**Ministerul Educaţiei și Cercetării al Republicii Moldova**

**Universitatea Tehnică a Moldovei**

**Facultatea Calculatoare, Informatică și Microelectronică**

**RAPORT**

Lucrarea de laborator nr.3

*la Analiza și Proiectarea Algoritmilor*

A efectuat: Buza Cătălin

st. gr. TI-214

A verificat:

asist. univ. Andrievschi-Bagrin Veronica

Chişinău - 2022

**Tema**: Algoritmi greedy

**Scopul lucrarii**:

l. Studierea tehnicii greedy.

2. Analiza si implementarea algoritmilor greedy.

**Note de curs:**

**1. Tehnica greedy**

Algoritmii *greedy* (*greedy = lacom*) sunt in general simpli si sunt folositi Ia rezolvarea problemelor de optimizare, cum ar fi: să se găsească cea mai bună ordine de executare a unor lucrări pe calculator, să se găsească cel mai scurt drum intr-un graf etc. In cele mai multe situalii de acest fel avem:

-*multime de candida*(lucrări de executat, vârfuri ale grafului etc);

-o functie care verifică dacă o anumită multime de candidati constituie o solutie posibilü, nu neapărat optimă, a problemei;

-o functie care verifică dacă o mullime de candidali este fezabilă, adică dacă este posibil să completăm această multime

astfel incât să oblinem 0 solulie posibilă, nu neapărat optimă, a problemei; o functie de selectie care indică la orice moment care este cel mai promitător dintre candidatii incă nefolositi;

-*o functie obiectiv* care dă valoarea unci solutii (timpul necesar executării tuturor lucrărilor intr-o anumită ordine, lungimea drumului pe care I-am găsit etc); aceasta este functia pe care urmărim să o optimizăm (minimizăm/maximizăm).

Pentru a rezolva problema de optimizare, se caută o solutie posibilă care să optimizeze valoarea functiei obiectiv. Un algoritm greedy construieste solutia pas cu pas. Initial, multimea candidaților selectali este vidă. La fiecare pas, se adaugă acestei mullimi cel mai promițător candidat, conform funcției de selecție. Dacă, după o astfel de adăugare, mulțimea de candidați selectați nu mai este fezabilă, se elimină ultimul candidat adăugat; acesta nu va mai fi niciodată considerat. Dacă, după adăugare, mulțimea de candidați selectați este fezabilă, ultimul candidat adăugat va rămâne de acum încolo în ea, De fiecare dată când se lărgește mulțimea candidaților selectați, se verifică dacă această mulțime nu constituie o soluție posibilă a problemei. Dacă algoritmul greedy funcționează corect, prima soluție găsită va fi totodată o soluție optimă a problemei. Soluția optimă nu este în mod necesar unică: se poate că funcția obiectiv să aibă aceeași valoare optimă pentru mai multe soluții posibile. Funcția de selecție este de obicei derivată din funcția obiectiv; uneori aceste două funcții sunt chiar identice.

**2. Arbori de acoperire cu cost minim**

Fie G = M> un graf neorientat conex, unde V este mulțimea vârfurilor și M este mulțimea muchiilor. Fiecare muchie are un cost nenegativ w (sau o lungime nenegativă). Problema este să găsim o submulțime A M, astfel încât toate vârfurile din V să rămână conectate atunci când sunt folosite doar muchii din A, iar suma lungimilor muchiilor din A să fie cat mai mică. Căutăm deci o submulțime A de cost total minim. Această problemă se mai numește și problema conectării orașelor cu cost minim, având numeroase aplicații.

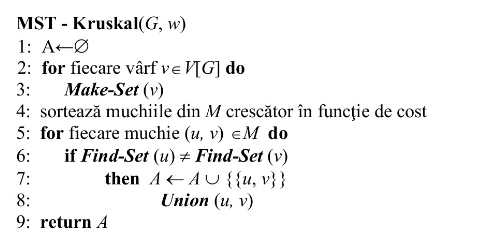
Graful parțial este un arbore și este numit arborele de acoperire minim al grafului G (minimal spanning tree (MST)). Un graf poate avea mai mulți arbori de acoperire de cost minim. Vom prezenta doi algoritmi greedy care determină arborele de acoperire minim al unui graf. În terminologia metodei greedy, vom spune că o mulțime de muchii este o solutie, dacă constituie un arbore de acoperire al grafiilui G, și este fëzabila, dacă nu conține cicluri. O mulțime fezabilă de muchii este promițătoare, dacă poate fi completată pentru a forma soluția optimă. O muchie atinge o mulțime dată de vârfuri, dacă exact un capăt al muchiei este în mulțime. Mulțimea inițiala a candidaților este M. Cei doi algoritmi greedy aleg muchiile una cate una intr-o anumita ordine, această ordine fiind specifică fiecărui algoritm.

**2.1. Algoritmul lui Kruskal**

Arborele de acoperire minim poate fi construit muchie, cu muchie, după următoarea metoda a lui Kruskal (1956): se alege întâi muchia de cost minim, iar apoi se adaugă repetat muchia de cost minim nealeasă anterior și care nu formează cu precedentele un ciclu. Alegem astfel V—I muchii. Este ușor de dedus că obținem în final un arbore.

În algoritmul Iui Kruskal, Ia fiecare pas, graful parțial < V, A> formează o pădure de componente conexe, în care fiecare componentă conexă este Ia rândul ei un arbore de acoperire minim pentru vârfurile pe care le conectează. În final, se obține arborele parțial de cost minim al grafiilui G.

Pentru a implementa algoritmul, trebuie să putem manipula submulțimile formate din vârfurile componentelor conexe. Folosim pentru aceasta 0 structura de date pentru mulțimi disjuncte pentru prezentarea mai multor mulțimi de elemente disjuncte [Corrnen]. Fiecare mulțime conține vârfurile unui arbore din pădurea curentă. Funcția Find-Set (u) returnează un clement reprezentativ din mulțimea care îl conține pe u. Astfel, putem determina dacă două vârfuri u și v aparțin aceluiați arbore testând dacă Find-Set (u) este egal cu Find-Set (v). Combinarea arborilor este realizată de procedura Union. În acest caz, este preferabil să reprezentăm graful ca o lista de muchii cu costul asociat lor, astfel încât să putem ordona această listă în funcție de cost. In continuare este prezentat algoritmul:



**Modul de lucru al algoritmului Kruskal:**

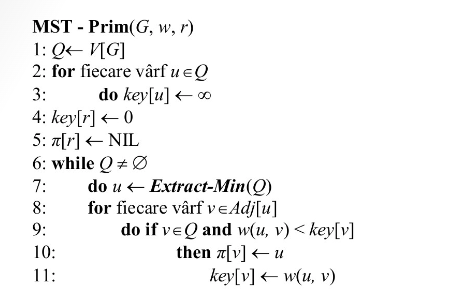
Liniile 1-3 inițializează mulțimea A cu mulțimea vidă și creează IVI arbori, unul pentru fiecare vârf. Muchiile din M sunt ordonate crescător după cost, în linia 4. Bucla for din liniile 5-8 verifică, pentru fiecare muchie (u, v), dacă punctele terminale u și v aparțin aceluiași arbore. Dacă fac parte din același arbore, atunci muchia (u, v) nu poate fi adăugată Ia pădure tără a se forma un ciclu și ea este respinsă. Altfel, cele două vârfuri aparțin unor arbori diferiți, și

muchia (u, v) este adăugată Ia A în linia 7, vârfurile din cei doi arbori fiind reunite în linia 8.

**2.2. Algoritmul lui Prim**

Cel de-al doilea algoritm greedy pentru determinarea arborelui de acoperire minimal al unui graf se datorează lui Prim (1957). În acest algoritm, Ia fiecare pas, mulțimea A de muchii alese împreună cu mulțimea U a vârfurilor pe care le conectează formează un arbore parțial de cost minim pentru subgraful A> al lui G. Inițial, mulțimea U a vârfurilor acestui arbore conține un singur vârf oarecare din V, care va fi rădăcina, iar mulțimea A a muchiilor este vidă. La fiecare pas, se alege o muchie de cost minim, care se adaugă Ia arborele precedent, dând naștere unui nou arbore parțial de cost minim. Arborele parțial de cost minim creste ' 'natural", cu cate o ramură, până când va atinge toate vârfurile din V, adică până când U- V.

Cheia implementării eficiente a algoritmului lui Prim este să procedăm în așa fel încât să fie ușor să selectăm o nouă muchie pentru a fi adăugată Ia arborele format de muchiile din A. În pseudocodul de mai jos, graful conex G și rădăcina r a arborelui minim de acoperire, care urmează a fi dezvoltat, sunt privite ca date de intrare pentru algoritm. În timpul execuției algoritmului, toate vârfurile care nu sunt în arbore se află într-o coadă de prioritate Q bazată pe un câmp key. Pentru fiecare vârf v, key[v] este costul minim al oricărei muchii care îl unește pe v cu un vărf din arbore. Prin convenție, key[v] = dacă nu există o astfel de muchie. Câmpul reține „părintele” lui v din arbore. Adj[u] este lista de adiacență cu vârful u.



Modul de lucru al algoritmului Prim: in liniile 1-4 se ini!ializează coada de prioritate Q, astfel incât aceasta să contină toate vârfurile ii se initializează câmpul key al ticcărui vârfcu T, exceptie ficând rădăcina r, al cărei camp key este initializat cu O. in linia 5 se initializează ;rlr] cu NIL, deoarece rădăcina r nu are nici un părinte.

Pe parcursul executiei algoritmului, multimea V — Q confine vârfurile arborelui curent. In linia 7 este identificat un vârf u€Q incident unei muchii u»are care traversează tăietura (V— Q, Q) (cu exceplia primei iteralii, in care u = v datorită liniei 4). Eliminarea lui u din multimea Q il adaugă pe acesta multimii V — Q a vârfurilor din arbore. in liniile 8-11 se actualizează câmpurile key r ale fiecărui vârf v adiacent lui u, dar care nu se află in arbore. Actualizarea respectă conditiile key[v]= w(v, 7t[vl), si (v, 7T[vl) să fie o muchie usoară care il uneste pe v cu vârf din arbore.

**Implementarea in cod c++**

#include <iostream>

#include <time.h>

#define nr\_maxim 15

using namespace std;

const int INF=999 ;

int muchie[3];

int nr\_varf[3] = {5, 10, 15};

int parinte[nr\_maxim] = { 0 }, vizitat[nr\_maxim] = { 0 };

int grafPrim[nr\_maxim][nr\_maxim],grafKrusk[nr\_maxim][nr\_maxim];

int it1 = 0, it2 = 0;

void afisare\_graf(int nr\_varf)

{

    cout<<"Graful reprezentat prin matricea de adiacenta ponderata:"<<endl;

    for(int i = 0; i < nr\_varf; i++)

    {

        for(int j = 0; j < nr\_varf; j++)

            if(grafPrim[i][j] != 999)

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

            else cout << "0 ";

        cout << endl;

    }

}

void initializare()

{

    it1 = it2 = 0;

    for(int i = 0; i < nr\_maxim; i++)

    {

       vizitat[i] = 0;

        parinte[i] = 0;

        for(int j = 0; j  < nr\_maxim; j++)

        {

            grafPrim[i][j] = 0;

            grafKrusk[i][j] = 0;

        }

    }

}

void creare\_graf(int nr\_varf,int dens)

{

    int k = 0, i = 0, j = 0;

    cout <<"\nNr de muchii: "<<muchie[dens] << endl;

    for(i = 0; i < nr\_varf; i++)

        for(j = i + 1; j < nr\_varf; j++)

        {

            if(k < muchie[dens])

            {

                int r = rand() % 80 + 1;

                grafPrim[i][j] = r;

                grafPrim[j][i] = grafPrim[i][j];

                grafKrusk[i][j] = r;

                grafKrusk[j][i] = grafKrusk[i][j];

                k++;

            }

        }

    for(int i = 0; i < nr\_varf; i++)

        for(int j = 0; j < nr\_varf; j++)

            if(!grafPrim[i][j] && i != j)

            {

                grafPrim[i][j] = INF;

                grafKrusk[i][j] = INF;

            }

}

//============================PRIM==========================

void Prim(int n)

{

    int i, p, min, k,x=0;

    vizitat[x]=1;

    for(i=1; i<=n; i++)

    {

        parinte[i]=x;

        it1++;

    }

   parinte[x]=0;

    for(k=1; k<=n-1; k++)

    {

        min=INF;

        p=0;

        it1++;

        for(i=1; i<=n; i++)

        {

            it1++;

            if(!vizitat[i] && min>grafPrim[i][parinte[i]])

            {

                min=grafPrim[i][parinte[i]];

                p=i;

                it1++;

            }

            vizitat[p]=1;

        }

        for(i=1; i<=n; i++)

        {

            it1++;

            if(!vizitat[i] && grafPrim[i][parinte[i]]>grafPrim[i][p])

               {

                   parinte[i]=p;

                   it1++;

               }

        }

    }

     cout << "Numarul de iteratii(p) : " << it1 << endl;

}

//===============================KRUSKAL==========================

int find(int i)

{

    while(parinte[i])

    {

        i = parinte[i];

        it2++;

    }

    return i;

}

void UNION(int x, int y)

{

    it2++;

    if(x != y)

        parinte[y] = x;

}

void kruskal(int nr\_varf)

{

    int u,v,x,y,n=0,minimum;

    while(n < nr\_varf - 1)

    {

        it2++;

        minimum = INF;

        for(int i = 0; i < nr\_varf; i++)

        {

            it2++;

            for(int j = 0; j < nr\_varf; j++)

            {

                it2++;

                if(grafKrusk[i][j] < minimum && i != j)

                {

                    it2++;

                    minimum = grafKrusk[i][j];

                    x = i;

                    y = j;

                }

            }

        }

        u = find(x);

        v = find(y);

        UNION(u,v);

        grafKrusk[x][y] = grafKrusk[y][x] = INF;

        n++;

    }

    cout << "Numarul de iteratii(k) : " << it2 << endl;

}

int main()

{

    srand(time(NULL));

    for(int i = 0; i < 3; i++)

    {

        muchie[0] = nr\_varf[i]-1;

        muchie[1] = nr\_varf[i] \* (nr\_varf[i]-1)/4;

        muchie[2] = nr\_varf[i] \* (nr\_varf[i]-1)/2;

        for(int j = 0; j < 3; j++)

        {

            switch(j)

            {

            case 0 :

                cout << "Graf caz favorabil cu " << nr\_varf[i] << " varfuri.";

                break;

            case 1 :

                cout << "Graf caz mediu cu " << nr\_varf[i] << " varfuri.";

                break;

            case 2 :

                cout << "Graf caz defavorabil cu " << nr\_varf[i] << " varfuri.";

                break;

            }

            initializare();

            creare\_graf(nr\_varf[i],j);

            Prim(nr\_varf[i]);

            for(int i=0; i<nr\_varf[i];i++)

            {

                parinte[i]=0;

                vizitat[i]=0;

            }

            kruskal(nr\_varf[i]);

            cout << endl;

        }

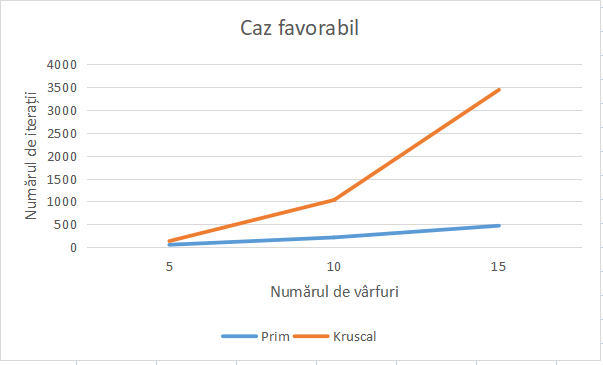
    }

    return 0;

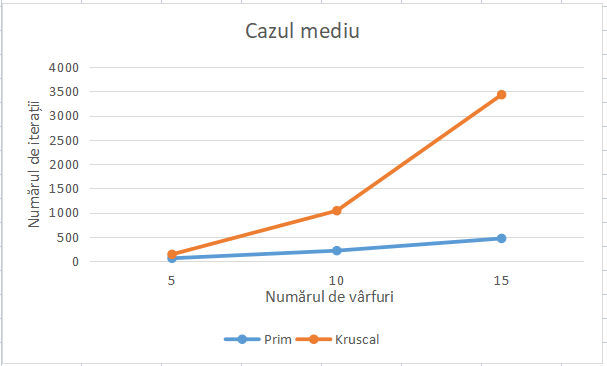
}

Rezultate:

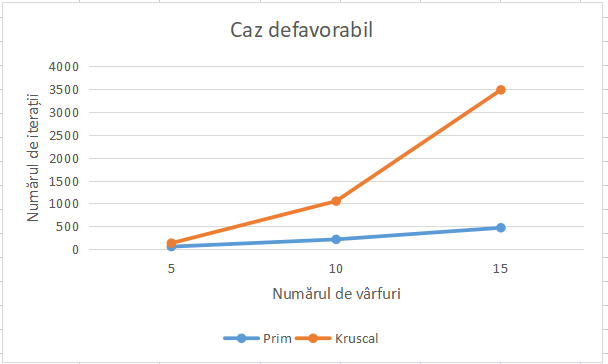
|  |  |  |  |
| --- | --- | --- | --- |
|  |  | Caz favorabil |  |
|  | 5 | 10 | 15 |
| Prim | 55 | 215 | 471 |
| Kruscal | 136 | 1033 | 3438 |



|  |  |  |  |
| --- | --- | --- | --- |
|  |  | Caz mediu |  |
|  | 5 | 10 | 15 |
| Prim | 57 | 214 | 466 |
| Kruscal | 138 | 1038 | 3431 |



|  |  |  |  |
| --- | --- | --- | --- |
|  |  | Caz defavorabil |  |
|  | 5 | 10 | 15 |
| Prim | 58 | 216 | 470 |
| Kruscal | 134 | 1052 | 3481 |



**Concluzie**:

În urma efectuării lucrări de laborator numărul 3 cu tema:,,Algoritmi greedy’’ am studiat 2 algoritmi ce folosesc tehnica greedy pentru a determina graful de acoperire de drum minin: Algoritmul Prim și Kruskal. Pentru a putea determina care algoritm e mai eficient trebuie

să-i studiem pentru 3 cazuri:caz favorabil,mediu și defavorabil. Cazul favorabil constă că graful nostru este deja un graf de acoperire(m=n-1).Cazul defavorabil reprezintă cazul când graful nostru este unul complet adică toate vîrfurile formează muchii cu restu vârfurilor grafului(m=n\*(n-1)/2).Cazul mediu reprezintă cazul în care numărul de muchii este cuprins între n-1<m< n\*(n-1)/2; În urma cercetării algoritmilor pentru toate cazuri s-a observat ca algoritmul Prim efectuează mai puține iterații pentru rezolvarea problemei propuse decât Kruscal,iar numărul de iterații pentru toate 3 cazuri nu diferă mult unul de altul.