# КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ імені ТАРАСА ШЕВЧЕНКА



## ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

**Кафедра прикладних інформаційних систем**

**Звіт до лабораторної роботи №10**

# з курсу

**«Алгоритми і структури даних»**

*Студента 1 курсу*

*групи ПП-11 спеціальності 122 «Комп'ютерні науки» ОП «Прикладне програмування»*

Селецького В. Р.

*Викладач:*

д.е.н., к.т.н., проф. Плескач В.Л.

к.ф.-м.н., доц. Шолохов О.В.

## Київ – 2021

**1.Назва роботи**

Пошук найкоротших шляхів на орієнтованих графах

1. **Тема роботи**

Пошук найкоротших шляхів на орієнтованих графах

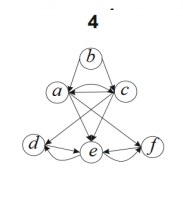
1. **Мета роботи**

Набуття практичних вмінь і навичок з використання алгоритмів

Дейкстри та Флойда

1. **Умова завдання**

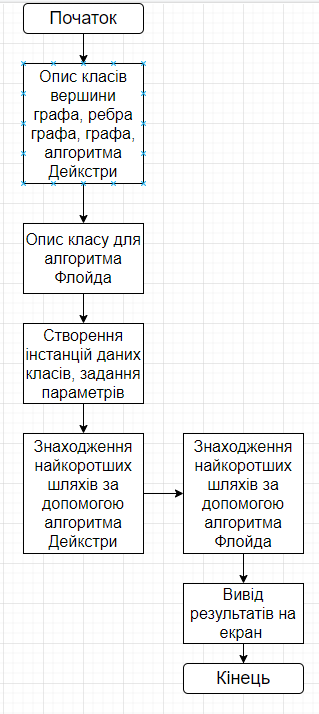
(4 варіант)



1. **Рішення**

Описуємо клас ребра графа, клас вершини графа, власне графа, клас алгоритму Дейкстри та функцію для алгоритму Флойду. У головній функції створюємо інстанцію класу графа, задаємо потрібні вершини та ребра, знаходимо найкоротший шлях серед кожної пари вершин. Викликаємо функцію для виконання задачі алгоритмом Флойда.

Блок-схема:



Код:

using System;

using System.Collections.Generic;

namespace Лаба9 {

class Program {

public class GraphEdge {

public GraphVertex ConnectedVertex { get; }

public int EdgeWeight { get; }

public GraphEdge(GraphVertex connectedVertex, int weight) {

ConnectedVertex = connectedVertex;

EdgeWeight = weight;

}

}

public class GraphVertex {

public string Name { get; }

public List<GraphEdge> Edges { get; }

public GraphVertex(string vertexName) {

Name = vertexName;

Edges = new List<GraphEdge>();

}

public void AddEdge(GraphEdge newEdge) {

Edges.Add(newEdge);

}

public void AddEdge(GraphVertex vertex, int edgeWeight) {

AddEdge(new GraphEdge(vertex, edgeWeight));

}

public override string ToString() => Name;

}

public class Graph {

public List<GraphVertex> Vertices { get; }

public Graph() {

Vertices = new List<GraphVertex>();

}

public void AddVertex(string vertexName) {

Vertices.Add(new GraphVertex(vertexName));

}

public GraphVertex FindVertex(string vertexName) {

foreach (var v in Vertices) {

if (v.Name.Equals(vertexName))

{

return v;

}

}

return null;

}

public void AddEdge(string firstName, string secondName, int weight) {

var v1 = FindVertex(firstName);

var v2 = FindVertex(secondName);

if (v2 != null && v1 != null) {

v1.AddEdge(v2, weight);

v2.AddEdge(v1, weight);

}

}

}

public class GraphVertexInfo {

public GraphVertex Vertex { get; set; }

public bool IsUnvisited { get; set; }

public int EdgesWeightSum { get; set; }

public GraphVertex PreviousVertex { get; set; }

public GraphVertexInfo(GraphVertex vertex) {

Vertex = vertex;

IsUnvisited = true;

EdgesWeightSum = int.MaxValue;

PreviousVertex = null;

}

}

public class Dijkstra

{

Graph graph;

List<GraphVertexInfo> infos;

public Dijkstra(Graph graph) {

this.graph = graph;

}

void InitInfo() {

infos = new List<GraphVertexInfo>();

foreach (var v in graph.Vertices) {

infos.Add(new GraphVertexInfo(v));

}

}

GraphVertexInfo GetVertexInfo(GraphVertex v) {

foreach (var i in infos) {

if (i.Vertex.Equals(v)) {

return i;

}

}

return null;

}

public GraphVertexInfo FindUnvisitedVertexWithMinSum() {

var minValue = int.MaxValue;

GraphVertexInfo minVertexInfo = null;

foreach (var i in infos) {

if (i.IsUnvisited && i.EdgesWeightSum < minValue) {

minVertexInfo = i;

minValue = i.EdgesWeightSum;

}

}

return minVertexInfo;

}

public string FindShortestPath(string startName, string finishName) {

return FindShortestPath(graph.FindVertex(startName), graph.FindVertex(finishName));

}

public string FindShortestPath(GraphVertex startVertex, GraphVertex finishVertex) {

InitInfo();

var first = GetVertexInfo(startVertex);

first.EdgesWeightSum = 0;

while (true)

{

var current = FindUnvisitedVertexWithMinSum();

if (current == null)

{

break;

}

SetSumToNextVertex(current);

}

return GetPath(startVertex, finishVertex);

}

void SetSumToNextVertex(GraphVertexInfo info) {

info.IsUnvisited = false;

foreach (var e in info.Vertex.Edges) {

var nextInfo = GetVertexInfo(e.ConnectedVertex);

var sum = info.EdgesWeightSum + e.EdgeWeight;

if (sum < nextInfo.EdgesWeightSum) {

nextInfo.EdgesWeightSum = sum;

nextInfo.PreviousVertex = info.Vertex;

}

}

}

string GetPath(GraphVertex startVertex, GraphVertex endVertex) {

var path = endVertex.ToString();

while (startVertex != endVertex) {

endVertex = GetVertexInfo(endVertex).PreviousVertex;

path = endVertex.ToString() + path;

}

return path;

}

}

public static void floyd() {

int[,] matrix = new int[,] {

{0, 1, 0, 1, 1, 0 },

{1, 0, 1, 0, 1, 0 },

{0, 1, 0, 0, 0, 1 },

{1, 0, 0, 0, 0, 1 },

{1, 1, 0, 0, 1, 0 },

{0, 0, 1, 1, 0, 0 }

};

int size = 6;

for (var k = 0; k < size; ++k)

{

for (var i = 0; i < size; ++i)

{

for (var j = 0; j < size; ++j)

{

if (matrix[i, j] > matrix[i, k] + matrix[k, j])

matrix[i, j] = matrix[i, k] + matrix[k, j];

}

}

}

Console.WriteLine("\nChanged matrix:");

for (int q = 0; q < 6; q++) {

for (int w = 0; w < 6; w++) {

Console.Write(matrix[q, w]);

}

Console.WriteLine();

}

}

static void Main(string[] args) {

var g = new Graph();

g.AddVertex("A");

g.AddVertex("B");

g.AddVertex("C");

g.AddVertex("D");

g.AddVertex("E");

g.AddVertex("F");

g.AddEdge("A", "B", 0);

g.AddEdge("A", "C", 0);

g.AddEdge("A", "D", 0);

g.AddEdge("B", "C", 0);

g.AddEdge("B", "E", 1);

g.AddEdge("C", "D", 0);

g.AddEdge("C", "E", 1);

g.AddEdge("C", "F", 0);

g.AddEdge("D", "F", 0);

g.AddEdge("E", "F", 1);

g.AddEdge("E", "F", 1);

var dijkstra = new Dijkstra(g);

var path = dijkstra.FindShortestPath("A", "F");

var path2 = dijkstra.FindShortestPath("A", "C");

var path3 = dijkstra.FindShortestPath("A", "D");

var path4 = dijkstra.FindShortestPath("A", "E");

var path5 = dijkstra.FindShortestPath("A", "F");

var path6 = dijkstra.FindShortestPath("B", "C");

var path7 = dijkstra.FindShortestPath("B", "D");

var path8 = dijkstra.FindShortestPath("B", "E");

var path9 = dijkstra.FindShortestPath("B", "F");

var path10 = dijkstra.FindShortestPath("C", "D");

var path11 = dijkstra.FindShortestPath("C", "E");

var path12 = dijkstra.FindShortestPath("C", "F");

var path13 = dijkstra.FindShortestPath("D", "E");

var path14 = dijkstra.FindShortestPath("D", "F");

Console.WriteLine(path);

Console.WriteLine(path2);

Console.WriteLine(path3);

Console.WriteLine(path4);

Console.WriteLine(path5);

Console.WriteLine(path6);

Console.WriteLine(path7);

Console.WriteLine(path8);

Console.WriteLine(path9);

Console.WriteLine(path10);

Console.WriteLine(path11);

Console.WriteLine(path12);

Console.WriteLine(path13);

Console.WriteLine(path14);

floyd();

}

}

}

Результат роботи програми:

[screenshot]

**Контрольні питання**

1. Для чого використовують алгоритми пошуку шляхів на графах?

Задачі, що потребують зазначених алгоритмів, можуть бути дуже різноманітними - від GPS-навігаторів до організації робочого місця (в першому випадку вершинами графа є перехрещення, в другому - потрібні предмети)

1. Особливість алгоритму Дейкстри

Полягає в тому, що алгоритм працює лише для графів без від’ємних ребер

1. і-та ітерація алгоритму Дейкстри

Якщо всі вершини відвідані, алгоритм завершено;

Знаходимо таку вершину (із ще невідвіданих), поточна найкоротша відстань до якої мінімальна. Обходимо всіх її сусідів і, якщо шлях в сусідню вершину через 1 менший за поточний мінімальний шлях в цю сусідню вершину, то запам'ятовуємо цей новий, коротший шлях як поточний найкоротший шлях до сусіда.

1. В чому особливість алгоритму Флойда

Полягає в тому, що алгоритм використовується у графах з додатними та від’ємними ребрами

1. і-та ітерація алгоритму Флойда

Необхідно пронумерувати вершини вихідного графа цілими числами від 1 до N. Після цього визначимо D0 , Задавши величину кожного її елемента (i, j) дорівнює довжині найкоротшою дуги, що з'єднує вершину i з вершиною j. Якщо у вихідному графі зазначені вершини не з'єднуються дугами, покласти d 0 ij = ∞. Крім того, для всіх i покласти d 0 ij = 0.

Для цілого m, послідовно приймає значення 1, 2, ..., N, визначити за величинами елементів матриці D m-1 величини елементів матриці D m , Використовуючи співвідношення d m ij = min {d m-1 im + d m-1 mj, d m-1 ij}.

При визначенні величини кожного елемента матриці D m фіксувати відповідний найкоротший шлях.

Після закінчення даної процедури величина елементів (i, j) матриці визначає довжину найкоротшого шляху, що веде з вершини i в вершину j.

1. Наведіть порівняльну характеристику алгоритмів Дейкстри та Флойда.

Алгоритм Дейкстри не може працювати у графах з від’ємними ребрами, в той час як для алгоритму Флойда це не стане на заваді. Алгоритм Дейкстри обраховує найкоротший шлях з вершини до інших вершин, алгоритм Флойда - найкоротший шлях для всіх пар вершин

1. Час для виконання алгоритму Дейкстри

O (E \* log(V))

1. Час для виконання алгоритму Флойда

O (V3)

1. Наведіть приклади доцільності використання алгоритму Дейкстри.

Алгоритм Дейкстри доцільніше використовувати у тих випадках, коли нам потрибно ставити пріоритети дослідження шляхів; у випадках, коли ми шукаємо найкоротший шлях від конкретно заданої вершини до іншої заданої вершини

1. Наведіть приклади доцільності використання алгоритму Флойда.

Доцільніше використовувати алгоритм Флойда у тих випадках, коли граф має від’ємні ваги ребер; для пошуку найкоротших шляхів в орієнтованих графах; для пошуку шляхів від кожної вершини, бо вони знаходяться в одну ітерацію алгоритму Флойда

1. **Висновки**

В результаті виконання даної лабораторної роботи я оволодів практичними навичками використання алгоритмів Дейкстри та Флойда, навчився складати програми для виконання операцій з використанням даних алгоритмів на графах. Вважаю дану лабораторну роботу виконаною в повному обсязі.