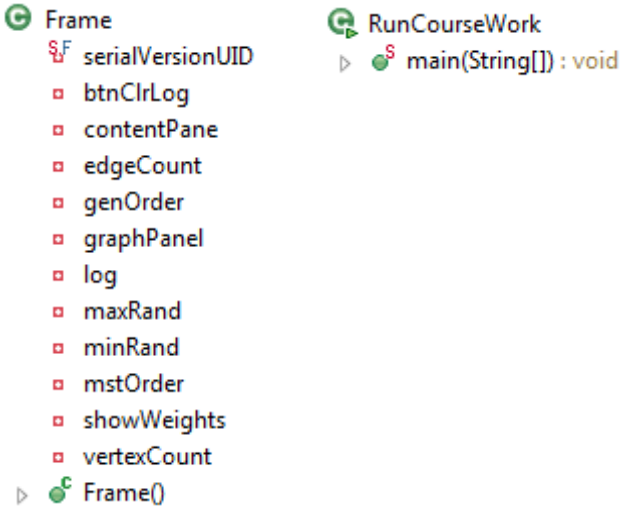
Класс графа:



Класс панели отрисовки графа:



Класс основного фрейма и класс с методом main:



4) Текст программы:

Graph.java

Класс, содержащий граф и решающий поставленную задачу

```
package Graph;

import java.util.ArrayList;
import java.util.Random;
/**
 *
 * @author Akutenshi
 * Неориентрованный, связный, взвешенный граф
 *
 */
public class Graph {
	//Вершина
	public final class Vertex {
		public String name;
		public int numComponent;

		public Vertex() {
			name = "default";
			numComponent = 0;
		}

		public Vertex(String name, int numComponent) {
			setVertex(name, numComponent);
		}

		public void setVertex(String name, int numComponent) {
			this.name = name;
			this.numComponent = numComponent;
		}
	}
	//Ребро
	public final class Edge {
		public int from;
		public int to;
		public int weight;

		public Edge() {
			from = 0;
			to = 0;
			weight = 0;
		}

		public Edge(int from, int to, int weigth) {
			setEdge(from, to, weigth);
		}

		public void setEdge(int from, int to, int weigth) {
```

```
                this.from = from;
                this.to = to;
                this.weight = weigth;
            }

            public void swap() {
                int buffer = to;
                to = from;
                from = buffer;
            }
        }
        //Компонента связности
        public final class Component {
            public int number;
            public ArrayList<Vertex> vertex;
            public Edge minEdge;

            public Component() {
                number = 0;
                vertex = new ArrayList<Vertex>(0);
            }

            public Component(int number, ArrayList<Vertex> vertex) {
                this.setComponent(number, vertex);
            }

            public void setComponent(int number, ArrayList<Vertex> vertex)
{
                this.number = number;
                this.vertex = vertex;
            }

            public void addVertex(Vertex vertex) {
                this.vertex.add(vertex);
            }

            public void union(Component in) {
                //Все вершины присоединяемого множества помечаем членами
этого
                // и добавляем в конец списка вершин этого компонента
                for (int i = 0; i < in.vertex.size(); i++) {
                    in.vertex.get(i).numComponent = this.number;
                    this.vertex.add(in.vertex.get(i));
                }
            }
        }
        //Граф
        private Vertex[] vertex = null;
        private Edge[] edge = null;
        //Генератор случайных чисел для конструктора
        private Random rand = new Random();
```

```
	//Контейнеры для алгоритма Боровки:
	// 1) Компоненты связности
	private ArrayList<Component> component = null;
	// 2) МОД
	private ArrayList<Edge> minSpanningTree = new ArrayList<Edge>(0);
	//отчеты:
	// 1) генерации графа
	private String genOrder =
"------------------------------------------------\n";
	// 2) нахождения МОД
	private String mstOrder = "\nFinding MST:\n";
	/**
	 * Конструктор с рандомизированием графа по заданным параметрам:
	 * 1) Количество вершин
	 * 2) Количество ребер
	 * Вес ребер и их инцедентность определяется случайно. Первое в
	 * интервале [minRand, maxRand], второе - любая вариация ребер, в
количестве
	 * не менее 0 и не более кол-ва ребер полного графа
	 */
	public Graph(int vertexCount, int edgeCount, int minRand, int
maxRand) {
		//проверка, что ребер <= чем в полном графе и вес
положительный
		int maxEdge = vertexCount * (vertexCount - 1) / 2;
		if (0 < edgeCount && edgeCount <= maxEdge && minRand > 0 &&
vertexCount > 0) {
			vertex = new Vertex[vertexCount];
			edge = new Edge[edgeCount];

			//заполнение вершин
			for (int i = 0; i < vertex.length; i++) {
				vertex[i] = new Vertex(Integer.toString(i), i);
			}
			//заполнение ребер
			//Делаем до тех пор, пока граф не связен
			//строим ребра для полного графа
			ArrayList<Edge> full = new ArrayList<Edge>(0);
			int f = 0; //from
			while (f != vertex.length - 1) {
				//j - to
				for (int j = f + 1; j <= vertex.length - 1; j++) {
					full.add(new Edge(f, j, rand.nextInt(maxRand
- minRand) + minRand));
				}
				f++;
			}
			for (int i = 1; i <= maxEdge - edgeCount; i++) {
				full.remove(rand.nextInt(full.size()));
			}
			for (int i = 0; i < edge.length; i++) {
```

```
                    edge[i] = full.get(i);
                }
                full.clear();
                //пишем отчет
                writeGenOrder();
            }
        }
        //Алгоритм Боровки по поиску МОД
        // Ответ хранится в списке minSpanningTree
        public boolean buildMST() {
            //Если решение уже найдено - выходим
            if (minSpanningTree.size() != 0) {
                return true;
            }
            //Инициируем пустое МОД
            minSpanningTree.clear();
            //Инициализируем список компонент связности - каждая вершина
            // есть отдельная компонента связности
            component = new ArrayList<Component>(0);
            ArrayList<Vertex> singleComp;
            for (int i = 0; i < vertex.length; i++) {
                singleComp = new ArrayList<Vertex>(0);
                singleComp.add(vertex[i]);
                component.add(new Component(i, singleComp));
            }

            Component attach;
        //объединяющий
            int attachIndex;
            Component attachable;
        //включаемое
            int attachableIndex;
            Component current;
        //Текущий компонент
            int[] minCompWeight = new int[component.size()]; //минимальные
веса
            Edge[] minEdge;
        //Массив минимальных ребер
            int step = 1;
        //Номер шага главного цикла
            //Главный цикл
            while (component.size() != 1) {
                //минимумы для компонент ставим максимально большими
                for (int i = 0; i< component.size(); i++) {
                    minCompWeight[i] = Integer.MAX_VALUE;
                }
                //чистим массив минимальных ребер
                minEdge = new Edge[component.size()];

                //Перебираем ребра и проверяем на вхождение их в
компоненты
```

```
                for (int i = 0; i < edge.length; i++) {
                      for (int j = 0; j < component.size(); j++) {
                            current = component.get(j);
                            //Если ребро лежит одной вершиной в j-й
компоненте, то сравниваем с его минимумом
                            if (isEdgeEndInComp(edge[i], current)) {
                                  if (edge[i].weight < minCompWeight[j]) {
                                        minCompWeight[j] = edge[i].weight;
                                        minEdge[j] = edge[i];
                                  }
                            }
                      }
                }
                //записываем компоненты связности в отчет
                writeCompsToOrder(step);
                //Объединяем множества
                for (int i = 0; i < minEdge.length; i++) {
                      if (minEdge[i] == null) {
                            mstOrder += (" Component #" + i + " is
isolated!\n");
                            return false;
                      }
                      if (!inComp(minEdge[i])) {
                            minSpanningTree.add(minEdge[i]);
                            attachIndex =
findCompIndex(vertex[minEdge[i].from].numComponent);
                            attach = component.get(attachIndex);
                            attachableIndex =
findCompIndex(vertex[minEdge[i].to].numComponent);
                            attachable = component.get(attachableIndex);

                            //пишем объединение в лог
                            mstOrder += (" " + attach.number + " with " +
attachable.number);
                            mstOrder += (" by edge " +
vertex[minEdge[i].from].name +
                                        "-" + vertex[minEdge[i].to].name +
                                        " weight = " + minEdge[i].weight +
"\n");

                            attach.union(attachable);
                            component.remove(attachableIndex);
                      }
                }
                //переходим к следующему шагу
                step++;
          }


          mstOrder += ("MST:\n");
          for (int i = 0; i < minSpanningTree.size(); i++) {
```

```
                mstOrder += (minSpanningTree.get(i).from + "-" +
                            minSpanningTree.get(i).to + " weight = " +
minSpanningTree.get(i).weight + "\n");
           }

           return true;
     }
     //Лежит ли ребро e в одной комопненте связности
     private boolean inComp(Edge e) {
           return (vertex[e.from].numComponent ==
vertex[e.to].numComponent);
     }
     //is the Edge's end in Component
     private boolean isEdgeEndInComp(Edge e, Component c) {
           if (!inComp(e)
                && (vertex[e.from].numComponent == c.number
                || vertex[e.to].numComponent == c.number)) {
                return true;
           } else {
                return false;
           }
     }
     //Найти индекс в списке компонент связности с данной меткой num
     private int findCompIndex(int num) {
           for (int i = 0; i < component.size(); i++) {
                if (component.get(i).number == num) {
                     return i;
                }
           }
           return -1;
     }

     //Заносит в отчет поиска МОД текущие компоненты связности
     private void writeCompsToOrder(int step) {
           mstOrder += ("Step #" + step + ":\n");
           mstOrder += ("Count of connected sets = " + component.size() +
"\n");
           Component current;
           for (int i = 0; i < component.size(); i++) {
                current = component.get(i);
                mstOrder += (" " + i + ") Mark of component: " +
current.number + " | Vertex:");
                for (int j = 0; j < current.vertex.size(); j++) {
                     mstOrder += (" " + current.vertex.get(j).name);
                }
                mstOrder += (";\n");
           }
           mstOrder += ("Unions:\n");
     }

     //Пишет отчет генерации графа
```

```
	private void writeGenOrder() {
		//информация о вершинах
		genOrder += ("New graph generated!\n");
		genOrder += ("Vertexes count = " + vertex.length + "\n");
		genOrder += ("Vertexes: ");
		for (int i = 0; i < vertex.length; i++) {
			genOrder += (vertex[i].name + ", ");
		}
		//информация о ребрах
		genOrder += ("\n Edges count = " + edge.length + "\n");
		genOrder += ("Edges:\n");
		for (int i = 0; i < edge.length; i++) {
			genOrder += (vertex[edge[i].from].name + "-" +
					vertex[edge[i].to].name + " weight = " +
edge[i].weight + "\n");
		}

	}

	//Возвращает отчет поиска МОД
	public String getMSTOrder() {
		return mstOrder;
	}

	//Возвращает отчет генерации
	public String getGenOrder() {
		return genOrder;
	}

	//Возвращает вершины
	public Vertex[] getVertex() {
		return vertex;
	}

	//Возвращает ребро
	public Edge[] getEdge() {
		return edge;
	}

	//Возвращает МОД
	public Edge[] getMST() {
		Edge[] mst = new Edge[minSpanningTree.size()];
		for (int i = 0; i < minSpanningTree.size(); i++) {
			mst[i] = minSpanningTree.get(i);
		}
		return mst;
	}
}
```

Панель, отрисовывающая данный граф

```
package Window;

import java.awt.Color;
import java.awt.Graphics;
import java.lang.Math;
import javax.swing.JPanel;

import Graph.Graph;

public class DrawingPanel extends JPanel {
	private static final long serialVersionUID = 1L;
	private final int VERTEX_RADIUS = 15;
	private final int CIRCLE_RADIUS = 240;
	private final int START_X = 300;
	private final int START_Y = 270;
	//рабочий граф
	private Graph testGraph = null;  //вершины для отрисовки
	public class DrawableVertex {
		int x;
		int y;
		String name;

		public DrawableVertex(int x, int y, String name) {
			this.x = x;
			this.y = y;
			this.name = name;
		}
	}
	//ребра для отрисовки
	public class DrawableEdge {
		int from;
		int to;
		int weight;
		Color color;
		public DrawableEdge(int from, int to, int weight, Color color)
{
			this.from = from;
			this.to = to;
			this.weight = weight;
			this.color = color;
		}
	}
	//рисуемый граф
	private DrawableVertex[] vertex;
	private DrawableEdge[] edge;
	private boolean drawingWeight = false;//рисовать ли веса
	private int draggedIndex = -1;   //индеекс перемещаемой точки
	private boolean isDragging = false;   //осуществляется ли
```

```
перетаскивание вершины
	//перегружаем перерисовку панели
	@Override
	public void paintComponent(Graphics g) {
		super.paintComponent(g);
		if (testGraph != null && vertex != null && vertex.length > 0)
{
			for (int i = 0; i < edge.length; i++) {
				this.drawEdge(i, g);
			}
			for (int i = 0; i < vertex.length; i++) {
				this.drawVertex(i, g);
			}
		}
	}

	//генерируем граф
	public String setGraph(int vertexCount, int edgeCount, int minRand,
int maxRand) {
		this.clearData();
		this.repaint();
		String genOrder = "";
		if (0 < edgeCount && edgeCount <= vertexCount * (vertexCount -
1) / 2
				&& minRand > 0
				&& vertexCount > 0) {
			testGraph = new Graph(vertexCount, edgeCount, minRand,
maxRand);
			Graph.Vertex currentVertex;
			Graph.Vertex[] tempVert = testGraph.getVertex();
			vertex = new DrawableVertex[tempVert.length];
			float dAlpha = (float) (2 * Math.PI / vertex.length);
			float alpha = 0f;
			int x;
			int y;
			for (int i = 0; i < tempVert.length; i++) {
				currentVertex = tempVert[i];
				x = (int) (START_X +
Math.sin(alpha)*CIRCLE_RADIUS);
				y = (int) (START_Y +
Math.cos(alpha)*CIRCLE_RADIUS);
				vertex[i] = new DrawableVertex(x, y,
currentVertex.name);
				alpha += dAlpha;
			}
			Graph.Edge currentEdge;
			Graph.Edge[] tempEdge = testGraph.getEdge();
			edge = new DrawableEdge[tempEdge.length];
			for (int i = 0; i < tempEdge.length; i++) {
				currentEdge = tempEdge[i];
				edge[i] = new DrawableEdge(currentEdge.from,
```

```
                                    currentEdge.to,
                                    currentEdge.weight,
                                    Color.BLACK);
                        }
                        this.repaint();
                        genOrder = testGraph.getGenOrder();
                  } else {
                        genOrder +=
("-----------------------------------------------\n"
                                    + "Generation error!\n"
                                    + "Check edges count (0 < count <= n(n -
1)/2)\n,"
                                    + "vertex count > 0"
                                    + "and minimal weight > 0\n");
                  }
                  return genOrder;
            }
            //ищем МОД
            public String findMST() {
                  String mstOrder = "";
                  //Если граф не загружен
                  if (testGraph == null) {
                        return "\nGraph is not loadded!\n";
                  }
                  //иначе ищем МОД
                  if (testGraph.buildMST()) {
                        //Если нашлось
                        mstOrder += (testGraph.getMSTOrder() +
                                    "Solution is find!\n");
                        //делаем ребра из МОД красными
                        Graph.Edge[] checkMST = testGraph.getMST();
                        for (int i = 0; i < edge.length; i++) {
                              for (int j = 0; j < checkMST.length; j++) {
                                    if (edge[i].from == checkMST[j].from &&
                                                edge[i].to == checkMST[j].to) {
                                          edge[i].color = Color.RED;
                                    }
                              }
                        }
                        this.printComponent(this.getGraphics());
                  } else {
                        //иначе выводим, что граф не связен
                        mstOrder += (testGraph.getMSTOrder() +
                                    "Graph have more then 1 connected components
and haven't solution!\n");
                  };
                  return mstOrder;
            }
            //отрисовка вершины
            private void drawVertex(int i, Graphics g) {
                  g.setColor(Color.WHITE);
```

```
            fillCircle(vertex[i].x, vertex[i].y, VERTEX_RADIUS, g);
            g.setColor(Color.BLACK);
            drawCircle(vertex[i].x, vertex[i].y, VERTEX_RADIUS, g);
            g.drawString(vertex[i].name,
                        vertex[i].x - 5,
                        vertex[i].y + 5);
      }
      //отрисовка ребра
      private void drawEdge(int i, Graphics g) {
            int fromX = vertex[edge[i].from].x;
            int fromY = vertex[edge[i].from].y;
            int toX = vertex[edge[i].to].x;
            int toY = vertex[edge[i].to].y;
            g.setColor(edge[i].color);
            g.drawLine(fromX, fromY, toX, toY);
            if (drawingWeight) {
                  int middleX = (fromX + toX) / 2;
                  int middleY = (fromY + toY) / 2;
                  g.drawString(Integer.toString(edge[i].weight),
                              middleX, middleY);
            }

      }
      //Определение индекса передвигаемой вершины
      public boolean isSelecting(int x, int y) {
            boolean isSelecting = false;
            for (int i = 0; i < vertex.length; i++) {
                  if ( Math.pow(x - vertex[i].x, 2) + Math.pow(y -
vertex[i].y, 2) < Math.pow(VERTEX_RADIUS, 2)) {
                        draggedIndex = i;
                        isSelecting = true;
                        break;
                  }
            }
            return isSelecting;
      }
      //передает координаты передвигаемой вершине
      public void setCoordinates(int x, int y) {
            vertex[draggedIndex].x = x;
            vertex[draggedIndex].y = y;
            this.repaint();
      }
      //передвижение вершины окончено
      public void setDragging(boolean isDragging) {
            this.isDragging = isDragging;
      }
      public boolean isDragging() {
            return isDragging;
      }
      //отрисовка окружности
      private void drawCircle(int x, int y, int r, Graphics g) {
```

```
            g.drawOval(x - r, y - r, r * 2, r * 2);
      }
      //отрисовка круга
      private void fillCircle(int x, int y, int r, Graphics g) {
            g.fillOval(x - r, y - r, r * 2, r * 2);
      }
      //делать ли подписи весов рядом с ребрами
      public void setDrawingWeights(boolean in) {
            drawingWeight = in;
            this.repaint();
      }
      //очистка данных
      private void clearData() {
            vertex = null;
            edge = null;
            draggedIndex = -1;
            testGraph = null;
      }
}
```

Frame.java

Класс основного фрейма

```
package Window;
import javax.swing.JFrame;

public class Frame extends JFrame {
      private static final long serialVersionUID = 1L;
      private JPanel contentPane;
      private JTextField vertexCount;
      private JTextField edgeCount;
      private JTextField minRand;
      private JTextField maxRand;
      private DrawingPanel graphPanel;
      private JTextArea log;
      private JButton btnClrLog;
      private String genOrder;
      private String mstOrder;
      private JCheckBox showWeights;

      public Frame() {
            setTitle("Course Work");
            setResizable(false);
            setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            setBounds(100, 100, 1051, 589);
            contentPane = new JPanel();
            contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
            setContentPane(contentPane);
            contentPane.setLayout(null);
```

```
		JButton btnRand = new JButton("Rand");
		btnRand.addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent arg0) {
				genOrder =
graphPanel.setGraph(Integer.parseInt(vertexCount.getText()),
						Integer.parseInt(edgeCount.getText()),
						Integer.parseInt(minRand.getText()),
						Integer.parseInt(maxRand.getText()));
				log.append(genOrder);
			}
		});
		btnRand.setBounds(958, 427, 75, 23);
		contentPane.add(btnRand);

		vertexCount = new JTextField();
		vertexCount.setText("5");
		vertexCount.setBounds(758, 442, 86, 20);
		contentPane.add(vertexCount);
		vertexCount.setColumns(10);

		edgeCount = new JTextField();
		edgeCount.setText("7");
		edgeCount.setBounds(854, 442, 86, 20);
		contentPane.add(edgeCount);
		edgeCount.setColumns(10);

		JLabel lblVertexCount = new JLabel("Vertexes count:");
		lblVertexCount.setBounds(758, 417, 86, 14);
		contentPane.add(lblVertexCount);

		JLabel lblEdgeCount = new JLabel("Edges count:");
		lblEdgeCount.setBounds(854, 417, 86, 14);
		contentPane.add(lblEdgeCount);

		graphPanel = new DrawingPanel();
		graphPanel.addMouseMotionListener(new MouseMotionAdapter() {
			@Override
			public void mouseDragged(MouseEvent e) {
				if (graphPanel.isDragging()) {
					graphPanel.setCoordinates(e.getX(),
e.getY());
				}
			}
		});
		graphPanel.addMouseListener(new MouseAdapter() {
			@Override
			public void mouseReleased(MouseEvent e) {
				graphPanel.setDragging(false);
			}
			@Override
			public void mousePressed(MouseEvent e) {
```

```
            //Если попали в вершину, то двигаем
            if (graphPanel.isSelecting(e.getX(), e.getY())) {
                  graphPanel.setDragging(true);
            }
      }
});
graphPanel.setBounds(10, 11, 738, 541);
//contentPane.add(graphPanel);
JScrollPane panelScroll = new JScrollPane(graphPanel);
panelScroll.setBounds(graphPanel.getBounds());
contentPane.add(panelScroll);

log = new JTextArea();
log.setEditable(false);
log.setLineWrap(true);
log.setBounds(758, 11, 257, 395);

JScrollPane textScroll = new JScrollPane(log);
textScroll.setBounds(new Rectangle(758, 11, 275, 395));
contentPane.add(textScroll);

JButton btnNewButton = new JButton("MST");
btnNewButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
            mstOrder = graphPanel.findMST();
            log.append(mstOrder);
      }
});
btnNewButton.setBounds(958, 461, 75, 23);
contentPane.add(btnNewButton);

JLabel lblMinWeight = new JLabel("min Weight >0:");
lblMinWeight.setBounds(758, 470, 86, 14);
contentPane.add(lblMinWeight);

JLabel lblMaxWeight = new JLabel("max Weight");
lblMaxWeight.setBounds(854, 473, 86, 14);
contentPane.add(lblMaxWeight);

minRand = new JTextField();
minRand.setText("1");
minRand.setBounds(758, 495, 86, 20);
contentPane.add(minRand);
minRand.setColumns(10);

maxRand = new JTextField();
maxRand.setText("10");
maxRand.setBounds(854, 495, 86, 20);
contentPane.add(maxRand);
maxRand.setColumns(10);
```

```
            btnClrLog = new JButton("Clr log");
            btnClrLog.addActionListener(new ActionListener() {
                  public void actionPerformed(ActionEvent arg0) {
                        log.setText("");
                  }
            });
            btnClrLog.setBounds(958, 494, 75, 23);
            contentPane.add(btnClrLog);

            showWeights = new JCheckBox("Show weights");
            showWeights.addMouseListener(new MouseAdapter() {
                  @Override
                  public void mouseClicked(MouseEvent arg0) {

      graphPanel.setDrawingWeights(showWeights.isSelected());
                  }
            });

            showWeights.setBounds(758, 522, 106, 23);
            contentPane.add(showWeights);

            JLabel lblAutorEfremovMa = new JLabel("Author: Efremov M.A
#2304 LETI 2014");
            lblAutorEfremovMa.setForeground(Color.LIGHT_GRAY);
            lblAutorEfremovMa.setBounds(824, 547, 221, 14);
            contentPane.add(lblAutorEfremovMa);
      }
}
```

RunCourseWork.java

Класс с методом main

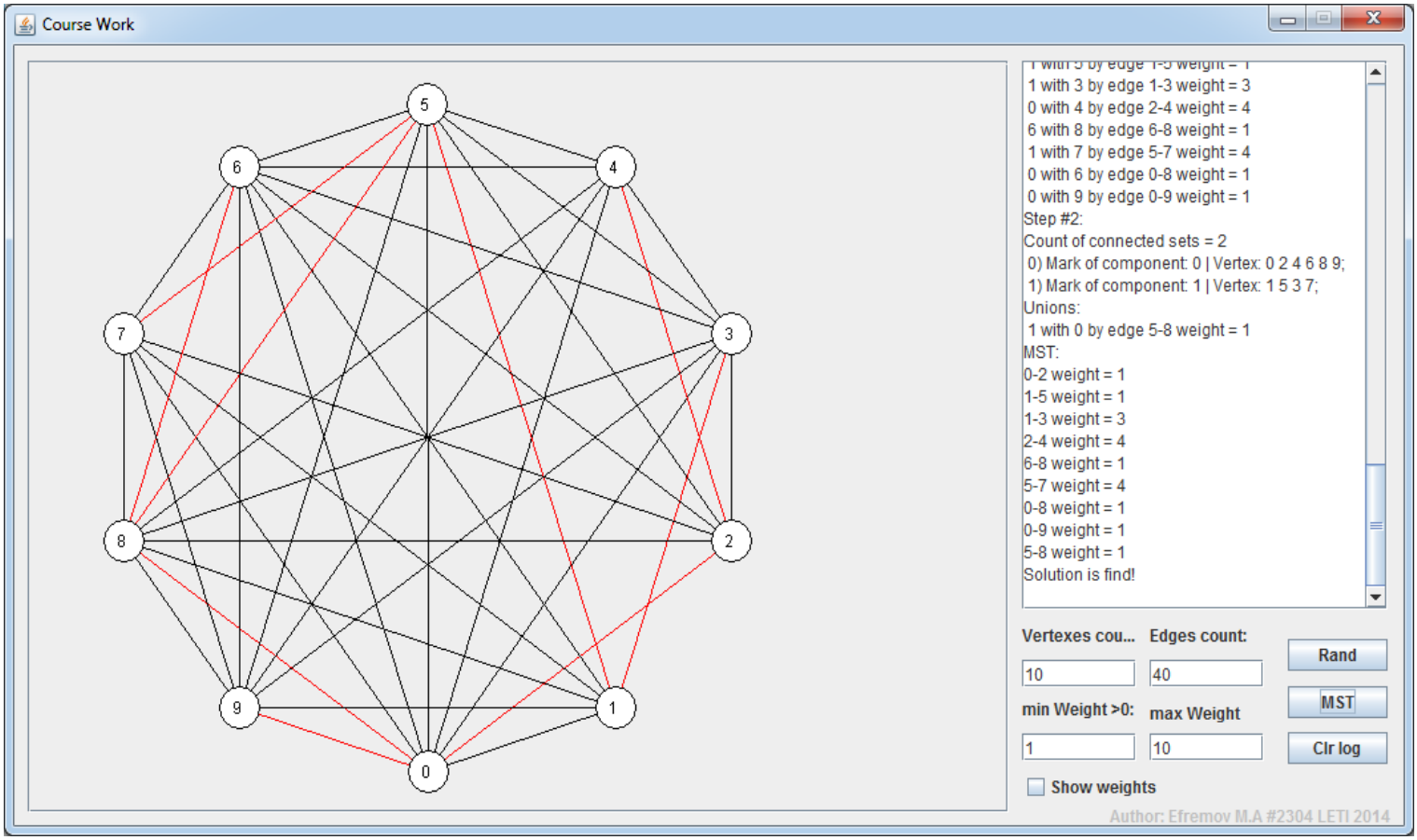
```
package Window;

import java.awt.EventQueue;

public class RunCourseWork {

      public static void main(String[] args) {
            EventQueue.invokeLater(new Runnable() {
                  public void run() {
                        try {
                              Frame frame = new Frame();
                              frame.setVisible(true);
                        } catch (Exception e) {
                              e.printStackTrace();
                        }
                  }
            });
      }
}
```

5) Пример выполнения:



6) Вывод:

Алгоритм работает очень быстро, по принципу медленный поиск – быстрое объединение. Занимает среднюю нишу между алгоритмами Крускала и Прима-Дейкстры. В ходе семестра, я так же начал осваивать язык Java, на котором и писал свои работы. В частности, хотелось бы отметить работы с фреймворком Swing, на котором писался графический интерфейс.

7) Список литературы:

1. Ивановский С.А. Лекционные презентации. Лекция №8, с33-с.39
2. К. Хорстманн, Г. Корнелл. Java 2. Том 2. Основы. ISBN 978-5-8459-1378-4
3. Java docs http://docs.oracle.com/javase/7/docs/api/