Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

Лабораторна робота №3

з дисципліни
«Алгоритми і структури даних»

Виконав:

студент групи ІМ-43
Балалаєв Максим Юрійович
номер варіанту: 3

Перевірив:

Сергієнко А. М.

Київ 2025

Завдання

1. Представити у програмі напрямлений і ненапрямлений графи з заданими параметрами:

- кількість вершин n;
- розміщення вершин;
- матриця суміжності A.

2. Створити програму для формування зображення напрямленого і ненапрямленого графів у графічному вікні.

Варіант №3

Номер варіанту: 4303

Кількість вершин n = 10

Розміщення вершин - квадратом (прямокутником), бо n4 = 3

Коефіцієнт k = 0,735

Тексти програм

src/Forms/DirectedGraph/DirectedForm.cs

```
namespace lab3;

public partial class DirectedForm : Form
{
    private readonly NodeFactory _nodeFactory;
    private readonly LinkFactory _linkFactory;

    private Graphics? _graphics;

    public DirectedForm(NodeFactory nodeFactory, LinkFactory linkFactory)
    {
        _nodeFactory = nodeFactory;
        _linkFactory = linkFactory;

        InitializeComponent();
    }

    private const int _textOffset = 10;

    protected override void OnPaint(PaintEventArgs e)
    {
        base.OnPaint(e);
        _graphics = e.Graphics;
```

```
    foreach (var link in _linkFactory.Links)
    {
        Draw(link);
    }
    foreach (var node in _nodeFactory.Nodes)
    {
        Draw(node);
    }
}

private void Draw(Node node)
{
    if (_graphics == null) return;

    _graphics.FillEllipse(
        new SolidBrush(Color.LightGray),
        node.Point.X - Node.Radius,
        node.Point.Y - Node.Radius,
        Node.Radius * 2,
        Node.Radius * 2
    );

    _graphics.DrawString(
        node.Id.ToString(),
        Font,
        Brushes.Black,
        node.Point.X - _textOffset,
        node.Point.Y - _textOffset
    );
}

private void Draw(Link link)
{
    if (_graphics == null) return;

    switch (link.Type)
    {
        case LinkType.Normal:
            {
                var (start, end) = link.GetLineCoords();

                if (!link.HasInvertion)
                {
                    DrawArrow(start, end);
                }
                else
                {
                    DrawLine(start, link.PolygonalLinkVertice);
                    DrawArrow(link.PolygonalLinkVertice, end);
                }

                break;
            }
```

```
                case LinkType.VisibilityObstructed:
                    {
                        var (start, end) = link.GetLineCoords();
                        DrawLine(start, link.PolygonalLinkVertice);
                        DrawArrow(link.PolygonalLinkVertice, end);
                        break;
                    }

                case LinkType.SelfPointing:
                    DrawLine(link.From.Point, link.SelfLinkVertices[0]);
                    DrawLine(link.SelfLinkVertices[0], link.SelfLinkVertices[1]);
                    DrawArrow(link.SelfLinkVertices[1], link.SelfLinkVertices[2]);

                    break;
            }

    }

    private void DrawArrow(PointF start, PointF end)
    {
        if (_graphics == null) return;

        Pen pen = Pens.Black;
        _graphics.DrawLine(pen, start, end);

        // Arrowhead
        const float arrowSize = 10f;
        double angle = Math.Atan2(end.Y - start.Y, end.X - start.X);
        PointF arrow1 = new PointF(
            end.X - arrowSize * (float)Math.Cos(angle - Math.PI / 6),
            end.Y - arrowSize * (float)Math.Sin(angle - Math.PI / 6));
        PointF arrow2 = new PointF(
            end.X - arrowSize * (float)Math.Cos(angle + Math.PI / 6),
            end.Y - arrowSize * (float)Math.Sin(angle + Math.PI / 6));

        _graphics.DrawLine(pen, end, arrow1);
        _graphics.DrawLine(pen, end, arrow2);
    }

    private void DrawLine(PointF start, Point end)
    {
        if (_graphics == null) return;
        _graphics.DrawLine(Pens.Black, start, end);
    }
}
```

src/Forms/UndirectedGraph/UndirectedForm.cs

```
namespace lab3;

public partial class UndirectedForm : Form
```

```
{
    private readonly NodeFactory _nodeFactory;
    private readonly LinkFactory _linkFactory;

    private Graphics? _graphics;

    public UndirectedForm(NodeFactory nodeFactory, LinkFactory linkFactory)
    {
        _nodeFactory = nodeFactory;
        _linkFactory = linkFactory;

        InitializeComponent();
    }

    private const int _textOffset = 10;

    protected override void OnPaint(PaintEventArgs e)
    {
        base.OnPaint(e);
        _graphics = e.Graphics;

        foreach (var link in _linkFactory.Links)
        {
            Draw(link);
        }
        foreach (var node in _nodeFactory.Nodes)
        {
            Draw(node);
        }

    }

    private void Draw(Node node)
    {
        if (_graphics == null) return;

        _graphics.FillEllipse(
            new SolidBrush(Color.LightGray),
            node.Point.X - Node.Radius,
            node.Point.Y - Node.Radius,
            Node.Radius * 2,
            Node.Radius * 2
        );

        _graphics.DrawString(
            node.Id.ToString(),
            Font,
            Brushes.Black,
            node.Point.X - _textOffset,
            node.Point.Y - _textOffset
        );
    }
```

```
    private void Draw(Link link)
    {
        if (_graphics == null) return;

        switch (link.Type)
        {
            case LinkType.Normal:
                {
                    DrawLine(link.From.Point, link.To.Point);
                    break;
                }

            case LinkType.VisibilityObstructed:
                {
                    DrawLine(link.From.Point, link.PolygonalLinkVertice);
                    DrawLine(link.PolygonalLinkVertice, link.To.Point);
                    break;
                }

            case LinkType.SelfPointing:
                DrawLine(link.From.Point, link.SelfLinkVertices[0]);
                DrawLine(link.SelfLinkVertices[0], link.SelfLinkVertices[1]);
                DrawLine(link.SelfLinkVertices[1], link.From.Point);

                break;
        }

    }

    private void DrawLine(Point start, Point end)
    {
        if (_graphics == null) return;
        _graphics.DrawLine(Pens.Black, start, end);
    }
}
```

src/Service/Generator.cs

```
namespace lab3;

public class Generator
{
    private readonly string _code;
    private uint _n;
    private readonly Random _random;
    private float _k;

    public Generator(string code)
    {
        _code = code;
        Console.WriteLine($"Номер варіанту: {_code}");
```

```
            ParseFromCode();

            _random = new(int.Parse(_code));
        }

        private uint GetN(int position) => uint.Parse(_code[position - 1].ToString());

        private void ParseFromCode()
        {
            _n = GetN(3) + 10;
            Console.WriteLine($"Кількість вершин n = {_n}");

            Console.WriteLine($"Розміщення вершин - квадратом (прямокутником), бо n4 =
{GetN(4)}");

            _k = 1 - GetN(3) * 0.02f - GetN(4) * 0.005f - 0.25f;
            Console.WriteLine($"Коефіцієнт k = {_k}");
        }

        public int[,] GenerateMatrix()
        {
            var matrix = new int[_n, _n];

            for (var i = 0; i < _n; i++)
            {
                for (var j = 0; j < _n; j++)
                {
                    double value = _random.NextDouble() * 2;
                    value *= _k;
                    value = value < 1 ? 0 : 1;

                    matrix[i, j] = (int)value;
                }
            }

            return matrix;
        }

        public int[,] MakeMatrixUndirected(int[,] matrix)
        {
            for (var i = 0; i < _n; i++)
            {
                for (var j = i + 1; j < _n; j++)
                {
                    matrix[j, i] = matrix[i, j];
                }
            }

            return matrix;
        }

        public static void Await()
        {
```

```
        Console.Write($"Натисніть Enter, щоб продовжити...");
        Console.ReadLine();
    }

    public static void PrintMatrix(int[,] matrix)
    {
        for (var i = 0; i < matrix.GetLength(0); i++)
        {
            for (var j = 0; j < matrix.GetLength(1); j++)
            {
                Console.Write($"{matrix[i, j]} ");
            }

            Console.WriteLine();
        }
    }
}
```

src/Service/LinkFactory.cs

```
using System.Windows.Forms.VisualStyles;

namespace lab3;

public class LinkFactory(NodeFactory nodeFactory, bool undirected = false)
{
    private readonly NodeFactory _nodeFactory = nodeFactory;
    private readonly List<Link> _links = [];
    public List<Link> Links => _links;
    private readonly bool _undirected = undirected;

    public void CreateAll(int[,] matrix)
    {
        for (int i = 0; i < matrix.GetLength(0); i++)
        {
            for (int j = 0; j < matrix.GetLength(1); j++)
            {
                if (matrix[i, j] == 1)
                {
                    bool hasInvertion = false;
                    if (matrix[j, i] == 1)
                    {
                        if (!_undirected)
                        {
                            matrix[j, i] = 2;
                            hasInvertion = true;
                        }

                    }
                    _links.Add(new Link(_nodeFactory.Nodes[i],
_nodeFactory.Nodes[j], hasInvertion));
                }
```

```
                else if (matrix[i, j] == 2)
                {
                    _links.Add(new Link(_nodeFactory.Nodes[i],
_nodeFactory.Nodes[j], false));
                }
            }
        }
    }

    public static int RandomizeOffset() => new Random().Next(40, 80);
}

public class Link
{
    public Node From { get; }
    public Node To { get; }
    public LinkType Type { get; }
    public bool HasInvertion { get; }
    private readonly int _offset = 40;

    public Point PolygonalLinkVertice;
    public Point[] SelfLinkVertices = new Point[3];

    public Link(Node from, Node to, bool hasInvertion)
    {
        From = from;
        To = to;
        HasInvertion = hasInvertion;

        if (hasInvertion)
            _offset = -_offset;

        if (from == to)
        {
            Type = LinkType.SelfPointing;
            SelfLinkVertices = [
                new Point(from.Point.X, from.Point.Y - Node.Radius * 2),
                new Point(from.Point.X - Node.Radius * 2, from.Point.Y),
                new Point(from.Point.X - Node.Radius, from.Point.Y),
            ];
        }
        else if (
            from.Point.X == to.Point.X &&
            MathF.Abs(from.Point.Y - to.Point.Y) != NodeFactory.Gap
        )
        {
            Type = LinkType.VisibilityObstructed;
            PolygonalLinkVertice = new(
                from.Point.X + (
                    from.Outer == Direction.Right ? +_offset : -_offset
                ),
                (from.Point.Y + to.Point.Y) / 2
            );
```

```
        }

        else if (
            from.Point.Y == to.Point.Y &&
            MathF.Abs(from.Point.X - to.Point.X) != NodeFactory.Gap
        )
        {
            Type = LinkType.VisibilityObstructed;
            PolygonalLinkVertice = new(
                (from.Point.X + to.Point.X) / 2,
                from.Point.Y + (
                    from.Outer == Direction.Down ? +_offset : -_offset
                )
            );
        }

        else
        {
            Type = LinkType.Normal;
            if (hasInvertion) PolygonalLinkVertice = new Point(
                (from.Point.X + to.Point.X) / 2 + _offset,
                (from.Point.Y + to.Point.Y) / 2 + _offset
            );
        }

    }
    public static bool IsRectangleSide(Link link) => link.From.Outer ==
link.To.Outer;
    public (PointF, PointF) GetLineCoords()
    {
        float dx = To.Point.X - From.Point.X;
        float dy = To.Point.Y - From.Point.Y;
        float dist = (float)Math.Sqrt(dx * dx + dy * dy);

        float ux = dx / dist;
        float uy = dy / dist;

        PointF start = new(From.Point.X + ux * Node.Radius, From.Point.Y + uy *
Node.Radius);
        PointF end = new(To.Point.X - ux * Node.Radius, To.Point.Y - uy *
Node.Radius);

        return (start, end);
    }

    public override string ToString() => $"{From.Id} -> {To.Id}, HasInvertion:
{HasInvertion}";

}

public enum LinkType
{
    Normal,
```

```
    SelfPointing,
    VisibilityObstructed
}
```

src/Service/NodeFactory.cs

```
namespace lab3;

public class NodeFactory(int n)
{
    private readonly int _n = n;
    private readonly List<Node> _nodes = [];
    public List<Node> Nodes => _nodes;

    private const int _startX = 100, _startY = 100;
    public const int Gap = 200;

    private Direction _currentDirection = Direction.Right;
    private Direction GetNext() => _currentDirection switch
    {
        Direction.Right => Direction.Down,
        Direction.Down => Direction.Left,
        Direction.Left => Direction.Up,
        Direction.Up => Direction.Right,
        _ => Direction.Right,
    };

    public static Direction GetPrevious(Direction direction) => direction switch
    {
        Direction.Right => Direction.Up,
        Direction.Left => Direction.Down,
        Direction.Up => Direction.Left,
        Direction.Down => Direction.Right,
        _ => Direction.Right
    };

    public static Direction GetOpposite(Direction direction) => direction switch
    {
        Direction.Right => Direction.Left,
        Direction.Left => Direction.Right,
        Direction.Up => Direction.Down,
        Direction.Down => Direction.Up,
        _ => Direction.Right
    };

    public void CreateAll()
    {
        for (int i = 0; i < _n; i++)
        {
            int x, y;
            Direction outer;
```

```
            if (_nodes.Count == 0)
            {
                x = _startX;
                y = _startY;
                outer = Direction.Up;
            }


            else
            {
                x = _nodes[^1].Point.X;
                y = _nodes[^1].Point.Y;

                if (i == (int)_currentDirection)
                    _currentDirection = GetNext();

                switch (_currentDirection)
                {
                    case Direction.Right:
                        x += Gap;
                        break;
                    case Direction.Down:
                        y += Gap;
                        break;
                    case Direction.Left:
                        x -= Gap;
                        break;
                    case Direction.Up:
                        y -= Gap;
                        break;
                }

                outer = GetPrevious(_currentDirection);
            }

            _nodes.Add(new Node(i, x, y, outer));
        }
    }
}

public class Node(int id, int x, int y, Direction outer)
{
    public int Id { get; } = id;
    public Point Point { get; } = new Point(x, y);
    public Direction Outer { get; } = outer;
    public const int Radius = 25;

    public override string ToString() => $"{Id}: ({Point.X}, {Point.Y})";
}

public enum Direction
{
```

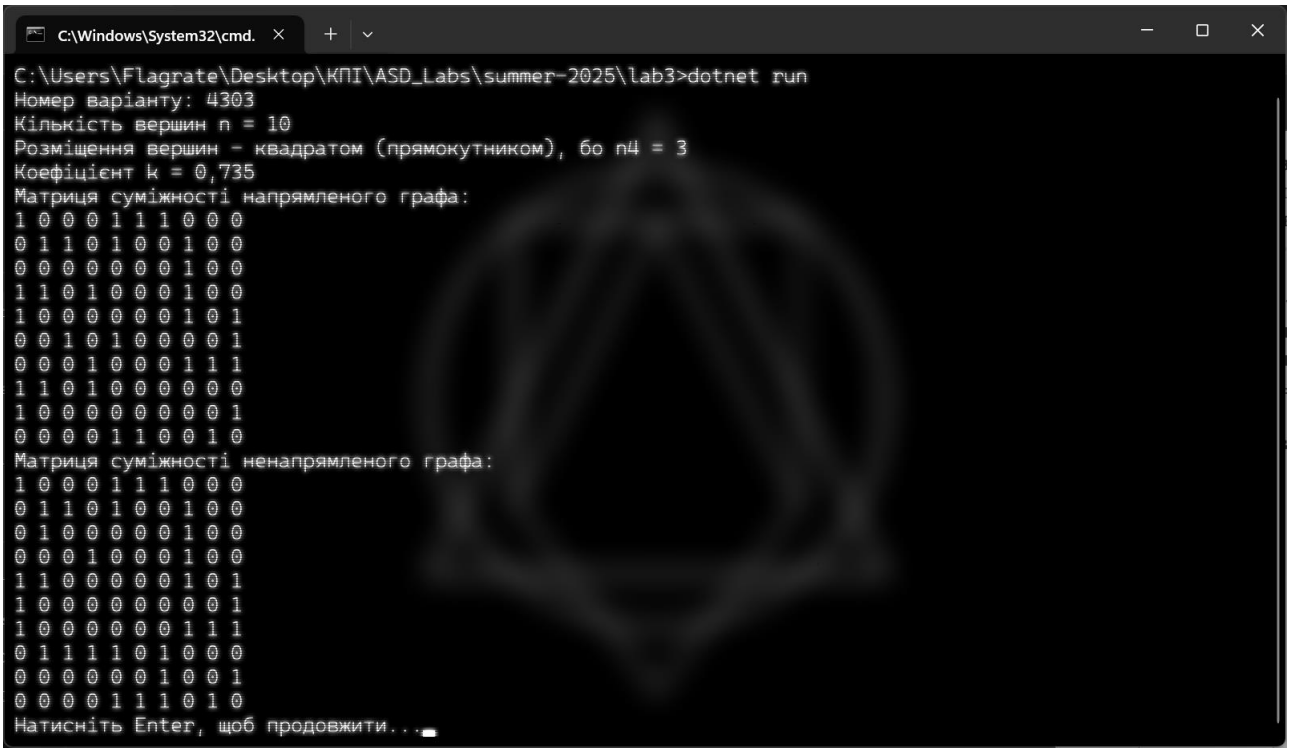
```
    Right = 4,
    Down = 6,
    Left = 9,
    Up = 1
}
```

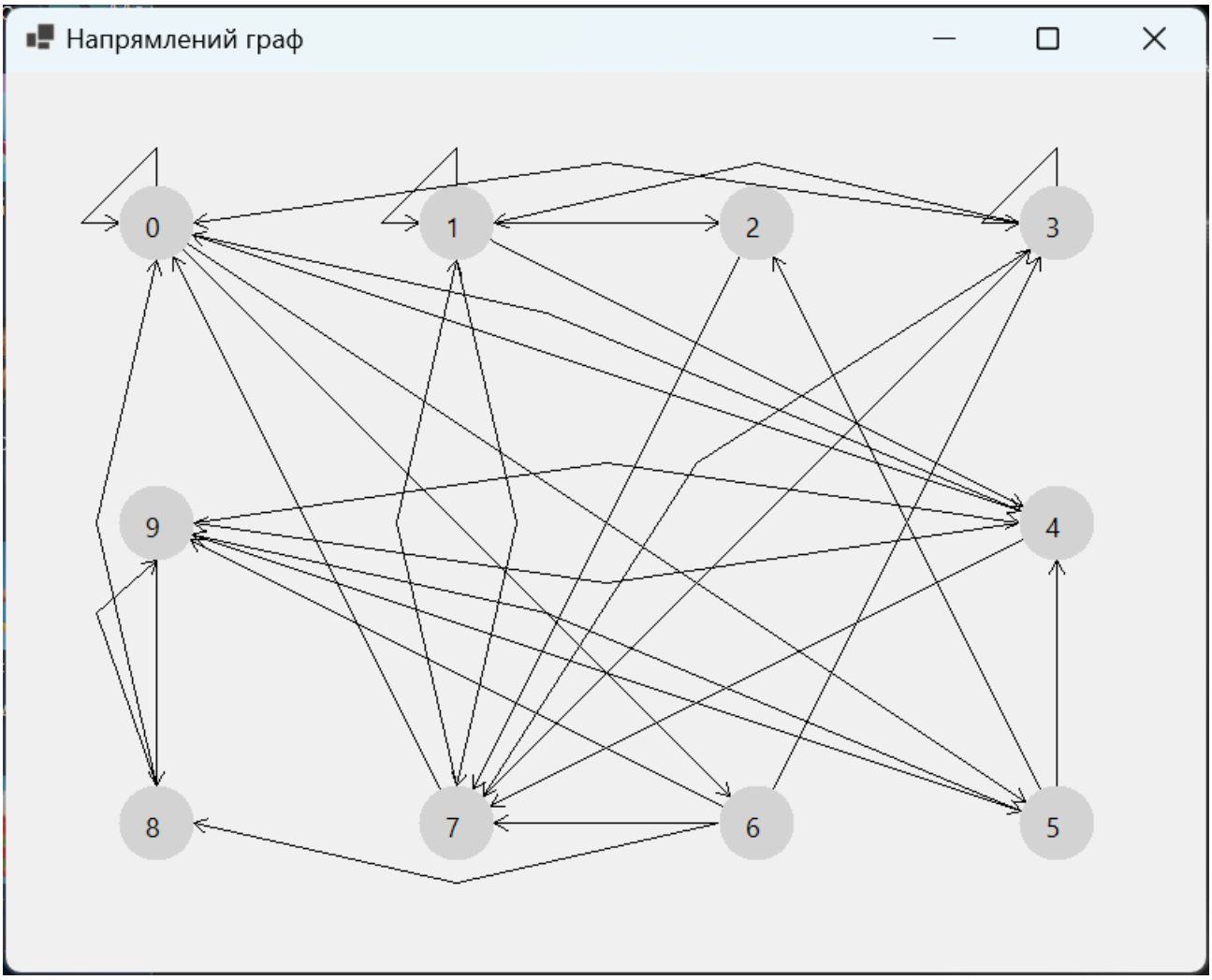
Program.cs (точка входу)

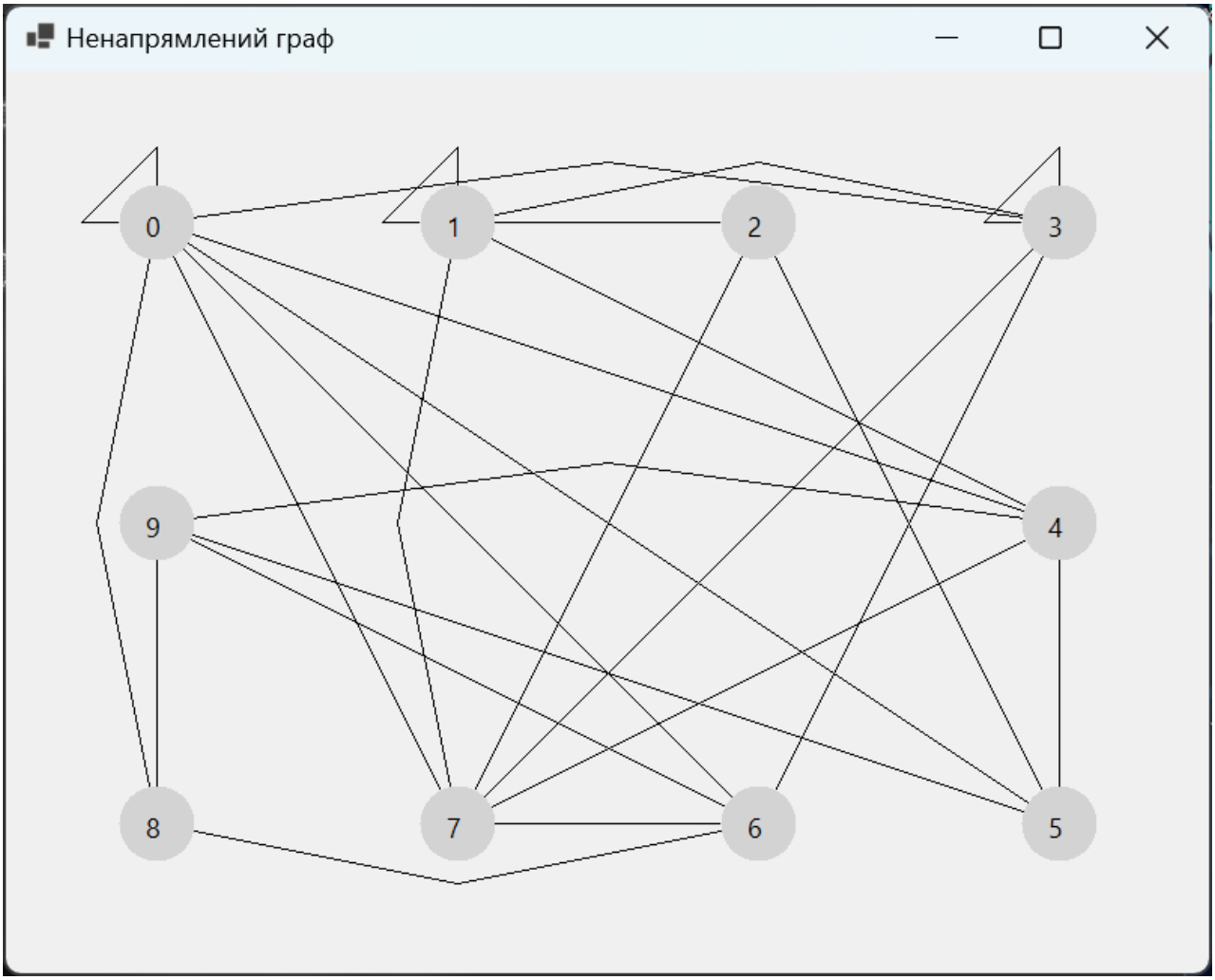
```
namespace lab3;

static class Program
{
    public static void Main()
    {
        Generator generator = new("4303");
        int[,] matrix = generator.GenerateMatrix();

        System.Console.WriteLine("Матриця суміжності напрямленого графа:");
        Generator.PrintMatrix(matrix);
        System.Console.WriteLine("Матриця суміжності ненапрямленого графа:");
        Generator.PrintMatrix(generator.MakeMatrixUndirected(matrix));

        Generator.Await();

        NodeFactory nodeFactory = new(10);
        nodeFactory.CreateAll();

        LinkFactory directedLinkFactory = new(nodeFactory);
        directedLinkFactory.CreateAll(matrix);
        LinkFactory undirectedLinkFactory = new(nodeFactory, true);
        undirectedLinkFactory.CreateAll(matrix);

        ApplicationConfiguration.Initialize();
        new DirectedForm(nodeFactory, directedLinkFactory).Show();
        Application.Run(new UndirectedForm(nodeFactory, undirectedLinkFactory));
    }
}
```

Примітка. Файли DirectedForm.Designer.cs та UndirectedForm.Designer.cs – службові файли, що необхідні для роботи інтерфейсу програмування WinForms.

Тестування програми







Висновки

У цій лабораторній роботі було створено програму для генерації матриць суміжностей напрямленого та ненапрямленого графів та формування зображення цих графів у графічному вікні мовою програмування C# (платформа .NET, інтерфейс програмування графіки WinForms).

Програма послідовно виконує алгоритм дій, задіюючи відповідні модулі, а саме:

1. Вираховує необхідні дані з номера варіанту та генерує матриці суміжностей графів (модуль Generator.cs)
2. Створює та обчислює вигляд верхівок графу (модуль NodeFactory.cs)
3. Створює та обчислює вигляд ребер графу (модуль LinkFactory.cs)
4. Відрисовує зображення напрямленого (DirectedForm.cs) та ненапрямленого (UndirectedForm.cs) графів у графічних вікнах, використовуючи примітиви з інтерфейсу WinForms

Під час створення програми було задіяно набуті знання із модульного програмування, комп’ютерної та дискретної математики, а також лінійної алгебри та тригонометрії (для відрисовки зображень). Цікавим технічним

випробуванням став прорахунок шляхів ребер графу, аби вони не перетинали вершини.

Графи є незамінним інструментом, адже вони дозволяють ефективно моделювати та аналізувати взаємозв'язки між об'єктами. Вони лежать в основі алгоритмів пошуку (наприклад, у Google), соціальних мереж, маршрутизації в комп'ютерних мережах, рекомендаційних систем і навіть штучного інтелекту. Використовуючи графи, можна зменшити складність задачі, швидше знаходити оптимальні рішення та будувати гнучкі, масштабовані архітектури програмного забезпечення.

Напрямлені та ненапрямлені графи потрібні для моделювання різних типів зв'язків між об'єктами. Ненапрямлені графи використовуються, коли зв'язок між вершинами є симетричним — наприклад, у соціальних мережах для зображення дружби або у мережах доріг з двостороннім рухом. Напрямлені графи застосовуються там, де важливий напрямок взаємодії, як-от в моделюванні потоків (транспорт, електрика), організаційних структур, веб-посилань чи залежностей між задачами. Такий поділ дозволяє точно та ефективно описувати і аналізувати реальні процеси та системи.