Класс Programпредназначен для запуска программы.

import face.MainFrame;

import javax.swing.\*;

import java.awt.\*;

public class Program {

public static Rectangle getDefaultBounds() {

Toolkit toolkit = Toolkit.getDefaultToolkit();

Dimension screenSize = toolkit.getScreenSize();

Rectangle bounds = new Rectangle((int) screenSize.getWidth() / 12, (int) screenSize.getHeight() / 12,

(int) screenSize.getWidth() / 12 \* 10, (int) screenSize.getHeight() / 12 \* 10);

return bounds;

}

public static void main(String[] args) {

if (System.getProperty("os.name").toLowerCase().contains("windows")) {

try {

UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");

} catch (ClassNotFoundException e) {

} catch (InstantiationException e) {

} catch (IllegalAccessException e) {

} catch (UnsupportedLookAndFeelException e) {

}

}

SwingUtilities.invokeLater(new Runnable() {

public void run() {

MainFrame frame = new MainFrame(Program.getDefaultBounds());

frame.setVisible(true);

}

});

}

}

Класс MainFrame реализует отрисовку главного окна программы и реализует основные действия с погружаемым графом.

package face;

import graph.GraphFileFilter;

import graph.GraphPanel;

import scheduler.Scheduler;

import javax.swing.\*;

import java.awt.\*;

import java.awt.event.ActionEvent;

import java.awt.event.KeyEvent;

import java.io.\*;

public class MainFrame extends JFrame {

private JMenuBar menuBar;

private JTabbedPane tabbedPane;

private JLabel statusLabel;

private GraphPanel graphPanel;

private JFileChooser chooser;

private File openedFile;

private JTable table;

private NewAction newAction;

private OpenAction openAction;

private SaveAction saveAction;

private SaveAsAction saveAsAction;

private CloseAction closeAction;

private ExitAction exitAction;

private LoadAction loadAction;

private AboutAction aboutAction;

public MainFrame(Rectangle bounds) {

super();

setBounds(bounds);

setMinimumSize(bounds.getSize());

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

setTitle("Операционные системы - Курсовой проект");

setLayout(new BorderLayout());

newAction = new NewAction(this);

openAction = new OpenAction(this);

saveAction = new SaveAction(this);

saveAsAction = new SaveAsAction(this);

closeAction = new CloseAction(this);

exitAction = new ExitAction(this);

loadAction = new LoadAction(this);

aboutAction = new AboutAction(this);

chooser = new JFileChooser();

chooser.setCurrentDirectory(new File("."));

chooser.setMultiSelectionEnabled(false);

menuBar = new JMenuBar();

JMenu fileMenu = new JMenu("Файл");

JMenu actionMenu = new JMenu("Действия");

JMenu helpMenu = new JMenu("Справка");

JMenuItem tempItem = new JMenuItem(newAction);

tempItem.setText("Создатьграф");

tempItem.setMnemonic('N');

tempItem.setAccelerator(KeyStroke.getKeyStroke('N', KeyEvent.CTRL\_MASK));

fileMenu.add(tempItem);

tempItem = new JMenuItem(openAction);

tempItem.setText("Открытьграф...");

tempItem.setMnemonic('O');

tempItem.setAccelerator(KeyStroke.getKeyStroke('O', KeyEvent.CTRL\_MASK));

fileMenu.add(tempItem);

tempItem = new JMenuItem(saveAction);

tempItem.setText("Сохранитьграф");

tempItem.setMnemonic('S');

tempItem.setAccelerator(KeyStroke.getKeyStroke('S', KeyEvent.CTRL\_MASK));

fileMenu.add(tempItem);

tempItem = new JMenuItem(saveAsAction);

tempItem.setText("Сохранить граф как...");

tempItem.setMnemonic('S');

tempItem.setAccelerator(KeyStroke.getKeyStroke('S', (KeyEvent.CTRL\_MASK | KeyEvent.SHIFT\_MASK)));

fileMenu.add(tempItem);

tempItem = new JMenuItem(closeAction);

tempItem.setText("Закрытьграф");

tempItem.setMnemonic('W');

tempItem.setAccelerator(KeyStroke.getKeyStroke('W', KeyEvent.CTRL\_MASK));

fileMenu.add(tempItem);

fileMenu.addSeparator();

tempItem = new JMenuItem(exitAction);

tempItem.setText("Выход");

tempItem.setMnemonic('E');

tempItem.setAccelerator(KeyStroke.getKeyStroke('E', KeyEvent.CTRL\_MASK));

fileMenu.add(tempItem);

tempItem = new JMenuItem(loadAction);

tempItem.setText("Погрузитьграф...");

tempItem.setMnemonic('L');

tempItem.setAccelerator(KeyStroke.getKeyStroke('L', KeyEvent.CTRL\_MASK));

actionMenu.add(tempItem);

tempItem = new JMenuItem(aboutAction);

tempItem.setText("Опрограмме...");

helpMenu.add(tempItem);

menuBar.add(fileMenu);

menuBar.add(actionMenu);

menuBar.add(helpMenu);

setJMenuBar(menuBar);

tabbedPane = new JTabbedPane();

graphPanel = new GraphPanel();

addGraphTab();

add(tabbedPane);

}

private void addGraphTab() {

JPanel graphEditorPanel = new JPanel();

graphEditorPanel.setLayout(new BorderLayout());

JToolBar toolBar = new JToolBar(JToolBar.VERTICAL);

toolBar.setFloatable(false);

toolBar.setRollover(true);

JButton tempButton = toolBar.add(new AbstractAction() {

public void actionPerformed(ActionEvent e) {

graphPanel.setAction((byte) 0);

statusLabel.setText("Перемещениевершин");

}

});

tempButton.setToolTipText("Перемещениевершин");

tempButton.setIcon(new ImageIcon("img/no\_action.png"));

tempButton = toolBar.add(new AbstractAction() {

public void actionPerformed(ActionEvent e) {

graphPanel.setAction((byte) 1);

statusLabel.setText("Добавлениевершины");

}

});

tempButton.setToolTipText("Добавлениевершины");

tempButton.setIcon(new ImageIcon("img/add\_vertex.png"));

tempButton = toolBar.add(new AbstractAction() {

public void actionPerformed(ActionEvent e) {

graphPanel.setAction((byte) 2);

statusLabel.setText("Добавлениеребра");

}

});

tempButton.setToolTipText("Добавлениеребра");

tempButton.setIcon(new ImageIcon("img/add\_connection.png"));

tempButton = toolBar.add(new AbstractAction() {

public void actionPerformed(ActionEvent e) {

graphPanel.setAction((byte) 3);

statusLabel.setText("Удалениевершины");

}

});

tempButton.setToolTipText("Удалениевершины");

tempButton.setIcon(new ImageIcon("img/delete\_vertex.png"));

graphEditorPanel.add(BorderLayout.WEST, toolBar);

statusLabel = new JLabel(" ");

graphEditorPanel.add(BorderLayout.SOUTH, statusLabel);

graphEditorPanel.add(graphPanel);

tabbedPane.addTab("Граф", graphEditorPanel);

}

private void addTableTab() {

JPanel tablePanel = new JPanel();

tablePanel.setLayout(new BorderLayout());

tablePanel.add(new JScrollPane(table));

tabbedPane.addTab("Погружение", tablePanel);

tabbedPane.setSelectedIndex(tabbedPane.getTabCount() - 1);

}

private class NewAction extends AbstractAction {

private MainFrame frame;

public NewAction(MainFrame frame) {

this.frame = frame;

}

public void actionPerformed(ActionEvent e) {

closeAction.actionPerformed(e);

graphPanel = new GraphPanel();

openedFile = null;

addGraphTab();

}

}

private class OpenAction extends AbstractAction {

private MainFrame frame;

public OpenAction(MainFrame frame) {

this.frame = frame;

}

public void actionPerformed(ActionEvent e) {

chooser.resetChoosableFileFilters();

chooser.addChoosableFileFilter(new GraphFileFilter());

int result = chooser.showOpenDialog(frame);

if (result == JFileChooser.APPROVE\_OPTION) {

try {

closeAction.actionPerformed(e);

ObjectInputStream inputStream = new ObjectInputStream(new FileInputStream(chooser.getSelectedFile()));

graphPanel = (GraphPanel) inputStream.readObject();

inputStream.close();

openedFile = new File(chooser.getSelectedFile().getName());

addGraphTab();

} catch (IOException e1) {

JOptionPane.showMessageDialog(frame, "Произошлаошибкаприоткрытиифайла.", "Ошибка",

JOptionPane.ERROR\_MESSAGE);

} catch (ClassNotFoundException e1) {

JOptionPane.showMessageDialog(frame, "Файлимеетнесовместимыйформат.", "Ошибка",

JOptionPane.ERROR\_MESSAGE);

}

}

}

}

private class SaveAction extends AbstractAction {

private MainFrame frame;

public SaveAction(MainFrame frame) {

this.frame = frame;

}

public void actionPerformed(ActionEvent e) {

if (openedFile == null) {

saveAsAction.actionPerformed(e);

}

else {

if (graphPanel.isModified()) {

try {

ObjectOutputStream outputStream = new ObjectOutputStream(new FileOutputStream(openedFile));

graphPanel.setModified(false);

outputStream.writeObject(graphPanel);

outputStream.close();

} catch (IOException e1) {

JOptionPane.showMessageDialog(frame, "Произошлаошибкапризаписивфайл.", "Ошибка",

JOptionPane.ERROR\_MESSAGE);

}

}

}

}

}

private class SaveAsAction extends AbstractAction {

private MainFrame frame;

public SaveAsAction(MainFrame frame) {

this.frame = frame;

}

public void actionPerformed(ActionEvent e) {

chooser.resetChoosableFileFilters();

chooser.addChoosableFileFilter(new GraphFileFilter());

int result = chooser.showSaveDialog(frame);

if (result == JFileChooser.APPROVE\_OPTION) {

if (!chooser.getSelectedFile().getName().toLowerCase().endsWith(GraphFileFilter.GRAPH\_EXTENSION)) {

chooser.setSelectedFile(new File(chooser.getSelectedFile().getAbsolutePath() +

GraphFileFilter.GRAPH\_EXTENSION));

}

try {

ObjectOutputStream outputStream = new ObjectOutputStream(new FileOutputStream(chooser.getSelectedFile()));

graphPanel.setModified(false);

outputStream.writeObject(graphPanel);

outputStream.close();

openedFile = new File(chooser.getSelectedFile().getName());

} catch (IOException e1) {

JOptionPane.showMessageDialog(frame, "Произошлаошибкаприсозданиифайла.", "Ошибка",

JOptionPane.ERROR\_MESSAGE);

}

}

}

}

private class CloseAction extends AbstractAction {

private MainFrame frame;

public CloseAction(MainFrame frame) {

this.frame = frame;

}

public void actionPerformed(ActionEvent e) {

if ((graphPanel != null) && (graphPanel.isModified())) {

int result = JOptionPane.showConfirmDialog(frame,

"Граф содержит несохраненные изменения. Желаете сохранить их перед закрытием?",

"Подтверждение", JOptionPane.YES\_NO\_OPTION);

if (result == JOptionPane.YES\_OPTION) {

if (openedFile != null) {

saveAction.actionPerformed(e);

}

else {

saveAsAction.actionPerformed(e);

}

}

}

openedFile = null;

graphPanel = null;

while (tabbedPane.getTabCount() > 0) {

tabbedPane.remove(0);

}

}

}

private class ExitAction extends AbstractAction {

private MainFrame frame;

public ExitAction(MainFrame frame) {

this.frame = frame;

}

public void actionPerformed(ActionEvent e) {

closeAction.actionPerformed(e);

System.exit(0);

}

}

private class LoadAction extends AbstractAction {

private MainFrame frame;

public LoadAction(MainFrame frame) {

this.frame = frame;

}

public void actionPerformed(ActionEvent e) {

if (graphPanel == null) {

JOptionPane.showMessageDialog(frame, "Сначаланеобходимосоздатьграфдляпогружения.", "Ошибка",

JOptionPane.ERROR\_MESSAGE);

} else {

String processorsCountString = JOptionPane.showInputDialog(frame,

"Введите количество процессоров в вычислительной системе:");

int processorsCount = Integer.valueOf(processorsCountString);

Scheduler scheduler = new Scheduler(processorsCount);

if (tabbedPane.getTabCount() > 1) {

tabbedPane.remove(tabbedPane.getTabCount() - 1);

table = null;

}

table = new JTable(scheduler.loadGraph(graphPanel.getConnectionMatrix(), graphPanel.getVertexesWeightVector(),

graphPanel.getVertexesNameVector()));

addTableTab();

}

}

}

private class AboutAction extends AbstractAction {

private MainFrame frame;

public AboutAction(MainFrame frame) {

this.frame = frame;

}

public void actionPerformed(ActionEvent e) {

JOptionPane.showMessageDialog(frame,

"Операционныесистемы - Курсовойпроект\nCopyright (c) 2011 ЗахожийИгорь",

"Опрограмме", JOptionPane.INFORMATION\_MESSAGE);

}

}

}

КлассыConnectionAlreadyExistExceptionиConnectionDoesntExistExceptionпредназначеныдляобработкиошибокприпристроенииграфа.

package graph;

change this template use File | Settings | File Templates.

class ConnectionAlreadyExistException extends Exception {

private static String TEXT = "Связьужесуществует.";

@Override

public String getMessage() {

return TEXT;

}

}

package graph;

class ConnectionDoesntExistException extends Exception {

private static String TEXT = "Связьнесуществует.";

@Override

public String getMessage() {

return TEXT;

}

}

Класс GraphFileFilter предназначен для выбора файлов необходимого формата при открытии созданного ранее графа.

package graph;

import javax.swing.filechooser.FileFilter;

import java.io.File;

public class GraphFileFilter extends FileFilter {

public static String GRAPH\_EXTENSION = ".graph";

private static String GRAPH\_DESCRIPTION = "Graph File";

public boolean accept(File pathname) {

return (pathname.getName().toLowerCase().endsWith(GRAPH\_EXTENSION) || pathname.isDirectory());

}

public String getDescription() {

return GRAPH\_DESCRIPTION;

}

}

Класс GraphPanelпредставляет панель для отображения, построения и редактирования погружаемого графа. Классы VertexиLine – вспомогательные классы.

packagegraph;

importjavax.swing.\*;

import java.awt.\*;

import java.awt.event.ActionEvent;

import java.awt.event.MouseAdapter;

import java.awt.event.MouseEvent;

import java.awt.event.MouseMotionAdapter;

import java.io.Serializable;

import java.util.ArrayList;

import java.util.Arrays;

public class GraphPanel extends JPanel implements Serializable {

private static Color PANEL\_COLOR = Color.WHITE;

private ArrayList<Vertex> vertexes;

private ArrayList<Line> lines;

private boolean modified;

private byte action;

// 0 - Moving vertexes

// 1 - Adding vertex

// 2 - Adding connection

// 3 - Deleting vertex

private Vertex draggedVertex;

private Point tempPoint;

private Line tempLine;

private Vertex tempVertex;

public GraphPanel() {

super();

addMouseListener(new GraphMouseListener(this));

addMouseMotionListener(new GraphMouseMotionListener(this));

setBackground(PANEL\_COLOR);

lines = new ArrayList<Line>();

vertexes = new ArrayList<Vertex>();

modified = false;

action = 0;

draggedVertex = null;

tempPoint = null;

tempLine = null;

tempVertex = null;

}

public void setAction(byte action) {

this.action = action;

}

public boolean isModified() {

return modified;

}

public void setModified(boolean modified) {

this.modified = modified;

}

public String[] getVertexesNameVector() {

String[] vector = new String[vertexes.size()];

for (int i = 0; i < vertexes.size(); i++) {

vector[i] = new String(vertexes.get(i).getName());

}

return vector;

}

public int[] getVertexesWeightVector() {

int[] vector = new int[vertexes.size()];

for (int i = 0; i < vertexes.size(); i++) {

vector[i] = vertexes.get(i).getWeight();

}

return vector;

}

public int[][] getConnectionMatrix() {

int[][] matrix = new int[vertexes.size()][];

for (int i = 0; i < matrix.length; i++) {

matrix[i] = new int[vertexes.size()];

for (int j = 0; j < matrix[i].length; j++) {

matrix[i][j] = 0;

}

}

for (int i = 0; i < vertexes.size(); i++) {

ArrayList<Vertex> nextVertexes = vertexes.get(i).getNextVertexes();

for (int j = 0; j < nextVertexes.size(); j++) {

int k = 0;

boolean found = false;

while ((k < vertexes.size()) && (!found)) {

if (vertexes.get(k) == nextVertexes.get(j)) {

found = true;

}

k++;

}

if (found) {

k--;

boolean lineFound = false;

int lineWeight = 0;

int l = 0;

while ((!lineFound) && (l < lines.size())) {

if ((lines.get(l).getP1().getX() == vertexes.get(i).getCenter().getX()) &&

(lines.get(l).getP1().getY() == vertexes.get(i).getCenter().getY()) &&

(lines.get(l).getP2().getX() == vertexes.get(k).getCenter().getX()) &&

(lines.get(l).getP2().getY() == vertexes.get(k).getCenter().getY())) {

lineFound = true;

lineWeight = lines.get(l).getWeight();

}

l++;

}

if (lineFound) {

matrix[i][k] = lineWeight;

}

}

}

}

return matrix;

}

public void paintComponent(Graphics g) {

super.paintComponent(g);

Graphics2D g2 = (Graphics2D) g;

if (!lines.isEmpty()) {

for (Line l : lines) {

l.draw(g2);

}

}

if (tempLine != null) {

tempLine.draw(g2);

}

if (!vertexes.isEmpty()) {

for (Vertex v : vertexes) {

v.draw(g2);

}

}

}

private class GraphMouseListener extends MouseAdapter implements Serializable {

private GraphPanel panel;

private GraphMouseListener(GraphPanel panel) {

this.panel = panel;

}

@Override

public void mousePressed(MouseEvent e) {

switch (action) {

case 0: {

int i = 0;

while ((i < vertexes.size()) && (draggedVertex == null)) {

if (vertexes.get(i).contains(e.getPoint())) {

draggedVertex = vertexes.get(i);

}

i++;

}

break;

}

case 2: {

int i = 0;

while ((i < vertexes.size()) && (tempPoint == null)) {

if (vertexes.get(i).contains(e.getPoint())) {

tempPoint = vertexes.get(i).getCenter();

tempVertex = vertexes.get(i);

}

i++;

}

break;

}

}

}

@Override

public void mouseReleased(MouseEvent e) {

switch (action) {

case 0: {

draggedVertex = null;

break;

}

case 2: {

int i = 0;

boolean found = false;

while ((i < vertexes.size()) && (!found)) {

if (vertexes.get(i).contains(e.getPoint())) {

found = true;

}

i++;

}

if (found) {

i--;

String weightString = JOptionPane.showInputDialog(panel, "Введитевесребра:");

int weight = Integer.valueOf(weightString);

if (weight > 0) {

lines.add(new Line(tempPoint, vertexes.get(i).getCenter(), weight));

try {

tempVertex.addNextVertex(vertexes.get(i));

modified = true;

} catch (ConnectionAlreadyExistException e1) {

JOptionPane.showMessageDialog(panel, e1.getMessage(), "Ошибка",

JOptionPane.ERROR\_MESSAGE);

}

}

else {

JOptionPane.showMessageDialog(panel, "Некорректныйвесребра!", "Ошибка",

JOptionPane.ERROR\_MESSAGE);

}

}

tempPoint = null;

tempLine = null;

tempVertex = null;

repaint();

break;

}

}

}

@Override

public void mouseClicked(MouseEvent e) {

if (e.getButton() == MouseEvent.BUTTON1) {

switch (action) {

case 1: {

String name = JOptionPane.showInputDialog(panel, "Введитеимявершины:");

String weightString = JOptionPane.showInputDialog(panel, "Введитевесвершины:");

int weight = Integer.valueOf(weightString);

if (weight > 0) {

vertexes.add(new Vertex(name, weight, e.getPoint()));

modified = true;

repaint();

}

else {

JOptionPane.showMessageDialog(panel, "Некорректныйвесвершины!", "Ошибка",

JOptionPane.ERROR\_MESSAGE);

}

break;

}

case 3: {

int i = 0;

boolean found = false;

while ((i < vertexes.size()) && (!found)) {

if (vertexes.get(i).contains(e.getPoint())) {

found = true;

}

i++;

}

if (found) {

int result = JOptionPane.showConfirmDialog(panel, "Выдействительнохотитеудалитьэтувершину?",

"Подтверждениеудаления", JOptionPane.YES\_NO\_OPTION, JOptionPane.QUESTION\_MESSAGE);

if (result == JOptionPane.YES\_OPTION) {

i--;

for (Vertex v : vertexes) {

try {

v.removeNextVertex(vertexes.get(i));

} catch (ConnectionDoesntExistException e1) {}

}

for (int j = lines.size() - 1; j >= 0; j--) {

if (((lines.get(j).getP1().getX() == vertexes.get(i).getCenter().getX()) &&

(lines.get(j).getP1().getY() == vertexes.get(i).getCenter().getY())) ||

((lines.get(j).getP2().getX() == vertexes.get(i).getCenter().getX()) &&

(lines.get(j).getP2().getY() == vertexes.get(i).getCenter().getY()))) {

lines.remove(j);

}

}

vertexes.remove(i);

modified = true;

repaint();

}

}

break;

}

}

}

else {

if (e.getButton() == MouseEvent.BUTTON3) {

int i = 0;

boolean found = false;

while ((i < vertexes.size()) && (!found)) {

if (vertexes.get(i).contains(e.getPoint())) {

found = true;

}

i++;

}

if (found) {

i--;

JPopupMenu popupMenu = new JPopupMenu();

JMenu deleteConnectionMenu = new JMenu("Удалитьребро...");

for (Vertex v : vertexes.get(i).getNextVertexes()) {

JMenuItem menuItem = new JMenuItem(new DeleteConnectionAction(vertexes.get(i), v));

StringBuilder builder = new StringBuilder();

builder.append(vertexes.get(i).getName());

builder.append(" -> ");

builder.append(v.getName());

menuItem.setText(builder.toString());

deleteConnectionMenu.add(menuItem);

}

popupMenu.add(deleteConnectionMenu);

popupMenu.show(panel, e.getX(), e.getY());

}

}

}

}

private class DeleteConnectionAction extends AbstractAction implements Serializable {

private Vertex parentVertex;

private Vertex childVertex;

public DeleteConnectionAction(Vertex parentVertex, Vertex childVertex) {

this.parentVertex = parentVertex;

this.childVertex = childVertex;

}

public void actionPerformed(ActionEvent e) {

try {

parentVertex.removeNextVertex(childVertex);

} catch (ConnectionDoesntExistException e1) {}

int i = 0;

boolean found = false;

while ((!found) && (i < lines.size())) {

if ((lines.get(i).getP1().getX() == parentVertex.getCenter().getX()) &&

(lines.get(i).getP1().getY() == parentVertex.getCenter().getY()) &&

(lines.get(i).getP2().getX() == childVertex.getCenter().getX()) &&

(lines.get(i).getP2().getY() == childVertex.getCenter().getY())) {

found = true;

}

i++;

}

if (found) {

lines.remove(--i);

modified = true;

}

repaint();

}

}

}

private class GraphMouseMotionListener extends MouseMotionAdapter implements Serializable {

private GraphPanel panel;

private GraphMouseMotionListener(GraphPanel panel) {

this.panel = panel;

}

@Override

public void mouseDragged(MouseEvent e) {

if (draggedVertex != null) {

ArrayList<Line> inLines = new ArrayList<Line>();

ArrayList<Line> outLines = new ArrayList<Line>();

for (Line l : lines) {

if ((l.getP1().getX() == draggedVertex.getCenter().getX()) &&

(l.getP1().getY() == draggedVertex.getCenter().getY())) {

outLines.add(l);

}

else {

if ((l.getP2().getX() == draggedVertex.getCenter().getX()) &&

(l.getP2().getY() == draggedVertex.getCenter().getY())) {

inLines.add(l);

}

}

}

draggedVertex.setLeftUpperCorner(e.getPoint());

for (Line l : inLines) {

l.setP2(draggedVertex.getCenter());

}

for (Line l : outLines) {

l.setP1(draggedVertex.getCenter());

}

modified = true;

repaint();

}

if (tempPoint != null) {

tempLine = new Line(tempPoint, e.getPoint(), -1);

repaint();

}

}

}

}

package graph;

import java.awt.\*;

import java.awt.font.FontRenderContext;

import java.awt.geom.Point2D;

import java.awt.geom.Rectangle2D;

import java.io.Serializable;

class Line implements Serializable {

private static Color LINE\_COLOR = Color.BLACK;

private static Color TEXT\_COLOR = Color.RED;

private static Font TEXT\_FONT = new Font("Monospaced", Font.PLAIN, 14);

private static int ARROW\_LENGTH = 20;

private static double ARROW\_ANGLE = 0.5;

private Point2D p1;

private Point2D p2;

private int weight;

public Line(Point2D p1, Point2D p2, int weight) {

this.p1 = p1;

this.p2 = p2;

this.weight = weight;

}

public Point2D getP1() {

return p1;

}

public void setP1(Point2D p1) {

this.p1 = p1;

}

public Point2D getP2() {

return p2;

}

public void setP2(Point2D p2) {

this.p2 = p2;

}

public int getWeight() {

return weight;

}

public void setWeight(int weight) {

this.weight = weight;

}

public void draw(Graphics2D g2) {

g2.setColor(LINE\_COLOR);

g2.drawLine((int) p1.getX(), (int) p1.getY(), (int) p2.getX(), (int) p2.getY());

int length = (int) Math.sqrt(Math.pow(p2.getX() - p1.getX(), 2) + Math.pow(p2.getY() - p1.getY(), 2));

int lambda = (int) (length / (Vertex.getVERTEX\_DIAMETER() / 2));

int arrowX = (int) ((p1.getX() + lambda \* p2.getX()) / (1 + lambda));

int arrowY = (int) ((p1.getY() + lambda \* p2.getY()) / (1 + lambda));

double temp = Math.atan2(p1.getX() - arrowX, p1.getY() - arrowY);

g2.drawLine(arrowX, arrowY,

(int) (arrowX + ARROW\_LENGTH \* Math.sin(temp + ARROW\_ANGLE)),

(int) (arrowY + ARROW\_LENGTH \* Math.cos(temp + ARROW\_ANGLE)));

g2.drawLine(arrowX, arrowY,

(int) (arrowX + ARROW\_LENGTH \* Math.sin(temp - ARROW\_ANGLE)),

(int) (arrowY + ARROW\_LENGTH \* Math.cos(temp - ARROW\_ANGLE)));

if (weight != -1) {

g2.setColor(TEXT\_COLOR);

g2.setFont(TEXT\_FONT);

FontRenderContext context = g2.getFontRenderContext();

String weightString = String.valueOf(weight);

int centerX = (int) ((p1.getX() + p2.getX()) / 2);

int centerY = (int) ((p1.getY() + p2.getY()) / 2);

Rectangle2D bounds = TEXT\_FONT.getStringBounds(weightString, context);

int x = centerX - (int) bounds.getWidth() / 2;

int y = centerY + (int) bounds.getHeight() / 2;

g2.drawString(weightString, x, y);

}

}

}

package graph;

import java.awt.\*;

import java.awt.font.FontRenderContext;

import java.awt.geom.Point2D;

import java.awt.geom.Rectangle2D;

import java.io.Serializable;

import java.util.ArrayList;

class Vertex implements Serializable {

private static Color VERTEX\_COLOR = Color.CYAN;

private static Color TEXT\_COLOR = Color.BLACK;

private static int VERTEX\_DIAMETER = 50;

private static Font TEXT\_FONT = new Font("Monospaced", Font.PLAIN, 14);

private String name;

private int weight;

private Point2D leftUpperCorner;

private ArrayList<Vertex> nextVertexes;

public Vertex(String name, int weight, Point2D leftUpperCorner) {

this.name = name;

this.weight = weight;

this.leftUpperCorner = leftUpperCorner;

nextVertexes = new ArrayList<Vertex>();

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getWeight() {

return weight;

}

public void setWeight(int weight) {

this.weight = weight;

}

public Point2D getLeftUpperCorner() {

return leftUpperCorner;

}

public void setLeftUpperCorner(Point2D leftUpperCorner) {

this.leftUpperCorner = leftUpperCorner;

}

public ArrayList<Vertex> getNextVertexes() {

return nextVertexes;

}

public static int getVERTEX\_DIAMETER() {

return VERTEX\_DIAMETER;

}

public void addNextVertex(Vertex v) throws ConnectionAlreadyExistException {

if (nextVertexes.contains(v)) {

throw new ConnectionAlreadyExistException();

}

nextVertexes.add(v);

}

public void removeNextVertex(Vertex v) throws ConnectionDoesntExistException {

if (!nextVertexes.contains(v)) {

throw new ConnectionDoesntExistException();

}

nextVertexes.remove(v);

}

public boolean contains(Point p) {

if ((p.getX() >= leftUpperCorner.getX()) && (p.getX() <= leftUpperCorner.getX() + VERTEX\_DIAMETER) &&

(p.getY() >= leftUpperCorner.getY()) && (p.getY() <= leftUpperCorner.getY() + VERTEX\_DIAMETER)) {

return true;

}

else {

return false;

}

}

public Point getCenter() {

return new Point((int) (leftUpperCorner.getX() + (VERTEX\_DIAMETER / 2)),

(int) (leftUpperCorner.getY() + (VERTEX\_DIAMETER / 2)));

}

public void draw(Graphics2D g2) {

g2.setColor(VERTEX\_COLOR);

g2.fillOval((int) leftUpperCorner.getX(), (int) leftUpperCorner.getY(), VERTEX\_DIAMETER, VERTEX\_DIAMETER);

g2.setColor(TEXT\_COLOR);

g2.setFont(TEXT\_FONT);

FontRenderContext context = g2.getFontRenderContext();

String weightString = String.valueOf(weight);

Rectangle2D bounds = TEXT\_FONT.getStringBounds(weightString, context);

int x = ((int) leftUpperCorner.getX() + VERTEX\_DIAMETER / 2 - (int) bounds.getWidth() / 2);

int y = ((int) leftUpperCorner.getY() + VERTEX\_DIAMETER / 2 + (int) bounds.getHeight() / 4);

g2.drawString(weightString, x ,y);

bounds = TEXT\_FONT.getStringBounds(name, context);

x = ((int) leftUpperCorner.getX() + VERTEX\_DIAMETER / 2 - (int) bounds.getWidth() / 2);

y = (int) leftUpperCorner.getY() - 5;

g2.drawString(name, x, y);

}

}

Класс GanttTableModel представляет собой модель таблицы для вывода результатов планирования.

package scheduler;

import javax.swing.table.AbstractTableModel;

import java.util.ArrayList;

public class GanttTableModel extends AbstractTableModel {

private int columnCount;

private String[] columnNames;

private ArrayList<String[]> table;

public GanttTableModel(int columnCount) {

this.columnCount = columnCount;

table = new ArrayList<String[]>();

}

public void addColumnNamesRow(String[] row) {

columnNames = row;

}

public void addRow(String[] row) {

table.add(row);

}

public void setValueAt(String value, int row, int column) {

while (row >= table.size()) {

table.add(new String[columnCount]);

}

table.get(row)[column] = value;

}

public int getRowCount() {

return table.size();

}

public int getColumnCount() {

return columnCount;

}

@Override

public String getColumnName(int column) {

return columnNames[column];

}

public Object getValueAt(int rowIndex, int columnIndex) {

return table.get(rowIndex)[columnIndex];

}

}

Класс Schedulerсодержит все методы, реализующие планирование.

package scheduler;

import java.util.ArrayList;

public class Scheduler {

private static int STEP\_COUNT = 5;

private static double CROSS\_PROBABILITY = 0.5;

private static double MUTATION\_PROBABILITY = 0.05;

private static int POPULATION\_STEP = 10;

private static int BEST\_GENOTYPE\_NOT\_CHANGED\_MAX\_COUNT = 5;

private int processorCount;

private int channelCount;

private int populationSize;

public Scheduler(int processorCount) {

this.processorCount = processorCount;

channelCount = processorCount - 1;

populationSize = POPULATION\_STEP \* processorCount;

}

private ArrayList<Integer> getReadyTasks(int[][] connectionMatrix) {

ArrayList<Integer> readyTasks = new ArrayList<Integer>();

for (int i = 0; i < connectionMatrix[0].length; i++) {

boolean ready = true;

int j = 0;

while ((ready) && (j < connectionMatrix.length)) {

if (connectionMatrix[j][i] > 0) {

ready = false;

}

j++;

}

if (ready) {

readyTasks.add(i);

}

}

return readyTasks;

}

private boolean isAllTasksDone(int[] vertexWeight) {

boolean result = true;

int i = 0;

while ((i < vertexWeight.length) && (result)) {

if (vertexWeight[i] > 0) {

result = false;

}

i++;

}

return result;

}

private ArrayList<Integer> getWay(int from, int to) {

ArrayList<Integer> way = new ArrayList<Integer>();

ArrayList<ArrayList<Integer>> allWays = new ArrayList<ArrayList<Integer>>();

int waysCount = (processorCount + 1) / 2;

int waysLength = (int) (Math.log(processorCount + 1) / Math.log(2));

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

allWays.add(new ArrayList<Integer>());

for (int j = 0; j < waysLength; j++) {

allWays.get(i).add(0);

}

allWays.get(i).set(0, 0);

}

int processorNumber = 1;

int counter = 0;

int counterEdge = waysCount / 2;

for (int i = 0; i < (waysLength - 1); i++) {

for (int j = 0; j < waysCount; j++) {

allWays.get(j).set(i + 1, processorNumber);

counter++;

if (counter >= counterEdge) {

processorNumber++;

counter = 0;

}

}

counterEdge /= 2;

}

int foundFrom = -1;

int foundTo = -1;

int fromIndex = -1;

int toIndex = -1;

int i = 0;

while ((i < allWays.size()) && ((foundFrom < 0) || (foundTo < 0))) {

for (int j = 0; j < allWays.get(i).size(); j++) {

if (allWays.get(i).get(j) == from) {

foundFrom = i;

fromIndex = j;

}

if (allWays.get(i).get(j) == to) {

foundTo = i;

toIndex = j;

}

}

i++;

}

if (foundFrom == foundTo) {

if (fromIndex < toIndex) {

for (int j = fromIndex; j <= toIndex; j++) {

way.add(allWays.get(foundFrom).get(j));

}

}

else {

for (int j = fromIndex; j >= toIndex; j--) {

way.add(allWays.get(foundFrom).get(j));

}

}

}

else {

for (int j = fromIndex; j > 0; j--) {

way.add(allWays.get(foundFrom).get(j));

}

for (int j = 0; j <= toIndex; j++) {

way.add(allWays.get(foundTo).get(j));

}

}

return way;

}

private int getExecutionTime(ArrayList<Integer> processorNumbers, int[][] originalConnectionMatrix,

int[] originalVertexWeight) {

int[][] connectionMatrix = new int[originalConnectionMatrix.length][];

for (int i = 0; i < connectionMatrix.length; i++) {

connectionMatrix[i] = new int[originalConnectionMatrix[i].length];

for (int j = 0; j < connectionMatrix[i].length; j++) {

connectionMatrix[i][j] = originalConnectionMatrix[i][j];

}

}

int[] vertexWeight = new int[originalVertexWeight.length];

for (int i = 0; i < vertexWeight.length; i++) {

vertexWeight[i] = originalVertexWeight[i];

}

int time = 0;

boolean[] processorsStatus = new boolean[processorCount];

for (boolean e : processorsStatus) {

e = false;

}

int[] processorsTask = new int[processorCount];

for (int e : processorsTask) {

e = -1;

}

int[] channelProcessorFrom = new int[channelCount];

int[] channelProcessorTo = new int[channelCount];

int fromProcessor = 0;

int tempCounter = 0;

for (int j = 0; j < channelCount; j++) {

channelProcessorFrom[j] = fromProcessor;

channelProcessorTo[j] = j + 1;

tempCounter++;

if (tempCounter > 1) {

tempCounter = 0;

fromProcessor++;

}

}

boolean[] channelsStatus = new boolean[channelCount];

for (boolean e : channelsStatus) {

e = false;

}

int[] channelFrom = new int[channelCount];

for (int e : channelFrom) {

e = -1;

}

int[] channelTo = new int[channelCount];

for (int e : channelTo) {

e = -1;

}

ArrayList<ArrayList<Boolean>> startNextTact = new ArrayList<ArrayList<Boolean>>();

boolean[] isAlreadySend = new boolean[channelCount];

int[] channelTaskWeight = new int[channelCount];

int[] channelTaskWeightBackup = new int[channelCount];

ArrayList<ArrayList<Integer>> channelTaskWay = new ArrayList<ArrayList<Integer>>();

ArrayList<ArrayList<Integer>> channelFromQueue = new ArrayList<ArrayList<Integer>>();

ArrayList<ArrayList<Integer>> channelToQueue = new ArrayList<ArrayList<Integer>>();

ArrayList<ArrayList<ArrayList<Integer>>> channelWayQueue = new ArrayList<ArrayList<ArrayList<Integer>>>();

ArrayList<ArrayList<Integer>> channelWeightQueue = new ArrayList<ArrayList<Integer>>();

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

startNextTact.add(new ArrayList<Boolean>());

isAlreadySend[i] = false;

channelTaskWeight[i] = -1;

channelTaskWay.add(new ArrayList<Integer>());

channelFromQueue.add(new ArrayList<Integer>());

channelToQueue.add(new ArrayList<Integer>());

channelWayQueue.add(new ArrayList<ArrayList<Integer>>());

channelWeightQueue.add(new ArrayList<Integer>());

}

while (!isAllTasksDone(vertexWeight)) {

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

for (int j = 0; j < startNextTact.get(i).size(); j++) {

startNextTact.get(i).set(j, false);

}

}

ArrayList<Integer> readyTasks = getReadyTasks(connectionMatrix);

for (int i = 0; i < readyTasks.size(); i++) {

if (vertexWeight[readyTasks.get(i)] > 0) {

if (!processorsStatus[processorNumbers.get(readyTasks.get(i))]) {

processorsStatus[processorNumbers.get(readyTasks.get(i))] = true;

processorsTask[processorNumbers.get(readyTasks.get(i))] = readyTasks.get(i);

}

}

}

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

if (processorsStatus[i]) {

vertexWeight[processorsTask[i]]--;

if (vertexWeight[processorsTask[i]] == 0) {

for (int j = 0; j < connectionMatrix[processorsTask[i]].length; j++) {

if (connectionMatrix[processorsTask[i]][j] > 0) {

if (processorNumbers.get(j) != i) {

ArrayList<Integer> way = getWay(i, processorNumbers.get(j));

int channelNumber = -1;

int k = 0;

while ((channelNumber < 0) && (k < channelCount)) {

if (((channelProcessorFrom[k] == way.get(0)) && (channelProcessorTo[k] == way.get(1))) ||

((channelProcessorTo[k] == way.get(0)) && (channelProcessorFrom[k] == way.get(1)))) {

channelNumber = k;

}

k++;

}

channelFromQueue.get(channelNumber).add(processorsTask[i]);

channelToQueue.get(channelNumber).add(j);

channelWayQueue.get(channelNumber).add(way);

channelWeightQueue.get(channelNumber).add(connectionMatrix[processorsTask[i]][j]);

startNextTact.get(channelNumber).add(true);

}

else {

connectionMatrix[processorsTask[i]][j] = 0;

}

}

}

processorsStatus[i] = false;

}

}

}

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

if (channelsStatus[i]) {

channelTaskWeight[i]--;

if (!((channelProcessorTo[i] == channelTaskWay.get(i).get(channelTaskWay.get(i).size() - 1)) ||

(channelProcessorFrom[i] == channelTaskWay.get(i).get(channelTaskWay.get(i).size() - 1)))) {

if (!isAlreadySend[i]) {

isAlreadySend[i] = true;

ArrayList<Integer> nextTaskWay = new ArrayList<Integer>();

for (int l = 1; l < channelTaskWay.get(i).size(); l++) {

nextTaskWay.add(channelTaskWay.get(i).get(l));

}

int channelNumber = -1;

int k = 0;

while ((channelNumber < 0) && (k < channelCount)) {

if (((channelProcessorFrom[k] == nextTaskWay.get(0)) &&

(channelProcessorTo[k] == nextTaskWay.get(1))) ||

((channelProcessorTo[k] == nextTaskWay.get(0)) &&

(channelProcessorFrom[k] == nextTaskWay.get(1)))) {

channelNumber = k;

}

k++;

}

channelFromQueue.get(channelNumber).add(channelFrom[i]);

channelToQueue.get(channelNumber).add(channelTo[i]);

channelWayQueue.get(channelNumber).add(nextTaskWay);

channelWeightQueue.get(channelNumber).add(channelTaskWeightBackup[i]);

channelTaskWeightBackup[i] = 0;

startNextTact.get(channelNumber).add(true);

}

}

if (channelTaskWeight[i] == 0) {

if ((channelProcessorTo[i] == channelTaskWay.get(i).get(channelTaskWay.get(i).size() - 1)) ||

(channelProcessorFrom[i] == channelTaskWay.get(i).get(channelTaskWay.get(i).size() - 1))) {

connectionMatrix[channelFrom[i]][channelTo[i]] = 0;

}

channelsStatus[i] = false;

}

}

else {

if (!channelWeightQueue.get(i).isEmpty()) {

if (!startNextTact.get(i).get(0)) {

isAlreadySend[i] = false;

startNextTact.get(i).remove(0);

channelFrom[i] = channelFromQueue.get(i).get(0);

channelFromQueue.get(i).remove(0);

channelTo[i] = channelToQueue.get(i).get(0);

channelToQueue.get(i).remove(0);

channelTaskWay.set(i, channelWayQueue.get(i).get(0));

channelWayQueue.get(i).remove(0);

channelTaskWeight[i] = channelWeightQueue.get(i).get(0);

channelTaskWeightBackup[i] = channelWeightQueue.get(i).get(0);

channelWeightQueue.get(i).remove(0);

channelsStatus[i] = true;

channelTaskWeight[i]--;

if (!((channelProcessorTo[i] == channelTaskWay.get(i).get(channelTaskWay.get(i).size() - 1)) ||

(channelProcessorFrom[i] == channelTaskWay.get(i).get(channelTaskWay.get(i).size() - 1)))) {

if (!isAlreadySend[i]) {

isAlreadySend[i] = true;

ArrayList<Integer> nextTaskWay = new ArrayList<Integer>();

for (int l = 1; l < channelTaskWay.get(i).size(); l++) {

nextTaskWay.add(channelTaskWay.get(i).get(l));

}

int channelNumber = -1;

int k = 0;

while ((channelNumber < 0) && (k < channelCount)) {

if (((channelProcessorFrom[k] == nextTaskWay.get(0)) &&

(channelProcessorTo[k] == nextTaskWay.get(1))) ||

((channelProcessorTo[k] == nextTaskWay.get(0)) &&

(channelProcessorFrom[k] == nextTaskWay.get(1)))) {

channelNumber = k;

}

k++;

}

channelFromQueue.get(channelNumber).add(channelFrom[i]);

channelToQueue.get(channelNumber).add(channelTo[i]);

channelWayQueue.get(channelNumber).add(nextTaskWay);

channelWeightQueue.get(channelNumber).add(channelTaskWeightBackup[i]);

channelTaskWeightBackup[i] = 0;

startNextTact.get(channelNumber).add(true);

}

}

if (channelTaskWeight[i] == 0) {

if ((channelProcessorTo[i] == channelTaskWay.get(i).get(channelTaskWay.get(i).size() - 1)) ||

(channelProcessorFrom[i] == channelTaskWay.get(i).get(channelTaskWay.get(i).size() - 1))) {

connectionMatrix[channelFrom[i]][channelTo[i]] = 0;

}

channelsStatus[i] = false;

}

}

}

}

}

time++;

}

return time;

}

private void fillTable(ArrayList<Integer> processorNumbers, int[][] originalConnectionMatrix, int[] originalVertexWeight,

String[] vertexNames, GanttTableModel model) {

int[][] connectionMatrix = new int[originalConnectionMatrix.length][];

for (int i = 0; i < connectionMatrix.length; i++) {

connectionMatrix[i] = new int[originalConnectionMatrix[i].length];

for (int j = 0; j < connectionMatrix[i].length; j++) {

connectionMatrix[i][j] = originalConnectionMatrix[i][j];

}

}

int[] vertexWeight = new int[originalVertexWeight.length];

for (int i = 0; i < vertexWeight.length; i++) {

vertexWeight[i] = originalVertexWeight[i];

}

int time = 0;

boolean[] processorsStatus = new boolean[processorCount];

for (boolean e : processorsStatus) {

e = false;

}

int[] processorsTask = new int[processorCount];

for (int e : processorsTask) {

e = -1;

}

int[] channelProcessorFrom = new int[channelCount];

int[] channelProcessorTo = new int[channelCount];

int fromProcessor = 0;

int tempCounter = 0;

for (int j = 0; j < channelCount; j++) {

channelProcessorFrom[j] = fromProcessor;

channelProcessorTo[j] = j + 1;

tempCounter++;

if (tempCounter > 1) {

tempCounter = 0;

fromProcessor++;

}

}

boolean[] channelsStatus = new boolean[channelCount];

for (boolean e : channelsStatus) {

e = false;

}

int[] channelFrom = new int[channelCount];

for (int e : channelFrom) {

e = -1;

}

int[] channelTo = new int[channelCount];

for (int e : channelTo) {

e = -1;

}

ArrayList<ArrayList<Boolean>> startNextTact = new ArrayList<ArrayList<Boolean>>();

boolean[] isAlreadySend = new boolean[channelCount];

int[] channelTaskWeight = new int[channelCount];

int[] channelTaskWeightBackup = new int[channelCount];

ArrayList<ArrayList<Integer>> channelTaskWay = new ArrayList<ArrayList<Integer>>();

ArrayList<ArrayList<Integer>> channelFromQueue = new ArrayList<ArrayList<Integer>>();

ArrayList<ArrayList<Integer>> channelToQueue = new ArrayList<ArrayList<Integer>>();

ArrayList<ArrayList<ArrayList<Integer>>> channelWayQueue = new ArrayList<ArrayList<ArrayList<Integer>>>();

ArrayList<ArrayList<Integer>> channelWeightQueue = new ArrayList<ArrayList<Integer>>();

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

startNextTact.add(new ArrayList<Boolean>());

isAlreadySend[i] = false;

channelTaskWeight[i] = -1;

channelTaskWay.add(new ArrayList<Integer>());

channelFromQueue.add(new ArrayList<Integer>());

channelToQueue.add(new ArrayList<Integer>());

channelWayQueue.add(new ArrayList<ArrayList<Integer>>());

channelWeightQueue.add(new ArrayList<Integer>());

}

while (!isAllTasksDone(vertexWeight)) {

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

for (int j = 0; j < startNextTact.get(i).size(); j++) {

startNextTact.get(i).set(j, false);

}

}

String[] newRow = new String[processorCount + channelCount + 1];

newRow[0] = String.valueOf(time);

ArrayList<Integer> readyTasks = getReadyTasks(connectionMatrix);

for (int i = 0; i < readyTasks.size(); i++) {

if (vertexWeight[readyTasks.get(i)] > 0) {

if (!processorsStatus[processorNumbers.get(readyTasks.get(i))]) {

processorsStatus[processorNumbers.get(readyTasks.get(i))] = true;

processorsTask[processorNumbers.get(readyTasks.get(i))] = readyTasks.get(i);

}

}

}

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

if (processorsStatus[i]) {

newRow[i + 1] = vertexNames[processorsTask[i]];

vertexWeight[processorsTask[i]]--;

if (vertexWeight[processorsTask[i]] == 0) {

for (int j = 0; j < connectionMatrix[processorsTask[i]].length; j++) {

if (connectionMatrix[processorsTask[i]][j] > 0) {

if (processorNumbers.get(j) != i) {

ArrayList<Integer> way = getWay(i, processorNumbers.get(j));

int channelNumber = -1;

int k = 0;

while ((channelNumber < 0) && (k < channelCount)) {

if (((channelProcessorFrom[k] == way.get(0)) && (channelProcessorTo[k] == way.get(1))) ||

((channelProcessorTo[k] == way.get(0)) && (channelProcessorFrom[k] == way.get(1)))) {

channelNumber = k;

}

k++;

}

channelFromQueue.get(channelNumber).add(processorsTask[i]);

channelToQueue.get(channelNumber).add(j);

channelWayQueue.get(channelNumber).add(way);

channelWeightQueue.get(channelNumber).add(connectionMatrix[processorsTask[i]][j]);

startNextTact.get(channelNumber).add(true);

}

else {

connectionMatrix[processorsTask[i]][j] = 0;

}

}

}

processorsStatus[i] = false;

}

}

}

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

if (channelsStatus[i]) {

StringBuilder builder = new StringBuilder();

builder.append(vertexNames[channelFrom[i]]);

builder.append(" -> ");

builder.append(vertexNames[channelTo[i]]);

newRow[i + processorCount + 1] = builder.toString();

channelTaskWeight[i]--;

if (!((channelProcessorTo[i] == channelTaskWay.get(i).get(channelTaskWay.get(i).size() - 1)) ||

(channelProcessorFrom[i] == channelTaskWay.get(i).get(channelTaskWay.get(i).size() - 1)))) {

if (!isAlreadySend[i]) {

isAlreadySend[i] = true;

ArrayList<Integer> nextTaskWay = new ArrayList<Integer>();

for (int l = 1; l < channelTaskWay.get(i).size(); l++) {

nextTaskWay.add(channelTaskWay.get(i).get(l));

}

int channelNumber = -1;

int k = 0;

while ((channelNumber < 0) && (k < channelCount)) {

if (((channelProcessorFrom[k] == nextTaskWay.get(0)) &&

(channelProcessorTo[k] == nextTaskWay.get(1))) ||

((channelProcessorTo[k] == nextTaskWay.get(0)) &&

(channelProcessorFrom[k] == nextTaskWay.get(1)))) {

channelNumber = k;

}

k++;

}

channelFromQueue.get(channelNumber).add(channelFrom[i]);

channelToQueue.get(channelNumber).add(channelTo[i]);

channelWayQueue.get(channelNumber).add(nextTaskWay);

channelWeightQueue.get(channelNumber).add(channelTaskWeightBackup[i]);

channelTaskWeightBackup[i] = 0;

startNextTact.get(channelNumber).add(true);

}

}

if (channelTaskWeight[i] == 0) {

if ((channelProcessorTo[i] == channelTaskWay.get(i).get(channelTaskWay.get(i).size() - 1)) ||

(channelProcessorFrom[i] == channelTaskWay.get(i).get(channelTaskWay.get(i).size() - 1))) {

connectionMatrix[channelFrom[i]][channelTo[i]] = 0;

}

channelsStatus[i] = false;

}

}

else {

if (!channelWeightQueue.get(i).isEmpty()) {

if (!startNextTact.get(i).get(0)) {

isAlreadySend[i] = false;

channelFrom[i] = channelFromQueue.get(i).get(0);

channelFromQueue.get(i).remove(0);

channelTo[i] = channelToQueue.get(i).get(0);

channelToQueue.get(i).remove(0);

channelTaskWay.set(i, channelWayQueue.get(i).get(0));

channelWayQueue.get(i).remove(0);

channelTaskWeight[i] = channelWeightQueue.get(i).get(0);

channelTaskWeightBackup[i] = channelWeightQueue.get(i).get(0);

channelWeightQueue.get(i).remove(0);

channelsStatus[i] = true;

StringBuilder builder = new StringBuilder();

builder.append(vertexNames[channelFrom[i]]);

builder.append(" -> ");

builder.append(vertexNames[channelTo[i]]);

newRow[i + processorCount + 1] = builder.toString();

channelTaskWeight[i]--;

if (!((channelProcessorTo[i] == channelTaskWay.get(i).get(channelTaskWay.get(i).size() - 1)) ||

(channelProcessorFrom[i] == channelTaskWay.get(i).get(channelTaskWay.get(i).size() - 1)))) {

if (!isAlreadySend[i]) {

isAlreadySend[i] = true;

ArrayList<Integer> nextTaskWay = new ArrayList<Integer>();

for (int l = 1; l < channelTaskWay.get(i).size(); l++) {

nextTaskWay.add(channelTaskWay.get(i).get(l));

}

int channelNumber = -1;

int k = 0;

while ((channelNumber < 0) && (k < channelCount)) {

if (((channelProcessorFrom[k] == nextTaskWay.get(0)) &&

(channelProcessorTo[k] == nextTaskWay.get(1))) ||

((channelProcessorTo[k] == nextTaskWay.get(0)) &&

(channelProcessorFrom[k] == nextTaskWay.get(1)))) {

channelNumber = k;

}

k++;

}

channelFromQueue.get(channelNumber).add(channelFrom[i]);

channelToQueue.get(channelNumber).add(channelTo[i]);

channelWayQueue.get(channelNumber).add(nextTaskWay);

channelWeightQueue.get(channelNumber).add(channelTaskWeightBackup[i]);

channelTaskWeightBackup[i] = 0;

startNextTact.get(channelNumber).add(true);

}

}

if (channelTaskWeight[i] == 0) {

if ((channelProcessorTo[i] == channelTaskWay.get(i).get(channelTaskWay.get(i).size() - 1)) ||

(channelProcessorFrom[i] == channelTaskWay.get(i).get(channelTaskWay.get(i).size() - 1))) {

connectionMatrix[channelFrom[i]][channelTo[i]] = 0;

}

channelsStatus[i] = false;

}

}

}

}

}

model.addRow(newRow);

time++;

}

}

private void sortPopulation(ArrayList<int[]> population, ArrayList<Integer> populationTime) {

for (int i = 0; i < populationTime.size() - 1; i++) {

int minTimeIndex = i;

for (int j = i + 1; j < populationTime.size(); j++) {

if (populationTime.get(j) < populationTime.get(minTimeIndex)) {

minTimeIndex = j;

}

}

if (minTimeIndex != i) {

int tempTime = populationTime.get(i);

int[] tempPopulation = population.get(i);

populationTime.set(i, populationTime.get(minTimeIndex));

population.set(i, population.get(minTimeIndex));

populationTime.set(minTimeIndex, tempTime);

population.set(minTimeIndex, tempPopulation);

}

}

}

private void removeClones(ArrayList<int[]> population, ArrayList<Integer> populationTime) {

for (int i = 0; i < population.size() - 1; i++) {

for (int j = i + 1; j < population.size(); j++) {

boolean isClone = true;

int k = 0;

while ((isClone) && (k < population.get(j).length)) {

if (population.get(j)[k] != population.get(i)[k]) {

isClone = false;

}

k++;

}

if (isClone) {

population.remove(j);

populationTime.remove(j);

}

}

}

}

public GanttTableModel loadGraph(int[][] connectionMatrix, int[] vertexWeights, String[] vertexNames) {

GanttTableModel tableModel = new GanttTableModel(processorCount + channelCount + 1);

String[] columnNamesRow = new String[processorCount + channelCount + 1];

columnNamesRow[0] = "Такт";

int i = 1;

for (int j = 0; j < processorCount; j++) {

StringBuilder builder = new StringBuilder();

builder.append("Процессор ");

builder.append(String.valueOf(j));

columnNamesRow[i] = builder.toString();

i++;

}

int fromProcessor = 0;

int tempCounter = 0;

for (int j = 0; j < channelCount; j++) {

StringBuilder builder = new StringBuilder();

builder.append("Канал ");

builder.append(String.valueOf(fromProcessor));

builder.append("-");

builder.append(String.valueOf(j + 1));

columnNamesRow[i] = builder.toString();

i++;

tempCounter++;

if (tempCounter > 1) {

tempCounter = 0;

fromProcessor++;

}

}

tableModel.addColumnNamesRow(columnNamesRow);

ArrayList<Integer> theBestGenotype = null;

int theBestTime = Integer.MAX\_VALUE;

for (int x = 0; x < STEP\_COUNT; x++) {

ArrayList<int[]> population = new ArrayList<int[]>();

ArrayList<Integer> populationTime = new ArrayList<Integer>();

for (int j = 0; j < populationSize; j++) {

int[] newGenotype = new int[vertexNames.length];

for (int k = 0; k < newGenotype.length; k++) {

int randomValue = (int) (Math.random() \* 1000);

int step = 1000 / processorCount;

int processorNumber = (randomValue / step);

while (processorNumber >= processorCount) {

processorNumber--;

}

newGenotype[k] = processorNumber;

}

population.add(newGenotype);

ArrayList<Integer> newGenotypeList = new ArrayList<Integer>();

for (int k = 0; k < newGenotype.length; k++) {

newGenotypeList.add(newGenotype[k]);

}

populationTime.add(getExecutionTime(newGenotypeList, connectionMatrix, vertexWeights));

}

sortPopulation(population, populationTime);

int[] bestGenotype = population.get(0);

int bestTime = populationTime.get(0);

int bestGenotypeNotChangedCount = 0;

do {

for (int j = 0; j < populationSize; j++) {

int randomValue = (int) (Math.random() \* 1000);

int populationStep = 1000 / populationSize;

int crossIndex = randomValue / populationStep;

if (crossIndex == j) {

if (j > 0) {

crossIndex--;

} else {

crossIndex++;

}

}

while (crossIndex >= population.size()) {

crossIndex--;

}

int[] newGenotype = new int[population.get(j).length];

for (int k = 0; k < population.get(j).length; k++) {

double randomValueDounle = Math.random();

if (randomValueDounle >= CROSS\_PROBABILITY) {

newGenotype[k] = population.get(j)[k];

} else {

newGenotype[k] = population.get(crossIndex)[k];

}

}

population.add(newGenotype);

ArrayList<Integer> newGenotypeList = new ArrayList<Integer>();

for (int k = 0; k < newGenotype.length; k++) {

newGenotypeList.add(newGenotype[k]);

}

populationTime.add(getExecutionTime(newGenotypeList, connectionMatrix, vertexWeights));

}

for (int j = 0; j < populationSize; j++) {

int[] newGenotype = new int[population.get(j).length];

for (int k = 0; k < population.get(j).length; k++) {

double randomValue = Math.random();

if (randomValue < MUTATION\_PROBABILITY) {

int randomValueInt = (int) (Math.random() \* 1000);

int step = 1000 / processorCount;

int processorNumber = (randomValueInt / step);

while (processorNumber >= processorCount) {

processorNumber--;

}

newGenotype[k] = processorNumber;

} else {

newGenotype[k] = population.get(j)[k];

}

}

population.add(newGenotype);

ArrayList<Integer> newGenotypeList = new ArrayList<Integer>();

for (int k = 0; k < newGenotype.length; k++) {

newGenotypeList.add(newGenotype[k]);

}

populationTime.add(getExecutionTime(newGenotypeList, connectionMatrix, vertexWeights));

}

removeClones(population, populationTime);

sortPopulation(population, populationTime);

for (int j = population.size() - 1; j >= populationSize; j--) {

population.remove(j);

populationTime.remove(j);

}

boolean isBestChanged = false;

int j = 0;

while ((!isBestChanged) && (j < population.get(0).length)) {

if (population.get(0)[j] != bestGenotype[j]) {

isBestChanged = true;

}

j++;

}

if (isBestChanged) {

bestGenotype = population.get(0);

bestTime = populationTime.get(0);

bestGenotypeNotChangedCount = 0;

} else {

bestGenotypeNotChangedCount++;

}

}

while (bestGenotypeNotChangedCount < BEST\_GENOTYPE\_NOT\_CHANGED\_MAX\_COUNT);

if (bestTime < theBestTime) {

ArrayList<Integer> bestGenotypeList = new ArrayList<Integer>();

for (int k = 0; k < bestGenotype.length; k++) {

bestGenotypeList.add(bestGenotype[k]);

}

theBestGenotype = bestGenotypeList;

theBestTime = bestTime;

}

}

fillTable(theBestGenotype, connectionMatrix, vertexWeights, vertexNames, tableModel);

return tableModel;

}

}