**Санкт-Петербургский**

**Государственный электротехнический университет**

**Факультет Компьютерных Технологий и Информатики**

**Кафедра МО ЭВМ**

Отчет

по курсовой работе по дисциплине

***"Алгоритмы и структуры данных"*** на тему *"****Построение минимального остовного дерева. Алгоритм Прима****"*

**Выполнил:** Скорюков М. С.

**Группа #** 2304

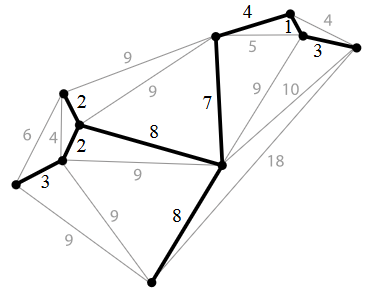
**Проверил:** Балтрашевич В. Э.

***Санкт-Петербург***

***2014 год.***

**Постановка задачи:** спроектировать систему классов, описывающих граф; реализовать алгоритм Прима для построения минимального остовного дерева.

***Минимальное остовное дерево***(или ***минимальное покрывающее дерево***) в связанном взвешенном неориентированном графе — это остовное дерево данного графа, имеющее минимальный возможный вес, где под весом дерева понимается сумма весов входящих в него рёбер.



*Остовное дерево*

**Алгоритм Прима** — алгоритм построения минимального остовного дерева взвешенного связного неориентированного графа. Алгоритм впервые был открыт в 1930 году чешским математиком Войцехом Ярником, позже переоткрыт Робертом Примом в 1957 году, и, независимо от них, Э. Дейкстрой в 1959 году.

Построение начинается с дерева, включающего в себя одну (произвольную) вершину. В течение работы алгоритма дерево разрастается, пока не охватит все вершины исходного графа. На каждом шаге алгоритма к текущему дереву присоединяется самое лёгкое из рёбер, соединяющих вершину из построенного дерева и вершину не из дерева.

**Реализация:** решение задачи осуществлено на языке *Java 8*, разработка велась с использованием среды *Intellij Idea*.

Реализацию задачи можно разбить на четыре части:

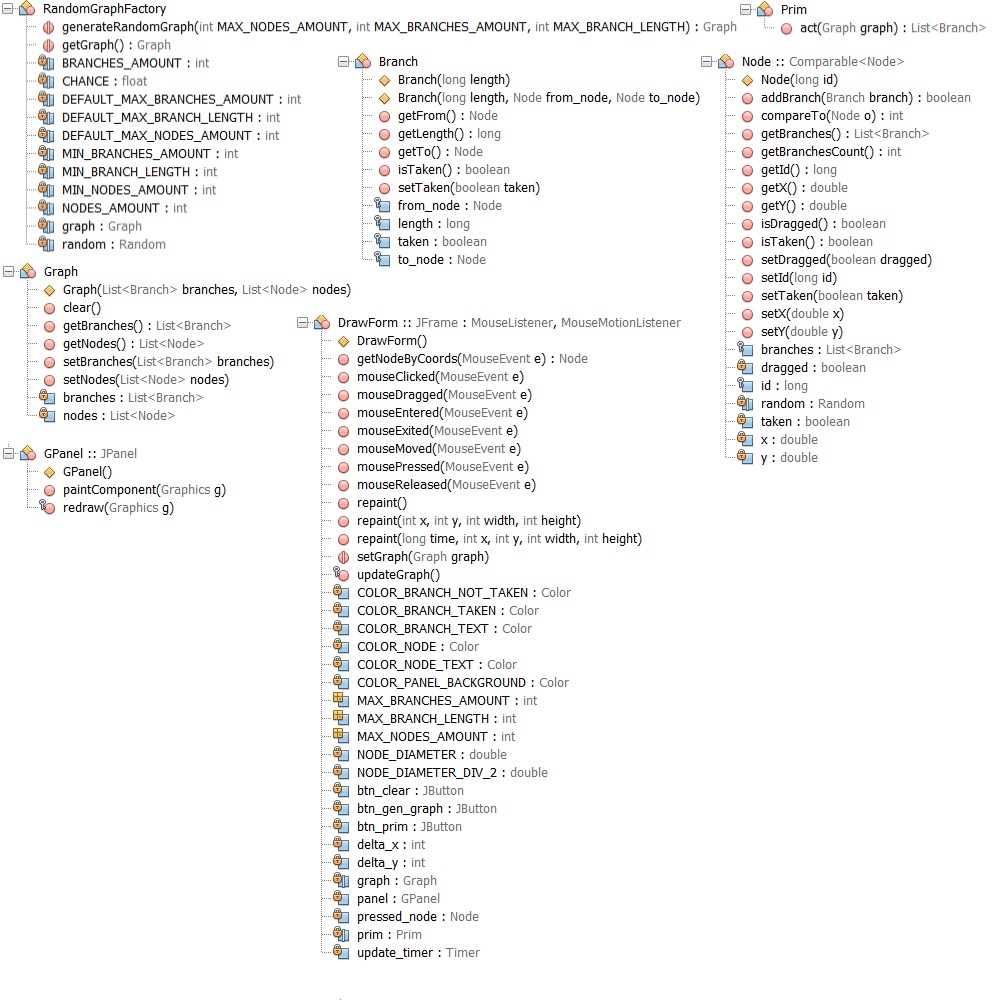
1. Реализация классов *Node*(узел графа), *Branch*(ветвь графа) и *Graph*(представление графа).
2. Реализация алгоритма генерации случайных графов.
3. Реализация алгоритма Прима для построения минимального остовного дерева.
4. Отрисовка графа и построенного пути.

1. Узел(*Node*) характеризуется координатами на двумерной плоскости, флагом, сигнализирующим о том, учтён ли узел при работе алгоритма Прима, уникальным идентификатором и набором ветвей, исходящих из него. Ветвь(*Branch*) графа характеризуется весом, узлами *источником и приёмником* и флагом, сигнализирующим о том, учтёна ли ветвь при работе алгоритма Прима. Граф(*Graph*) представляет собой набор вершин и веток.

2. Геренация случайных графов происходит путём построения графа, в котором связана любая пара вершин, и последующим удалением из него случайных веток. Такой механизм выбран из соображений простоты реализации и кастомизации.

3. Алгоритм Прима реализуется согласно описанию, приведённому выше.

4. Механика отрисовки графа будет выглядеть так: при создании нового узла, случайным образом генерируются координаты этого узла на плоскости. При запуске программы инициализируется и стартует таймер, который с интервалом в 50мс обновляет поверхность холста, отображая на ней текущее расположение графа(рисуются узлы с их идентификаторами и ветки графа с их весами). В любой момент времени в переменной *graph* хранится представление графа, с которым происходит работа в настоящий момент времени. Возможно перетаскивание вершин графа. Оно реализовано так: 1) в начальный момент перетаскивания определяется и запоминается узел, находящийся под курсором мыши; 2) если перетаскивание узла возможно, для него устанавливаются новые координаты в соответствии с новыми координатами курсора; 3) в момент окончания перетаскивания узел помечается как «не перетаскиваемый в данный момент» и очищается переменная, хранящая текущий перетаскиваемый узел.



*Спроектированные классы*

**Ниже представлен код программы.**

Реализация алгоритма Прима(*Prim.java*):

package mstprim.alghoritm.model;

import com.sun.istack.internal.NotNull;

import mstprim.graph.model.Graph;

import mstprim.graph.model.Node;

import mstprim.graph.model.Branch;

import java.util.\*;

public class Prim {

@NotNull

public List<Branch> act(@NotNull final Graph graph) {

final List<Node> nodes = graph.getNodes();

final List<Branch> taken\_branches = new LinkedList<>();

final int rand\_start\_index = Math.abs(new Random().nextInt() % nodes.size());

nodes.get(rand\_start\_index).setTaken(true);

while (nodes.stream().anyMatch(node -> !node.isTaken())) {

Branch shortest\_branch = null;

for (Node node: nodes) {

for (Branch branch\_of\_node: node.getBranches()) {

if (!branch\_of\_node.getFrom().isTaken() && !branch\_of\_node.getTo().isTaken()

|| branch\_of\_node.getFrom().isTaken() && branch\_of\_node.getTo().isTaken()

|| branch\_of\_node.isTaken()

|| branch\_of\_node == shortest\_branch)

continue;

if (shortest\_branch == null

|| (branch\_of\_node.getLength() <= shortest\_branch.getLength()))

shortest\_branch = branch\_of\_node;

}

}

if (shortest\_branch == null || shortest\_branch.isTaken())

continue;

taken\_branches.add(shortest\_branch);

shortest\_branch.setTaken(true);

shortest\_branch.getFrom().setTaken(true);

shortest\_branch.getTo().setTaken(true);

}

return taken\_branches;

}

}

Описание ветви графа(*Branch.java*):

package mstprim.graph.model;

import com.sun.istack.internal.NotNull;

/\*\*

\* Created by Max on 4/27/14.

\*/

public class Branch {

protected Node from\_node;

protected Node to\_node;

protected long length;

protected boolean taken; // Учтена ли ветка при поиске МОД

public Branch(final long length) {

this.length = length;

}

public Branch(final long length, @NotNull final Node from\_node, @NotNull final Node to\_node) {

this(length);

this.from\_node = from\_node;

this.to\_node = to\_node;

}

@NotNull

public Node getFrom() {

return from\_node;

}

@NotNull

public Node getTo() {

return to\_node;

}

public boolean isTaken() {

return taken;

}

public void setTaken(final boolean taken) {

this.taken = taken;

}

public long getLength() {

return length;

}

}

Описание узла графа(*Node.java*):

package mstprim.graph.model;

import com.sun.istack.internal.NotNull;

import java.util.LinkedList;

import java.util.List;

import java.util.Random;

/\*\*

\* Created by Max on 4/27/14.

\*/

public class Node implements Comparable<Node> {

protected long id;

protected List<Branch> branches;

private boolean taken;

//

private static Random random = new Random();

private double x = 0;

private double y = 0;

private boolean dragged = false;

public Node(final long id) {

this.setId(id);

branches = new LinkedList<>();

//

x = (int) (200 + (random.nextFloat() \* 300));

y = (int) (100 + (random.nextFloat() \* 200));

}

public boolean addBranch(@NotNull final Branch branch) {

if (branch.getFrom().getId() != getId()

&& branch.getTo().getId() != getId()

|| !getBranches().add(branch))

return false;

return true;

}

public int getBranchesCount() {

return getBranches().size();

}

public long getId() {

return id;

}

public void setId(long id) {

this.id = id;

}

@NotNull

public List<Branch> getBranches() {

return branches;

}

public boolean isDragged() {

return dragged;

}

public void setDragged(boolean dragged) {

this.dragged = dragged;

}

public double getX() {

return x;

}

public void setX(double x) {

this.x = x;

}

public double getY() {

return y;

}

public void setY(double y) {

this.y = y;

}

@Override

public int compareTo(Node o) {

return getBranchesCount() - o.getBranchesCount();

}

public boolean isTaken() {

return taken;

}

public void setTaken(boolean taken) {

this.taken = taken;

}

}

Описание графа(*Graph.java*):

package mstprim.graph.model;

import com.sun.istack.internal.NotNull;

import java.util.List;

public class Graph {

private List<Branch> branches;

private List<Node> nodes;

public Graph(@NotNull final List<Branch> branches, @NotNull final List<Node> nodes) {

this.branches = branches;

this.nodes = nodes;

}

@NotNull

public List<Branch> getBranches() {

return branches;

}

public void setBranches(@NotNull final List<Branch> branches) {

this.branches = branches;

}

@NotNull

public List<Node> getNodes() {

return nodes;

}

public void setNodes(@NotNull final List<Node> nodes) {

this.nodes = nodes;

}

public void clear() {

branches.forEach(branch -> branch.setTaken(false));

nodes.forEach(node -> node.setTaken(false));

}

}

Описание генератора случайных графов(*RandomGraphFactory.java*):

package mstprim;

import mstprim.graph.model.Branch;

import mstprim.graph.model.Graph;

import mstprim.graph.model.Node;

import java.util.\*;

public class RandomGraphFactory {

static private final Random random = new Random();

static private Graph graph = null;

static private final int MIN\_NODES\_AMOUNT = 3;

static private final int MIN\_BRANCHES\_AMOUNT = MIN\_NODES\_AMOUNT - 1;

static private final int MIN\_BRANCH\_LENGTH = 5;

static private int NODES\_AMOUNT;

static private int BRANCHES\_AMOUNT;

static private final int DEFAULT\_MAX\_NODES\_AMOUNT = 13;

static private final int DEFAULT\_MAX\_BRANCHES\_AMOUNT = DEFAULT\_MAX\_NODES\_AMOUNT \* 2;

static private final int DEFAULT\_MAX\_BRANCH\_LENGTH = 32;

static private final float CHANCE = 0.3f;

static public Graph getGraph() throws Exception {

if (graph == null)

graph = generateRandomGraph(DEFAULT\_MAX\_NODES\_AMOUNT, DEFAULT\_MAX\_BRANCHES\_AMOUNT, DEFAULT\_MAX\_BRANCH\_LENGTH);

return graph;

}

static public Graph generateRandomGraph(final int MAX\_NODES\_AMOUNT,

final int MAX\_BRANCHES\_AMOUNT,

final int MAX\_BRANCH\_LENGTH) throws Exception {

if (MAX\_NODES\_AMOUNT < MIN\_NODES\_AMOUNT)

throw new Exception("MAX\_NODES\_AMOUNT can not be less than MIN\_NODES\_AMOUNT(" + MIN\_NODES\_AMOUNT + ")");

if (MAX\_BRANCHES\_AMOUNT < MIN\_BRANCHES\_AMOUNT)

throw new Exception("MAX\_BRANCHES\_AMOUNT can not be less than MIN\_BRANCHES\_AMOUNT(" + MIN\_BRANCHES\_AMOUNT + ")");

if (MAX\_BRANCH\_LENGTH < MIN\_BRANCH\_LENGTH)

throw new Exception("MAX\_BRANCH\_LENGTH can not be less than MIN\_BRANCH\_LENGTH(" + MIN\_BRANCH\_LENGTH + ")");

do

NODES\_AMOUNT = Math.abs(random.nextInt() % MAX\_NODES\_AMOUNT) + 1;

while (NODES\_AMOUNT < MIN\_NODES\_AMOUNT);

do

BRANCHES\_AMOUNT = Math.abs(random.nextInt() % MAX\_BRANCHES\_AMOUNT) + 1;

while (BRANCHES\_AMOUNT < NODES\_AMOUNT);

final List<Node> nodes = new ArrayList<>(NODES\_AMOUNT);

final List<Branch> branches = new ArrayList<>();

for (int i = 0; i < NODES\_AMOUNT; i++)

nodes.add(new Node(i + 1));

for (int from = 0; from < NODES\_AMOUNT - 1; from++) {

for (int to = from + 1; to < NODES\_AMOUNT; to++) {

int branch\_length = 0;

do

branch\_length = Math.abs(random.nextInt() % MAX\_BRANCH\_LENGTH + 1);

while (branch\_length < MIN\_BRANCH\_LENGTH);

final Node from\_node = nodes.get(from);

final Node to\_node = nodes.get(to);

final Branch branch\_from\_to = new Branch(branch\_length, from\_node, to\_node);

branches.add(branch\_from\_to);

from\_node.addBranch(branch\_from\_to);

}

}

final int iters = (int)((1.0f - CHANCE) \* 10);

for (int i = 0; i < iters; i++) {

List<Branch> to\_remove = new ArrayList<>();

branches.forEach(branch -> {

if (random.nextFloat() <= CHANCE) {

if (branch.getFrom().getBranchesCount() > 1 &&

branch.getTo().getBranchesCount() > 1) {

to\_remove.add(branch);

}

}

});

to\_remove.forEach(branch -> {

branches.remove(branch);

nodes.forEach(node -> node.getBranches().remove(branch));

});

}

graph = new Graph(branches, nodes);

return graph;

}

}

Интерфейс приложения и отрисовка графов(*DrawForm.java*):

package mstprim.view;

import com.sun.istack.internal.NotNull;

import mstprim.RandomGraphFactory;

import mstprim.alghoritm.model.Prim;

import mstprim.graph.model.Branch;

import mstprim.graph.model.Graph;

import mstprim.graph.model.Node;

import javax.swing.\*;

import java.awt.\*;

import java.awt.event.MouseAdapter;

import java.awt.event.MouseEvent;

import java.awt.event.MouseListener;

import java.awt.event.MouseMotionListener;

import java.awt.geom.Rectangle2D;

import java.util.List;

public class DrawForm extends JFrame implements MouseListener, MouseMotionListener {

private final GPanel panel;

private final JButton btn\_gen\_graph;

private final JButton btn\_prim;

private final JButton btn\_clear;

static private Graph graph;

static private Prim prim;

//

final private Color COLOR\_BRANCH\_TAKEN = Color.GREEN;

final private Color COLOR\_BRANCH\_NOT\_TAKEN = Color.GRAY;

final private Color COLOR\_NODE = Color.YELLOW;

final private Color COLOR\_NODE\_TEXT = Color.BLACK;

final private Color COLOR\_BRANCH\_TEXT = Color.WHITE;

final private Color COLOR\_PANEL\_BACKGROUND = Color.BLACK;

final private double NODE\_DIAMETER = 30;

final private double NODE\_DIAMETER\_DIV\_2 = NODE\_DIAMETER / 2;

final private Timer update\_timer;

final int MAX\_NODES\_AMOUNT = 8;

final int MAX\_BRANCHES\_AMOUNT = MAX\_NODES\_AMOUNT \* 2;

final int MAX\_BRANCH\_LENGTH = 24;

//

private Node pressed\_node = null;

private int delta\_x = 0;

private int delta\_y = 0;

public DrawForm() {

super("Visualisation");

prim = new Prim();

panel = new GPanel();

btn\_gen\_graph = new JButton();

btn\_prim = new JButton();

btn\_clear = new JButton();

panel.add(btn\_gen\_graph);

panel.add(btn\_prim);

panel.add(btn\_clear);

setContentPane(panel);

pack();

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

update\_timer = new Timer(50, listener -> {

repaint();

});

update\_timer.start();

btn\_gen\_graph.setLocation(10, 10);

btn\_gen\_graph.setText("Gen graph");

btn\_gen\_graph.addMouseListener(new MouseAdapter() {

@Override

public void mouseClicked(MouseEvent e) {

super.mouseClicked(e);

try {

setGraph(RandomGraphFactory.generateRandomGraph(MAX\_NODES\_AMOUNT, MAX\_BRANCHES\_AMOUNT, MAX\_BRANCH\_LENGTH));

} catch (Exception e1) {

e1.printStackTrace();

}

}

});

btn\_prim.setText("Prim");

btn\_prim.addMouseListener(new MouseAdapter() {

@Override

public void mouseClicked(MouseEvent e) {

super.mouseClicked(e);

prim.act(graph);

}

});

btn\_clear.setText("Clear");

btn\_clear.addMouseListener(new MouseAdapter() {

@Override

public void mouseClicked(MouseEvent e) {

super.mouseClicked(e);

graph.clear();

}

});

setSize(800, 600);

setLocation(500, 200);

setVisible(true);

}

@Override

public void repaint() {

super.repaint();

}

@Override

public void repaint(final int x, final int y, final int width, final int height) {

super.repaint(x, y, width, height);

}

@Override

public void repaint(final long time, final int x, final int y, final int width, final int height) {

super.repaint(time, x, y, width, height);

}

@Override

public void mouseClicked(MouseEvent e) {}

@Override

public void mousePressed(MouseEvent e) {

pressed\_node = getNodeByCoords(e);

if (pressed\_node == null)

return;

pressed\_node.setDragged(true);

delta\_x = e.getX() - (int)pressed\_node.getX();

delta\_y = e.getY() - (int)pressed\_node.getY();

}

@Override

public void mouseReleased(MouseEvent e) {

if (pressed\_node == null)

return;

pressed\_node.setDragged(false);

pressed\_node = null;

delta\_x = 0;

delta\_y = 0;

}

@Override

public void mouseEntered(MouseEvent e) {}

@Override

public void mouseExited(MouseEvent e) {}

public Node getNodeByCoords(@NotNull final MouseEvent e) {

final int x = e.getX();

final int y = e.getY();

final List<Node> nodes = graph.getNodes();

for (Node node: nodes)

if (x >= node.getX() && y >= node.getY()

&& x <= node.getX() + NODE\_DIAMETER

&& y <= node.getY() + NODE\_DIAMETER)

return node;

return null;

}

@Override

public void mouseDragged(MouseEvent e) {

if (pressed\_node == null)

return;

if (pressed\_node.isDragged()) {

pressed\_node.setX(e.getX() - delta\_x);

pressed\_node.setY(e.getY() - delta\_y);

}

}

@Override

public void mouseMoved(MouseEvent e) {}

static public void setGraph(@NotNull final Graph graph) {

DrawForm.graph = graph;

}

public class GPanel extends JPanel {

public GPanel() {

addMouseListener(DrawForm.this);

addMouseMotionListener(DrawForm.this);

setBackground(COLOR\_PANEL\_BACKGROUND);

}

@Override

public void paintComponent(Graphics g) {

super.paintComponent(g);

synchronized (DrawForm.this) {

redraw(g);

}

}

protected void redraw(@NotNull Graphics g) {

if (graph == null)

return;

@NotNull List<Node> nodes = graph.getNodes();

@NotNull List<Branch> branches = graph.getBranches();

branches.forEach(branch -> {

final Node from = branch.getFrom();

final Node to = branch.getTo();

g.setColor(branch.isTaken() ? COLOR\_BRANCH\_TAKEN : COLOR\_BRANCH\_NOT\_TAKEN);

g.drawLine((int)(from.getX() + NODE\_DIAMETER\_DIV\_2),

(int)(from.getY() + NODE\_DIAMETER\_DIV\_2),

(int)(to.getX() + NODE\_DIAMETER\_DIV\_2),

(int)(to.getY() + NODE\_DIAMETER\_DIV\_2));

int gX = (int) from.getX();

int mX = (int) to.getX();

int gY = (int) from.getY();

int mY = (int) to.getY();

if (mX > gX) {

int tmp = gX;

gX = mX;

mX = tmp;

}

if (mY > gY) {

int tmp = gY;

gY = mY;

mY = tmp;

}

String weight = branch.getLength() + "";

char[] chars = weight.toCharArray();

g.setColor(COLOR\_BRANCH\_TEXT);

g.drawChars(chars, 0, weight.length(), mX + ((gX - mX) / 2), mY + ((gY - mY) / 2));

});

nodes.forEach(node -> {

g.setColor(COLOR\_NODE);

g.fillOval((int) node.getX(),

(int) node.getY(),

(int) NODE\_DIAMETER,

(int) NODE\_DIAMETER);

g.setColor(COLOR\_NODE\_TEXT);

String text = node.getId() + "";

Rectangle2D text\_bounds = g.getFontMetrics().getStringBounds(text, g);

int x = (int)(node.getX() - text\_bounds.getX() / 2 + NODE\_DIAMETER\_DIV\_2);

int y = (int)(node.getY() - text\_bounds.getY() / 2 + NODE\_DIAMETER\_DIV\_2);

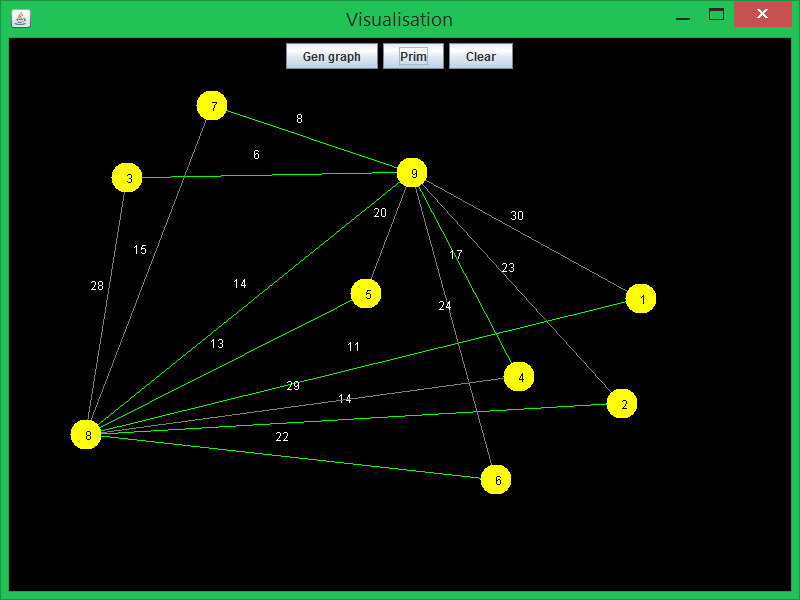
g.drawString(text, x, y);

});

}

}

}



*Пример работы программы*

**Вывод:** в ходе курсовой работы мною был реализован алгоритм Прима для построения минимального остовного дерева и система классов, описывающая граф; так же Я улучшил навыки владения ЯВУ *Java*(в частности, на практике изучил некоторые нововведения 8 версии языка), поработал с библиотекой для создания кроссплатформенных интерфейсов SWING и реализовал отрисовку графов с возможностью перетаскивания узлов.

*Использованные материалы:*

1. <http://ru.wikipedia.org/wiki/Алгоритм_Прима>
2. E-maxx::algo (<http://e-maxx.ru/upload/e-maxx_algo.pdf>)