Ministerul Educației și Cercetării al Republicii Moldova

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică și Microelectronică

Raport

Lucrarea de laborator nr. 4

Disciplina: **Analiza si proiectarea algortimilor**

Tema: **Metoda programarii dinamice**

A efectuat:	Student grupa TI-231 FR	Apareci Aurica
A verificat:	Asistent universitar	Andrievschi-Bagrin Veronica

Chișinău 2025

Cuprins

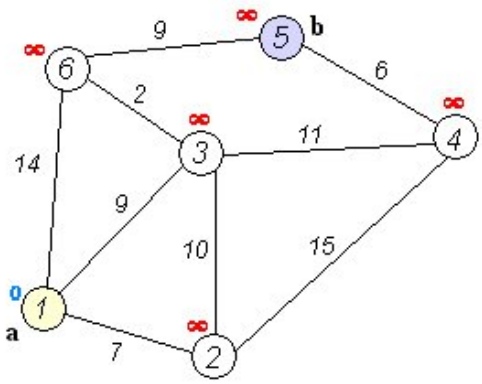
1. Cadru teoretic

Tema: Metoda programării dinamice

Sarcina (conform variantei): Analiza și implementarea algoritmilor de programare dinamică

Programarea dinamică este o metodă de rezolvare a unor probleme de informatică în care se cere de regulă determinarea unei valori maxime sau minime, sau numărarea elementelor unei mulțimi. Similar cu metoda Divide et Impera, problema se împarte în subprobleme:

- de aceeași natură cu problema inițială;
- de dimensiuni mai mici;
- spre deosebire de Divide et Impera, problemele nu mai sunt independente, ci se suprapun – *probleme superpozabile*!
- rezolvarea optimă a problemei inițiale depinde de rezolvarea optimă a subproblemelor – *principiul optimalității*!



2. Listingul programului

```
//se utilizeaza urmatoarea biblioteca: GitHub - dotnet/BenchmarkDot-
Net: Powerful .NET library for benchmarking versiunea: 0.13.2
using BenchmarkDotNet.Running;
using Lab_3_utils;

namespace Laboratorul_4
{
    internal class Program
    {
        static void Main(string[] args)
        {
            //run this in Release mode
            BenchmarkRunner.Run<Benchmark2>();
        }
    }
}
```

```
using Lab_3_utils;
namespace Laboratorul_4
{
    internal class Algs2
    {
        //dijkstra algorthm
        public static void Dijkstra(Graf graph, int start)
        {
            int[] dist = new int[graph.Noduri.Count];
            int[] prev = new int[graph.Noduri.Count];
            bool[] visited = new bool[graph.Noduri.Count];
            for (int i = 0; i < graph.Noduri.Count; i++)
            {
                dist[i] = int.MaxValue;
                prev[i] = -1;
                visited[i] = false;
            }
            dist[start] = 0;
            for (int i = 0; i < graph.Noduri.Count; i++)
            {
                int u = MinDistance(dist, visited);
                visited[u] = true;
                foreach (var muchie in graph.Muchii)
                    if (muchie.Start == u)
                    if (!visited[muchie.End] && dist[u] != int.Max-
Value && dist[u] + muchie.Cost < dist[muchie.End])
                        {
           dist[muchie.End] = dist[u] + muchie.Cost;
           prev[muchie.End] = u;
                        }
            }
```

```
   PrintSolution(dist, prev, start);
        }
        //Floyd algorithm
        public static void Floyd(Graf graph)
        {
            int[,] dist = new int[graph.Noduri.Count, graph.Noduri.Count];
            int[,] prev = new int[graph.Noduri.Count, graph.Noduri.Count];
            for (int i = 0; i < graph.Noduri.Count; i++)
                for (int j = 0; j < graph.Noduri.Count; j++)
                {
                    dist[i, j] = int.MaxValue;
                    prev[i, j] = -1;
                }
            for (int i = 0; i < graph.Noduri.Count; i++)
                dist[i, i] = 0;
            foreach (var muchie in graph.Muchii)
            {
                dist[muchie.Start, muchie.End] = muchie.Cost;
                prev[muchie.Start, muchie.End] = muchie.Start;
            }
            for (int k = 0; k < graph.Noduri.Count; k++)
                for (int i = 0; i < graph.Noduri.Count; i++)
                    for (int j = 0; j < graph.Noduri.Count; j++)
                        if (dist[i, k] != int.MaxValue && dist[k, j] != int.MaxValue && dist[i, k] + dist[k, j]
< dist[i, j])
                        {
```

```
   using BenchmarkDotNet.Attributes;

namespace Lab_3_utils
{
    [MemoryDiagnoser]
    public class Benchmark
    {
        Graf Graph { get; set; }

        [GlobalSetup]
        public void Setup()
        {
            //Graph = GraphGenerator.Defav(500);
            //Graph = GraphGenerator.Fav(500);
            Graph = GraphGenerator.Med(500);
        }

        [Benchmark]
        public void Prims()
        {
            Algs.Prim(Graph);
        }
        [Benchmark]
        public void Kruskal()
        {
            Algs.Kruskal(Graph);
        }
    }
}
```

```
namespace Lab_3_utils
{
    public class DisjointSet
    {
        public List<int> Nodes { get; set; }
        public DisjointSet(int node)
        {
            Nodes = new List<int>();
            Nodes.Add(node);
        }
        public bool Contains(int node)
        {
            return Nodes.Contains(node);
        }
        public void Merge(DisjointSet set)
        {
            Nodes.AddRange(set.Nodes);
        }
    }
}
```

```
   PrintSolution(dist, prev, start);
        }
        //Floyd algorithm
        public static void Floyd(Graf graph)
        {
            int[,] dist = new int[graph.Noduri.Count,
graph.Noduri.Count];
            int[,] prev = new int[graph.Noduri.Count,
graph.Noduri.Count];
            for (int i = 0; i < graph.Noduri.Count; i++)
                for (int j = 0; j < graph.Noduri.Count; j++)
                {
                    dist[i, j] = int.MaxValue;
                    prev[i, j] = -1;
                }
            for (int i = 0; i < graph.Noduri.Count; i++)
                dist[i, i] = 0;
            foreach (var muchie in graph.Muchii)
            {
                dist[muchie.Start, muchie.End] = muchie.Cost;
                prev[muchie.Start, muchie.End] = muchie.Start;
            }
            for (int k = 0; k < graph.Noduri.Count; k++)
                for (int i = 0; i < graph.Noduri.Count; i++)
                    for (int j = 0; j < graph.Noduri.Count; j++)
                        if (dist[i, k] != int.MaxValue && dist[k,
j] != int.MaxValue && dist[i, k] + dist[k, j] < dist[i, j])
                        {
                            dist[i, j] = dist[i, k] + dist[k, j];
                            prev[i, j] = prev[k, j];
                        }
            PrintSolution(dist);
        }

        private static void PrintSolution(int[,] dist)
        {
            for (int i = 0; i < dist.GetLength(0); ++i)
            {
                for (int j = 0; j < dist.GetLength(1); ++j)
                {
                    if (dist[i, j] == int.MaxValue)
                    {
                        Console.Write("I ");
                    }
                    else
                    {
                        Console.Write(dist[i, j] + " ");
                    }
                }
                Console.WriteLine();
            }
        }
```

```
private static int MinDistance(int[] dist, bool[] visited)
        {
            int min = int.MaxValue;
            int min_index = -1;
            for (int i = 0; i < dist.Length; i++)
            {   if (!visited[i] && dist[i] <= min)
                {
                    min = dist[i];
                    min_index = i;
                }
            }
            return min_index;
        }
private static void PrintSolution(int[] dist, int[] prev, int start)
        {
            Console.WriteLine("Vertex\tDistance\tPath");
            for (int i = 0; i < dist.Length; i++)
            {   if (i != start)
                {
                    Console.Write(start + " -> " + "\t\t" + start);
                    PrintPath(prev, i);
                    Console.WriteLine();
                }
            }
        }
private static void PrintPath(int[] prev, int i)
        {   if (prev[i] == -1)
                return;
            PrintPath(prev, prev[i]);
            Console.Write(" -> " + i);
        }
    }
}
```

```
using BenchmarkDotNet.Attributes;
using Lab_3_utils;
namespace Laboratorul_4
{
    [MemoryDiagnoser]
    public class Benchmark2
    {
        Graf Graph { get; set; }
        [GlobalSetup]
        public void Setup()
        { Graph = GraphGenerator.Defav(1000); }
        [Benchmark]
        public void DijkstraBench()
        {  Algs2.Dijkstra(Graph,0);  }
        [Benchmark]
        public void FloydBench()
        {  Algs2.Floyd(Graph);   }
    }
}
```

```
namespace Lab_3_utils
{
    public class Graf
    {
        public List<int> Noduri { get; set; }
        public List<Muchie> Muchii { get; set; }
        public Graf()
        {   Noduri = new List<int>();
            Muchii = new List<Muchie>();  }
        public void LoadGraph(int[,] arr)
        {
            for (int i = 0; i < arr.GetLength(0); i++)
            {
                for (int j = 0; j < arr.GetLength(1); j++)
                {
                    if (arr[i, j] != 0)
                    {
                        if (!Noduri.Contains(i))
                            Noduri.Add(i);
                        if (!Noduri.Contains(j))
                            Noduri.Add(j);
                        Muchii.Add(new Muchie(i, j, arr[i, j]));
                    }
                }
            }
        }
        public int[,] ToMatrix()
        {
            int[,] arr = new int[Noduri.Count, Noduri.Count];
            foreach (var muchie in Muchii)
            {   arr[muchie.Start, muchie.End] = muchie.Cost;  }
            return arr;
        }
 public static int[,] ReadFromFile(string path)
        {
            StreamReader rd = new StreamReader(path);
            string line = rd.ReadLine();
            int n = int.Parse(line.Split(' ')[0]);
            int v = int.Parse(line.Split(' ')[1]);
            int[,] arr = new int[n+1,n+1];
            for (int i = 0; i < v; i++)
            {
                line = rd.ReadLine();
                int start = int.Parse(line.Split(' ')[0]);
                int end = int.Parse(line.Split(' ')[1]);
                int cost = int.Parse(line.Split(' ')[2]);
                arr[start, end] = cost;
            }
            return arr;
        }
    }
}
```

```
namespace Lab_3_utils
{
    public class GraphGenerator
    {   //fav - 1 - graph with n-1 edges
        //med - 2 - graph with n(n-1)/4
        //defav - 3 - graph with n(n-1)/2
        public static Graf Fav(int nodes)
        {   Graf graph = new Graf();
            Random rnd = new Random();
            for (int i = 0; i < nodes; i++)
            {
                graph.Noduri.Add(i);
            }
            for (int i = 0; i < nodes - 1; i++)
            {
           graph.Muchii.Add(new Muchie(i, i + 1, rnd.Next(1, 100)));
            }
            return graph;
        }
        public static Graf Med(int nodes)
        {   Graf graph = new Graf();
            Random rnd = new Random();
            for (int i = 0; i < nodes; i++)
            {
                graph.Noduri.Add(i);
            }
            for (int i = 0; i < nodes; i++)
            {
                for (int j = i + 1; j < nodes; j++)
                {
               if (rnd.Next(0, 2) == 1)// 50% chance of adding
               graph.Muchii.Add(new Muchie(i, j, rnd.Next(1, 100)));
                }
            }
            return graph;
        }
        public static Graf Defav(int nodes)
        {   Graf graph = new Graf();
            Random rnd = new Random();
            for (int i = 0; i < nodes; i++)
            {
                graph.Noduri.Add(i);
            }
            for (int i = 0; i < nodes; i++)
            {
                for (int j = i + 1; j < nodes; j++)
                {
               graph.Muchii.Add(new Muchie(i, j, rnd.Next(1, 100)));
                }
            }
            return graph;
        }
    }
}
```

```
namespace Lab_3_utils
{
    public class Muchie:IComparable<Muchie>
    {
        public int Start { get; set; }
        public int End { get; set; }
        public int Cost { get; set; }
        public Muchie(int start, int end, int cost)
        {
            Start = start;
            End = end;
            Cost = cost;
        }
        public override string ToString()
        {
            return $"({Start},{End})\t{Cost}";
        }
        public int CompareTo(Muchie? other)
        {
            if (other == null)
                return 1;
            else
                return this.Cost.CompareTo(other.Cost);
        }
    }
```

3. Cazuri de testare

N – Numărul de noduri in graf | V – Numărul muchii din care este alcătuit graful

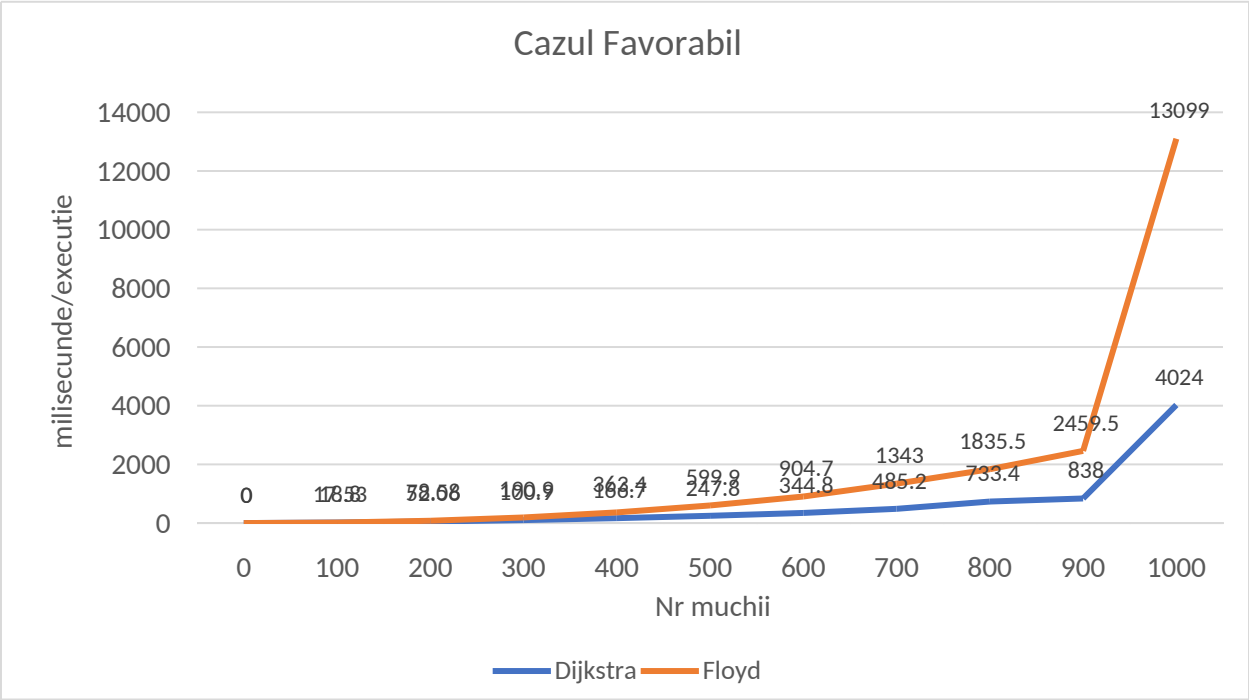
Algoritmii au fost testați grafuri generate din V=N–1 muchii pentru cazul favorabil

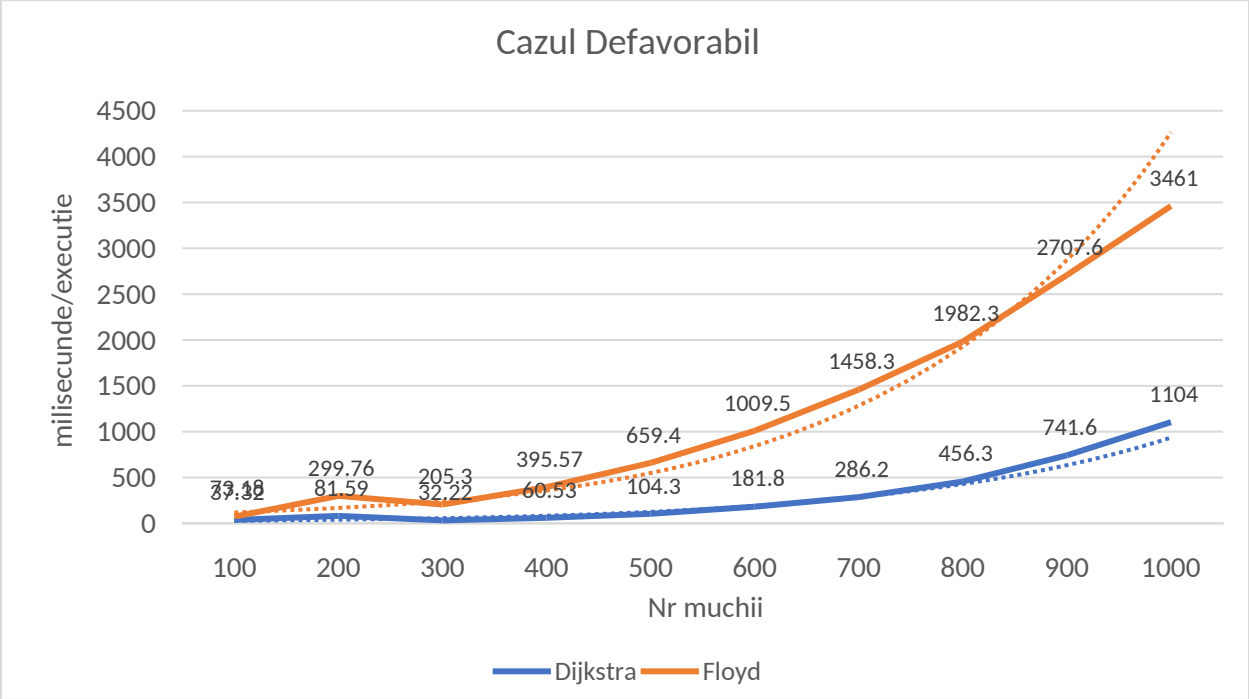
Algoritmii au fost testați grafuri generate din V=N–1 muchii pentru cazul favorabil

Tabelul valorilor obtinute

Cazul Favorabil										
	100	200	300	400	500	600	700	800	900	1000
Dijkstra	18.8	52.06	100.9	166.7	247.8	344.8	485.2	733.4	838	4024
Floyd	17.53	78.58	190.9	363.4	599.9	904.7	1,343.00	1,835.50	2,459.50	13099
Cazul Defavorabil										
	100	200	300	400	500	600	700	800	900	1000
Dijkstra	37.32	81.59	32.22	60.53	104.3	181.8	286.2	456.3	741.6	1104
Floyd	73.18	299.76	205.3	395.57	659.4	1,009.5	1,458.30	1,982.30	2,707.60	3461

Analiza valorilor obtinute





4. Concluzii

Algoritmul Floyd este un algoritm pentru găsirea celei mai scurte căi între toate perechile de noduri dintr-un grafic ponderat. Acest algoritm funcționează atât pentru graficele ponderate direcționate, cât și pentru cele nedirijate. Datorita celor 3 cicluri de repetare folosite acesta prezinta o complexitate de timp $O(n^3)$.

Algoritmul lui Dijkstra ne permite să găsim cea mai scurtă cale între oricare două noduri ale unui graf. Acesta diferă de arborele minim care se întinde, deoarece cea mai scurtă distanță dintre două noduri ar putea să nu includă toate nodurile grafului.

Are o complexitate de timp $O(V^2)$ folosind reprezentarea matricei adiacente a grafului. Complexitatea de timp poate fi redusă la $O((V+E) \log V)$ utilizând reprezentarea listei adiacente a graficului, unde E este numărul de muchii din grafic, iar V este numărul de noduri din grafic.