|  |  |
| --- | --- |
| 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»**

|  |  |
| --- | --- |
| СОГЛАСОВАНО  Доцент департамента 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. |

Факультет компьютерных наук  
Департамент программной инженерии

**ПРОГРАММА ДЛЯ МОДЕЛИРОВАНИЯ ДВИЖЕНИЯ ТОЧЕК НА**

**ОРИЕНТИРОВАННОМ МЕТРИЧЕСКОМ ГРАФЕ, С УСЛОВИЕМ**

**СИНХРОНИЗАЦИИ В ВЕРШИНАХ**

**Текст программы**

**ЛИСТ УТВЕРЖДЕНИЯ**

**RU.17701729.04.13-01 12 01-1-ЛУ**

Исполнитель

студент группы БПИ196

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / А. А. Баранова /

«\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2020 г.

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

УТВЕРЖДЕНRU.17701729.04.13-01 12 01-1-ЛУ

**ПРОГРАММА ДЛЯ МОДЕЛИРОВАНИЯ ДВИЖЕНИЯ ТОЧЕК НА**

**ОРИЕНТИРОВАННОМ МЕТРИЧЕСКОМ ГРАФЕ, С УСЛОВИЕМ**

**СИНХРОНИЗАЦИИ В ВЕРШИНАХ**

**Текст программы**

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

**Листов 199**

**СОДЕРЖАНИЕ**

[1. ТЕКСТ ПРОГРАММЫ 4](#_Toc40927768)

[1.1. DotsMovementModelingApp 4](#_Toc40927769)

[1.1.1. RandomDigraphGeneratorForm.cs 4](#_Toc40927770)

[1.1.2. SquareLatticeForm.cs 6](#_Toc40927771)

[1.1.3. TriangularLatticeForm.cs 10](#_Toc40927772)

[1.1.4. MainWindow класс 14](#_Toc40927773)

[1.1.4.1. Drawing.cs 14](#_Toc40927774)

[1.1.4.2. DrawingItemsSidebar.cs 25](#_Toc40927775)

[1.1.4.3. GridBehavior.cs 33](#_Toc40927776)

[1.1.4.4. Main.cs 44](#_Toc40927777)

[1.1.4.5. MainMenu.cs 53](#_Toc40927778)

[1.1.4.6. ModelingParameters.cs 59](#_Toc40927779)

[1.1.4.7. MovementModeling.cs 64](#_Toc40927780)

[1.1.4.8. TopMenu.cs 72](#_Toc40927781)

[1.1.4.9. Variables.cs 76](#_Toc40927782)

[1.2. DotsMovementModelingAppLib 80](#_Toc40927783)

[1.2.1. DotsMovementModelingAppLib.Commands 80](#_Toc40927784)

[1.2.1.1. AddArcCommand.cs 80](#_Toc40927785)

[1.2.1.2. AddVertexCommand.cs 81](#_Toc40927786)

[1.2.1.3. ChangeArcLengthCommand.cs 83](#_Toc40927787)

[1.2.1.4. ChangeColorCommand.cs 86](#_Toc40927788)

[1.2.1.5. ChangeRadiusCommand.cs 88](#_Toc40927789)

[1.2.1.6. CommandsManager.cs 90](#_Toc40927790)

[1.2.1.7. EraseArcCommand.cs 93](#_Toc40927791)

[1.2.1.8. EraseVertexCommand.cs 94](#_Toc40927792)

[1.2.1.9. ICommand.cs 97](#_Toc40927793)

[1.2.1.10. MoveDigraphCommand.cs 97](#_Toc40927794)

[1.2.1.11. MoveVertexCommand.cs 99](#_Toc40927795)

[1.2.1.12. ResizeDigraphCommand.cs 101](#_Toc40927796)

[1.2.2. DotsMovementModelingAppLib.Graph 103](#_Toc40927797)

[1.2.2.1. Arc.cs 103](#_Toc40927798)

[1.2.2.2. Dihraph.cs 106](#_Toc40927799)

[1.2.2.3. Vertex.cs 114](#_Toc40927800)

[1.2.3. DotsMovementModelingAppLib.Modeling 115](#_Toc40927801)

[1.2.3.1. ChartWindow.cs 115](#_Toc40927802)

[1.2.3.2. MovementModeling класс 118](#_Toc40927803)

[1.2.3.2.1. MovementModeling.cs 119](#_Toc40927804)

[1.2.3.2.2. TicksAnimation.cs 131](#_Toc40927805)

[1.2.3.2.3. TicksChartGif.cs 138](#_Toc40927806)

[1.2.4. AppAdditions.cs 141](#_Toc40927807)

[1.2.5. ConnectivityCheck.cs 146](#_Toc40927808)

[1.2.6. GraphDrawing.cs 150](#_Toc40927809)

[1.2.7. MathParser.cs 168](#_Toc40927810)

**ТЕКСТ ПРОГРАММЫ**

* 1. **DotsMovementModelingApp**
     1. **RandomDigraphGeneratorForm.cs**

using DotsMovementModelingAppLib;

using System;

using System.Linq;

using System.Windows.Forms;

namespace DotsMovementModelingApp

{

public partial class RandomDigraphGeneratorForm : Form

{

private static readonly Random Rnd = new Random(); //Random values generator

/// <summary>

/// Generated random digraph

/// </summary>

public Digraph Digraph { get; private set; }

private readonly int width; //Drawing surface width (maximum width)

private readonly int height; //Drawing surface height (maximum height)

/// <summary>

/// Initializes a new instance of RandomDigraphGeneratorForm

/// </summary>

/// <param name="width">Drawing surface width (maximum width)</param>

/// <param name="height">Drawing surface height (maximum height)</param>

/// <exception cref="ArgumentOutOfRangeException"/>

public RandomDigraphGeneratorForm(int width, int height)

{

if (width <= 0)

throw new ArgumentOutOfRangeException(nameof(width));

if (height <= 0)

throw new ArgumentOutOfRangeException(nameof(height));

InitializeComponent();

Digraph = null;

this.width = width;

this.height = height;

}

private void VNRandom\_Click(object sender, EventArgs e) =>

NumOfVertices.Value = Rnd.Next(3, 21);

/// <summary>

/// Generates a random digraph

/// </summary>

private void Button2\_Click(object sender, EventArgs e)

{

Digraph = new Digraph();

bool[] visitedV = new bool[(int)NumOfVertices.Value];

for (int i = 0; i < (int)NumOfVertices.Value; i++)

{

int th = Rnd.Next(1, 5);

int p = Rnd.Next(1, 10001);

int s = Rnd.Next(0, 2 \* th);

Digraph.AddVertex(new Vertex(Rnd.Next(10, width - 10), Rnd.Next(10, height - 10)), th, p, s);

visitedV[i] = i == 0;

}

int start = 0;

while (visitedV.Contains(false))

{

int end;

do

{

} while ((end = Rnd.Next(1, visitedV.Length)) == start

|| !Digraph.Arcs.TrueForAll(arc => arc.StartVertex != start || arc.EndVertex != end));

Digraph.AddArc(new Arc(start, end, Rnd.Next(3, 11) + Rnd.NextDouble()));

visitedV[end] = true;

start = end;

}

Digraph.AddArc(new Arc(start, 0, Rnd.Next(3, 11) + Rnd.NextDouble()));

Close();

}

/// <summary>

/// Closes the form

/// </summary>

private void Cancel\_Click(object sender, EventArgs e) => Close();

}

}

* + 1. **SquareLatticeForm.cs**

using DotsMovementModelingAppLib;

using System;

using System.Drawing;

using System.Windows.Forms;

namespace DotsMovementModelingApp

{

public partial class SquareLatticeForm : Form

{

private Random rnd; //Random values generator

/// <summary>

/// Generated square lattice digraph

/// </summary>

public Digraph SquareLatticeDigraph { get; private set; }

private readonly int width; //Drawing surface width (maximum width)

private readonly int height; //Drawing surface height (maximum height)

/// <summary>

/// Initializes a new instance of SquareLatticeForm

/// </summary>

/// <param name="width">Drawing surface width (maximum width)</param>

/// <param name="height">Drawing surface height (maximum height)</param>

/// <exception cref="ArgumentOutOfRangeException"/>

public SquareLatticeForm(int width, int height)

{

if (width <= 0)

throw new ArgumentOutOfRangeException(nameof(width));

if (height <= 0)

throw new ArgumentOutOfRangeException(nameof(height));

InitializeComponent();

SquareLatticeDigraph = null;

this.width = width;

this.height = height;

}

/// <summary>

/// Generates square lattice digraph with chosen parameters

/// and closes the form

/// </summary>

private void OK\_Click(object sender, EventArgs e)

{

SquareLatticeDigraph = new Digraph();

AddVertices();

AddArcs();

Close();

}

/// <summary>

/// Adds vertices to the digraph

/// </summary>

private void AddVertices()

{

//Distance between adjacent vertices

int step = (Math.Min(width, height) - 100) / Math.Max((int)Xvalue.Value - 1, (int)Yvalue.Value - 1);

//Current vertex coordinates

Point p = new Point((width - 100 - step \* ((int)Yvalue.Value - 1)) / 2 + 50,

(height - 100 - step \* ((int)Xvalue.Value - 1)) / 2 + 50);

for (int i = 0; i < Xvalue.Value; i++, p.Y += step, p.X = (width - 100 - step \* ((int)Yvalue.Value - 1)) / 2 + 50)

for (int j = 0; j < Yvalue.Value; j++, p.X += step)

{

if (rnd != null)

{

int th = rnd.Next(1, 5);

int rp = rnd.Next(1, 10001);

int s = rnd.Next(0, 2 \* th);

SquareLatticeDigraph.AddVertex(new Vertex(p.X, p.Y), th, rp, s);

continue;

}

SquareLatticeDigraph.AddVertex(new Vertex(p.X, p.Y));

}

}

/// <summary>

/// Adds arcs to the digraph

/// </summary>

private void AddArcs()

{

for (int i = 0; i < Xvalue.Value; i++)

for (int j = 0; j < Yvalue.Value; j++)

{

if (j != Yvalue.Value - 1)

{

SquareLatticeDigraph.AddArc(new Arc(i \* (int)Yvalue.Value + j, i \* (int)Yvalue.Value + 1 + j,

rnd != null ? rnd.Next(1, 5) + rnd.NextDouble() : 1));

SquareLatticeDigraph.AddArc(new Arc(i \* (int)Yvalue.Value + 1 + j, i \* (int)Yvalue.Value + j,

rnd != null ? rnd.Next(1, 5) + rnd.NextDouble() : 1));

}

if (i != Xvalue.Value - 1)

{

SquareLatticeDigraph.AddArc(new Arc(i \* (int)Yvalue.Value + j, i \* (int)Yvalue.Value + j % (int)Yvalue.Value + (int)Yvalue.Value,

rnd != null ? rnd.Next(1, 5) + rnd.NextDouble() : 1));

SquareLatticeDigraph.AddArc(new Arc(i \* (int)Yvalue.Value + j % (int)Yvalue.Value + (int)Yvalue.Value, i \* (int)Yvalue.Value + j,

rnd != null ? rnd.Next(1, 5) + rnd.NextDouble() : 1));

}

}

}

/// <summary>

/// Changes the random values generator value

/// to allow or forbid random digraph parameters filling

/// </summary>

private void ParamsCheckBox\_CheckedChanged(object sender, EventArgs e) =>

rnd = ParamsCheckBox.Checked ? new Random() : null;

/// <summary>

/// Closes the form

/// </summary>

private void Cancel\_Click(object sender, EventArgs e) => Close();

}

}

* + 1. **TriangularLatticeForm.cs**

using DotsMovementModelingAppLib;

using System;

using System.Drawing;

using System.Windows.Forms;

namespace DotsMovementModelingApp

{

public partial class TriangularLatticeForm : Form

{

private Random rnd; // Random values generator

/// <summary>

/// Generated square lattice digraph

/// </summary>

public Digraph TriangularLatticeDigraph { get; private set; }

private readonly int width; //Drawing surface width (maximum width)

private readonly int height; //Drawing surface height (maximum height)

/// <summary>

/// Initializes a new instance of TriangularLatticeForm

/// </summary>

/// <param name="width">Drawing surface width (maximum width)</param>

/// <param name="height">Drawing surface height (maximum height)</param>

/// <exception cref="ArgumentOutOfRangeException"/>

public TriangularLatticeForm(int width, int height)

{

if (width <= 0)

throw new ArgumentOutOfRangeException(nameof(width));

if (height <= 0)

throw new ArgumentOutOfRangeException(nameof(height));

InitializeComponent();

TriangularLatticeDigraph = null;

this.width = width;

this.height = height;

}

private void OK\_Click(object sender, EventArgs e)

{

TriangularLatticeDigraph = new Digraph();

AddVertices();

AddArcs();

Close();

}

/// <summary>

/// Adds vertices to the digraph

/// </summary>

private void AddVertices()

{

//Distance between adjacent vertices

int step = (int)((Math.Min(width, height)) \* 1.0 / Math.Max((int)Xvalue.Value - 0.5, (int)Yvalue.Value - 1));

//Current vertex coordinates

var x = (int)((width - step \* ((int)Xvalue.Value - 0.5)) / 2.0 + step / 2);

var p = new Point(x, (height - 100 - step \* ((int)Yvalue.Value - 1)) / 2 + 75);

for (int i = 0; i < Yvalue.Value; i++, p.Y += (int)(step \* 0.866),

p.X = i % 2 == 0 ? x : x - step / 2)

for (int j = 0; j < Xvalue.Value; j++, p.X += step)

{

if (rnd != null)

{

int th = rnd.Next(1, 5);

int rp = rnd.Next(1, 10001);

int s = rnd.Next(0, 2 \* th);

TriangularLatticeDigraph.AddVertex(new Vertex(p.X, p.Y), th, rp, s);

continue;

}

TriangularLatticeDigraph.AddVertex(new Vertex(p.X, p.Y));

}

}

/// <summary>

/// Adds arcs to the digraph

/// </summary>

private void AddArcs()

{

for (int i = 0; i < Yvalue.Value; i++)

for (int j = 0; j < Xvalue.Value; j++)

{

if (j != Xvalue.Value - 1)

{

TriangularLatticeDigraph.AddArc(new Arc(i \* (int)Xvalue.Value + j, i \* (int)Xvalue.Value + 1 + j,

rnd != null ? rnd.Next(1, 5) + rnd.NextDouble() : 1));

TriangularLatticeDigraph.AddArc(new Arc(i \* (int)Xvalue.Value + 1 + j, i \* (int)Xvalue.Value + j,

rnd != null ? rnd.Next(1, 5) + rnd.NextDouble() : 1));

}

if (i != Yvalue.Value - 1)

{

TriangularLatticeDigraph.AddArc(new Arc(i \* (int)Xvalue.Value + j, i \* (int)Xvalue.Value + j % (int)Xvalue.Value + (int)Xvalue.Value,

rnd != null ? rnd.Next(1, 5) + rnd.NextDouble() : 1));

TriangularLatticeDigraph.AddArc(new Arc(i \* (int)Xvalue.Value + j % (int)Xvalue.Value + (int)Xvalue.Value, i \* (int)Xvalue.Value + j,

rnd != null ? rnd.Next(1, 5) + rnd.NextDouble() : 1));

}

if (i != Yvalue.Value - 1 && ((j != 0 && i % 2 != 0) || (j != Xvalue.Value - 1 && i % 2 == 0)))

{

TriangularLatticeDigraph.AddArc(new Arc(i \* (int)Xvalue.Value + j, i \* (int)Xvalue.Value + j % (int)Xvalue.Value + (int)Xvalue.Value + (int)Math.Pow(-1, i),

rnd != null ? rnd.Next(1, 5) + rnd.NextDouble() : 1));

TriangularLatticeDigraph.AddArc(new Arc(i \* (int)Xvalue.Value + j % (int)Xvalue.Value + (int)Xvalue.Value + (int)Math.Pow(-1, i), i \* (int)Xvalue.Value + j,

rnd != null ? rnd.Next(1, 5) + rnd.NextDouble() : 1));

}

}

}

/// <summary>

/// Changes the random values generator value

/// to allow or forbid random digraph parameters filling

/// </summary>

private void ParamsCheckBox\_CheckedChanged(object sender, EventArgs e) =>

rnd = ParamsCheckBox.Checked ? new Random() : null;

/// <summary>

/// Closes the form

/// </summary>

private void Cancel\_Click(object sender, EventArgs e) => Close();

}

}

* + 1. **MainWindow класс**
       1. **Drawing.cs**

using DotsMovementModelingAppLib;

using DotsMovementModelingAppLib.Commands;

using System;

using System.Drawing;

using System.Threading.Tasks;

using System.Windows.Forms;

using System.Windows.Media.Imaging;

namespace DotsMovementModelingApp

{

public partial class MainWindow

{

/// <summary>

/// Draws vertices and arcs by mouse click on a drawing surface

/// or selects a vertex to add sand to during sandpile modeling

/// </summary>

private void DrawingSurface\_MouseClick(object sender, MouseEventArgs e)

{

if (SandpilePanel.Visible)

{

if (SandpilePanel.Size.Height < 50)

SelectStock(e.X, e.Y);

else SelectVertexToAddSand(e.X, e.Y);

return;

}

if (isOnMovement) return;

if (!VertexButton.Enabled)

{

if (digraph.Vertices.Count >= 200)

{

MessageBox.Show(@"Too many vertices. Unable to add a new one.",

@"Failed", MessageBoxButtons.OK, MessageBoxIcon.Error);

return;

}

var command = new AddVertexCommand(digraph, new Vertex(e.X, e.Y));

commandsManager.Execute(command);

return;

}

if (!EdgeButton.Enabled)

{

if (FindArcVertices(e.X, e.Y))

{

var command = new AddArcCommand(digraph, new Arc(vStart, vEnd));

commandsManager.Execute(command);

}

}

}

/// <summary>

/// Deletes vertices and arcs by mouse double click on a drawing surface

/// </summary>

private void DrawingSurface\_MouseDoubleClick(object sender, MouseEventArgs e)

{

if (DeleteButton.Enabled || isOnMovement) return;

if (DigraphComponentsRemover.TryToDeleteVertexAt(e.X, e.Y, digraph, graphDrawing.R, out int i))

{

var command = new EraseVertexCommand(digraph, digraph.Vertices[i]);

commandsManager.Execute(command);

graphDrawing.DrawTheWholeGraph(digraph);

DrawingSurface.Image = graphDrawing.Image;

}

else if (DigraphComponentsRemover.TryToDeleteArcAt(e.X, e.Y, digraph, out Arc arc))

{

var command = new EraseArcCommand(digraph, arc);

commandsManager.Execute(command);

graphDrawing.DrawTheWholeGraph(digraph);

DrawingSurface.Image = graphDrawing.Image;

}

}

/// <summary>

/// Remembers vertex chosen for moving and point where the movement began

/// </summary>

private void DrawingSurface\_MouseDown(object sender, MouseEventArgs e)

{

if (VertexColorDialogOpen.Visible) VerticesColorPanel\_Leave(sender, e);

if (ArcsColorDialogOpen.Visible) ArcsColorPanel\_Leave(sender, e);

if (CursorButton.Enabled || isOnMovement) return;

isPressed = true;

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

if (Math.Pow((digraph.Vertices[i].X - e.X), 2) + Math.Pow((digraph.Vertices[i].Y - e.Y), 2) <= Math.Pow(graphDrawing.R, 2))

{

movedVertexIndex = i;

movedVertex = digraph.Vertices[i];

ticks = DateTime.Now;

return;

}

}

/// <summary>

/// Moves chosen vertex or the whole graph

/// </summary>

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

{

if (isOnMovement || !isPressed || CursorButton.Enabled || movedVertexIndex == -1) return;

digraph.Vertices[movedVertexIndex] = new Vertex(e.X, e.Y);

graphDrawing.DrawTheWholeGraph(digraph);

DrawingSurface.Image = graphDrawing.Image;

}

/// <summary>

/// Redraws graph if it's needed and

/// refreshes all the variables connected with moving

/// </summary>

private void DrawingSurface\_MouseUp(object sender, MouseEventArgs e)

{

if (isOnMovement || !isPressed || CursorButton.Enabled) return;

bool highlight = (DateTime.Now - ticks).Ticks < 2200000 &&

Math.Pow(e.X - movedVertex.X, 2) + Math.Pow(e.Y - movedVertex.Y, 2) <= graphDrawing.R \* graphDrawing.R;

if (movedVertexIndex != -1 && !highlight)

{

// Keeping the image inside the borders of the sheet

float x = e.X, y = e.Y;

if (x < graphDrawing.R + 5)

x = graphDrawing.R + 5;

if (y < graphDrawing.R + 5)

{

y = graphDrawing.R + 5;

if (GridAdjacencyMatrix[movedVertexIndex, movedVertexIndex].Value.ToString() != "0")

y += graphDrawing.R;

}

if (x > DrawingSurface.Width - graphDrawing.R - 5)

{

x = DrawingSurface.Width - graphDrawing.R - 5;

if (GridAdjacencyMatrix[movedVertexIndex, movedVertexIndex].Value.ToString() != "0")

x -= graphDrawing.R;

}

if (y > DrawingSurface.Height - graphDrawing.R - 5)

y = DrawingSurface.Height - graphDrawing.R - 5;

var command = new MoveVertexCommand(digraph, movedVertexIndex,

new Point(movedVertex.X, movedVertex.Y),

new Point((int)x, (int)y));

commandsManager.Execute(command);

}

if (highlight) digraph.Vertices[movedVertexIndex] = new Vertex(movedVertex.X, movedVertex.Y);

graphDrawing.DrawTheWholeGraph(digraph);

if (highlight) graphDrawing.HighlightVertex(digraph.Vertices[movedVertexIndex]);

DrawingSurface.Image = graphDrawing.Image;

movedVertexIndex = -1;

isPressed = false;

}

/// <summary>

/// Finds vertices chosen for creating a new arc

/// </summary>

/// <param name="x">X coordinate of a place where mouse click occurred</param>

/// <param name="y">Y coordinate of a place where mouse click occurred</param>

private bool FindArcVertices(int x, int y)

{

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

if (Math.Pow((digraph.Vertices[i].X - x), 2) + Math.Pow((digraph.Vertices[i].Y - y), 2) <= Math.Pow(graphDrawing.R, 2))

{

if (vStart == -1)

{

vStart = i;

graphDrawing.HighlightVertex(digraph.Vertices[i]);

DrawingSurface.Image = graphDrawing.Image;

return false;

}

if (vEnd == -1)

{

if (GridAdjacencyMatrix[i, vStart].Value.ToString() != "0" || vStart == i)

{

graphDrawing.UnhighlightVertex(digraph.Vertices[vStart]);

DrawingSurface.Image = graphDrawing.Image;

if (GridAdjacencyMatrix[i, vStart].Value.ToString() != "0")

MessageBox.Show($@"Unable to add {vStart + 1}-{i + 1} arc. Such an arc already exists", @"Error",

MessageBoxButtons.OK, MessageBoxIcon.Error);

if (vStart == i)

MessageBox.Show($@"Unable to add {vStart + 1}-{i + 1} arc. Arc cannot be a loop", @"Error",

MessageBoxButtons.OK, MessageBoxIcon.Error);

vStart = vEnd = -1;

return false;

}

vEnd = i;

graphDrawing.HighlightVertex(digraph.Vertices[i]);

DrawingSurface.Image = graphDrawing.Image;

return true;

}

}

return false;

}

/// <summary>

/// Searches for a vertex at (x, y) and turns it into a stock vertex

/// </summary>

private void SelectStock(int x, int y)

{

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

if (Math.Pow((digraph.Vertices[i].X - x), 2) + Math.Pow((digraph.Vertices[i].Y - y), 2) <=

Math.Pow(graphDrawing.R, 2))

{

if (digraph.Stock.Contains(i)) digraph.Stock.Remove(i);

else digraph.Stock.Add(i);

graphDrawing.DrawTheWholeGraphSandpile(digraph, false);

DrawingSurface.Image = graphDrawing.Image;

return;

}

}

/// <summary>

/// Searches for a vertex at (x, y) and adds sand to it

/// </summary>

private async void SelectVertexToAddSand(int x, int y)

{

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

if (Math.Pow((digraph.Vertices[i].X - x), 2) + Math.Pow((digraph.Vertices[i].Y - y), 2) <=

Math.Pow(graphDrawing.R, 2))

{

if (digraph.Stock.Contains(i)) return;

digraph.State[i]++;

SandpilePanel.Visible = false;

graphDrawing.HighlightVertexToAddSand(digraph.Vertices[i]);

DrawingSurface.Image = graphDrawing.Image;

await Task.Delay(200);

if (SaveGifCheckBox.Checked && movement.MovementGif.Frames.Count < 250)

{

var bmp = ((Bitmap)DrawingSurface.Image).GetHbitmap();

var src = System.Windows.Interop.Imaging.CreateBitmapSourceFromHBitmap(

bmp,

IntPtr.Zero,

System.Windows.Int32Rect.Empty, BitmapSizeOptions.FromEmptyOptions());

movement.MovementGif.Frames.Add(BitmapFrame.Create(src));

DeleteObject(bmp);

}

movement.Go();

}

}

[System.Runtime.InteropServices.DllImport("gdi32.dll")]

public static extern bool DeleteObject(IntPtr hObject);

}

}

* + - 1. **DrawingItemsSidebar.cs**

using DotsMovementModelingAppLib;

using DotsMovementModelingAppLib.Commands;

using System;

using System.Windows.Forms;

namespace DotsMovementModelingApp

{

public partial class MainWindow

{

#region Click handlers for drawing tools buttons

/// <summary>

/// Disables CursorButton and enables other tools buttons

/// </summary>

private void CursorButton\_Click(object sender, EventArgs e)

{

CursorButton.Enabled = false;

VertexButton.Enabled = true;

EdgeButton.Enabled = true;

DeleteButton.Enabled = true;

EscButton\_Click(sender, e);

}

/// <summary>

/// Disables VertexButton and enables other tools buttons

/// </summary>

private void VertexButton\_Click(object sender, EventArgs e)

{

CursorButton.Enabled = true;

VertexButton.Enabled = false;

EdgeButton.Enabled = true;

DeleteButton.Enabled = true;

EscButton\_Click(sender, e);

}

/// <summary>

/// Disables EdgeButton and enables other tools buttons

/// </summary>

private void EdgeButton\_Click(object sender, EventArgs e)

{

CursorButton.Enabled = true;

VertexButton.Enabled = true;

EdgeButton.Enabled = false;

DeleteButton.Enabled = true;

EscButton\_Click(sender, e);

}

/// <summary>

/// Disables DeleteButton and enables other tools buttons

/// </summary>

private void DeleteButton\_Click(object sender, EventArgs e)

{

CursorButton.Enabled = true;

VertexButton.Enabled = true;

EdgeButton.Enabled = true;

DeleteButton.Enabled = false;

EscButton\_Click(sender, e);

}

#endregion

/// <summary>

/// Deletes drawn digraph and refreshes all the connected controls

/// </summary>

private void ClearButton\_Click(object sender, EventArgs e)

{

if (digraph.Vertices.Count == 0) return;

const string message = "Would you like to save the graph? Otherwise, your graph will be lost";

const string caption = "Saving";

if (SaveGraph(message, caption) != DialogResult.Cancel)

RefreshVariables();

}

/// <summary>

/// Changes vertices radius

/// </summary>

private void RadiusTrackBar\_ValueChanged(object sender, EventArgs e)

{

graphDrawing.R = RadiusTrackBar.Value;

RadiusValueLabel.Text = @"R = " + RadiusTrackBar.Value;

MouseWheelTimer.Start();

radiusChanged = !radiusChanged;

}

/// <summary>

/// Enlarges digraph image by moving vertices

/// </summary>

private void EnlargeButton\_Click(object sender, EventArgs e)

{

var command = new ResizeDigraphCommand(digraph, 1.1);

commandsManager.Execute(command);

UpdateImage();

}

/// <summary>

/// Reduces digraph image by moving vertices

/// </summary>

private void ReduceButton\_Click(object sender, EventArgs e)

{

var command = new ResizeDigraphCommand(digraph, 0.9);

commandsManager.Execute(command);

UpdateImage();

}

#region Graph moving

private void Up\_Click(object sender, EventArgs e)

{

var command = new MoveDigraphCommand(digraph, 0, 10);

commandsManager.Execute(command);

UpdateImage();

}

private void Left\_Click(object sender, EventArgs e)

{

var command = new MoveDigraphCommand(digraph, -10, 0);

commandsManager.Execute(command);

UpdateImage();

}

private void Right\_Click(object sender, EventArgs e)

{

var command = new MoveDigraphCommand(digraph, 10, 0);

commandsManager.Execute(command);

UpdateImage();

}

private void Down\_Click(object sender, EventArgs e)

{

var command = new MoveDigraphCommand(digraph, -10, 0);

commandsManager.Execute(command);

UpdateImage();

}

#endregion

/// <summary>

/// Redraws the digraph

/// </summary>

private void UpdateImage()

{

if (isOnMovement && SandpileTypeCheckBox.Checked)

graphDrawing.DrawTheWholeGraphSandpile(digraph, false);

else graphDrawing.DrawTheWholeGraph(digraph);

DrawingSurface.Image = graphDrawing.Image;

}

#region Color panels

/// <summary>

/// Opens color dialog to select new vertices color

/// </summary>

private void VertexColorDialogOpen\_Click(object sender, EventArgs e)

{

if (GraphStyleColorDialog.ShowDialog() == DialogResult.Cancel) return;

var command = new ChangeColorCommand(graphDrawing, typeof(Vertex),

graphDrawing.VerticesColor, GraphStyleColorDialog.Color);

command.Executed += (s, ea) =>

{

VerticesColorPanel.BackColor = graphDrawing.VerticesColor;

graphDrawing.DrawTheWholeGraph(digraph);

DrawingSurface.Image = graphDrawing.Image;

};

commandsManager.Execute(command);

}

/// <summary>

/// Opens color dialog to select new arcs color

/// </summary>

private void ArcsColorDialogOpen\_Click(object sender, EventArgs e)

{

if (GraphStyleColorDialog.ShowDialog() == DialogResult.Cancel) return;

var command = new ChangeColorCommand(graphDrawing, typeof(Arc),

graphDrawing.ArcsColor, GraphStyleColorDialog.Color);

command.Executed += (s, ea) =>

{

ArcsColorPanel.BackColor = graphDrawing.ArcsColor;

graphDrawing.DrawTheWholeGraph(digraph);

DrawingSurface.Image = graphDrawing.Image;

};

commandsManager.Execute(command);

}

#region behavior

private void VerticesColorPanel\_Leave(object sender, EventArgs e) =>

VertexColorDialogOpen.Visible = false;

private void VerticesColorPanel\_Enter(object sender, EventArgs e) =>

VertexColorDialogOpen.Visible = true;

private void VerticesColorPanel\_Click(object sender, EventArgs e) =>

VerticesColorPanel.Focus();

private void ArcsColorPanel\_Click(object sender, EventArgs e) =>

ArcsColorPanel.Focus();

private void ArcsColorPanel\_Enter(object sender, EventArgs e) =>

ArcsColorDialogOpen.Visible = true;

private void ArcsColorPanel\_Leave(object sender, EventArgs e) =>

ArcsColorDialogOpen.Visible = false;

#endregion

#endregion

private void UndoButton\_Click(object sender, EventArgs e)

{

if (isOnMovement) return;

radiusChanged = true;

commandsManager.Undo();

graphDrawing.DrawTheWholeGraph(digraph);

DrawingSurface.Image = graphDrawing.Image;

radiusChanged = false;

}

private void RedoButton\_Click(object sender, EventArgs e)

{

if (isOnMovement) return;

radiusChanged = true;

commandsManager.Redo();

graphDrawing.DrawTheWholeGraph(digraph);

DrawingSurface.Image = graphDrawing.Image;

radiusChanged = false;

}

}

}

* + - 1. **GridBehavior.cs**

using DotsMovementModelingAppLib.Commands;

using System;

using System.Drawing;

using System.Globalization;

using System.Windows.Forms;

namespace DotsMovementModelingApp

{

public partial class MainWindow

{

#region Adjacency and arcs

private void GridAdjacencyMatrix\_CellClick(object sender, DataGridViewCellEventArgs e)

{

foreach (var arc in digraph.Arcs)

{

if (arc.StartVertex != e.RowIndex || arc.EndVertex != e.ColumnIndex)

continue;

ArcName.SelectedIndex = ArcName.Items.IndexOf(arc.ToString());

ArcLength.Text = arc.Length.ToString(CultureInfo.CurrentCulture);

return;

}

GridAdjacencyMatrix.ClearSelection();

ArcName.Text = (e.RowIndex + 1) + @"-" + (e.ColumnIndex + 1);

ArcLength.Text = @"Error";

}

private void ArcLength\_TextChanged(object sender, EventArgs e) =>

ArcLength.ReadOnly = ArcLength.Text == @"Error";

private void ArcName\_TextChanged(object sender, EventArgs e)

{

try

{

int start = int.Parse(ArcName.Text[0].ToString()) - 1;

int end = int.Parse(ArcName.Text[2].ToString()) - 1;

GridAdjacencyMatrix[end, start].Selected = true;

if (GridAdjacencyMatrix[end, start].Value.ToString() == "0")

{

ArcLength.Text = @"Error";

GridAdjacencyMatrix.ClearSelection();

}

else ArcLength.Text = GridAdjacencyMatrix[end, start].Value.ToString();

}

catch (Exception) { ArcLength.Text = @"Error"; }

}

/// <summary>

/// Changes the length of the arc selected in a ComboBox to a value in a TextBox

/// </summary>

private void OkWeight\_Click(object sender, EventArgs e)

{

ArcLength.Text = ArcLength.Text.Trim(' ');

if (string.IsNullOrEmpty(ArcName.Text))

{

MessageBox.Show(@"Please, select an edge", @"Error", MessageBoxButtons.OK, MessageBoxIcon.Error);

return;

}

var selectedArc = ArcName.Items.IndexOf(ArcName.Text);

if (selectedArc == -1)

{

MessageBox.Show(@"The edge doesn't exist", @"Error", MessageBoxButtons.OK, MessageBoxIcon.Error);

return;

}

try

{

var length = new MathParserTK.MathParser().Parse(ArcLength.Text);

var command = new ChangeArcLengthCommand(digraph, selectedArc, digraph.Arcs[selectedArc].Length, length);

command.Executed += (s, ea) => GridAdjacencyMatrix[digraph.Arcs[selectedArc].EndVertex, digraph.Arcs[selectedArc].StartVertex].Value = s;

commandsManager.Execute(command);

ArcLength.Text = length.ToString(CultureInfo.CurrentCulture);

}

catch (Exception)

{

MessageBox.Show(@"Invalid value. Make sure the input value is greater than zero and is the correct mathematical expression.",

@"Error", MessageBoxButtons.OK, MessageBoxIcon.Error);

}

}

#endregion

private void GridParameters\_CellValueChanged(object sender, DataGridViewCellEventArgs e)

{

if (e.RowIndex < 0 || e.ColumnIndex < 0) return;

bool isValid = int.TryParse(GridParameters[e.ColumnIndex, e.RowIndex].Value.ToString(), out int value)

&& value >= 0;

switch (e.ColumnIndex)

{

case 0:

if (!isValid || value < 1)

{

GridParameters[e.ColumnIndex, e.RowIndex].Value = digraph.Thresholds[e.RowIndex];

return;

}

digraph.Thresholds[e.RowIndex] = value;

break;

case 1:

if (!isValid)

{

GridParameters[e.ColumnIndex, e.RowIndex].Value = digraph.RefractoryPeriods[e.RowIndex];

return;

}

digraph.RefractoryPeriods[e.RowIndex] = value;

break;

case 2:

if (!isValid)

{

GridParameters[e.ColumnIndex, e.RowIndex].Value = digraph.State[e.RowIndex];

return;

}

digraph.State[e.RowIndex] = value;

break;

}

GridParameters.CellValueChanged -= GridParameters\_CellValueChanged;

GridParameters[e.ColumnIndex, e.RowIndex].Value = value;

GridParameters.CellValueChanged += GridParameters\_CellValueChanged;

}

private void GridParameters\_RowsAdded(object sender, DataGridViewRowsAddedEventArgs e)

{

if (GridParameters.RowCount == 0) return;

if ((GridParameters.RowCount + 1) \* GridParameters.Rows[0].Height > GridParameters.Height)

for (int i = 0; i < GridParameters.ColumnCount; i++)

GridParameters.Columns[i].Width = 61;

}

private void GridParameters\_RowsRemoved(object sender, DataGridViewRowsRemovedEventArgs e)

{

if (GridParameters.RowCount == 0) return;

if ((GridParameters.RowCount + 1) \* GridParameters.Rows[0].Height <= GridParameters.Height)

for (int i = 0; i < GridParameters.ColumnCount; i++)

GridParameters.Columns[i].Width = 70;

}

#region Displaying

private void GridAdjacencyMatrix\_RowPrePaint(object sender, DataGridViewRowPrePaintEventArgs e)

{

e.PaintCells(e.ClipBounds, DataGridViewPaintParts.All);

e.PaintHeader(DataGridViewPaintParts.Background

| DataGridViewPaintParts.Border

| DataGridViewPaintParts.Focus

| DataGridViewPaintParts.SelectionBackground);

e.Handled = true;

e.Graphics.DrawString((e.RowIndex + 1).ToString(),

e.InheritedRowStyle.Font,

Brushes.Black,

new PointF(e.RowBounds.X + 5, e.RowBounds.Y + 2));

}

private void GridParameters\_RowPrePaint(object sender, DataGridViewRowPrePaintEventArgs e)

{

e.PaintCells(e.ClipBounds, DataGridViewPaintParts.All);

e.PaintHeader(DataGridViewPaintParts.Background

| DataGridViewPaintParts.Border

| DataGridViewPaintParts.Focus

| DataGridViewPaintParts.SelectionBackground);

e.Handled = true;

e.Graphics.DrawString((e.RowIndex + 1).ToString(),

e.InheritedRowStyle.Font,

Brushes.Black,

new PointF(e.RowBounds.X + 3, e.RowBounds.Y + 1));

}

private void SandpilePalette\_RowPrePaint(object sender, DataGridViewRowPrePaintEventArgs e)

{

e.PaintCells(e.ClipBounds, DataGridViewPaintParts.All);

e.PaintHeader(DataGridViewPaintParts.Background

| DataGridViewPaintParts.Border

| DataGridViewPaintParts.Focus

| DataGridViewPaintParts.SelectionBackground);

e.Handled = true;

e.Graphics.DrawString(e.RowIndex.ToString(),

e.InheritedRowStyle.Font,

Brushes.Black,

new PointF(e.RowBounds.X + 2, e.RowBounds.Y + 2));

}

#endregion

#region Adding and removing vertices

/// <summary>

/// Adds a new line and a row to Adjacency Matrix in DataGridView

/// </summary>

private void AddVertexToGridAdjacencyMatrix(int index)

{

var column = new DataGridViewColumn

{

Name = string.Empty,

HeaderText = (index + 1).ToString(),

FillWeight = 1,

Width = 35,

SortMode = DataGridViewColumnSortMode.NotSortable,

CellTemplate = new DataGridViewTextBoxCell()

};

GridAdjacencyMatrix.Columns.Insert(index, column);

GridAdjacencyMatrix.Rows.Insert(index);

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

{

GridAdjacencyMatrix[index, i].Value = 0;

GridAdjacencyMatrix[i, index].Value = 0;

GridAdjacencyMatrix.Columns[i].HeaderCell.Value = (i + 1).ToString();

GridAdjacencyMatrix.Rows[i].HeaderCell.Value = (i + 1).ToString();

}

GridAdjacencyMatrix.Rows[index].HeaderCell.Value = (index + 1).ToString();

}

private void RemoveVertexFromGridAdjacencyMatrix(int index)

{

GridAdjacencyMatrix.Columns.RemoveAt(index);

if (GridAdjacencyMatrix.Columns.Count != 0) GridAdjacencyMatrix.Rows.RemoveAt(index);

for (int j = index; j < digraph.Vertices.Count; j++)

{

GridAdjacencyMatrix.Columns[j].HeaderCell.Value = (j + 1).ToString();

GridAdjacencyMatrix.Rows[j].HeaderCell.Value = (j + 1).ToString();

}

}

private void AddVertexToGridParameters(int index)

{

if (GridParameters.ColumnCount == 0)

{

GridParameters.Columns.Add(String.Empty, "th");

GridParameters.Columns.Add(String.Empty, "p");

GridParameters.Columns.Add(String.Empty, "s");

for (int i = 0; i < GridParameters.ColumnCount; ++i)

{

GridParameters.Columns[i].FillWeight = 1;

GridParameters.Columns[i].Width = 70;

GridParameters.Columns[i].SortMode = DataGridViewColumnSortMode.NotSortable;

}

}

GridParameters.Rows.Insert(index);

GridParameters.Rows[index].HeaderCell.Value = digraph.Vertices.Count.ToString();

GridParameters[0, index].Value = digraph.Thresholds[index];

GridParameters[1, index].Value = digraph.RefractoryPeriods[index];

GridParameters[2, index].Value = digraph.State[index];

}

private void RemoveVertexFromGridParameters(int index)

{

GridParameters.Rows.RemoveAt(index);

for (int j = index; j < digraph.Vertices.Count; j++)

GridParameters.Rows[j].HeaderCell.Value = (j + 1).ToString();

}

#endregion

}

}

* + - 1. **Main.cs**

using DotsMovementModelingAppLib;

using DotsMovementModelingAppLib.Commands;

using System;

using System.Drawing;

using System.Windows.Forms;

using KeyEventArgs = System.Windows.Forms.KeyEventArgs;

using MouseEventArgs = System.Windows.Forms.MouseEventArgs;

namespace DotsMovementModelingApp

{

public partial class MainWindow : Form, IDisposable

{

public MainWindow()

{

InitializeComponent();

GraphBuilder\_SizeChanged(null, null);

graphDrawing = new GraphDrawing(DrawingSurface.Width, DrawingSurface.Height);

graphDrawing.RadiusChanged += (sender, args1) =>

{

if (BasicTypeCheckBox.Checked) graphDrawing.DrawTheWholeGraph(digraph);

else graphDrawing.DrawTheWholeGraphSandpile(digraph, false);

DrawingSurface.Image = graphDrawing.Image;

};

VerticesColorPanel.BackColor = graphDrawing.VerticesColor;

ArcsColorPanel.BackColor = graphDrawing.ArcsColor;

graphDrawing.SandpilePaletteChanged +=

(sender, e) =>

DigraphInformationDemonstration.DisplaySandpileColors(graphDrawing, SandpilePalette);

CommandsManager.CanRedoChanged += (sender, e) => RedoButton.Enabled = (bool)sender;

CommandsManager.CanUndoChanged += (sender, e) => UndoButton.Enabled = (bool)sender;

}

/// <summary>

/// Adjusts controls to fit the form's size

/// </summary>

private void GraphBuilder\_SizeChanged(object sender, EventArgs e)

{

#region Main menu items

Build.Location = new Point(Size.Width / 2 - Build.Size.Width / 2, Size.Height / 2 - Build.Size.Height - 50);

RandomGraph.Location = new Point(Build.Location.X, Build.Location.Y + Build.Size.Height + 10);

Open.Location = new Point(RandomGraph.Location.X, RandomGraph.Location.Y + RandomGraph.Size.Height + 10);

SquareLattice.Location = new Point(Open.Location.X, Open.Location.Y + Open.Size.Height + 10);

TriangleLattice.Location = new Point(Open.Location.X + Open.Size.Width - TriangleLattice.Size.Width,

Open.Location.Y + Open.Size.Height + 10);

#endregion

if (graphDrawing == null) return;

Cursor = Cursors.WaitCursor;

graphDrawing.Size = DrawingSurface.Size;

if (BasicTypeCheckBox.Checked || !isOnMovement) graphDrawing.DrawTheWholeGraph(digraph);

else graphDrawing.DrawTheWholeGraphSandpile(digraph, false);

DrawingSurface.Image = graphDrawing.Image;

Cursor = Cursors.Default;

}

/// <summary>

/// Moves digraph image on the drawing surface

/// </summary>

private void GraphBuilder\_KeyDown(object sender, KeyEventArgs e)

{

if (e.KeyCode == Keys.ControlKey) isControlPressed = true;

if (e.Modifiers != Keys.Control && !(sender is MouseEventArgs)) return;

switch (e.KeyCode)

{

case Keys.Right:

xCoefficient += 10;

break;

case Keys.Left:

xCoefficient -= 10;

break;

case Keys.Up:

yCoefficient -= 10;

break;

case Keys.Down:

yCoefficient += 10;

break;

case Keys.Oemplus:

resizeCoefficient \*= 1.1;

break;

case Keys.OemMinus:

resizeCoefficient \*= 0.9;

break;

case Keys.Z:

if (isOnMovement) break;

UndoButton\_Click(sender, e);

break;

case Keys.Y:

if (isOnMovement) break;

RedoButton\_Click(sender, e);

break;

default: return;

}

if (isOnMovement && SandpileTypeCheckBox.Checked)

graphDrawing.DrawTheWholeGraphSandpile(digraph, false, xCoefficient, yCoefficient, resizeCoefficient);

else graphDrawing.DrawTheWholeGraph(digraph, xCoefficient, yCoefficient, resizeCoefficient);

DrawingSurface.Image = graphDrawing.Image;

}

/// <summary>

/// Executes commands to move digraph itself

/// </summary>

private void MainWindow\_KeyUp(object sender, KeyEventArgs e)

{

if (e.KeyCode == Keys.ControlKey) isControlPressed = false;

if (e.Modifiers != Keys.Control && sender != MouseWheelTimer) return;

if (e.KeyCode == Keys.Up || e.KeyCode == Keys.Down ||

e.KeyCode == Keys.Right || e.KeyCode == Keys.Left)

{

if (xCoefficient == 0 && yCoefficient == 0) return;

var command = new MoveDigraphCommand(digraph, xCoefficient, yCoefficient);

commandsManager.Execute(command);

xCoefficient = yCoefficient = 0;

}

if (e.KeyCode == Keys.OemMinus || e.KeyCode == Keys.Oemplus)

{

var command = new ResizeDigraphCommand(digraph, resizeCoefficient);

commandsManager.Execute(command);

resizeCoefficient = 1;

}

}

/// <summary>

/// Shifts focus to allow user to move the graph

/// </summary>

private void Movement\_PreviewKeyDown(object sender, PreviewKeyDownEventArgs e)

{

if (e.Modifiers == Keys.Control)

Tools.Focus();

}

/// <summary>

/// Moves digraph image according to mouse wheel scrolling

/// </summary>

private void MainWindow\_MouseWheel(object sender, MouseEventArgs e)

{

MouseWheelTimer.Start();

if (isControlPressed)

{

if (e.Delta > 0) GraphBuilder\_KeyDown(e, new KeyEventArgs(Keys.Oemplus));

else if (e.Delta < 0) GraphBuilder\_KeyDown(e, new KeyEventArgs(Keys.OemMinus));

}

else

{

if (e.Delta > 0) GraphBuilder\_KeyDown(e, new KeyEventArgs(Keys.Up));

else if (e.Delta < 0) GraphBuilder\_KeyDown(e, new KeyEventArgs(Keys.Down));

}

}

/// <summary>

/// Moves digraph itself according to mouse wheel scrolling

/// </summary>

private void WheelStopped(object sender, EventArgs e)

{

MouseWheelTimer.Stop();

if (xCoefficient != 0 || yCoefficient != 0)

MainWindow\_KeyUp(MouseWheelTimer, new KeyEventArgs(Keys.Up));

if (Math.Abs(resizeCoefficient - 1) > 0)

MainWindow\_KeyUp(MouseWheelTimer, new KeyEventArgs(Keys.OemMinus));

if (!radiusChanged) return;

radiusChanged = false;

var command = new ChangeRadiusCommand(graphDrawing, radius, graphDrawing.R);

command.Executed += (s, ea) => RadiusTrackBar.Value = (int)s;

commandsManager.Execute(command);

radius = graphDrawing.R;

}

/// <summary>

/// Removes selections

/// </summary>

private void EscButton\_Click(object sender, EventArgs e)

{

if (isOnMovement) return;

vStart = vEnd = -1;

movedVertexIndex = -1;

graphDrawing.DrawTheWholeGraph(digraph, xCoefficient, yCoefficient, resizeCoefficient);

DrawingSurface.Image = graphDrawing.Image;

}

private void MainWindow\_FormClosing(object sender, FormClosingEventArgs e)

{

if (digraph.Vertices.Count == 0) return;

if (SaveGraph("Would you like to save the graph before leaving? Otherwise, your graph will be lost", "Saving") == DialogResult.Cancel)

e.Cancel = true;

}

public new void Dispose()

{

graphDrawing.Dispose();

}

/// <summary>

/// Subscribes handlers to digraph events

/// </summary>

private void SubscribeToDigraphEvents()

{

digraph.VertexAdded += (sender, e) =>

{

graphDrawing.DrawVertex(((Vertex)sender).X, ((Vertex)sender).Y,

e.Index + 1);

DrawingSurface.Image = graphDrawing.Image;

AddVertexToGridAdjacencyMatrix(e.Index);

AddVertexToGridParameters(e.Index);

ArcName.Items.Clear();

foreach (var arc in digraph.Arcs)

ArcName.Items.Add(arc.ToString());

};

digraph.ArcAdded += (sender, e) =>

{

ArcName.Items.Insert(e.Index, ((Arc)sender).ToString());

graphDrawing.DrawArc(digraph.Vertices[((Arc)sender).StartVertex],

digraph.Vertices[((Arc)sender).EndVertex],

(Arc)sender);

DrawingSurface.Image = graphDrawing.Image;

GridAdjacencyMatrix[((Arc)sender).EndVertex, ((Arc)sender).StartVertex].Value = ((Arc)sender).Length;

vStart = vEnd = -1;

};

digraph.VertexRemoved += (sender, e) =>

{

RemoveVertexFromGridAdjacencyMatrix(e.Index);

RemoveVertexFromGridParameters(e.Index);

ArcName.Items.Clear();

foreach (var arc in digraph.Arcs)

ArcName.Items.Add(arc.ToString());

};

digraph.ArcRemoved += (sender, e) =>

{

GridAdjacencyMatrix[((Arc)sender).EndVertex, ((Arc)sender).StartVertex].Value = 0;

ArcName.Items.RemoveAt(e.Index);

};

}

}

}

* + - 1. **MainMenu.cs**

using DotsMovementModelingAppLib;

using DotsMovementModelingAppLib.Commands;

using System;

using System.Drawing;

using System.IO;

using System.Windows.Forms;

using System.Xml.Serialization;

namespace DotsMovementModelingApp

{

public partial class MainWindow

{

/// <summary>

/// Opens the editor for creating a new graph

/// </summary>

private void Build\_Click(object sender, EventArgs e)

{

Cursor = Cursors.WaitCursor;

RefreshVariables();

UpdateDigraphInfo();

ChangeMainMenuState(false);

ChangeDrawingElementsState(true);

Cursor = Cursors.Default;

}

/// <summary>

/// Opens a form allowing to build a random graph

/// </summary>

private void RandomGraph\_Click(object sender, EventArgs e)

{

using (var randomDigraphForm = new RandomDigraphGeneratorForm(DrawingSurface.Width, DrawingSurface.Height))

{

randomDigraphForm.ShowDialog();

if (randomDigraphForm.Digraph == null) return;

digraph = randomDigraphForm.Digraph;

}

SubscribeToDigraphEvents();

UpdateDigraphInfo();

ChangeMainMenuState(false);

ChangeDrawingElementsState(true);

}

/// <summary>

/// Opens a file dialog to select a file with a digraph info,

/// reads it and opens a digraph in the editor

/// </summary>

private void Open\_Click(object sender, EventArgs e)

{

try

{

using (var openDialog = DigraphOpenFileDialog())

{

if (openDialog.ShowDialog() != DialogResult.OK) return;

RefreshVariables();

using (FileStream fs = new FileStream(openDialog.FileName, FileMode.Open))

{

XmlSerializer formatter = new XmlSerializer(typeof(Digraph));

digraph = (Digraph)formatter.Deserialize(fs);

}

}

SubscribeToDigraphEvents();

UpdateDigraphInfo();

ChangeMainMenuState(false);

ChangeDrawingElementsState(true);

}

catch (Exception ex)

{

MessageBox.Show(@"Invalid file:" + Environment.NewLine + ex.Message, @"Error",

MessageBoxButtons.OK, MessageBoxIcon.Error);

}

}

/// <summary>

/// Opens a form allowing to build a square lattice graph

/// </summary>

private void SquareLattice\_Click(object sender, EventArgs e)

{

using (var square = new SquareLatticeForm(DrawingSurface.Width, DrawingSurface.Height))

{

square.ShowDialog();

if (square.SquareLatticeDigraph == null) return;

digraph = square.SquareLatticeDigraph;

}

SubscribeToDigraphEvents();

UpdateDigraphInfo();

ChangeMainMenuState(false);

ChangeDrawingElementsState(true);

}

/// <summary>

/// Opens a form allowing to build a triangular lattice graph

/// </summary>

private void TriangleLattice\_Click(object sender, EventArgs e)

{

using (var triangle = new TriangularLatticeForm(DrawingSurface.Width, DrawingSurface.Height))

{

triangle.ShowDialog();

if (triangle.TriangularLatticeDigraph == null) return;

digraph = triangle.TriangularLatticeDigraph;

}

SubscribeToDigraphEvents();

UpdateDigraphInfo();

ChangeMainMenuState(false);

ChangeDrawingElementsState(true);

}

#region Additional methods

/// <summary>

/// Activates or deactivates graph editor controls

/// </summary>

private void ChangeDrawingElementsState(bool state)

{

DrawingSurface.Visible = state;

AppParameters.Visible = state;

if (!state)

{

TimeTextBox.Visible = false;

SandpilePanel.Visible = false;

}

saveToolStripMenuItem.Visible = state;

MainMenuToolStripMenuItem.Visible = state;

MovementToolStripMenuItem.Visible = state;

SandpilePanel.Size = new Size(358, 32);

Tools.Visible = state;

Tools.Refresh();

}

/// <summary>

/// Activates or deactivates main menu controls

/// </summary>

private void ChangeMainMenuState(bool state)

{

Build.Visible = state;

RandomGraph.Visible = state;

Open.Visible = state;

SquareLattice.Visible = state;

TriangleLattice.Visible = state;

}

/// <summary>

/// Redraws the digraph, refill DataGridViews with the digraph info

/// </summary>

private void UpdateDigraphInfo()

{

graphDrawing.DrawTheWholeGraph(digraph);

DrawingSurface.Image = graphDrawing.Image;

ArcName.Items.Clear();

ArcName.Text = String.Empty;

ArcLength.Text = String.Empty;

digraph.Arcs.ForEach(arc => ArcName.Items.Add((arc.StartVertex + 1) + "-" + (arc.EndVertex + 1)));

DigraphInformationDemonstration.DisplayGraphAdjacencyInfo(digraph.AdjacencyMatrix, GridAdjacencyMatrix);

DigraphInformationDemonstration.DisplayGraphParameters(digraph, GridParameters);

}

/// <summary>

/// Returns all the variables to its initial state

/// </summary>

private void RefreshVariables()

{

CursorButton\_Click(null, null);

if (isOnMovement) ResetToolStripMenuItem\_Click(null, null);

digraph = new Digraph();

SubscribeToDigraphEvents();

graphDrawing.ClearTheSurface();

DrawingSurface.Image = graphDrawing.Image;

ArcName.Items.Clear();

ArcName.Text = String.Empty;

ArcLength.Text = String.Empty;

GridAdjacencyMatrix.Columns.Clear();

GridParameters.Columns.Clear();

BasicTypeCheckBox.Checked = true;

ChartCheckBox.Checked = SaveGifCheckBox.Checked = SandpileChartType2.Checked = false;

SpeedNumeric.Value = 1;

RadiusTrackBar.Value = 8;

commandsManager = new CommandsManager();

UndoButton.Enabled = RedoButton.Enabled = false;

graphDrawing.ArcsColor = ArcsColorPanel.BackColor = Color.FromArgb(80, Color.MidnightBlue);

graphDrawing.VerticesColor = VerticesColorPanel.BackColor = Color.MidnightBlue;

}

#endregion

}

}

* + - 1. **ModelingParameters.cs**

using System;

using System.Drawing;

using System.Threading.Tasks;

using System.Windows.Media.Imaging;

namespace DotsMovementModelingApp

{

public partial class MainWindow

{

/// <summary>

/// Changes movement modeling type to basic

/// </summary>

private void BasicTypeCheckBox\_CheckedChanged(object sender, EventArgs e) =>

SandpileTypeCheckBox.Checked = !BasicTypeCheckBox.Checked;

/// <summary>

/// Changes movement modeling type to Sandpile

/// </summary>

private void SandpileTypeCheckBox\_CheckedChanged(object sender, EventArgs e)

{

BasicTypeCheckBox.Checked = !SandpileTypeCheckBox.Checked;

if (SandpileTypeCheckBox.Checked) ChartCheckBox\_CheckedChanged(sender, e);

else if (ChartCheckBox.Checked)

{

SandpileChartType1.Visible = SandpileChartType2.Visible = false;

SaveGifCheckBox.Location = new Point(11, 246);

SpeedLabel.Location = new Point(6, 308);

SpeedNumeric.Location = new Point(11, 346);

}

}

/// <summary>

/// Shows additional chart types for sandpile modeling

/// </summary>

private void ChartCheckBox\_CheckedChanged(object sender, EventArgs e)

{

if (SandpileTypeCheckBox.Checked && ChartCheckBox.Checked)

{

SaveGifCheckBox.Location =

new Point(SaveGifCheckBox.Location.X,

SandpileChartType2.Location.Y + SandpileChartType2.Size.Height + 10);

SpeedLabel.Location =

new Point(SpeedLabel.Location.X,

SaveGifCheckBox.Location.Y + SaveGifCheckBox.Size.Height + 28);

SpeedNumeric.Location =

new Point(SpeedNumeric.Location.X,

SpeedLabel.Location.Y + SpeedLabel.Size.Height + 10);

SandpileChartType1.Visible = SandpileChartType2.Visible = true;

return;

}

if (SandpileTypeCheckBox.Checked && !ChartCheckBox.Checked)

{

SandpileChartType1.Visible = SandpileChartType2.Visible = false;

SaveGifCheckBox.Location = new Point(11, 246);

SpeedLabel.Location = new Point(6, 308);

SpeedNumeric.Location = new Point(11, 346);

}

}

/// <summary>

/// Starts Sandpile movement after stock vertices are selected

/// </summary>

private void StockLabel\_Click(object sender, EventArgs e)

{

if (SandpilePanel.Size.Height > 60) return;

SandpilePanel.Visible = false;

TimeTextBox.Visible = true;

SandpileLabel.Enabled = false;

TimeTextBox.BringToFront();

SandpileLabel.Text = @"Select vertex to add a grain of sand to ";

SandpileLabel.Font = new Font("Segoe UI", 9);

SandpilePanel.Size = new Size(SandpilePanel.Size.Width, 91);

movement.MovementEnded += MovementEndedSandpileEventHandler;

movement.StartMovementModeling();

}

/// <summary>

/// Shows a tip for selecting a vertex to add sand to

/// </summary>

private void MovementEndedSandpileEventHandler(object sender, EventArgs e)

{

if (sender is int) return;

SandpilePanel.Visible = true;

SandpilePanel.BringToFront();

}

/// <summary>

/// Adds a grain of sand to a random vertex

/// </summary>

private async void RandomAddingLabel\_Click(object sender, EventArgs e)

{

if (sender is int) return;

int rndVertex;

do { rndVertex = Rnd.Next(digraph.Vertices.Count); }

while (digraph.Stock.Contains(rndVertex));

digraph.State[rndVertex]++;

SandpilePanel.Visible = false;

graphDrawing.HighlightVertexToAddSand(digraph.Vertices[rndVertex]);

DrawingSurface.Image = graphDrawing.Image;

if (SaveGifCheckBox.Checked && movement.MovementGif.Frames.Count < 250)

{

var bmp = ((Bitmap)DrawingSurface.Image).GetHbitmap();

var src = System.Windows.Interop.Imaging.CreateBitmapSourceFromHBitmap(

bmp,

IntPtr.Zero,

System.Windows.Int32Rect.Empty, BitmapSizeOptions.FromEmptyOptions());

movement.MovementGif.Frames.Add(BitmapFrame.Create(src));

DeleteObject(bmp);

}

await Task.Delay(1000);

movement?.Go();

}

private void RandomAddingCheckBox\_CheckedChanged(object sender, EventArgs e)

{

if (RandomAddingCheckBox.Checked)

{

movement.MovementEnded -= MovementEndedSandpileEventHandler;

movement.MovementEnded += RandomAddingLabel\_Click;

return;

}

movement.MovementEnded += MovementEndedSandpileEventHandler;

movement.MovementEnded -= RandomAddingLabel\_Click;

}

private void SandpileChartType1\_CheckedChanged(object sender, EventArgs e)

{

if (!SandpileChartType1.Checked)

SandpileChartType2.Checked = true;

}

private void SandpileChartType2\_CheckedChanged(object sender, EventArgs e)

{

if (!SandpileChartType2.Checked)

SandpileChartType1.Checked = true;

}

}

}

* + - 1. **MovementModeling.cs**

using DotsMovementModelingAppLib;

using DotsMovementModelingAppLib.Modeling;

using System;

using System.Collections.Generic;

using System.Drawing;

using System.IO;

using System.Windows.Forms;

using System.Windows.Media.Imaging;

namespace DotsMovementModelingApp

{

public partial class MainWindow

{

/// <summary>

/// Checks if the digraph is strongly connected and goes to the next step if it is

/// </summary>

private void MovementToolStripMenuItem\_Click(object sender, EventArgs e)

{

if (movement != null)

if (movement.IsActive || SandpilePanel.Visible) return; //returns is modeling is active or the app is waiting for other user's action

else //restarts movement if it's not over yet but not active at the moment

{

movement.Go();

MovementToolStripMenuItem.Enabled = false;

StopToolStripMenuItem.Enabled = true;

return;

}

if (movement == null && isOnMovement) return; //returns if movement is over but reset button wasn't clicked yet

if (!CheckConnectivity()) return; //Connectivity check before movement modeling start

ChangeWindowStateForMovementModeling(true);

var type = BasicTypeCheckBox.Checked

? MovementModelingType.Basic

: MovementModelingType.Sandpile;

var actions = GetModelingActions();

var sandpileChartTypes = GetChartTypes();

PrepareMovementModeling(type, actions, sandpileChartTypes);

movement.StartMovementModeling();

}

/// <summary>

/// Prepares MovementModeling instance for modeling the movement

/// </summary>

/// <param name="type">Modeling type</param>

/// <param name="actions">Additional actions</param>

/// <param name="sandpileChartTypes">Sandpile chart types</param>

private void PrepareMovementModeling(MovementModelingType type,

MovementModelingActions[] actions,

SandpileChartType[] sandpileChartTypes)

{

movement = new MovementModeling(digraph, (double)SpeedNumeric.Value / 1000, type, actions)

{

GraphDrawing = graphDrawing,

DrawingSurface = DrawingSurface,

SandpileChartTypes = sandpileChartTypes

};

isOnMovement = true;

movement.Tick += UpdateElapsedTime;

movement.MovementEnded += StopToolStripMenuItem\_Click;

MovementToolStripMenuItem.Text = @"Continue";

MovementToolStripMenuItem.Enabled = false;

StopToolStripMenuItem.Enabled = true;

if (type == MovementModelingType.Sandpile)

{

graphDrawing.DrawTheWholeGraphSandpile(digraph, true);

DrawingSurface.Image = graphDrawing.Image;

SandpilePanel.Visible = true;

SandpilePanel.BringToFront();

}

movement.MovementEnded += (s, ea) =>

{

movement.Tick -= UpdateElapsedTime;

movement = null;

};

if (SaveGifCheckBox.Checked) movement.MovementEnded += SaveGif;

TimeTextBox.Visible = true;

TimeTextBox.BringToFront();

}

#region Params Collecting methods

private MovementModelingActions[] GetModelingActions()

{

var modes = new List<MovementModelingActions>(2);

if (AnimationCheckBox.Checked) modes.Add(MovementModelingActions.Animation);

if (ChartCheckBox.Checked) modes.Add(MovementModelingActions.Chart);

if (SaveGifCheckBox.Checked) modes.Add(MovementModelingActions.Gif);

return modes.ToArray();

}

private SandpileChartType[] GetChartTypes()

{

if (!SandpileTypeCheckBox.Checked || !ChartCheckBox.Checked) return null;

var types = new List<SandpileChartType>(2);

if (SandpileChartType1.Checked) types.Add(SandpileChartType.NumberOfDotsChart);

if (SandpileChartType2.Checked) types.Add(SandpileChartType.AvalancheSizesDistributionChart);

return types.ToArray();

}

#endregion

/// <summary>

/// Checks if graph is valid for dots movement modeling

/// </summary>

private bool CheckConnectivity()

{

if (!ConnectivityCheck.IsGraphValid(digraph))

{

MessageBox.Show(

digraph.Vertices.Count >= 3

? @"The graph is not strongly connected"

: @"Not enough vertices",

@"Graph validation failed", MessageBoxButtons.OK,

MessageBoxIcon.Error);

return false;

}

return true;

}

/// <summary>

/// Saves movement modeling gif

/// </summary>

private void SaveGif(object sender, EventArgs e)

{

using (var fileDialog = SaveFileDialogForGifSaving())

if (fileDialog.ShowDialog() == DialogResult.OK)

using (FileStream stream = new FileStream(fileDialog.FileName, FileMode.Create))

{

var bmp = ((Bitmap)DrawingSurface.Image).GetHbitmap();

var src = System.Windows.Interop.Imaging.CreateBitmapSourceFromHBitmap(

bmp,

IntPtr.Zero,

System.Windows.Int32Rect.Empty, BitmapSizeOptions.FromEmptyOptions());

movement.MovementGif.Frames.Add(BitmapFrame.Create(src));

movement.MovementGif.Save(stream);

DeleteObject(bmp);

}

}

public void UpdateElapsedTime(object sender, MovementTickEventArgs e) =>

TimeTextBox.Text = @" Elapsed time, s: " + (e.ElapsedTime / 1000.0) + @"\*";

/// <summary>

/// Stops movement modeling

/// </summary>

private void StopToolStripMenuItem\_Click(object sender, EventArgs e)

{

if (movement == null || !movement.IsActive) return;

movement?.Stop();

MovementToolStripMenuItem.Enabled = true;

StopToolStripMenuItem.Enabled = false;

}

/// <summary>

/// Resets movement

/// </summary>

private void ResetToolStripMenuItem\_Click(object sender, EventArgs e)

{

StopToolStripMenuItem\_Click(sender, e);

isOnMovement = false;

//Saves gif if it's needed

if ((movement != null || SandpileTypeCheckBox.Checked)

&& SaveGifCheckBox.Checked) SaveGif(sender, e);

//Resets graph parameters

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

{

digraph.State[i] = int.Parse(GridParameters[2, i].Value.ToString());

if (digraph.RefractoryPeriods[i] == 0) continue;

digraph.TimeTillTheEndOfRefractoryPeriod?[i]?.Stop();

}

digraph.ResetStock();

graphDrawing.DrawTheWholeGraph(digraph);

DrawingSurface.Image = graphDrawing.Image;

movement = null;

TimeTextBox.Visible = false;

TimeTextBox.Text = @" Elapsed time, s: 0";

SandpilePanel.Visible = false;

SandpileLabel.Text = @"Select sink vertices and then click here ";

SandpileLabel.Font = new Font("Segoe UI", 9, FontStyle.Underline);

SandpilePanel.Size = new Size(SandpilePanel.Size.Width, 32);

ChangeWindowStateForMovementModeling(false);

MovementToolStripMenuItem.Text = @"Movement";

MovementToolStripMenuItem.Enabled = true;

GC.Collect();

}

/// <summary>

/// Clears selection from sandpile palette

/// </summary>

private void SandpilePalette\_SelectionChanged(object sender, EventArgs e) =>

SandpilePalette.ClearSelection();

/// <summary>

/// Changes selected controls state

/// </summary>

/// <param name="state">true if modeling starts, false if it's over</param>

private void ChangeWindowStateForMovementModeling(bool state)

{

CursorButton.Enabled = VertexButton.Enabled = EdgeButton.Enabled =

DeleteButton.Enabled = ClearButton.Enabled = !state;

if (!state) CursorButton.Enabled = false;

if (SandpileTypeCheckBox.Checked)

{

SandpilePalette.BringToFront();

SandpilePalette.Visible = state;

}

foreach (var page in AppParameters.Controls)

foreach (var control in ((TabPage)page).Controls)

if (control is DataGridView dgv) dgv.ReadOnly = state;

else ((Control)control).Enabled = !state;

if (!state)

{

GridAdjacencyMatrix.ReadOnly = true;

AnimationCheckBox.Enabled = false;

}

MovementToolStripMenuItem.Enabled = state;

StopToolStripMenuItem.Visible = state;

ResetToolStripMenuItem.Visible = state;

}

}

}

* + - 1. **TopMenu.cs**

using DotsMovementModelingAppLib;

using System;

using System.Drawing.Imaging;

using System.IO;

using System.Windows.Forms;

using System.Xml.Serialization;

namespace DotsMovementModelingApp

{

public partial class MainWindow

{

private void NewProjectToolStripMenuItem\_Click(object sender, EventArgs e) => Build\_Click(sender, e);

private void OpenProjectToolStripMenuItem\_Click(object sender, EventArgs e) => Open\_Click(TopMenu, e);

#region Digraph saving

/// <summary>

/// Saves digraph data

/// </summary>

private void DataToolStripMenuItem\_Click(object sender, EventArgs e)

{

using (var fileDialog = SaveFileDialogForDataSaving())

{

if (fileDialog.ShowDialog() != DialogResult.OK) return;

using (FileStream stream = new FileStream(fileDialog.FileName, FileMode.Create))

{

XmlSerializer format = new XmlSerializer(typeof(Digraph));

format.Serialize(stream, digraph);

}

}

}

/// <summary>

/// Saves digraph image

/// </summary>

private void SaveImageToolStripMenuItem\_Click(object sender, EventArgs e)

{

using (var fileDialog = SaveFileDialogForImageSaving())

{

if (fileDialog.ShowDialog() != DialogResult.OK) return;

using (FileStream stream = new FileStream(fileDialog.FileName, FileMode.Create))

{

graphDrawing.DrawTheWholeGraph(digraph);

graphDrawing.Image.Save(stream, ImageFormat.Jpeg);

}

}

}

/// <summary>

/// Saves digraph data and image to a folder

/// </summary>

private void SaveAllToolStripMenuItem\_Click(object sender, EventArgs e)

{

using (var folderDialog = FolderBrowserDialogForGraphSaving())

{

if (folderDialog.ShowDialog() != DialogResult.OK) return;

Directory.CreateDirectory(folderDialog.SelectedPath + @"\Graph");

using (var stream = new FileStream(folderDialog.SelectedPath + @"\Graph\Image.jpg", FileMode.Create))

{

graphDrawing.DrawTheWholeGraph(digraph);

graphDrawing.Image.Save(stream, ImageFormat.Jpeg);

}

using (var stream = new StreamWriter(folderDialog.SelectedPath + @"\Graph\Data.dgmm", false))

{

XmlSerializer format = new XmlSerializer(typeof(Digraph));

format.Serialize(stream, digraph);

}

}

}

#endregion

/// <summary>

/// Closes the app

/// </summary>

private void ExitToolStripMenuItem\_Click(object sender, EventArgs e) => Close();

/// <summary>

/// Saves the graph if user wants to and goes to a start window

/// </summary>

private void MainMenuToolStripMenuItem\_Click(object sender, EventArgs e)

{

if (digraph.Vertices.Count != 0

&& SaveGraph("Would you like to save the graph? Otherwise, your graph will be lost.",

"Saving") == DialogResult.Cancel) return;

ChangeDrawingElementsState(false);

RefreshVariables();

ChangeMainMenuState(true);

}

/// <summary>

/// Giving information about the application and the developer

/// </summary>

private void AboutToolStripMenuItem\_Click(object sender, EventArgs e) =>

MessageBox.Show(AboutApp, @"About", MessageBoxButtons.OK, MessageBoxIcon.Information);

/// <summary>

/// Saves the digraph to file if user wants to

/// </summary>

private DialogResult SaveGraph(string message, string caption)

{

var result = MessageBox.Show(message, caption, MessageBoxButtons.YesNoCancel, MessageBoxIcon.Question);

if (result == DialogResult.Yes)

SaveAllToolStripMenuItem\_Click(this, null);

return result;

}

}

}

* + - 1. **Variables.cs**

using DotsMovementModelingAppLib;

using DotsMovementModelingAppLib.Commands;

using DotsMovementModelingAppLib.Modeling;

using System;

using System.Windows.Forms;

namespace DotsMovementModelingApp

{

public partial class MainWindow

{

/// <summary>

/// GraphDrawing instance containing methods for digraph drawing

/// </summary>

private readonly GraphDrawing graphDrawing;

/// <summary>

/// Digraph instance with lists of vertices and arcs

/// </summary>

private Digraph digraph = new Digraph();

/// <summary>

/// Commands manager providing udo redo functions

/// </summary>

private CommandsManager commandsManager = new CommandsManager();

// Indices of the vertices selected for edge drawing

private int vStart = -1; private int vEnd = -1;

// Indicator showing whether the mouse button is pressed

private bool isPressed;

// Index of the moving vertex

private int movedVertexIndex = -1;

// Moving vertex itself

private Vertex movedVertex;

// Time during which the movement occurred

private DateTime ticks;

/// <summary>

/// Models dots movement on a digraph

/// </summary>

private MovementModeling movement;

/// <summary>

/// Shows if the program is currently modeling the movement

/// </summary>

private bool isOnMovement;

/// <summary>

/// Random values generator

/// </summary>

private static readonly Random Rnd = new Random();

/// <summary>

/// Information about the application

/// </summary>

private static readonly string AboutApp =

"The application developed as a part of a coursework" + Environment.NewLine +

Environment.NewLine +

Environment.NewLine + "Developed by Baranova Anastasia Andreevna, BSE196." +

Environment.NewLine +

"Supervisor: Vsevolod L. Chernyshev, Associate Professor, Big Data and Information Retrieval School, Faculty of Computer Science." +

Environment.NewLine +

Environment.NewLine + "Higher School of Economics, Moscow, 2020";

/// <summary>

/// X-axis offset

/// </summary>

private int xCoefficient;

/// <summary>

/// Y-axis offset

/// </summary>

private int yCoefficient;

/// <summary>

/// Resize coefficient

/// </summary>

private double resizeCoefficient = 1;

/// <summary>

/// Shows is Control button is currently pressed

/// </summary>

private bool isControlPressed;

/// <summary>

/// Vertices radius

/// </summary>

private int radius = 8;

/// <summary>

/// Shows if vertices radius was changed

/// </summary>

private bool radiusChanged;

#region File dialods

private static SaveFileDialog SaveFileDialogForDataSaving() =>

new SaveFileDialog()

{

FileName = "DigraphData",

DefaultExt = ".dgmm",

Filter = @"Digraph data files (.dgmm)|\*.dgmm"

};

private static SaveFileDialog SaveFileDialogForImageSaving() =>

new SaveFileDialog()

{

FileName = "GraphImage",

DefaultExt = ".jpg",

Filter = @"JPG Image (.jpg)|\*.jpg"

};

private static SaveFileDialog SaveFileDialogForGifSaving() =>

new SaveFileDialog()

{

FileName = "Movement",

DefaultExt = ".gif",

Filter = @"Gif Image (.gif)|\*.gif",

};

private static FolderBrowserDialog FolderBrowserDialogForGraphSaving() =>

new FolderBrowserDialog() { SelectedPath = @"Digraph" };

private static OpenFileDialog DigraphOpenFileDialog() =>

new OpenFileDialog()

{

DefaultExt = ".dgmm",

Filter = @"Digraph data files (.dgmm)|\*.dgmm"

};

#endregion

}

}

* 1. **DotsMovementModelingAppLib**
     1. **DotsMovementModelingAppLib.Commands**
        1. **AddArcCommand.cs**

using System;

namespace DotsMovementModelingAppLib.Commands

{

public class AddArcCommand : ICommand

{

/// <summary>

/// Digraph to which an arc is added

/// </summary>

private readonly Digraph digraph;

/// <summary>

/// Adding arc

/// </summary>

private Arc arc;

/// <summary>

/// Arc index

/// </summary>

private int index = -1;

/// <summary>

/// Initializes a new AddArcCommand instance

/// </summary>

/// <param name="digraph">Digraph to which an arc is added</param>

/// <param name="arc">Adding Arc</param>

/// <exception cref="ArgumentNullException"/>

public AddArcCommand(Digraph digraph, Arc arc)

{

this.digraph = digraph ?? throw new ArgumentNullException(nameof(digraph));

this.arc = arc;

}

/// <summary>

/// Executes the command

/// </summary>

public void Execute()

{

digraph.AddArc(arc, index);

index = digraph.Arcs.IndexOf(arc);

}

/// <summary>

/// UnExecutes the command

/// </summary>

public void UnExecute()

{

arc = digraph.Arcs[index];

digraph.RemoveArc(index);

}

}

}

* + - 1. **AddVertexCommand.cs**

using System;

namespace DotsMovementModelingAppLib.Commands

{

public class AddVertexCommand : ICommand

{

/// <summary>

/// Digraph to which a vertex is added

/// </summary>

private readonly Digraph digraph;

/// <summary>

/// Adding vertex

/// </summary>

private Vertex vertex;

/// <summary>

/// Vertex threshold

/// </summary>

private int threshold = 1;

/// <summary>

/// Vertex refractory period

/// </summary>

private int refractoryPeriod;

/// <summary>

/// Vertex initial state

/// </summary>

private int state;

/// <summary>

/// Vertex index

/// </summary>

private int index = -1;

/// <summary>

/// Initializes a new AddVertexCommand instance

/// </summary>

/// <param name="digraph">Digraph to which a vertex is added</param>

/// <param name="vertex">Adding vertex</param>

/// <exception cref="ArgumentNullException"/>

public AddVertexCommand(Digraph digraph, Vertex vertex)

{

this.digraph = digraph ?? throw new ArgumentNullException(nameof(digraph));

this.vertex = vertex;

}

/// <summary>

/// Executes the command

/// </summary>

public void Execute()

{

digraph.AddVertex(vertex, threshold, refractoryPeriod, state, index);

index = digraph.Vertices.IndexOf(vertex);

}

/// <summary>

/// UnExecutes the command

/// </summary>

public void UnExecute()

{

vertex = digraph.Vertices[index];

threshold = digraph.Thresholds[index];

refractoryPeriod = digraph.RefractoryPeriods[index];

state = digraph.State[index];

digraph.RemoveVertex(index);

}

}

}

* + - 1. **ChangeArcLengthCommand.cs**

using System;

namespace DotsMovementModelingAppLib.Commands

{

public class ChangeArcLengthCommand : ICommand

{

/// <summary>

/// Digraph to which a vertex is added

/// </summary>

private readonly Digraph digraph;

/// <summary>

/// Changed value index

/// </summary>

private readonly int index;

/// <summary>

/// Parameter value before changes

/// </summary>

private readonly double oldValue;

/// <summary>

/// New Parameter value

/// </summary>

private readonly double newValue;

/// <summary>

/// Occurs when the command executes or unexecutes

/// </summary>

public event EventHandler Executed;

/// <summary>

/// Initializes a new ChangeGraphParameterCommand instance

/// </summary>

/// <param name="digraph">Digraph whose arc is changed</param>

/// <param name="index">Changed arc index</param>

/// <param name="oldValue">Arc length before changes</param>

/// <param name="newValue">New length</param>

/// <exception cref="ArgumentNullException"/>

/// <exception cref="ArgumentOutOfRangeException"/>

public ChangeArcLengthCommand(Digraph digraph, int index, double oldValue, double newValue)

{

this.digraph = digraph ?? throw new ArgumentNullException(nameof(digraph));

if (oldValue <= 0)

throw new ArgumentOutOfRangeException(nameof(newValue), @"Arc length must be positive");

if (newValue <= 0)

throw new ArgumentOutOfRangeException(nameof(newValue), @"Arc length must be positive");

this.index = index;

this.oldValue = oldValue;

this.newValue = newValue;

}

/// <summary>

/// Executes the command

/// </summary>

public void Execute()

{

digraph.Arcs[index] = new Arc(digraph.Arcs[index].StartVertex, digraph.Arcs[index].EndVertex, newValue);

Executed?.Invoke(newValue, null);

}

/// <summary>

/// UnExecutes the command

/// </summary>

public void UnExecute()

{

digraph.Arcs[index] = new Arc(digraph.Arcs[index].StartVertex, digraph.Arcs[index].EndVertex, oldValue);

Executed?.Invoke(oldValue, null);

}

}

}

* + - 1. **ChangeColorCommand.cs**

using System;

using System.Drawing;

namespace DotsMovementModelingAppLib.Commands

{

public class ChangeColorCommand : ICommand

{

/// <summary>

/// GraphDrawing instance in which color is changed

/// </summary>

private readonly GraphDrawing target;

/// <summary>

/// Shows if the color of arcs or vertices is changed

/// </summary>

private readonly Type type;

/// <summary>

/// Color before changes

/// </summary>

private readonly Color oldColor;

/// <summary>

/// New color

/// </summary>

private readonly Color newColor;

/// <summary>

/// Occurs when the command executes or unexecutes

/// </summary>

public event EventHandler Executed;

/// <summary>

/// Initializes a new ChangeColorCommand instance

/// </summary>

/// <param name="target">GraphDrawing instance in which color is changed</param>

/// <param name="type">Shows if the color of arcs or vertices is changed</param>

/// <param name="oldColor">Color before changes</param>

/// <param name="newColor">New color</param>

/// <exception cref="ArgumentException"/>

public ChangeColorCommand(GraphDrawing target, Type type, Color oldColor, Color newColor)

{

if (type != typeof(Arc) && type != typeof(Vertex))

throw new ArgumentException(nameof(type));

this.target = target;

this.type = type;

this.oldColor = oldColor;

this.newColor = newColor;

}

/// <summary>

/// Executes the command

/// </summary>

public void Execute()

{

if (type == typeof(Vertex)) target.VerticesColor = newColor;

else target.ArcsColor = newColor;

Executed?.Invoke(newColor, null);

}

/// <summary>

/// UnExecutes the command

/// </summary>

public void UnExecute()

{

if (type == typeof(Vertex)) target.VerticesColor = oldColor;

else target.ArcsColor = oldColor;

Executed?.Invoke(oldColor, null);

}

}

}

* + - 1. **ChangeRadiusCommand.cs**

using System;

namespace DotsMovementModelingAppLib.Commands

{

public class ChangeRadiusCommand : ICommand

{

/// <summary>

/// GraphDrawing instance in which radius is changed

/// </summary>

private readonly GraphDrawing target;

/// <summary>

/// Radius before changes

/// </summary>

private readonly int oldRadius;

/// <summary>

/// New radius

/// </summary>

private readonly int newRadius;

/// <summary>

/// Occurs when the command executes or unexecutes

/// </summary>

public event EventHandler Executed;

/// <summary>

/// Initializes a new ChangeColorCommand instance

/// </summary>

/// <param name="target">GraphDrawing instance in which color is changed</param>

/// <param name="oldRadius">Previous vertices radius</param>

/// <param name="newRadius">New vertices radius</param>

/// <exception cref="ArgumentException"/>

public ChangeRadiusCommand(GraphDrawing target, int oldRadius, int newRadius)

{

if (oldRadius < 8)

throw new ArgumentOutOfRangeException(nameof(oldRadius));

if (newRadius < 8)

throw new ArgumentOutOfRangeException(nameof(newRadius));

this.target = target;

this.oldRadius = oldRadius;

this.newRadius = newRadius;

}

/// <summary>

/// Executes the command

/// </summary>

public void Execute()

{

target.R = newRadius;

Executed?.Invoke(newRadius, null);

}

/// <summary>

/// UnExecutes the command

/// </summary>

public void UnExecute()

{

target.R = oldRadius;

Executed?.Invoke(oldRadius, null);

}

}

}

* + - 1. **CommandsManager.cs**

using System;

using System.Collections.Generic;

namespace DotsMovementModelingAppLib.Commands

{

public class CommandsManager

{

/// <summary>

/// Shows if there are commands to undo

/// </summary>

public bool CanUndo => UndoStack.Count > 0;

/// <summary>

/// Shows if there are commands to redo

/// </summary>

public bool CanRedo => RedoStack.Count > 0;

/// <summary>

/// Stack of commands to undo

/// </summary>

private Stack<ICommand> UndoStack { get; }

/// <summary>

/// Stack of commands to redo

/// </summary>

private Stack<ICommand> RedoStack { get; }

/// <summary>

/// Initializes a new CommandsManager instance

/// </summary>

public CommandsManager()

{

UndoStack = new Stack<ICommand>();

RedoStack = new Stack<ICommand>();

}

/// <summary>

/// Undoes last command

/// </summary>

public void Undo()

{

if (!CanUndo) return;

var command = UndoStack.Pop();

command.UnExecute();

RedoStack.Push(command);

if (!CanUndo) CanUndoChanged?.Invoke(false, null);

if (RedoStack.Count == 1) CanRedoChanged?.Invoke(true, null);

}

/// <summary>

/// Redoes command

/// </summary>

public void Redo()

{

if (!CanRedo) return;

var command = RedoStack.Pop();

command.Execute();

UndoStack.Push(command);

if (!CanRedo) CanRedoChanged?.Invoke(false, null);

if (UndoStack.Count == 1) CanUndoChanged?.Invoke(true, null);

}

/// <summary>

/// Executes command, pushes it into the undo stack

/// and clears the redo stack

/// </summary>

public void Execute(ICommand command)

{

command.Execute();

UndoStack.Push(command);

RedoStack.Clear();

if (UndoStack.Count == 1) CanUndoChanged?.Invoke(true, null);

CanRedoChanged?.Invoke(false, null);

}

/// <summary>

/// Occurs when CanUndo value changes

/// </summary>

public static event EventHandler CanUndoChanged;

/// <summary>

/// Occurs when CanRedo value changes

/// </summary>

public static event EventHandler CanRedoChanged;

}

}

* + - 1. **EraseArcCommand.cs**

using System;

namespace DotsMovementModelingAppLib.Commands

{

public class EraseArcCommand : ICommand

{

/// <summary>

/// Digraph from which an arc is removed

/// </summary>

private readonly Digraph digraph;

/// <summary>

/// Removing arc

/// </summary>

private Arc arc;

/// <summary>

/// Arc index

/// </summary>

private readonly int index;

/// <summary>

/// Initializes a new EraseArcCommand instance

/// </summary>

/// <param name="digraph">Digraph from which an arc is removed</param>

/// <param name="arc">Removing Arc</param>

/// <exception cref="ArgumentNullException"/>

/// <exception cref="ArgumentException"/>

public EraseArcCommand(Digraph digraph, Arc arc)

{

this.digraph = digraph ?? throw new ArgumentNullException(nameof(digraph));

index = digraph.Arcs.IndexOf(arc);

if (index == -1)

throw new ArgumentException(@"The digraph doesn't contain this arc", nameof(arc));

this.arc = arc;

}

/// <summary>

/// Executes the command

/// </summary>

public void Execute()

{

arc = digraph.Arcs[index];

digraph.RemoveArc(index);

}

/// <summary>

/// UnExecutes the command

/// </summary>

public void UnExecute() =>

digraph.AddArc(arc, index);

}

}

* + - 1. **EraseVertexCommand.cs**

using System;

using System.Collections.Generic;

namespace DotsMovementModelingAppLib.Commands

{

public class EraseVertexCommand : ICommand

{

/// <summary>

/// Digraph from which a vertex is removed

/// </summary>

private readonly Digraph digraph;

/// <summary>

/// Removing vertex

/// </summary>

private Vertex vertex;

/// <summary>

/// Vertex threshold

/// </summary>

private int threshold = 1;

/// <summary>

/// Vertex refractory period

/// </summary>

private int refractoryPeriod;

/// <summary>

/// Vertex initial state

/// </summary>

private int state;

/// <summary>

/// Vertex index

/// </summary>

private readonly int index;

/// <summary>

/// Array of arcs incident to this vertex

/// </summary>

private List<Arc> incidentArcs = new List<Arc>();

/// <summary>

/// Array of indices of the arcs incident to this vertex

/// </summary>

private List<int> arcsIndices = new List<int>();

/// <summary>

/// Initializes a new EraseVertexCommand instance

/// </summary>

/// <param name="digraph">Digraph from which a vertex is removed</param>

/// <param name="vertex">Removing vertex</param>

/// <exception cref="ArgumentNullException"/>

/// <exception cref="ArgumentException"/>

public EraseVertexCommand(Digraph digraph, Vertex vertex)

{

this.digraph = digraph ?? throw new ArgumentNullException(nameof(digraph));

index = digraph.Vertices.IndexOf(vertex);

if (index == -1)

throw new ArgumentException(@"The digraph doesn't contain this vertex", nameof(vertex));

this.vertex = vertex;

}

/// <summary>

/// Executes the command

/// </summary>

public void Execute()

{

vertex = digraph.Vertices[index];

threshold = digraph.Thresholds[index];

refractoryPeriod = digraph.RefractoryPeriods[index];

state = digraph.State[index];

incidentArcs = new List<Arc>();

arcsIndices = new List<int>();

for (var i = 0; i < digraph.Arcs.Count; i++)

{

if (digraph.Arcs[i].StartVertex != index && digraph.Arcs[i].EndVertex != index)

continue;

arcsIndices.Add(i);

incidentArcs.Add(digraph.Arcs[i]);

}

digraph.RemoveVertex(index);

}

/// <summary>

/// Executes the command

/// </summary>

public void UnExecute()

{

digraph.AddVertex(vertex, threshold, refractoryPeriod, state, index);

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

digraph.AddArc(incidentArcs[i], arcsIndices[i]);

}

}

}

* + - 1. **ICommand.cs**

namespace DotsMovementModelingAppLib.Commands

{

public interface ICommand

{

void Execute();

void UnExecute();

}

}

* + - 1. **MoveDigraphCommand.cs**

using System;

namespace DotsMovementModelingAppLib.Commands

{

public class MoveDigraphCommand : ICommand

{

/// <summary>

/// Digraph whose vertex is moving

/// </summary>

private readonly Digraph digraph;

/// <summary>

/// X axis offset

/// </summary>

private readonly int xCoefficient;

/// <summary>

/// Y axis offset

/// </summary>

private readonly int yCoefficient;

/// <summary>

/// Initializes a new MoveVertexCommand instance

/// </summary>

/// <param name="digraph">Digraph whose vertex is moving</param>

/// <param name="xCoefficient">X axis offset</param>

/// <param name="yCoefficient">Y axis offset</param>

/// <exception cref="ArgumentNullException"/>

public MoveDigraphCommand(Digraph digraph, int xCoefficient, int yCoefficient)

{

this.digraph = digraph ?? throw new ArgumentNullException(nameof(digraph));

this.xCoefficient = xCoefficient;

this.yCoefficient = yCoefficient;

}

/// <summary>

/// Executes the command

/// </summary>

public void Execute()

{

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

digraph.Vertices[i] =

new Vertex(digraph.Vertices[i].X + xCoefficient, digraph.Vertices[i].Y + yCoefficient);

}

/// <summary>

/// UnExecutes the command

/// </summary>

public void UnExecute()

{

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

digraph.Vertices[i] =

new Vertex(digraph.Vertices[i].X - xCoefficient, digraph.Vertices[i].Y - yCoefficient);

}

}

}

* + - 1. **MoveVertexCommand.cs**

using System;

using System.Drawing;

namespace DotsMovementModelingAppLib.Commands

{

public class MoveVertexCommand : ICommand

{

/// <summary>

/// Digraph whose vertex is moving

/// </summary>

private readonly Digraph digraph;

/// <summary>

/// Vertex coordinates before moving

/// </summary>

private Point oldPoint;

/// <summary>

/// New vertex coordinates

/// </summary>

private Point newPoint;

/// <summary>

/// Vertex index

/// </summary>

private readonly int index;

/// <summary>

/// Initializes a new MoveVertexCommand instance

/// </summary>

/// <param name="digraph">Digraph whose vertex is moving</param>

/// <param name="index">Vertex index</param>

/// <param name="oldPoint">Vertex coordinates before moving</param>

/// <param name="newPoint">New vertex coordinates</param>

/// <exception cref="ArgumentNullException"/>

/// <exception cref="ArgumentOutOfRangeException"/>

public MoveVertexCommand(Digraph digraph, int index, Point oldPoint, Point newPoint)

{

this.digraph = digraph ?? throw new ArgumentNullException(nameof(digraph));

if (digraph.Vertices.Count <= index)

throw new ArgumentOutOfRangeException(nameof(index));

this.oldPoint = oldPoint;

this.newPoint = newPoint;

this.index = index;

}

/// <summary>

/// Executes the command

/// </summary>

public void Execute() =>

digraph.Vertices[index] = new Vertex(newPoint.X, newPoint.Y);

/// <summary>

/// UnExecutes the command

/// </summary>

public void UnExecute() =>

digraph.Vertices[index] = new Vertex(oldPoint.X, oldPoint.Y);

}

}

* + - 1. **ResizeDigraphCommand.cs**

using System;

namespace DotsMovementModelingAppLib.Commands

{

public class ResizeDigraphCommand : ICommand

{

/// <summary>

/// Digraph whose vertex is moving

/// </summary>

private readonly Digraph digraph;

/// <summary>

/// Scaling coefficient

/// </summary>

private readonly double coefficient;

/// <summary>

/// Initializes a new ResizeDigraphCommand instance

/// </summary>

/// <param name="digraph">Digraph whose vertex is moving</param>

/// <param name="coefficient">X axis offset</param>

/// <exception cref="ArgumentNullException"/>

public ResizeDigraphCommand(Digraph digraph, double coefficient)

{

this.digraph = digraph ?? throw new ArgumentNullException(nameof(digraph));

this.coefficient = coefficient;

}

/// <summary>

/// Executes the command

/// </summary>

public void Execute()

{

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

digraph.Vertices[i] =

new Vertex((int)(digraph.Vertices[i].X \* coefficient), (int)(digraph.Vertices[i].Y \* coefficient));

}

/// <summary>

/// UnExecutes the command

/// </summary>

public void UnExecute()

{

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

digraph.Vertices[i] =

new Vertex((int)(digraph.Vertices[i].X / coefficient), (int)(digraph.Vertices[i].Y / coefficient));

}

}

}

* + 1. **DotsMovementModelingAppLib.Graph**
       1. **Arc.cs**

using System;

namespace DotsMovementModelingAppLib

{

/// <summary>

/// Represents a digraph arc

/// </summary>

[Serializable]

public struct Arc

{

/// <summary>

/// Index of the starting vertex of the Arc

/// </summary>

private int startVertex;

/// <summary>

/// Index of the ending vertex of the Arc

/// </summary>

///

private int endVertex;

/// <summary>

/// Length of the arc

/// </summary>

private double length;

/// <summary>

/// Initializes a new instance of the Arc class

/// </summary>

/// <param name="startVertex">Index of the starting vertex of the Arc</param>

/// <param name="endVertex">Index of the ending vertex of the Arc</param>

/// <param name="length">Length of the arc</param>

/// <exception cref="ArgumentOutOfRangeException"/>

/// <exception cref="ArgumentException"/>

public Arc(int startVertex, int endVertex, double length = 1)

{

if (startVertex == endVertex)

throw new ArgumentException(@"Arc cannot be a loop", nameof(endVertex));

if (startVertex < 0)

throw new ArgumentOutOfRangeException(nameof(startVertex), @"Index of the vertex was negative");

if (endVertex < 0)

throw new ArgumentOutOfRangeException(nameof(endVertex), @"Index of the vertex was negative");

if (length <= 0)

throw new ArgumentOutOfRangeException(nameof(length), @"Length of the arc should be a positive number");

this.startVertex = startVertex;

this.endVertex = endVertex;

this.length = length;

}

/// <summary>

/// Index of the starting vertex of the Arc

/// </summary>

/// <exception cref="ArgumentOutOfRangeException"/>

public int StartVertex

{

get => startVertex;

set

{

if (value < 0)

throw new ArgumentOutOfRangeException(nameof(value), @"Index of the vertex was negative");

startVertex = value;

}

}

/// <summary>

/// Index of the ending vertex of the Arc

/// </summary>

/// <exception cref="ArgumentOutOfRangeException"/>

public int EndVertex

{

get => endVertex;

set

{

if (value < 0)

throw new ArgumentOutOfRangeException(nameof(EndVertex), @"Index of the vertex was negative");

endVertex = value;

}

}

/// <summary>

/// Length of the arc

/// </summary>

/// <exception cref="ArgumentOutOfRangeException"/>

public double Length

{

get => length;

set

{

if (value < 0)

throw new ArgumentOutOfRangeException(nameof(Length), @"Length of the arc should be a positive number");

length = value;

}

}

public override string ToString() => $"{startVertex + 1}-{endVertex + 1}";

}

}

* + - 1. **Dihraph.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Windows.Forms;

using System.Xml.Serialization;

namespace DotsMovementModelingAppLib

{

/// <summary>

/// Represents a digraph

/// </summary>

[Serializable]

public class Digraph

{

/// <summary>

/// Digraph vertices list

/// </summary>

public List<Vertex> Vertices { get; private set; }

/// <summary>

/// Digraph arcs list

/// </summary>

public List<Arc> Arcs { get; private set; }

#region Digraph parameters

/// <summary>

/// Digraph vertices thresholds list

/// </summary>

public List<int> Thresholds { get; private set; }

/// <summary>

/// Vertices refractory periods in milliseconds

/// </summary>

public List<int> RefractoryPeriods { get; private set; }

/// <summary>

/// Number of dots at each vertex

/// </summary>

public List<int> State { get; private set; }

/// <summary>

/// Remaining time until the end of refractory period in milliseconds

/// </summary>

[XmlIgnore] public List<Timer> TimeTillTheEndOfRefractoryPeriod { get; private set; }

/// <summary>

/// List of indices of sink vertices (sandpile modeling)

/// </summary>

public List<int> Stock { get; private set; }

#endregion

/// <summary>

/// Initializes a new Digraph instance

/// </summary>

public Digraph()

{

Vertices = new List<Vertex>();

Arcs = new List<Arc>();

Thresholds = new List<int>();

RefractoryPeriods = new List<int>();

State = new List<int>();

Stock = new List<int>();

}

#region Events

/// <summary>

/// Occurs when vertex is added

/// </summary>

public event EventHandler<DigraphChangedEventArgs> VertexAdded;

/// <summary>

/// Occurs when vertex is removed

/// </summary>

public event EventHandler<DigraphChangedEventArgs> VertexRemoved;

/// <summary>

/// Occurs when arc is added

/// </summary>

public event EventHandler<DigraphChangedEventArgs> ArcAdded;

/// <summary>

/// Occurs when arc is removed

/// </summary>

public event EventHandler<DigraphChangedEventArgs> ArcRemoved;

#endregion

/// <summary>

/// Adds vertex to the list of digraph vertices

/// </summary>

/// <param name="vertex">Vertex to add</param>

/// <param name="threshold">Vertex threshold</param>

/// <param name="refractoryPeriod">Vertex refractory period</param>

/// <param name="initialState">Vertex initial state</param>

/// <param name="index">Vertex index</param>

/// <exception cref="ArgumentOutOfRangeException"/>

public void AddVertex(Vertex vertex, int threshold = 1, int refractoryPeriod = 0, int initialState = 0, int index = -1)

{

if (Vertices.Count >= 200)

throw new InvalidOperationException(@"Too many vertices. Unable to add a new one.");

if (threshold <= 0)

throw new ArgumentOutOfRangeException(nameof(threshold), @"The value of the vertex threshold must be a positive number");

if (refractoryPeriod < 0)

throw new ArgumentOutOfRangeException(nameof(refractoryPeriod),

@"The value of the vertex refractory period must be a non-negative number");

if (initialState < 0)

throw new ArgumentOutOfRangeException(nameof(initialState),

@"The value of the vertex initial state must be a non-negative number");

if (index == -1) index = Vertices.Count;

Vertices.Insert(index, vertex);

Thresholds.Insert(index, threshold);

RefractoryPeriods.Insert(index, refractoryPeriod);

State.Insert(index, initialState);

Arcs = Arcs.ConvertAll(arc =>

new Arc(arc.StartVertex >= index ? arc.StartVertex + 1 : arc.StartVertex,

arc.EndVertex >= index ? arc.EndVertex + 1 : arc.EndVertex, arc.Length));

VertexAdded?.Invoke(vertex, new DigraphChangedEventArgs(index));

}

/// <summary>

/// Removes vertex from the list of digraph vertices

/// </summary>

/// <param name="index">Index of the vertex in the list</param>

/// <exception cref="ArgumentOutOfRangeException"/>

public void RemoveVertex(int index)

{

if (Vertices.Count <= index || index < 0)

throw new ArgumentOutOfRangeException(nameof(index),

@"Index of the vertex must be a non-negative number less than the number of elements in the vertices list");

Arcs = Arcs.Where(arc => arc.StartVertex != index && arc.EndVertex != index).ToList();

Arcs = Arcs.ConvertAll(arc =>

new Arc(arc.StartVertex > index ? arc.StartVertex - 1 : arc.StartVertex,

arc.EndVertex > index ? arc.EndVertex - 1 : arc.EndVertex));

var removed = Vertices[index];

Vertices.RemoveAt(index);

Thresholds.RemoveAt(index);

RefractoryPeriods.RemoveAt(index);

State.RemoveAt(index);

VertexRemoved?.Invoke(removed, new DigraphChangedEventArgs(index));

}

/// <summary>

/// Adds arc

/// </summary>

/// <param name="arc">Arc for adding</param>

/// <param name="index">Arc index</param>

/// <exception cref="ArgumentOutOfRangeException"/>

public void AddArc(Arc arc, int index = -1)

{

if (arc.StartVertex >= Vertices.Count)

throw new ArgumentOutOfRangeException(nameof(arc.StartVertex),

@"Index of the vertex must be a non-negative number less than the number of elements in the vertices list");

if (arc.EndVertex >= Vertices.Count)

throw new ArgumentOutOfRangeException(nameof(arc.EndVertex),

@"Index of the vertex must be a non-negative number less than the number of elements in the vertices list");

if (!Arcs.TrueForAll(arc1 => arc1.StartVertex != arc.StartVertex || arc1.EndVertex != arc.EndVertex))

throw new ArgumentOutOfRangeException(nameof(arc), @"The Arc already exists");

if (index == -1) index = Arcs.Count;

Arcs.Insert(index, arc);

ArcAdded?.Invoke(arc, new DigraphChangedEventArgs(index));

}

/// <summary>

/// Removes arc from the list of digraph vertices

/// </summary>

/// <param name="index">Index of the arc in the list</param>

/// <exception cref="ArgumentOutOfRangeException"/>

public void RemoveArc(int index)

{

if (Arcs.Count <= index || index < 0)

throw new ArgumentOutOfRangeException(nameof(index),

@"Index of the arc must be a non-negative number less than the number of elements in the arcs list");

var removed = Arcs[index];

Arcs.RemoveAt(index);

ArcRemoved?.Invoke(removed, new DigraphChangedEventArgs(index));

}

/// <summary>

/// Resets digraph sink (sandpile modeling)

/// </summary>

public void ResetStock() => Stock = new List<int>();

/// <summary>

/// Sets timers responsible for compliance with refractory periods

/// </summary>

public void SetTimeTillTheEndOfRefractoryPeriod()

{

TimeTillTheEndOfRefractoryPeriod = new List<Timer>(RefractoryPeriods.Count);

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

{

if (RefractoryPeriods[i] > 0)

{

TimeTillTheEndOfRefractoryPeriod.Add(new Timer() { Interval = RefractoryPeriods[i] });

TimeTillTheEndOfRefractoryPeriod[i].Tick +=

(sender, e) => (sender as Timer)?.Stop();

}

else TimeTillTheEndOfRefractoryPeriod.Add(null);

}

}

/// <summary>

/// Graph Adjacency Matrix

/// </summary>

public double[,] AdjacencyMatrix

{

get

{

double[,] adjacencyMatrix = new double[Vertices.Count, Vertices.Count];

foreach (Arc arc in Arcs)

adjacencyMatrix[arc.StartVertex, arc.EndVertex] = arc.Length;

return adjacencyMatrix;

}

}

}

public class DigraphChangedEventArgs : EventArgs

{

/// <summary>

/// Index of changed item

/// </summary>

public readonly int Index;

/// <summary>

/// Initializes a new DigraphChangedEventArgs instance

/// </summary>

/// <param name="index">Index of changed item</param>

public DigraphChangedEventArgs(int index) => Index = index;

}

}

* + - 1. **Vertex.cs**

using System;

namespace DotsMovementModelingAppLib

{

/// <summary>

/// Represents a digraph vertex

/// </summary>

[Serializable]

public struct Vertex

{

/// <summary>

/// X coordinate of the vertex

/// </summary>

public int X;

/// <summary>

/// Y coordinate of the vertex

/// </summary>

public int Y;

/// <summary>

/// Initializes a new instance of the Vertex class

/// </summary>

/// <param name="x">X coordinate of the vertex</param>

/// <param name="y">Y coordinate of the vertex</param>

public Vertex(int x, int y)

{

X = x;

Y = y;

}

}

}

* + 1. **DotsMovementModelingAppLib.Modeling**
       1. **ChartWindow.cs**

using System;

using System.Drawing;

using System.IO;

using System.Windows.Forms;

using System.Windows.Forms.DataVisualization.Charting;

namespace DotsMovementModelingAppLib.Modeling

{

public partial class ChartWindow : Form

{

/// <summary>

/// Initializes a new ChartWindow instance

/// </summary>

public ChartWindow()

{

InitializeComponent();

}

/// <summary>

/// Prepares chart area for displaying Avalanche Sizes Distribution

/// </summary>

public void AvalancheSizesDistributionChartPrepare()

{

Text = @"Distribution of Avalanche Sizes Chart";

chart1.Series[0] = new Series("Distribution of\n\rAvalanche Sizes")

{

ChartType = SeriesChartType.Point,

Color = Color.DarkCyan,

MarkerStyle = MarkerStyle.Circle,

MarkerColor = Color.DarkCyan,

ChartArea = "Chart",

BorderWidth = 1

};

chart1.Series[0].ToolTip = "Size = #VALX,\n\rFrequency = #VALY";

chart1.ChartAreas[0].AxisX.Title = "Avalanche Size";

chart1.ChartAreas[0].AxisY.Title = "Frequency";

chart1.ChartAreas[0].AxisX.Interval = 2;

chart1.ChartAreas[0].AxisY.Interval = 5;

label1.Visible = false;

}

/// <summary>

/// Saves chart data (.csv)

/// </summary>

private void SaveData\_Click(object sender, EventArgs e)

{

using (var saveDialog = new SaveFileDialog

{

FileName = "ChartImage",

DefaultExt = ".jpg",

Filter = @"JPEG Image (.jpeg)|\*.jpeg"

})

{

if (saveDialog.ShowDialog() != DialogResult.OK) return;

using (var stream = new FileStream(saveDialog.FileName, FileMode.Create))

chart1.SaveImage(stream, ChartImageFormat.Jpeg);

}

}

/// <summary>

/// Saves chart Image

/// </summary>

private void SaveImage\_Click(object sender, EventArgs e)

{

using (var saveDialog = new SaveFileDialog

{

FileName = "ChartData",

DefaultExt = ".csv",

Filter = @"CSV file (.csv)|\*.csv"

})

{

if (saveDialog.ShowDialog() != DialogResult.OK) return;

using (var stream = new StreamWriter(saveDialog.FileName, false))

{

stream.WriteLine(chart1.ChartAreas[0].AxisX.Title + ";" + chart1.ChartAreas[0].AxisY.Title);

foreach (var point in chart1.Series[0].Points)

stream.WriteLine(point.XValue + ";" + point.YValues[0]);

}

}

}

/// <summary>

/// Saves folder with chart image and data

/// </summary>

private void SaveAll\_Click(object sender, EventArgs e)

{

using (var folderBrowser = new FolderBrowserDialog { SelectedPath = @"Chart" })

{

if (folderBrowser.ShowDialog() != DialogResult.OK) return;

Directory.CreateDirectory(folderBrowser.SelectedPath + @"\Chart");

using (var stream = new FileStream(folderBrowser.SelectedPath + @"\Chart\ChartImage.jpg", FileMode.Create))

chart1.SaveImage(stream, ChartImageFormat.Jpeg);

using (var stream = new StreamWriter(folderBrowser.SelectedPath + @"\Chart\ChartData.csv", false))

{

stream.WriteLine(chart1.ChartAreas[0].AxisX.Title + ";" + chart1.ChartAreas[0].AxisY.Title);

foreach (var point in chart1.Series[0].Points)

stream.WriteLine(point.XValue + ";" + point.YValues[0]);

}

}

}

}

}

* + - 1. **MovementModeling класс**
         1. **MovementModeling.cs**

using System;

using System.Collections.Generic;

using System.Diagnostics;

using System.Drawing;

using System.Linq;

using System.Windows.Forms;

using Timer = System.Windows.Forms.Timer;

#pragma warning disable 67

namespace DotsMovementModelingAppLib.Modeling

{

/// <summary>

/// Contains instance methods for Modeling the Movement of Points on Directed Metric Graph,

/// with the Condition of Synchronization at the Vertices

/// </summary>

public partial class MovementModeling

{

/// <summary>

/// Initializes new instance of MovementModeling class

/// </summary>

/// <param name="digraph">Digraph</param>

/// <param name="speed">Speed in unit per millisecond</param>

/// <param name="type">Modeling type</param>

/// <param name="actions">Array of additional actions</param>

/// <exception cref="ArgumentNullException"/>

/// <exception cref="ArgumentOutOfRangeException"/>

public MovementModeling(Digraph digraph, double speed, MovementModelingType type, MovementModelingActions[] actions)

{

if (speed <= 0)

throw new ArgumentOutOfRangeException(nameof(speed), @"Speed of movement should be positive");

this.digraph = digraph ?? throw new ArgumentNullException(nameof(digraph));

this.speed = speed;

this.type = type;

this.actions = actions;

IsActive = false;

MovementEnded += (sender, e) =>

{

if (type == MovementModelingType.Sandpile) GraphDrawing.DrawTheWholeGraphSandpile(digraph, false);

if (type == MovementModelingType.Basic) GraphDrawing.DrawTheWholeGraph(digraph);

DrawingSurface.Image = GraphDrawing.Image;

};

}

/// <summary>

/// Shows if the movement is currently active

/// </summary>

public bool IsActive { get; private set; }

#region Modeling parameters viriables

/// <summary>

/// Digraph for which movement is modeling

/// </summary>

private readonly Digraph digraph;

/// <summary>

/// Movement speed

/// </summary>

private readonly double speed;

/// <summary>

/// Modeling type

/// </summary>

private readonly MovementModelingType type;

/// <summary>

/// Array of additional actions

/// </summary>

private readonly MovementModelingActions[] actions;

/// <summary>

/// GraphDrawing for drawing animation frames

/// </summary>

public GraphDrawing GraphDrawing;

/// <summary>

/// PictureBox for displaying animation

/// </summary>

public PictureBox DrawingSurface;

/// <summary>

/// Array of sandpile chart types

/// </summary>

public SandpileChartType[] SandpileChartTypes;

#endregion

#region Variables directly related to modeling

/// <summary>

/// Digraph incidence list

/// </summary>

private List<Arc>[] incidenceList;

/// <summary>

/// List of arcs along which dots are currently moving

/// </summary>

private readonly List<Arc> involvedArcs = new List<Arc>();

/// <summary>

/// Stopwatches for each moving dot

/// </summary>

private readonly List<Stopwatch> stopwatches = new List<Stopwatch>();

/// <summary>

/// Timer updating the process data every few milliseconds

/// </summary>

private Timer mainTimer;

/// <summary>

/// Current avalanche size

/// </summary>

private bool[] avalanche;

private Predicate<int> releaseCondition;

private Action<int> stateChange;

private ChartWindow numberOfDotsChart;

private ChartWindow distributionChart;

private double time;

private int indexOfFixedDot = -1;

readonly Stopwatch stopwatchTime = new Stopwatch();

#endregion

/// <summary>

/// Starts dots movement

/// </summary>

public void StartMovementModeling()

{

digraph.SetTimeTillTheEndOfRefractoryPeriod();

// Fill incidence list

incidenceList = GetIncidenceList(digraph);

mainTimer = new Timer { Interval = 1 };

mainTimer.Tick += TickModeling;

// Select dots release condition

// and changes occurring to vertex state after the release

if (type == MovementModelingType.Basic)

{

releaseCondition = i => digraph.State[i] >= digraph.Thresholds[i]

&& (digraph.RefractoryPeriods[i] == 0 ||

!digraph.TimeTillTheEndOfRefractoryPeriod[i].Enabled);

stateChange = i => digraph.State[i] -= digraph.Thresholds[i];

}

else

{

releaseCondition = i => !digraph.Stock.Contains(i) && digraph.State[i] >= incidenceList[i].Count

&& (digraph.RefractoryPeriods[i] == 0 ||

!digraph.TimeTillTheEndOfRefractoryPeriod[i].Enabled);

stateChange = i => digraph.State[i] -= incidenceList[i].Count;

}

//Prepare chart windows if it's needed

if (actions.Contains(MovementModelingActions.Chart))

{

PrepareSandpileCharts();

PrepareBasicCharts();

numberOfDotsChart?.Show();

distributionChart?.Show();

}

//Add gif frames collecting

if (actions.Contains(MovementModelingActions.Gif))

mainTimer.Tick += TickAddFrame;

Go();

}

/// <summary>

/// Stops the movement

/// </summary>

public void Stop()

{

mainTimer.Stop();

stopwatches.ForEach(timer => timer.Stop());

stopwatchTime.Stop();

IsActive = false;

}

/// <summary>

/// Starts or restarts the movement

/// </summary>

public void Go()

{

if (IsMovementEnded)

{

MovementEnded?.Invoke(this, null);

return;

}

mainTimer.Start();

if (time > 0)

stopwatchTime.Start();

stopwatches.ForEach(timer => timer.Start());

IsActive = true;

}

/// <summary>

/// Occurs when the movement is ended

/// </summary>

public event EventHandler MovementEnded;

/// <summary>

/// Occurs when main timer tick occurs

/// </summary>

public event EventHandler<MovementTickEventArgs> Tick;

/// <summary>

/// Shows if modeling of basic movement is ended

/// </summary>

private bool IsMovementEndedBasic

{

get

{

if (involvedArcs.Count != 0) return false;

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

if (digraph.State[i] >= digraph.Thresholds[i])

return false;

return true;

}

}

/// <summary>

/// Shows if modeling of sandpile movement is ended

/// </summary>

private bool IsMovementEndedSandpile

{

get

{

if (involvedArcs.Count != 0) return false;

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

if (!digraph.Stock.Contains(i) && digraph.State[i] >= incidenceList[i].Count)

return false;

return true;

}

}

/// <summary>

/// Shows if the movement is ended

/// </summary>

public bool IsMovementEnded

{

get

{

if (type == MovementModelingType.Basic) return IsMovementEndedBasic;

return IsMovementEndedSandpile;

}

}

/// <summary>

/// Processes selected types of sandpile chart and prepares windows for displaying these charts

/// </summary>

private void PrepareSandpileCharts()

{

if (type != MovementModelingType.Sandpile) return;

if (SandpileChartTypes.Contains(SandpileChartType.NumberOfDotsChart))

numberOfDotsChart = new ChartWindow();

if (SandpileChartTypes.Contains(SandpileChartType.AvalancheSizesDistributionChart))

{

distributionChart = new ChartWindow();

avalanche = new bool[digraph.Vertices.Count];

MovementEnded += delegate

{

AddAvalancheSize();

avalanche = new bool[digraph.Vertices.Count];

};

distributionChart.AvalancheSizesDistributionChartPrepare();

distributionChart.Closing += (sender, e) => distributionChart = null;

}

}

/// <summary>

/// Processes basic modeling chart selection and prepares windows for displaying the chart

/// </summary>

private void PrepareBasicCharts()

{

if (type != MovementModelingType.Basic) return;

numberOfDotsChart = new ChartWindow();

numberOfDotsChart.Closing += (sender, e) => numberOfDotsChart = null;

}

#region Helper static methods

/// <summary>

/// Creates an array of lists of the arcs coming from the digraph vertices

/// </summary>

public static List<Arc>[] GetIncidenceList(Digraph digraph)

{

List<Arc>[] incidenceList = new List<Arc>[digraph.Vertices.Count];

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

incidenceList[i] = digraph.Arcs.Where(arc => arc.StartVertex == i).ToList();

return incidenceList;

}

/// <summary>

/// Returns time required to travel a given path at a given speed

/// </summary>

/// <param name="length">Path length</param>

/// <param name="speed">Speed in units per millisecond</param>

/// <returns>Time in milliseconds</returns>

public static double GetTime(double length, double speed) => length / speed;

/// <summary>

/// Returns current coordinates of the dot traveling along the arc

/// </summary>

/// <param name="start">Vertex from which the dot exited</param>

/// <param name="end">Vertex towards which the dot is now going</param>

/// <param name="length">Arc length</param>

/// <param name="timer">Elapsed time since the dot was exited</param>

/// <returns></returns>

public PointF GetPoint(Vertex start, Vertex end, double length, Stopwatch timer)

{

double len = timer.ElapsedMilliseconds \* speed;

double x = 1.0 \* start.X + (end.X - start.X) \* len / length;

double y = 1.0 \* start.Y + (end.Y - start.Y) \* len / length;

return new PointF((float)x, (float)y);

}

#endregion

}

public enum MovementModelingActions

{

Animation = 0,

Chart = 1,

Gif = 2

}

public enum MovementModelingType

{

Basic = 0,

Sandpile = 1

}

public enum SandpileChartType

{

NumberOfDotsChart = 0,

AvalancheSizesDistributionChart = 1

}

public class MovementTickEventArgs : EventArgs

{

public readonly long ElapsedTime;

/// <summary>

/// Initializes a new MovementTickEventArgs instance

/// </summary>

/// <param name="time">Elapsed time in milliseconds</param>

public MovementTickEventArgs(long time) =>

ElapsedTime = time;

}

}

* + - * 1. **TicksAnimation.cs**

using System;

using System.Diagnostics;

using System.Drawing;

using System.Linq;

using System.Windows;

namespace DotsMovementModelingAppLib.Modeling

{

public partial class MovementModeling

{

/// <summary>

/// Models and animates the process of dots movement

/// </summary>

private void TickModeling(object source, EventArgs e)

{

Tick?.Invoke(this, new MovementTickEventArgs(

indexOfFixedDot == -1 || indexOfFixedDot >= involvedArcs.Count

? (long)time

: (long)(time - GetTime(involvedArcs[indexOfFixedDot].Length, speed) +

stopwatches[indexOfFixedDot].ElapsedMilliseconds)));

int initialCount = involvedArcs.Count;

ProcessDots();

ProcessVertices();

UpdateChart(initialCount);

if (IsMovementEnded)

{

AddNumberOfDotsChartPoint((long)(time), initialCount);

AddNumberOfDotsChartPoint((long)(time), involvedArcs.Count);

Tick?.Invoke(this, new MovementTickEventArgs((long)time));

MovementEnded?.Invoke(this, null);

}

}

/// <summary>

/// Process vertices states to release new dots

/// </summary>

private void ProcessVertices()

{

int count = involvedArcs.Count; // number of 'old' dots

for (var i = 0; i < digraph.Vertices.Count; i++)

{

if (indexOfFixedDot == -1 && i == index && releaseCondition(i))

{

indexOfFixedDot = 0;

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

{

if (incidenceList[i][j].Length > incidenceList[i][indexOfFixedDot].Length)

indexOfFixedDot = j;

}

time += GetTime(incidenceList[i][indexOfFixedDot].Length, speed);

indexOfFixedDot += involvedArcs.Count;

stopwatchTime.Reset();

stopwatchTime.Start();

index = -1;

}

else if (i == index) index = -1;

if (!releaseCondition(i)) continue;

ReleaseDots(i);

}

CheckDotsNumber(20000);

if (indexOfFixedDot == -1)

{

if (involvedArcs.Count == 0) return;

indexOfFixedDot = involvedArcs.Count - 1;

if (Math.Abs(time) <= 0)

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

{

if (involvedArcs[i].Length > involvedArcs[indexOfFixedDot].Length)

indexOfFixedDot = i;

}

time += GetTime(involvedArcs[indexOfFixedDot].Length, speed) -

stopwatches[indexOfFixedDot].ElapsedMilliseconds;

stopwatchTime.Reset();

stopwatchTime.Start();

}

StartNewTimers(stopwatches.Count - count);

}

private int index = -1;

/// <summary>

/// Draws all the moving dots, removes all the dots got to their destination

/// and releases new dots if destination vertices are ready

/// </summary>

private void ProcessDots()

{

if (type == MovementModelingType.Basic)

GraphDrawing.DrawTheWholeGraph(digraph);

else GraphDrawing.DrawTheWholeGraphSandpile(digraph, false);

for (var i = 0; i < involvedArcs.Count; i++)

{

if (stopwatches[i].ElapsedMilliseconds >= GetTime(involvedArcs[i].Length, speed))

{

if (i == indexOfFixedDot)

{

stopwatchTime.Reset();

stopwatchTime.Start();

indexOfFixedDot = -1;

index = involvedArcs[i].EndVertex;

}

if (indexOfFixedDot > i) indexOfFixedDot--;

digraph.State[involvedArcs[i].EndVertex]++;

stopwatches.RemoveAt(i);

involvedArcs.RemoveAt(i);

i--;

continue;

}

PointF point =

GetPoint(digraph.Vertices[involvedArcs[i].StartVertex],

digraph.Vertices[involvedArcs[i].EndVertex],

involvedArcs[i].Length,

stopwatches[i]);

GraphDrawing.DrawDot(point);

}

CheckDotsNumber(20000);

if (type == MovementModelingType.Basic)

GraphDrawing.DrawVertices(digraph);

else GraphDrawing.DrawVerticesSandpile(digraph);

DrawingSurface.Image = GraphDrawing.Image;

}

/// <summary>

/// Checks if the vertex is ready to fire and releases new dots

/// </summary>

private void ReleaseDots(int vertexIndex)

{

if (!releaseCondition(vertexIndex)) return;

while (releaseCondition(vertexIndex))

{

involvedArcs.AddRange(incidenceList[vertexIndex]);

stopwatches.AddRange(incidenceList[vertexIndex].ConvertAll(arc => new Stopwatch()));

stateChange(vertexIndex);

digraph.TimeTillTheEndOfRefractoryPeriod[vertexIndex]?.Start();

}

if (distributionChart != null) avalanche[vertexIndex] = true;

}

/// <summary>

/// Checks current number of dots

/// and aborts the process if it exceeds the limit

/// </summary>

private void CheckDotsNumber(int limit)

{

if (stopwatches.Count > limit)

{

Stop();

MessageBox.Show("Operation has been aborted prematurely." + Environment.NewLine +

$"The number of dots exceeded the allowable mark of {limit}",

"Operation Aborted", MessageBoxButton.OK, MessageBoxImage.Error);

Tick?.Invoke(this, new MovementTickEventArgs((long)time));

MovementEnded?.Invoke(limit, null);

}

}

/// <summary>

/// Starts count last timers

/// </summary>

/// <param name="count">Number of timers to start</param>

private void StartNewTimers(int count)

{

int fired = 0;

for (int i = stopwatches.Count - 1; fired < count; i--, fired++)

{

stopwatches[i].Start();

digraph.TimeTillTheEndOfRefractoryPeriod[involvedArcs[i].StartVertex]?.Stop();

digraph.TimeTillTheEndOfRefractoryPeriod[involvedArcs[i].StartVertex]?.Start();

}

}

/// <summary>

/// Updates number of dots chart

/// </summary>

/// <param name="val">Number of dots before changes</param>

private void UpdateChart(int val)

{

if (!actions.Contains(MovementModelingActions.Chart)

|| numberOfDotsChart == null

|| val == stopwatches.Count

|| indexOfFixedDot == -1) return;

long point = (long)(time -

GetTime(involvedArcs[indexOfFixedDot].Length, speed) +

stopwatches[indexOfFixedDot].ElapsedMilliseconds);

AddNumberOfDotsChartPoint(point, val);

AddNumberOfDotsChartPoint(point, involvedArcs.Count);

}

}

}

* + - * 1. **TicksChartGif.cs**

using System;

using System.Drawing;

using System.Linq;

using System.Windows.Forms.DataVisualization.Charting;

using System.Windows.Media.Imaging;

namespace DotsMovementModelingAppLib.Modeling

{

public partial class MovementModeling

{

/// <summary>

/// GIF image of the process of the movement of points on digraph

/// </summary>

public readonly GifBitmapEncoder MovementGif = new GifBitmapEncoder();

/// <summary>

/// Collects frames for a GIF image of the process of the movement of points on digraph

/// (with a limit of 300 frames)

/// </summary>

private void TickAddFrame(object source, EventArgs e)

{

var bmp = ((Bitmap)DrawingSurface.Image).GetHbitmap();

var src = System.Windows.Interop.Imaging.CreateBitmapSourceFromHBitmap(

bmp,

IntPtr.Zero,

System.Windows.Int32Rect.Empty, BitmapSizeOptions.FromEmptyOptions());

MovementGif.Frames.Add(BitmapFrame.Create(src));

DeleteObject(bmp);

if (MovementGif.Frames.Count >= 300)

mainTimer.Tick -= TickAddFrame;

}

[System.Runtime.InteropServices.DllImport("gdi32.dll")]

public static extern bool DeleteObject(IntPtr hObject);

/// <summary>

/// Adds point to number of dots chart

/// </summary>

private void AddNumberOfDotsChartPoint(long t, int count)

{

if (numberOfDotsChart == null) return;

numberOfDotsChart?.chart1.Series[0].Points.AddXY(t / 1000.0, count);

if (count >= numberOfDotsChart.chart1.ChartAreas[0].AxisY.Maximum

|| t / 1000.0 >= numberOfDotsChart?.chart1.ChartAreas[0].AxisX.Maximum)

ChangeChartInterval(numberOfDotsChart?.chart1);

}

/// <summary>

/// Adds Avalanche Size to Avalanche Size distribution chart

/// </summary>

private void AddAvalancheSize()

{

if (distributionChart == null || !IsMovementEndedSandpile) return;

if (!avalanche.Contains(true)) return;

int size = avalanche.Count(v => v);

foreach (var point in distributionChart.chart1.Series[0].Points)

{

if (Math.Abs(point.XValue - size) > 0) continue;

distributionChart.chart1.Series[0].Points.AddXY(size, point.YValues[0] + 1);

distributionChart.chart1.Series[0].Points.Remove(point);

return;

}

distributionChart.chart1.Series[0].Points.AddXY(size, 1);

}

/// <summary>

/// Changes chart area axis intervals to fit values

/// </summary>

private static void ChangeChartInterval(Chart chart)

{

if (chart == null) return;

chart.ChartAreas[0].AxisY.Interval = (int)(chart.ChartAreas[0].AxisY.Maximum / 5);

chart.ChartAreas[0].AxisX.Interval = (int)(chart.ChartAreas[0].AxisX.Maximum / 10);

}

}

}

* + 1. **AppAdditions.cs**

using System;

using System.Windows.Forms;

namespace DotsMovementModelingAppLib

{

/// <summary>

/// Provides static methods for arcs and vertices deletion

/// </summary>

public static class DigraphComponentsRemover

{

#region Vertices and arcs deletion

/// <summary>

/// Searches for a digraph vertex at (x, y) to delete

/// </summary>

/// <param name="x">X coordinate of search point</param>

/// <param name="y">Y coordinate of search point</param>

/// <param name="digraph">Digraph among the vertices of which to search</param>

/// <param name="r">Vertex radius</param>

/// <param name="index">Index of the found vertex</param>

/// <returns>true if the vertex was found, false otherwise</returns>

public static bool TryToDeleteVertexAt(int x, int y, Digraph digraph, float r, out int index)

{

for (var i = 0; i < digraph.Vertices.Count; i++)

{

if (Math.Pow(digraph.Vertices[i].X - x, 2) + Math.Pow(digraph.Vertices[i].Y - y, 2) > r \* r)

continue;

index = i;

return true;

}

index = -1;

return false;

}

/// <summary>

/// Searches for a digraph arc at (x, y) and removes it

/// </summary>

/// <param name="x">X coordinate of search point</param>

/// <param name="y">Y coordinate of search point</param>

/// <param name="digraph">Digraph among the vertices of which to search</param>

/// <param name="deletedArc">Found arc</param>

/// <returns>true if the arc was found, false otherwise</returns>

public static bool TryToDeleteArcAt(int x, int y, Digraph digraph, out Arc deletedArc)

{

var selectedArc = FindSelectedArc(x, y, digraph);

if (selectedArc != -1)

{

deletedArc = digraph.Arcs[selectedArc];

return true;

}

deletedArc = new Arc();

return false;

}

/// <summary>

/// Searches for a digraph arc at (x, y)

/// </summary>

/// <returns>Found Arc index</returns>

private static int FindSelectedArc(int x, int y, Digraph digraph)

{

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

{

if (IsArcSelected(x, y, digraph.Vertices[digraph.Arcs[i].StartVertex].X,

digraph.Vertices[digraph.Arcs[i].StartVertex].Y,

digraph.Vertices[digraph.Arcs[i].EndVertex].X,

digraph.Vertices[digraph.Arcs[i].EndVertex].Y))

return i;

}

return -1;

}

/// <summary>

/// Checks if the arc is selected for deletion

/// </summary>

private static bool IsArcSelected(int x, int y, int startVertexX, int startVertexY, int endVertexX, int endVertexY)

{

return (Math.Abs((x - startVertexX) \* (endVertexY - startVertexY) - (y - startVertexY) \* (endVertexX - startVertexX)) <= 350 &&

(x > Math.Min(startVertexX, endVertexX) && x < Math.Max(startVertexX, endVertexX) ||

y > Math.Min(startVertexY, endVertexY) && y < Math.Max(startVertexY, endVertexY)));

}

#endregion

}

/// <summary>

/// Provides static methods for digraph parameters displaying

/// </summary>

public static class DigraphInformationDemonstration

{

#region DataGridView information display

/// <summary>

/// Shows Adjacency Matrix in DataGridView

/// </summary>

public static void DisplayGraphAdjacencyInfo(double[,] adjacencyMatrix, DataGridView dataGridView)

{

dataGridView.Rows.Clear();

dataGridView.Columns.Clear();

for (int i = 0; i < adjacencyMatrix.GetLength(1); i++)

{

var column = new DataGridViewColumn

{

Name = string.Empty,

HeaderText = (i + 1).ToString(),

FillWeight = 1,

Width = 35,

SortMode = DataGridViewColumnSortMode.NotSortable,

CellTemplate = new DataGridViewTextBoxCell(),

};

dataGridView.Columns.Add(column);

if (i == 0) dataGridView.Rows.Add(adjacencyMatrix.GetLength(1));

for (int j = 0; j < adjacencyMatrix.GetLength(0); j++)

{

dataGridView[i, j].Value = adjacencyMatrix[j, i];

dataGridView.Rows[j].HeaderCell.Value = (j + 1).ToString();

dataGridView.Rows[j].Height = 30;

}

}

}

/// <summary>

/// Shows all the digraph parameters in one DataGridView

/// </summary>

public static void DisplayGraphParameters(Digraph digraph, DataGridView dataGridView)

{

dataGridView.Rows.Clear();

dataGridView.Columns.Clear();

dataGridView.Columns.Add(String.Empty, "th");

dataGridView.Columns.Add(String.Empty, "p");

dataGridView.Columns.Add(String.Empty, "s");

for (int i = 0; i < dataGridView.ColumnCount; ++i)

{

dataGridView.Columns[i].FillWeight = 1;

dataGridView.Columns[i].Width = 70;

dataGridView.Columns[i].SortMode = DataGridViewColumnSortMode.NotSortable;

}

if (digraph.Vertices.Count <= 0) return;

dataGridView.Rows.Add(digraph.Vertices.Count);

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

{

dataGridView.Rows[i].HeaderCell.Value = (i + 1).ToString();

dataGridView.Rows[i].Height = 30;

dataGridView[0, i].Value = digraph.Thresholds[i];

dataGridView[1, i].Value = digraph.RefractoryPeriods[i];

dataGridView[2, i].Value = digraph.State[i];

}

}

/// <summary>

/// Shows sandpile colors palette in DataGridView

/// </summary>

public static void DisplaySandpileColors(GraphDrawing graphDrawing, DataGridView dataGridView)

{

dataGridView.Rows.Clear();

dataGridView.Columns.Clear();

dataGridView.Columns.Add("colors", "Color");

dataGridView.Columns[0].FillWeight = 1;

dataGridView.Columns[0].Width = 70;

dataGridView.Columns[0].SortMode = DataGridViewColumnSortMode.NotSortable;

dataGridView.Rows.Add(graphDrawing.SandpilePalette.Length);

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

{

dataGridView.Rows[i].HeaderCell.Value = i.ToString();

dataGridView.Rows[i].Height = 25;

dataGridView.Rows[i].Cells[0].Style.BackColor = graphDrawing.SandpilePalette[i];

}

dataGridView.Visible = true;

}

#endregion

}

}

* + 1. **ConnectivityCheck.cs**

using System;

using System.Collections.Generic;

using System.Linq;

namespace DotsMovementModelingAppLib

{

public class ConnectivityCheck

{

/// <summary>

/// Number of vertices of the graph

/// </summary>

private readonly int numberOfVertices;

/// <summary>

/// Graph Adjacency List

/// </summary>

private readonly List<int>[] adjacencyList;

/// <summary>

/// Initializes a new instance of the ConnectivityCheck class

/// </summary>

/// <param name="numberOfVertices">Number of vertices of the graph</param>

public ConnectivityCheck(int numberOfVertices)

{

if (numberOfVertices <= 0)

throw new ArgumentOutOfRangeException(nameof(numberOfVertices),

@"Number of vertices of the graph should be a positive number");

this.numberOfVertices = numberOfVertices;

adjacencyList = new List<int>[numberOfVertices];

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

adjacencyList[i] = new List<int>();

}

/// <summary>

/// Adds graph edge to Adjacency List

/// </summary>

public void AddArc(Arc edge) => adjacencyList[edge.StartVertex].Add(edge.EndVertex);

/// <summary>

/// Recursive DFS function

/// </summary>

/// <param name="startVertex">The vertex with which we start the traversing</param>

/// <param name="visited">Shows if vertices have been visited</param>

private void DFS(int startVertex, bool[] visited)

{

visited[startVertex] = true;

foreach (var i in adjacencyList[startVertex])

if (!visited[i])

DFS(i, visited);

}

/// <summary>

/// Returns an inverted graph

/// </summary>

/// <returns></returns>

private ConnectivityCheck GetInvertedGraph()

{

ConnectivityCheck g = new ConnectivityCheck(numberOfVertices);

for (int v = 0; v < numberOfVertices; v++)

foreach (int i in adjacencyList[v])

g.adjacencyList[i].Add(v);

return g;

}

/// <summary>

/// Connectivity check method (using Kosaraju algorithm)

/// </summary>

public bool IsStronglyConnected()

{

bool[] visited = new bool[numberOfVertices];

// Fist DFS traversing

DFS(0, visited);

// Returning false there is a vertex that hasn't been visited

if (visited.Contains(false)) return false;

// Inverting the graph

ConnectivityCheck gr = GetInvertedGraph();

visited = new bool[numberOfVertices]; // Refreshing

// DFS traversing for inverted graph

gr.DFS(0, visited);

// Returning false if there is a vertex that hasn't been visited

return Array.TrueForAll(visited, v => v);

}

/// <summary>

/// Checks if graph is valid

/// </summary>

public static bool IsGraphValid(Digraph digraph)

{

if (digraph.Vertices.Count < 3) return false;

ConnectivityCheck check = new ConnectivityCheck(digraph.Vertices.Count);

foreach (Arc arc in digraph.Arcs)

check.AddArc(arc);

return check.IsStronglyConnected();

}

}

}

* + 1. **GraphDrawing.cs**

using DotsMovementModelingAppLib.Modeling;

using System;

using System.Collections.Generic;

using System.Drawing;

using System.Linq;

namespace DotsMovementModelingAppLib

{

public class GraphDrawing : IDisposable

{

#region Variables

/// <summary>

/// Image of the Graph

/// </summary>

public Bitmap Image { get; private set; }

/// <summary>

/// Graphics instance for drawing

/// </summary>

private Graphics graphics;

/// <summary>

/// Pen for drawing vertices

/// </summary>

private readonly Pen verticesPen = new Pen(Color.MidnightBlue, 2.5f);

public Color VerticesColor

{

get => verticesPen.Color;

set => verticesPen.Color = value;

}

/// <summary>

/// Pen for drawing arcs

/// </summary>

private Pen arcsPen = new Pen(Color.FromArgb(80, Color.MidnightBlue), 3);

public Color ArcsColor

{

get => arcsPen.Color;

set => arcsPen = new Pen(Color.FromArgb(80, value), 3);

}

/// <summary>

/// Pen for highlighting vertices

/// </summary>

private readonly Pen highlightPen = new Pen(Color.MediumAquamarine, 2.5f);

/// <summary>

/// Pen for highlighting vertices for adding sand (sandpile modeling)

/// </summary>

private readonly Pen highlightSandpilePen = new Pen(Color.Gold, 4);

/// <summary>

/// Font for vertices titles

/// </summary>

private Font font = new Font("Segoe UI", 5);

/// <summary>

/// Brush for writing titles

/// </summary>

private readonly Brush brush = Brushes.Black;

/// <summary>

/// Drawing back color

/// </summary>

public Color BackColor { get; set; }

/// <summary>

/// Vertices radius

/// </summary>

private static int \_radius = 8;

/// <summary>

/// Vertices radius

/// </summary>

public int R

{

get => \_radius;

set

{

if (value < 8)

throw new ArgumentOutOfRangeException(nameof(value));

\_radius = value;

font = new Font(font.FontFamily.Name, \_radius \* 0.625f);

RadiusChanged?.Invoke(null, EventArgs.Empty);

}

}

/// <summary>

/// Occurs when the Vertices radius changes

/// </summary>

public event EventHandler RadiusChanged;

#endregion

/// <summary>

/// Initializes a new instance of the GraphDrawing class

/// </summary>

/// <param name="width">Width of the drawing surface</param>

/// <param name="height">Height of the drawing surface</param>

/// <exception cref="ArgumentException"/>

public GraphDrawing(int width, int height)

{

if (width <= 0 || height <= 0)

throw new ArgumentException("The image cannot have negative dimensions");

Image = new Bitmap(width, height);

graphics = Graphics.FromImage(Image);

BackColor = Color.White;

ClearTheSurface();

}

/// <summary>

/// Cleans the drawing surface

/// </summary>

public void ClearTheSurface() => graphics.Clear(BackColor);

/// <summary>

/// Draws graph vertices

/// </summary>

/// <param name="x">X coordinate of the point where the vertex is</param>

/// <param name="y">Y coordinate of the point where the vertex is</param>

/// <param name="number">Number of the point</param>

/// <param name="pen">Pen to draw</param>

public void DrawVertex(int x, int y, int number, Pen pen = null)

{

if (pen == null) pen = verticesPen;

graphics.FillEllipse(Brushes.White, (x - R), (y - R), 2 \* R, 2 \* R);

graphics.DrawEllipse(pen, (x - R), (y - R), 2 \* R, 2 \* R);

PointF point = number >= 100

? new PointF(x - font.Size \* 2f, y - font.Size \* 1.4f)

: number >= 10

? new PointF(x - font.Size \* 1.4f, y - font.Size \* 1.4f)

: new PointF(x - font.Size \* 0.8f, y - font.Size \* 1.4f);

graphics.DrawString(number.ToString(), font, brush, point);

}

/// <summary>

/// Highlights graph vertex

/// </summary>

public void HighlightVertex(Vertex vertex) =>

graphics.DrawEllipse(highlightPen, (vertex.X - R), (vertex.Y - R), 2 \* R, 2 \* R);

/// <summary>

/// Highlights graph vertex before adding sand to it (sandpile modeling)

/// </summary>

public void HighlightVertexToAddSand(Vertex vertex) =>

graphics.DrawEllipse(highlightSandpilePen, (vertex.X - R \* 1.1f), (vertex.Y - R \* 1.1f), 2 \* R \* 1.1f, 2 \* R \* 1.1f);

/// <summary>

/// Removes highlighting from graph vertex

/// </summary>

public void UnhighlightVertex(Vertex vertex) =>

graphics.DrawEllipse(verticesPen, (vertex.X - R), (vertex.Y - R), 2 \* R, 2 \* R);

/// <summary>

/// Draws graph arc

/// </summary>

/// <param name="startVertex">Starting vertex</param>

/// <param name="endVertex">Ending vertex</param>

/// <param name="arc">Arc itself</param>

/// <param name="xOffset">X-axis image offset</param>

/// <param name="yOffset">Y-axis image offset</param>

/// <param name="sizeCoef">Resizing coefficient</param>

public void DrawArc(Vertex startVertex, Vertex endVertex, Arc arc, int xOffset = 0, int yOffset = 0, double sizeCoef = 1)

{

startVertex.X = (int)((startVertex.X + xOffset) \* sizeCoef);

startVertex.Y = (int)((startVertex.Y + yOffset) \* sizeCoef);

endVertex.X = (int)((endVertex.X + xOffset) \* sizeCoef);

endVertex.Y = (int)((endVertex.Y + yOffset) \* sizeCoef);

if (arc.StartVertex == arc.EndVertex)

throw new ArgumentException("Arc cannot be a loop");

graphics.DrawLine(arcsPen, startVertex.X, startVertex.Y, endVertex.X, endVertex.Y);

DrawVertex(startVertex.X, startVertex.Y, arc.StartVertex + 1);

DrawVertex(endVertex.X, endVertex.Y, arc.EndVertex + 1);

// Drawing the edge's direction

double[] l = { startVertex.X - endVertex.X, startVertex.Y - endVertex.Y };

double length = Math.Sqrt(l[0] \* l[0] + l[1] \* l[1]);

l[0] /= length;

l[1] /= length;

double[] w = { -l[1] \* R / 3, l[0] \* R / 3 };

double x = endVertex.X + l[0] \* 2 \* R + w[0];

double y = endVertex.Y + l[1] \* 2 \* R + w[1];

graphics.DrawLine(arcsPen, (float)x, (float)y, (float)(endVertex.X + l[0] \* R), (float)(endVertex.Y + l[1] \* R));

x -= 2 \* w[0];

y -= 2 \* w[1];

graphics.DrawLine(arcsPen, (float)x, (float)y, (float)(endVertex.X + l[0] \* R), (float)(endVertex.Y + l[1] \* R));

}

/// <summary>

/// Draws all the digraph vertices

/// </summary>

public void DrawVertices(Digraph digraph, int xOffset = 0, int yOffset = 0, double sizeCoef = 1)

{

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

DrawVertex((int)((digraph.Vertices[i].X + xOffset) \* sizeCoef), (int)((digraph.Vertices[i].Y + yOffset) \* sizeCoef), i + 1);

}

/// <summary>

/// Draws the whole digraph

/// </summary>

public void DrawTheWholeGraph(Digraph digraph, int xOffset = 0, int yOffset = 0, double sizeCoef = 1)

{

ClearTheSurface();

digraph.Arcs.ForEach(arc =>

DrawArc(digraph.Vertices[arc.StartVertex], digraph.Vertices[arc.EndVertex], arc,

xOffset, yOffset, sizeCoef));

DrawVertices(digraph, xOffset, yOffset, sizeCoef);

}

/// <summary>

/// Draws a moving dot

/// </summary>

/// <param name="p"></param>

public void DrawDot(PointF p) =>

graphics.FillEllipse(Brushes.Black, p.X - 4, p.Y - 4, 8, 8);

#region Sandpile

/// <summary>

/// Colors palette for sandpile drawing

/// </summary>

private Color[] sandpilePalette;

/// <summary>

/// Digraph incidence list

/// </summary>

private List<Arc>[] incidenceList;

/// <summary>

/// Number of dots in a vertex font

/// </summary>

private readonly Font sandpileFont = new Font("Segoe UI", 5);

/// <summary>

/// Occurs when the palette colors changes

/// </summary>

public event EventHandler SandpilePaletteChanged;

/// <summary>

/// Colors palette for sandpile drawing

/// </summary>

public Color[] SandpilePalette

{

get => sandpilePalette;

set

{

sandpilePalette = value;

SandpilePaletteChanged?.Invoke(value, new EventArgs());

}

}

/// <summary>

/// Draws the whole digraph using sandpile palette

/// </summary>

/// <param name="digraph">Digraph</param>

/// <param name="update">Is it needed to update an incidence list and colors palette

/// (true if digraph has changed since the last call, or if the method is called for the first time)</param>

/// <param name="xOffset">X-axis image offset</param>

/// <param name="yOffset">Y-axis image offset</param>

/// <param name="sizeCoef">Resizing coefficient</param>

public void DrawTheWholeGraphSandpile(Digraph digraph, bool update, int xOffset = 0, int yOffset = 0, double sizeCoef = 1)

{

if (update)

{

incidenceList = MovementModeling.GetIncidenceList(digraph);

SandpilePalette = GetSandpilePalette(incidenceList.Max(arcs => arcs.Count));

}

ClearTheSurface();

digraph.Arcs.ForEach(arc =>

DrawArc(digraph.Vertices[arc.StartVertex], digraph.Vertices[arc.EndVertex], arc,

xOffset, yOffset, sizeCoef));

DrawVerticesSandpile(digraph, xOffset, yOffset, sizeCoef);

}

/// <summary>

/// Draws all the digraph vertices in sandpile format

/// </summary>

/// <param name="digraph">Digraph</param>

/// <param name="xOffset">X-axis image offset</param>

/// <param name="yOffset">Y-axis image offset</param>

/// <param name="sizeCoef">Resizing coefficient</param>

public void DrawVerticesSandpile(Digraph digraph, int xOffset = 0, int yOffset = 0, double sizeCoef = 1)

{

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

{

DrawVertex((int)((digraph.Vertices[i].X + xOffset) \* sizeCoef),

(int)((digraph.Vertices[i].Y + yOffset) \* sizeCoef),

i + 1,

new Pen(digraph.State[i] >= incidenceList[i].Count || digraph.Stock.Contains(i)

? Color.Black

: SandpilePalette[digraph.State[i]], 4f));

if (digraph.Stock.Contains(i))

graphics.FillEllipse(Brushes.Black,

(int)((digraph.Vertices[i].X + xOffset) \* sizeCoef) - R,

(int)((digraph.Vertices[i].Y + yOffset) \* sizeCoef) - R,

2 \* R, 2 \* R);

else

graphics.DrawString($"({digraph.State[i]})", sandpileFont, brush,

(int)((digraph.Vertices[i].X + xOffset) \* sizeCoef) + R,

(int)((digraph.Vertices[i].Y + yOffset) \* sizeCoef) - R - 10f);

}

}

/// <summary>

/// Returns a gradient colors palette

/// </summary>

/// <param name="start">Start color</param>

/// <param name="end">End color</param>

/// <param name="steps">Number of colors</param>

private static Color[] GetGradientColors(Color start, Color end, int steps)

{

Color[] colors = new Color[steps];

colors[steps - 1] = end;

colors[0] = start;

var aStep = (end.A - start.A) \* 1.0 / steps;

var rStep = (end.R - start.R) \* 1.0 / steps;

var gStep = (end.G - start.G) \* 1.0 / steps;

var bStep = (end.B - start.B) \* 1.0 / steps;

for (int i = 1; i < steps - 1; i++)

{

var a = start.A + (aStep \* i);

var r = start.R + (rStep \* i);

var g = start.G + (gStep \* i);

var b = start.B + (bStep \* i);

colors[i] = Color.FromArgb((byte)a, (byte)r, (byte)g, (byte)b);

}

return colors;

}

/// <summary>

/// Returns colors palette for sandpile drawing

/// </summary>

/// <param name="paletteSize">Number of colors in palette</param>

private Color[] GetSandpilePalette(int paletteSize)

{

List<Color> colors = new List<Color>(paletteSize);

int steps = paletteSize > 5 ? 5 : paletteSize;

int stepLength = paletteSize / steps;

int currStep = stepLength;

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

{

switch (i)

{

case 0:

colors.AddRange(GetGradientColors(Color.Red, Color.Yellow,

steps \* stepLength < paletteSize

? stepLength + 1 : stepLength));

break;

case 1:

if (stepLength > 1 || steps \* stepLength < paletteSize && stepLength > 0)

{

colors.Remove(Color.Yellow);

currStep++;

}

colors.AddRange(GetGradientColors(Color.Yellow, Color.Green,

steps \* stepLength < paletteSize && paletteSize % steps \* stepLength >= 2

? currStep + 1 : currStep));

currStep = stepLength;

break;

case 2:

if (stepLength > 1 || steps \* stepLength < paletteSize && steps \* stepLength % paletteSize >= 2 && stepLength > 0)

{

colors.Remove(Color.Green);

currStep++;

}

colors.AddRange(GetGradientColors(Color.Green, Color.LightSkyBlue,

steps \* stepLength < paletteSize && paletteSize % steps \* stepLength >= 3

? currStep + 1 : currStep));

currStep = stepLength;

break;

case 3:

if (stepLength > 1 || steps \* stepLength < paletteSize && steps \* stepLength % paletteSize >= 3 && stepLength > 0)

{

colors.Remove(Color.LightSkyBlue);

currStep++;

}

colors.AddRange(GetGradientColors(Color.LightSkyBlue, Color.Blue,

steps \* stepLength < paletteSize && paletteSize % steps \* stepLength >= 4

? currStep + 1 : currStep));

currStep = stepLength;

break;

case 4:

if (stepLength > 1 || steps \* stepLength < paletteSize && steps \* stepLength % paletteSize >= 4 && stepLength > 0)

{

colors.Remove(Color.Blue);

currStep++;

}

colors.AddRange(GetGradientColors(Color.Blue, Color.DeepPink, currStep));

currStep = stepLength;

break;

}

}

return colors.ToArray();

}

#endregion

/// <summary>

/// Drawing surface size

/// </summary>

public Size Size

{

get => Image.Size;

set

{

if (value.Height == 0 || value.Width == 0) return;

Image = new Bitmap(value.Width, value.Height);

graphics = Graphics.FromImage(Image);

}

}

public void Dispose()

{

Image.Dispose();

graphics.Dispose();

verticesPen.Dispose();

arcsPen.Dispose();

highlightPen.Dispose();

highlightSandpilePen.Dispose();

font.Dispose();

sandpileFont.Dispose();

brush.Dispose();

}

}

}

* + 1. **MathParser.cs**

using System;

using System.Collections.Generic;

using System.Globalization;

using System.Text;

namespace MathParserTK

{

public class MathParser

{

#region Fields

#region Markers (each marker should have length equals to 1)

private const string NumberMaker = "#";

private const string OperatorMarker = "$";

private const string FunctionMarker = "@";

#endregion

#region Internal tokens

private const string Plus = OperatorMarker + "+";

private const string UnPlus = OperatorMarker + "un+";

private const string Minus = OperatorMarker + "-";

private const string UnMinus = OperatorMarker + "un-";

private const string Multiply = OperatorMarker + "\*";

private const string Divide = OperatorMarker + "/";

private const string Degree = OperatorMarker + "^";

private const string LeftParent = OperatorMarker + "(";

private const string RightParent = OperatorMarker + ")";

private const string Sqrt = FunctionMarker + "sqrt";

private const string Sin = FunctionMarker + "sin";

private const string Cos = FunctionMarker + "cos";

private const string Tg = FunctionMarker + "tg";

private const string Ctg = FunctionMarker + "ctg";

private const string Sh = FunctionMarker + "sh";

private const string Ch = FunctionMarker + "ch";

private const string Th = FunctionMarker + "th";

private const string Log = FunctionMarker + "log";

private const string Ln = FunctionMarker + "ln";

private const string Exp = FunctionMarker + "exp";

private const string Abs = FunctionMarker + "abs";

#endregion

#region Dictionaries (containts supported input tokens, exclude number)

// Key -> supported input token, Value -> internal token or number

/// <summary>

/// Contains supported operators

/// </summary>

private readonly Dictionary<string, string> supportedOperators =

new Dictionary<string, string>

{

{ "+", Plus },

{ "-", Minus },

{ "\*", Multiply },

{ "/", Divide },

{ "^", Degree },

{ "(", LeftParent },

{ ")", RightParent }

};

/// <summary>

/// Contains supported functions

/// </summary>

private readonly Dictionary<string, string> supportedFunctions =

new Dictionary<string, string>

{

{ "sqrt", Sqrt },

{ "√", Sqrt },

{ "sin", Sin },

{ "cos", Cos },

{ "tg", Tg },

{ "ctg", Ctg },

{ "sh", Sh },

{ "ch", Ch },

{ "th", Th },

{ "log", Log },

{ "exp", Exp },

{ "abs", Abs }

};

private readonly Dictionary<string, string> supportedConstants =

new Dictionary<string, string>

{

{"pi", NumberMaker + Math.PI.ToString() },

{"e", NumberMaker + Math.E.ToString() }

};

#endregion

#endregion

private readonly char decimalSeparator;

private bool isRadians;

#region Constructors

/// <summary>

/// Initialize new instance of MathParser

/// (symbol of decimal separator is read

/// from regional settings in system)

/// </summary>

public MathParser()

{

try

{

decimalSeparator = Char.Parse(System.Globalization.CultureInfo.CurrentCulture.NumberFormat.NumberDecimalSeparator);

}

catch (FormatException ex)

{

throw new FormatException("Error: can't read char decimal separator from system, check your regional settings.", ex);

}

}

/// <summary>

/// Initialize new instance of MathParser

/// </summary>

/// <param name="decimalSeparator">Set decimal separator</param>

public MathParser(char decimalSeparator)

{

this.decimalSeparator = decimalSeparator;

}

#endregion

/// <summary>

/// Produce result of the given math expression

/// </summary>

/// <param name="expression">Math expression (infix/standard notation)</param>

/// <returns>Result</returns>

public double Parse(string expression, bool isRadians = true)

{

this.isRadians = isRadians;

try

{

return Calculate(ConvertToRPN(FormatString(expression)));

}

catch (DivideByZeroException e)

{

throw e;

}

catch (FormatException e)

{

throw e;

}

catch (InvalidOperationException e)

{

throw e;

}

catch (ArgumentOutOfRangeException e)

{

throw e;

}

catch (ArgumentException e)

{

throw e;

}

catch (Exception e)

{

throw e;

}

}

/// <summary>

/// Produce formatted string by the given string

/// </summary>

/// <param name="expression">Unformatted math expression</param>

/// <returns>Formatted math expression</returns>

private string FormatString(string expression)

{

if (string.IsNullOrEmpty(expression))

{

throw new ArgumentNullException("Expression is null or empty");

}

StringBuilder formattedString = new StringBuilder();

int balanceOfParenth = 0; // Check number of parenthesis

// Format string in one iteration and check number of parenthesis

// (this function do 2 tasks because performance priority)

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

{

char ch = expression[i];

if (ch == '(')

{

balanceOfParenth++;

}

else if (ch == ')')

{

balanceOfParenth--;

}

if (Char.IsWhiteSpace(ch))

{

continue;

}

else if (Char.IsUpper(ch))

{

formattedString.Append(Char.ToLower(ch));

}

else

{

formattedString.Append(ch);

}

}

if (balanceOfParenth != 0)

{

throw new FormatException("Number of left and right parenthesis is not equal");

}

return formattedString.ToString();

}

#region Convert to Reverse-Polish Notation

/// <summary>

/// Produce math expression in reverse polish notation

/// by the given string

/// </summary>

/// <param name="expression">Math expression in infix notation</param>

/// <returns>Math expression in postfix notation (RPN)</returns>

private string ConvertToRPN(string expression)

{

int pos = 0; // Current position of lexical analysis

StringBuilder outputString = new StringBuilder();

Stack<string> stack = new Stack<string>();

// While there is unhandled char in expression

while (pos < expression.Length)

{

string token = LexicalAnalysisInfixNotation(expression, ref pos);

outputString = SyntaxAnalysisInfixNotation(token, outputString, stack);

}

// Pop all elements from stack to output string

while (stack.Count > 0)

{

// There should be only operators

if (stack.Peek()[0] == OperatorMarker[0])

{

outputString.Append(stack.Pop());

}

else

{

throw new FormatException("Format exception,"

+ " there is function without parenthesis");

}

}

return outputString.ToString();

}

/// <summary>

/// Produce token by the given math expression

/// </summary>

/// <param name="expression">Math expression in infix notation</param>

/// <param name="pos">Current position in string for lexical analysis</param>

/// <returns>Token</returns>

private string LexicalAnalysisInfixNotation(string expression, ref int pos)

{

// Receive first char

StringBuilder token = new StringBuilder();

token.Append(expression[pos]);

// If it is a operator

if (supportedOperators.ContainsKey(token.ToString()))

{

// Determine it is unary or binary operator

bool isUnary = pos == 0 || expression[pos - 1] == '(';

pos++;

switch (token.ToString())

{

case "+":

return isUnary ? UnPlus : Plus;

case "-":

return isUnary ? UnMinus : Minus;

default:

return supportedOperators[token.ToString()];

}

}

else if (Char.IsLetter(token[0])

|| supportedFunctions.ContainsKey(token.ToString())

|| supportedConstants.ContainsKey(token.ToString()))

{

// Read function or constant name

while (++pos < expression.Length

&& Char.IsLetter(expression[pos]))

{

token.Append(expression[pos]);

}

if (supportedFunctions.ContainsKey(token.ToString()))

{

return supportedFunctions[token.ToString()];

}

else if (supportedConstants.ContainsKey(token.ToString()))

{

return supportedConstants[token.ToString()];

}

else

{

throw new ArgumentException("Unknown token");

}

}

else if (Char.IsDigit(token[0]) || token[0] == decimalSeparator)

{

// Read number

// Read the whole part of number

if (Char.IsDigit(token[0]))

{

while (++pos < expression.Length

&& Char.IsDigit(expression[pos]))

{

token.Append(expression[pos]);

}

}

else

{

// Because system decimal separator

// will be added below

token.Clear();

}

// Read the fractional part of number

if (pos < expression.Length

&& expression[pos] == decimalSeparator)

{

// Add current system specific decimal separator

token.Append(CultureInfo.CurrentCulture

.NumberFormat.NumberDecimalSeparator);

while (++pos < expression.Length

&& Char.IsDigit(expression[pos]))

{

token.Append(expression[pos]);

}

}

// Read scientific notation (suffix)

if (pos + 1 < expression.Length && expression[pos] == 'e'

&& (Char.IsDigit(expression[pos + 1])

|| (pos + 2 < expression.Length

&& (expression[pos + 1] == '+'

|| expression[pos + 1] == '-')

&& Char.IsDigit(expression[pos + 2]))))

{

token.Append(expression[pos++]); // e

if (expression[pos] == '+' || expression[pos] == '-')

token.Append(expression[pos++]); // sign

while (pos < expression.Length

&& Char.IsDigit(expression[pos]))

{

token.Append(expression[pos++]); // power

}

// Convert number from scientific notation to decimal notation

return NumberMaker + Convert.ToDouble(token.ToString());

}

return NumberMaker + token.ToString();

}

else

{

throw new ArgumentException("Unknown token in expression");

}

}

/// <summary>

/// Syntax analysis of infix notation

/// </summary>

/// <param name="token">Token</param>

/// <param name="outputString">Output string (math expression in RPN)</param>

/// <param name="stack">Stack which contains operators (or functions)</param>

/// <returns>Output string (math expression in RPN)</returns>

private StringBuilder SyntaxAnalysisInfixNotation(string token, StringBuilder outputString, Stack<string> stack)

{

// If it's a number just put to string

if (token[0] == NumberMaker[0])

{

outputString.Append(token);

}

else if (token[0] == FunctionMarker[0])

{

// if it's a function push to stack

stack.Push(token);

}

else if (token == LeftParent)

{

// If its '(' push to stack

stack.Push(token);

}

else if (token == RightParent)

{

// If its ')' pop elements from stack to output string

// until find the ')'

string elem;

while ((elem = stack.Pop()) != LeftParent)

{

outputString.Append(elem);

}

// if after this a function is in the peek of stack then put it to string

if (stack.Count > 0 &&

stack.Peek()[0] == FunctionMarker[0])

{

outputString.Append(stack.Pop());

}

}

else

{

// While priority of elements at peek of stack >= (>) token's priority

// put these elements to output string

while (stack.Count > 0 &&

Priority(token, stack.Peek()))

{

outputString.Append(stack.Pop());

}

stack.Push(token);

}

return outputString;

}

/// <summary>

/// Is priority of token less (or equal) to priority of p

/// </summary>

private bool Priority(string token, string p)

{

return IsRightAssociated(token) ?

GetPriority(token) < GetPriority(p) :

GetPriority(token) <= GetPriority(p);

}

/// <summary>

/// Is right associated operator

/// </summary>

private bool IsRightAssociated(string token)

{

return token == Degree;

}

/// <summary>

/// Get priority of operator

/// </summary>

private int GetPriority(string token)

{

switch (token)

{

case LeftParent:

return 0;

case Plus:

case Minus:

return 2;

case UnPlus:

case UnMinus:

return 6;

case Multiply:

case Divide:

return 4;

case Degree:

case Sqrt:

return 8;

case Sin:

case Cos:

case Tg:

case Ctg:

case Sh:

case Ch:

case Th:

case Log:

case Ln:

case Exp:

case Abs:

return 10;

default:

throw new ArgumentException("Unknown operator");

}

}

#endregion

#region Calculate expression in RPN

/// <summary>

/// Calculate expression in reverse-polish notation

/// </summary>

/// <param name="expression">Math expression in reverse-polish notation</param>

/// <returns>Result</returns>

private double Calculate(string expression)

{

int pos = 0; // Current position of lexical analysis

var stack = new Stack<double>(); // Contains operands

// Analyse entire expression

while (pos < expression.Length)

{

string token = LexicalAnalysisRPN(expression, ref pos);

stack = SyntaxAnalysisRPN(stack, token);

}

// At end of analysis in stack should be only one operand (result)

if (stack.Count > 1)

{

throw new ArgumentException("Excess operand");

}

return stack.Pop();

}

/// <summary>

/// Produce token by the given math expression

/// </summary>

/// <param name="expression">Math expression in reverse-polish notation</param>

/// <param name="pos">Current position of lexical analysis</param>

/// <returns>Token</returns>

private string LexicalAnalysisRPN(string expression, ref int pos)

{

StringBuilder token = new StringBuilder();

// Read token from marker to next marker

token.Append(expression[pos++]);

while (pos < expression.Length && expression[pos] != NumberMaker[0]

&& expression[pos] != OperatorMarker[0]

&& expression[pos] != FunctionMarker[0])

{

token.Append(expression[pos++]);

}

return token.ToString();

}

/// <summary>

/// Syntax analysis of reverse-polish notation

/// </summary>

/// <param name="stack">Stack which contains operands</param>

/// <param name="token">Token</param>

/// <returns>Stack which contains operands</returns>

private Stack<double> SyntaxAnalysisRPN(Stack<double> stack, string token)

{

// if it's operand then just push it to stack

if (token[0] == NumberMaker[0])

{

stack.Push(double.Parse(token.Remove(0, 1)));

}

// Otherwise apply operator or function to elements in stack

else if (NumberOfArguments(token) == 1)

{

double arg = stack.Pop();

double rst;

switch (token)

{

case UnPlus:

rst = arg;

break;

case UnMinus:

rst = -arg;

break;

case Sqrt:

rst = Math.Sqrt(arg);

break;

case Sin:

rst = ApplyTrigFunction(Math.Sin, arg);

break;

case Cos:

rst = ApplyTrigFunction(Math.Cos, arg);

break;

case Tg:

rst = ApplyTrigFunction(Math.Tan, arg);

break;

case Ctg:

rst = 1 / ApplyTrigFunction(Math.Tan, arg);

break;

case Sh:

rst = Math.Sinh(arg);

break;

case Ch:

rst =

rst = Math.Cosh(arg);

break;

case Th:

rst = Math.Tanh(arg);

break;

case Ln:

rst = Math.Log(arg);

break;

case Exp:

rst = Math.Exp(arg);

break;

case Abs:

rst = Math.Abs(arg);

break;

default:

throw new ArgumentException("Unknown operator");

}

stack.Push(rst);

}

else

{

// otherwise operator's number of arguments equals to 2

double arg2 = stack.Pop();

double arg1 = stack.Pop();

double rst;

switch (token)

{

case Plus:

rst = arg1 + arg2;

break;

case Minus:

rst = arg1 - arg2;

break;

case Multiply:

rst = arg1 \* arg2;

break;

case Divide:

if (arg2 == 0)

{

throw new DivideByZeroException("Second argument is zero");

}

rst = arg1 / arg2;

break;

case Degree:

rst = Math.Pow(arg1, arg2);

break;

case Log:

rst = Math.Log(arg2, arg1);

break;

default:

throw new ArgumentException("Unknown operator");

}

stack.Push(rst);

}

return stack;

}

/// <summary>

/// Apply trigonometric function

/// </summary>

/// <param name="func">Trigonometric function</param>

/// <param name="arg">Argument</param>

/// <returns>Result of function</returns>

private double ApplyTrigFunction(Func<double, double> func, double arg)

{

if (!isRadians)

{

arg = arg \* Math.PI / 180; // Convert value to degree

}

return func(arg);

}

/// <summary>

/// Produce number of arguments for the given operator

/// </summary>

private int NumberOfArguments(string token)

{

switch (token)

{

case UnPlus:

case UnMinus:

case Sqrt:

case Tg:

case Sh:

case Ch:

case Th:

case Ln:

case Ctg:

case Sin:

case Cos:

case Exp:

case Abs:

return 1;

case Plus:

case Minus:

case Multiply:

case Divide:

case Degree:

case Log:

return 2;

default:

throw new ArgumentException("Unknown operator");

}

}

#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 |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |