Grafuri

• Parcurgerea DF a grafurilor neorientate

Fie `G=(V,M)` un graf neorientat. Ca de obicei, notăm `n=|V|` şi `m=|M|`.

Pentru reprezentarea grafului, fiecărui vârf `i` sunt precizate informaţia asociată, precum şi lista L_i a vecinilor săi.

Parcurgere în adâncime a grafurilor (*DF = D*epth *F*irst) generalizează parcurgerea în preordine a arborilor oarecare. Eventuala existenţă a ciclurilor conduce la necesitatea de a marca vârfurile vizitate

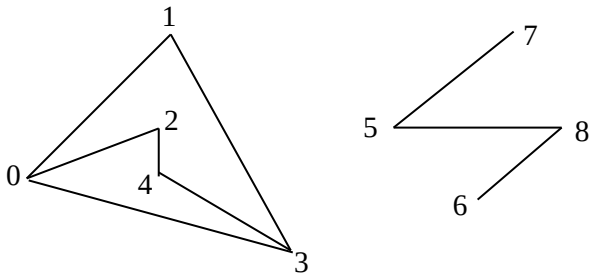
Programul principal este:

citeşte graful;
```
for i=0,n-1
   atins(i)←f
for i=0,n-1
   if !atins(i) then DF(i)
```

unde:

```
procedure DF(i)
   atins(i)←t; vizit(i);
   for toţi j∈Li
      if !atins(j) then DF(j)
```

Vom folosi pentru exemplificare următorul graf:



cu următoarele liste ale vecinilor vârfurilor:

L_0={3,1,2};	L_1={0,3};	L_2={0,4};
L_3={0,1,4};	L_4={2,3};	L_5={7,8};
L_6={8};	L_7={5}	L_8={5,6}

DF(0):	DF(3):	DF(1):	
		DF(4):	DF(2)
	~~DF(1)~~		
	~~DF(2)~~		

Observaţia 1. Dacă dorim doar să determinăm dacă un graf este conex, vom înlocui al doilea ciclu `for` din programul principal prin:

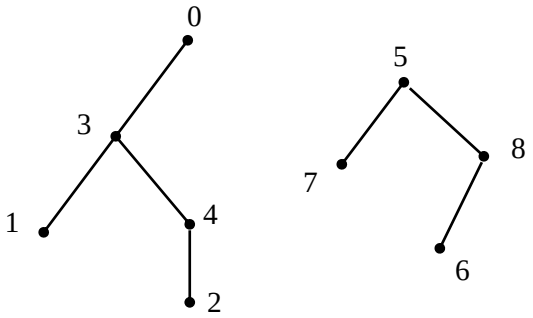
```
DF(0);
```
şi ţinem în `ndf` evidenţa numărului de vârfuri atinse. În final:
```
if ndf=n then write('CONEX')
         else write('NECONEX');
```

Observaţia 2. Parcurgerea DF împarte muchiile grafului în:
- *muchii de avansare*: sunt acele muchii `(i,j)` pentru care în cadrul apelului `DF(i)` are loc apelul `DF(j)`. Aceste muchii formează un graf parţial care este o pădure: fiecare vârf este vizitat exact o dată, deci nu există un ciclu format din muchii de avansare. Arborii din această pădure se numesc *arbori DF*.
- *muchii de întoarcere*: sunt acele muchii ale grafului care nu sunt muchii de avansare.

Pentru determinarea arborilor DF alegem pentru arbori reprezentarea cu listele F_i ale fiilor vârfurilor `i`. Aceste liste sunt iniţializate ca vide, iar procedura DF devine:

```
procedure DF(i)
   atins(i)←t; vizit(i);
   for toţi j∈Li
      if !atins(j)
      then Fi ← j; DF(j)
```

Pentru graful din exemplul considerat, pădurea DF este formată din următorii arbori parţiali corespunzători componentelor conexe:



Timpul cerut de algoritmul de mai sus este `O(max{n,m})=O(n+m)`, adică liniar în `n` şi `m`, deoarece:
- pentru fiecare vârf `i`, apelul `DF(i)` are loc exact o dată;
- executarea unui apel `DF(i)` necesită un timp proporţional cu `grad(i)=|`L_i`|`; în consecinţă timpul total va fi proporţional cu `m=|M|`;
- evident, pentru un graf complet, timpul devine $O(n^2)$.

Propoziţie. Dacă `(i,j)` este muchie de întoarcere, atunci `i` este descendent al lui `j` în arborele parţial ce conţine pe `i` şi `j`.

Muchia `(i,j)` este detectată ca fiind muchie de întoarcere în cadrul executării apelului `DF(i)`, deci `j` a fost atins înaintea lui `i` şi există drum de la `j` la `i` în arborele DF curent (din `i` s-a ajuns la `j` prin muchii de avansare).

Propoziţia de mai sus spune că muchiile de întoarcere leagă totdeauna două vârfuri situate pe aceeaşi ramură a unui arbore DF. În particular, nu există muchii de traversare (care să lege doi descendenţi ai aceluiaşi vârf dintr-un arbore DF).

Observaţia 3. Un graf este ciclic (conţine cel puţin un ciclu) dacă şi numai dacă în timpul parcurgerii sale în adâncime este detectată o muchie de întoarcere.

Aplicaţie. *Să se determine dacă un graf este ciclic şi în caz afirmativ să se identifice un ciclu.*

Vom memora pădurea formată din muchiile de avansare cu ajutorul vectorului `tata` şi în momentul în care este detectată o muchie de întoarcere `(i,j)` vom lista drumul de la `i` la `j` format din muchii de avansare şi apoi muchia `(j,i)`.

Procedura `DF` va fi modificată astfel:

```
procedure DF(i)
   atins(i)←t; vizit(i);
   for toţi j∈Li
      if !atins(i)
      then tata(j) ← i; DF(j)
      else if tata(j)≠i
           then k←j;
                while k≠i
                   write(k,tata(k)); k←tata(k);
                write(i,j); stop
end.
```

- **O aplicaţie: Problema bârfei**

Se consideră `n` persoane. Fiecare dintre ele emite o bârfă care trebuie cunoscută de toate celelalte persoane.

Prin *mesaj* înţelegem o pereche de numere `(i,j)` cu `i,j∈{0,...,n-1}` şi cu semnificaţia că persoana `i` transmite persoanei `j` bârfa sa, dar şi toate bârfele care i-au parvenit până la momentul acestui mesaj.

Se cere una dintre cele mai scurte succesiuni de mesaje prin care toate persoanele află toate bârfele.

Cu enunţul de mai sus, o soluţie este imediată şi constă în succesiunea de mesaje: `(0,1),(1,2),...,(n-2,n-1),(n-1,n-2),(n-2,n-3),...,(1,0)`.

Sunt transmise deci `2n-2` mesaje. După cum vom vedea mai jos, acesta este numărul minim de mesaje prin care toate persoanele află toate bârfele.

Problema se complică dacă există persoane care nu comunică între ele (sunt certate) şi deci nu-şi vor putea transmite una alteia mesaje.

Această situaţie poate fi modelată printr-un graf în care vârfurile corespund persoanelor, iar muchiile leagă persoane care nu sunt certate între ele. Evident, problema are soluţie numai dacă graful este conex, ipoteză în care vom lucra în continuare.

Considerăm în continuare că graful este reprezentat prin listele L_i ale vecinilor vârfurilor `i`. Paşii sunt următorii:

Pasul 1. Printr-o parcurgere DF, este construit arborele DF corespunzător aşa cum s-a prezentat anterior. Reamintim că pentru reprezentarea sa au fost determinate listele F_i ale fii lor fiecărui vârf `i`. Fie `rad` rădăcina sa.

Pasul 2. Printr-o parcurgere în postordine, în care vizitarea unui vârf constă în transmiterea de mesaje de la fiii săi la el, rădăcina va ajunge să cunoască toate bârfele. Realizăm acest lucru prin apelul `post(rad)`, unde:

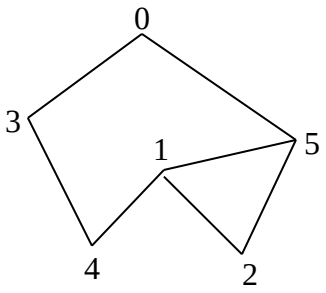
```
procedure post(i)
   for toţi j∈ Fi
      post(j); scrie(j,i)
```

Pasul 3. Printr-o parcurgere în preordine, în care vizitarea unui vârf constă în transmiterea de mesaje de la el la fiii săi, toate vârfurile vor ajunge să cunoască toate bârfele. Realizăm acest lucru prin apelul `post(rad)`, unde:

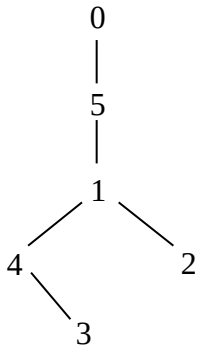
```
procedure pre(i)
   for toţi j∈ Fi
      scrie(i,j) pre(j);
```

Observăm că atât la parcurgerea în postordine, cât şi la cea în preordine au fost listate `n-1` perechi (mesaje), deoarece un arbore cu `n` vârfuri are `n-1` muchii. Rezultă că soluţia de mai sus constă într-o succesiune de `2n-2` mesaje. Mai rămâne de demonstrat că acesta este numărul minim posibil de mesaje care rezolvă problema.

Exemplu. În continuare apar: un graf conex, listele vecinilor, arborele DF şi mutările obţinute conform algoritmului.



0 : 5 3
1 : 4 2 5
2 : 1 5
3 : 4 0
4 : 3 1
5 : 1 2 0



(3,4) (4,1) (2,1) (1,5) (5,0)
(0,5) (5,1) (1,4) (4,3) (1,2)

Propoziţie. Orice soluţie pentru problema bârfei conţine cel puţin `2n-2` mesaje.

Considerăm o soluţie oarecare a problemei.

Punem în evidenţă **primul** moment în care un vârf /persoană cunoaşte toate bârfele. Deci toate celelalte vârfuri au emis o bârfă, deci până acum soluţia cuprinde cel puţin `n`-1 mesaje.

Dar acele `n`-1 vârfuri trebuie să mai primească fiecare cel puţin o bârfă, adică mai sunt necesare cel puţin `n`-1 mesaje.

- **Parcurgerea BF a grafurilor neorientate**

Fie un graf `G=(V,M)` şi fie v_0 un vârf al său. În unele situaţii se pune probleme determinării vârfului `j` cel mai apropiat de v_0 cu o anumită proprietate. Parcurgerea DF nu mai este adecvată.

Parcurgerea pe lăţime BF (*B*readth *F*irst) urmăreşte vizitarea vârfurilor în ordinea crescătoare a distanţelor lor faţă de v_0. Este generalizată parcurgerea pe niveluri a arborilor, ţinându-se cont că graful poate conţine cicluri. Va fi deci folosită o coadă *C*.

Generalizăm noţiunea de nivel de la arbori astfel: nivelul `i` va conţine vârfurile aflate la distanţă `d(i)` de v_0. La fel ca şi la parcurgerea DF, vârfurile vizitate vor fi marcate.

Algoritmul care urmează parcurge pe lăţime componenta conexă a lui i_0, cu determinarea distanţelor de la vârful iniţial v_0 la toate celelalte vârfuri (distanţe măsurate în funcţie de numărul de muchii, adică pentru situaţia când toate muchiile au lungimea 1).

```
for i=0,n-1
   d(i) ← -1
C ⇐ {v0}; d(v0)←0
while C ≠ ∅
   i ⇐ C; vizitează i
   for toţi j vecini ai lui i
      if d(j)=-1
      then d(j)←d(i)+1; j ⇒ C
```

Observaţiile de la parcurgerea în adâncime privitoare la timpul de executare şi la punerea în evidenţă a unui arbore parţial rămân valabile.

Grafuri bipartite

Un graf `G=(V,M)` se numeşte *bipartit* dacă există o partiţie $V = X \cup Y$ astfel încât orice muchie din `M` are o extremitate în `X` şi una în `Y`.

Grafurile bipartite apar de exemplu în probleme de asocieri: (femeie, bărbat) sau (job, persoană).

Ne vom ocupa de următoarea problemă: *fiind dat un graf conex, să se determine dacă el este bipartit; în caz afirmativ să se determine o partiţie adecvată a vârfurilor*.

Un caz particular banal este cel în care graful este un arbore. Un arbore este un graf bipartit cu `X` mulţimea vârfurilor de pe nivelurile pare, iar `Y` mulţimea vârfurilor de pe nivelurile impare.

Revenind la cazul general, pentru orice vârf `v` vom nota prin `d(v)` distanţa sa la vârful iniţial v_0.

Observaţii:

- într-un graf bipartit putem folosi două culori pentru vârfuri: una pentru vârfurile din `X`, iar alta pentru vârfurile din `Y`; în acest mod orice muchie are extremităţile colorate diferit;
- dacă un graf este bipartit, atunci el nu poate conţine cicluri de lungime impară.

Vom folosi parcurgerea BF.

Ca urmare vârfurile vor fi vizitate în ordinea crescătoare a distanţelor lor `d` faţă de rădăcină, deci pentru orice vecin `j` al vârfului curent `i`, `|d(i)-d(j)|` este 0 sau 1.

În timpul parcurgerii BF putem ajunge doar într-una dintre următoarele două situaţii:

1) unul dintre vecinii `j` ai vârfului curent `i` are `d(i)=d(j)`; considerăm primul vârf `i` cu această proprietate. Fie `k` ultimul vârf comun din drumurile de la v_0 la `i` şi `j`; deci `d(i)-`

d(k)=d(j)-d(k). Există atunci ciclul format din drumul de la k la i, muchia (i,j) şi drumul de la j la k. Cum lungimea sa este impară, rezultă că graful nu este bipartit.

2) nu există muchie (i,j) cu d(i)=d(j). Rezultă că orice muchie are extremităţile pe niveluri de paritate diferită. Putem lua atunci drept X mulţimea vârfurilor de pe nivelurile pare, iar drept Y mulţimea vârfurilor de pe nivelurile impare.

```
for i=0,n-1
   d(i) ← -1
C ⇐ {v0}; d(v0)← 0
while C ≠ ∅
   i ⇐ C; vizitează i
   for toţi j vecini ai lui i
      if d(i)=-1
      then d(j) ← d(i)+1; j ⇒ C
      else if d(i)=d(j)
           then scrie("NU"); exit
           else -
scrie(toate vârfurile i cu d(i) par)
scrie(toate vârfurile i cu d(i) impar)
```