Emanuela Cerchez

Arbori parțiali

Adeseori apar în practică probleme a căror rezolvare implică noțiuni şi rezultate din teoria grafurilor. Să considerăm, de exemplu, proiectarea unei rețele de telecomunicații care să conecteze un număr dat de centrale, de amplasare cunoscută, astfel încât între oricare două centrale să existe legătură. Unei astfel de probleme i se poate asocia un graf neorientat în care mulțimea vârfurilor este formată din centralele ce trebuie interconectate, iar mulțimea muchiilor din perechile de centrale între care se poate realiza o legătură directă. Pentru ca între oricare două centrale să existe legătură, ar trebui ca graful să fie conex, dar, ca în orice problemă practică, intervine factorul de cost şi atunci ar fi de dorit să se selecteze un număr cât mai mic posibil de legături directe între centrale, cu alte cuvinte interesează un graf parțial conex minimal al grafului, deci un arbore.

Definiție

Fie G = (X, U) un graf neorientat. Numim *arbore parțial* al grafului G un graf parțial care este arbore.

Teoremă

Condiția necesară şi suficientă ca un graf să conțină un arbore parțial este ca graful să fie conex.

Demonstrație:

Necesitatea Presupunem că G = (X,U) admite un arbore parțial H = (X, U'), U' $\subseteq$ U. H, fiind arbore, este conex şi adăugând la H muchiile din U-U' el rămâne conex. Deci G conex.

Suficiența Presupunem că G este conex. Dacă G este conex minimal, el este arborele parțial căutat. Altfel, există o muchie [x,y]$\in$U astfel încât graful G_1 = (X, U-{[x,y]}) este conex. Dacă G_1 este conex minimal, arborele parțial căutat este G_1, altfel continuăm procedeul de eliminare a muchiilor până când obținem un graf conex minimal, care va fi un arbore parțial a lui G.

Q.E.D.

Observație

Procedeul descris are un număr finit de paşi, deoarece la fiecare pas este eliminată o muchie, iar G are un număr finit de muchii. Mai mult, putem demonstra următoarea proprietate :

Propoziție

Fie G = (X, U) conex, |X| = n, |U| = m. Numărul de muchii ce trebuie înlăturate pentru ca graful să devină un arbore este m-n+1 (*numărul ciclomatic* al grafului).

Demonstrație:

Presupunem că prin eliminarea unui număr oarecare de muchii din G am obținut un graf G' fără cicluri (o pădure). Fiecare din componentele conexe ale lui G' este un arbore.

Notăm :

- p numărul componentelor conexe,
- n_i numărul de vârfuri din componenta conexă i, $i \in \{1,2,...,p\}$
- m_i numărul de muchii din componenta conexă i, $i \in \{1,2,...,p\}$.

Evident că $m_i = n_i - 1$, $\forall i \in \{1,2,...,p\}$.

Numărul de muchi din G' este

$$\sum_{i=1}^{p} (n_i - 1) = n - p$$

Deci au fost eliminate m-n+p muchii. Când G' este arbore, deci conex (p=1), numărul muchiilor eliminate este m-n+1.

Q.E.D.

O primă problemă, cu aplicații, de exemplu, în chimie ar fi generarea tuturor arborilor parțiali ai unui graf dat. Pentru acesta vom folosi metoda backtracking.

Reprezentarea informațiilor

-Vom reprezenta graful G= (X, U) cu ajutorul matricii muchiilor, o matrice G cu două linii şi m coloane (m = |U|), în care pentru fiecare muchie reținem cele două extremități.

-Reprezentăm un arbore parțial ca un vector A cu n-1 componente, în care reținem indicii din G ai muchiilor arborelui. Pentru a nu genera de mai multe ori același arbore, vom conveni ca muchiile să fie memorate în ordinea crescătoare a indicilor lor din G.

- Pentru a verifica dacă o muchie nou selectată nu formează cicluri cu muchiile deja selectate, vom ține evidența componentelor conexe într-un vector c de dimensiune n (n = |X|), c[i] desemnând componenta conexă din care face parte nodul i, pentru $\forall i \in \{1,2,..., n\}$.

Condiții interne

1. $A[i] \in \{1, 2, ..., m\}$, $\forall i \in \{1, 2, ..., n-1\}$
2. $A[i] \leq A[i+1]$, $\forall i \in \{1, 2, ..., n-2\}$

3. c[G[1, A[i]]] ≠ c[G[2, A[i]]], ∀i ∈ {1, 2, ..., n-1}(nu se formează cicluri, extremitățile muchiei fiind în componente conexe diferite).

procedure ConstrArbore (i: byte); [*]

```
//generează toți arborii parțiali care au primele i-1 muchii
//A[1],...,A[i-1]
var j: byte;
begin
if  i = n then //am selectat n-1 muchii care nu formează cicluri
   AfişareArbore
   else
   //selectez o muchie cu indice mai mare decât A[i-1]
  for j := A[i-1]+1 to m do
     if c[G[1, j]] ≠ c[G[2, j]] then
        begin
          A[i] := j; //selectez muchia j
          Unific(c[G[1, j]], c[G[2, j]]);
          //unific componentele conexe ale  extremităților muchiei j
          ConstrArbore(i+1);
          Separ(c[G[1, j]], c[G[2, j]]);
         //restaurez situația dinaintea selectării muchiei j
       end;
end;
```

Inițial, c[i] := i, ∀i ∈ {1, 2, ..., n}şi apelăm ConstrArbore(1).

Arbori parțiali BFS

Breadth-First-Search este tehnica de explorare a grafurilor în lățime.

Dat fiind un graf conex G = (X, U) şi un nod sursă s ∈ X, metoda *BFS* impune vizitarea mai întâi a nodului s, apoi a tuturor nodurilor nevizitate adiacente cu s, apoi a tuturor nodurilor nevizitate adiacente nodurilor adiacente cu s, ş.a.m.d.

[*] Programul *Generare-Arbori-Parțiali* de la sfârşitul capitolului curent generează toți arborii parțiali ai unui graf conex dat.

De exemplu, pentru graful din figura de mai jos, parcurgerea BFS, cu nodul sursă s = 6, este: 6, 4, 5, 8, 9, 2, 3, 7, 10, 11, 1, 12.

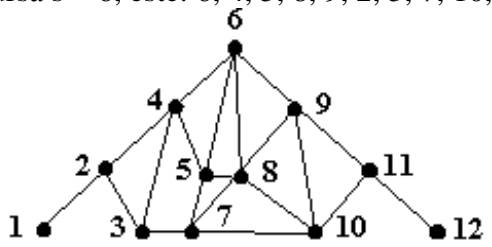

Fig. 13.

Reținând toate muchiile utilizate în timpul parcurgerii obținem arborele parțial BFS, cu rădăcina s = 6 (figura 14): [6,4], [6,5], [6,8], [6,9], [4,2], [4,3], [5,7], [8,10], [9,11], [2,1], [11,12].

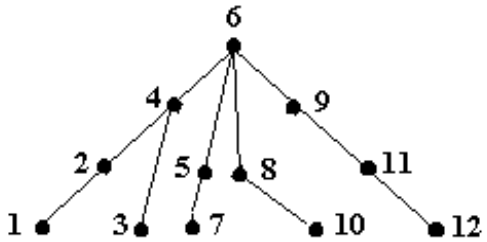

Fig. 14.

Observație

Pentru orice vârf v din arbore, lanțul unic care unește rădăcina s de v reprezintă lanțul cu număr minim de muchii de la s la v în graf.

Reprezentarea informațiilor

1. Reprezentăm graful prin listele de adiacență.

   G: array[Vf] of Lista;

   Deci pentru fiecare vârf din graf reținem lista vârfurilor adiacente cu vârful respectiv.
2. Arborele parțial BFS îl reprezentăm cu ajutorul unui vector T în care pentru fiecare vârf reținem vârful din care a fost atins în timpul parcurgerii BFS.

   T: array[Vf] of Vf;
3. Utilizăm un vector boolean V, în care pentru fiecare vârf din graf reținem dacă a fost sau nu atins în timpul parcurgerii BFS.

   V: array[Vf] of boolean;

4. Pentru parcurgerea grafului în lățime vom utiliza o coadă pe care o inițializăm cu vârful sursă. La fiecare pas extragem un element din

coadă, vizităm toate vârfurile nevizitate adiacente cu vârful extras şi le inserăm în coadă, reținând pentru fiecare vârf vizitat vârful din care a fost atins, pentru reconstituirea arborelui parțial BFS.

Observație

G, T, n (numărul de vârfuri din graf) şi s (vârful sursă) sunt variabile globale.

```
procedure BFS;
//parcurge în lățime graful G, începând cu vârful s construind
//arborele parțial BFS
var  C: Coada; q: Lista; i: Vf;
     V: array[ Vf ] of boolean;
begin
for i := 1 to n do V[i] := false;
C ⇐ s; //inițializez coada cu vârful sursă
while C ≠ [] do
    begin
        x ⇐ C; //extrage un vârf x din coadă
        q := G[x];
        while q ≠ nil do //parcurg lista de adiacență a nodului x
            begin
                i := q^.inf;
                if  not V[i] then // nodul i este nevizitat
                    begin
                        V[i] := true;
                        T[i] := x;//rețin vârful din care a fost atins i
                         C ⇐ i; //introduc vârful i în coadă
                    end;
                q := q^.urm;
            end;
    end;
end;
```

Observații

1. Dacă graful G nu este conex parcurgând BFS fiecare componentă conexă obținem o pădure, formată din arborii parțiali corespunzători fiecărei componente conexe.

2. Complexitatea algoritmului, în cazul în care graful este reprezentat prin listele de adiacență, este *de O(n+m)*, unde n este numărul de vârfuri, iar m numărul de muchii din graf.

Arbori parțiali DFS

O altă tehnică de parcurgere (explorare) a grafurilor este metoda *Depth-First-Search* (parcurgerea în adâncime).

Dat fiind G un graf conex şi un nodul sursă s vizităm mai întâi nodul s, apoi primul nod nevizitat adiacent cu s, mergând în adâncime cât este posibil. Când un nod x nu mai are vecini nevizitați ne întoarcem să cercetăm dacă nodul din care a fost atins x mai are sau nu vecini nevizitați şi continuăm parcurgerea.

De exemplu, pentru graful din figura 13, parcurgerea după metoda DFS, cu nodul inițial s = 6, determină următoarea ordine de vizitarea a nodurilor: 6, 4,2,1,3,7,5,8,9,10,11,12. Marcând muchiile utilizate prin vizitarea nodurilor obținem un arbore parțial numit arbore parțial DFS, cu rădăcina s = 6 (figura 15): [6,4], [4,2], [2,1], [2,3], [3,7], [7,5], [5,8], [8,9], [9,10], [10,11], [11,12].

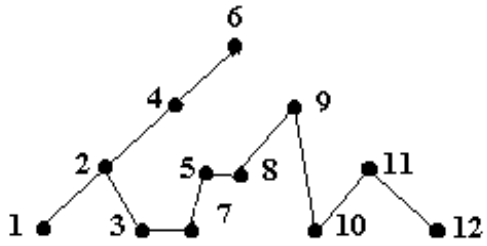

Fig. 15.

Observație

Reprezentarea informațiilor se face în acelaşi mod ca la parcurgerea BFS, în plus vectorul V fiind de asemeni variabilă globală.

Algoritmul poate fi descris folosind o procedură recursivă DFS, pe care inițial o apelăm cu parametrul s, astfel:

```
procedura DFS(x: Vf);
//parcurge vârfurile nevizitate ale grafului începând cu x
begin
V[x] := true;
q := G[x];
while q ≠ nil do //parcurg lista de adiacență a vârfului x
     begin
       i := q^.inf;
       if  not V[i] then //i este nevizitat
             begin
```

```
                T[i] := x; //reţin vârful din care a fost atins i
                DFS(i); //parcurge vârfurile nevizitate  începând cu i
              end;
        q := q^.urm
      end;
  end;
```

Observaţii

1. Dacă graful G nu este conex parcurgând DFS fiecare componentă conexă obţinem o pădure, formată din arborii parţiali corespunzători fiecărei componente conexe.
2. Complexitatea algoritmului, în cazul în care graful este reprezentat prin listele de adiacenţă este *de O(n+m)*, unde n este numărul de vârfuri, iar m numărul de muchii din graf.

Aplicaţie. Descompunerea unui graf în componente biconexe

Definiţie

Fie G = (X, U) un graf neorientat conex. Vârful v ∈ X se numeşte *punct de articulaţie* dacă subgraful obţinut prin eliminarea vârfului v şi a muchiilor incidente cu acesta nu mai este conex.

De exemplu, pentru graful din figura 16 punctele de articulaţie sunt 1,3,5,7.

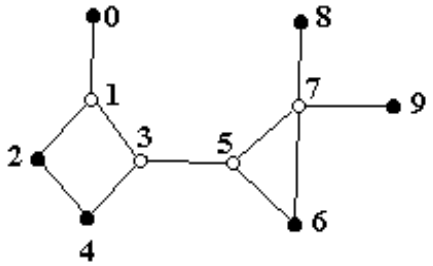

Fig. 16.

Definiţie

Un graf se numeşte *biconex* dacă nu are puncte de articulaţie.

În multe aplicaţii practice, ce se pot modela cu ajutorul grafurilor, nu sunt de dorit punctele de articulaţie. Revenind la exemplul cu reţeaua de telecomunicaţii, dacă o centrală dintr-un punct de articulaţie se defectează rezultatul este nu numai întreruperea comunicării cu centrala respectivă ci şi cu alte centrale.

Definiție

O *componentă biconexă* a unui graf este un subgraf biconex maximal cu această proprietate.

De exemplu, pentru graful din figura 16 componentele biconexe sunt:

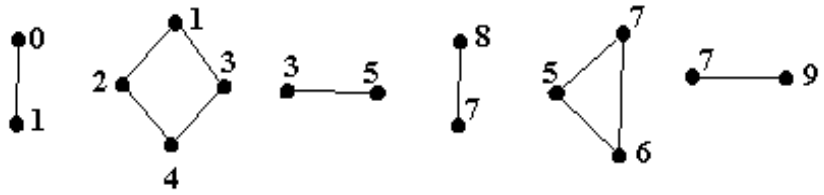

Fig. 17.

Pentru a descompune graful în componente biconexe vom utiliza proprietățile parcurgerii DFS. Parcurgând graful DFS putem clasifica muchiile grafului în:

-muchii care aparțin arborelui parțial DFS (tree edges);

-muchii [u, v] care nu aparțin arborelui şi care unesc vârful u cu un strămoş al său v în arborele parțial DFS numite muchii de întoarcere (back edges). Acestea sunt marcate în exemplu punctat.

De exemplu graful de mai sus poate fi redesenat, clasificând muchiile ținând cont de arborele parțial DFS cu rădăcina 3:

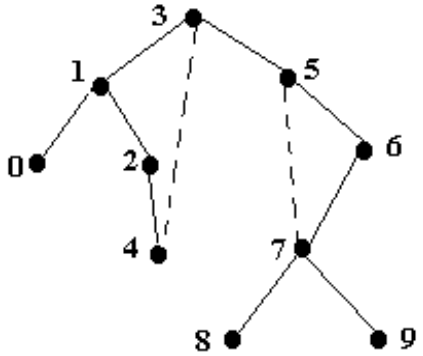

Fig. 18.

Observăm că rădăcina arborelui parțial DFS este punct de articulație dacă şi numai dacă are cel puțin doi descendenți, între vârfuri din subarbori diferiți ai rădăcinii neexistând muchii. Mai mult, un vârf x oarecare nu este punct de articulație dacă şi numai dacă din orice fiu y al lui x poate fi atins un strămoş al lui x pe un lanț format din descendenți ai lui x şi o muchie de întoarcere (un drum "de siguranță" între x şi y).

Pentru fiecare vârf x al grafului putem defini :

dfn(x) = numărul de ordine al vârfului x în parcurgerea DFS a grafului (*depth-first-number*).

De exemplu:

x	0	1	2	3	4	5	6	7	8	9
dfn(x)	2	1	3	0	4	5	6	7	8	9

Dacă x este un strămoş al lui y în arborele parțial DFS atunci

dfn(x) < dfn(y).

De asemeni pentru fiecare vârf x din graf putem defini :

low(x) = numărul de ordine al primului vârf din parcurgerea DFS ce poate fi atins din x pe un alt lanț decât lanțul unic din arborele parțial DFS.

low(x) = min{dfn(x), min{ low(y) | y fiu al lui x }, min{dfn(y) | [x,y] muchie de întoarcere}.

De exemplu:

x	0	1	2	3	4	5	6	7	8	9
dfn(x)	2	1	3	0	4	5	6	7	8	9
low(x)	2	1	1	0	1	5	5	5	8	9

Deci putem caracteriza mai exact punctele de articulație dintr-un graf astfel:

x este punct de articulație dacă şi numai dacă este rădăcina unui arbore parțial DFS cu cel puțin doi descendenți sau, dacă nu este rădăcină, are un fiu y astfel încât low(y) ≥ dfn(x).

Pentru exemplul din figura 16:

- nodul 3 este punct de articulație deoarece este rădăcina arborelui parțial DFS şi are doi descendenți,
- nodul 7 este punct de articulație deoarece low(8)=8 ≥dfn(7)=7,
- nodul 5 este punct de articulație deoarece low(6)=5≥dfn(5)=5,
- nodul 1 este punct de articulație deoarece low(0)=2≥dfn(1)=1.

Modificăm procedura DFS, pentru a calcula pentru fiecare vârf din graf valorile dfn şi low.

Intrare:-graful G, reprezentat prin listele de adiacență;

Ieşire:-vectorii dfn şi low.

Utilizăm o variabilă globală num, pentru a calcula numărul de ordine al vârfului curent în parcurgerea în adâncime.

//Inițializare

num := 0;

```
for x := 1 to n do dfn[x] := -1;
DfnLow(s, -1) // s  este rădăcina arborelui parţial DFS
procedure DfnLow(u, tu: Vf);
//parcurge DFS graful G începând cu vârful u, calculând
// valorile dfn[u] şi low(u); tu este tatăl lui u
var q: Lista; x: Vf;
begin
dfn[u] := num; low[u] := dfn[u]; inc(num);
q := G[u];
while q ≠ nil do //parcurg lista de adiacenţă a lui u
     begin
        x := q^.inf;
        if  dfn[x] = -1 then //x este nevizitat
              begin
                 DfnLow(x, u)
                 low[u] := min(low[u], low[x]);
                 //funcţia min returnează minimul argumentelor ei
              end
              else
              if x ≠ tu then low[u] := min(low[u], dfn[x]);
        q := q^.urm;
     end
end;
```

Vom folosi această idee de calcul recursiv al valorilor dfn şi low pentru descompunerea în componente biconexe a unui graf neorientat conex.
Reprezentarea informațiilor se face în acelaşi mod ca la parcurgerea DFS, dar vectorul V nu mai este necesar, pentru vârfurile nevizitate dfn fiind -1. În plus, vom folosi o stivă S în care vom reține muchiile din graf (atât cele care aparţin arborelui cât şi cele de întoarcere) în ordinea în care sunt întâlnite în timpul parcurgerii. Atunci când identificăm o componentă biconexă, mai exact identificăm un nod u care are un fiu x astfel încât low(x) ≥ dfn(u), eliminăm din stivă şi afişăm toate muchiile din componenta biconexă respectivă.

```
//Iniţializare
num := 0;
S ⇐ (s, -1);
//stiva este iniţializată cu o muchie fictivă [s,-1], s fiind sursa
for x := 1 to n do dfn[x] := -1;
Biconex(s, -1) // s  este rădăcina arborelui parţial DFS
```

```
procedure biconex(u, tu: Vf);
//afişează componentele biconexe , parcurgând graful începând
// cu vârful u; tu este tatăl lui u
var q: Lista; x: Vf;
begin
dfn[u] := num; low[u] := dfn[u]; inc(num);
q := G[u];
while q ≠ nil do // parcurg lista de adiacenţă a lui u
     begin
       x := q^.inf;
       if (tu ≠ x) and (dfn[x] < dfn[u]) then S ⇐ [u, x];
       //dacă tu=x sau dfn(x)>dfn(u) muchia (u,x) a fost deja
       //reţinută în stivă
       if dfn[x] = -1  then //x este nevizitat
              begin
                Biconex(x, u)
                low[u] := min(low[u], low[x]);
                if low[x] ≥ dfn[u] then
                      //am identificat o nouă componentă biconexă
                      Afisare(x, u) // extrage din S şi afişează muchiile
                      //componentei biconexe curente
                      else
                      if x ≠ tu then low[u] := min(low[u], dfn[x]);
             end;
       q := q^.urm;
     end;
end;
```

Observaţie

Oricare două componente biconexe au cel mult un vârf comun, deci nici o muchie nu poate fi în două componente biconexe.

Să urmărim execuţia algoritmului pe următorul exemplu:

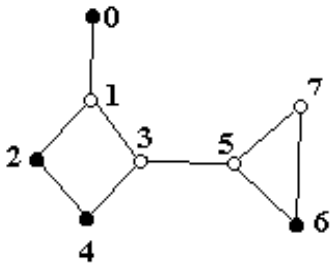

Fig. 19.

Arborele parțial DFS este:

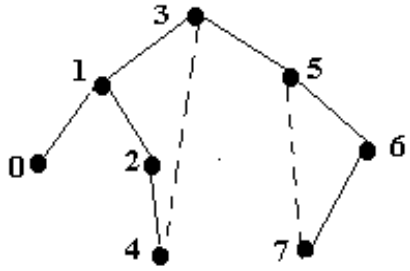

Fig. 20.

Inițial:

x	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
dfn(x)	-1	-1	-1	-1	-1	-1	-1	-1
low(x)								

num	0

S:

3	-1

Apelez biconex(3, -1)

x	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
dfn(x)	-1	-1	-1	0	-1	-1	-1	-1
low(x)				0				

num	1

x ← 1

S:

3	1
3	-1

Apelez biconex(1, 3)

x	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
dfn(x)	-1	1	-1	0	-1	-1	-1	-1
low(x)		1		0				

num	2

x ← 0

S:

1	0
3	1
3	-1

Apelez biconex(0, 1):

x	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
dfn(x)	2	1	-1	0	-1	-1	-1	-1
low(x)	2	1		0				

num	3

x ← 1

Revin în biconex(1, 3)

low[1] ← min(low[1], low[0]) = low[1]

low[0] > dfn[3]. Am obținut o componentă biconexă, afişez **[1,0].**

S:

3	1
3	-1

x ← 2

S:

1	2
3	1
3	-1

Apelez biconex(2, 1)

x	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
dfn(x)	2	1	3	0	-1	-1	-1	-1
low(x)	2	1	3	0				

num	4

x ← 1

x ← 4

S:

2	4
1	2
3	1
3	-1

Apelez biconex(4, 2)

x	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
dfn(x)	2	1	3	0	4	-1	-1	-1
low(x)	2	1	3	0	4			

num	5

x ← 2

x ← 3

S:

4	3
2	4
1	2
3	1
3	-1

low[4] ← min(low[4], dfn[3]) = 0

x	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
dfn(x)	2	1	3	0	4	-1	-1	-1
low(x)	2	1	3	0	0			

Revin în biconex(2, 1)

low[2] ← min(low[2], low[4]) = 0

x	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
dfn(x)	2	1	3	0	4	-1	-1	-1
low(x)	2	1	0	0	0			

Revin în biconex(1, 3)

low[1] ← min(low[1], low[2]) = 0

x	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
dfn(x)	2	1	3	0	4	-1	-1	-1
low(x)	2	0	0	0	0			

low[1] ← min(low[1], dfn[2])

Revin în biconex(3, -1)

low[3] ← min(low[3], low[1]) ⇒ low[1] = dfn[3].
Am obținut o nouă componentă biconexă: **[4,3], [2,4], [1,2], [3,1].**
S:

3	-1

x ← 4
low[3] ← min(low[3], dfn[4]) = 0
x ← 5
S:

3	5
3	-1

Apelez biconex(5, 3)

x	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
dfn(x)	2	1	3	0	4	5	-1	-1
low(x)	1	0	0	0	0	5		

num	6

x ← 3
x ← 6
S:

5	6
3	5
3	-1

Apelez biconex(6, 5)

x	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
dfn(x)	2	1	3	0	4	5	6	-1
low(x)	1	0	0	0	0	5	6	

num	7

x ← 5
x ← 7
S:

6	7
5	6
3	5
3	-1

Apelez biconex(7, 6)

x	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
dfn(x)	2	1	3	0	4	5	6	7
low(x)	1	0	0	0	0	5	6	7

num	8

x ← 5

S:

7	5
6	7
5	6
3	5
3	-1

low[7] ← min(low[7], dfn[5]) = 5

x	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
dfn(x)	2	1	3	0	4	5	6	7
low(x)	1	0	0	0	0	5	6	5

x ← 6

Revin în biconex(6, 5)

low[6] ← min(low[6], low[7]) = 5

x	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
dfn(x)	2	1	3	0	4	5	6	7
low(x)	1	0	0	0	0	5	5	5

low[6] ← min(low[6], dfn[7]) = 5

Revin în biconex (5, 3)

low[5] ← min(low[5], low[6]) = 5

low[6] > dfn[5], deci afişez o nouă componentă biconexă: **[7,5],[6,7],[5,6].**

S:

3	5
3	-1

Revin în biconex(3, -1)

low[3] = min(low[3], low[5]) = 0

low[5] = dfn[3], deci afişez o nouă componentă biconexă: **[5,3].**

S:

3	-1

Stop

Teoremă

Procedura *biconex(s,-1)* generează componentele biconexe ale grafului conex G.

Demonstrație:

Vom lua în considerare cazul în care n, numărul de vârfuri din G, este cel puțin 2 (dacă n = 1, G are o singură componentă biconexă, formată dintr-un singur vârf).

Vom face demonstrația prin inducție după B, numărul de componente biconexe.

P(1) B = 1 ⇔ graful este biconex: rădăcina s a arborelui parțial DFS va avea un singur fiu, să-l notăm x. Apelul *biconex(x, s)* a explorat toate muchiile grafului şi le-a introdus în stiva S. Cum x este singurul vârf pentru care low[x] ≥ dfn[s], muchiile grafului vor fi afişate într-o singură componentă biconexă.

P(B) Să presupunem acum că algoritmul funcționează corect pentru toate grafurile conexe cu cel mult B componente biconexe.

P(B+1) Vom demonstra că algoritmul funcționează pentru toate grafurile conexe cu cel mult B+1 componente biconexe.

Fie G un graf cu B+1 componente biconexe şi fie x primul vârf pentru care este îndeplinită condiția low[x] ≥ dfn[u]. Până în acest moment nu a fost identificată nici o componentă biconexă. În urma apelului *biconex(x, u),* toate muchiile incidente cu descendenți ai lui x se află în stiva S, deasupra muchiei [u, x]. Cum x este primul vârf care îndeplineşte condiția low[x] ≥ dfn[u], deducem că descendenții lui x nu sunt puncte de articulație. Cum u este punct de articulație, rezultă că muchiile situate în S, deasupra muchiei [u, x], inclusiv, formează o componentă biconexă. Muchiile acestei componente biconexe sunt extrase din stivă şi afişate, deci, de la acest pas algoritmul revine la aflarea componentelor biconexe ale unui graf G', obținut din G prin eliminarea unei componente biconexe. Cum G' are B componente biconexe, conform ipotezei inductive, algoritmul determină corect componentele biconexe ale lui G'.

Q.E.D.

Observație

Algoritmul de descompunere în componente biconexe a unui graf reprezentat prin listele de adiacență este liniar (*O(n+m)*), unde n este numărul de vârfuri, iar m numărul de muchii din graf.

Exercițiu

1. Fie G = (X, U) un graf neorientat conex. Muchia [x, y] se numeşte punte dacă prin eliminarea ei din graf, graful parțial obținut nu este conex. Scrieți un algoritm care determină toate punțile unui graf conex. Algoritmul să funcționeze în timp de O(n+m) (n = $|x|$, m = $|U|$).

Observație
O muchie este punte dacă şi numai dacă nu aparține unui ciclu.