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 №6

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 зважений ненапрямлений граф із заданими параметрами так само, як у лабораторній роботі №3.
   Відмінність: коефіцієнт k = 1.0 - n3 * 0.01 - n4 * 0.005 - 0.05
2. Створити програму для знаходження мінімального кістяка за алгоритмом Краскала при n4 — парному і за алгоритмом Пріма — при непарному *(n4 = 3, тому, обрано алгоритм Пріма)*.
3. Під час обходу графа побудувати дерево його кістяка. У програмі дерево кістяка виводити покроково у процесі виконання алгоритм. *Це зроблено виділенням іншим кольором ребер та вершин графа.*

Варіант №3

Номер варіанту: 4303

Кількість вершин n = 10

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

Обраний алгоритм – алгоритм Пріма, бо n4 – непарне число

Коефіцієнт k = 0,935

Тексти програм

src/Forms/UndirectedGraph/UndirectedForm.cs

```
namespace lab6;

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

    private int _path;

    public UndirectedForm(
        List<Vertex> vertices,
        List<Edge> edges,
        int[,] undirMatrix,
        int[,] weights
    )
    {
        _vertices = vertices;
        _edges = edges;


        MatrixUtils.OnRedrawNeeded += Invalidate;
```

```
        MatrixUtils.OnEdgeAdded += (v1, v2) =>
        {
            Edge edge;
            try
            {
                edge = _edges.First(e => e.From.Id == v1 && e.To.Id == v2);

            }
            catch
            {
                edge = _edges.First(e => e.From.Id == v2 && e.To.Id == v1);
            }
            _path += edge.Weight;
            edge.InMST = true;
        };

        InitializeComponent();

        _steps = MatrixUtils.Prim(vertices, undirMatrix, weights).GetEnumerator();
        MatrixUtils.OnPrimFinished += () =>
            {
                Console.WriteLine("\nАлгоритм Пріма завершив роботу.");
                Console.WriteLine("Сума ваг ребер знайденого мінімального кістяка:
" + _path);
            };
    }

    private const int _textOffset = 10;

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

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

        foreach (var edge in _edges)
            DrawWeight(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.InMST ? Color.Red : Color.Black;

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

                    break;
                }

            case EdgeType.VisibilityObstructed:
                {
                    var (start, end) = edge.GetLineCoords();
                    DrawLine(g, start, edge.PolygonalLinkVertex, color);
                    DrawLine(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);
                DrawLine(g, edge.SelfLinkVertices[1], edge.SelfLinkVertices[2],
color);

                break;
        }

    }
```

```
private void DrawWeight(Graphics g, Edge edge)
{
    string weight = edge.Weight.ToString();
    SizeF size = g.MeasureString(weight, Font);
    PointF pos = new(0, 0);

    switch (edge.Type)
    {
        case EdgeType.Normal:
            var (start, end) = edge.GetLineCoords();
            pos = new(
                ((start.X + end.X) / 2) - (size.Width / 2),
                ((start.Y + end.Y) / 2) - (size.Height / 2)
            );

            break;

        case EdgeType.SelfPointing:
            return;

        case EdgeType.VisibilityObstructed:
            pos = new(
                edge.PolygonalLinkVertex.X - (size.Width / 2),
                edge.PolygonalLinkVertex.Y - (size.Height / 2)
            );

            break;
    }


    g.DrawRectangle(
        edge.InMST ? Pens.Red : Pens.Black,
        pos.X - 1,
        pos.Y - 1,
        size.Width + 1,
        size.Height + 2
    );

    g.FillRectangle(
        Brushes.White,
        pos.X,
        pos.Y,
        size.Width,
        size.Height
    );

    g.DrawString(
        weight,
        Font,
        edge.InMST ? Brushes.Red : Brushes.Black,
        pos.X, pos.Y
    );
}
```

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

src/Service/Generator.cs

```
namespace lab6;

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 - 0.05f;
        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();
}

private float[,] B()
{
    var matrix = new float[_n, _n];

    for (var i = 0; i < _n; i++)
    {
        for (var j = 0; j < _n; j++)
        {
            matrix[i, j] = (float)_random.NextDouble() * 2;
        }
    }

    return matrix;
}

private int[,] C(float[,] B, int[,] undirMatrix)
{
    var matrix = new int[_n, _n];

    for (int i = 0; i < _n; i++)
    {
        for (int j = 0; j < _n; j++)
        {
            matrix[i, j] = (int)MathF.Ceiling(
                B[i, j] * 100 * undirMatrix[i, j]
            );
        }
    }

    return matrix;
}

private int[,] D(int[,] C)
{
    var matrix = new int[_n, _n];
```

```
    for (int i = 0; i < _n; i++)
    {
        for (int j = 0; j < _n; j++)
        {
            matrix[i, j] = C[i, j] > 0 ? 1 : 0;
        }
    }

    return matrix;
}

private int[,] H(int[,] D)
{
    var matrix = new int[_n, _n];

    for (int i = 0; i < _n; i++)
    {
        for (int j = 0; j < _n; j++)
        {
            matrix[i, j] = D[i, j] == D[j, i] ? 1 : 0;
        }
    }

    return matrix;
}

private int[,] UpperTriangle()
{
    var matrix = new int[_n, _n];

    for (int i = 0; i < _n; i++)
    {
        for (int j = 0; j < _n; j++)
        {
            matrix[i, j] = i < j ? 1 : 0;
        }
    }

    return matrix;
}

public int[,] Weights(int[,] undirMatrix)
{
    int[,] c = C(B(), undirMatrix);
    int[,] d = D(c);
    int[,] h = H(d);
    int[,] tri = UpperTriangle();

    var matrix = new int[_n, _n];

    for (int i = 0; i < _n; i++)
    {
```

```
            for (int j = 0; j < _n; j++)
            {
                int elem = 0;

                if (i != j)
                {
                    elem = (d[i, j] + h[i, j] * tri[i, j]) * c[i, j];

                    if (elem == 0) elem = -1;
                }

                matrix[i, j] = elem;
            }
        }

        return MatrixUtils.ToSymmetrical(matrix);
    }
}
```

src/Service/VertexFactory.cs

```
    namespace lab6;

    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 lab6;

public class EdgeFactory(VertexFactory vertexFactory, int[,] weights)
{
    private readonly VertexFactory _nodeFactory = vertexFactory;
    private readonly List<Edge> _edges = [];
    public List<Edge> Edges => _edges;
    private int[,] _weights = weights;

    public void CreateAll(int[,] matrix)
    {
```

```
        _edges.Clear();
        var copy = (int[,])matrix.Clone();

        for (int i = 0; i < copy.GetLength(0); i++)
        {
            for (int j = 0; j < copy.GetLength(1); j++)
            {
                if (copy[i, j] == 1)
                {
                    if (copy[j, i] == 1)
                    {
                        copy[j, i] = 2;
                    }
                    _edges.Add(new Edge(
                        _nodeFactory.Vertices[i], _nodeFactory.Vertices[j],
_weights[i, j]
                    ));
                }
            }
        }
    }

    public void Reset()
    {
        foreach (var edge in _edges)
            edge.InMST = false;
    }
}

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


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

    public Edge(Vertex from, Vertex to, int weight)
    {
        From = from;
        To = to;
        Weight = weight;


        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
                )
            );
        }
    }
    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}";

}

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

src/Service/MatrixUtils.cs

```
namespace lab6;

public static class MatrixUtils
{
    public static event Action? OnRedrawNeeded;
    public static event Action<int, int>? OnEdgeAdded;
    public static event Action? OnPrimFinished;

    public static IEnumerable<Action> Prim(
        List<Vertex> vertices,
        int[,] adjacencyMatrix,
        int[,] weights
    )
    {
        Console.WriteLine("\nАлгоритм Пріма:");

        int n = vertices.Count;

        // чи в кістяку
        bool[] inMST = new bool[n];

        // мінімальна вага ребра для додавання
        int[] key = new int[n];

        // індекс "батьківської" вершини
        int[] parent = new int[n];

        // ініціалізація
        for (int i = 0; i < n; i++)
        {
            key[i] = int.MaxValue;
            parent[i] = -1;
        }

        for (int start = 0; start < n; start++)
        {
            if (inMST[start])
                continue;
```

```
            key[start] = 0;

            for (int count = 0; count < n; count++)
            {
                int u = -1;
                int minKey = int.MaxValue;
                for (int v = 0; v < n; v++)
                {
                    if (!inMST[v] && key[v] < minKey)
                    {
                        minKey = key[v];
                        u = v;
                    }
                }

                // якщо всі вершини вже в кістяку
                if (u == -1)
                    break;

                inMST[u] = true;

                if (parent[u] != -1)
                {
                    int pu = parent[u];
                    Console.WriteLine($"Додано ребро {vertices[pu].Id} ->
{vertices[u].Id} (вага = {weights[pu, u]})");
                    OnEdgeAdded?.Invoke(pu, u);
                }

                for (int v = 0; v < n; v++)
                {
                    if (adjacencyMatrix[u, v] == 1 && !inMST[v])
                    {
                        int w = weights[u, v];
                        if (w >= 0 && w < key[v])
                        {
                            key[v] = w;
                            parent[v] = u;

                        }
                    }
                }

                vertices[u].State = VertexState.Closed;
                Console.WriteLine($"Вершина {vertices[u].Id} додана до мінімального
кістяка");
                yield return () => OnRedrawNeeded?.Invoke();
            }
        }

        OnPrimFinished?.Invoke();
    }
```

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

        return matrix;
    }

    public static int[,] Copy(int[,] matrix) => (int[,])matrix.Clone();
    public static void PrintMatrix(int[,] matrix) => Print(matrix, 1);
    public static void PrintWeights(int[,] matrix) => Print(matrix, 4);

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

                if (elem == -1)
                {
                    Console.Write($"+inf ");
                }
                else
                {
                    Console.Write($"{new string(' ', order -
elem.ToString().Length)}{elem} ");
                }

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

Program.cs (точка входу)

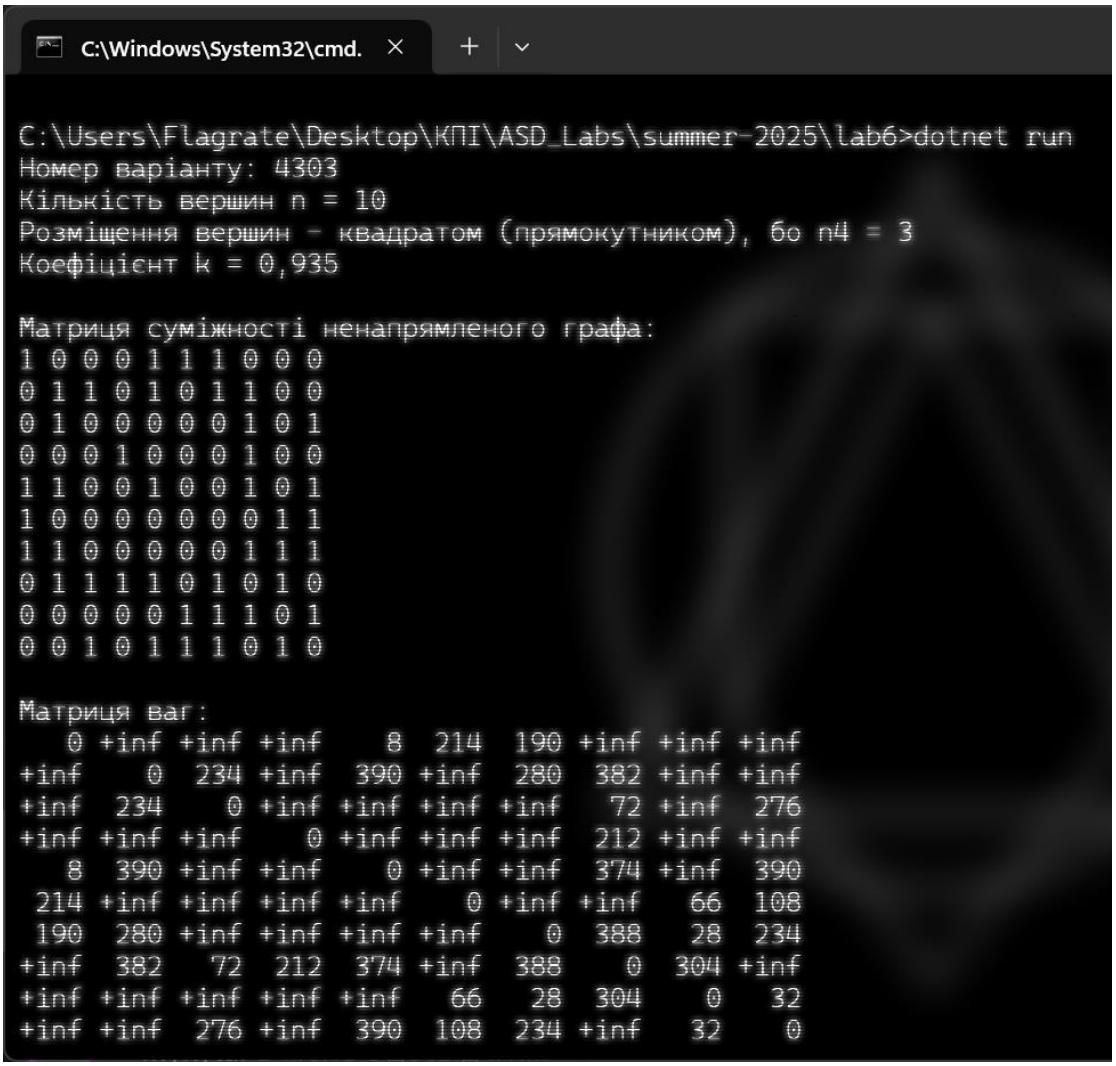
```
namespace lab6;

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

```
        int[,] undirMatrix = MatrixUtils.ToSymmetrical(dirMatrix);

        Console.WriteLine("\nМатриця суміжності ненапрямленого графа:");
        MatrixUtils.PrintMatrix(undirMatrix);

        int[,] weights = generator.Weights(undirMatrix);
        Console.WriteLine("\nМатриця ваг:");
        MatrixUtils.PrintWeights(weights);

        Generator.Await();

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

        EdgeFactory undirectedEdgeFactory = new(vertexFactory, weights);
        undirectedEdgeFactory.CreateAll(undirMatrix);

        ApplicationConfiguration.Initialize();
        Application.Run(new UndirectedForm(
            vertexFactory.Vertices,
            undirectedEdgeFactory.Edges,
            undirMatrix,
            weights
        ));
    }
}
```

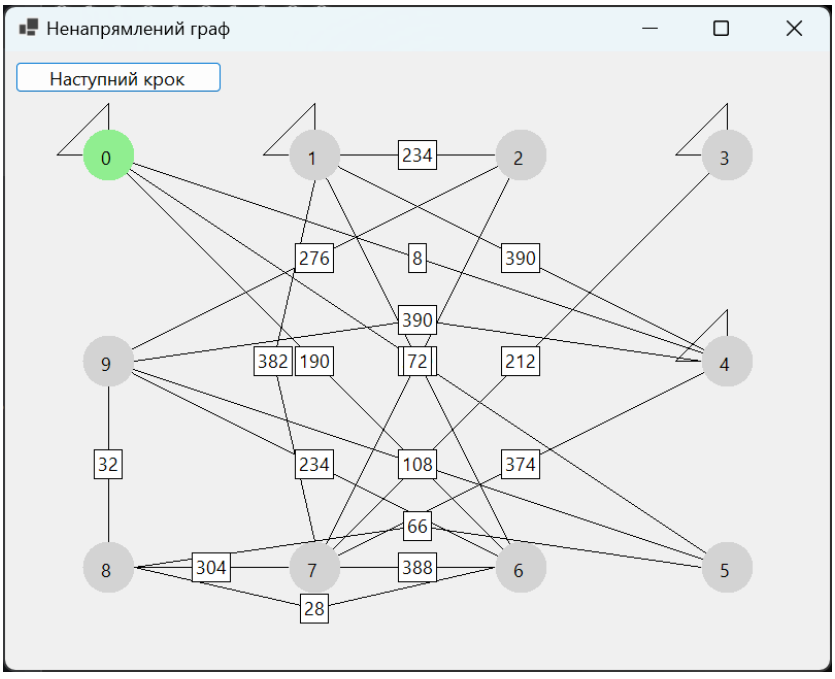
Примітка. Файл UndirectedForm.Designer.cs – службовий файл, що необхідний для роботи інтерфейсу програмування WinForms.

Тестування програми

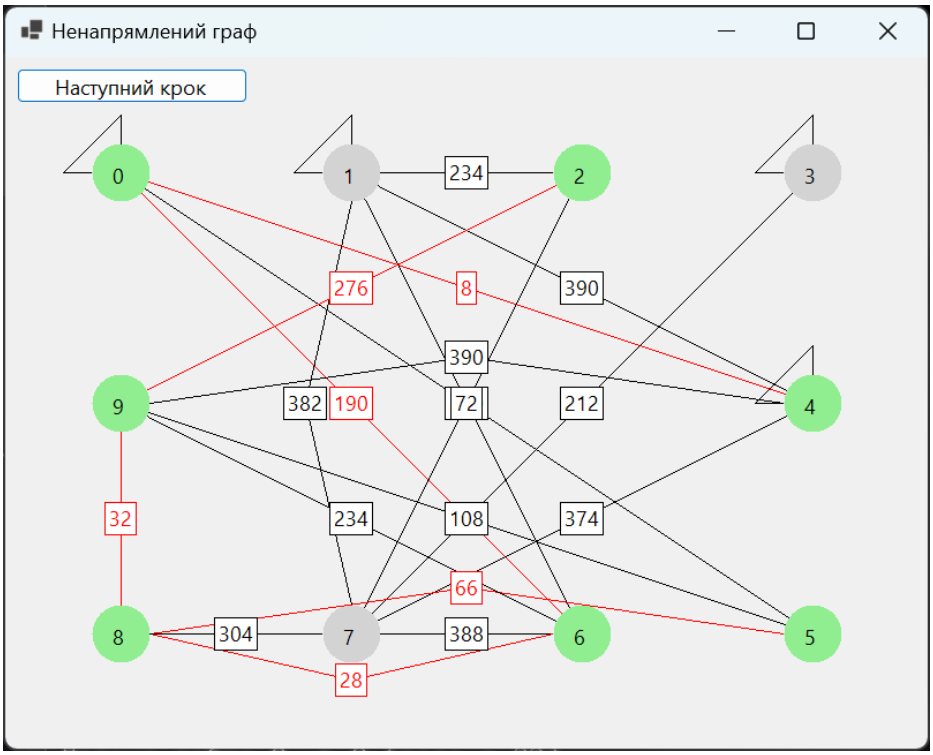
Умови, матриці суміжності та ваг



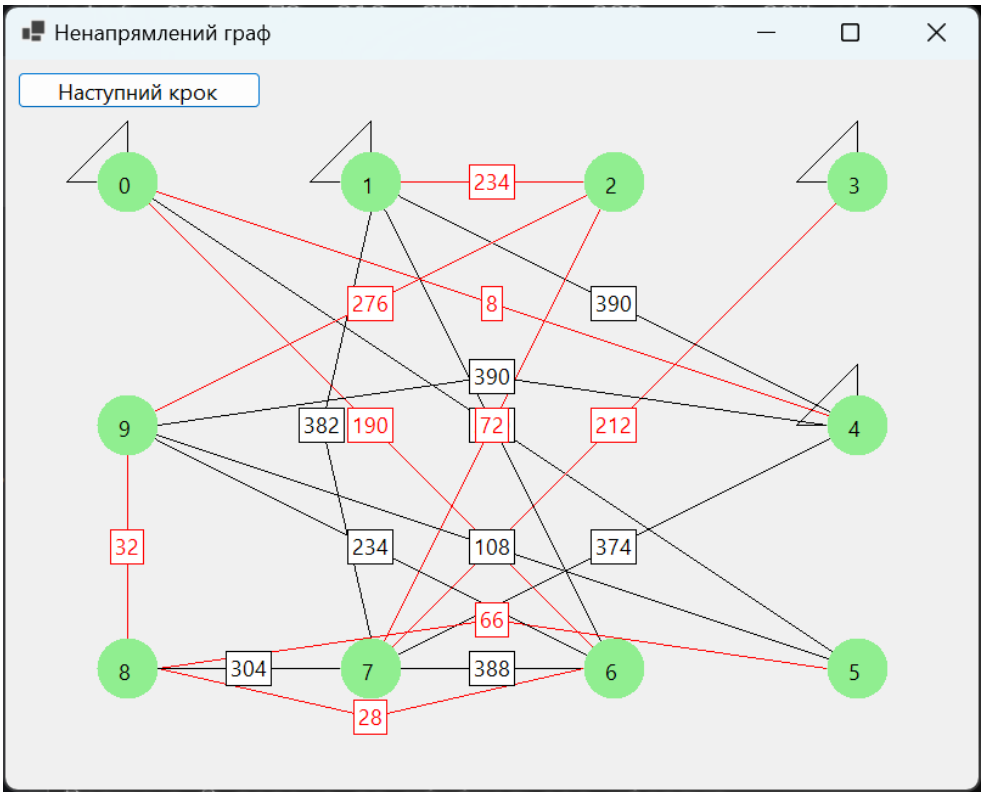
Початок алгоритму Пріма



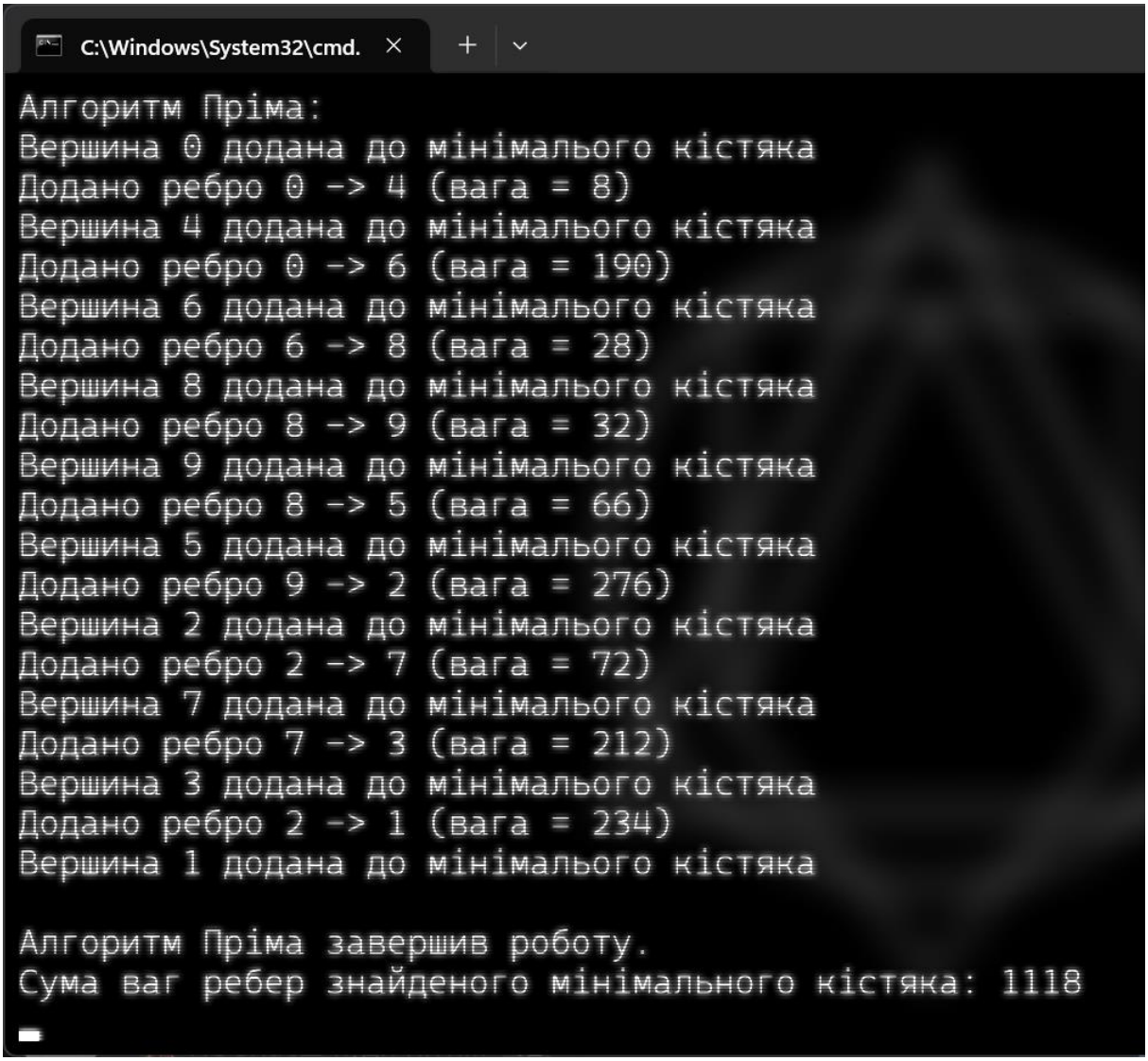
В процесі алгоритму



Кінець алгоритму



Результат роботи



Висновки

У цій лабораторній роботі було відтворено алгоритм Пріма для пошуку мінімального кістяка ненапрямленого графа та формування покрокового зображення у графічному вікні мовою програмування C# (платформа .NET, інтерфейс програмування графіки WinForms).

Алгоритм Пріма знаходить мінімальний кістяк зваженого неорієнтованого графа. Починаючи з довільної вершини, він поступово будує кістяк кожного разу додаючи найменше ребро, що з’єднує вже включену вершину з невключеною. Для цього зберігаються найменші ваги ребер до кожної ще не включеної вершини. Алгоритм повторюється, поки всі вершини не буде охоплено. Його складність – *O(E * log V).*

Для наочності виконання обходу, програма обладнана кнопкою для переходу на наступний крок. Таким чином, користувач може відслідкувати в довільному темпі роботу алгоритму Пріма крок за кроком. При зміні статусу

вершини та ребра змінюють колір (вершина, додана до кістяка, загорається зеленим, ребро – разом із вагою – червоним).

Алгоритм Пріма та інші алгоритми пошуку мінімального кістяка (як-от Краскала) широко застосовуються в інженерії та інформатиці. Їх використовують для проєктування мінімально витратних мереж: електромереж, доріг, оптоволоконних ліній. Наприклад, при побудові інтернет-мережі між містами алгоритм Пріма допомагає з'єднати всі вузли з мінімальними витратами. Також вони застосовуються в кластеризації в машинному навчанні, де алгоритм пошуку мінімального кістяка допомагає об'єднати подібні об'єкти. У комп'ютерній графіці - для генерації лабіринтів або побудови геометричних об'єктів. Загалом, ці алгоритми критично важливі всюди, де треба ефективно з'єднати елементи системи з мінімальними витратами.