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 №5

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 граф із заданими параметрами так само, як у лабораторній роботі №3.
2. Створити програму, яка виконує обхід напрямленого графа вшир (BFS) та вглиб (DFS). Результати вивести у графічне вікно, консоль або файл.
3. Під час обходу графа побудувати дерево обходу. У програмі дерево обходу виводити покроково у процесі виконання обходу графа. *Це зроблено виділенням іншим кольором ребер графа.*
4. Зміну статусів вершин у процесі обходу продемонструвати зміною кольорів вершин, графічними позначками тощо, або ж у процесі обходу виводити протокол обходу у графічне вікно або в консоль *(обрано 1 варіант)*
5. Якщо після обходу графа лишилися невідвідані вершини, продовжувати обхід з невідвіданої вершини з найменшим номером, яка має щонайменше одну вихідну дугу

Варіант №3

Номер варіанту: 4303

Кількість вершин n = 10

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

Коефіцієнт k = 0,83500004

Тексти програм

src/Forms/DirectedGraph/DirectedForm.cs

```
namespace lab5;

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



    public DirectedForm(
        List<Vertex> vertices,
        List<Edge> edges,
        int[,] matrix,
        GraphTraversalStrategy strategy
    )
    {
        _vertices = vertices;
        _edges = edges;
```

```
    GraphSearchUtils.OnRedrawNeeded += Invalidate;

    GraphSearchUtils.OnEdgeVisited += (v1, v2) =>
    {
        var edge = _edges.First(e => e.From.Id == v1 && e.To.Id == v2);
        edge.Visited = true;
        Invalidate();
    };

    InitializeComponent();

    if (strategy == GraphTraversalStrategy.BFS)
    {
        GraphSearchUtils.OnBFSFinished += Generator.Print;
        Text = "Пошук вшир (BFS)";
        _steps = GraphSearchUtils.BFS(_vertices, matrix, 0).GetEnumerator();
    }
    else
    {
        GraphSearchUtils.OnDFSFinished += Generator.Print;
        Text = "Пошук вглиб (DFS)";
        _steps = GraphSearchUtils.DFS(_vertices, matrix, 0).GetEnumerator();
    }

}

private const int _textOffset = 10;

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

    foreach (var edge in _edges)
        DrawEdge(e.Graphics, edge);

    for (int i = 0; i < _vertices.Count; i++)
    {
        var vertex = _vertices[i];
        DrawVertex(e.Graphics, vertex);
    }

}

private void DrawVertex(Graphics g, Vertex vertex)
{
    var color = vertex.State switch
    {
        VertexState.Visited => Color.Gold,
        VertexState.Active => Color.LightCoral,
        VertexState.Closed => Color.LightGreen,
        _ => Color.LightGray,
    };
```

```
            g.FillEllipse(new SolidBrush(color),
                vertex.Point.X - Vertex.Radius,
                vertex.Point.Y - Vertex.Radius,
                Vertex.Radius * 2,
                Vertex.Radius * 2);

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

        private static void DrawEdge(Graphics g, Edge edge)
        {
            var color = edge.Visited ? Color.Red : Color.Black;

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

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

                        break;
                    }

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

                case EdgeType.SelfPointing:
                    DrawLine(g, edge.From.Point, edge.SelfLinkVertices[0], color);
                    DrawLine(g, edge.SelfLinkVertices[0], edge.SelfLinkVertices[1],
color);
                    DrawArrow(g, edge.SelfLinkVertices[1], edge.SelfLinkVertices[2],
color);
```

```
                break;
            }

        }

        private static void DrawArrow(Graphics g, PointF start, PointF end, Color
color)
        {
            Pen pen = new(color);
            g.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));

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

        private static void DrawLine(Graphics g, PointF start, PointF end, Color color)
        {
            using Pen pen = new(color);
            g.DrawLine(pen, start, end);
        }
    }

    public enum GraphTraversalStrategy
    {
        BFS,
        DFS
    }
```

src/Service/Generator.cs

```
namespace lab5;

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) * 0.005f - 0.15f;
        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();
    }

    public static void Print(int[,] matrix)
    {
        for (int i = 0; i < matrix.GetLength(0); i++)
        {
            for (int j = 0; j < matrix.GetLength(1); j++)
```

```
            {
                Console.Write(matrix[i, j] + " ");
            }
            Console.WriteLine();
        }
    }
}
```

src/Service/VertexFactory.cs

```
namespace lab5;

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 void Reset()
    {
        foreach (var node in _nodes)
            node.State = VertexState.New;
    }
}

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 VertexState State;

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

public enum Direction
{
    Right = 4,
    Down = 6,
    Left = 9,
    Up = 1
}

public enum VertexState
{
    New,
    Visited,
    Active,
    Closed
}
```

src/Service/EdgeFactory.cs

```
namespace lab5;

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 void Reset()
    {
        foreach (var edge in _links)
            edge.Visited = 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 bool Visited;


    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/GraphSearchUtils.cs

```
namespace lab5;

public static class GraphSearchUtils
{
    public static event Action? OnRedrawNeeded;
    public static event Action<int, int>? OnEdgeVisited;
    public static event Action<int[,]>? OnBFSFinished;
    public static event Action<int[,]>? OnDFSFinished;

    public static IEnumerable<Action> BFS(List<Vertex> vertices, int[,]
adjacencyMatrix, int startId)
    {
        System.Console.WriteLine("Обхід в ширину (BFS):");

        int n = vertices.Count;
        bool[] visited = new bool[n];
        int[] parent = new int[n];
        for (int i = 0; i < n; i++)
            parent[i] = -1;

        int[,] treeTraversalMatrix = new int[n, n];
        Queue<int> queue = new();
        visited[startId] = true;
        queue.Enqueue(startId);

        int j = 0;
        while (queue.Count > 0)
        {
            int u = queue.Dequeue();
            vertices[u].State = VertexState.Active;
            Console.WriteLine($"Вершина {vertices[u].Id} була відвідана {++j}-ою");
            yield return () => OnRedrawNeeded?.Invoke();
```

```
            for (int v = 0; v < n; v++)
            {
                if (adjacencyMatrix[u, v] == 1 && !visited[v])
                {
                    visited[v] = true;
                    parent[v] = u;
                    treeTraversalMatrix[u, v] = 1;
                    queue.Enqueue(v);

                    vertices[v].State = VertexState.Visited;
                    yield return () => OnEdgeVisited?.Invoke(u, v);
                }
            }

            vertices[u].State = VertexState.Closed;
            yield return () => OnRedrawNeeded?.Invoke();
        }

        Console.WriteLine("\nОбхід в ширину завершено.");
        Console.WriteLine("Матриця суміжності дерева обходу: ");
        OnBFSFinished?.Invoke(treeTraversalMatrix);
    }

    public static IEnumerable<Action> DFS(List<Vertex> vertices, int[,]
adjacencyMatrix, int startId)
    {
        System.Console.WriteLine("\nОбхід в глибину (DFS):");

        int n = vertices.Count;

        bool[] visited = new bool[n];
        int[] parent = new int[n];
        for (int i = 0; i < n; i++)
            parent[i] = -1;

        int[,] treeTraversalMatrix = new int[n, n];

        int prev = -1;
        int j = 0;
        // рекурсія
        IEnumerable<Action> DFS(int u)
        {
            visited[u] = true;
            Console.WriteLine($"Вершина {vertices[u].Id} відвідана {++j}-ою");
            if (prev != -1) vertices[prev].State = VertexState.Visited;
            vertices[u].State = VertexState.Active;
            prev = u;
            yield return () => OnRedrawNeeded?.Invoke();

            for (int v = 0; v < n; v++)
            {
                if (adjacencyMatrix[u, v] == 1 && !visited[v])
                {
```

```
                    parent[v] = u;
                    treeTraversalMatrix[u, v] = 1;

                    vertices[v].State = VertexState.Visited;
                    yield return () => OnEdgeVisited?.Invoke(u, v);

                    foreach (var action in DFS(v))
                        yield return action;
                }
            }

            vertices[u].State = VertexState.Closed;
            yield return () => OnRedrawNeeded?.Invoke();
        }

        foreach (var action in DFS(startId))
            yield return action;

        Console.WriteLine("\nОбхід в глибину завершено.");
        Console.WriteLine("Матриця суміжності дерева обходу: ");
        OnDFSFinished?.Invoke(treeTraversalMatrix);
    }
}
```

Program.cs (точка входу)

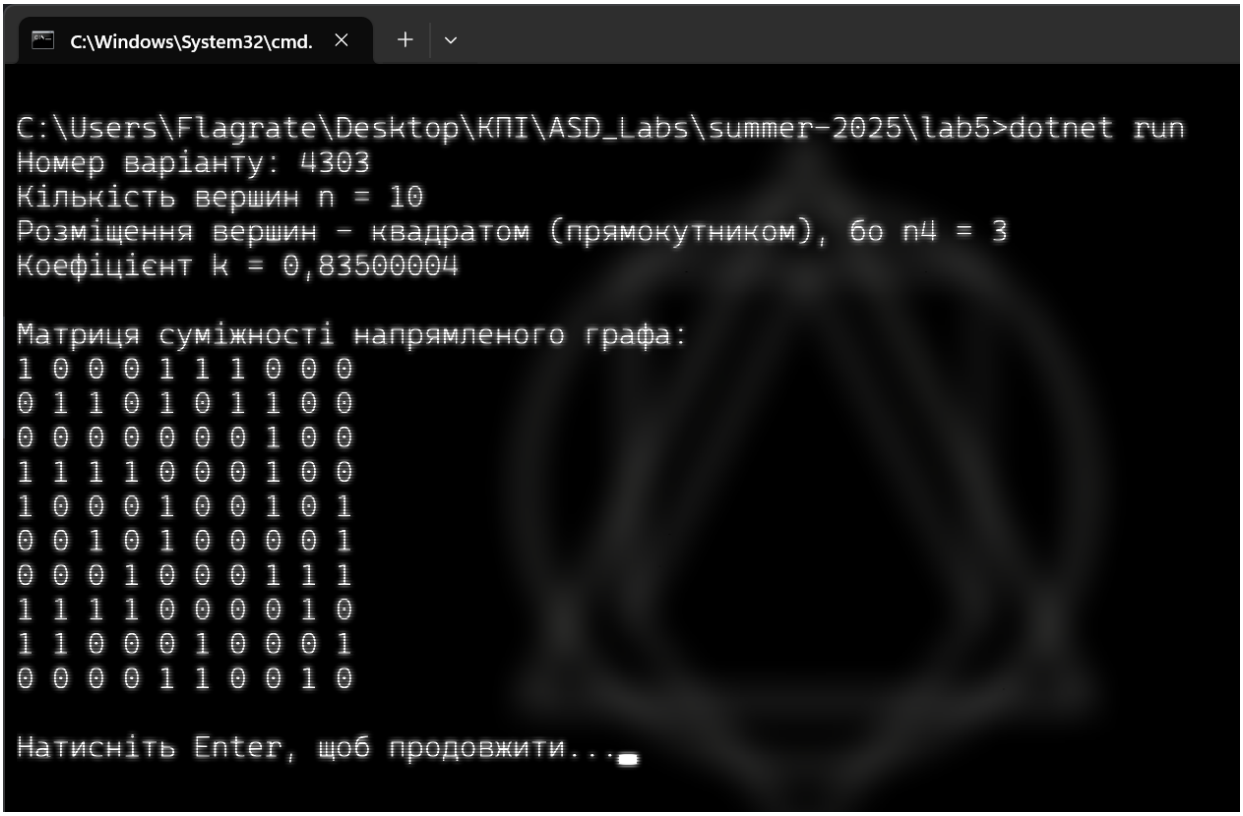
```
namespace lab5;

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

        Console.WriteLine("\nМатриця суміжності напрямленого графа:");
        Generator.Print(matrix);

        Generator.Await();

        VertexFactory vertexFactory = new(10);
        vertexFactory.CreateAll();

        EdgeFactory directedEdgeFactory = new(vertexFactory);
        directedEdgeFactory.CreateAll(matrix);

        ApplicationConfiguration.Initialize();
        Application.Run(new DirectedForm(
            vertexFactory.Vertices,
            directedEdgeFactory.Edges,
            matrix,
            GraphTraversalStrategy.BFS
```

```
        ));
        vertexFactory.Reset();
        directedEdgeFactory.Reset();
        Application.Run(new DirectedForm(
            vertexFactory.Vertices,
            directedEdgeFactory.Edges,
            matrix,
            GraphTraversalStrategy.DFS
        ));
    }
}
```

Примітка. Файл DirectedForm.Designer.cs – службовий файл, що необхідний для роботи інтерфейсу програмування WinForms.

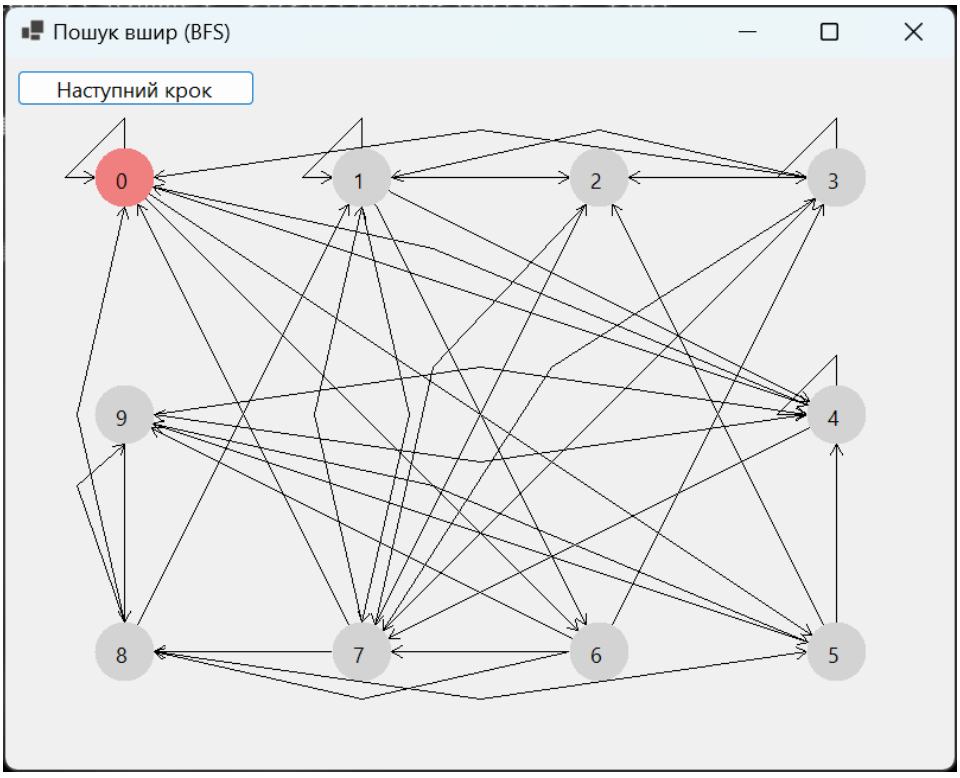
Тестування програми

Умови та матриця суміжності

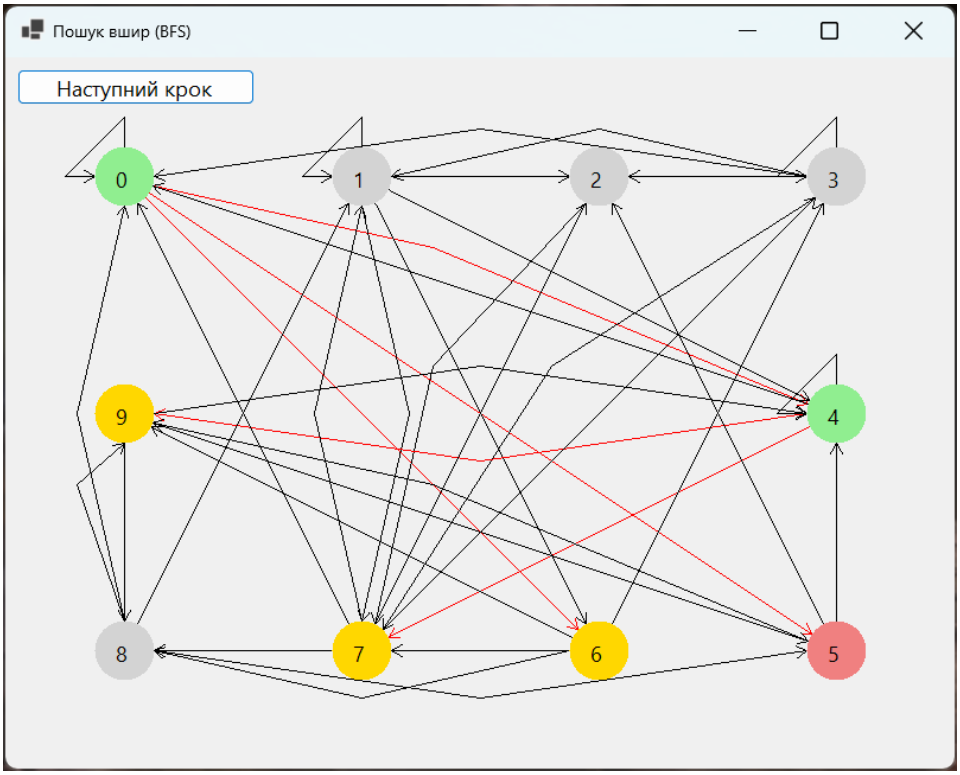


Обхід вшир (BFS)

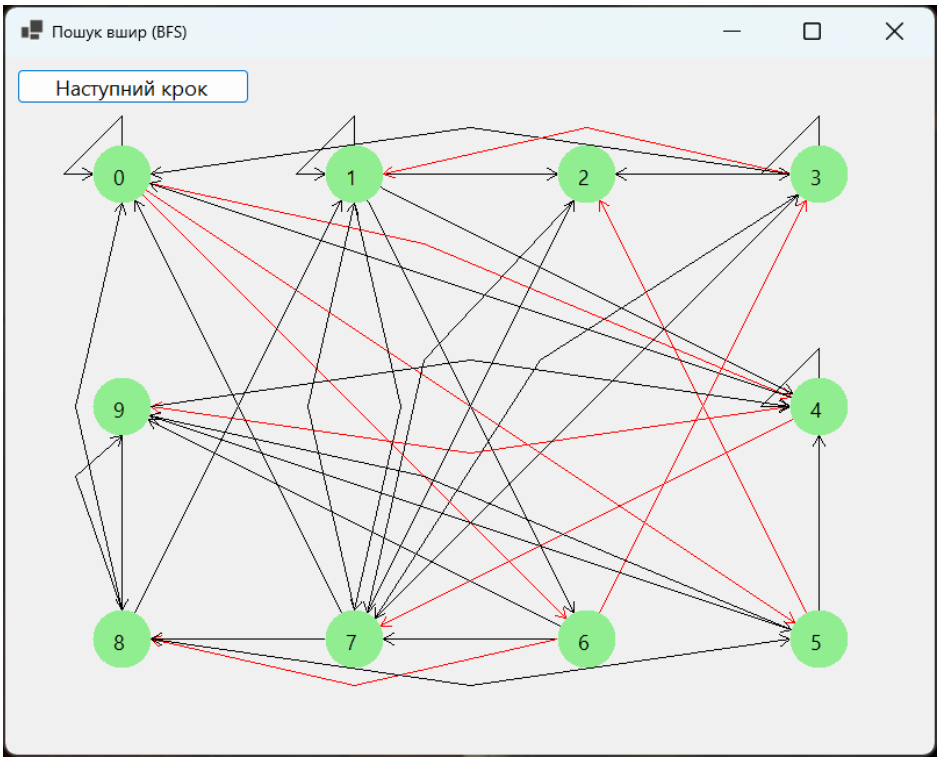
Початок обходу



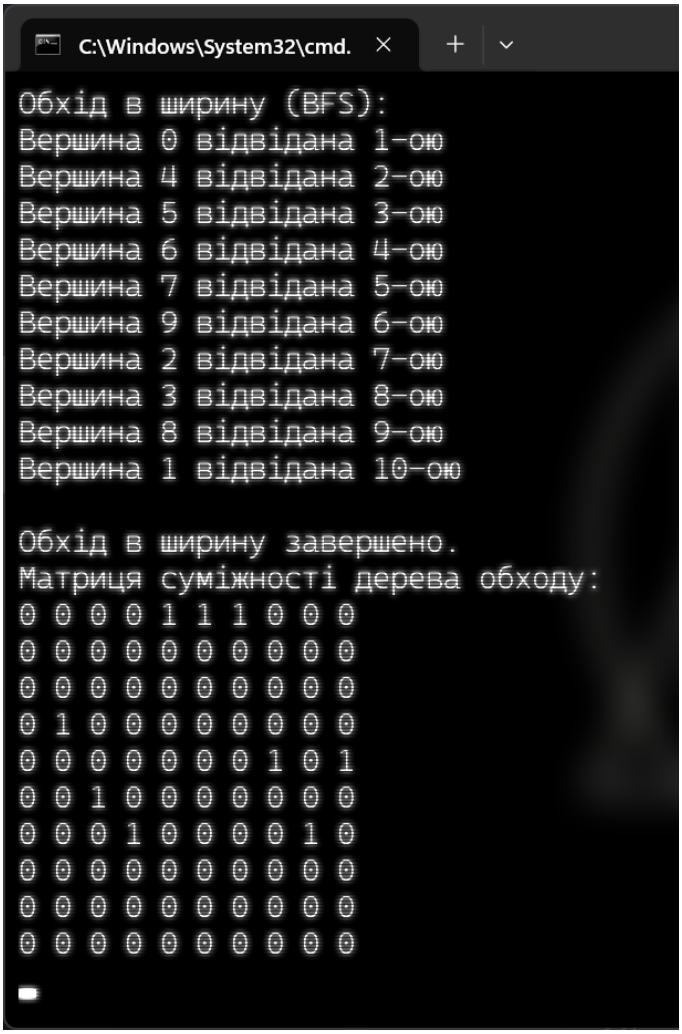
В процесі обходу



Кінець обходу

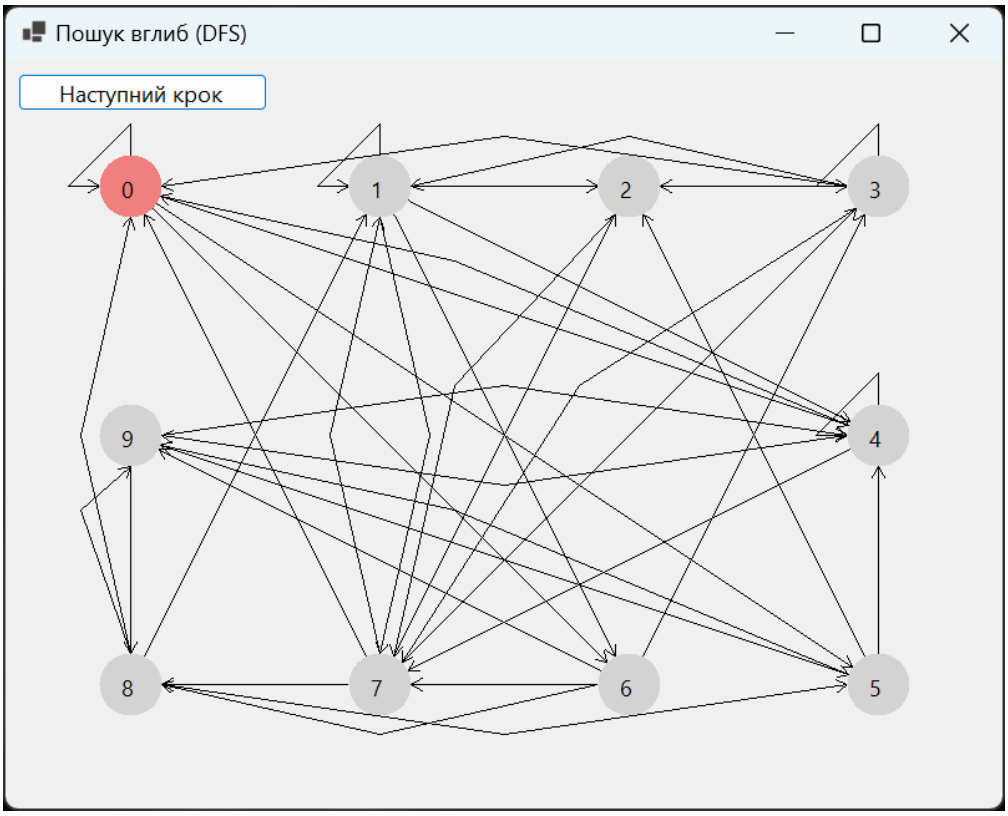


Результат обходу

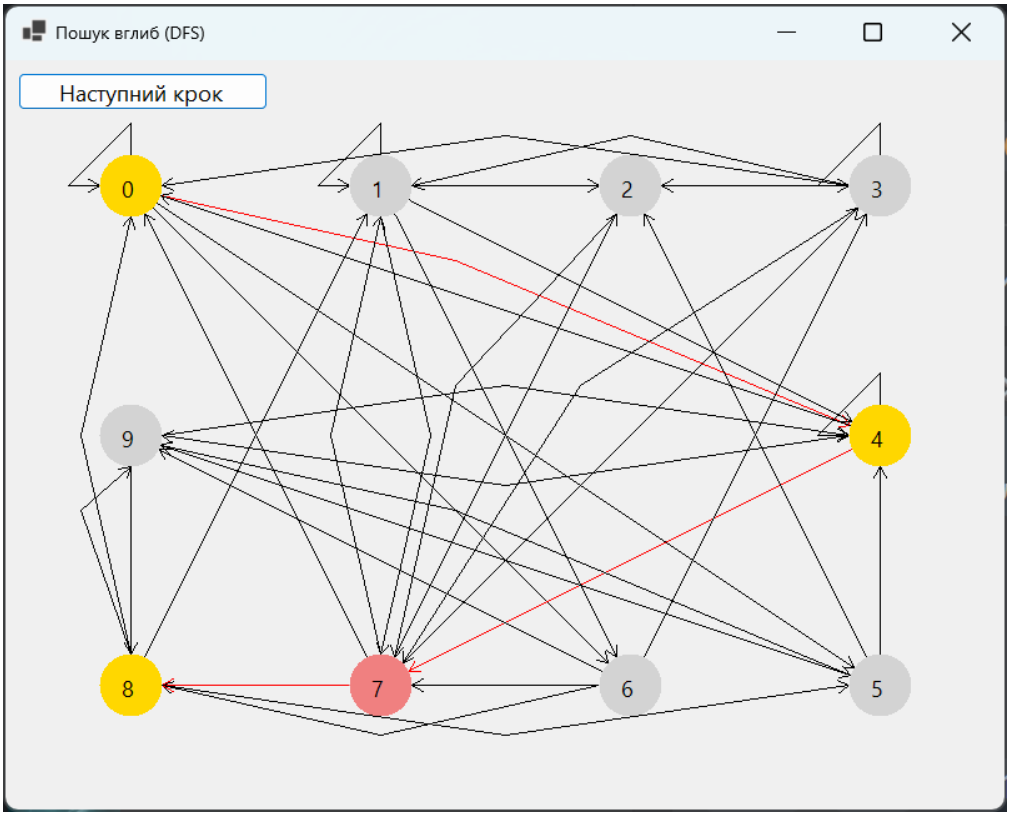


Обхід вглиб (DFS)

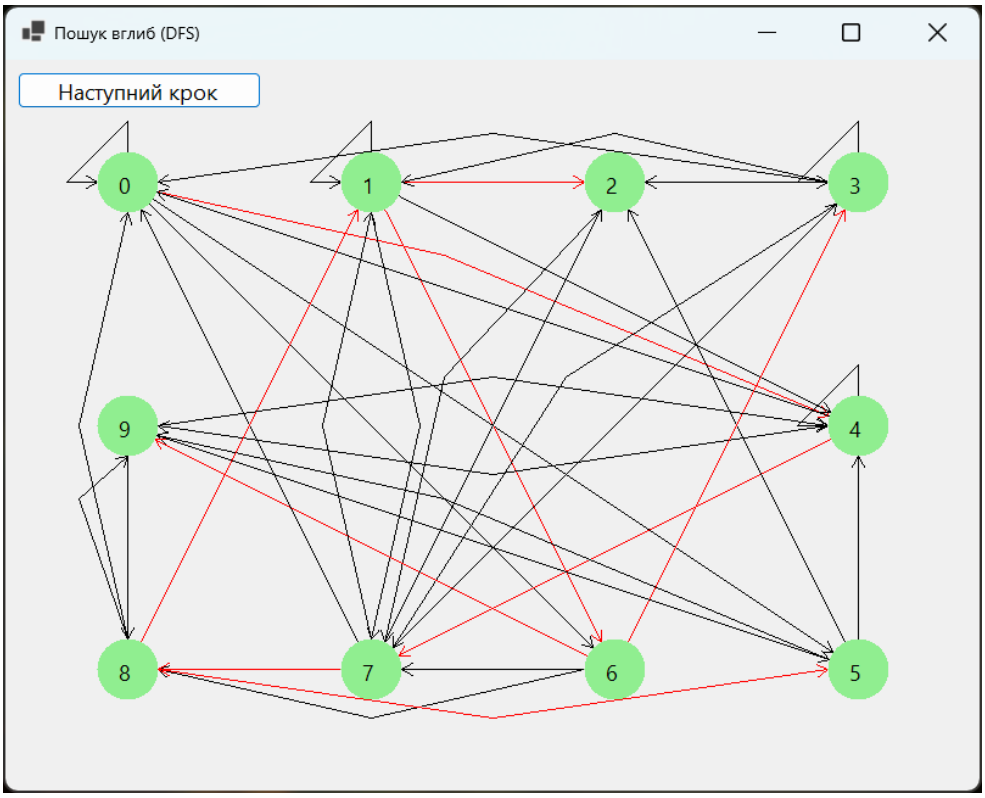
Початок обходу



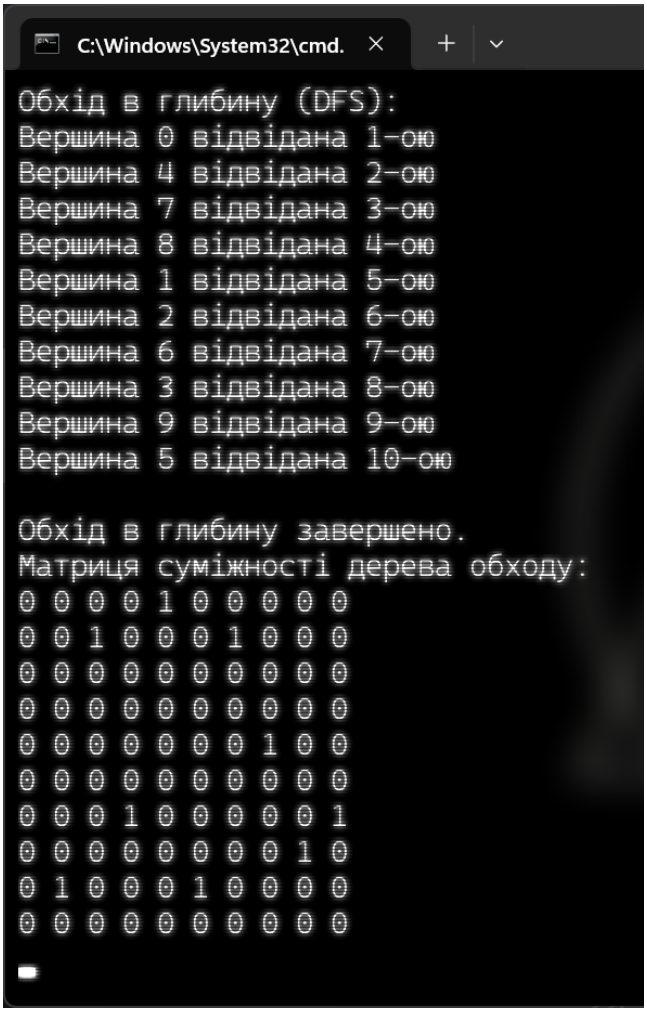
В процесі обходу



Кінець обходу



Результати обходу



Висновки

У цій лабораторній роботі було створено програму для обходу напрямленого графа та формування покрокового зображення у графічному вікні мовою програмування C# (платформа .NET, інтерфейс програмування графіки WinForms).

Обхід напрямленого графа було виконано за допомогою двох стратегій: пошук вшир (Breadth First Search, BFS) та пошуку вглиб (Depth First Search, DFS). Перший сканує спочатку усі доступні шляхи зі стартової вершини і переходить до першої доступної, другий же – «проходить» можливий маршрут до кінця і рекурсивно повертається до початку, поки всі вершини не будуть відвідані.

Для наочності виконання обходу, програма обладнана кнопкою для переходу на наступний крок. Таким чином, користувач може відслідкувати в довільному темпі роботу алгоритму обходу крок за кроком. При зміні статусу вершини показово змінюють колір (сірий = нова, жовтий = відвідана, червоний = відвідана й активна, зелений = закрита). Так само роблять і ребра (чорний = не відвідана, червона = відвідана).

Алгоритми обходу графів, такі як BFS (пошук у ширину) та DFS (пошук у глибину), мають широке застосування в реальному світі. BFS використовується для знаходження найкоротших шляхів у незважених графах, що корисно в навігаційних системах та соціальних мережах для пошуку користувачів на певній відстані. Також він застосовується в веб-краулерах для індексації сторінок. DFS ефективний для виявлення замкненостей у графах, топологічного сортування та вирішення головоломок, де потрібно дослідити всі можливі шляхи, наприклад, у розв'язанні лабіринтів. Обидва алгоритми є фундаментальними в комп'ютерних науках і мають критичне значення в багатьох прикладних задачах.