**Ministerul Educaţiei și Cercetării al Republicii Moldova**

**Universitatea Tehnică a Moldovei**

**Facultatea Calculatoare, Informatică și Microelectronică**

**RAPORT**

Lucrarea de laborator nr.4

*la Analiza și Proiectarea Algoritmilor*

A efectuat:

st. gr. TI-214 Buza Cătălin

A verificat:

asist. univ. Andrievschi-Bagrin Veronica

Chişinău - 2022

**Tema:** Metoda programării dinamice

**Scopul lucrării:**

1. Studierea metodei programării dinamice.

2. Analiza si implementarea algoritmilor de programare

dinamică.

3. Compararea tehnicii greedy cu metoda de programare

dinamică.

**Note de curs:**

**1. Programarea dinamică.**

O problemă rezolvabilă prin metoda programării dinamice trebuie adusă mai intåi Ia o formă discretă in timp. Deciziile care se iau pentru a obtine un rezultat trebuie să se poată lua pas cu pas. De asemenca, foarte importantă este ordinea in care acestea se iau. Programarea dinamică este (si nu luati aceste rånduri ca pe o definitie) in esentă un proces decizional in mai multe etape: in starea inilială a problemei Iuăm prima decizie, care determină o nouă stare a problemei in care luăm o decizie. Termenul dinamic se referă chiar la acest lucru: problema este rezolvată in etape dependente de timp. Variabilele, sau functiile care descriu fiecare etapă trebuie să fie in aia fel definite incåt să descrie complet un proces, deci pentru acest lucru va trebui să răspundem la două intrebări:

-care este etapa initială (caz in care avem de a face cu un proces decizional descendent) sau care este etapafinală (caz in care avcm de a face cu un proces decizional ascendent)?

-care este regula după care trecem dintr-o etapă in alta ? De

obicei această regulă este exprimată printr-o recurentă.

Deoarece, avem de a face cu o problemă care se rezolvă in mai multe etape, nu ne mai rămåne decåt să vedem cum luăm deciziile dintr-o etapă in alta.

De exemplu, problema calculului numerelor lui Fibonaci se incadrează in categoria programării dinamice deoarece:

-este un proces in etape;

-fiecărei etape k ii corespunde calculul celui de al k-lea număr Fibonacci;

-există o singură decizie pentru a trece la o etapă superioară;

Determinarea unui drum ce leagă două A B ii care trece printr-un număr minim de alte orase este tot o problemă de

programare dinamică deoarece: este un proces in etape, fiecărei etape k ii corespunde determinarea unui drum de

lungime k ce pleacă din orasul A, dar există mai multe decizii pentru trecerea la drumul de lungimea k + l.

In cele ce urmează prin strategie inlelegem un sir de decizii.

Conform principiului lui Bellman, numit principiul optimalitătii avem:

***O strategie are proprietatea oricare ar fi starea initială 'I decicia initială, deciziile rămase trebuie să constituie o strategie optimă privitoare la starea care rezultă din decizia anterioară.***

Demonstrarea corectitudinii unui algoritm de programare dinamică se face, aia cum rezultă ii din principiul optimalitătii, prin induc(ie matematică.

**Definitia 1.** Fie P problema care trebuie rezolvată. Simbolul P codifică problema initială impreună cu dimensiunea datelor de intrare. O subproblemă Qi (care este rezolvată la etapa i) are aceea$ formă ca P, dar datele de intrare pe care le prelucrează sunt mai mici in comparatie cu cele cu care lucrează P. Cuvåntul dinamic vrea să sugereze tocmai acest lucru: uniformitatea subproblemelorsi rezolvarea Ior in mod ascendent. Pe scurt Qi se obline din P printr-o restriclionare a datelor de intrare.

Din această definilie rezultă asa-zisa dependenlă (mai precis incluziune) dintre subprobleme. O subproblemă Qi obline alte subprobleme dacă datele asupra cărora operează Qi\_sunt mai mari decat datele asupra cărora operează subproblemele continute, deci nu este vorba doar de o simplă dependentă intre subprobleme, acestea fiind incluse una in cealaltă.

Există două tipuri de subprobleme: directe care rezultă din relalia de recurentă si subprobleme indirecte care de fapt sunt sub-sub-subprobleme ale problemei initiale. De exemplu, in cazul numerelor lui Fibonacci, pentru determinarea termenului F(5) subproblemele directe sunt F(4) F(3), iar F(2), F(I) F(O) sunt subprobleme indirecte.

Definirea relatiilor dintre etape se face recursiv. Prin prisma unui matematician acest lucru este inconvenient, dar orice informaticianitie că recursivitatea inseamnă resurse (timp memorie) consumate. Pe långă inconvenientele legate de apelurile recursive, o subproblemă este calculată de mai multe ori. Aceste lucruri pot fi evitate dacă subproblemcle se calculează incepånd de jos in sus (adică de Ia cea mai "mică" Ia cea mai "mare") 'i se relin rezultatele obtinute. in cazul numerelor Fibonacci, cea mai mică subproblemă este calcularea lui Fibonacci(O) si a lui Fibonacci(l), iar cea mai mare subproblemă este calcularea lui Fibonacci(S) unde N este un număr dat.

O problemă de programare dinamică se poate prezenta sub forma unui graforientat. Fiecărui nod ii corespunde o etapă (sau o subproblemă), iar din relaliile de recurență se deduce modul de adăugare a arcelor. Mai precis, vom adăuga un arc de la starea (etapa) i la starea (etapa)j dacă stareaj depinde direct de starea i. Dependenta directă dintre etape este dată de relatiile de recurentă. Constmirea unui astfel de graf este echivalentă cu rezolvarea problemei. Determinarea sirului deciziilor care au adus la solutie se reduce la o problemă de drum in grafuri.

Notăm cu P, problema pe care o dorim să o rezolvăm. Intelesul pe care il dăm lu P include 'i dimensiunea datelor de la intrare. Am spus anterior că 0 etapă, sau 0 subproblemă are aceea$ formă ca si problema de rezolvat la care sunt adăugate cåteva restrictii.

Pentru ca aceste probleme să poată fi calculate trebuie stabilită 0 ordine in care ele vor fi prelucrate. Considerånd reprezentarea sub forma unui graf (nodurile corespund etapelor, muchiilor deciziilor), va trebui să efectuăm o sortare topologică asupra nodurilor grafului. Fie Q), QI,. QN ordinea rezultată unei astfel de sortări. Prin Q, am notat o subproblemă (Qo este subproblema cea mai mică). In cazul submultimii de sumă dată, Qi este chiar descompunerea lui i in sumă de numere din vectorul dat. Fie Si solulia problemei Q. Solulia subproblemei QN este solulia

problemei P.

Algoritmul general este:

**procedure** Rezolva(P)

{initializează (So) }

l: **for** i el to Ndo

2: **progresează** (So, ... , S,-l, S); { aceasta procedura calculează solutia problemei Q pe baza solutiilor problemelor Q), ..., QI-I, cu o recurentă dejos in sus.}

3: **return** (Sn)

Demonstrarea corectitudinii unui astfel de algoritm se face prin inductie după i. Referitor la complexitatea unui algoritm de programare dinamică, putem spune că aceasta depinde de mai mulli factori: numărul de stări, numărul de decizii cu care se poate trece intr-o Stare, complexitatea subproblemei iniliale . Ceea ce apare insă in toate problemele, este numărul de stări (etape). Deci complexitatea va avea forma O(N• .. .). Notiunea de complexitate pseudo — polinomială, de asemenea este legată uneori de complexitatea unui algoritm de programare dinamică, Fie D multimea tuturor intrărilor corecte pentru o problemă dată. Definim două funclii:

Lungimea: D —4 Z. Functia Max indică valoarea maximă a datelor de intrare, iar functia Lungimea indică lungimea Ior. Un algoritm este numit algoritm pseudo — polinomial, dacă functia sa de complexitate este mărginită superior de o functie polinomială in două variabile: Max[l] 'i Lungime[l]. Prin definilie orice algoritm polinomial este ii pseudo — polinomial, deoarece se execută intr-un timp mărginit de un polinom de Lungime[l], dar nu toti algoritmii pseudo-polinomiali sunt polinomiali. Pentru problemele care au proprietatea că Max[/] este mărginită de 0 funclie polinomială in variabila Lungime [l] nu există nici o distinctie intre algoritmii polinomiali si cei pseudo-polinomiali, Să vedem cånd se poate si cånd nu se poate construi un algoritm pseudo-polinomial? Spunem despre o problemă că este number-

problem dacă nu există nici o functie polinomială p, astfel incåt Max[ll Sp(LungimeVl) pentru orice intrare I corectă. Singurele probleme NP-complete care sunt candidate pentru a fi rezolvate prin algoritmi pseudo-polinomială se află in această clasă de probleme.

Există mai multe clasificări ale problemelor de programare dinamică:

l. După natura deciziilor pot fi deterministe sau nedeterministe. La cele nedeterministe, in scrierea relatiilor de recurentă intervine o probabilitate (probabilitatea ca produsul să fie våndut, ca individul x să ajungă intr-un punct dat, ca moneda să cadă pe partea cu stema, ca o masină să se strice după ce a produs un număr de piese.)

2. După valoarea deciziilor pot fi de optimizare (deteminarea drumurilor de cost minim intre două noduri ale unui graf), de neoptimizare (deteminarea numerelor Fibonacci).

3. După numărul de dimensiuni (sau numărul de parametri ai functiilor de recurentă) pot fi unidimensionale (precum numerele lui Fibonacci), sau multidimensionale.

4. După numărul deciziilor care se pot lua Ia un pas pot fi unidecizionale (ca numerele lui Fibonacci), sau multidecizionale (ca drumul de lungime minimă intre două noduri ale unui gram Problemele unidecizionale sunt de obicei de neoptimizare.

5. După directia deciziilor putem avea programare dinamică inainte in care starea i se calculează in funclie de stările i + I, i + 2, , sau programare dinamică inapoi in care starea i se calculea7.ă in funclie de stările i— l, i — 2, .

Dezvoltarea unui algoritm bazat pe programarea dinamică poate fi impărtită intr-o secventă de patru paii:

l. Caracterizarea structurii unei solutii optime.

2. Definirea recursivă a valorii unei solulii optime.

3. Calculul valorii unei solutii optime intr-o manieră de tip

"bottom-up".

1. Construirea unei solutii optime din informatia calculată. Pasii 1-3 sunt baza unei abordări de tip programare dinamică. Pasul 4 poate fi omis dacă se doreste doar calculul unei singure solutii optime. In vederea realizării pasului 4, deseori se păstrează informatie suplimentară de la executia pasului 3, pentru a usura constructia unei solutii optimale.

**2. Probleme de drum minim in grafuri**

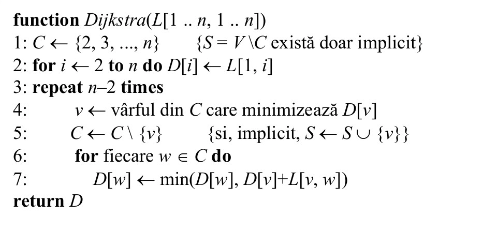
**2.1. Cele mai scurte drumuri care pleacă din acelasi punct**

Fie G=<V,A> un graf oriental, unde V este multimea vårfurilor A este multimea arcelo. Fiecare arc are 0 lungime nenegativa. Unul din vårfuri este ales că vårf sursă. Problema este de a determina lungimea celui mai scurt drum de la sursă către fiecare vårfdin graf. Se va folosi un algoritm greedy, datorat lui Dijkstra (1959).

Notăm cu C multimea vårfurilor disponibile (candidatii) cu S mullimea vårfurilor deja selectate. in fiecare moment, S contine acele vårfuri a căror distantă minimă de la sursă este deja cunoscută, in timp ce mullimea C contine toate celelalte vårfuri. La inceput, S contine doar vårful sursă, iar in final S contine toate vårfurile grafului. La fiecare pas, adăugam in S acel vårf din C a cărui distantă de la sursă este cea mai mică. Se spune, că un drum de Ia sursă cåtre un alt vårf este special, dacă toate vårfurile intermediare de-a lungul drumului apanin lui S. Algoritmul lui Dijkstra in felul următor. La fiecare pas al algoritmului, un tablou D conline lungimea celui mai scurt drum special către fiecare vårf al grafului. După ce se adaugă un nou vårf v la S, cel mai scurt drum special cåtre v va fi, de asemenea, cel mai scurt dintre toate drumurile cåtre v. Cånd algoritmul se termină, toate vårfurile din graf sunt in S, deci toate drumurile de Ia sursă citre celelalte vårfuri sunt speciale 'i valorile din D reprezintă solutia problemei.

Presupunem că vårfurile sunt numerotate, V— {I, 2, n}, vårful 1 fiind sursa, că matricea L dă lungimea fiecărui arc, cu = -Fm, dacă arcul (i, j) nu există. Solutia se va construi in tabloul D[2 n].

Algoritmul este:



**Proprietatea 1.** In algoritmul lui Dijkstra, dacă un vårf i

a) este in S, atunci D[i] dă lungimea celui mai scurt drum de Ia sursă cåtre i;

b) nu este in S, atunci D[i] dă lungimea celui mai scurt drum special de la sursă citre i. La terminarea algoritmului, toate vårfurile grafului, cu exceptia unuia, sunt in S. Din proprietatea precedenta, rezulta că algoritrnul lui Dijkstra funclionează corect.

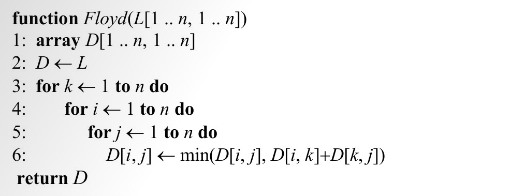
**2.2. Determinarea celor mai scurte drumuri intr-un graf**

Fie un graf orientat, unde V este multimea vårfurilor si A este mullimea arcelor. Fiecărui arc i se asociază o lungime nenegativă. Să se calculeze lungimea celui mai scurt drum intre fiecare pereche de varfuri. Vom presupune că vårfurile sunt numerotate de la 1 la n ii că matricea L dă lungimea fiecărui arc: L[i, i] = 0, 2 0 pentru i \*j, = +T dacă arcul (i,j) nu există.

Principiul optimalitătii este valabil: dacă cel mai scurt drum de la i laj trece prin varful k, atunci porliunea de drum de la i la k, cåt cea de la k laj, trebuie să fie, de asemenea, optime.

Construim o matrice D care să contină lungimea celui mai scuff drum intre fiecare pereche de vårfuri. Algoritmul de programare dinamică initializează pe D cu L. Apoi, efectuează n iteratii. După iteratia k, D va contine lungimile celor mai sculte drumuri care folosesc ca vårfuri intermediare doar vårfurile din k}. După n iteratii, oblinem rezultatul final. La iteratia k, algoritmul trebuie să verifice, pentru fiecare pereche de vårfuri (i, j), dacă există sau nu un drum, trecånd prin varful k, care este mai bun decit actualul drum optim ce trece doar prin vărfurile din k—l}. Fie Dk matricea D după iteratia k. Verificarea necesară este atunci: = IDA-I [i, k] + Dk-l [k, j]) unde s-a facut uz de principiul optimalității pentru a calcula lungimea celui mai scurt drum faw de k. Implicit, s-a considerat că un drum optim care trece prin k nu poate trece de două ori prin k.

Acest algoritm simplu este datorat lui Floyd (1962):



Se poate deduce că algoritmul lui Floyd necesită un timp in Un alt mod de a rezolva această problemă este să se aplice algoritmul Dijkstra prezentat mai sus de n ori, alegånd mereu un alt vårf sursă. Se obtine un timp in n adică tot in Algoritmul lui Floyd, datorită simplitălii lui, are insă constanta multiplicativă mai mică, fiind probabil mai rapid in practică.

**Codul**:

#include<iostream>

#include<iomanip>

using namespace std;

#define INF 999

#define MAX 15

int nod\_start, cost[MAX][MAX];

int distanta[MAX];

bool vizitat[MAX]= { 0 };

int parent[MAX];

int it1,it2;

int virfuri[3]={5,10,15},muchii[3];

void initializare(int n)

{

    it1=it2=0;

    for(int i=0 ; i < n ; i++)

    {

        distanta[i] = INF ;

        parent[i]=i;

    }

    distanta[nod\_start] = 0 ;

}

void creare\_graf(int n,int dens){

    for(int i = 0; i < n; i++)

        for(int j = 0; j < n; j++)

            if(j > i){

                int r = rand() % 10 + 1;

                if(r <= muchii[dens]){

                    r = rand() % 20 + 1;

                    cost[i][j] = r;

                }

            } else if(i > j){

                cost[i][j] = cost[j][i];

    for(int i=0; i<n; i++)

        for(int j=0; j<n; j++)

        {

          if(i!=j && cost[i][j]==0)

          {

              cost[i][j]=INF;

          }

        }

}

}

void printGraf(int nrVarf){

    for(int i = 0; i < nrVarf; i++){

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

            cout <<setw(4)<<cost[i][j] << " ";

        cout << endl;

    }

}

int getNearest(int n)

{

    int minvalue =INF ;

    int minnode=0;

    for(int i =0 ; i<n; i++)

    {

        it1++;

        if(!vizitat[i] && distanta[i] < minvalue)

        {

            minvalue = distanta[i] ;

            minnode = i;

           it1++;

        }

    }

    return minnode ;

}

void dijkstra(int n)

{

    for(int i=0; i<n; i++)

    {

        int k = getNearest(n) ;

        vizitat[k]=true ;

        it1++;

        for(int j=0; j < n ; j++)

        {

            it1++;

            if(cost[k][j]!=INF && distanta[j]>distanta[k]+cost[k][j] )

            {

                it1++;

                distanta[j]=distanta[k]+cost[k][j];

                parent[j]=k ;

            }

        }

    }

    //afisare\_drum\_dijkstra(n);

    cout<<"\nNumarul de iteratii Dijkstra: "<<it1<<endl;

}

void afisare\_drum\_dijkstra(int n)

{

    cout<<"Node:\t\t\tCost :\t\t\tPath \n";

    for(int i=0; i<n; i++)

    {

        cout<<i+1<<"\t\t\t"<<distanta[i]<<"\t\t\t"<<" ";

        cout<<i+1<<" ";

        int parnode=parent[i];

        while(parnode!=nod\_start)

        {

            cout<<" <-- "<<parnode+1<<" ";

            parnode=parent[parnode];

        }

        cout<<" <-- "<<nod\_start+1;

        cout<<endl;

    }

}

void afisare\_drum\_floyd(int n,int D[][MAX]);

void floyd(int n)

{

    int D[n][n], i, j, k;

    for (i = 0; i < n; i++)

    {

        it2++;

        for (j = 0; j < n; j++)

        {

            D[i][j] = cost[i][j];

            it2++;

        }

    }

    for (k = 0; k < n; k++)

    {

        it2++;

        for (i = 0; i < n; i++)

        {

            it2++;

            for (j = 0; j < n; j++)

            {

                D[i][j]=min(D[i][j],D[i][k]+D[k][j]);//Dacă vârful k se află pe calea cea mai scurtă de la i la j,

                it2++;                               //atunci se actualizeaza valoarea lui D[i][j]

            }

        }

    }

    cout<<"Numarul de iteratii Floyd: "<<it2<<endl;

    //afisare\_drum\_floyd(D);

}

void afisare\_drum\_floyd(int n,int D[][MAX])

{

    cout<<"\n------------------------Algoritmul floyd-----------------------------";

    cout<<endl<<"Matricea distantelor minime:"<<endl;

    for (int i = 0; i < n; i++)

    {

        for (int j = 0; j < n; j++)

        {

            if (D[i][j] == INF)

                cout<<setw(4)<<"INF";

            else

                cout<<setw(4)<<D[i][j];

        }

        cout<<endl;

    }

}

int main()

{

    for(int i = 0; i < 3; i++){

        muchii[0]=virfuri[i]-1;

        muchii[1]=virfuri[i]\*(virfuri[i]-1)/4;

        muchii[2]=virfuri[i]\*(virfuri[i]-1)/2;

        for(int j = 0; j < 3; j++)

        {

            switch(j)

            {

            case 0 :

                cout << "\nCaz favorabil  " << virfuri[i] << " varfuri." << endl;

                break;

            case 1 :

                cout << "\nCaz mediu  " << virfuri[i] << " varfuri." << endl;

                break;

            case 2 :

                cout << "\nCaz defavorabil  " << virfuri[i] << " varfuri." << endl;

                break;

            }

            initializare(virfuri[i]);

            creare\_graf(virfuri[i],j);

            printGraf(virfuri[i]);

            dijkstra(virfuri[i]);

            floyd(virfuri[i]);

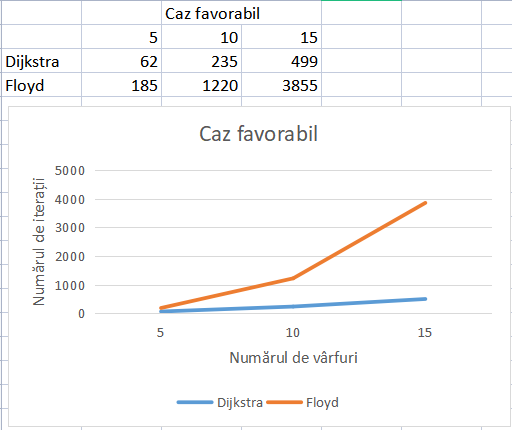
        }

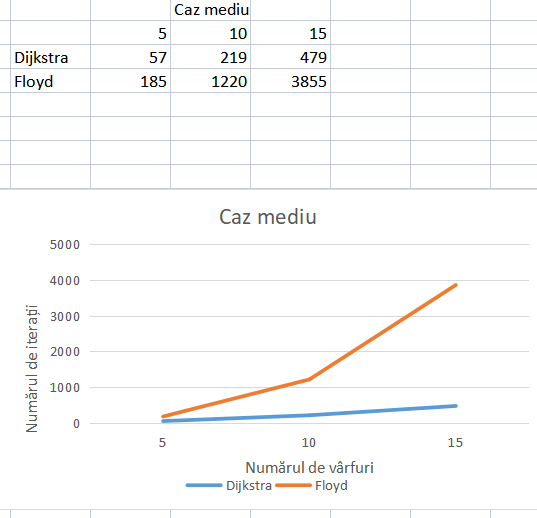
    }

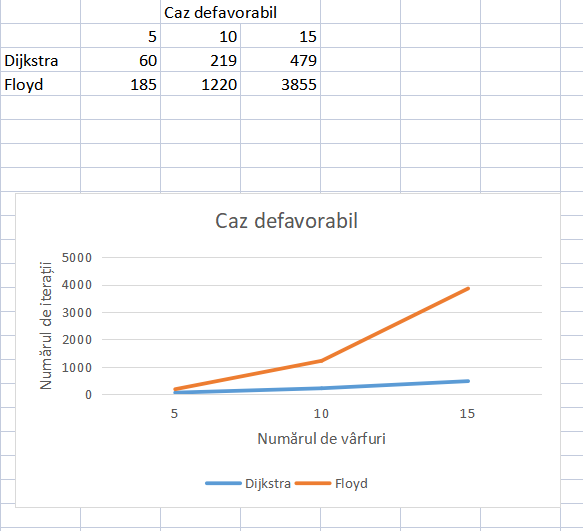
    return 0;

}

Rezultate:







**Concluzie**:

Efectuind lucrarea data m-am facut cunoscutina cu Metoda programării dinamice am analizat eficienta programarii dinamice pentru diferite cazuri ale problemei.

Implementind si analizind algoritmele Dijksta si Floyd am concluzionat ca acesti algoritmi sunt rapizi si deci sunt algoritmi eficienti si foarte buni pentru rezolvarea problemei de construire a arborelui de cost minim.

Observ ca dindui numarul de virufuri in cazul meu 5 10 15 in algoritmului Dijkstra numarul de iteratii atit in cazul cu un numar de arce dens si rar ramine acelas doar ca incepe a varia in dependenta dupa numarul de virfuri, timpul de executie creste cu marirea numarului de virfuri.

In cazul algoritmului Floyd iteratiile nu depind de densitatea arcelor ci doar de numarul de virfuri si semnificativ timpul de executie este mai mic decit la Dijkstra in schimb numarul de iteratii este mult mai mare.