Ministerul Educaţiei și Cercetării al Republicii Moldova

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică şi Microelectronică

**RAPORT**

Lucrare de laborator Nr. 4

*la Matematica Discretă*

Tema: ALGORITMI DE DETERMINARE A DRUMULUI MINIM

A efectuat: st. gr. SI-212 Șeremet Alexandru

A verificat: lect. asist. Popovici Nadejda

Chişinău 2022

1. **SCOPUL LUCRĂRII:**

* Studierea algoritmilor de determinare a drumurilor minime și maxime într-un graf.
* Elaborarea programelor de determinare a drumului minim și maxim într-un graf ponderat.

**2.NOTE DE CURS**

**Noţiune de drum minim**

Pentru un graf orientat *G = (X,U)* se va numi *drum* un şir de vârfuri *D* = *(x0, x1,..., xr)* cu proprietatea că *(x0, x1)*, *(x1, x2)*,..., *(xr-1, xr)* aparţin lui *U*, deci sunt arce ale grafului şi extremitatea finală a arcului precedent coincide cu extremitatea iniţială a arcului următor.

Vârfurile *x0* şi *xr* se numesc extremităţile drumului *D*. Lungimea unui drum este dată de numărul de arce pe care le conţine. Dacă vârfurile *x0, x1,..., xr* sunt distincte două câte două drumul *D* este elementar.

Adeseori, fiecărui arc (muchii) i se pune în corespondenţă un număr real care se numeşte *ponderea* (lungimea) arcului. Lungimea arcului *(xi, xj)* se va nota *w(i,j)*, iar în cazul în care un arc este lipsă ponderea lui va fi considerată foarte mare (pentru calculator cel mai mare număr pozitiv posibil). În cazul grafurilor cu arce ponderate (grafuri ponderate) se va considera lungime a unui drum suma ponderilor arcelor care formează acest drum. Drumul care uneşte două vârfuri concrete şi are lungimea cea mai mică se va numi *drum minim* iar lungimea drumului minim vom numi *distanţă*. Vom nota distanţa dintre *x* şi *t* prin *d(x, t)*, evident, *d(x,x)=0*.

**Algoritmul lui Ford pentru detrminarea drumului minim**

Permite determinarea drumului minim care începe cu un vârf iniţial *xi* până la oricare vârf al grafului *G*. Dacă prin *Lij* se va nota ponderea arcului *(xi, xj)* atunci algoritmul conţine următorii paşi:

1. Fiecărui vârf *xj* al grafului *G* se va ataşa un număr foarte mare *Hj(∞)*. Vârfului iniţial i se va ataşa *Ho = 0*;
2. Se vor calcula diferenţele *Hj - Hi* pentru fiecare arc *(xi, xj)*. Sunt posibile trei cazuri:
3. *Hj - Hi < Lij,*
4. *Hj - Hi = Lij,*
5. *Hj - Hi > Lij.*

Cazul "*c*" permite micşorarea distanţei dintre vârful iniţial şi *xj* din care cauză se va realiza *Hj = Hi + Lij*.

Pasul 2 se va repeta atâta timp cât vor mai exista arce pentru care are loc inegalitatea “c”. La terminare, etichetele *Hi* vor defini distanţa de la vârful iniţial până la vârful dat *xi*.

3. Acest pas presupune stabilirea secvenţei de vârfuri care va forma drumul minim. Se va pleca de la vârful final *xj* spre cel iniţial. Predecesorul lui *xj* va fi considerat vârful *xi* pentru care va avea loc *Hj - Hi = Lij*. Dacă vor exista câteva arce pentru care are loc această relaţie se va alege la opţiune.

**Algoritmul Bellman - Kalaba**

Permite determinarea drumului minim dintre oricare vârf al grafului până la un vârf, numit vârf final.

Etapa iniţială presupune ataşarea grafului dat *G* a unei matrice ponderate de adiacenţă, care se va forma în conformitate cu următoarele:

1. *M(i,j)* = *Lij*, dacă există arcul *(xi, xj)* de pondere *Lij*;
2. *M(i,j)* = ∞, unde ∞ este un număr foarte mare (de tip întreg maximal pentru calculatorul dat), dacă arcul *(xi, xj)* este lipsă;
3. *M(i,j)* = *0*, dacă *i = j*.

La etapa a doua se va elabora un vector *V0* în felul următor:

1. *V0(i) = Lin*, dacă există arcul *(xi, xn)*, unde *xn* este vârful final pentru care se caută drumul minim, *Lin* este ponderea acestui arc;
2. *V0(i) =* ∞, dacă arcul *(xi, xn)* este lipsă;
3. *V0(i) = 0*, dacă *i = j*.

Algoritmul constă în calcularea iterativă a vectorului *V* în conformitate cu următorul procedeu:

1. *Vk(i) = min{Vk-1; Lij+Vk-1(j)}*, unde *i = 1, 2,…, n - 1, j = 1, 2,..., n*; *i<>j;*
2. *Vk(n) = 0*.

Când se va ajunge la *Vk = Vk-1* - STOP.

Componenta cu numărul *i* a vectorului *Vk* cu valoarea diferită de zero ne va da valoarea minimă a drumului care leagă vârful *i* cu vârful *n*.

**3. SARCINA DE BAZĂ**

1. Elaboraţi procedura introducerii unui graf ponderat;

2. Elaboraţi procedurile determinării drumului minim;

3. Realizaţi un program cu următoarele funcţii:

* introducerea grafului ponderat cu posibilităţi de analiză sintactică şi semantică şi de corectare a informaţiei;
* determinarea drumului minim;
* extragerea informaţiei la display şi printer (valoarea drumului minim şi succesiunea vârfurilor care formează acest drum).

**4. CODUL PROGRAMULUI**

#include <iostream>

#include <vector>

#include <algorithm>

#include <iomanip>

#include <climits>

using namespace std;

// Data structure to store graph edges

struct Edge

{

int source, dest, weight;

};

bool isLong = false;

// Recursive function to print the path of (just) the first given vertex from source vertex.

void getPath(vector<vector<int>> const parent, vector<int> const vlist, int pathNum, vector<int> &resPath)

{

if (vlist.empty())

return;

int v = vlist[pathNum];

getPath(parent, parent[v], pathNum, resPath);

resPath.push\_back(v);

}

vector<vector<int>> parse(vector<vector<int>> const parent, int dest, int N)

{

vector<vector<int>> res(N);

for (int i = 0; i < parent.size(); i++)

{

getPath(parent, parent[dest], i, res[i]);

}

return res;

}

bool isValid(int from, int to, vector<Edge> const &edges)

{

for (int i = 0; i < edges.size(); i++)

{

if (from == edges[i].source && to == edges[i].dest)

{

return true;

}

}

return false;

}

void removeDuplicates(std::vector<vector<int>> &v)

{

auto end = v.end();

for (auto it = v.begin(); it != end; ++it)

{

end = std::remove(it + 1, end, \*it);

}

v.erase(end, v.end());

}

void cleanPaths(vector<vector<int>> &resPath, vector<Edge> const &edges)

{

removeDuplicates(resPath);

for (int k = 0; k < resPath.size(); k++)

{

for (int i = 0, j = 1; j < resPath[k].size(); i++, j++)

{

if (!isValid(resPath[k][i], resPath[k][j], edges))

resPath.erase(resPath.begin() + k);

}

}

}

void printPaths(vector<vector<int>> parent, int dest, vector<int> distance, vector<Edge> const &edges)

{

if (!isLong)

{

cout << "\nShortest distance of vertex " << dest << " from the source is "

<< distance[dest] << ". Its paths are:\n";

}

else

{

cout << "\nLongest distance of vertex " << dest << " from the source is "

<< -distance[dest] << ". Its paths are:\n";

}

vector<vector<int>> res = parse(parent, dest, parent.size());

cleanPaths(res, edges);

for (int i = 0; i < res.size(); i++)

{

cout << "[ ";

for (int j = 0; j < res[i].size(); j++)

cout << res[i][j] << " ";

cout << "]" << endl;

}

}

// Function to run Bellman Ford Algorithm from given source

void BellmanFord(vector<Edge> const &edges, int source, int dest, int N)

{

// count number of edges present in the graph

int E = edges.size();

// distance[] and parent[] stores shortest-path (least cost/path)

// information. Initially all vertices except source vertex have

// a weight of infinity and no parent

vector<int> distance(N, INT\_MAX);

distance[source] = 0;

vector<vector<int>> parent(N);

int u, v, w, k = N;

// Relaxation step (run V-1 times)

while (--k)

{

for (int j = 0; j < E; j++)

{

// edge from u to v having weight w

u = edges[j].source, v = edges[j].dest;

w = edges[j].weight;

// if the distance to the dest v can be

// shortened by taking the edge u-> v

if (distance[u] != INT\_MAX)

{

// if the distance to the dest v can be

// shortened by taking the edge u-> v

if (distance[u] + w < distance[v])

{

// update distance to the new lower value

distance[v] = distance[u] + w;

// forget the previous parent list.

parent[v].clear();

}

// if u-> v is a way to get the shortest

// distance to the dest v.

if (distance[u] + w == distance[v])

{

// add u as a possible parent for v

parent[v].push\_back(u);

}

}

}

}

// Run Relaxation step once more for Nth time to

// check for negative-weight cycles

for (int i = 0; i < E; i++)

{

// edge from u to v having weight w

u = edges[i].source, v = edges[i].dest;

w = edges[i].weight;

// if the distance to the dest u can be

// shortened by taking the edge u-> v

if (distance[u] != INT\_MAX && distance[u] + w < distance[v])

{

cout << "Negative Weight Cycle Found!!";

return;

}

}

printPaths(parent, dest, distance, edges);

}

void togglePathType(vector<Edge> &edges)

{

for (int i = 0; i < edges.size(); i++)

{

edges[i].weight \*= -1;

}

isLong = !isLong;

}

int introduceEdges(vector<Edge> &edges)

{

int E, V = 0;

cout << "Cate muchii va avea graful? ";

cin >> E;

for (int e = 0; e < E; e++)

{

int from = 0;

int to = 0;

int cost;

while (true)

{

std::cout << "Ce varfuri uneste muchia " << e << "?\n";

cin >> from;

cin >> to;

if (from < 0 || to < 0)

{

std::cout << "Valoare invalida, introduceti din nou" << endl;

}

else

{

if (from > V)

V = from;

if (to > V)

V = to;

std::cout << "Ponderea: ";

cin >> cost;

break;

}

}

Edge newEdge;

newEdge.source = from;

newEdge.dest = to;

newEdge.weight = cost;

edges.push\_back(newEdge);

}

V++;

return V;

}

// main function

int main()

{

/\*vector<Edge> edges =

{

// (x, y, w) -> edge from x to y having weight w

{1, 2, 5},

{1, 3, 3},

{1, 4, 5},

{1, 5, 6},

{1, 6, 8},

{2, 5, 4},

{2, 4, 1},

{3, 5, 2},

{4, 5, 3},

{4, 6, 5},

{5, 6, 4},

{5, 7, 6},

{6, 7, 5},

};\*/

vector<Edge> edges;

int N = introduceEdges(edges);

cout << "Care varf este sursa? ";

int source = 1;

cin >> source;

cout << "Care varf este destinatia? ";

int dest = 7;

cin >> dest;

BellmanFord(edges, source, dest, N);

togglePathType(edges);

BellmanFord(edges, source, dest, N);

return 0;

}

**5. EXECUTIA CODULUI**





**6. CONCLUZII:**

* Există mai mulți algoritmi pentru găsirea celui mai scurt⁠ și a celui mai lung⁠ drum în graf, cu distincția importantă că prima problemă este mult mai ușoară din punct de vedere computațional decât cea din urmă.
* Algoritmul lui Dijkstra produce o listă cu cele mai scurte drumuri de la un nod sursă la orice alt nod într-un graf orientat sau neorientat cu ponderi nenegative (sau fără ponderi), în timp ce algoritmul Bellman-Ford⁠ poate fi aplicat grafurilor orientate cu ponderi negative. Algoritmul Floyd-Warshall⁠ poate fi utilizat pentru a găsi cele mai scurte drumuri între toate perechile de noduri din grafurile orientate ponderate.