Ministerul Educaţiei și Cercetării al Republicii Moldova

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică şi Microelectronică

**RAPORT**

Lucrare de laborator Nr. 6

*la Matematica Discretă*

Tema: DETERMINĂRII GRAFULUI DE ACOPERIRE

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

A verificat: lect. asist. Popovici Nadejda

Chişinău 2022

1. **SCOPUL LUCRĂRII:**

* Studierea algoritmului de determinare a grafului de acoperire şi elaborarea programelor care vor realiza acest algoritm.

**2.NOTE DE CURS**

**Noţiune de graf de acoperire**

Fie *H* un subgraf care conţine toate vârfurile unui graf arbitrar *G*. Dacă pentru fiecare componentă de conexitate a lui *G* subgraful *H* va defini un arbore atunci *H* se va numi graf de acoperire (scheletul sau carcasă) grafului G. Este evident că graful de acoperire există pentru oricare graf: eliminând ciclurile din fiecare componentă de conexitate, adică eliminând muchiile care sunt în plus, vom ajunge la graful de acoperire.

Se numeşte graf aciclic orice graf care nu conţine cicluri. Pentru un graf arbitrar *G* cu *n* vârfuri şi *m* muchii sunt echivalente următoarele afirmaţii:

1. *G* este arbore;
2. *G* este un graf conex şi *m* = *n - 1*;
3. *G* este un graf aciclic şi *m = n - 1*;
4. oricare două vârfuri distincte (diferite) ale lui *G* sunt unite printr-un lanţ simplu care este unic;
5. *G* este un graf aciclic cu proprietatea că, dacă o pereche oarecare de vârfuri neadiacente vor fi unite cu o muchie, atunci graful obţinut va conţine exact un ciclu.

**Consecinţă**: numărul de muchii pentru un graf arbitrar G, care va fi necesar a fi eliminate spre a obţine un graf de acoperire nu depinde de ordinea eliminării lor şi este egal cu

*m(G)-n(G)+k(G)*,

unde *m(G), n(G)* şi *k(G)* sunt numărul de muchii, vârfuri şi componente conexe, respectiv.

Numărul *s(G) = m(G)-n(G)+ k(G)* se numeşte *rang* *ciclic* sau număr *ciclomatic* al grafului *G*. Numărul *r(G) = n(G)-k(G)* – *rang cociclomatic* sau *număr cociclomatic.*

Deci, *s(G)+r(G)=m(G)*.

Este adevărată următoarea afirmaţie: orice subgraf a unui graf arbitrar *G* se conţine într-un graf de acoperire a grafului *G*.

**Algoritmul de determinare a grafului de acoperire**

Există mai mulţi algoritmi de determinare a grafului de acoperire. Algoritmul de mai jos nu este un algoritm-standard, ci este unul elaborat în bază algoritmului de căutare în lărgime. Esenţa algoritmului constă în aceea că folosind două fire de aşteptare în unul din care sunt înscrise (pe rând) numerele vârfurilor adiacente cu vârfurile din celălalt FA (ca şi în cazul căutării în lărgime), vor fi eliminate muchiile dintre vârfurile unui FA şi toate muchiile în afară de una dintre fiecare vârf al FA curent şi vârfurile din FA precedent. în cazul În care ambele FA vor deveni vide procedura se va termina.

Pentru a nu admite ciclarea şi ca să fim siguri că au fost prelucrate toate componentele conexe se va utiliza marcarea vârfurilor. Dacă după terminarea unui ciclu ordinar nu au mai rămas vârfuri nemarcate procedura ia sfârşit, în caz contrar în calitate de vârf iniţial se va lua oricare din vârfurile nemarcate.

**Descrierea algoritmului:**

1. Se vor declara două FA (FA1 şi FA2) vide.
2. Se va lua în calitate de vârf iniţial un vârf arbitrar al grafului.
3. Se va introduce vârful iniţial în firul de aşteptare vid FA1 şi se va marca acest vârf.
4. Se vor introduce în FA2 toate vârfurile adiacente cu vârfurile din FA1 şi se vor marca. Dacă FA2 este vid se va trece la p.7, în caz contrar - la p. 4.
5. Se vor elimina toate muchiile care leagă vârfurile din FA2.
6. Pentru toate vârfurile din FA2 vor fi eliminate toate muchiile în afară de una care leagă vârful dat cu vârfurile din FA1.
7. Se vor schimba cu numele FA1 şi FA2 (FA1 va deveni FA2 şi invers).
8. Dacă există cel puţin un vârf nemarcat se va lua în calitate de vârf iniţial oricare din acestea şi se va trece la p.1, altfel
9. STOP.

Graful obţinut este graful de acoperire.

**3. SARCINA DE BAZĂ**

2. Elaboraţi organigrama algoritmului şi programul procedurii de determinare a grafului de acoperire cu posibilităţi de pornire a procedurii din oricare vârf al grafului.

3. Utilizând procedurile de introducere a grafului în memoria CE din lucrarea Nr. 1, elaboraţi un program cu următoarele facilităţi:

* introducerea grafului care este dat sub formă de matrice de incidenţă, adiacenţă sau listă de adiacenţă;
* determinarea grafului de acoperire, pornind de la un vârf arbitrar;
* extragerea informaţiei la display sau imprimantă în oricare din formele numite.

**4. CODUL PROGRAMULUI**

// C++ program for Kruskal's algorithm to find Minimum

// Spanning Tree of a given connected, undirected and

// weighted graph

#include <bits/stdc++.h>

using namespace std;

// Creating shortcut for an integer pair

typedef pair<int, int> iPair;

// Structure to represent a graph

struct Graph

{

int V, E;

vector<pair<int, iPair>> edges;

// Constructor

Graph(int V, int E)

{

this->V = V;

this->E = E;

}

// Utility function to add an edge

void addEdge(int u, int v, int w)

{

edges.push\_back({w, {u, v}});

}

// Function to find MST using Kruskal's

// MST algorithm

int kruskalMST();

};

// To represent Disjoint Sets

struct DisjointSets

{

int \*parent, \*rnk;

int n;

// Constructor.

DisjointSets(int n)

{

// Allocate memory

this->n = n;

parent = new int[n + 1];

rnk = new int[n + 1];

// Initially, all vertices are in

// different sets and have rank 0.

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

{

rnk[i] = 0;

// every element is parent of itself

parent[i] = i;

}

}

// Find the parent of a node 'u'

// Path Compression

int find(int u)

{

/\* Make the parent of the nodes in the path

from u--> parent[u] point to parent[u] \*/

if (u != parent[u])

parent[u] = find(parent[u]);

return parent[u];

}

// Union by rank

void merge(int x, int y)

{

x = find(x), y = find(y);

/\* Make tree with smaller height

a subtree of the other tree \*/

if (rnk[x] > rnk[y])

parent[y] = x;

else // If rnk[x] <= rnk[y]

parent[x] = y;

if (rnk[x] == rnk[y])

rnk[y]++;

}

};

/\* Functions returns weight of the MST\*/

int Graph::kruskalMST()

{

int mst\_wt = 0; // Initialize result

// Sort edges in increasing order on basis of cost

sort(edges.begin(), edges.end());

// Create disjoint sets

DisjointSets ds(V);

// Iterate through all sorted edges

vector<pair<int, iPair>>::iterator it;

for (it = edges.begin(); it != edges.end(); it++)

{

int u = it->second.first;

int v = it->second.second;

int set\_u = ds.find(u);

int set\_v = ds.find(v);

// Check if the selected edge is creating

// a cycle or not (Cycle is created if u

// and v belong to same set)

if (set\_u != set\_v)

{

// Current edge will be in the MST

// so print it

cout << u << " - " << v << endl;

// Update MST weight

mst\_wt += it->first;

// Merge two sets

ds.merge(set\_u, set\_v);

}

}

return mst\_wt;

}

void printAdjMatrix(Graph g)

{

cout << endl

<< endl

<< "Reprezentarea grafului ca matrice de adiacenta:";

vector<vector<int>> m(

g.V,

vector<int>(g.V));

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

{

std::pair<int, iPair> coords;

coords = g.edges[i];

int x = coords.second.first;

int y = coords.second.second;

int w = coords.first;

m[x - 1][y - 1] = w;

m[y - 1][x - 1] = w;

}

for (int i = 0; i < g.V; i++)

{

cout << endl;

for (int j = 0; j < g.V; j++)

cout << m[i][j] << ' ';

}

}

void printAdjList(Graph g)

{

cout << endl

<< endl

<< "Reprezentarea grafului ca lista de adiacenta:";

vector<vector<int>> m;

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

{

vector<int> tmp;

tmp.push\_back(i + 1);

m.push\_back(tmp);

}

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

{

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

{

std::pair<int, iPair> coords;

coords = g.edges[i];

int x = coords.second.first;

int y = coords.second.second;

int w = coords.first;

if (!std::count(m[x - 1].begin(), m[x - 1].end(), y))

m[x - 1].push\_back(y);

if (!std::count(m[y - 1].begin(), m[y - 1].end(), x))

m[y - 1].push\_back(x);

}

}

for (int i = 0; i < g.V; i++)

{

cout << endl;

cout << m[i][0] << " -> ";

for (int j = 1; j < m[i].size(); j++)

cout << m[i][j] << ' ';

}

}

void printIncMatrix(Graph g)

{

cout << endl

<< endl

<< "Reprezentarea grafului ca matrice de incidenta:";

vector<vector<int>> m(

g.E,

vector<int>(g.V));

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

{

std::pair<int, iPair> coords;

coords = g.edges[i];

int x = coords.second.first;

int y = coords.second.second;

int w = coords.first;

m[i][x - 1] = 1;

m[i][y - 1] = 1;

}

for (int i = 0; i < g.V; i++)

{

cout << endl;

for (int j = 0; j < g.V; j++)

cout << m[i][j] << ' ';

}

}

// Driver program to test above functions

int main()

{

/\* Let us create above shown weighted

and undirected graph \*/

cout << "Cate varfuri si muchii are graful?" << endl;

int V, E;

cin >> V >> E;

Graph g(V, E);

int u, v, w;

cout << "introduceti muchiile" << endl;

int i = 0;

while (i < g.E)

{

cout << "varful " << i + 1 << endl;

cin >> u >> v >> w;

g.addEdge(u, v, w);

i++;

}

/\*g.addEdge(1, 2, 4);

g.addEdge(1, 7, 8);

g.addEdge(2, 3, 8);

g.addEdge(2, 7, 11);

g.addEdge(3, 4, 7);

g.addEdge(3, 9, 2);

g.addEdge(3, 6, 4);

g.addEdge(9, 7, 6);

g.addEdge(7, 6, 2);

g.addEdge(4, 5, 9);

g.addEdge(6, 4, 10);\*/

cout << "Edges of MST are \n";

int mst\_wt = g.kruskalMST();

cout << "\nWeight of MST is " << mst\_wt;

printAdjMatrix(g);

printAdjList(g);

printIncMatrix(g);

return 0;

}

**5. EXECUTIA CODULUI**





**6. CONCLUZII:**

* Un arbore minim de acoperire sau un arbore de acoperire de pondere minimă este o submulțime a muchiilor unui graf neorientat conex cu muchii ponderate, care toate nodurile între ele, fără cicluri și cu ponderea totală a muchiilor minimă. Adică este un arbore de acoperire⁠ a cărui sumă a ponderilor muchiilor este cât mai mică posibil. Mai general, orice graf neorientat cu muchii ponderate (nu neapărat conex) are o pădure minimă de acoperire, care este o reuniune a arborilor minimi de acoperire ai componentelor sale conexe.
* Arborii minimi de acoperire au aplicații directe în proiectarea rețelelor, inclusiv a rețelelor de calculatoare, de telecomunicații⁠, de transport, de alimentare cu apă⁠ și rețele electrice (pentru care au fost inventate pentru prima dată, așa cum s-a menționat mai sus). Ele sunt invocate ca subrutine în algoritmi pentru alte probleme, inclusiv în algoritmul Christofides pentru aproximarea problemei comis-voiajorului, aproximând problema decupării minime multi-terminal (care este echivalentă în cazul unui singur terminal cu problema fluxului maxim⁠), și pentru aproximarea cuplajului perfect ponderat de cost minim.