Дискретна математика. Теорія графів

Лабораторна робота №5(1)

Алгоритми Дейкстри та Флойда-Уоршола

Ціль

Розглянути задачу пошуку найкоротших шляхів в графі за допомогою алгоритмів Дейкстри та Флойда-Уоршола.

Завдання

Реалізувати програмне застосування (програму), яке виконує наступні функції. *При реалізації вважати, що заданий граф є орієнтованим.*

Частина перша:

1. Зчитування графу з вхідного файлу. На вхід подається текстовий файл наступного вигляду:

   ```
   n m
   v1 u1 w1
   v2 u2 w2
   .......
   vm um wm
   ```

   Тут n – кількість вершин графу (ціле число, більше нуля), m – кількість ребер графу (ціле число, більше нуля), v_i та u_i – початкова та кінцева вершина ребра i ($1 \le v_i \le n$, $1 \le u_i \le n$, цілі числа), w_i – вага ребра (v_i, u_i). Індексація вершин у файлі ведеться з 1.
2. Визначити найкоротший маршрут між двома вершинами та його довжину. За допомогою алгоритму Дейкстри визначити найкоротшу відстань між двома заданими вершинами (які вводяться користувачем), а також вивести сам знайдений найкоротший маршрут. Програма повинна коректно опрацьовувати факт наявності ребер у графі з від'ємною вагою.
3. Визначити найкоротшу відстань від заданої вершини до всіх інших вершин. За допомогою алгоритму Дейкстри визначити найкоротшу відстань від заданої вершини (вводиться користувачем) до всіх інших вершин графу. Програма виводить на екран список вершин із відповідними значеннями найкоротших відстаней.

Частина друга:

1. Зчитування графу з вхідного файлу. На вхід подається текстовий файл наступного вигляду:

$$
\begin{array}{l}
n\ \ m \\
v_1\ \ u_1\ \ w_1 \\
v_2\ \ u_2\ \ w_2 \\
\ldots\ldots\ldots \\
v_m\ \ u_m\ \ w_m
\end{array}
$$

1. Визначити найкоротші відстані між усіма парами вершин в графі за допомогою алгоритму Флойда-Уоршела. За допомогою алгоритму Флойда-Уоршола визначити найкоротшу відстань між усіма парами вершин. Програма повинна виводити на екран або у файл знайдену матрицю відстаней та матрицю історії. За запитом користувача програма повинна виводити знайдений найкоротший маршрут для початкової та кінцевої вершини (вводяться користувачем). Програма повинна коректно опрацьовувати факт наявності в графі циклів з від'ємною вагою.

Код:
```
using System;
using System.Collections.Generic;
using System.IO;

namespace DS_LAB_5
{
    class Program
    {//C:\Users\Богдан\Desktop\dss.txt
        static void Main(string[] args)
        {
            Console.WriteLine("Input path:");
            string path = Console.ReadLine();
            Graph g1 = ReadGraph(path, true);  // Dejksta v1 - v2
            Graph g2= ReadGraph(path, true);   // Dejkstra v1 -all
            Graph g3 = ReadGraph(path, true);  // Floyd
        Console.WriteLine("Input 1st vertex:");
        int v = Convert.ToInt32(Console.ReadLine());
        g1.Dejkstra(v);
        Console.WriteLine("Input 2nd vertex:");
        int l = Convert.ToInt32(Console.ReadLine());
        Console.WriteLine($"path from {v} to {l} costs: {g1.vertices[l - 1].cost}");
        Console.Write(" Verticies of the path (including start point and final point) : ");
        foreach (int item in g1.vertices[l - 1].path)
        {
            Console.Write($" {item} ");
        }
        Console.WriteLine();
        Console.WriteLine("Input a vertex again (We will find all pathes from it) :");
```

```
        int k = Convert.ToInt32(Console.ReadLine());
        g2.Dejkstra(k);
        foreach (Graph.Vertex vert in g2.vertices)
        {
            Console.WriteLine($" From {k} vertex to {vert.name} vertex path costs : {vert.cost}");
        }

        g3.Floyd();


    }
    static Graph ReadGraph(string path, bool or)
    {
        StreamReader reader = new StreamReader(path);
        string s0 = reader.ReadLine();
        int v = Convert.ToInt32(s0.Split(new char[] { ' ' })[0]);
        int e = Convert.ToInt32(s0.Split(new char[] { ' ' })[1]);
        Graph graph = new Graph(true);
        // Заполнение с единицы, не с нуля!!!
        for (int i = 1; i <= v; i++)
        {
            Graph.Vertex temp = new Graph.Vertex(i);
            graph.vertices.Add(temp);
        }

        for (int i = 0; i < e; i++)
        {
            s0 = reader.ReadLine();
            Graph.Edge tempEdge = new Graph.Edge(
                or,
                graph.vertices[Convert.ToInt32(s0.Split(new char[] { ' ' })[0])-1],
                graph.vertices[Convert.ToInt32(s0.Split(new char[] { ' ' })[1])-1],
                Convert.ToInt32(s0.Split(new char[] { ' ' })[2]));
            graph.edges.Add(tempEdge);
        }
        return graph;
    }


}

class Graph
{
    #region properties
    bool oriented;
    public List<Vertex> vertices;
    public List<Edge> edges;
    public List<Vertex> PassedVertices;
    public List<Vertex> ReCountedVertices;
    #endregion

    #region Components
```

```
public class Vertex
{
public int name, cost;
public List<Vertex> neighbours;
public List<Edge> edges;
public List<int> path;

    public void ReCount(List<Vertex> ReCountedVertices)
    {
        foreach (Edge edge in edges)
        {
            edge.pathCost = cost + edge.cost;
            if (!ReCountedVertices.Contains(edge.v2))
            {
                ReCountedVertices.Add(edge.v2);
                edge.v2.ReCount(ReCountedVertices);
            }
        }
    }
public Vertex(int num)
{
    name = num;
    cost = 999999999;
        edges = new List<Edge>();
        neighbours = new List<Vertex>();
        path = new List<int>();
    }
}
public class Edge
{
    public int cost, pathCost;
    public Vertex v1, v2;
    public Edge(bool or,Vertex vertex1, Vertex vertex2, int price)
    {
        v1 = vertex1;
        v2 = vertex2;
        cost = price;
        pathCost = cost;
        if (or)
        {
            v1.neighbours.Add(v2);
            v1.edges.Add(this);
        }
        else
        {
            v1.neighbours.Add(v2);
            v2.neighbours.Add(v1);
            v1.edges.Add(this);
            v2.edges.Add(this);
        }
    }
```

```
        }

        #endregion

        #region methods
        public void Dejkstra(int v)
        {
            List<Edge> checkedEdges = new List<Edge>();
            vertices[v - 1].cost = 0;
            vertices[v - 1].path.Add(vertices[v - 1].name);
            while (edges.Count != checkedEdges.Count)
            {
                PassedVertices.Add(vertices[v - 1]);
                int lowCost = 999999999;
                Edge lowCostEdge = new Edge(true, new Vertex(9998), new Vertex(9999), 0);
                foreach (Vertex vertex in PassedVertices)
                {
                    foreach (Edge edge in vertex.edges)
                    {
                        if (vertex.cost + edge.cost <= lowCost && !checkedEdges.Contains(edge))
                        {
                            lowCost = edge.cost + vertex.cost;
                            lowCostEdge = edge;
                        }
                    }
                }
                if (lowCostEdge.v2.cost <= lowCost)
                {
                   PassedVertices.Add(lowCostEdge.v2);
                    foreach (int item in lowCostEdge.v1.path) ;

                }
                else
                {
                    lowCostEdge.v2.cost = lowCostEdge.pathCost;
                    ReCountedVertices = new List<Vertex>();  //   Recount is recursive function. This
list helps to avoid endless cycle in the graph
                    lowCostEdge.v2.path = new List<int>();
                    foreach (int item in lowCostEdge.v1.path)
                    {
                        lowCostEdge.v2.path.Add(item);
                    }
                    lowCostEdge.v2.path.Add(lowCostEdge.v2.name);
                    lowCostEdge.v2.ReCount(ReCountedVertices);
                    PassedVertices.Add(lowCostEdge.v2);
                }
                checkedEdges.Add(lowCostEdge);
            }
        }

        void Draw(int[,] mat)
        {
```

```
            for (int i = 0; i < mat.GetLength(0); i++)
            {
                Console.WriteLine();
                for (int j = 0; j < mat.GetLength(1); j++)
                {
                    Console.Write(mat[i, j] + "  ");
                }
            }
            Console.WriteLine();
        }

        public void Floyd()
        {
            int[,] mat = new int[vertices.Count, vertices.Count];
            int[,] dist = new int[vertices.Count, vertices.Count];
            foreach (Vertex vertex in vertices)
            {
                List<Edge> inEdges = new List<Edge>();
                foreach (Edge item in edges)
                {
                    if (item.v2.name == vertex.name)
                    {
                        inEdges.Add(item);
                    }
                }
                List<Edge> outEdges = vertex.edges;

                foreach (Edge edgeIn in inEdges)
                {
                    foreach (Edge edgeOut in outEdges)
                    {
                        bool createNewEdge = true;
                        foreach (Edge edge in edgeIn.v1.edges)
                        {
                            if (edge.v2.name == edgeOut.v2.name)
                            {
                                createNewEdge = false;
                                if (edge.cost > edgeIn.cost + edgeOut.cost)
                                {
                                    edge.cost = edgeIn.cost + edgeOut.cost;
                                    mat[edgeIn.v1.name - 1, edgeOut.v2.name - 1] = vertex.name;
                                }

                            }
                        }
                        if (createNewEdge)
                        {
                            Edge newEdge = new Edge(true, edgeIn.v1, edgeOut.v2, edgeIn.cost +
edgeOut.cost);
                            edges.Add(newEdge);
                        if(newEdge.v1.name!=newEdge.v2.name)
                            mat[newEdge.v1.name - 1, newEdge.v2.name - 1] = vertex.name;
```

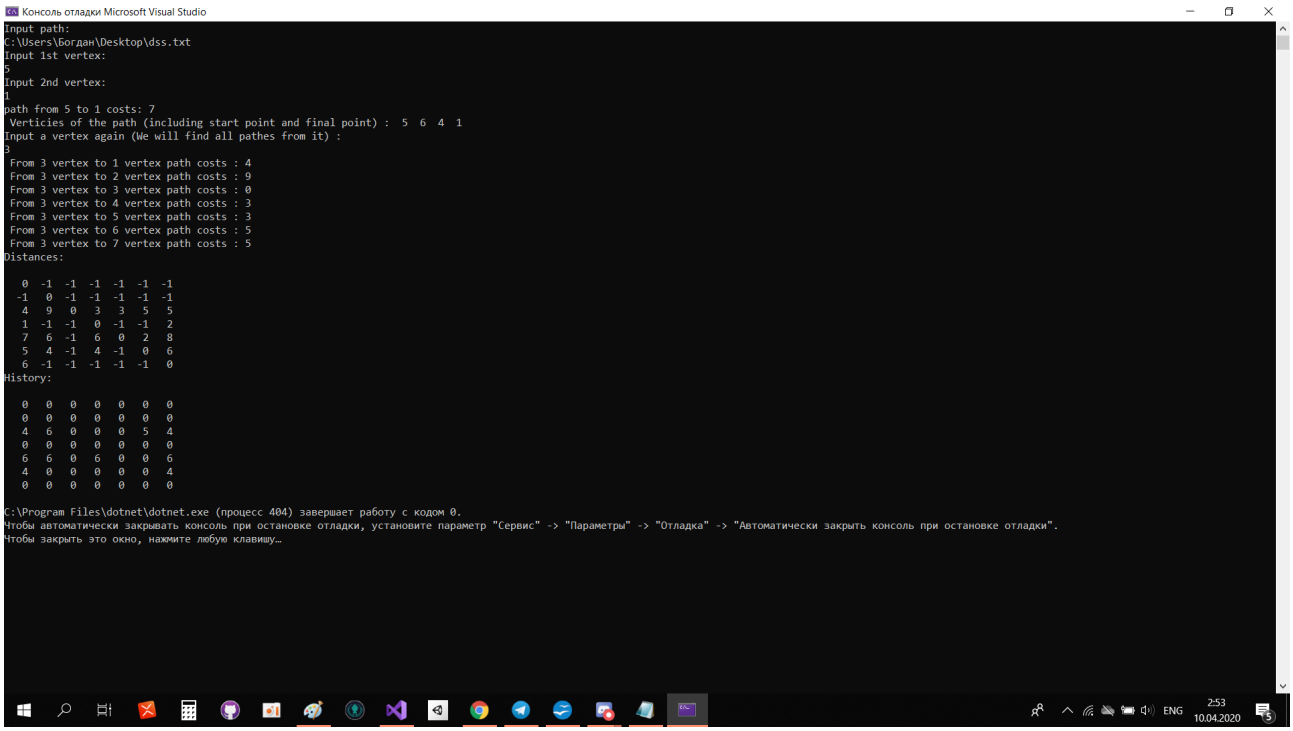
```
                    }
                }
            }

        }
        Console.WriteLine("Distances: ");
        foreach (Vertex v1 in vertices)
        {
            foreach (Vertex v2 in vertices)
            {
                if (v1 == v2)
                {
                    dist[v1.name - 1, v2.name - 1] = 0;
                }
                else
                {
                    foreach (Edge edge in v1.edges)
                    {
                        if (edge.v2 == v2)
                        {
                            dist[v1.name - 1, v2.name - 1] = edge.cost;
                        }
                    }
                }
            }
        }
        Draw(dist);
        Console.WriteLine("History: ");
        Draw(mat);

    }
    #endregion
    public Graph(bool oriented)
    {
        this.oriented = oriented;
        edges = new List<Edge>();
        vertices = new List<Vertex>();
        PassedVertices = new List<Vertex>();
    }
}
}
```

Вивід для такого графу:

7 8
3 4 3
4 1 1
3 5 3
5 6 2
6 2 4
7 1 6
4 7 2
6 4 4


Консоль отладки Microsoft Visual Studio
Input path:
C:\Users\Богдан\Desktop\dss.txt
Input 1st vertex:
5
Input 2nd vertex:
1
path from 5 to 1 costs: 7
Verticies of the path (including start point and final point) : 5 6 4 1
Input a vertex again (We will find all pathes from it) :
3
From 3 vertex to 1 vertex path costs : 4
From 3 vertex to 2 vertex path costs : 9
From 3 vertex to 3 vertex path costs : 0
From 3 vertex to 4 vertex path costs : 3
From 3 vertex to 5 vertex path costs : 3
From 3 vertex to 6 vertex path costs : 5
From 3 vertex to 7 vertex path costs : 5
Distances:
0 -1 -1 -1 -1 -1 -1
-1 0 -1 -1 -1 -1 -1
4 9 0 3 3 5 5
1 -1 -1 0 -1 -1 2
7 6 -1 6 0 2 8
5 4 -1 4 -1 0 6
6 -1 -1 -1 -1 -1 0
History:
0 0 0 0 0 0 0
0 0 0 0 0 0 0
4 6 0 0 0 5 4
0 0 0 0 0 0 0
6 6 0 6 0 0 6
4 0 0 0 0 0 4
0 0 0 0 0 0 0
C:\Program Files\dotnet\dotnet.exe (процесс 404) завершает работу с кодом 0.
Чтобы автоматически закрывать консоль при остановке отладки, установите параметр "Сервис" -> "Параметры" -> "Отладка" -> "Автоматически закрыть консоль при остановке отладки".
Чтобы закрыть это окно, нажмите любую клавишу…
ENG
2:53
10.04.2020