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 №4

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 напрямлений та ненапрямлений графи із заданими параметрами так само, як у лабораторній роботі №3.
    - кількість вершин n;
    - розміщення вершин;
    - матриця суміжності A.
2. Обчислити:
    - степені вершин напрямленого і ненапрямленого графів;
    - напівстепені виходу та заходу напрямленого графа;
    - чи є граф однорідним (регулярним), і якщо так, вказати степінь однорідності графа;
    - перелік висячих та ізольованих вершин.

   Результати вивести у графічне вікно, консоль або файл.
3. Змінити матрицю A, коефіцієнт k = 1.0 - n3 * 0.005 - n4 * 0.005 - 0.27
4. Для нового орграфа обчислити:
    - півстепені вершин;
    - всі шляхи довжини 2 і 3;
    - матрицю досяжності;
    - матрицю сильної зв’язності;
    - перелік компонент сильної зв’язності;
    - граф конденсації.

Варіант №3

Номер варіанту: 4303

Кількість вершин n = 10

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

Початковий коефіцієнт k (2 перших графи) = 0,696(9)

Тексти програм

src/Forms/DirectedGraph/DirectedForm.cs

```
namespace lab4;

public partial class DirectedForm : Form
{
    private readonly List<Vertex> _vertices;
    private readonly List<Edge> _edges;

    private Graphics? _graphics;
```

```
public DirectedForm(List<Vertex> vertices, List<Edge> edges)
{
    _vertices = vertices;
    _edges = edges;

    InitializeComponent();
}

private const int _textOffset = 10;

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

    foreach (var edge in _edges)
    {
        Draw(edge);
    }
    foreach (var vertex in _vertices)
    {
        Draw(vertex);
    }
}

private void Draw(Vertex vertex)
{
    if (_graphics == null) return;

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

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

private void Draw(Edge edge)
{
    if (_graphics == null) return;

    switch (edge.Type)
    {
        case EdgeType.Normal:
```

```
            {
                var (start, end) = edge.GetLineCoords();

                if (!edge.HasInvertion)
                {
                    DrawArrow(start, end);
                }
                else
                {
                    DrawLine(start, edge.PolygonalLinkVertex);
                    DrawArrow(edge.PolygonalLinkVertex, end);
                }

                break;
            }

        case EdgeType.VisibilityObstructed:
            {
                var (start, end) = edge.GetLineCoords();
                DrawLine(start, edge.PolygonalLinkVertex);
                DrawArrow(edge.PolygonalLinkVertex, end);
                break;
            }

        case EdgeType.SelfPointing:
            DrawLine(edge.From.Point, edge.SelfLinkVertices[0]);
            DrawLine(edge.SelfLinkVertices[0], edge.SelfLinkVertices[1]);
            DrawArrow(edge.SelfLinkVertices[1], edge.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 lab4;

public partial class UndirectedForm : Form
{
    private readonly List<Vertex> _vertices;
    private readonly List<Edge> _edges;

    private Graphics? _graphics;

    public UndirectedForm(List<Vertex> vertices, List<Edge> edges)
    {
        _vertices = vertices;
        _edges = edges;

        InitializeComponent();
    }

    private const int _textOffset = 10;

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

        foreach (var edge in _edges)
        {
            Draw(edge);
        }
        foreach (var vertex in _vertices)
        {
            Draw(vertex);
        }
    }

    private void Draw(Vertex vertice)
    {
        if (_graphics == null) return;

        _graphics.FillEllipse(
            new SolidBrush(Color.LightGray),
            vertice.Point.X - Vertex.Radius,
```

```
            vertice.Point.Y - Vertex.Radius,
            Vertex.Radius * 2,
            Vertex.Radius * 2
        );

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

    private void Draw(Edge edge)
    {
        if (_graphics == null) return;

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

            case EdgeType.VisibilityObstructed:
                {
                    DrawLine(edge.From.Point, edge.PolygonalLinkVertex);
                    DrawLine(edge.PolygonalLinkVertex, edge.To.Point);
                    break;
                }

            case EdgeType.SelfPointing:
                DrawLine(edge.From.Point, edge.SelfLinkVertices[0]);
                DrawLine(edge.SelfLinkVertices[0], edge.SelfLinkVertices[1]);
                DrawLine(edge.SelfLinkVertices[1], edge.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 lab4;

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

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

        ParseFromCode();

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

    public 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) * 0.01f - GetN(4) * 0.001f - 0.3f;
        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 static void Await()
    {
        Console.Write($"\nНатисніть Enter, щоб продовжити...");
        Console.ReadLine();
    }
}
```

src/Service/VertexFactory.cs

```
namespace lab4;

public class VertexFactory(int n)
{
    private readonly int _n = n;
    private readonly List<Vertex> _nodes = [];
    public List<Vertex> Vertices => _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 Vertex(i, x, y, outer));
        }
    }
}

public class Vertex(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
}
```

src/Service/EdgeFactory.cs

```
namespace lab4;

public class EdgeFactory(VertexFactory nodeFactory, bool undirected = false)
{
    private readonly VertexFactory _nodeFactory = nodeFactory;
    private readonly List<Edge> _links = [];
    public List<Edge> Edges => _links;
    private readonly bool _undirected = undirected;

    public void CreateAll(int[,] matrix)
    {
        _links.Clear();

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

```
}

public class Edge
{
    public Vertex From { get; }
    public Vertex To { get; }
    public EdgeType Type { get; }
    public bool HasInvertion { get; }
    private readonly int _offset = 40;

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

    public Edge(Vertex from, Vertex to, bool hasInvertion)
    {
        From = from;
        To = to;
        HasInvertion = hasInvertion;

        if (hasInvertion)
            _offset = -_offset;

        if (from == to)
        {
            Type = EdgeType.SelfPointing;
            SelfLinkVertices = [
                new Point(from.Point.X, from.Point.Y - Vertex.Radius * 2),
                new Point(from.Point.X - Vertex.Radius * 2, from.Point.Y),
                new Point(from.Point.X - Vertex.Radius, from.Point.Y),
            ];
        }
        else if (
            from.Point.X == to.Point.X &&
            MathF.Abs(from.Point.Y - to.Point.Y) != VertexFactory.Gap
        )
        {
            Type = EdgeType.VisibilityObstructed;
            PolygonalLinkVertex = 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) != VertexFactory.Gap
        )
        {
            Type = EdgeType.VisibilityObstructed;
            PolygonalLinkVertex = new(
                (from.Point.X + to.Point.X) / 2,
```

```
                from.Point.Y + (
                    from.Outer == Direction.Down ? +_offset : -_offset
                )
            );
        }

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

    }
    public static bool IsRectangleSide(Edge 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 * Vertex.Radius, From.Point.Y + uy *
Vertex.Radius);
        PointF end = new(To.Point.X - ux * Vertex.Radius, To.Point.Y - uy *
Vertex.Radius);

        return (start, end);
    }

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

}

public enum EdgeType
{
    Normal,
    SelfPointing,
    VisibilityObstructed
}
```

src/Service/MatrixUtils.cs

```
using System.Linq;

namespace lab4;
```

```
public static class MatrixUtils
{
    public static void Print(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();
        }
    }

    public static int[,] ToUndirected(int[,] matrix)
    {
        int n = matrix.GetLength(0);
        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 int[,] Multiply(int[,] matrix1, int[,] matrix2)
    {
        int n = matrix1.GetLength(0);
        int[,] result = new int[n, n];

        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
            {
                result[i, j] = 0;
                for (int k = 0; k < n; k++)
                    result[i, j] += matrix1[i, k] * matrix2[k, j];
            }

        return result;
    }

    public static int[,] PerElementMultiply(int[,] matrix1, int[,] matrix2)
    {
        int n = matrix1.GetLength(0);
        int[,] result = new int[n, n];

        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
```

```
            result[i, j] = matrix1[i, j] * matrix2[i, j];

    return result;
}

public static (int, int)[] DirGraphVerticeDegrees(int[,] compatabilityMatrix)
{
    int n = compatabilityMatrix.GetLength(0);
    (int, int)[] degrees = new (int, int)[n];

    for (int i = 0; i < n; i++)
    {
        (int, int) degree = (0, 0); // in, out
        for (int j = 0; j < n; j++)
        {
            degree.Item1 += compatabilityMatrix[j, i];
            degree.Item2 += compatabilityMatrix[i, j];
        }
        degrees[i] = degree;
    }
    return degrees;
}

public static int[] UndirGraphVerticeDegrees(int[,] compatabilityMatrix)
{
    int n = compatabilityMatrix.GetLength(0);
    int[] degrees = new int[n];

    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            degrees[i] += compatabilityMatrix[i, j];

            if (compatabilityMatrix[i, j] == 1 && i == j) degrees[i]++;
        }
    }

    return degrees;
}

public static void PrintVerticeDegrees(int[] undirDegrees)
{
    System.Console.WriteLine("Степені вершин:");
    for (int i = 0; i < undirDegrees.Length; i++)
    {
        Console.WriteLine($"Вершина {i}: степінь = {undirDegrees[i]}");
    }
}

public static void PrintVerticeDegrees((int, int)[] dirDegrees)
{
    System.Console.WriteLine("Степені вершин:");
```

```
        for (int i = 0; i < dirDegrees.Length; i++)
        {
            Console.WriteLine($"Вершина {i}: степінь = {dirDegrees[i].Item1 +
dirDegrees[i].Item2} (від'ємна = {dirDegrees[i].Item2}, додатна =
{dirDegrees[i].Item1})");
        }
    }

    public static void CheckForRegularity(int[] undirDegrees)
    {
        int degree = undirDegrees[0];

        if (undirDegrees.All(d => d == degree))
        {
            Console.WriteLine($"Граф регулярний, степінь однорідності: {degree}");
        }
        else
        {
            Console.WriteLine("Граф не регулярний");
        }
    }

    public static void CheckForRegularity((int, int)[] dirDegrees)
    {
        int degree = dirDegrees[0].Item1 + dirDegrees[0].Item2;

        if (dirDegrees.All(d => d.Item1 + d.Item2 == degree))
        {
            Console.WriteLine($"Граф регулярний, степінь однорідності: {degree}");
        }
        else
        {
            Console.WriteLine("Граф не регулярний");
        }
    }

    public static void CheckForHangingVertices(int[] undirDegrees)
    {
        bool flag = false;
        for (int i = 0; i < undirDegrees.Length; i++)
        {
            if (undirDegrees[i] == 1)
            {
                Console.WriteLine($"Вершина {i} - кінцева (висяча)");
                flag = true;
            }
        }

        if (!flag) Console.WriteLine("В графі немає кінцевих (висячих) вершин");
    }

    public static void CheckForHangingVertices((int, int)[] dirDegrees)
    {
```

```
    bool flag = false;
    for (int i = 0; i < dirDegrees.Length; i++)
    {
        if (dirDegrees[i].Item1 + dirDegrees[i].Item2 == 1)
        {
            Console.WriteLine($"Вершина {i} - кінцева (висяча)");
            flag = true;
        }
    }

    if (!flag) Console.WriteLine("В графі немає кінцевих (висячих) вершин");
}

public static void PrintPathsLength2(int[,] matrix)
{
    System.Console.WriteLine("Всі шляхи довжиною 2:");

    int n = matrix.GetLength(0);
    int[,] matrixSquared = Multiply(matrix, matrix);

    for (int i = 0; i < n; i++)
    {
        bool first = true;
        for (int j = 0; j < n; j++)
        {
            if (matrixSquared[i, j] > 0)
            {
                for (int k = 0; k < n; k++)
                {
                    if (matrix[i, k] == 1 && matrix[k, j] == 1)
                    {
                        if (!first) Console.Write(", ");
                        first = false;
                        Console.Write($"{i} → {k} → {j}");
                    }
                }
            }
        }
        Console.WriteLine("\n");
    }
}

public static void PrintPathsLength3(int[,] matrix)
{
    System.Console.WriteLine("Всі шляхи довжиною 3:");

    int n = matrix.GetLength(0);
    int[,] matrixSquared = Multiply(matrix, matrix);
    int[,] matrixCubed = Multiply(matrixSquared, matrix);

    for (int i = 0; i < n; i++)
    {
        bool first = true;
```

```
            for (int j = 0; j < n; j++)
            {
                if (matrixCubed[i, j] > 0)
                {
                    for (int k = 0; k < n; k++)
                        for (int l = 0; l < n; l++)
                        {
                            if (matrix[i, k] == 1 && matrix[k, l] == 1 && matrix[l,
j] == 1)
                            {
                                if (!first) Console.Write(", ");
                                first = false;
                                Console.Write($"{i} → {k} → {l} → {j}");
                            }
                        }
                }
            }
            Console.WriteLine("\n");
        }
    }

    public static int[,] GetReachabilityMatrix(int[,] adjMatrix)
    {
        int n = adjMatrix.GetLength(0);
        int[,] reach = new int[n, n];

        // Копія
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                reach[i, j] = adjMatrix[i, j];

        // Алгоритм Воршелла
        for (int k = 0; k < n; k++)
            for (int i = 0; i < n; i++)
                for (int j = 0; j < n; j++)
                    if (reach[i, k] == 1 && reach[k, j] == 1)
                        reach[i, j] = 1;

        return reach;
    }

    public static int[,] Transpone(int[,] matrix)
    {
        int n = matrix.GetLength(0);
        int[,] result = new int[n, n];

        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                result[i, j] = matrix[j, i];

        return result;
    }
```

```
    public static int[,] GetStrongConnectivityMatrix(int[,] reachMatrix)
    {
        int[,] transpone = Transpone(reachMatrix);
        return PerElementMultiply(reachMatrix, transpone);
    }

    public static List<List<int>> GetStrongConnectedComponents(int[,]
strongConnectivityMatrix)
    {
        int n = strongConnectivityMatrix.GetLength(0);
        bool[] visited = new bool[n];
        List<List<int>> components = [];

        for (int i = 0; i < n; i++)
        {
            if (!visited[i])
            {
                List<int> component = new List<int> { i };
                visited[i] = true;

                for (int j = i + 1; j < n; j++)
                {
                    if (!visited[j] && strongConnectivityMatrix[i, j] == 1)
                    {
                        component.Add(j);
                        visited[j] = true;
                    }
                }

                components.Add(component);
            }
        }

        return components;
    }

    public static void Print(List<List<int>> strongConnectedComponents)
    {
        System.Console.WriteLine("Компоненти сильної зв'язності:");
        for (int i = 0; i < strongConnectedComponents.Count; i++)
        {
            Console.WriteLine($"Компонента {i}: {string.Join(", ",
strongConnectedComponents[i])}");
        }
    }

    public static (List<Vertex> condVertices, List<Edge> condEdges)
        ComputeCondensation(
            List<List<int>> strongConnectedComponents,
            List<Vertex> vertices,
            List<Edge> edges)
    {
        int n = vertices.Count;
```

```
        int[] compIndex = new int[n];
        for (int i = 0; i < strongConnectedComponents.Count; i++)
            foreach (int v in strongConnectedComponents[i])
                compIndex[v] = i;


        var vf = new VertexFactory(strongConnectedComponents.Count);
        vf.CreateAll();
        var condVertices = vf.Vertices;

        var condEdges = new List<Edge>();
        var seen = new HashSet<(int, int)>(); // уникаючи дублікатів
        foreach (var e in edges)
        {
            int cu = compIndex[e.From.Id];
            int cv = compIndex[e.To.Id];
            if (cu != cv && seen.Add((cu, cv)))
                condEdges.Add(new Edge(condVertices[cu], condVertices[cv], false));
        }

        return (condVertices, condEdges);
    }
}
```

Program.cs (точка входу)

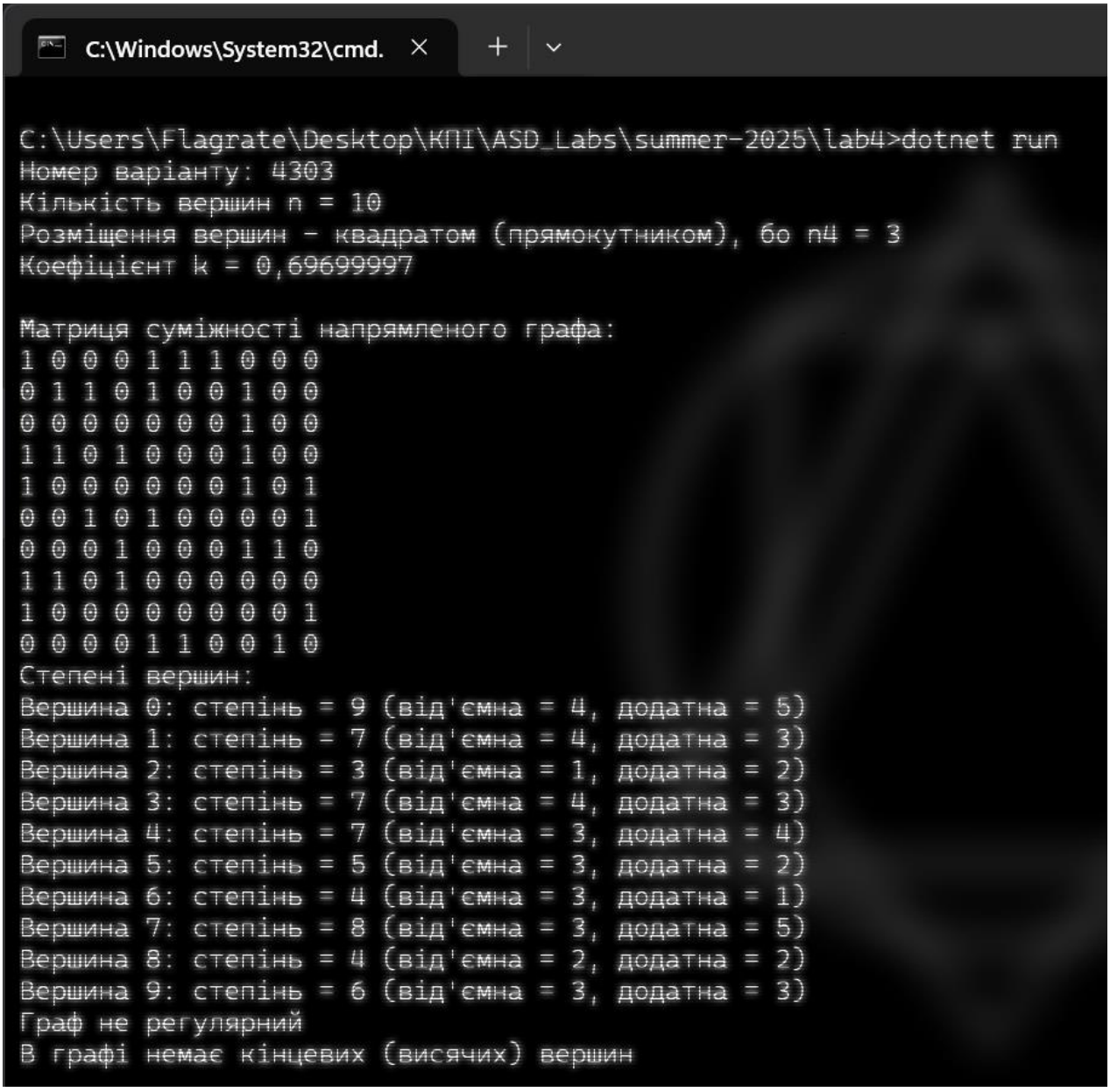
```
namespace lab3;

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);
```

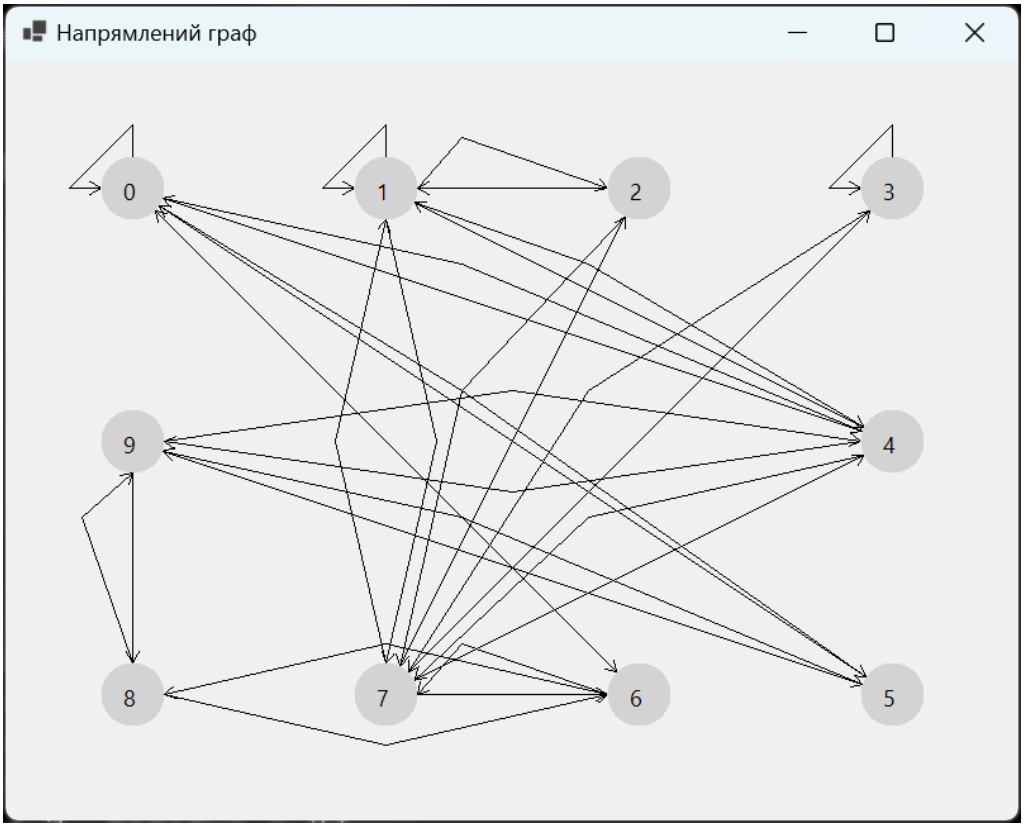
```
            ApplicationConfiguration.Initialize();
            new DirectedForm(nodeFactory, directedLinkFactory).Show();
            Application.Run(new UndirectedForm(nodeFactory, undirectedLinkFactory));
        }
    }
```

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

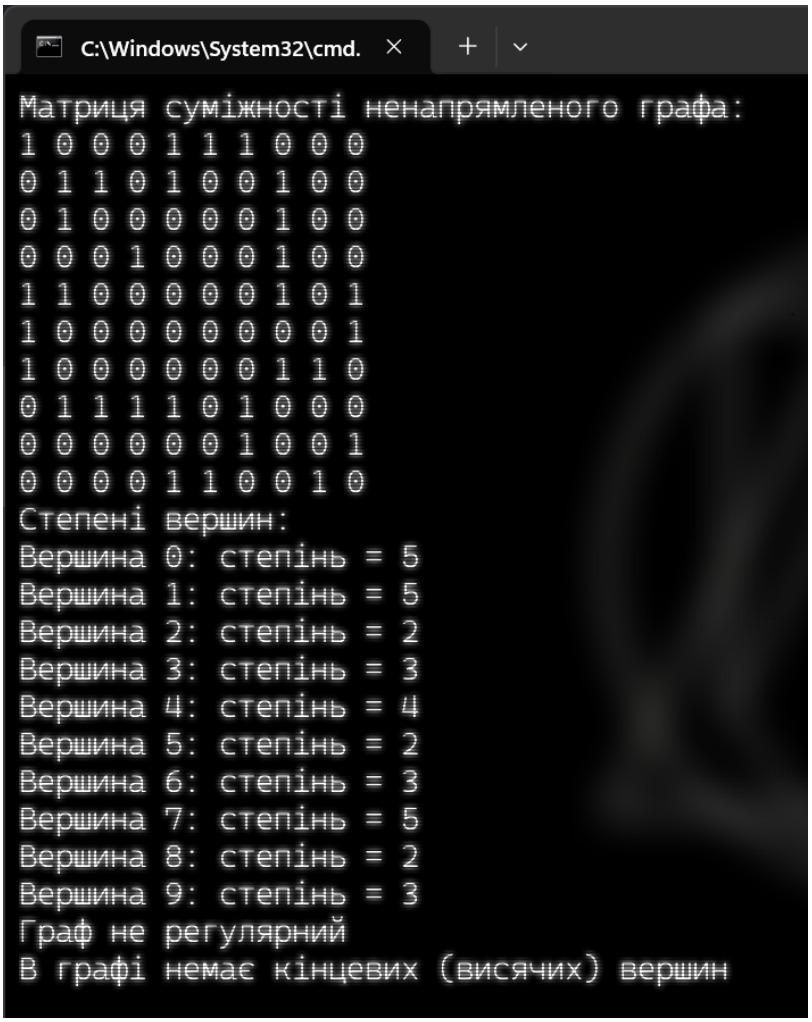
Тестування програми

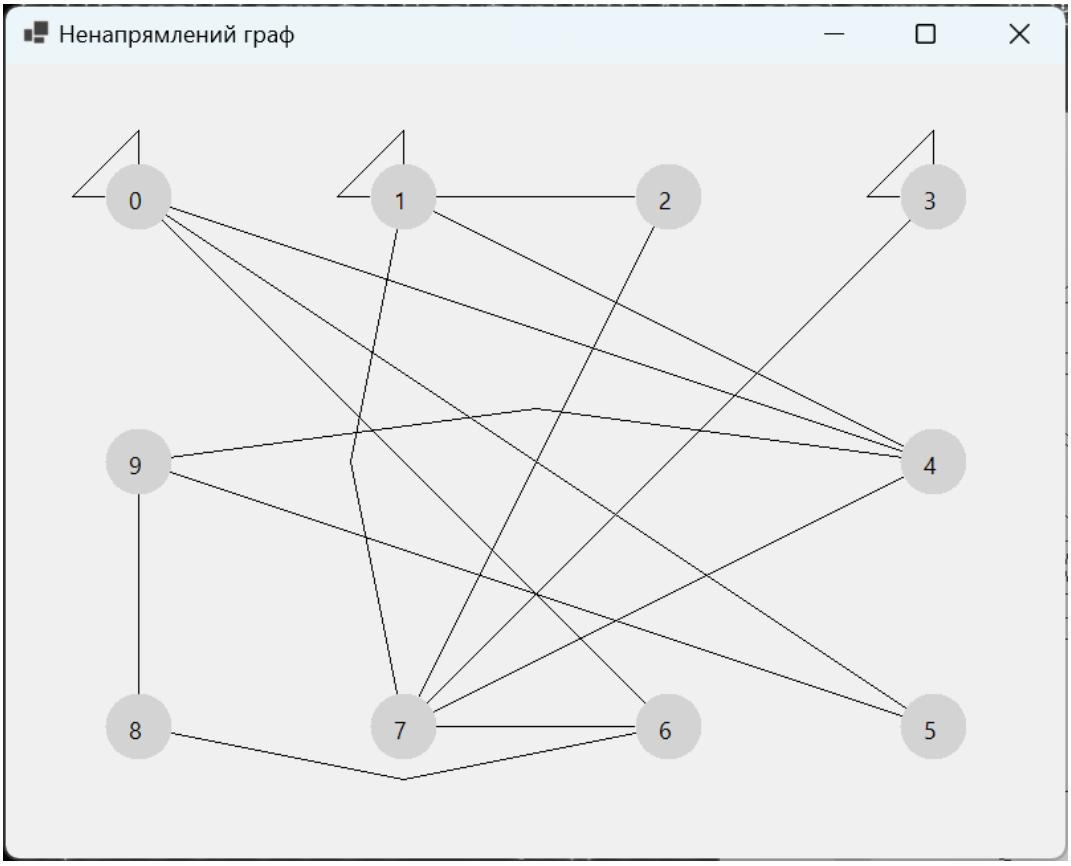
Умови та напрямлений граф



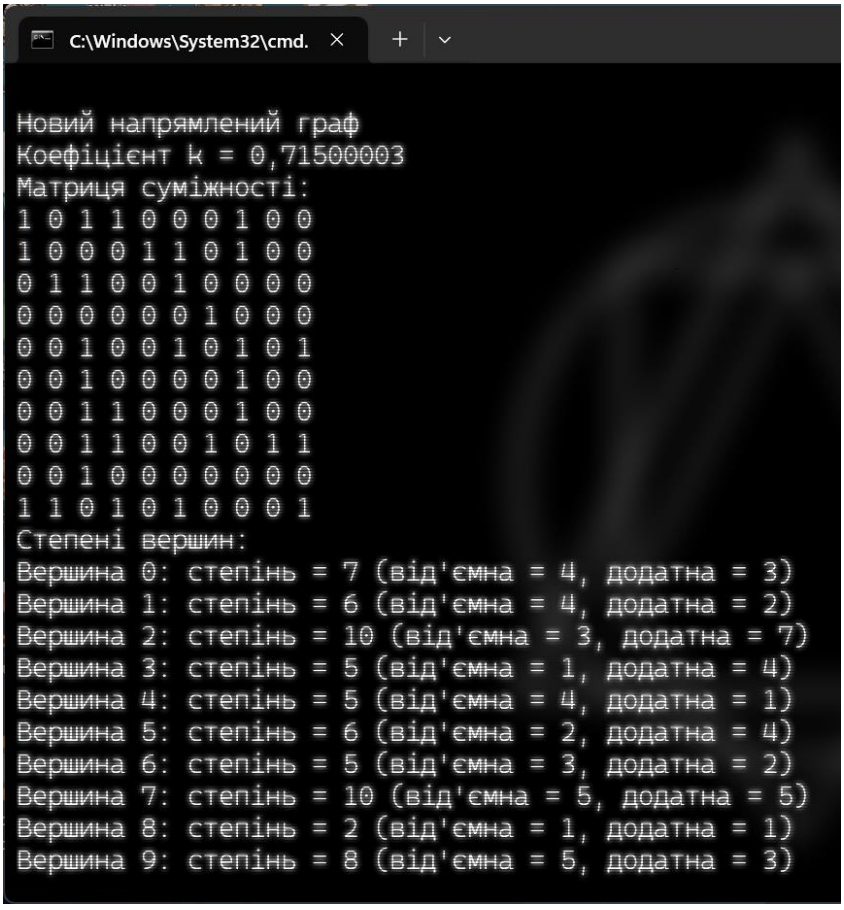


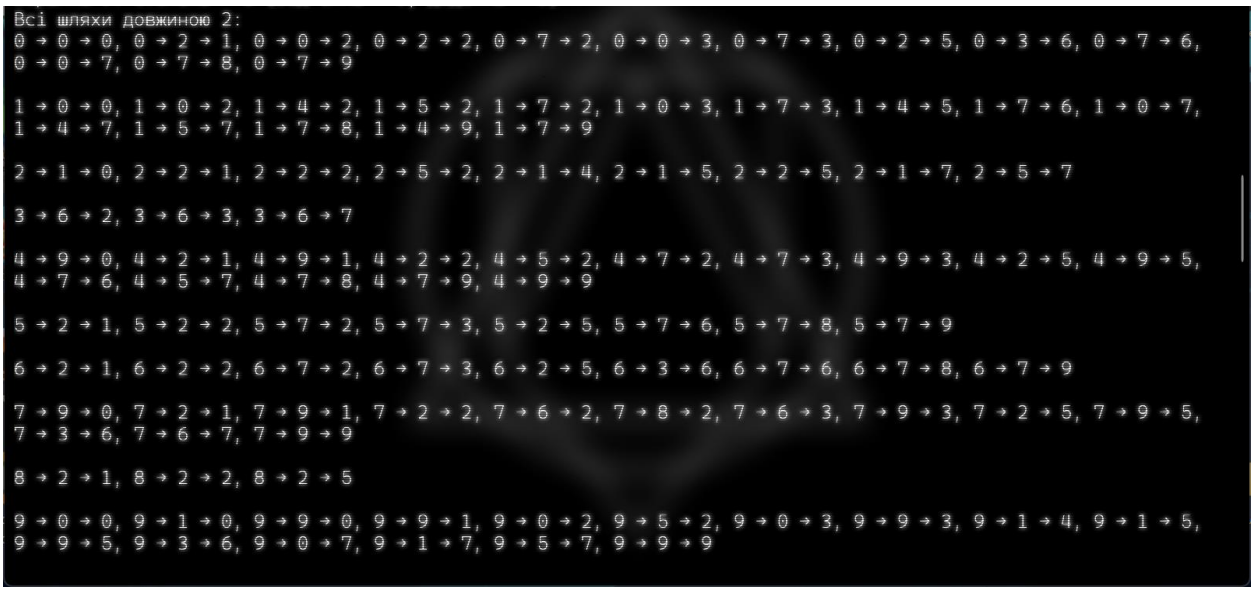
Ненапрямлений граф

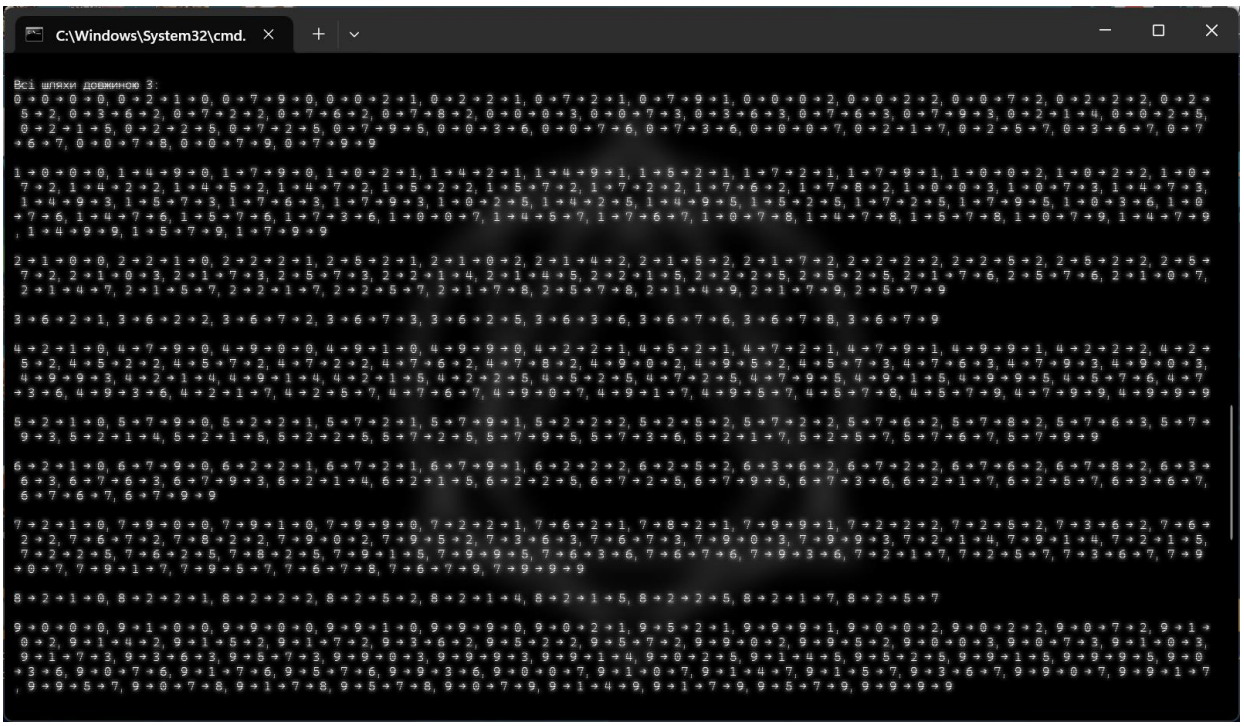




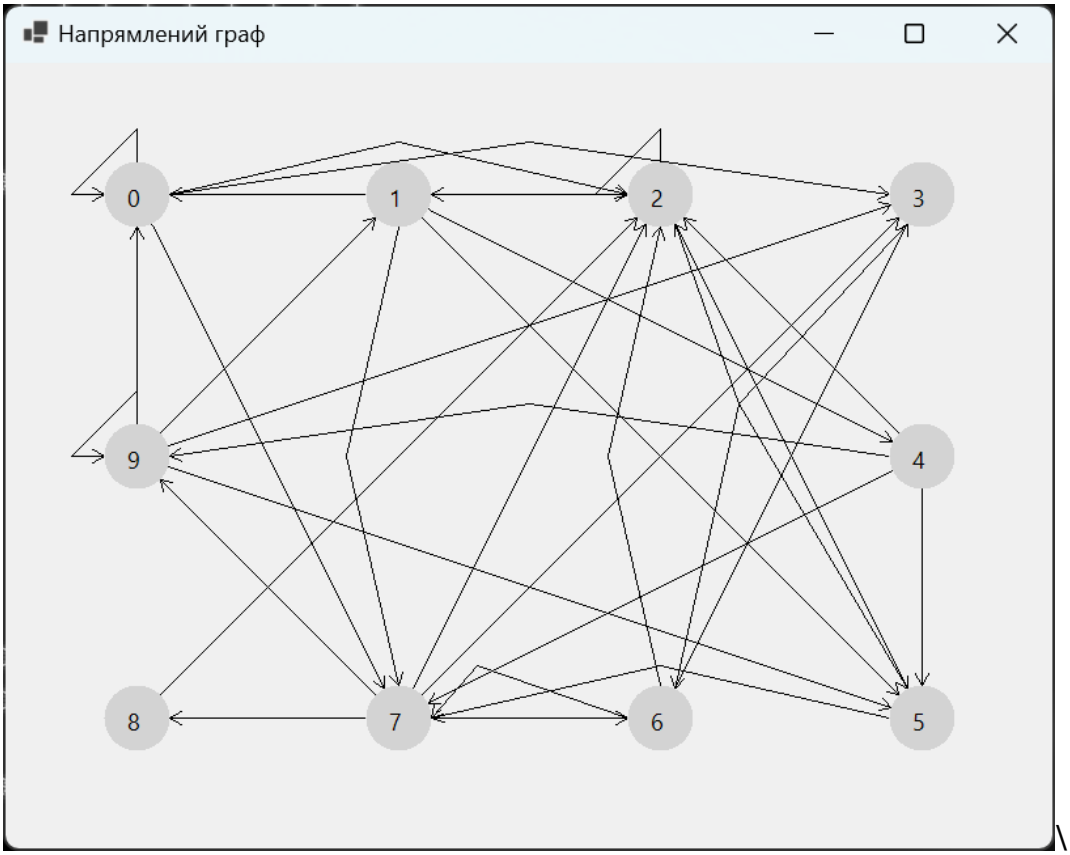
Модифікований напрямлений граф



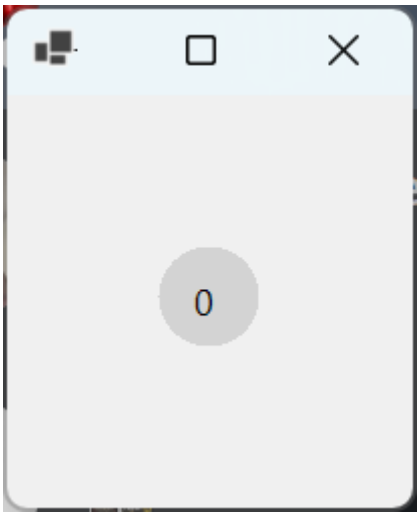

Всі шляхи довжиною 2:
0 → 0 → 0, 0 → 2 → 1, 0 → 0 → 2, 0 → 2 → 2, 0 → 7 → 2, 0 → 0 → 3, 0 → 7 → 3, 0 → 2 → 5, 0 → 3 → 6, 0 → 7 → 6,
0 → 0 → 7, 0 → 7 → 8, 0 → 7 → 9
1 → 0 → 0, 1 → 0 → 2, 1 → 4 → 2, 1 → 5 → 2, 1 → 7 → 2, 1 → 0 → 3, 1 → 7 → 3, 1 → 4 → 5, 1 → 7 → 6, 1 → 0 → 7,
1 → 4 → 7, 1 → 5 → 7, 1 → 7 → 8, 1 → 4 → 9, 1 → 7 → 9
2 → 1 → 0, 2 → 2 → 1, 2 → 2 → 2, 2 → 5 → 2, 2 → 1 → 4, 2 → 1 → 5, 2 → 2 → 5, 2 → 1 → 7, 2 → 5 → 7
3 → 6 → 2, 3 → 6 → 3, 3 → 6 → 7
4 → 9 → 0, 4 → 2 → 1, 4 → 9 → 1, 4 → 2 → 2, 4 → 5 → 2, 4 → 7 → 2, 4 → 7 → 3, 4 → 9 → 3, 4 → 2 → 5, 4 → 9 → 5,
4 → 7 → 6, 4 → 5 → 7, 4 → 7 → 8, 4 → 7 → 9, 4 → 9 → 9
5 → 2 → 1, 5 → 2 → 2, 5 → 7 → 2, 5 → 7 → 3, 5 → 2 → 5, 5 → 7 → 6, 5 → 7 → 8, 5 → 7 → 9
6 → 2 → 1, 6 → 2 → 2, 6 → 7 → 2, 6 → 7 → 3, 6 → 2 → 5, 6 → 3 → 6, 6 → 7 → 6, 6 → 7 → 8, 6 → 7 → 9
7 → 9 → 0, 7 → 2 → 1, 7 → 9 → 1, 7 → 2 → 2, 7 → 6 → 2, 7 → 8 → 2, 7 → 6 → 3, 7 → 9 → 3, 7 → 2 → 5, 7 → 9 → 5,
7 → 3 → 6, 7 → 6 → 7, 7 → 9 → 9
8 → 2 → 1, 8 → 2 → 2, 8 → 2 → 5
9 → 0 → 0, 9 → 1 → 0, 9 → 9 → 0, 9 → 9 → 1, 9 → 0 → 2, 9 → 5 → 2, 9 → 0 → 3, 9 → 9 → 3, 9 → 1 → 4, 9 → 1 → 5,
9 → 9 → 5, 9 → 3 → 6, 9 → 0 → 7, 9 → 1 → 7, 9 → 5 → 7, 9 → 9 → 9


C:\Windows\System32\cmd.
Всі шляхи довжиною 3:
0 → 0 → 0 → 0, 0 → 2 → 1 → 0, 0 → 7 → 9 → 0, 0 → 0 → 2 → 1, 0 → 2 → 2 → 1, 0 → 7 → 2 → 1, 0 → 7 → 9 → 1, 0 → 0 → 0 → 2, 0 → 0 → 2 → 2, 0 → 0 → 7 → 2, 0 → 2 → 2 → 2, 0 → 2 → 5 → 2, 0 → 3 → 6 → 2, 0 → 7 → 2 → 2, 0 → 7 → 6 → 2, 0 → 7 → 8 → 2, 0 → 0 → 0 → 3, 0 → 0 → 7 → 3, 0 → 3 → 6 → 3, 0 → 7 → 6 → 3, 0 → 7 → 9 → 3, 0 → 2 → 1 → 4, 0 → 0 → 2 → 5, 0 → 2 → 1 → 5, 0 → 2 → 2 → 5, 0 → 7 → 2 → 5, 0 → 7 → 9 → 5, 0 → 0 → 3 → 6, 0 → 0 → 7 → 6, 0 → 7 → 3 → 6, 0 → 0 → 0 → 7, 0 → 2 → 1 → 7, 0 → 2 → 5 → 7, 0 → 3 → 6 → 7, 0 → 7 → 6 → 7, 0 → 0 → 7 → 8, 0 → 0 → 7 → 9, 0 → 7 → 9 → 9
1 → 0 → 0 → 0, 1 → 4 → 9 → 0, 1 → 7 → 9 → 0, 1 → 0 → 2 → 1, 1 → 4 → 2 → 1, 1 → 4 → 9 → 1, 1 → 5 → 2 → 1, 1 → 7 → 2 → 1, 1 → 7 → 9 → 1, 1 → 0 → 0 → 2, 1 → 0 → 2 → 2, 1 → 0 → 7 → 2, 1 → 4 → 2 → 2, 1 → 4 → 5 → 2, 1 → 4 → 7 → 2, 1 → 5 → 2 → 2, 1 → 5 → 7 → 2, 1 → 7 → 2 → 2, 1 → 7 → 6 → 2, 1 → 7 → 8 → 2, 1 → 0 → 0 → 3, 1 → 0 → 7 → 3, 1 → 4 → 7 → 3, 1 → 4 → 9 → 3, 1 → 5 → 7 → 3, 1 → 7 → 6 → 3, 1 → 7 → 9 → 3, 1 → 0 → 2 → 5, 1 → 4 → 2 → 5, 1 → 4 → 9 → 5, 1 → 5 → 2 → 5, 1 → 7 → 2 → 5, 1 → 7 → 9 → 5, 1 → 0 → 3 → 6, 1 → 0 → 7 → 6, 1 → 4 → 7 → 6, 1 → 5 → 7 → 6, 1 → 7 → 3 → 6, 1 → 0 → 0 → 7, 1 → 4 → 5 → 7, 1 → 7 → 6 → 7, 1 → 0 → 7 → 8, 1 → 4 → 7 → 8, 1 → 5 → 7 → 8, 1 → 0 → 7 → 9, 1 → 4 → 7 → 9, 1 → 4 → 9 → 9, 1 → 5 → 7 → 9, 1 → 7 → 9 → 9
2 → 1 → 0 → 0, 2 → 2 → 1 → 0, 2 → 2 → 2 → 1, 2 → 5 → 2 → 1, 2 → 1 → 0 → 2, 2 → 1 → 4 → 2, 2 → 1 → 5 → 2, 2 → 1 → 7 → 2, 2 → 2 → 2 → 2, 2 → 2 → 5 → 2, 2 → 5 → 2 → 2, 2 → 5 → 7 → 2, 2 → 1 → 0 → 3, 2 → 1 → 7 → 3, 2 → 5 → 7 → 3, 2 → 2 → 1 → 4, 2 → 1 → 4 → 5, 2 → 2 → 1 → 5, 2 → 2 → 2 → 5, 2 → 5 → 2 → 5, 2 → 1 → 7 → 6, 2 → 5 → 7 → 6, 2 → 1 → 0 → 7, 2 → 1 → 4 → 7, 2 → 1 → 5 → 7, 2 → 2 → 1 → 7, 2 → 2 → 5 → 7, 2 → 1 → 7 → 8, 2 → 5 → 7 → 8, 2 → 1 → 4 → 9, 2 → 1 → 7 → 9, 2 → 5 → 7 → 9
3 → 6 → 2 → 1, 3 → 6 → 2 → 2, 3 → 6 → 7 → 2, 3 → 6 → 7 → 3, 3 → 6 → 2 → 5, 3 → 6 → 3 → 6, 3 → 6 → 7 → 6, 3 → 6 → 7 → 8, 3 → 6 → 7 → 9
4 → 2 → 1 → 0, 4 → 7 → 9 → 0, 4 → 9 → 0 → 0, 4 → 9 → 1 → 0, 4 → 9 → 9 → 0, 4 → 2 → 2 → 1, 4 → 5 → 2 → 1, 4 → 7 → 2 → 1, 4 → 7 → 9 → 1, 4 → 9 → 9 → 1, 4 → 2 → 2 → 2, 4 → 2 → 5 → 2, 4 → 5 → 2 → 2, 4 → 5 → 7 → 2, 4 → 7 → 2 → 2, 4 → 7 → 6 → 2, 4 → 7 → 8 → 2, 4 → 9 → 0 → 2, 4 → 9 → 5 → 2, 4 → 5 → 7 → 3, 4 → 7 → 6 → 3, 4 → 7 → 9 → 3, 4 → 9 → 0 → 3, 4 → 9 → 9 → 3, 4 → 2 → 1 → 4, 4 → 9 → 1 → 4, 4 → 2 → 1 → 5, 4 → 2 → 2 → 5, 4 → 5 → 2 → 5, 4 → 7 → 2 → 5, 4 → 7 → 9 → 5, 4 → 9 → 1 → 5, 4 → 9 → 9 → 5, 4 → 5 → 7 → 6, 4 → 7 → 3 → 6, 4 → 9 → 3 → 6, 4 → 2 → 1 → 7, 4 → 2 → 5 → 7, 4 → 7 → 6 → 7, 4 → 9 → 0 → 7, 4 → 9 → 1 → 7, 4 → 9 → 5 → 7, 4 → 5 → 7 → 8, 4 → 5 → 7 → 9, 4 → 7 → 9 → 9, 4 → 9 → 9 → 9
5 → 2 → 1 → 0, 5 → 7 → 9 → 0, 5 → 2 → 2 → 1, 5 → 7 → 2 → 1, 5 → 7 → 9 → 1, 5 → 2 → 2 → 2, 5 → 2 → 5 → 2, 5 → 7 → 2 → 2, 5 → 7 → 6 → 2, 5 → 7 → 8 → 2, 5 → 7 → 6 → 3, 5 → 7 → 9 → 3, 5 → 2 → 1 → 4, 5 → 2 → 1 → 5, 5 → 2 → 2 → 5, 5 → 7 → 2 → 5, 5 → 7 → 9 → 5, 5 → 7 → 3 → 6, 5 → 2 → 1 → 7, 5 → 2 → 5 → 7, 5 → 7 → 6 → 7, 5 → 7 → 9 → 9
6 → 2 → 1 → 0, 6 → 7 → 9 → 0, 6 → 2 → 2 → 1, 6 → 7 → 2 → 1, 6 → 7 → 9 → 1, 6 → 2 → 2 → 2, 6 → 2 → 5 → 2, 6 → 3 → 6 → 2, 6 → 7 → 2 → 2, 6 → 7 → 6 → 2, 6 → 7 → 8 → 2, 6 → 3 → 6 → 3, 6 → 7 → 6 → 3, 6 → 7 → 9 → 3, 6 → 2 → 1 → 4, 6 → 2 → 1 → 5, 6 → 2 → 2 → 5, 6 → 7 → 2 → 5, 6 → 7 → 9 → 5, 6 → 7 → 3 → 6, 6 → 2 → 1 → 7, 6 → 2 → 5 → 7, 6 → 3 → 6 → 7, 6 → 7 → 6 → 7, 6 → 7 → 9 → 9
7 → 2 → 1 → 0, 7 → 9 → 0 → 0, 7 → 9 → 1 → 0, 7 → 9 → 9 → 0, 7 → 2 → 2 → 1, 7 → 6 → 2 → 1, 7 → 8 → 2 → 1, 7 → 9 → 9 → 1, 7 → 2 → 2 → 2, 7 → 2 → 5 → 2, 7 → 3 → 6 → 2, 7 → 6 → 2 → 2, 7 → 6 → 7 → 2, 7 → 8 → 2 → 2, 7 → 9 → 0 → 2, 7 → 9 → 5 → 2, 7 → 3 → 6 → 3, 7 → 6 → 7 → 3, 7 → 9 → 0 → 3, 7 → 9 → 9 → 3, 7 → 2 → 1 → 4, 7 → 9 → 1 → 4, 7 → 2 → 1 → 5, 7 → 2 → 2 → 5, 7 → 6 → 2 → 5, 7 → 8 → 2 → 5, 7 → 9 → 1 → 5, 7 → 9 → 9 → 5, 7 → 6 → 3 → 6, 7 → 6 → 7 → 6, 7 → 9 → 3 → 6, 7 → 2 → 1 → 7, 7 → 2 → 5 → 7, 7 → 3 → 6 → 7, 7 → 9 → 0 → 7, 7 → 9 → 1 → 7, 7 → 9 → 5 → 7, 7 → 6 → 7 → 8, 7 → 6 → 7 → 9, 7 → 9 → 9 → 9
8 → 2 → 1 → 0, 8 → 2 → 2 → 1, 8 → 2 → 2 → 2, 8 → 2 → 5 → 2, 8 → 2 → 1 → 4, 8 → 2 → 1 → 5, 8 → 2 → 2 → 5, 8 → 2 → 1 → 7, 8 → 2 → 5 → 7
9 → 0 → 0 → 0, 9 → 1 → 0 → 0, 9 → 9 → 0 → 0, 9 → 9 → 1 → 0, 9 → 9 → 9 → 0, 9 → 0 → 2 → 1, 9 → 5 → 2 → 1, 9 → 9 → 9 → 1, 9 → 0 → 0 → 2, 9 → 0 → 2 → 2, 9 → 0 → 7 → 2, 9 → 1 → 0 → 2, 9 → 1 → 4 → 2, 9 → 1 → 5 → 2, 9 → 1 → 7 → 2, 9 → 3 → 6 → 2, 9 → 5 → 2 → 2, 9 → 5 → 7 → 2, 9 → 9 → 0 → 2, 9 → 9 → 5 → 2, 9 → 0 → 0 → 3, 9 → 0 → 7 → 3, 9 → 1 → 0 → 3, 9 → 1 → 7 → 3, 9 → 3 → 6 → 3, 9 → 5 → 7 → 3, 9 → 9 → 0 → 3, 9 → 9 → 9 → 3, 9 → 9 → 1 → 4, 9 → 0 → 2 → 5, 9 → 1 → 4 → 5, 9 → 5 → 2 → 5, 9 → 9 → 1 → 5, 9 → 9 → 9 → 5, 9 → 0 → 3 → 6, 9 → 0 → 7 → 6, 9 → 1 → 7 → 6, 9 → 5 → 7 → 6, 9 → 9 → 3 → 6, 9 → 0 → 0 → 7, 9 → 1 → 0 → 7, 9 → 1 → 4 → 7, 9 → 1 → 5 → 7, 9 → 3 → 6 → 7, 9 → 9 → 0 → 7, 9 → 9 → 1 → 7, 9 → 9 → 5 → 7, 9 → 0 → 7 → 8, 9 → 1 → 7 → 8, 9 → 5 → 7 → 8, 9 → 0 → 7 → 9, 9 → 1 → 4 → 9, 9 → 1 → 7 → 9, 9 → 5 → 7 → 9, 9 → 9 → 9 → 9


Матриця досяжності:
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
Матриця сильної зв'язності:
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
Компоненти сильної зв'язності:
Компонента 0: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Натисніть Enter, щоб продовжити...



Конденсований граф модифікованого орграфа



Висновки

У цій лабораторній роботі було створено програму для обрахування напрямленого та ненапрямленого графів та формування зображення цих графів у графічному вікні мовою програмування C# (платформа .NET, інтерфейс програмування графіки WinForms), а також визначення їх характеристик (досяжність, сильна зв’язність, тощо).

Всього було обраховано та виведено користувачу у графічне вікно та у консоль 4 графи:

- Напрямлений граф
- Ненапрямлений граф
- Модифікований напрямлений граф (після зміни коефіцієнту k)
- Конденсований граф модифікованого напрямленого графа

Характеристики графів: степені вершин, шляхи заданої довжини, досяжність, сильна зв'язність та конденсація – мають численні практичні застосування в комп'ютерних науках, соціальних мережах, біоінформатиці, телекомунікаціях і транспорті. Степені та напівстепені вершин допомагають оцінювати центральність та стійкість мереж (наприклад, ідентифікувати "хаби" в соціальних чи епідеміологічних моделях). Аналіз шляхів фіксованої довжини використовують у маршрутизації, криптографії та біоінформатиці для пошуку патернів взаємодії. Досяжність забезпечує швидке опитування, чи існує зв'язок між двома вузлами, що критично для СУБД та мережевих протоколів. Сильна зв'язність виділяє автономні підмережі в Інтернеті, соціальних платформах та транспортних системах. Конденсація спрощує аналіз великих орієнтованих графів, перетворюючи їх на ациклічні, що застосовується в компіляторах, оптимізації робочих процесів та GNN-моделях.