Universitatea Tehnica a Moldovei

Catedra Automatica şi Tehnologii Informaționale

Disciplina:

Analiza şi proiectarea algoritmilor

RAPORT

Lucrare de laborator Nr. 4

Tema: Metoda programării dinamice

A efectuat : studentul grupei TI-151 Poseletchi Cristi

A verificat: Bagrin Veronica

Chișinău 2016

1. Scopul lucrării:

1. Studierea metodei programării dinamice.

2. Analiza şi implementarea algoritmilor de programare dinamică.

3. Compararea tehnicii greedy cu metoda de programare dinamică.

2. SARCINA DE BAZĂ:

1. De studiat metoda programării dinamice de proiectare a algoritmilor.

2. De implementat într-un limbaj de programare algoritmii lui Dijkstra și Floyd.

3. De făcut analiza empirică a acestor algoritmi pentru un graf rar şi pentru un graf dens.

4. De alcătuit un raport.

3. Indicații teoretice

3.1. Programarea dinamică.

Programarea dinamică este (şi nu luați aceste rânduri ca pe o definiție) în esență un proces decizional în mai multe etape: în starea inițială a problemei luăm prima decizie, care determină o nouă stare a problemei în care luăm o decizie. Termenul dinamic se referă chiar la acest lucru: problema este rezolvată în etape dependente de timp. Variabilele, sau funcțiile care descriu fiecare etapă trebuie să fie în aşa fel definite încât să descrie complet un proces, deci pentru acest lucru va trebui să răspundem la două întrebări:

- care este etapa *inițială* (caz în care avem de a face cu un proces decizional descendent) sau care este etapa *finală* (caz în care avem de a face cu un proces decizional ascendent)?
- care este *regula* după care trecem dintr-o etapă în alta ? De obicei această regulă este exprimată printr-o *recurență*.

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

În cele ce urmează prin strategie înțelegem un şir de decizii. Conform principiului lui Bellman, numit ***principiul optimalității*** avem:

O strategie are proprietatea că oricare ar fi starea inițială şi decizia inițială, deciziile rămase trebuie să constituie o strategie optimă privitoare la starea care rezultă din decizia anterioară.

Definiția 1. Fie P problema care trebuie rezolvată. Simbolul P codifică problema inițială împreună cu dimensiunea datelor de intrare. O subproblemă Qi (care este rezolvată la etapa i) are aceeaşi formă ca P, dar datele de intrare pe care le prelucrează sunt mai mici în comparație cu cele cu care lucrează P. Cuvântul dinamic vrea să sugereze tocmai acest lucru: uniformitatea subproblemelor şi rezolvarea lor în mod ascendent. Pe scurt Qi se obține din P printr-o restricționare a datelor de intrare.

Există două tipuri de subprobleme: *directe* care rezultă din relația de recurență şi subprobleme *indirecte* care de fapt sunt sub-sub-subprobleme ale problemei inițiale. De exemplu, în cazul numerelor lui Fibonacci, pentru determinarea termenului $F(5)$ subproblemele directe sunt $F(4)$ şi $F(3)$, iar $F(2)$, $F(1)$ şi $F(0)$ sunt subprobleme indirecte.

Definirea relațiilor dintre etape se face recursiv. Prin prisma unui matematician acest lucru este inconvenient, dar orice informatician ştie că recursivitatea înseamnă resurse (timp şi memorie) consumate. Pe lângă inconveniențele legate de apelurile recursive, o subproblemă este calculată de mai multe ori. Aceste lucruri pot fi evitate dacă subproblemele se calculează începând de jos în sus (adică de la cea mai "mică" la cea mai "mare") şi se rețin rezultatele obținute.

O problemă de programare dinamică se poate prezenta sub forma unui graf orientat. Fiecărui nod îi corespunde o etapă (sau o subproblemă), iar din relațiile 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 starea j depinde direct de starea i. Dependența directă dintre etape este dată de relațiile de recurență.

Construirea unui astfel de graf este echivalentă cu rezolvarea problemei. Determinarea şirului deciziilor care au adus la soluție se reduce la o problemă de drum în grafuri. Notăm cu P, problema pe care o dorim să o rezolvăm. Înțelesul pe care îl dăm lui P include şi dimensiunea datelor de la intrare.

Am spus anterior că o etapă, sau o subproblemă are aceeaşi formă ca şi problema de rezolvat la care sunt adăugate câteva restricții.

Pentru ca aceste probleme să poată fi calculate trebuie stabilită o ordine în care ele vor fi prelucrate. Considerând reprezentarea sub forma unui graf (nodurile corespund etapelor, muchiilor - deciziilor), va trebui să efectuăm o sortare topologică asupra nodurilor grafului. Fie Q_0, Q_1,…, Q_N ordinea rezultată unei astfel de sortări. Prin Q_i am notat o subproblemă (Q_0 este subproblema cea mai mică). În cazul submulțimii de sumă dată, Q_i este chiar descompunerea lui i în sumă de numere din vectorul dat. Fie S_i soluția problemei Q_i. Soluția subproblemei Q_N este soluția problemei P.

Algoritmul general este:

procedure Rezolva(P)

{inițializează (S_0) }

1: **for** $i \leftarrow$ 1 to N do

2: **progresează** (S_0, …, S_{i-1}, S_i); { aceasta procedura calculează soluția problemei Q_i pe baza soluțiilor

problemelor Q_0, …, Q_{i-1}, cu o recurență de jos în sus.}

3**: return** (S_N)

Demonstrarea corectitudinii unui astfel de algoritm se face prin inducție după i.

Referitor la complexitatea unui algoritm de programare dinamică, putem spune că aceasta depinde de mai mulți factori: numărul de stări, numărul de decizii cu care se poate trece într-o stare, complexitatea subproblemei inițiale (Q_0)… . Ceea ce apare însă în toate problemele, este numărul de stări (etape). Deci complexitatea va avea forma $O(N\cdot\ldots)$.

Dezvoltarea unui algoritm bazat pe programarea dinamică poate fi împărțită într-o secvență de patru paşi:

1. Caracterizarea structurii unei soluții optime.
2. Definirea recursivă a valorii unei soluții optime.
3. Calculul valorii unei soluții optime într-o manieră de tip "bottom-up".
4. Construirea unei soluții optime din informația calculată.

Paşii 1-3 sunt baza unei abordări de tip programare dinamică. Pasul 4 poate fi omis dacă se doreşte doar calculul unei singure soluții optime. In vederea realizării pasului 4, deseori se păstrează informație suplimentară de la execuția pasului 3, pentru a uşura construcția unei soluții optimale.

3.2 Cele mai scurte drumuri care pleacă din acelaşi punct

Fie $G = <V, A>$ un graf orientat, unde V este mulțimea vârfurilor şi A este mulțimea arcelo. Fiecare arc are o lungime nenegativa. Unul din vârfuri este ales că vârf *sursă*. Problema este de a determina lungimea celui mai scurt drum de la sursă către fiecare vârf din graf.

Se va folosi un algoritm greedy, datorat lui Dijkstra (1959). Notăm cu C mulțimea vârfurilor disponibile (candidații) şi cu S mulțimea vârfurilor deja selectate. În fiecare moment, S conține acele vârfuri a căror distanță minimă de la sursă este deja cunoscută, în timp ce mulțimea C conține toate celelalte vârfuri. La început, S conține doar vârful sursă, iar în final S conține toate vârfurile grafului. La fiecare pas, adăugam în S acel vârf din C a cărui distanță de la sursă este cea mai mică.

Se spune, că un drum de la sursă către un alt vârf este *special*, dacă toate vârfurile intermediare de-a lungul drumului aparțin lui S. Algoritmul lui Dijkstra lucrează în felul următor. La fiecare pas al algoritmului, un tablou D conține lungimea celui mai scurt drum special către fiecare vârf al grafului. După ce se adaugă un nou vârf v la S, cel mai scurt drum special către v va fi, de asemenea, cel mai scurt dintre toate drumurile către v. Când algoritmul se termină, toate vârfurile din graf sunt în S, deci toate drumurile de la sursă către

celelalte vârfuri sunt speciale şi valorile din D reprezintă soluția problemei.

Presupunem că vârfurile sunt numerotate, $V = \{1, 2, ..., n\}$, vârful 1 fiind sursa, şi că matricea L

dă lungimea fiecărui arc, cu $L[i, j] =$, dacă arcul (i, j) nu există. Soluția se va construi în tabloul $D[2 .. n]$. Algoritmul este:

function *Dijkstra*($L[1 .. n, 1 .. n]$)
1: $C \leftarrow \{2, 3, ..., n\}$ $\{S = V \setminus C$ există doar implicit$\}$
2: **for** $i \leftarrow 2$ **to** n **do** $D[i]$
$L[1, i]$
3: **repeat** n–2 **times**
4: $v \leftarrow$ vârful din C care minimizează $D[v]$
5: $C \leftarrow C \setminus \{v\}$ {si, implicit, $S \leftarrow S \cup \{v\}\}$
6: **for** fiecare $w \in C$ **do**
7: $D[w] \leftarrow \min(D[w], D[v]+L[v, w])$
return D

Proprietatea 1. În algoritmul lui Dijkstra, dacă un vârf i

a) este în S, atunci $D[i]$ dă lungimea celui mai scurt drum de la sursă către i;

b) nu este în S, atunci $D[i]$ dă lungimea celui mai scurt drum special de la sursă către i.

La terminarea algoritmului, toate vârfurile grafului, cu excepția unuia, sunt în S. Din proprietatea precedenta, rezulta că algoritmul lui Dijkstra funcționează corect.

2.2 Determinarea celor mai scurte drumuri intr-un graf

Fie $G = \langle V, A \rangle$ un graf orientat, unde V este mulțimea vârfurilor şi A este mulțimea arcelor. Fiecărui arc i se asociază o lungime nenegativă. Să se calculeze lungimea celui mai scurt drum între fiecare pereche de varfuri.

Vom presupune că vârfurile sunt numerotate de la 1 la n şi că matricea L dă lungimea fiecărui arc:

$L[i, i] = 0$, $L[i, j] \geq 0$ pentru $i \neq j$, $L[i, j] = \propto$ dacă arcul (i, j) nu există.

Principiul optimalității este valabil: dacă cel mai scurt drum de la i la j trece prin varful k, atunci porțiunea de drum de la i la k, cât şi cea de la k la j, trebuie să fie, de asemenea, optime.

Construim o matrice D care să conțină lungimea celui mai scurt drum între fiecare pereche de vârfuri. Algoritmul de programare dinamică inițializează pe D cu L. Apoi, efectuează n iterații. După iterația k, D va conține lungimile celor mai scurte drumuri care folosesc ca vârfuri intermediare doar vârfurile din $\{1, 2, ..., k\}$. După n iterații, obținem rezultatul final. La iterația k, algoritmul trebuie să verifice, pentru fiecare pereche de vârfuri (i, j), dacă există sau nu un drum, trecând prin varful k, care este mai bun decât actualul drum optim ce trece doar prin vârfurile din $\{1, 2, ..., k$ D_k matricea D după iterația k. Verificarea necesară este atunci:

$D_k[i, j] = \min(D_{k-1}[i, j], D_{k-1}[i, k]\ D_{k-1}[k, j])$

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):

function *Floyd*($L[1 .. n, 1 .. n]$)
1: **array** $D[1 .. n, 1 .. n]$
2: $D \leftarrow L$
3: **for** $k \leftarrow 1$ **to** n **do**
4: **for** $i \leftarrow 1$ **to** n **do**
5: **for** $j \leftarrow 1$ **to** n **do**
6: $D[i, j] \leftarrow \min(D[i, j], D[i, k]+D[k, j])$
return D

Se poate deduce că algoritmul lui Floyd necesită un timp în $O(n^3)$. Un alt mod de a rezolva această problemă este să se aplice algoritmul *Dijkstra* prezentat mai sus de n ori, alegând mereu un alt vârf sursă. Se obtine un timp în $n\ O(n^2)$, adică tot în $O(n^3)$. Algoritmul lui Floyd, datorită simplității lui, are însă constanta multiplicativă mai mică, fiind probabil mai rapid în practică.

4. Analiza empirică

Grafuri dense

Grafurile dense sunt grafurile a căror număr de muchii e aproximativ egal cu card(V)^2.

Tabelul. 1 Timpul de execuție în dependență de n pentru fiecare algoritm

n	20	60	100	140	180	220	260	300	340
dijkstra	0	0.002	0.005	0.008	0.013	0.023	0.031	0.047	0.047
floyd	0.15	0.204	0.735	1.693	3.539	6.659	11.272	16.256	23.982

Tabelul. 2 Numărul de iterații în dependență de n

n	20	60	100	140	180	220	260	300	340
dijkstra	400	3600	10000	19600	32400	48400	67600	90000	115600
floyd	7600	212400	990000	2724400	579960	10599600	17508400	26910000	39188400

Graficele construite conform datelor din tabele:

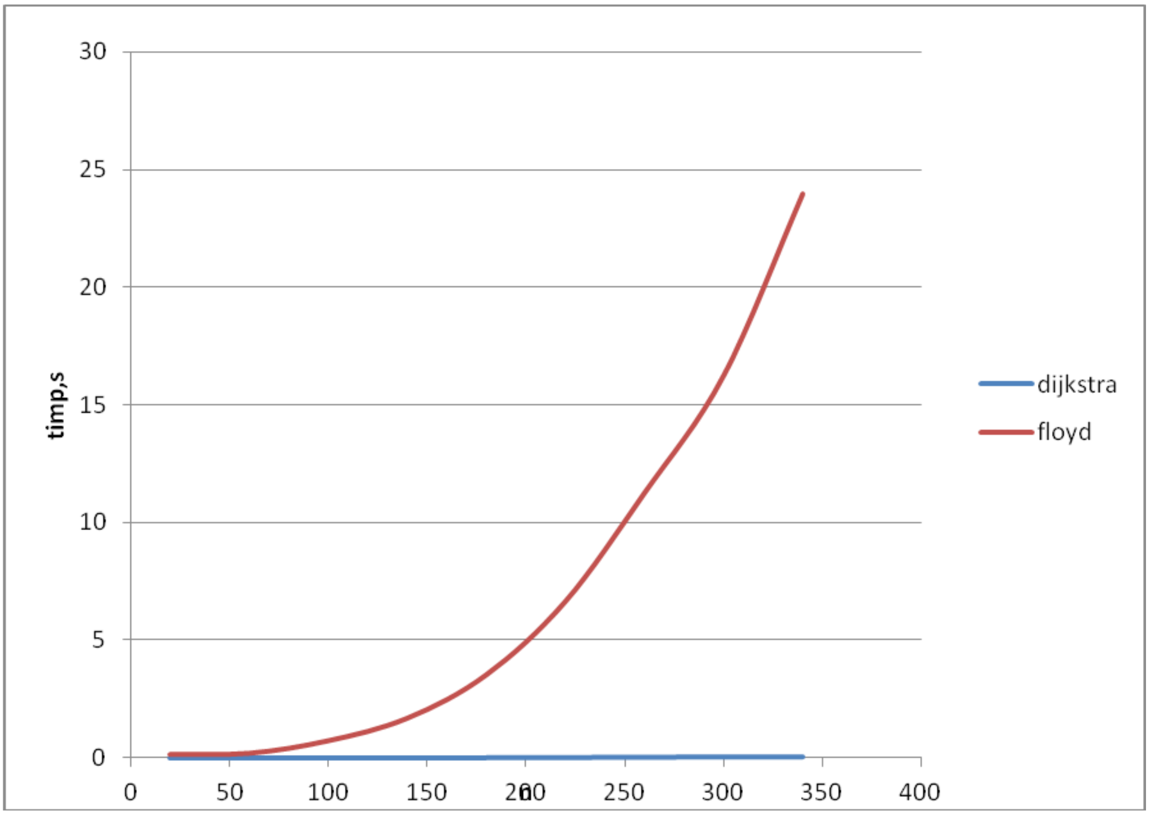

Fig.1 Graficul dependenței timpului de execuție față de numărul de vârfuri pentru un graf dens .

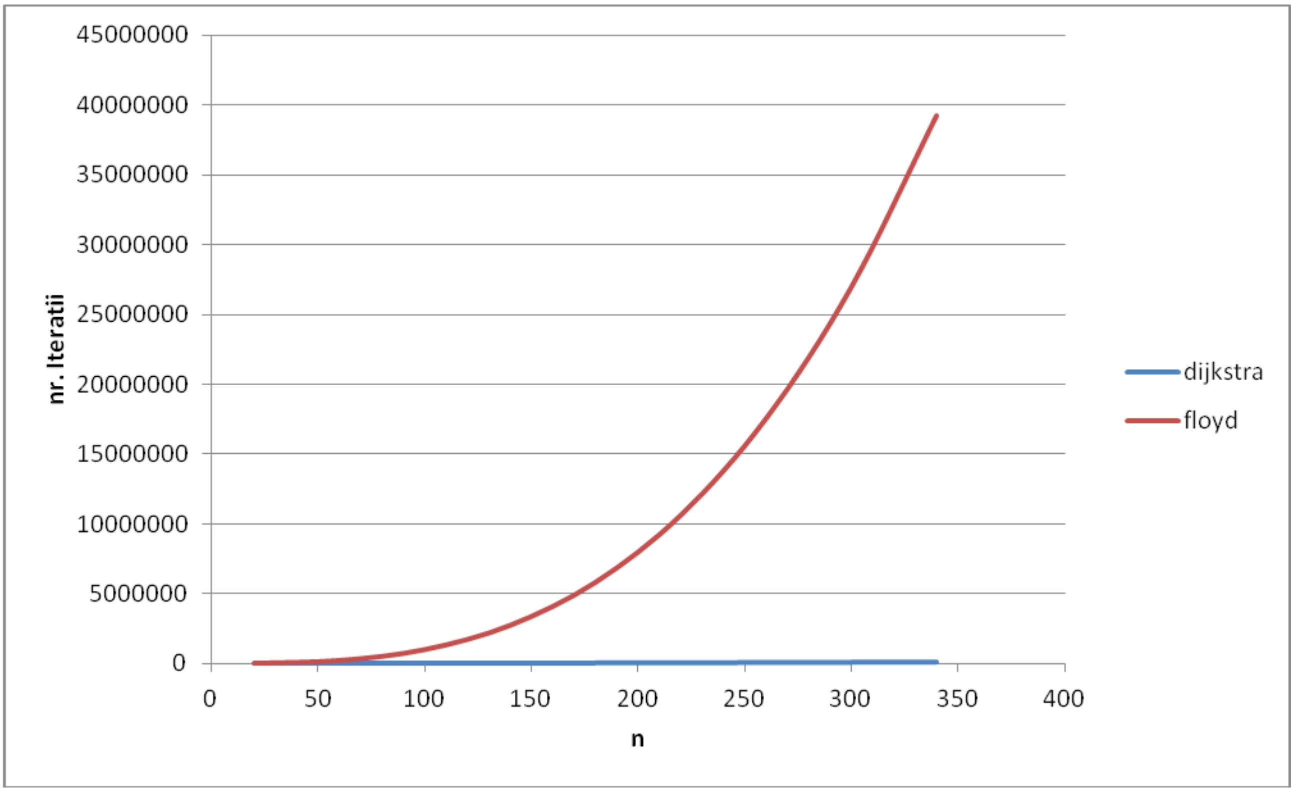

Fig.2 Graficul dependenței numărului de iterații față de numărul de vârfuri pentru un graf aleatoriu.

Grafuri rare

Grafurile rare sunt grafurile a căror număr de muchii e foarte mic

Tabelul. 3 Timpul de execuție în dependență de n pentru fiecare algoritm

n	20	60	100	140	180	220	260	300	340
dijkstra	0.001	0.002	0.008	0.008	0.013	0.019	0.026	0.04	0.048
floyd	0.096	0.244	0.726	1.674	3.539	6.559	10.952	17.083	23.425

Tabelul. 4 Numărul de iterații în dependență de n

n	20	60	100	140	180	220	260	300	340
dijkstra	400	3600	10000	19600	32400	48400	67600	90000	115600
floyd	7600	212400	990000	2724400	5799600	10599600	17508400	26910000	39188400

Graficele construite conform datelor din tabele:

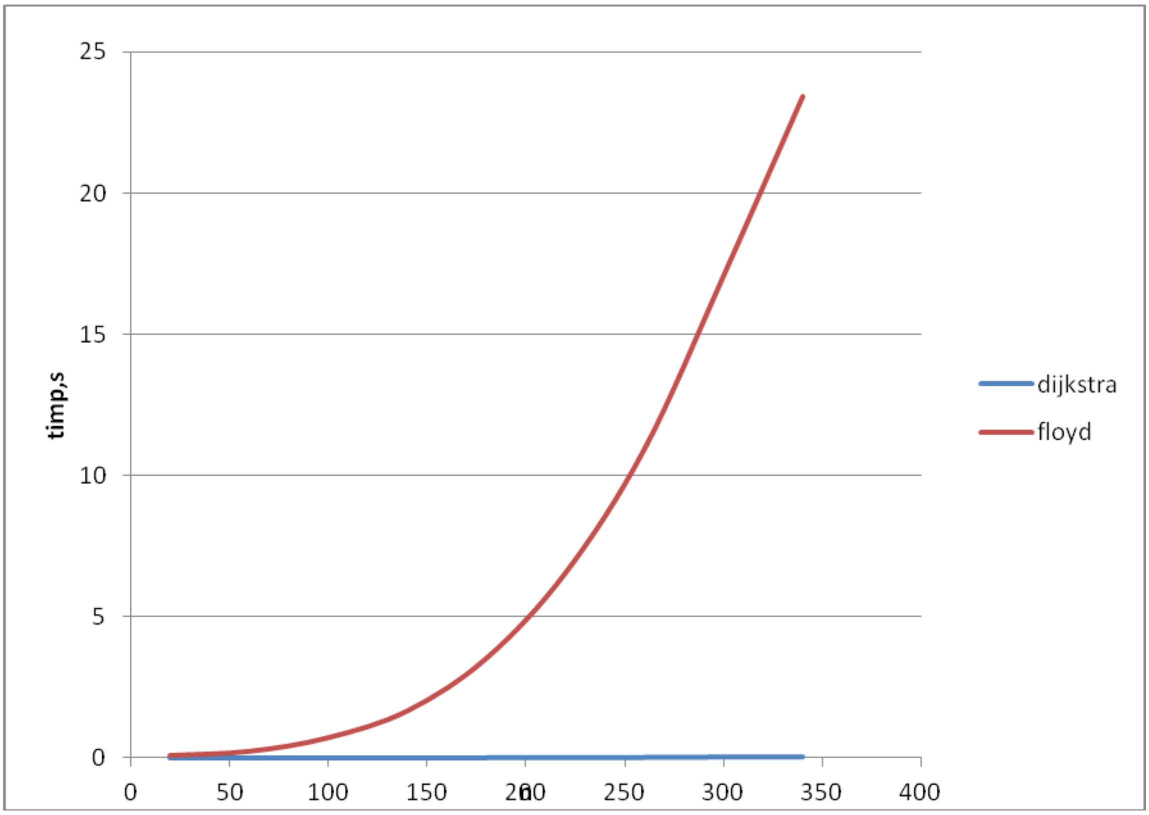

Fig.1 Graficul dependenței timpului de execuție față de numărul de vârfuri pentru un graf dens .

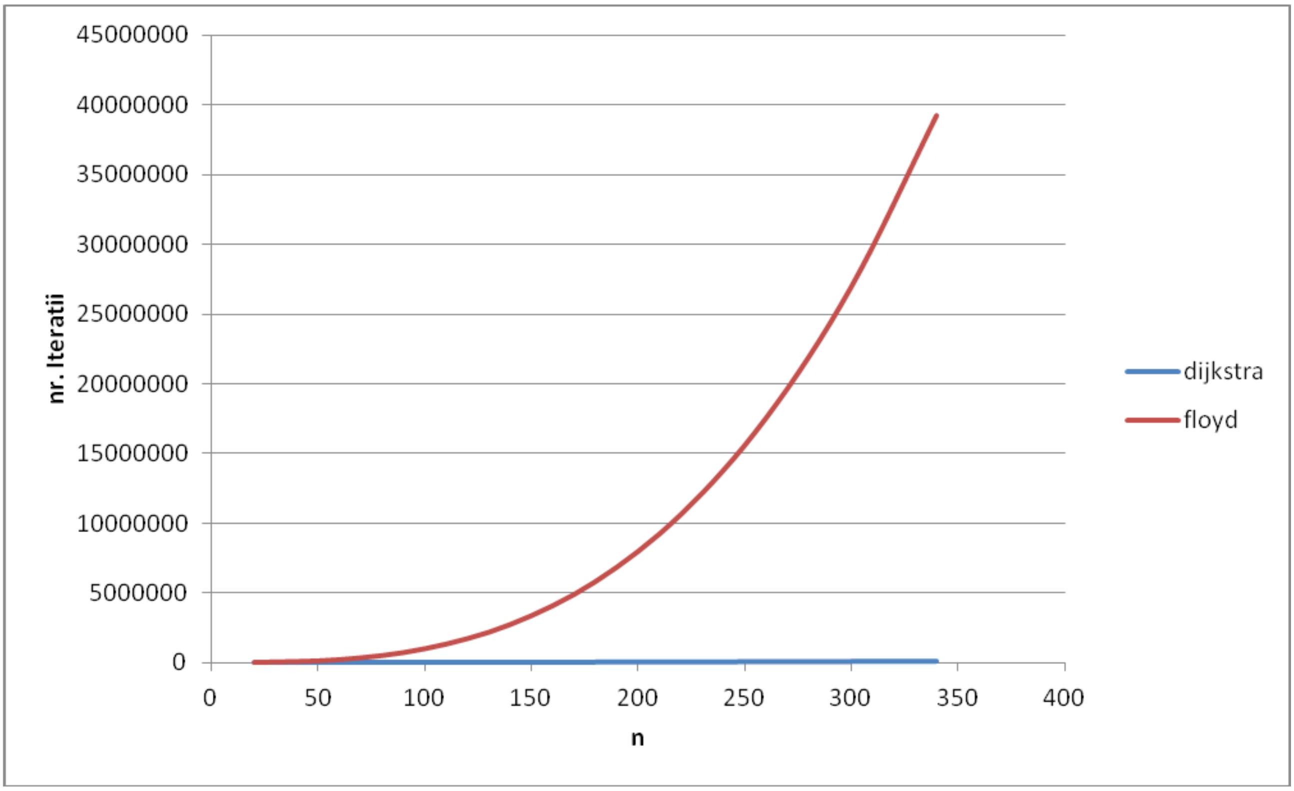

Fig.2 Graficul dependenței numărului de iterații față de numărul de vârfuri pentru un graf aleatoriu.

Concluzii

Analizând graficele observăm că atât pentru grafuri rare cât și pentru grafuri dense numărul de iterații nu diferă, deci atât grafuriule rare cât și cele dense nu reprezintă pentru acești algoritmi cazuri favorabile sau defavorabile. Timpul de execuție diferă nesemnificativ, cauza acestei diferențe este faptul că pentru grafurile dense linia de cod din corpul comparației la căutarea maximului se va efectua de mai multe ori decât le cele rare.

Observăm că datele exeperimentale coinicid aproximativ cu datele teoretice astfel pentru exemplul n=20, numărul de iterații pentru algoritmul Dijkstra e 400, iar pentru Floyd 7600, ce un număr aproape de 8000, deci primul are complexitatea $O(n^2)$, iar al doilea $O(n^3)$. Observăm deci şi faptul că algoritmul lui Floyd nu e altceva decât algoritmul lui Dijkstra aplicat pentru fiecare vârf al grafului.

Bibliografia

- Răzvan Andonie, Ilie Gârbacea, ”Algoritmi fundamentali, o perspectivă a C++”, Editura Libris ,Cluj-Napoca 1995.
- Îndrumar de laborator APA Nr.1.
- Ciclu prelegeri APA.

Anexa A

Listingul programului pentru implementarile Prim si Kruskal

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <conio.h>
#include <time.h>
#include <vector>
#define MAXINT 37000
using namespace std;
struct lista
{
 int inf;
 lista *urm,*ultim;
};

class graf
{
public:
   int n;
   vector<vector<int>> adiacenta;
         graf():n(0),adiacenta(0,vector<int>(0,0)){};
  // int nr_muchii;
   //(0,vector<int>(0,0));
   int iter_dij,iter_floyd;
   int citire_graf();
   void implicit();
   void aleator_dens();
    void aleator_rar();
         void implicit2();
          void dijkstra(int,vector<int>&,vector<int>&);
          void floyd(vector<vector<int>>&,vector<vector<int>>&);

};

 void sterg_ad(vector<vector<int>> &adiacenta)
          {
                    if(adiacenta.size())
                    {
                              for(int i=0;i<adiacenta.size();i++)
            adiacenta[i].clear();
          adiacenta.clear();
                    }
          }
          void init_ad(vector<vector<int>> &adiacenta,int n,int val)
          {
                    adiacenta.assign(n,vector<int>(n,val));
          }
void afis_matrice(vector<vector<int>> &adiacenta)
   {
         int n=adiacenta.size();
   for(int i=0;i<n;i++)
   {
    for(int j=0;j<n;j++)
      if(adiacenta[i][j]==MAXINT) cout<<setw(5)<<"inf";
            else cout<<setw(5)<<adiacenta[i][j];
    cout<<endl;
    }
}
 void afis_vect(vector<int> x)
          {
```

```
                for (int j=0;j<x.size();j++) cout<<setw(4)<<x[j];
            cout<<endl;
        }
 void graf::floyd(vector<vector<int>>&D,vector<vector<int>>&T)
 {
        sterg_ad(D);
        sterg_ad(T);
        iter_floyd=0;
        D.assign(adiacenta.begin(),adiacenta.end());
        for(int i=0;i<n;i++)
                D[i].assign(adiacenta[i].begin(),adiacenta[i].end());
        init_ad(T,n,0);
        for(int k=0;k<n;k++)
        {
                for(int i=0;i<n;i++)
                {
                        if(i!=k)
                         for(int j=0;j<n;j++)
                         {
                                iter_floyd++;
                                 if(j!=i)
                                        if(D[i][j]>D[i][k]+D[k][j])
                                        {
                                                D[i][j]=D[i][k]+D[k][j];
                                                T[i][j]=k;
                                        }
                         }
                }
        }

 }
void graf::dijkstra(int x,vector<int > &d,vector<int> &t)
{

 int i,j;
 vector<int> s(n,0);
 this->iter_dij=0;
 d.clear();
 t.clear();
 d.assign(n,0);
 t.assign(n,0);
 int min,poz;
 for (i=0;i<n;i++) {
        iter_dij++;
   d[i]=adiacenta[x][i];
    if (i!=x && d[i]<MAXINT) t[i]=x;
 }
 for (i=0;i<n-1;i++) {
   min=MAXINT;
   for (j=0;j<n;j++)
        {

     if (!s[j])
        if (min>d[j]) {
              min=d[j];
              poz=j;
        }
        }
   s[poz]=1;
   for (j=0;j<n;j++)
        {iter_dij++;
        if (!s[j])
          if (d[j]>d[poz]+adiacenta[poz][j]) {
              d[j]=d[poz]+adiacenta[poz][j];
              t[j]=poz;
```

```
            }
            }
 }
}
void meniu()
{
  char opt;
  graf *G1=new graf,*G2=NULL;
  G1->implicit2();
 // G1->n=5;
  //G1->aleator_dens();
 // G1->n=10;
// G1->aleator_rar();
     float t1,t2,t3,t4;
             vector<int> t,d;
             vector<vector<int>> D,L;
             int k;
  do
  {
     system("cls");
     cout<<"1.Citire graf:"<<endl;
     cout<<"2.Afis muchii:"<<endl;
     cout<<"3.Djikstra"<<endl;
     cout<<"4.Floyd"<<endl;
     cout<<"5.Timpi de executie grafuri dense"<<endl;
     cout<<"6.Timpi de executie grafuri rare"<<endl;
     cout<<"0.Exit"<<endl;
     cout<<"Citeste optiunea:";
     opt=getch();
     system("cls");
    switch(opt)
     {
      case '1':
         //if (G1->n) delete G1;
         G1->citire_graf();
         break;
      case '2':
                        afis_matrice(G1->adiacenta);
        break;
      case '3':
         G1->dijkstra(0,d,t);
                       cout<<"virful intial:"<<0<<endl;
                       cout<<"vectorul lungimilor minime"<<endl;
                       afis_vect(d);
                       cout<<"vectorul drumurilor minime"<<endl;
                       afis_vect(t);
         break;
      case '4':
                        G1->floyd(D,L);
                         cout<<"matricea lungimilor minime"<<endl;
                        afis_matrice(D);
                          cout<<"matricea drumurilor minime"<<endl;
                        afis_matrice(L);

         break;
       case '5':
          for(int i=20;i<=360;i+=40)
          {
             if(G2){ delete G2; G2=NULL;}
             G2=new graf;
             G2->n=i;
             G2->aleator_dens();
             cout<<endl<<"n="<<i<<endl;
             t1=clock();
                                G2->dijkstra(0,d,t);
```

```
           t2=clock();
           t3=clock();
                           G2->floyd(D,L);
          t4=clock();
          cout<<"Dijkstra:\ntimp  "<<(t2-t1)/CLOCKS_PER_SEC<<endl;
                          cout<<"iteratii:"<<G2->iter_dij<<endl;
          cout<<endl<<"FLOYD:\ntimp "<<(t4-t3)/CLOCKS_PER_SEC<<endl;
            cout<<"iteratii:"<<G2->iter_floyd<<endl;
           }
        break;
        case '6':
      for(int i=20;i<=360;i+=40)
         {
           if(G2){ delete G2; G2=NULL;}
           G2=new graf;
           G2->n=i;
           G2->aleator_rar();
           cout<<endl<<"n="<<i<<endl;
           t1=clock();
                           G2->dijkstra(0,d,t);
           t2=clock();
           t3=clock();
                           G2->floyd(D,L);
          t4=clock();
          cout<<"Dijkstra:\ntimp  "<<(t2-t1)/CLOCKS_PER_SEC<<endl;
                          cout<<"iteratii:"<<G2->iter_dij<<endl;
          cout<<endl<<"FLOYD:\ntimp "<<(t4-t3)/CLOCKS_PER_SEC<<endl;
            cout<<"iteratii:"<<G2->iter_floyd<<endl;
           }
        break;
        case '0':exit(0);
        default : cout<<"optiune incorecta";
      }
      cout<<endl;
      system("pause");
     }
     while(1);
}



int main()
{
  meniu();
    return 0;
}



int graf::citire_graf()
{
cout<<"da nr de noduri  ";
cin>>n;
sterg_ad(adiacenta);
init_ad(adiacenta,n,MAXINT);
cout<<"da lungimile muchiilor(0 -nu e drum)::\n";
for(int i=0;i<n;i++)
    {
        for(int j=0;j<n;j++)
        {
        cout<<i<<" , "<<j<<"  :";
        int w;
        cin>>w;
                if(w==0) adiacenta[i][j]=MAXINT;
        else adiacenta[i][j]=w;
```

```
       }
                    adiacenta[i][i]=0;
    }
return 0;
}

void graf::aleator_dens()
{
  int k;
sterg_ad(adiacenta);
init_ad(adiacenta,n,MAXINT);
for(int i=0;i<n;i++)
  {adiacenta[i][i]=0;
   for(int j=0;j<n;j++)
     {
        do {
        k=(int)rand()%20;
        }while(k==0);
        adiacenta[i][j]=k;

     }
   adiacenta[i][i]=0;
     }
   }

 void graf::aleator_rar()
{
  int k;
sterg_ad(adiacenta);
init_ad(adiacenta,n,MAXINT);
  for(int i=0;i<n;i++)for(int j=0;j<n;j++) adiacenta[i][j]=MAXINT;
  for(int i=0;i<n-1;i++)
  {
        do {
        k=(int)rand()%20;
        }while(k==0);
        adiacenta[i][i+1]=k;
        adiacenta[i+1][i]=k;
     }
   adiacenta[1][n-1]=adiacenta[n-1][1]=13;
  }
void graf::implicit()
{
   n=7;
  sterg_ad(adiacenta);
init_ad(adiacenta,n,MAXINT);
   for(int i=0;i<n;i++) adiacenta[i][i]=0;
   adiacenta[0][1]=10;
   adiacenta[0][4]=5;
   adiacenta[1][2]=5;
   adiacenta[1][4]=20;
   adiacenta[2][3]=5;
   adiacenta[2][4]=15;
   adiacenta[3][4]=2;
   adiacenta[3][5]=3;
   adiacenta[4][5]=50;
   adiacenta[4][6]=60;
   adiacenta[5][6]=10;
   adiacenta[6][0]=10;
  }

void graf::implicit2()
{
   n=4;
  sterg_ad(adiacenta);
```

```
init_ad(adiacenta,n,MAXINT);
   for(int i=0;i<n;i++) adiacenta[i][i]=0;
   adiacenta[0][1]=5;
   adiacenta[1][0]=50;
   adiacenta[1][2]=15;
   adiacenta[1][3]=5;
   adiacenta[2][0]=30;
   adiacenta[2][3]=15;
   adiacenta[3][0]=15;
   adiacenta[3][2]=5;
  }
```

Anexa B

Rezultatele execuției prog Anexa A

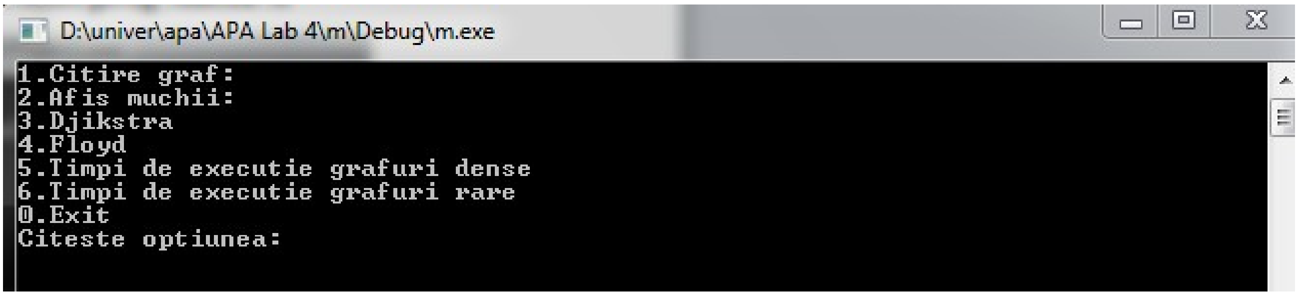

Fig. 4Meniul principal

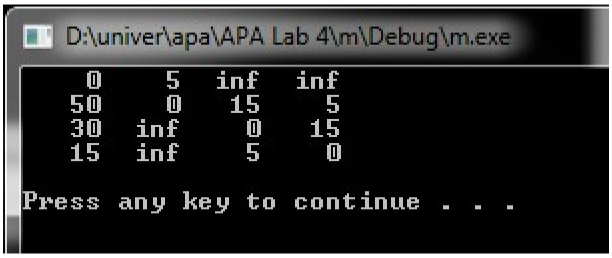

Fig. 5 optiunea 2

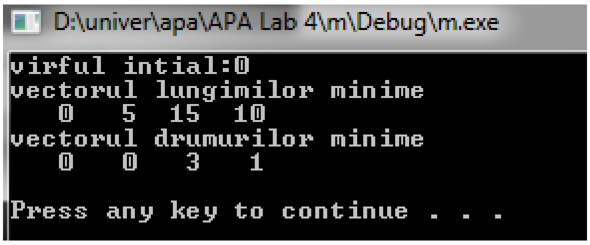

Fig. 6 optiunea 3

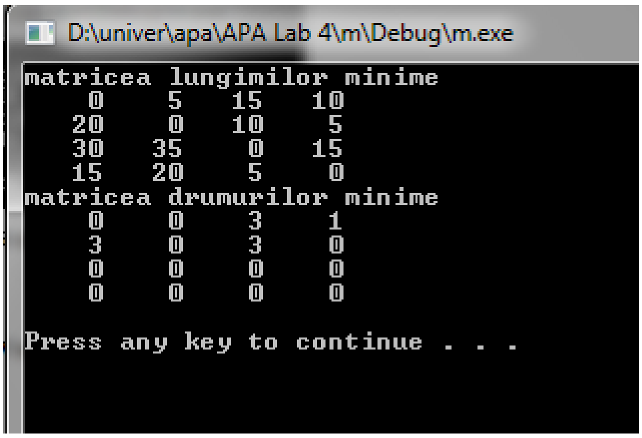

Fig. 7 optiunea 4

```
D:\univer\apa\APA Lab 4\m\Debug\m.exe

n=20
Dijkstra:
timp  0
iteratii:400

FLOYD:
timp 0.038
iteratii:7600

n=60
Dijkstra:
timp  0.004
iteratii:3600

FLOYD:
timp 0.263
iteratii:212400

n=100
Dijkstra:
timp  0.007
iteratii:10000

FLOYD:
timp 0.937
iteratii:990000

n=140
Dijkstra:
timp  0.011
iteratii:19600

FLOYD:
timp 1.779
iteratii:2724400

n=180
Dijkstra:
timp  0.012
iteratii:32400

FLOYD:
timp 3.801
iteratii:5799600

n=220
Dijkstra:
timp  0.02
iteratii:48400

FLOYD:
timp 6.843
iteratii:10599600
```

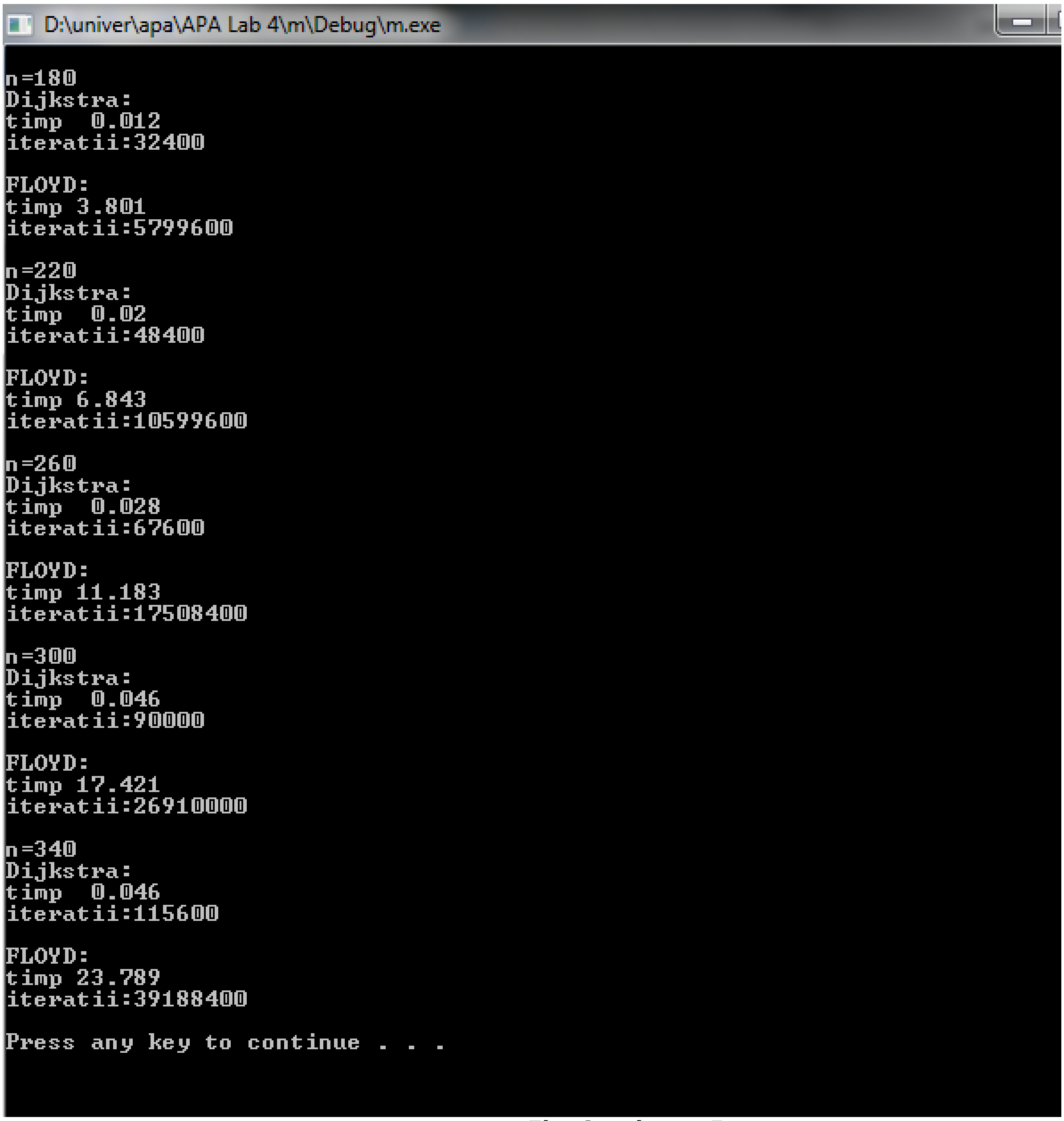

Fig. 8optiunea 5

```
D:\univer\apa\APA Lab 4\m\Debug\m.exe

n=20
Dijkstra:
timp  0
iteratii:400

FLOYD:
timp 0.201
iteratii:7600

n=60
Dijkstra:
timp  0.002
iteratii:3600

FLOYD:
timp 0.195
iteratii:212400

n=100
Dijkstra:
timp  0.005
iteratii:10000

FLOYD:
timp 0.887
iteratii:990000

n=140
Dijkstra:
timp  0.01
iteratii:19600

FLOYD:
timp 1.757
iteratii:2724400

n=180
Dijkstra:
timp  0.015
iteratii:32400

FLOYD:
timp 5.013
iteratii:5799600

n=220
Dijkstra:
timp  0.03
iteratii:48400

FLOYD:
timp 9.912
iteratii:10599600
```

```
FLOYD:
timp 9.912
iteratii:10599600

n=260
Dijkstra:
timp  0.036
iteratii:67600

FLOYD:
timp 13.299
iteratii:17508400

n=300
Dijkstra:
timp  0.094
iteratii:90000

FLOYD:
timp 21.356
iteratii:26910000

n=340
Dijkstra:
timp  0.056
iteratii:115600

FLOYD:
timp 27.345
iteratii:39188400

Press any key to continue . . .
```

Fig. 9optiunea 6