|  |  |
| --- | --- |
| ***Подп. и дата*** |  |
| ***Инв. № дубл.*** |  |
| ***Взам. инв. №*** |  |
| ***Подп. и дата*** |  |
| ***Инв. № подл*** | RU.17701729.04.13-01 12 01-1-ЛУ |

**ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ  
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ  
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»**

Факультет компьютерных наук

Образовательная программа бакалавриата «Программная инженерия»

|  |  |
| --- | --- |
| СОГЛАСОВАНО  Доцент департамент больших данных и информационного поиска факультета компьютерных наук, к.ф.-м.н.    \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ В. Л. Чернышев «\_\_» \_\_\_\_\_\_\_\_\_\_\_ 2020 г. | УТВЕРЖДАЮ  Академический руководитель образовательной программы «Программная инженерия» профессор департамента программной инженерии, канд. техн. наук  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ В. В. Шилов «\_\_» \_\_\_\_\_\_\_\_\_\_\_ 2020 г. |

**ПРОГРАММА ДЛЯ НАХОЖДЕНИЯ ВРЕМЕНИ НАСЫЩЕНИЯ ДЛЯ ОРИЕНТИРОВАННОГО МЕТРИЧЕСКОГО ГРАФА**

**Текст программы**

**ЛИСТ УТВЕРЖДЕНИЯ**

**RU.17701729.04.13-01 12 01-1-ЛУ**

Исполнитель  
студент группы БПИ196  
\_\_\_\_\_\_\_\_\_\_ / Татаринов Н.А. /  
«\_\_» \_\_\_\_\_\_\_\_\_\_\_ 2020 г.

**Москва 2020**

УТВЕРЖДЕНRU.17701729.04.13-01 12 01-1-ЛУ

|  |  |
| --- | --- |
| Подп. и дата |  |
| Инв. № дубл. |  |
| Взам. инв. № |  |
| Подп. и дата |  |
| Инв. № подл | RU.17701729.04.13-01 12 01-1 |

**Текст программы**

**RU.17701729.04.13-01 12 01-1**

**Листов 56**

**Москва 2020**

**Содержание**

[**1. ПРИЛОЖЕНИЕ WPF (.NET FRAMEWORK) “SimulationWPF”** 3](#_Toc41136929)

[**1.1. Окно (WPF) “MainWindow”** 3](#_Toc41136930)

[**1.1.1. XAML-код “MainWindow.xaml”** 3](#_Toc41136931)

[**1.1.2. Рабочий код “MainWindow.xaml.cs”** 4](#_Toc41136932)

[**1.2. Окно (WPF) “DrawnGraph”** 5](#_Toc41136933)

[**1.2.1. XAML-код “DrawnGraph.xaml”** 5](#_Toc41136934)

[**1.2.2. Рабочий код “DrawnGraph.xaml.cs”** 7](#_Toc41136935)

[**1.3. Окно (WPF) “FileGraph”** 15](#_Toc41136936)

[**1.3.1. XAML-код “FileGraph.xaml”** 15](#_Toc41136937)

[**1.3.2. Рабочий код “FileGraph.xaml.cs”** 16](#_Toc41136938)

[**1.4. Окно (WPF) “Simulation”** 17](#_Toc41136939)

[**1.4.1. XAML-код “Simulation.xaml”** 17](#_Toc41136940)

[**1.4.2. Рабочий код “Simulation.xaml.cs”** 18](#_Toc41136941)

[**1.5. Класс “Edge” (код “Edge.cs”)** 22](#_Toc41136942)

[**1.6. Класс “Vertex” (код “Vertex.cs”)** 26](#_Toc41136943)

[**1.7. Класс “EpsilonNeighbourhood” (код “EpsilonNeighbourhood.cs”)** 28](#_Toc41136944)

[**2. БИБЛИОТЕКА КЛАССОВ (.NET FRAMEWORK) “GraphLib”** 36](#_Toc41136945)

[**2.1. Класс “MetricOrientedGraph” (код “MetricOrientedGraph.cs")** 36](#_Toc41136946)

[**2.2. Класс “Pair” (код “Pair.cs”)** 42](#_Toc41136947)

# **1. ПРИЛОЖЕНИЕ WPF (.NET FRAMEWORK) “SimulationWPF”**

## **1.1. Окно (WPF) “MainWindow”**

### **1.1.1. XAML-код “MainWindow.xaml”**

<Window x:Class="SimulationWPF.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:SimulationWPF"

mc:Ignorable="d"

Title="MainWindow" Height="450" Width="800"

WindowStyle="None" WindowState="Maximized" ResizeMode="NoResize">

<Window.Resources>

<ControlTemplate TargetType="Button" x:Key="EllipseTemplate">

<Border CornerRadius="500" BorderBrush="White" Background="{TemplateBinding Background}">

<ContentControl Content="{TemplateBinding Content}"

HorizontalAlignment="{TemplateBinding HorizontalAlignment}"

VerticalAlignment="{TemplateBinding VerticalAlignment}"

Foreground="White">

</ContentControl>

</Border>

<ControlTemplate.Triggers>

<Trigger Property="IsMouseOver" Value="true">

<Setter Property="Opacity" Value="0.7"></Setter>

</Trigger>

</ControlTemplate.Triggers>

</ControlTemplate>

</Window.Resources>

<Grid x:Name="Choice" Background="White">

<Grid.RowDefinitions>

<RowDefinition/>

<RowDefinition/>

</Grid.RowDefinitions>

<Grid.ColumnDefinitions>

<ColumnDefinition/>

<ColumnDefinition/>

</Grid.ColumnDefinitions>

<TextBlock x:Name="ChoiceInfo" Grid.Column="0" Grid.Row="0" TextWrapping="Wrap"

Background="White" Foreground="Black" FontSize="30"

VerticalAlignment="Center" HorizontalAlignment="Center">

Выберите способ задать граф

</TextBlock>

<Button x:Name="ExitProgram" Background="Black" Foreground="White"

Grid.Column="1" Grid.Row="0" FontSize="30"

Click="ExitProgram\_Click"

Template="{StaticResource EllipseTemplate}">

<ContentControl Content="Выйти из программы" VerticalAlignment="Center"

HorizontalAlignment="Center"></ContentControl>

</Button>

<Button x:Name="FileChoice" Background="Maroon" Foreground="White"

Grid.Column="0" Grid.Row="1" FontSize="30" Click="FileChoice\_Click"

Template="{StaticResource EllipseTemplate}">

<ContentControl Content="Взять граф из файла" VerticalAlignment="Center"

HorizontalAlignment="Center"></ContentControl>

</Button>

<Button x:Name="DrawChoice" Background="DarkGreen" Foreground="White"

Grid.Column="1" Grid.Row="1" FontSize="30" Click="DrawChoice\_Click"

Template="{StaticResource EllipseTemplate}">

<ContentControl Content="Нарисовать граф самому" VerticalAlignment="Center"

HorizontalAlignment="Center"></ContentControl>

</Button>

</Grid>

</Window>

### **1.1.2. Рабочий код “MainWindow.xaml.cs”**

using System.Windows;

namespace SimulationWPF

{

/// <summary>

/// Логика взаимодействия для MainWindow.xaml

/// </summary>

public partial class MainWindow : Window

{

public MainWindow()

{

InitializeComponent();

}

/// <summary>

/// Кнопка для выхода из программы.

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void ExitProgram\_Click(object sender, RoutedEventArgs e)

{

Close();

}

/// <summary>

/// Кнопка для перехода в форму считывания графа из файла.

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void FileChoice\_Click(object sender, RoutedEventArgs e)

{

(new FileGraph()).Show();

Close();

}

/// <summary>

/// Кнопка для перехода в форму рисования графа.

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void DrawChoice\_Click(object sender, RoutedEventArgs e)

{

(new DrawnGraph()).Show();

Close();

}

}

}

## **1.2. Окно (WPF) “DrawnGraph”**

### **1.2.1. XAML-код “DrawnGraph.xaml”**

<Window x:Class="SimulationWPF.DrawnGraph"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:BeginningWPF"

mc:Ignorable="d"

Title="DrawnGraph" Height="450" Width="800"

WindowStyle="None" WindowState="Maximized" ResizeMode="NoResize">

<Window.Resources>

<ControlTemplate TargetType="Button" x:Key="EllipseTemplate">

<Border x:Name="NewButton" CornerRadius="70" BorderBrush="Blue"

Background="{TemplateBinding Background}">

<ContentControl Content="{TemplateBinding Content}"

HorizontalAlignment="{TemplateBinding HorizontalAlignment}"

VerticalAlignment="{TemplateBinding VerticalAlignment}"

Foreground="White">

</ContentControl>

</Border>

<ControlTemplate.Triggers>

<Trigger Property="IsMouseOver" Value="true">

<Setter Property="Opacity" Value="0.8"></Setter>

</Trigger>

<Trigger Property="IsEnabled" Value="false">

<Setter TargetName="NewButton" Property="ItemsControl.Background"

Value="DarkGray"></Setter>

</Trigger>

</ControlTemplate.Triggers>

</ControlTemplate>

</Window.Resources>

<Grid x:Name="Drawn" Background="White">

<Grid.RowDefinitions>

<RowDefinition Height="2\*"/>

<RowDefinition Height="2\*"/>

<RowDefinition Height="2\*"/>

<RowDefinition Height="2\*"/>

<RowDefinition/>

<RowDefinition/>

<RowDefinition/>

</Grid.RowDefinitions>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="10\*"/>

<ColumnDefinition/>

</Grid.ColumnDefinitions>

<Button x:Name="ExitProgram" Background="Black" Foreground="White"

Grid.Column="1" Grid.Row="0" Click="ExitProgram\_Click"

Template="{StaticResource EllipseTemplate}">

<ContentControl Content="Выйти из программы" VerticalAlignment="Center"

HorizontalAlignment="Center"></ContentControl>

</Button>

<Button x:Name="DrawVertices" Background="DarkGreen" Foreground="White"

Grid.Column="1" Grid.Row="1" Click="DrawVertices\_Click"

Template="{StaticResource EllipseTemplate}">

<ContentControl Content="Отмечаем вершины" VerticalAlignment="Center"

HorizontalAlignment="Center"></ContentControl>

</Button>

<Button x:Name="DrawEdges" Background="Maroon" Foreground="White"

Grid.Column="1" Grid.Row="2" Click="DrawEdges\_Click"

Template="{StaticResource EllipseTemplate}">

<ContentControl Content="Рисуем рёбра" VerticalAlignment="Center"

HorizontalAlignment="Center"></ContentControl>

</Button>

<Button x:Name="FindSaturationTime" Background="DarkBlue" Foreground="White"

Grid.Column="1" Grid.Row="3" Click="FindSaturationTime\_Click"

Template="{StaticResource EllipseTemplate}">

<ContentControl Content="Найти время насыщения" VerticalAlignment="Center"

HorizontalAlignment="Center"></ContentControl>

</Button>

<Canvas x:Name="PlaceToDraw" Grid.Row="0" Grid.RowSpan="7"

MouseLeftButtonDown="PlaceToDraw\_MouseLeftButtonDown"

MouseLeftButtonUp="PlaceToDraw\_MouseLeftButtonUp"

MouseMove="PlaceToDraw\_MouseMove" Background="White"/>

<TextBox x:Name="EpsilonTextBox" Grid.Row="4" Grid.Column="1"

TextWrapping="Wrap">

Введите в данное поле величину эпсилон-окрестности.

</TextBox>

<TextBox x:Name="StartVertexTextBox" Grid.Row="5" Grid.Column="1"

TextWrapping="Wrap">

Введите в данное поле номер начальной вершины.

</TextBox>

<TextBox x:Name="FilePathTextBox" Grid.Row="6" Grid.Column="1"

TextWrapping="Wrap">

Введите в данной поле путь для записи графа в файл.

</TextBox>

</Grid>

</Window>

### **1.2.2. Рабочий код “DrawnGraph.xaml.cs”**

using System;

using GraphLib;

using System.Collections.Generic;

using System.Windows;

using System.Windows.Input;

using System.Windows.Media;

namespace SimulationWPF

{

/// <summary>

/// Логика взаимодействия для DrawnGraph.xaml

/// </summary>

public partial class DrawnGraph : Window

{

/// <summary>

/// Количество вершин в графе.

/// </summary>

int amOfVertices = 0;

/// <summary>

/// Количество рёбер в графе.

/// </summary>

int amOfEdges = 0;

/// <summary>

/// Метрический ориентированный граф.

/// </summary>

List<Pair<int, double>>[] graph;

/// <summary>

/// Массив графических вершин.

/// </summary>

Vertex[] drawnVertices;

/// <summary>

/// Массив списков графических рёбер.

/// </summary>

List<Edge>[] drawnEdges;

/// <summary>

/// Индекс вершины для запоминания

/// (перемещение вершин или рисование рёбер).

/// </summary>

int index = -1;

/// <summary>

/// X-координата центра захваченной вершины.

/// </summary>

int startX = -1;

/// <summary>

/// Y-координата центра захваченной вершины.

/// </summary>

int startY = -1;

/// <summary>

/// Инициализирует массивы пустыми

/// элементами.

/// </summary>

public DrawnGraph()

{

InitializeComponent();

PlaceToDraw.IsEnabled = true;

DrawVertices.IsEnabled = false;

graph = new List<Pair<int, double>>[0];

drawnVertices = new Vertex[0];

drawnEdges = new List<Edge>[0];

}

/// <summary>

/// Инициализирует <see cref="graph"/> подготовленным

/// заранее графом <paramref name="\_graph"/>.

/// </summary>

/// <param name="\_graph">

/// Начальный вид графа.

/// </param>

public DrawnGraph(List<Pair<int, double>>[] \_graph)

{

InitializeComponent();

Random rnd = new Random();

PlaceToDraw.IsEnabled = true;

DrawVertices.IsEnabled = false;

graph = \_graph;

amOfVertices = graph.Length;

//Инициализируем массив нарисованных вершин.

drawnVertices = new Vertex[amOfVertices];

for (int i = 0; i < amOfVertices; i++)

{

//Располагаем их случайным образом на PlaceToDraw.

drawnVertices[i] = new Vertex(

rnd.Next(25, 500),

rnd.Next(25, 500),

i + 1);

PlaceToDraw.Children.Add(drawnVertices[i].Ring);

PlaceToDraw.Children.Add(drawnVertices[i].RingContent);

}

//Инициализируем массив списков рисуемых рёбер.

drawnEdges = new List<Edge>[amOfVertices];

//Массив количества рёбер между парой вершин

//(симметричен относительно диагонали).

int[,] amounts = new int[amOfVertices, amOfVertices];

for (int i = 0; i < amOfVertices; i++)

{

drawnEdges[i] = new List<Edge>();

for (int j = 0; j < graph[i].Count; j++, amOfEdges++)

{

//Между i-й и (graph[i][j].First-1)-й вершинами

//появляется +1 ребро.

amounts[i, graph[i][j].First - 1]++;

amounts[graph[i][j].First - 1, i]++;

drawnEdges[i].Add(new Edge(

new Point(drawnVertices[i].X, drawnVertices[i].Y),

new Point(drawnVertices[graph[i][j].First - 1].X,

drawnVertices[graph[i][j].First - 1].Y),

amounts[i, graph[i][j].First - 1]));

drawnEdges[i][j].Weight.Text = graph[i][j].Second.ToString();

PlaceToDraw.Children.Add(drawnEdges[i][j].DrawnEdge);

PlaceToDraw.Children.Add(drawnEdges[i][j].Arrow);

PlaceToDraw.Children.Add(drawnEdges[i][j].Weight);

}

}

}

#region Нажатия на кнопки

/// <summary>

/// Кнопка выхода из программы.

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void ExitProgram\_Click(object sender, RoutedEventArgs e)

{

Close();

}

/// <summary>

/// Кнопка, при нажатии которой взаимодействие с <see cref="PlaceToDraw"/>

/// переходит на режим отмечания (и перемещения) вершин.

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void DrawVertices\_Click(object sender, RoutedEventArgs e)

{

DrawEdges.IsEnabled = true;

DrawVertices.IsEnabled = false;

}

/// <summary>

/// Кнопка, при нажатии которой взаимодействие с <see cref="PlaceToDraw"/>

/// переходит на режим отмечания рёбер.

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void DrawEdges\_Click(object sender, RoutedEventArgs e)

{

DrawVertices.IsEnabled = true;

DrawEdges.IsEnabled = false;

}

/// <summary>

/// Кнопка, при нажатии которой производится попытка получения графа,

/// сохранения его в файл и запуска формы моделирования.

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void FindSaturationTime\_Click(object sender, RoutedEventArgs e)

{

if (amOfEdges == 0)

{

MessageBox.Show("Граф не содержит ни одного ребра. ");

return;

}

//Преобразуем значения TextBox-ов у рисуемых рёбер

//к дествительным числам в диапазоне [1; 100].

for (int i = 0; i < drawnEdges.Length; i++)

for (int j = 0; j < drawnEdges[i].Count; j++)

{

double edgeWeight;

if (!double.TryParse(drawnEdges[i][j].Weight.Text,

out edgeWeight))

{

MessageBox.Show($"Одно из рёбер из вершины {i + 1} " +

$"в вершину {graph[i][j].First} невозможно " +

$"преобразовать к действительному числу. ");

return;

}

if (edgeWeight < 1 || edgeWeight > 100)

{

MessageBox.Show($"Одно из рёбер из вершины {i + 1} " +

$"в вершину {graph[i][j].First} имеет " +

$"некорректный вес (вес должен находиться " +

$"в диапазоне [1; 100]). ");

return;

}

graph[i][j].Second = edgeWeight;

}

//Получаем значение для epsilon.

double epsilon;

if (!double.TryParse(EpsilonTextBox.Text, out epsilon))

{

MessageBox.Show("Невозможно преобразовать " +

"текстовое поле для эпсилон-окрестности в " +

"действительное число. ");

return;

}

if (epsilon <= 0 || epsilon >= 0.5)

{

MessageBox.Show("Эпсилон-окрестность должна " +

"находиться в диапазоне (0; 0.5). ");

return;

}

//Получаем номер стартовой вершины.

int startVertex;

if (!int.TryParse(StartVertexTextBox.Text, out startVertex))

{

MessageBox.Show("Невозможно преобразовать " +

"текстовое поле для стартовой вершины в " +

"целое число. ");

return;

}

if (startVertex < 1 || startVertex > graph.Length)

{

MessageBox.Show("Номер начальной вершины должен " +

"быть от 1 до количества вершиин включительно. ");

return;

}

//Проверяем, является ли граф сильно-связным.

if (MetricOrientedGraph.FindAmountOfStronglyConnectedComponents(graph) > 1)

{

MessageBox.Show("Граф не является сильно-связным. ");

return;

}

//Пытаемся сохранить граф в файл.

try

{

MetricOrientedGraph.SaveGraphToFile(FilePathTextBox.Text, graph);

}

catch (Exception ex)

{

MessageBox.Show("При попытке записать граф в файл \"" +

FilePathTextBox.Text + "\" возникло искючение типа " +

ex.GetType() + ": " + Environment.NewLine + "\"" +

ex.Message + "\". ");

return;

}

//Удаляем из PlaceToDraw все рисуемые рёбра и вершины

//(высвобождаем их для другого Canvas).

for (int i = 0; i < drawnVertices.Length; i++)

{

PlaceToDraw.Children.Remove(drawnVertices[i].Ring);

PlaceToDraw.Children.Remove(drawnVertices[i].RingContent);

}

for (int i = 0; i < drawnEdges.Length; i++)

for (int j = 0; j < drawnEdges[i].Count; j++)

{

PlaceToDraw.Children.Remove(drawnEdges[i][j].Arrow);

PlaceToDraw.Children.Remove(drawnEdges[i][j].Weight);

PlaceToDraw.Children.Remove(drawnEdges[i][j].DrawnEdge);

}

//Запускаем симуляцию.

(new Simulation(graph, drawnVertices, drawnEdges, epsilon,

startVertex)).Show();

//Закрываем форму.

Close();

}

#endregion

#region События мыши.

/// <summary>

/// События опускания левой кнопки мыши.

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void PlaceToDraw\_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)

{

//Если активирована кнопка DrawEdges, то пользователь

//отмечает (перемещает) вершины.

if (DrawEdges.IsEnabled)

{

//В цикле проверяем, не попал ли пользователь при нажатии на

//одну из вершин. Если попал, запоминаем индекс и координаты

//центра вершины для перемещения.

for (int i = 0; i < amOfVertices; i++)

if ((e.GetPosition(this).X - drawnVertices[i].X) \*

(e.GetPosition(this).X - drawnVertices[i].X) +

(e.GetPosition(this).Y - drawnVertices[i].Y) \*

(e.GetPosition(this).Y - drawnVertices[i].Y) <=

25 \* 25)

{

index = i;

startX = drawnVertices[i].X;

startY = drawnVertices[i].Y;

break;

}

//Проверяем, запомнили ли вершинку.

if (index != -1)

{

//Отмечаем выбранную вершинку особым цветом.

drawnVertices[index].Ring.Stroke = Brushes.Cyan;

//Захватываем курсор мыши для отслеживания её координат.

Mouse.Capture(PlaceToDraw);

}

else

{

/\*Так как не попали ни в какую вершину, отмечаем новую

\* на некоторых условиях: должна быть не ближе определённого

\* расстояния от других вершин и не должна заходить за рамки

\* PlaceToDraw.

\*/

for (int i = 0; i < amOfVertices; i++)

if ((e.GetPosition(this).X - drawnVertices[i].X) \*

(e.GetPosition(this).X - drawnVertices[i].X) +

(e.GetPosition(this).Y - drawnVertices[i].Y) \*

(e.GetPosition(this).Y - drawnVertices[i].Y) <=

60 \* 60)

return;

if (e.GetPosition(this).X < 25 ||

e.GetPosition(this).X > PlaceToDraw.ActualWidth - 25 ||

e.GetPosition(this).Y < 25 ||

e.GetPosition(this).Y > PlaceToDraw.ActualHeight)

return;

//Добавляем новую вершину в graph.

Array.Resize(ref graph, amOfVertices + 1);

//Инициализируем список рёбер из новой вершины.

graph[amOfVertices] = new List<Pair<int, double>>();

//Добавляем новую рисуемую вершину в drawnVertices.

Array.Resize(ref drawnVertices, amOfVertices + 1);

//Координаты берём по курсору, номер по количеству вершин.

drawnVertices[amOfVertices] =

new Vertex((int)e.GetPosition(this).X,

(int)e.GetPosition(this).Y,

drawnVertices.Length);

//Добавляем новую вершину в drawnEdges.

Array.Resize(ref drawnEdges, amOfVertices + 1);

//Инициализируем список рисуемых рёбер из новой вершины.

drawnEdges[amOfVertices] = new List<Edge>();

//Добавляем вершину в PlaceToDraw.

PlaceToDraw.Children.Add(drawnVertices[amOfVertices].Ring);

PlaceToDraw.Children.Add(drawnVertices[amOfVertices].RingContent);

//Увеличиваем количество вершин.

amOfVertices++;

}

}

//Кнопка DrawEdges неактивна, значит, добавляем рёбра.

else

{

//Проверяем, была ли запомнена какая-либо вершина.

if (index == -1)

{

//Если нет, то проходимся по вершинам и запоминаем

//какую-то, если в неё попали. Вершину помечаем

//отличающимся цветом.

for (int i = 0; i < amOfVertices; i++)

if ((e.GetPosition(this).X - drawnVertices[i].X) \*

(e.GetPosition(this).X - drawnVertices[i].X) +

(e.GetPosition(this).Y - drawnVertices[i].Y) \*

(e.GetPosition(this).Y - drawnVertices[i].Y) <=

25 \* 25)

{

index = i;

drawnVertices[index].Ring.Stroke = Brushes.Blue;

break;

}

}

else

{

//Если да, то опять проходимся по вершинамю

for (int i = 0; i < amOfVertices; i++)

if ((e.GetPosition(this).X - drawnVertices[i].X) \*

(e.GetPosition(this).X - drawnVertices[i].X) +

(e.GetPosition(this).Y - drawnVertices[i].Y) \*

(e.GetPosition(this).Y - drawnVertices[i].Y) <=

25 \* 25)

{

//Если отметили ещё одну, то нужно нарисовать между ними

//ребро (петли не предусмотрены).

if (index == i) break;

//Добавляем ребро в graph и получаем индекс на котором

//ребро стоит.

int curIndex = MetricOrientedGraph.AddEdgeToGraph(graph,

index + 1, new Pair<int, double>(i + 1, 0));

//Считаем количество рёбер между двумя рассматриваемыми вершинами.

int amOfEdgesBetweenVertices = 0;

foreach (Pair<int, double> edge in graph[index])

if (edge.First == i + 1) amOfEdgesBetweenVertices++;

foreach (Pair<int, double> edge in graph[i])

if (edge.First == index + 1) amOfEdgesBetweenVertices++;

//Вставляем по найдённому индексу новое рисуемое ребро

//(в качестве специального числа используем количество

//уже существущих между данными вершинами рёбер.

drawnEdges[index].Insert(curIndex, new Edge(

new Point(drawnVertices[index].X, drawnVertices[index].Y),

new Point(drawnVertices[i].X, drawnVertices[i].Y),

amOfEdgesBetweenVertices));

//Добавляем ребро в PlaceToDraw.

PlaceToDraw.Children.Add(

drawnEdges[index][curIndex].DrawnEdge);

PlaceToDraw.Children.Add(

drawnEdges[index][curIndex].Arrow);

PlaceToDraw.Children.Add(

drawnEdges[index][curIndex].Weight);

//Увеличиваем количество рёбер на 1 и выходим.

amOfEdges++;

break;

}

//Освобождаем индекс и возвращаем вершине прежний цвет.

drawnVertices[index].Ring.Stroke = Brushes.Goldenrod;

index = -1;

}

}

}

/// <summary>

/// Событие перемещения мыши.

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void PlaceToDraw\_MouseMove(object sender, MouseEventArgs e)

{

//Если курсор не захвачен (то есть не нужно отслеживать его кооринаты),

//то выходим.

if (!PlaceToDraw.IsMouseCaptured) return;

//В противном случае, мы перемещаем вершину,

//то есть меняем координаты её центра в соответствии

//с координатами курсора.

drawnVertices[index].X = (int)e.GetPosition(this).X;

drawnVertices[index].Y = (int)e.GetPosition(this).Y;

}

/// <summary>

/// Событи поднятия левой кнопки мыши.

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void PlaceToDraw\_MouseLeftButtonUp(object sender, MouseButtonEventArgs e)

{

//Если мышь не захвачена (то есть не нужно отслеживать её кооринаты),

//то выходим.

if (!PlaceToDraw.IsMouseCaptured) return;

//В противном случае, мы знаем, что переместили какую-то вершину.

//Проверяем, не слишком ли близко она находится к другим вершинам

//(если слишком, то возвращаем её назад).

for (int i = 0; i < amOfVertices; i++)

if (index != i &&

(e.GetPosition(this).X - drawnVertices[i].X) \*

(e.GetPosition(this).X - drawnVertices[i].X) +

(e.GetPosition(this).Y - drawnVertices[i].Y) \*

(e.GetPosition(this).Y - drawnVertices[i].Y) <=

60 \* 60)

{

drawnVertices[index].X = startX;

drawnVertices[index].Y = startY;

break;

}

//Проверяем, не слишком ли близко она находится к границам PlaceToDraw

//(если слишком, то возвращаем её назад).

if (e.GetPosition(this).X < 25 ||

e.GetPosition(this).Y < 25 ||

e.GetPosition(this).X > PlaceToDraw.ActualWidth - 25 ||

e.GetPosition(this).Y > PlaceToDraw.ActualHeight - 25)

{

drawnVertices[index].X = startX;

drawnVertices[index].Y = startY;

}

//Присоединяем к ней все рёбра, выходящие из неё.

for (int i = 0; i < graph[index].Count; i++)

drawnEdges[index][i].First = new Point(

drawnVertices[index].X,

drawnVertices[index].Y);

//Присоединяем к ней все рёбра, входящие в неё.

for (int i = 0; i < amOfVertices; i++)

{

if (i == index) continue;

for (int j = 0; j < graph[i].Count; j++)

if (graph[i][j].First == index + 1)

drawnEdges[i][j].Final = new Point(

drawnVertices[index].X,

drawnVertices[index].Y);

}

//Возвращаем вершине прежний цвет.

drawnVertices[index].Ring.Stroke = Brushes.Goldenrod;

//Высвобождаем индекс и координаты.

index = -1;

startX = -1;

startY = -1;

//Высвобождаем курсор.

Mouse.Capture(null);

}

#endregion

}

}

## **1.3. Окно (WPF) “FileGraph”**

### **1.3.1. XAML-код “FileGraph.xaml”**

<Window x:Class="SimulationWPF.FileGraph"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:SimulationWPF"

mc:Ignorable="d"

Title="FileGraph" Height="450" Width="800"

WindowStyle="None" WindowState="Maximized" ResizeMode="NoResize">

<Window.Resources>

<ControlTemplate TargetType="Button" x:Key="EllipseTemplate">

<Border CornerRadius="500" BorderBrush="Blue" Background="{TemplateBinding Background}">

<ContentControl Content="{TemplateBinding Content}"

HorizontalAlignment="{TemplateBinding HorizontalAlignment}"

VerticalAlignment="{TemplateBinding VerticalAlignment}"

Foreground="White">

</ContentControl>

</Border>

<ControlTemplate.Triggers>

<Trigger Property="IsMouseOver" Value="true">

<Setter Property="Opacity" Value="0.8"></Setter>

</Trigger>

</ControlTemplate.Triggers>

</ControlTemplate>

</Window.Resources>

<Grid>

<Grid.RowDefinitions>

<RowDefinition/>

<RowDefinition/>

</Grid.RowDefinitions>

<Grid.ColumnDefinitions>

<ColumnDefinition/>

<ColumnDefinition/>

</Grid.ColumnDefinitions>

<Button x:Name="ExitProgram" Background="Black" Foreground="White"

Grid.Column="1" Grid.Row="1" FontSize="30" Click="ExitProgram\_Click"

Template="{StaticResource EllipseTemplate}">

<ContentControl Content="Выйти из программы" VerticalAlignment="Center"

HorizontalAlignment="Center"></ContentControl>

</Button>

<Button x:Name="GetGraph" Background="Maroon" Foreground="White"

Grid.Column="0" Grid.Row="1" FontSize="30" Click="GetGraph\_Click"

Template="{StaticResource EllipseTemplate}">

<ContentControl Content="Получить граф" VerticalAlignment="Center"

HorizontalAlignment="Center"></ContentControl>

</Button>

<TextBox x:Name="FilePathTextBox" Grid.Row="0" Grid.ColumnSpan="2"

TextWrapping="Wrap" FontSize="25">

Введите в данное поле имя файла, из которого необходимо считать граф.

</TextBox>

</Grid>

</Window>

### **1.3.2. Рабочий код “FileGraph.xaml.cs”**

using System;

using GraphLib;

using System.Collections.Generic;

using System.Windows;

namespace SimulationWPF

{

/// <summary>

/// Логика взаимодействия для FileGraph.xaml

/// </summary>

public partial class FileGraph : Window

{

public FileGraph()

{

InitializeComponent();

}

/// <summary>

/// Кнопка для попытки считывания графа из файла (путь взят

/// из <see cref="FilePathTextBox"/>) для перехода в

/// форму рисования.

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void GetGraph\_Click(object sender, RoutedEventArgs e)

{

//Блокируем запись в FilePathTextBox на время чтения графа.

FilePathTextBox.IsEnabled = false;

List<Pair<int, double>>[] graph;

try

{

graph = MetricOrientedGraph.ReadGraphFromFile(

FilePathTextBox.Text);

}

catch (Exception ex)

{

MessageBox.Show("При попытке получить граф из файла \"" +

FilePathTextBox.Text + "\" возникло искючение типа " +

ex.GetType() + ": " + Environment.NewLine + "\"" +

ex.Message + "\". ");

//Разблокируем, так как придётся опять читать.

FilePathTextBox.IsEnabled = true;

return;

}

//Переходим в форму рисования графа.

(new DrawnGraph(graph)).Show();

Close();

}

/// <summary>

/// Кнопка для выхода из программы.

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void ExitProgram\_Click(object sender, RoutedEventArgs e)

{

Close();

}

}

}

## **1.4. Окно (WPF) “Simulation”**

### **1.4.1. XAML-код “Simulation.xaml”**

<Window x:Class="SimulationWPF.Simulation"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:SimulationWPF"

mc:Ignorable="d"

Title="Simulation" Height="450" Width="800"

WindowStyle="None" WindowState="Maximized" ResizeMode="NoResize">

<Window.Resources>

<ControlTemplate TargetType="Button" x:Key="EllipseTemplate">

<Border x:Name="NewButton" CornerRadius="70" BorderBrush="Blue"

Background="{TemplateBinding Background}">

<ContentControl Content="{TemplateBinding Content}"

HorizontalAlignment="{TemplateBinding HorizontalAlignment}"

VerticalAlignment="{TemplateBinding VerticalAlignment}"

Foreground="White">

</ContentControl>

</Border>

<ControlTemplate.Triggers>

<Trigger Property="IsMouseOver" Value="true">

<Setter Property="Opacity" Value="0.8"></Setter>

</Trigger>

</ControlTemplate.Triggers>

</ControlTemplate>

</Window.Resources>

<Grid>

<Grid.RowDefinitions>

<RowDefinition Height="10\*"/>

<RowDefinition/>

<RowDefinition/>

</Grid.RowDefinitions>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="10\*"/>

<ColumnDefinition/>

</Grid.ColumnDefinitions>

<Canvas x:Name="SimulationCanvas" Grid.Row="0" Grid.RowSpan="3"/>

<Button x:Name="ExitProgram" Background="Black" Foreground="White"

Grid.Column="1" Grid.Row="2" Click="ExitProgram\_Click"

Template="{StaticResource EllipseTemplate}">

<ContentControl Content="Выйти из программы" VerticalAlignment="Center"

HorizontalAlignment="Center"></ContentControl>

</Button>

<Button x:Name="SetSimulationBtn" Background="Maroon" Foreground="White"

Grid.Column="1" Grid.Row="1" Click="SetSimulationBtn\_Click"

Template="{StaticResource EllipseTemplate}">

<ContentControl Content="Запустить моделирование" VerticalAlignment="Center"

HorizontalAlignment="Center"></ContentControl>

</Button>

<TextBox x:Name="InfoTextBox" Grid.Row="0" Grid.Column="1" TextWrapping="Wrap"

FontSize="20">

Время: 0

</TextBox>

</Grid>

</Window>

### **1.4.2. Рабочий код “Simulation.xaml.cs”**

using System;

using GraphLib;

using System.Collections.Generic;

using System.Threading.Tasks;

using System.Windows;

using System.Windows.Threading;

using System.Threading;

namespace SimulationWPF

{

/// <summary>

/// Логика взаимодействия для Simulation.xaml

/// </summary>

public partial class Simulation : Window

{

/// <summary>

/// Метрический неориентированный граф.

/// </summary>

List<Pair<int, double>>[] graph;

/// <summary>

/// Массив графических вершин.

/// </summary>

Vertex[] drawnVertices;

/// <summary>

/// Массив списков графических рёбер.

/// </summary>

List<Edge>[] drawnEdges;

/// <summary>

/// Список эпсилон-окрестностей.

/// </summary>

List<EpsilonNeighbourhood> neighs;

/// <summary>

/// Величина эпсилонокрестности.

/// </summary>

double epsilon;

/// <summary>

/// Таймер для смещения всех эпсилон окрестностей.

/// </summary>

DispatcherTimer tmr;

/// <summary>

/// Время, выводимое пользователю.

/// </summary>

double timeForUser = 0;

/// <summary>

/// Инициализирует поля входными параметрами.

/// </summary>

/// <param name="\_graph">

/// Метрический оиентированный граф.

/// </param>

/// <param name="\_drawnVertices">

/// Массив графических вершин.

/// </param>

/// <param name="\_drawnEdges">

/// Массив списков графических рёбер.

/// </param>

/// <param name="\_epsilon">

/// Величина эпсилон-окрестности.

/// </param>

/// <param name="\_startVertex">

/// Номер начальной вершины.

/// </param>

public Simulation(List<Pair<int, double>>[] \_graph,

Vertex[] \_drawnVertices, List<Edge>[] \_drawnEdges,

double \_epsilon, int \_startVertex)

{

//Запоминаем переданные напрямую параметры.

InitializeComponent();

graph = \_graph;

drawnVertices = \_drawnVertices;

drawnEdges = \_drawnEdges;

epsilon = \_epsilon;

neighs = new List<EpsilonNeighbourhood>();

//Добавляем вершины в SimulationCanvas.

for (int i = 0; i < drawnVertices.Length; i++)

{

SimulationCanvas.Children.Add(drawnVertices[i].Ring);

SimulationCanvas.Children.Add(drawnVertices[i].RingContent);

}

//Добавляем рёбра в SimulationCanvas.

for (int i = 0; i < drawnEdges.Length; i++)

for (int j = 0; j < drawnEdges[i].Count; j++)

{

drawnEdges[i][j].Weight.IsEnabled = false;

SimulationCanvas.Children.Add(drawnEdges[i][j].DrawnEdge);

SimulationCanvas.Children.Add(drawnEdges[i][j].Weight);

SimulationCanvas.Children.Add(drawnEdges[i][j].Arrow);

}

//Добавляем первые эпсилон-окрестности в SimulationCanvas.

for (int i = 0; i < graph[\_startVertex - 1].Count; i++)

{

neighs.Add(new EpsilonNeighbourhood(

drawnEdges[\_startVertex - 1][i].First,

drawnEdges[\_startVertex - 1][i].Middle,

drawnEdges[\_startVertex - 1][i].Final,

\_startVertex - 1, i, graph[\_startVertex - 1][i].Second,

epsilon, 2 \* epsilon, 2 \* epsilon / 3));

SimulationCanvas.Children.Add(neighs[i].Neighbourhood);

}

//Отключаем возможность редактировать текст в

//информационном блоке текста.

InfoTextBox.IsEnabled = false;

//Включаем таймер, при каждом тике которого

//происходит смещение всех эпсилон-окрестностей.

tmr = new DispatcherTimer();

tmr.Interval = new TimeSpan(1);

tmr.Tick += TmrTick;

}

/// <summary>

/// Кнопка, останавливающая и возобновляющая моделирования.

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void SetSimulationBtn\_Click(object sender, RoutedEventArgs e)

{

if (tmr.IsEnabled)

tmr.Stop();

else

tmr.Start();

}

/// <summary>

/// Вызывает при каждом тике таймера <see cref="tmr"/>

/// для смещения всех эпсилон-окрестностей на шаг.

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private async void TmrTick(object sender, EventArgs e)

{

/\*Выключаем таймер, чтобы в процессе асинхронного смещения

всех окрестностей данный метод не был вызван ещё раз, то

привело бы к накладыванию добавления элементов в

SimulationCanvas и к некорректному вычислению времени

насыщения.\*/

tmr.Stop();

//Удаляем все элементы из SimulationCanvas.

//В противном случае затеряются те эпсилон-окресности,

//которые необходимо удалить.

for (int i = 0; i < neighs.Count; i++)

SimulationCanvas.Children.Remove(neighs[i].Neighbourhood);

try

{

//Выводим текущее значение времени.

timeForUser += checked(2 \* epsilon / 3);

InfoTextBox.Text = $"Время: {timeForUser}";

//Вызываем смещение всех эпсилон-окрестностей в другом потоке, чтобы

//основной поток не зависал и была возможность остановить моделирование

//или выйти из программы.

neighs = await Task.Run(() =>

Application.Current.Dispatcher.Invoke(MoveAllNeighs));

}

catch (Exception ex)

{

//Не хватило ресурсов для дальнейшего моделирования.

tmr.Stop();

MessageBox.Show("Дальнейшее моделирование невозможно " +

"по следующей причине : " + Environment.NewLine +

ex.Message + Environment.NewLine +

"Программа закроется через 10 секунд. ");

Thread.Sleep(10000);

Close();

}

//Добавляем все элементы в SimulationCanvas.

for (int i = 0; i < neighs.Count; i++)

SimulationCanvas.Children.Add(neighs[i].Neighbourhood);

//Запускаем заново таймер.

tmr.Start();

}

/// <summary>

/// Метод смещения всех эпсилон-окрестностей на шаг.

/// </summary>

/// <returns></returns>

private List<EpsilonNeighbourhood> MoveAllNeighs()

{

//Заводим список новых эпсилон-окрестностей.

List<EpsilonNeighbourhood> newNeighs =

new List<EpsilonNeighbourhood>();

for (int i = 0; i < neighs.Count; i++)

{

//Смещаем очередную эпсилон-окрестность

//и получаем число, определяющее дальнейшие действия.

double newEps = neighs[i].MoveNeighbourhood();

if (newEps >= 0)

{

//Добавляем текущую окрестность и все,

//выходящие из вершины, в которую

//текущая пришла.

newNeighs.Add(neighs[i]);

int vertexIndex = graph[neighs[i].EdgeIndexI]

[neighs[i].EdgeIndexJ].First - 1;

for (int j = 0; j < drawnEdges[vertexIndex].Count; j++)

newNeighs.Add(new EpsilonNeighbourhood(

drawnEdges[vertexIndex][j].First,

drawnEdges[vertexIndex][j].Middle,

drawnEdges[vertexIndex][j].Final,

vertexIndex, j, graph[vertexIndex][j].Second,

newEps, 2 \* epsilon, 2 \* epsilon / 3));

}

else if (newEps == -2)

//Добавляем только текущую.

newNeighs.Add(neighs[i]);

//Если вернулась (-1), то вершину не нужно обавлять.

}

return newNeighs;

}

/// <summary>

/// Кнопка для выхода из программы.

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void ExitProgram\_Click(object sender, RoutedEventArgs e)

{

Close();

}

}

}

## **1.5. Класс “Edge” (код “Edge.cs”)**

using System;

using System.Windows;

using System.Windows.Controls;

using System.Windows.Media;

using System.Windows.Shapes;

namespace SimulationWPF

{

/// <summary>

/// Класс, представляющий нарисованное в

/// <see cref="Canvas"/> ребро ориентированного

/// метрического графа.

/// </summary>

public class Edge

{

#region Точки ребра.

/// <summary>

/// Поле для свойства <see cref="First"/>.

/// </summary>

Point first;

/// <summary>

/// Точка центра начальной вершины графа.

/// </summary>

/// <remarks>

/// Поле для данного свойства - <see cref="first"/>.

/// </remarks>

public Point First

{

get { return first; }

set

{

//Запоминаем новые координаты центра начальной вершины.

first = value;

//Пересчитываем смещение для промеежуточной точки.

CalculateDelta();

//Смещаем центральную точку по новым переменным смещения

//и по специальному числу Number.

if (Number % 2 == 0)

Middle = new Point(

(first.X + final.X) / 2 + deltaX \* (Number / 2),

(first.Y + final.Y) / 2 + deltaY \* (Number / 2));

else

Middle = new Point(

(first.X + final.X) / 2 - deltaX \* (Number / 2),

(first.Y + final.Y) / 2 - deltaY \* (Number / 2));

//Переопределяем точки у DrawnEdge.

DrawnEdge.Points[0] = first;

DrawnEdge.Points[1] = Middle;

//Переопределяем стрелку направления.

CreateArrow();

//Переопределяем координаты для TextBox веса

//ребра Weight.

Canvas.SetLeft(Weight, Middle.X);

Canvas.SetTop(Weight, Middle.Y);

}

}

/// <summary>

/// Промежуточная точка ребра (определяется автоматически

/// по заданным начальной и конечной точке).

/// </summary>

public Point Middle { get; private set; }

/// <summary>

/// Поле для свойства <see cref="Final"/>.

/// </summary>

Point final;

/// <summary>

/// Точка центра начальной вершины графа.

/// </summary>

/// <remarks>

/// Поле для данного свойства - <see cref="first"/>.

/// </remarks>

public Point Final

{

get { return final; }

set

{

//Запоминаем новые координаты центра конечной вершины.

final = value;

//Пересчитываем смещение для промежуточной точки.

CalculateDelta();

//Смещаем центральную точку по новым переменным смещения

//и по специальному числу Number.

if (Number % 2 == 0)

Middle = new Point(

(first.X + final.X) / 2 + deltaX \* (Number / 2),

(first.Y + final.Y) / 2 + deltaY \* (Number / 2));

else

Middle = new Point(

(first.X + final.X) / 2 - deltaX \* (Number / 2),

(first.Y + final.Y) / 2 - deltaY \* (Number / 2));

//Переопределяем точки у DrawnEdge.

DrawnEdge.Points[1] = Middle;

DrawnEdge.Points[2] = final;

//Переопределяем стрелку направления.

CreateArrow();

//Переопределяем координаты для TextBox веса

//ребра Weight.

Canvas.SetLeft(Weight, Middle.X);

Canvas.SetTop(Weight, Middle.Y);

}

}

#endregion

/// <summary>

/// Переменная смещения промежуточной точки по оси OX

/// относительно середины отрезка между

/// <see cref="First"/> и <see cref="Final"/>.

/// </summary>

double deltaX;

/// <summary>

/// Переменная смещения промежуточной точки по оси OY

/// относительно середины отрезка между

/// <see cref="First"/> и <see cref="Final"/>.

/// </summary>

double deltaY;

/// <summary>

/// Специальное число, задающее расположение

/// промежуточной точки.

/// </summary>

public int Number { get; }

/// <summary>

/// Ребро графа (элемент для <see cref="Canvas"/>).

/// </summary>

public Polyline DrawnEdge { get; }

/// <summary>

/// Вес ребра (элемент для <see cref="Canvas"/>).

/// </summary>

public TextBox Weight { get; set; }

/// <summary>

/// Стрелка направления ребра (элемент для <see cref="Canvas"/>).

/// </summary>

public Polygon Arrow { get; }

/// <summary>

/// Создаёт ребро графа по начальной точке <paramref name="\_first"/>,

/// конечной точке <paramref name="\_last"/> специальному числу

/// <paramref name="\_number"/>.

/// </summary>

/// <param name="\_first">

/// Точка центра начальной вершины графа.

/// </param>

/// <param name="\_last">

/// Точка центра конечной вершины графа.

/// </param>

/// <param name="\_number">

/// Специальное число, задающее расположение промежуточной точки.

/// </param>

public Edge(Point \_first, Point \_last, int \_number)

{

//Запоминаем переданные данные.

first = \_first;

final = \_last;

Number = \_number;

//Вычисляем переменные смещения.

CalculateDelta();

//Вычислем координаты промежуточной точки.

if (Number % 2 == 0)

Middle = new Point(

(First.X + Final.X) / 2 + deltaX \* (Number / 2),

(First.Y + Final.Y) / 2 + deltaY \* (Number / 2));

else

Middle = new Point(

(first.X + final.X) / 2 - deltaX \* (Number / 2),

(first.Y + final.Y) / 2 - deltaY \* (Number / 2));

//Создаём ребро графа.

DrawnEdge = new Polyline();

DrawnEdge.Points = new PointCollection(new Point[] { First, Middle, Final });

DrawnEdge.Stroke = Brushes.DarkGoldenrod;

DrawnEdge.StrokeThickness = 1;

//Создаём стрелку направления ребра.

Arrow = new Polygon();

Arrow.Fill = Brushes.DarkGoldenrod;

CreateArrow();

//Создаём TextBox для веса ребра.

Weight = new TextBox();

Weight.Background = Brushes.White;

Weight.BorderBrush = Brushes.LightGray;

Weight.FontSize = 14;

Canvas.SetLeft(Weight, Middle.X);

Canvas.SetTop(Weight, Middle.Y);

Weight.Text = "0";

}

/// <summary>

/// Вычисляет переменные смещения для промежуточной точки.

/// </summary>

void CalculateDelta()

{

//Делим отрезок First-Final в отношении 9:10.

Point p1 = new Point(

(10 \* First.X + 9 \* Final.X) / 19,

(10 \* First.Y + 9 \* Final.Y) / 19);

//Делим отрезок First-Final в отношении 10:9.

Point p2 = new Point(

(9 \* First.X + 10 \* Final.X) / 19,

(9 \* First.Y + 10 \* Final.Y) / 19);

//Поворачиваем вектор p1-p2 на (PI/3)

//против часовой стрелки.

Point p3 = new Point(

p1.X + (p2.X - p1.X) \* Math.Cos(Math.PI / 3) -

(p2.Y - p1.Y) \* Math.Sin(Math.PI / 3),

p1.Y + (p2.X - p1.X) \* Math.Sin(Math.PI / 3) +

(p2.Y - p1.Y) \* Math.Cos(Math.PI / 3));

//Вычисляем переменные смещения как разность

//между координатой полученной p3 и серединой

//отрезка First-Final.

deltaX = p3.X - (First.X + Final.X) / 2;

deltaY = p3.Y - (First.Y + Final.Y) / 2;

//deltaX всегда должна быть положительной для

//корректного вычисления промежуточной точки.

if (deltaX < 0)

{

deltaX = -deltaX;

deltaY = -deltaY;

}

}

/// <summary>

/// Создаёт стрелку направления для ребра.

/// </summary>

void CreateArrow()

{

//Делим отрезок Middle-Final в отношении 4:5.

Point p1 = new Point(

(5 \* Middle.X + 4 \* Final.X) / 9,

(5 \* Middle.Y + 4 \* Final.Y) / 9);

//Делим отрезок Middle-Final в отношении 5:4.

Point p2 = new Point(

(4 \* Middle.X + 5 \* Final.X) / 9,

(4 \* Middle.Y + 5 \* Final.Y) / 9);

//Поворачиваем вектор p2-p1 на (PI/12)

//по часовой стрелке.

Point p3 = new Point(

p2.X + (p1.X - p2.X) \* Math.Cos(Math.PI / 12) -

(p1.Y - p2.Y) \* Math.Sin(Math.PI / 12),

p2.Y + (p1.X - p2.X) \* Math.Sin(Math.PI / 12) +

(p1.Y - p2.Y) \* Math.Cos(Math.PI / 12));

//Поворачиваем вектор p2-p1 на (PI/12)

//против часовой стрелки.

Point p4 = new Point(

p2.X + (p1.X - p2.X) \* Math.Cos(-Math.PI / 12) -

(p1.Y - p2.Y) \* Math.Sin(-Math.PI / 12),

p2.Y + (p1.X - p2.X) \* Math.Sin(-Math.PI / 12) +

(p1.Y - p2.Y) \* Math.Cos(-Math.PI / 12));

//Рисуем треугольник.

Arrow.Points = new PointCollection(new Point[] { p3, p4, p2 });

}

}

}

## **1.6. Класс “Vertex” (код “Vertex.cs”)**

using System.Windows.Controls;

using System.Windows.Media;

using System.Windows.Shapes;

namespace SimulationWPF

{

/// <summary>

/// Класс, представляющий нарисованную в

/// <see cref="Canvas"/> вершину ориентированного

/// метрического графа.

/// </summary>

public class Vertex

{

/// <summary>

/// Поле для свойства <see cref="X"/>.

/// </summary>

int x;

/// <summary>

/// X-координата центра вершины.

/// </summary>

/// <remarks>

/// Поле для данного свойства -

/// <see cref="x"/>.

/// </remarks>

public int X

{

get { return x; }

set

{

x = value;

Canvas.SetLeft(Ring, x - 20);

Canvas.SetLeft(RingContent, x - 10);

}

}

/// <summary>

/// Поле для свойства <see cref="Y"/>.

/// </summary>

int y;

/// <summary>

/// Y-координата центра вершины.

/// </summary>

/// <remarks>

/// Поле для данного свойства -

/// <see cref="y"/>.

/// </remarks>

public int Y

{

get { return y; }

set

{

y = value;

Canvas.SetTop(Ring, y - 20);

Canvas.SetTop(RingContent, y - 11);

}

}

/// <summary>

/// Кольцо, обозначающее границы вершинки (элемент

/// для <see cref="Canvas"/>).

/// </summary>

public Ellipse Ring { get; private set; }

/// <summary>

/// Номер вершины, расположенный внутри кольца

/// <see cref="Ring"/> (элемент для <see cref="Canvas"/>).

/// </summary>

public TextBox RingContent { get; set; }

/// <summary>

/// Инициализируют вершину для графа с координатами центра

/// (<paramref name="\_x"/>, <paramref name="\_y"/>) и с

/// номером <paramref name="\_ringContent"/>.

/// </summary>

/// <param name="\_x">

/// X-координата центра вершины.

/// </param>

/// <param name="\_y">

/// Y-координата центра вершины.

/// </param>

/// <param name="\_ringContent">

/// Номер вершины.

/// </param>

public Vertex(int \_x, int \_y, int \_ringContent)

{

Ring = new Ellipse();

Ring.Height = 40;

Ring.Width = 40;

Ring.Fill = Brushes.LightGray;

Ring.Stroke = Brushes.Goldenrod;

Ring.StrokeThickness = 5;

RingContent = new TextBox();

RingContent.Text = \_ringContent.ToString();

RingContent.Background = Brushes.LightGray;

RingContent.BorderBrush = Brushes.LightGray;

RingContent.FontSize = 14;

RingContent.IsEnabled = false;

X = \_x;

Y = \_y;

}

}

}

## **1.7. Класс “EpsilonNeighbourhood” (код “EpsilonNeighbourhood.cs”)**

using System;

using System.Windows;

using System.Windows.Media;

using System.Windows.Shapes;

namespace SimulationWPF

{

/// <summary>

/// Класс, представляющий нарисованную в

/// <see cref="Canvas"/> эпсилон-окрестность.

/// </summary>

public class EpsilonNeighbourhood

{

/// <summary>

/// Начальная точка ребра, по которому

/// передвигается эпсилон-окрестность.

/// </summary>

public Point FirstPoint { get; }

/// <summary>

/// Промежуточная точка ребра, по которому

/// передвигается эпсилон-окрестность.

/// </summary>

public Point MiddlePoint { get; }

/// <summary>

/// Конечная точка ребра, по которому

/// передвигается эпсилон-окрестность.

/// </summary>

public Point FinalPoint { get; }

/// <summary>

/// i-й индекс ребра, по которому

/// передвигается окрестность.

/// </summary>

public int EdgeIndexI { get; }

/// <summary>

/// j-й индекс ребра, по которому

/// передвигается окрестность.

/// </summary>

public int EdgeIndexJ { get; }

/// <summary>

/// Передвигающаяся по экрану эпсилон-окрестность.

/// </summary>

public Polyline Neighbourhood { get; private set; }

/// <summary>

/// Расположение задней точки окрестности.

/// </summary>

/// <remarks>

/// 0, если совпадает с <see cref="FirstPoint"/>;

/// 1, если находится между <see cref="FirstPoint"/>

/// и <see cref="MiddlePoint"/>;

/// 2, если совпадает с <see cref="MiddlePoint"/>;

/// 3, если находится между <see cref="MiddlePoint"/>

/// и <see cref="FinalPoint"/>;

/// 4, если совпадает с <see cref="FinalPoint"/>.

/// </remarks>

int backPosition = 0;

/// <summary>

/// Расположение передней точки окрестности.

/// </summary>

/// <remarks>

/// 0, если совпадает с <see cref="FirstPoint"/>;

/// 1, если находится между <see cref="FirstPoint"/>

/// и <see cref="MiddlePoint"/>;

/// 2, если совпадает с <see cref="MiddlePoint"/>;

/// 3, если находится между <see cref="MiddlePoint"/>

/// и <see cref="FinalPoint"/>;

/// 4, если совпадает с <see cref="FinalPoint"/>.

/// </remarks>

int frontPosition = 1;

/// <summary>

/// Длина ребра на экране.

/// </summary>

double drawnEdgeLength;

/// <summary>

/// Длина эпсилон-окрестности на экране.

/// </summary>

double epsilon;

/// <summary>

/// Коэффициент отношения линейных значений

/// на экране к номинальным линейным значениям.

/// </summary>

double coef;

/// <summary>

/// Смещение по оси OX на первой части

/// ребра при каждом шаге.

/// </summary>

double delta1X;

/// <summary>

/// Смещение по оси OY на первой части

/// ребра при каждом шаге.

/// </summary>

double delta1Y;

/// <summary>

/// Смещение по оси OX на второй части

/// ребра при каждом шаге.

/// </summary>

double delta2X;

/// <summary>

/// Смещение по оси OY на второй части

/// ребра при каждом шаге.

/// </summary>

double delta2Y;

/// <summary>

/// Инициализирует экземпляр эпсилон-окрестности

/// через параметры.

/// </summary>

/// <param name="\_firstPoint">

/// Начальная точка ребра.

/// </param>

/// <param name="\_middlePoint">

/// Промежуточная точка ребра.

/// </param>

/// <param name="\_finalPoint">

/// Конечная точка ребра.

/// </param>

/// <param name="\_edgeIndexI">

/// i-индекс ребра, на котором находися

/// эпсилон-окрестность.

/// </param>

/// <param name="\_edgeIndexJ">

/// j-индекс ребра, на котором находится

/// эпсилон-окрестность.

/// </param>

/// <param name="\_edgeWeight">

/// Номинальный вес ребра.

/// </param>

/// <param name="\_startPosition">

/// Номинальная длина, на которую изначально

/// выступает эпсилон-окрестность.

/// </param>

/// <param name="\_epsilon">

/// Номинальная длина эпсилон-окрестности.

/// </param>

/// <param name="\_step">

/// Номинальная длина шага.

/// </param>

public EpsilonNeighbourhood(Point \_firstPoint,

Point \_middlePoint, Point \_finalPoint,

int \_edgeIndexI, int \_edgeIndexJ,

double \_edgeWeight, double \_startPosition,

double \_epsilon, double \_step)

{

//Запоминаем необходимые величины.

FirstPoint = \_firstPoint;

MiddlePoint = \_middlePoint;

FinalPoint = \_finalPoint;

EdgeIndexI = \_edgeIndexI;

EdgeIndexJ = \_edgeIndexJ;

//Высчитываем длину ребра на экране.

drawnEdgeLength = 2 \* Math.Sqrt(

(MiddlePoint.X - FirstPoint.X) \*

(MiddlePoint.X - FirstPoint.X) +

(MiddlePoint.Y - FirstPoint.Y) \*

(MiddlePoint.Y - FirstPoint.Y));

//Получаем с её помощью коэффициент.

coef = drawnEdgeLength / \_edgeWeight;

//Получаем длину эпсилон окрестности на экране.

epsilon = \_epsilon \* coef;

//Высчитываем переменные смещения по двум частям ребра.

delta1X = 2 \* (MiddlePoint.X - FirstPoint.X) \* \_step / \_edgeWeight;

delta1Y = 2 \* (MiddlePoint.Y - FirstPoint.Y) \* \_step / \_edgeWeight;

delta2X = 2 \* (FinalPoint.X - MiddlePoint.X) \* \_step / \_edgeWeight;

delta2Y = 2 \* (FinalPoint.Y - MiddlePoint.Y) \* \_step / \_edgeWeight;

//Высчитываем длину на экране начального положения

//эпсилон-окрестности.

double startPosition = \_startPosition \* coef;

//Получаем начальную точку для эпсилон-окрестности

//на экране.

Point startPoint = new Point(

2 \* ((drawnEdgeLength / 2 - startPosition) \* FirstPoint.X +

startPosition \* MiddlePoint.X) / drawnEdgeLength,

2 \* ((drawnEdgeLength / 2 - startPosition) \* FirstPoint.Y +

startPosition \* MiddlePoint.Y) / drawnEdgeLength);

//Иницилизируем передвигающуюся по экрану

//эпсилон-окрестность.

Neighbourhood = new Polyline();

Neighbourhood.Stroke = Brushes.Black;

Neighbourhood.StrokeThickness = 5;

Neighbourhood.Points = new PointCollection(

new Point[] { FirstPoint, startPoint, startPoint });

if (startPosition == 0)

frontPosition = 0;

}

/// <summary>

/// Метод, пересчитывающий положение эпсилон-окрестности

/// при смещении на шаг.

/// </summary>

/// <returns>

/// Неотрицательное значение, если эпсилон-окрестность

/// впервые пересекла <see cref="FinalPoint"/>.

/// -1, если эпсилон-окрестность полностью пересекла

/// <see cref="FinalPoint"/>.

/// -2 в противном случае.

/// </returns>

public double MoveNeighbourhood()

{

//Создаём точки для дальнейшего перемещения.

Point nextFrontP = new Point(), nextBackP = new Point();

if (frontPosition == 0 || frontPosition == 1)

{

//Смещаем.

nextFrontP.X = Neighbourhood.Points[2].X + delta1X;

nextFrontP.Y = Neighbourhood.Points[2].Y + delta1Y;

//Если переступил через MiddlePoint.

if ((MiddlePoint.X - Neighbourhood.Points[2].X) \*

(MiddlePoint.X - nextFrontP.X) < 0)

{

frontPosition = 3;

//Отрезок, на который выступает.

double part = Math.Sqrt(

(MiddlePoint.X - nextFrontP.X) \*

(MiddlePoint.X - nextFrontP.X) +

(MiddlePoint.Y - nextFrontP.Y) \*

(MiddlePoint.Y - nextFrontP.Y));

nextFrontP.X = 2 \*

((drawnEdgeLength / 2 - part) \* MiddlePoint.X +

part \* FinalPoint.X) / drawnEdgeLength;

nextFrontP.Y = 2 \*

((drawnEdgeLength / 2 - part) \* MiddlePoint.Y +

part \* FinalPoint.Y) / drawnEdgeLength;

//Высчитываем заднюю точку.

if (drawnEdgeLength / 2 + part > epsilon)

{

backPosition = 1;

nextBackP.X = 2 \*

(((drawnEdgeLength / 2 + part - epsilon) \* MiddlePoint.X +

(epsilon - part) \* FirstPoint.X) / drawnEdgeLength);

nextBackP.Y = 2 \*

(((drawnEdgeLength / 2 + part - epsilon) \* MiddlePoint.Y +

(epsilon - part) \* FirstPoint.Y) / drawnEdgeLength);

}

else

{

backPosition = 0;

nextBackP = FirstPoint;

}

//Средняя точка в данной ветке всегда MiddlePoint.

Neighbourhood.Points[1] = MiddlePoint;

}

//Попадает в среднюю точку.

else if ((MiddlePoint.X - Neighbourhood.Points[2].X) \*

(MiddlePoint.X - nextFrontP.X) == 0)

{

frontPosition = 2;

nextFrontP = MiddlePoint;

//Средняя совпадает с передней.

Neighbourhood.Points[1] = MiddlePoint;

//Вычисляем заднюю.

if (drawnEdgeLength / 2 <= epsilon)

nextBackP = FirstPoint;

else

{

nextBackP.X = 2 \*

((drawnEdgeLength / 2 - epsilon) \* MiddlePoint.X +

(epsilon \* FirstPoint.X) / drawnEdgeLength);

nextBackP.Y = 2 \*

((drawnEdgeLength / 2 - epsilon) \* MiddlePoint.Y +

(epsilon \* FirstPoint.Y) / drawnEdgeLength);

}

}

//Если первая точка не переступила.

else

{

frontPosition = 1;

double helper = Math.Sqrt(

(nextFrontP.X - FirstPoint.X) \*

(nextFrontP.X - FirstPoint.X) +

(nextFrontP.Y - FirstPoint.Y) \*

(nextFrontP.Y - FirstPoint.Y));

//Вычисляем заднюю точку.

if (helper >

epsilon)

{

backPosition = 1;

nextBackP.X = (epsilon \* FirstPoint.X +

(helper - epsilon) \* nextFrontP.X) / helper;

nextBackP.Y = (epsilon \* FirstPoint.Y +

(helper - epsilon) \* nextFrontP.Y) / helper;

}

else

{

backPosition = 0;

nextBackP = FirstPoint;

}

//Средняя равна передней.

Neighbourhood.Points[1] = nextFrontP;

}

Neighbourhood.Points[0] = nextBackP;

Neighbourhood.Points[2] = nextFrontP;

return -2;

}

else if (frontPosition == 2 || frontPosition == 3)

{

//Смещаем переднюю.

nextFrontP.X = Neighbourhood.Points[2].X + delta2X;

nextFrontP.Y = Neighbourhood.Points[2].Y + delta2Y;

//Заранее вычисляем среднюю.

if ((FinalPoint.X - Neighbourhood.Points[2].X) \*

(FinalPoint.X - nextFrontP.X) <= 0)

{

frontPosition = 4;

Neighbourhood.Points[1] = FinalPoint;

}

else

{

frontPosition = 3;

Neighbourhood.Points[1] = nextFrontP;

}

//Вычисляем расстояние до средней точки.

double helper = Math.Sqrt(

(nextFrontP.X - MiddlePoint.X) \*

(nextFrontP.X - MiddlePoint.X) +

(nextFrontP.Y - MiddlePoint.Y) \*

(nextFrontP.Y - MiddlePoint.Y));

//Если эпсилон больше, чем расстояние до первой точки.

if (helper + drawnEdgeLength / 2 <= epsilon)

{

backPosition = 0;

nextBackP = FirstPoint;

Neighbourhood.Points[1] = MiddlePoint;

}

//Если эсилон меньше, чем расстояние до средней точки.

else if (helper > epsilon)

{

backPosition = 3;

nextBackP.X = (epsilon \* MiddlePoint.X +

(helper - epsilon) \* nextFrontP.X) / helper;

nextBackP.Y = (epsilon \* MiddlePoint.Y +

(helper - epsilon) \* nextFrontP.Y) / helper;

}

//Если эпсилон равно расстоянию до средней точки.

else if (helper == epsilon)

{

backPosition = 2;

nextBackP = MiddlePoint;

}

//Задняя точка лежит на первой половине ребра.

else

{

backPosition = 1;

nextBackP.X = 2 \*

((drawnEdgeLength / 2 + helper - epsilon) \* MiddlePoint.X +

(epsilon - helper) \* FirstPoint.X) / drawnEdgeLength;

nextBackP.Y = 2 \*

((drawnEdgeLength / 2 + helper - epsilon) \* MiddlePoint.Y +

(epsilon - helper) \* FirstPoint.Y) / drawnEdgeLength;

Neighbourhood.Points[1] = MiddlePoint;

}

Neighbourhood.Points[0] = nextBackP;

if ((FinalPoint.X - Neighbourhood.Points[2].X) \*

(FinalPoint.X - nextFrontP.X) <= 0)

{

//Окрестность в первый раз заступила.

Neighbourhood.Points[2] = FinalPoint;

return Math.Sqrt(

(nextFrontP.X - FinalPoint.X) \*

(nextFrontP.X - FinalPoint.X) +

(nextFrontP.Y - FinalPoint.Y) \*

(nextFrontP.Y - FinalPoint.Y)) / coef;

}

else

{

//Не заступила.

Neighbourhood.Points[2] = nextFrontP;

return -2;

}

}

//Если передня точка уже за FinalPoint (или на ней).

else

{

if (backPosition == 1)

{

nextBackP.X = Neighbourhood.Points[0].X + delta1X;

nextBackP.Y = Neighbourhood.Points[0].Y + delta1Y;

//Переносим заднюю точку.

if ((MiddlePoint.X - Neighbourhood.Points[0].X) \*

(MiddlePoint.X - nextBackP.X) < 0)

{

backPosition = 3;

//Расстояние, на которое выступает задняя

//точка окрестности за MiddlePoint.

double part = Math.Sqrt(

(MiddlePoint.X - nextBackP.X) \*

(MiddlePoint.X - nextBackP.X) +

(MiddlePoint.Y - nextBackP.Y) \*

(MiddlePoint.Y - nextBackP.Y));

nextBackP.X = 2 \*

((drawnEdgeLength / 2 - part) \* MiddlePoint.X +

part \* FinalPoint.X) / drawnEdgeLength;

nextBackP.Y = 2 \*

((drawnEdgeLength / 2 - part) \* MiddlePoint.Y +

part \* FinalPoint.Y) / drawnEdgeLength;

Neighbourhood.Points[1] = FinalPoint;

}//Задняя точка стала MiddlePoint.

else if ((MiddlePoint.X - Neighbourhood.Points[0].X) \*

(MiddlePoint.X - nextBackP.X) == 0)

{

backPosition = 2;

nextBackP = MiddlePoint;

Neighbourhood.Points[1] = FinalPoint;

}

//Задняя точка не перешли через MiddlePoint.

else

{

backPosition = 1;

Neighbourhood.Points[1] = MiddlePoint;

}

}

//Задняя точка уже на второй половине.

else

{

nextBackP.X = Neighbourhood.Points[0].X + delta2X;

nextBackP.Y = Neighbourhood.Points[0].Y + delta2Y;

Neighbourhood.Points[1] = FinalPoint;

//Переступила через FinalPoint.

if ((FinalPoint.X - Neighbourhood.Points[0].X) \*

(FinalPoint.X - nextBackP.X) <= 0)

{

backPosition = 4;

Neighbourhood.Points[1] = FinalPoint;

return -1;

}

else

backPosition = 3;

}

Neighbourhood.Points[0] = nextBackP;

Neighbourhood.Points[2] = FinalPoint;

return -2;

}

}//public double MoveNeighbourhood()

}

}

# **2. БИБЛИОТЕКА КЛАССОВ (.NET FRAMEWORK) “GraphLib”**

## **2.1. Класс “MetricOrientedGraph” (код “MetricOrientedGraph.cs")**

using System;

using System.Linq;

using System.IO;

using System.Collections.Generic;

namespace GraphLib

{

/// <summary>

/// Статический класс, содержащий методы для работы с

/// метрическими ориентированными графами, реализованными

/// через <see cref="Pair{T, U}"/>.

/// </summary>

public static class MetricOrientedGraph

{

/// <summary>

/// Записываем граф <paramref name="graph"/> в

/// файл <paramref name="path"/>.

/// </summary>

/// <param name="path">

/// Путь к файлу для записи.

/// </param>

/// <param name="graph">

/// Граф, который необходимо записать в файл.

/// </param>

public static void SaveGraphToFile(string path,

List<Pair<int, double>>[] graph)

{

//Создаём поток для записи в файл path.

using (StreamWriter sw = new StreamWriter(path))

{

//Записываем в файл количестве вершин графа.

sw.WriteLine(graph.Length);

//Записываем в файл количество рёбер графа.

sw.WriteLine(graph.Sum(list =>

{

return list.Count;

}));

//Записываем в файл строки по 3 числа, разделённых

//пробелом: номер начальной вершины, номер конечной

//вершины, вес ребра.

for (int i = 0; i < graph.Length; i++)

graph[i].ForEach(edge =>

{

sw.WriteLine((i + 1) + " " +

edge.First + " " + edge.Second);

});

}

}

/// <summary>

/// Считывает граф из файла <paramref name="path"/>.

/// </summary>

/// <param name="path">

/// Путь к файлу для чтения.

/// </param>

/// <returns>

/// Считанный из файла <paramref name="path"/> граф.

/// </returns>

/// <exception cref="FormatException">

/// Если данные в файле <paramref name="path"/>

/// имеют некорректный формат.

/// </exception>

public static List<Pair<int, double>>[] ReadGraphFromFile(string path)

{

//Создаём поток для чтения из файла path.

using (StreamReader sr = new StreamReader(path))

{

//Переменная, в которую будет считываться

//каждая новая строка файла.

string readedString;

//Граф, который необходимо вернуть.

List<Pair<int, double>>[] graph;

//1-я строка - количество вершин в графе

//(целое число, большее 0).

int amountOfVertices;

if ((readedString = sr.ReadLine()) is null)

throw new FormatException(

"Количество строк в файле меньше, " +

"чем требуемое. ");

if (!int.TryParse(readedString, out amountOfVertices))

throw new FormatException(

"Строка 1: количество вершин в графе должно " +

"преобразовываться к целому числу. ");

if (amountOfVertices <= 0)

throw new FormatException(

"Строка 1: количество вершин в графе должно " +

"быть положительным. ");

//Создаём граф из amountOfVertices вершин.

graph = new List<Pair<int, double>>[amountOfVertices];

//Инициализируем каждый список рёбер.

for (int i = 0; i < amountOfVertices; i++)

graph[i] = new List<Pair<int, double>>();

//2-я строка - количество рёбер в графе

//(целое число, большее 0).

int amountOfEdges;

if ((readedString = sr.ReadLine()) is null)

throw new FormatException(

"Количество строк в файле меньше, " +

"чем требуемое. ");

if (!int.TryParse(readedString, out amountOfEdges))

throw new FormatException(

"Строка 2: количество рёбер в графе должно " +

"преобразовываться к целому числу. ");

if (amountOfEdges <= 0)

throw new FormatException(

"Строка 2: количество рёбер в графе должно " +

"быть положительным. ");

/\* Далее, в файле должны присутствовать amOfEdges строк,

\* в каждой из которых должно находиться по 3 числа,

\* разделённых произвольным количеством пробелов.

\* 1-е число - номер начальной вершины (целое число

\* в диапазоне [1; amOfVertices]).

\* 2-е число - номер конечной вершины (целое число

\* в диапазоне [1; amOfVertices]).

\* 3-е число - вес ребра (действительное число

\* в диапазоне [1; 100]).

\* Вес может задаваться в виде √[число].

\*/

for (int i = 0; i < amountOfEdges; i++)

{

if ((readedString = sr.ReadLine()) is null)

throw new FormatException(

"Количество строк в файле меньше, " +

"чем требуемое. ");

//Разделяем очередную строку по пробелам.

string[] newEdge = readedString.Split(new char[] { ' ' },

StringSplitOptions.RemoveEmptyEntries);

//Начальная и конечная вершины соответственно.

int head, tail;

//Вес ребра.

double edgeWeight;

//Из разделения очередной строки из файла

//должны были получиться 3 элемента.

if (newEdge.Length != 3)

throw new FormatException(

$"Строка {i + 3}: после разделения строки " +

"по пробелам должно получаться 3 " +

"элемента. ");

//Преобразуем номер начальной вершины.

if (!int.TryParse(newEdge[0], out head))

throw new FormatException(

$"Строка {i + 3}: номер начальной вершины " +

$"должен преобразовываться к целому числу. ");

//Проверяем диапазон номера начальной вершины.

if (head < 1 || head > amountOfVertices)

throw new FormatException(

$"Строка {i + 3}: номер начальной вершины " +

$"должены быть больше либо равен 1 и " +

$"меньше либо равен количеству вершин в графе. ");

//Преобразуем номер конечной вершины.

if (!int.TryParse(newEdge[1], out tail))

throw new FormatException(

$"Строка {i + 3}: номер конечной вершины " +

$"должен преобразовываться к целому числу. ");

//Проверяем диапазон номера конечной вершины.

if (tail < 1 || tail > amountOfVertices)

throw new FormatException(

$"Строка {i + 3}: номер конечной вершины " +

$"должен быть больше либо равен 1 и " +

$"меньше либо равен количеству вершин в графе. ");

//Преобразуем вес ребра.

if (!double.TryParse(newEdge[2], out edgeWeight))

throw new FormatException(

$"Строка {i + 3}: вес ребра должен " +

$"преобразовываться к действительному числу. ");

//Проверяем диапазон веса ребра.

if (edgeWeight < 1 || edgeWeight > 100)

throw new FormatException(

$"Строка {i + 3}: вес ребра должен быть " +

$"больше либо равен 1 и меньше либо равен 100. ");

//Добавляем очередное ребро в граф.

AddEdgeToGraph(graph, head,

new Pair<int, double>(tail, edgeWeight));

}

//Возвращаем полученный из файла path граф.

return graph;

}

}

/// <summary>

/// Добавляет ребро <paramref name="edge"/>,

/// начинающееся в вершине с номером <paramref name="vertex"/>,

/// в граф <paramref name="graph"/>.

/// </summary>

/// <param name="graph">

/// Граф, в который добавляется вершина.

/// </param>

/// <param name="vertex">

/// Номер начальной вершины добавляемого ребра.

/// </param>

/// <param name="edge">

/// Добавляемое ребро.

/// </param>

/// <returns>

/// Индекс в списке из элемента графа

/// <paramref name="graph"/> под номером

/// <paramref name="vertex"/> (с индексом

/// <paramref name="vertex"/> - 1).

/// </returns>

public static int AddEdgeToGraph(

List<Pair<int, double>>[] graph,

int vertex,

Pair<int, double> edge)

{

//Если список пуст, добавляем в конец

//и возвращаем индекс 0.

if (graph[vertex - 1].Count == 0)

{

graph[vertex - 1].Add(edge);

return 0;

}

/\*Далее к рёбрам применяются операторы сравнения.

\* Для более подробной информации о сравнении

\* элементов типа Pair<T, U> обратитесь к

\* исходному коду этого типа.

\*/

if (edge <= graph[vertex - 1][0])

{

graph[vertex - 1].Insert(0, edge);

return 0;

}

if (edge >= graph[vertex - 1][graph[vertex - 1].Count - 1])

{

graph[vertex - 1].Add(edge);

return graph[vertex - 1].Count - 1;

}

for (int i = 0; i < graph[vertex - 1].Count - 1; i++)

if (edge >= graph[vertex - 1][i] &&

edge <= graph[vertex - 1][i + 1])

{

graph[vertex - 1].Insert(i + 1, edge);

return i + 1;

}

//Недостижимо при корректных данных, но компилятор этого

//просчитать не может.

return -1;

}

/// <summary>

/// Вычисляет количество компонент сильной связности

/// в графе <paramref name="graph"/>.

/// </summary>

/// <param name="graph">

/// Граф, в котором необхоимо найти количество

/// компонент сильной связности.

/// </param>

/// <remarks>

/// Данный метод представляет собой

/// <see href="https://e-maxx.ru/algo/strong\_connected\_components">

/// алгоритм построения конденсации графа</see>.

/// </remarks>

/// <returns>

/// Количество компонент сильной связности в

/// графе <paramref name="graph"/>.

/// </returns>

public static int FindAmountOfStronglyConnectedComponents(

List<Pair<int, double>>[] graph)

{

//Массив посещённых вершин.

bool[] visited = new bool[graph.Length];

//Список индексов вершин, расположенных по

//порядку топологической сортировки (1-й

//шаг алгоритма), т.е. по порядку выхода.

List<int> order = new List<int>();

//Выполняем топологическую сортировку.

for (int i = 0; i < graph.Length; i++)

if (!visited[i])

DFS(graph, i + 1,

visited, order);

//Освежаем массив посещённых вершин.

for (int i = 0; i < graph.Length; i++)

visited[i] = false;

//Получаем транспонированный граф для 2-го

//шага алгоритма.

List<Pair<int, double>>[] transposeGraph =

GetTransposeGraph(graph);

//Переменная для количества компонент сильной связности.

int amountOfComponents = 0;

for (int i = 0; i < transposeGraph.Length; i++)

//Проходимся по вершинам транспонированного

//графа в порядке топологической сортировки.

if (!visited[

order[transposeGraph.Length - i - 1] - 1])

{

DFS(transposeGraph,

order[graph.Length - i - 1], visited);

amountOfComponents++;

}

//Возвращаем количество компонент сильной связности.

return amountOfComponents;

}

/// <summary>

/// Алгоритм <see href="https://e-maxx.ru/algo/dfs"> поиска в глубину </see>

/// по графу <paramref name="graph"/> из вершины с номером <paramref name="vertex"/>,

/// определяющий, какие из вершин графа были посещены (через <paramref name="visited"/>).

/// </summary>

/// <param name="graph">

/// Граф, в котором производится поиск в глубину.

/// </param>

/// <param name="vertex">

/// Номер начальной вершины для поиска.

/// </param>

/// <param name="visited">

/// Массив посещённых вершин (должен быть предварительно проинициализирован).

/// </param>

public static void DFS(

List<Pair<int, double>>[] graph,

int vertex,

bool[] visited)

{

visited[vertex - 1] = true;

for (int i = 0; i < graph[vertex - 1].Count; i++)

if (!visited[graph[vertex - 1][i].First - 1])

DFS(graph, graph[vertex - 1][i].First,

visited);

}

/// <summary>

/// Алгоритм <see href="https://e-maxx.ru/algo/dfs"> поиска в глубину </see>

/// по графу <paramref name="graph"/> из вершины с номером <paramref name="vertex"/>,

/// определяющий, какие из вершин графа были посещены (через <paramref name="visited"/>)

/// и в каком порядке (через <paramref name="order"/>).

/// </summary>

/// <param name="graph">

/// Граф, в котором производится поиск в глубину.

/// </param>

/// <param name="vertex">

/// Номер начальной вершины для поиска.</param>

/// <param name="visited">

/// Массив посещённых вершин (должен быть предварительно проинициализирован).

/// </param>

/// <param name="order">

/// Список, определяющий порядок вершин (должен быть предварительно проинициализирован).

/// </param>

public static void DFS(

List<Pair<int, double>>[] graph,

int vertex,

bool[] visited,

List<int> order)

{

visited[vertex - 1] = true;

for (int i = 0; i < graph[vertex - 1].Count; i++)

if (!visited[graph[vertex - 1][i].First - 1])

DFS(graph, graph[vertex - 1][i].First,

visited, order);

order.Add(vertex);

}

/// <summary>

/// Создаёт граф, транспонированный (обратный) переданному

/// <paramref name="graph"/>.

/// </summary>

/// <param name="graph">

/// Граф, который необходимо транспонировать (обратить).

/// </param>

/// <returns>

/// Транспонированный (обратный) для <paramref name="graph"/>

/// граф.

/// </returns>

public static List<Pair<int, double>>[] GetTransposeGraph(

List<Pair<int, double>>[] graph)

{

//Инициализируем обратный граф на то же количество вершин.

List<Pair<int, double>>[] transposeGraph =

new List<Pair<int, double>>[graph.Length];

//Инициализируем списки рёбер для каждой верщины.

for (int i = 0; i < graph.Length; i++)

transposeGraph[i] = new List<Pair<int, double>>();

//Добавляем обратные рёбра в новый массив.

for (int i = 0; i < graph.Length; i++)

for (int j = 0; j < graph[i].Count; j++)

AddEdgeToGraph(transposeGraph,

graph[i][j].First,

new Pair<int, double>(

i + 1,

graph[i][j].Second));

//Возвращаем транспонированный массив.

return transposeGraph;

}

}//public static class MetricOrientedGraph

}//namespace GraphLib

## **2.2. Класс “Pair” (код “Pair.cs”)**

using System;

namespace GraphLib

{

/// <summary>

/// Класс, представляющий пару из 2-х

/// элементов - первый типа <typeparamref name="T"/> и

/// второй типа <typeparamref name="U"/>.

/// </summary>

/// <typeparam name="T">

/// Тип для первого элемента в паре.

/// </typeparam>

/// <typeparam name="U">

/// Тип дл второго элемента в паре.

/// </typeparam>

public class Pair<T, U> : IComparable<Pair<T, U>>

where T : IComparable<T>, new()

where U : IComparable<U>, new()

{

#region Элементы пары

/// <summary>

/// Поле для свойства <see cref="First"/>.

/// </summary>

T first;

/// <summary>

/// Первый элемент в паре.

/// </summary>

/// <remarks>

/// Поле данного свойства - <see cref="first"/>.

/// </remarks>

/// <exception cref="ArgumentNullException">

/// Если тип <see cref="T"/> является ссылочным и

/// поданное значение является null.

/// </exception>

public T First

{

get { return first; }

set

{

if (typeof(T).IsByRef && value == null)

throw new ArgumentNullException(

typeof(T) + " is type by reference, " +

"so first element in pair must not " +

"be null. ");

first = value;

}

}

/// <summary>

/// Поле для свойства <see cref="Second"/>.

/// </summary>

U second;

/// <summary>

/// Второй элемент в паре.

/// </summary>

/// <remarks>

/// Поле данного свойства - <see cref="second"/>.

/// </remarks>

/// <exception cref="ArgumentNullException">

/// Если тип <see cref="U"/> является ссылочным и

/// поданное значение является null.

/// </exception>

public U Second

{

get { return second; }

set

{

if (typeof(U).IsByRef && value == null)

throw new ArgumentNullException(

typeof(U) + " is type by reference, " +

"so second element in pair must not " +

"be null. ");

second = value;

}

}

#endregion

#region Конструкторы

/// <summary>

/// Присваивает элементам пары значения по умолчанию

/// (через конструкторы по умолчанию данных типов).

/// </summary>

public Pair()

{

First = new T();

Second = new U();

}

/// <summary>

/// Присвивает элементам пары значения, переданные

/// через параметры.

/// </summary>

/// <param name="\_first">

/// Значение для первого элемента пары.

/// </param>

/// <param name="\_second">

/// Значение для второго элемента пары.

/// </param>

public Pair(T \_first, U \_second)

{

First = \_first;

Second = \_second;

}

#endregion

#region Переопределённые базовые методы

/// <summary>

/// Определяет, равен ли заданный объект текущему объекту.

/// </summary>

/// <param name="obj">

/// Объект, который требуется сравнить с текущим объектом.

/// </param>

/// <returns>

/// Значение true, если указанный объект равен текущему объекту;

/// в противном случае — значение false.

/// </returns>

public override bool Equals(object obj)

{

if (obj is null) return false;

return Equals(obj as Pair<T, U>);

}

/// <summary>

/// Определяет, равен ли заданный объект типа <see cref="Pair{T, U}"/>

/// текущему.

/// </summary>

/// <param name="other">

/// Объект типа <see cref="Pair{T, U}"/>, который требуется

/// сравнить с текущим объектом.

/// </param>

/// <returns>

/// Значение true, если указанный объект типа <see cref="Pair{T, U}"/>

/// равен текущему объекту; в противном случае — значение false.

/// </returns>

public bool Equals(Pair<T, U> other)

{

if (other is null) return false;

return (this == other);

}

/// <summary>

/// Служит хэш-функцией для объектов типа <see cref="Pair{T, U}"/>.

/// </summary>

/// <returns>

/// Хэш-код текущего объекта типа <see cref="Pair{T, U}"/>.

/// </returns>

public override int GetHashCode()

{

return Tuple.Create(First, Second).GetHashCode();

}

/// <summary>

/// Возвращает строку, представляющую текущий объект типа

/// <see cref="Pair{T, U}"/>.

/// </summary>

/// <returns>

/// Строку, представляющую текущий объект типа <see cref="Pair{T, U}"/>.

/// </returns>

public override string ToString()

{

return $"Type of first element: {typeof(T)}; " +

$"first element: {First}; type of second " +

$"element: {typeof(U)}; second element: " +

$"{Second}. ";

}

#endregion

#region Реализация интерфейса IComparable<Pair<T, U>>

/// <summary>

/// Сравнивает заданный объект типа <see cref="Pair{T, U}"/>

/// с текущим и возвращает целое число, отражающее положение

/// текущего объекта относительно других в сортировке.

/// </summary>

/// <param name="other">

/// Заданный объект для сравнения с текущим.

/// </param>

/// <returns>

/// 1 - если текущий объект стоит выше в сортировке;

/// (-1) - если текущий объект стоит ниже в сортировке;

/// 0 - если положение объектов в сортировке равнозначно.

/// </returns>

public int CompareTo(Pair<T, U> other)

{

int result;

if ((result = First.CompareTo(other.First)) == 0)

return Second.CompareTo(other.Second);

return result;

}

#endregion

#region Операторы сравнения

public static bool operator >(Pair<T, U> p1, Pair<T, U> p2)

{

return (p1.CompareTo(p2) > 0);

}

public static bool operator <(Pair<T, U> p1, Pair<T, U> p2)

{

return (p1.CompareTo(p2) < 0);

}

public static bool operator >=(Pair<T, U> p1, Pair<T, U> p2)

{

return (p1.CompareTo(p2) >= 0);

}

public static bool operator <=(Pair<T, U> p1, Pair<T, U> p2)

{

return (p1.CompareTo(p2) <= 0);

}

public static bool operator ==(Pair<T, U> p1, Pair<T, U> p2)

{

return (p1.CompareTo(p2) == 0);

}

public static bool operator !=(Pair<T, U> p1, Pair<T, U> p2)

{

return (p1.CompareTo(p2) != 0);

}

#endregion

}

}

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Лист регистрации изменений | | | | | | | | | |
| Изм. | Номера листов | | | | Всего листов (страниц) в документе | № документа | Входя- щий № сопро- водит. докум. и дата | Под- пись | Дата |
| изменен- ных | заменен- ных | новых | аннулиро- ванных |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |