10.4.2. Traversarea grafurilor prin tehnica căutării ”prin cuprindere” (”Breadth-First Search”)

- O altă manieră sistematică de traversare a nodurilor unui graf o reprezintă **căutarea prin cuprindere** (“**breadth first search**”).

- **Căutarea prin cuprindere** se bazează pe următoarea tehnică:

  - Pentru fiecare nod vizitat `x`, se caută în imediata sa vecinătate “**cuprinzând**” în vederea vizitării toate nodurile adiacente lui.

- Pentru implementarea acestei tehnici de parcurgere, în locul stivei din metoda de căutare anterioară, pentru reținerea nodurilor vizate se poate utiliza o **structură de date coadă**.

- Schița de principiu a algoritmului de parcurgere apare în secvența [10.4.2.a].

  - Se precizează faptul că s-a utilizat o structură de date **coadă** `(Q)` asupra căreia acționează operatori specifici [Vol.1,&6.5.4.1, Cr00].

```
------------------------------------------------------------

{Căutare prin cuprindere. Schița de principiu. Varianta 1}

procedure CautaPrinCuprindere(x: TipNod; T: TipMultime);
  {se parcurg toate nodurile adiacente lui x prin căutare
prin cuprindere}

  var Q: CoadaDeNoduri;
      x,y: TipNod;
      marc: array[1..maxN] of integer;

  begin
    marc[x] := vizitat;
    Adauga(x,Q);
    while not Vid(Q) do                           [10.4.2.a]
      begin
        x := Cap(Q);
        Scoate(Q);
        for fiecare nod y adiacent lui x do
          if marc[y] = nevizitat then
            begin
              marc[y] := vizitat;
              Adauga(y,Q);
              INSERTIE((x,y),T)
            end
      end
  end; {CautaPrinCuprindere}
------------------------------------------------------------
```

- Algoritmul din secvența [10.4.2.a] inserează arcele parcurse ale grafului într-o mulțime `T` despre care se presupune că este inițial vidă, cu ajutorul operatorului ***INSERTIE***.

* Se presupune de asemenea că tabloul `marc` este inițializat integral cu marca "`nevizitat`".

* Procedura lucrează pentru o singură componentă conexă.

    * Dacă graful **nu** este conex, procedura `CautăPrinCuprindere` trebuie apelată pentru fiecare componentă conexă în parte.

* Se atrage atenția asupra faptului că în cazul căutării prin cuprindere, un nod trebuie marcat cu `vizitat` înaintea întroducerii sale în coadă pentru a se evita plasarea sa de mai multe ori în această structură.

10.4.2.1. Căutarea "prin cuprindere", varianta CLR

* Căutarea prin cuprindere este una dintre cele mai cunoscute metode de căutare, utilizate printre alții de către Djikstra şi Prim în celebrii lor algoritmi [CLR92].

* Ca şi în cazul căutării în adâncime, pentru a ține evidența procesului de căutare nodurile sunt colorate în alb, gri şi negru.

* Toate nodurile sunt colorate inițial alb şi ele devin mai târziu gri, apoi negre.

    * La prima descoperire a unui nod, acesta este colorat în gri.

    * Nodurile gri şi negre sunt noduri deja descoperite în procesul de căutare, dar ele sunt diferențiate pentru a se asigura funcționarea corectă a căutării.

* Dacă arcul `(n,v)∈ A` iar `n` este un nod negru, `v` este colorat fie în gri fie în negru.

    * În ultimul caz toate nodurile adiacente lui `v` au fost deja vizitate.

    * Nodurile gri mai pot avea ca adiacenți şi noduri albe, ele marchează de fapt frontiera dintre nodurile vizitate şi cele nevizitate.

* Şi în cazul căutării prin cuprindere se poate construi un **subgraf** `sp` al predecesorilor nodurilor vizitate.

    * Ori de câte ori un nod `v` este descoperit pentru prima oară în procesul de căutare la parcurgerea listei de adiacențe a nodului `u`, acest lucru se marchează prin `sp[v]=u`.

* După cum s-a mai precizat **subgraful predecesorilor** este de fapt un **arbore liber**, reprezentat printr-un tablou liniar cu în baza relației "indicator spre părinte".

* Procedura `TraversarePrinCurpindere` din secvența [10.4.2.1.a] presupune graful `G=(N,A)` reprezentat prin **structuri de adiacențe**.

    * Culoarea curentă a fiecărui nod `u ∈ N` este memorată în tabloul `culoare[u]`

    * Predecesorul nodului `u` adică nodul în lista căruia a fost descoperit, este înregistrat în tabloul `sp[u]`.

   * Dacă u **nu** are predecesor (adică este nodul de pornire) se marchează acest lucru prin sp[u]=**nil**.

* În cadrul procesului de traversare a grafului, se calculează şi distanţa de la nodul de start la fiecare din nodurile grafului.

   * Distanţa de la sursă la nodul curent u calculată de către algoritmi este memorată în d[u].

* Algoritmul de traversare utilizează o coadă FIFO notată cu Q pentru a gestiona nodurile implicate în procesul de căutare, scop în care face uz de operatorii consacraţi pentru această structură de date.

```
------------------------------------------------------------

{Căutarea prin cuprindere. Schiţa de principiu. Varianta 2.
(Cormen, Leiserson, Rivest)}

  CautaPrin Cuprindere(G:TipGraf,s:TipNod);
[1]   for fiecare nod u∈N(G)-{s} do  {s este nodul de start}
        begin
[2]       culoare[u]<-alb;
[3]       d[u]<-∞;
[4]       sp[u]<-nil
        end
[5]   culoare[s]<-gri;
[6]   d[s]<-0;
[7]   sp[s]<-nil;
[8]   Initializeaza(Q); Adauga(s,Q);
[9]   while Q<>Φ do
        begin                                        [10.4.2.1.a]
[10]      u<-Cap(Q);
[11]      for fiecare v∈Adj[u] do
[12]        if culoare[v]=alb then
              begin
[13]            culoare[v]<-gri;
[14]            d[v[<-d[u]+1;
[15]            sp[v]<-u;
[16]            Adauga(v,Q)
              END
[17]      Scoate(Q)
[18]      culoare[u]<-negru
        end
------------------------------------------------------------
```

* **Funcţionarea algoritmului.**

* Liniile 1- 4 iniţializează structurile de date, adică:

   * Toate nodurile u cu excepţia nodului de start sunt marcate cu alb în tabloul culoare

   * Distanţele corespunzătoare tuturor nodurilor sunt setate pe ∞ (d[u]=∞)

- Părintele fiecărui nod este inițializat cu `nil (sp[u]=nil)`.

- Linia 5 marchează nodul `s` furnizat ca parametru al procedurii de căutare cu gri, el fiind considerat sursa (originea) procesului de căutare.

- Liniile 6 şi 7 inițializează `d[s]` pe zero şi `sp[s]` cu **`nil.`**

- Linia 8 inițializează coada `Q` şi îl adaugă pe `s` în coadă.

  - De altfel coada `Q` va conține numai noduri colorate în gri.

- Bucla principală a programului apare între liniile 9-18 şi ea iterează atâta vreme cât există noduri (gri) în coadă.

  - Nodurile gri din coadă sunt noduri descoperite în procesul de căutare în liste de adiacențe care nu au fost încă epuizate.

- Linia 10 furnizează nodul gri `u` aflat în capul cozii `Q`.

- Bucla **`for`** (liniile 11-16) parcurge fiecare nod `v` al listei de adiacențe a lui `u`.

  - Dacă `v` este alb, el nu a fost încă descoperit, ca atare algoritmul îl descoperă executând liniile 13-16 adică:

    - Nodul `v` este colorat în gri

    - Distanța `d[v]` este setată pe `d[u]+1` indicînd creşterea acesteia cu o unitate în raport cu părintele nodului

    - `u` este marcat ca şi părinte al lui `v`

    - În final `v` este adăugat în coada `Q` la sfârşitul acesteia.

- După ce toate nodurile listei de adiacențe a lui `u` au fost examinate, `u` este extras din coada `Q` şi colorat în negru (liniile 17-18).

- **Analiza performanței**.

- Se analizează timpul de execuție al algoritmului pentru un graf `G=(N,A)`.

- După inițializare **nici un nod** nu mai este ulterior colorat în alb, ca atare testul din linia 12 asigură faptul că fiecare nod este adăugat cozii cel mult odată şi este scos din coadă **cel mult odată**.

- Operațiile ***`Adauga`*** şi ***`Scoate`*** din coadă consumă un timp O*(1)*, ca atare timpul total dedicat operării cozii `Q` este O*(N)*.

- Deoarece lista fiecărui nod este scanată exact înaintea scoaterii nodului din coadă această operație se realizează cel mult odată pentru fiecare nod.

- Deoarece suma lungimilor tuturor listelor de adiacență este O(*A*), timpul total necesar pentru scanarea listelor de adiacențe este cel mult O(*A*).

- Regia inițializării este O(*N*) deci timpul total de execuție al procedurii `CautăPrinCurpindere` este O(*N*+*A*).

- În concluzie, căutarea prin cuprindere necesită un timp de execuție liniar cu mărimea listelor de adiacențe ale reprezentării grafului `G`.

10.4.2.2. Căutare ”prin cuprindere” în grafuri reprezentate prin structuri de adiacențe

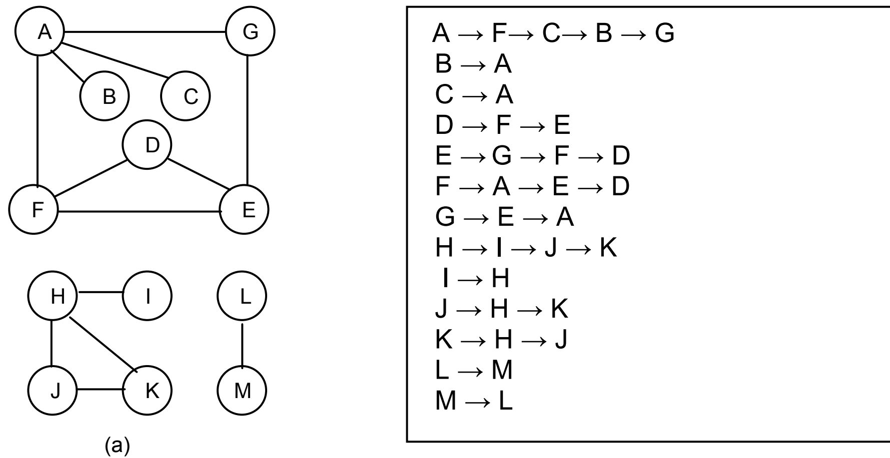
- În secvența [10.4.2.2.a] apare un exemplu de procedură care parcurge un graf în baza tehnicii de parcurgere prin cuprindere, utilizând o structură de date coadă.

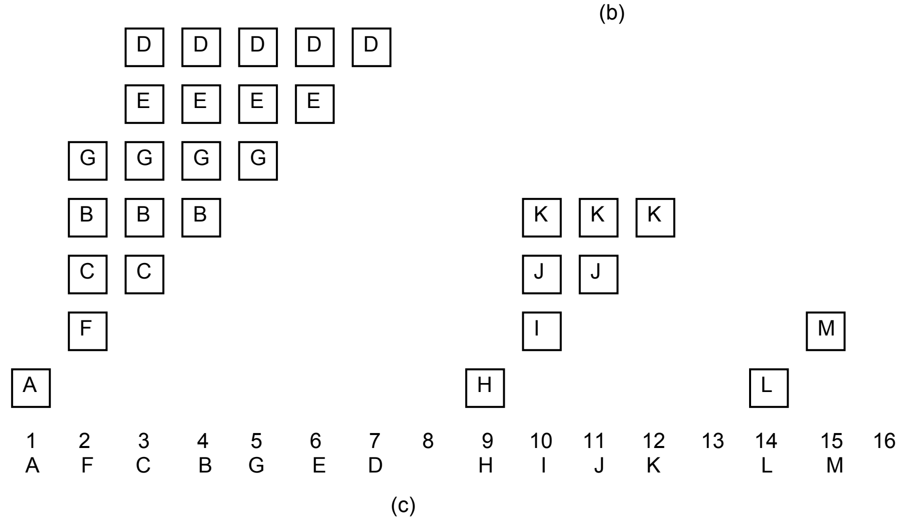
- Graful de consideră reprezentat cu ajutorul **structurilor de adiacențe** implementate cu liste înlănțuite simple.

```
{Traversarea prin cuprindere a grafurilor reprezentate prin
SA implementate cu ajutorul listelor înlănţuite simple}

procedure ParcurgerePrinCuprindere;
  var id,x: integer;
      mark: array[1..maxN] of integer;

  procedure CautaPrinCuprindere(x: integer);
    var t: RefTipNod;
    begin
      Adauga(x,Q);
      repeat
        x:= Cap(Q); Scoate(Q);
        id:= id + 1; marc[x]:= id;
        write(x);
        t:= Stradj[x];
        while t <> nil do
          begin
            if marc[t^.nume]=0 then              [10.4.2.2.a]
              begin
                Adauga(t^.nume,Q);
                marc[t^.nume]:=-1
              end
            t:= t^.urm
          end
      until Vid(Q)
    end; {CautaPrinCuprindere}

  begin {ParcurgerePrinCuprindere}
    id:= 0; Initializeaza(Q);
    for x:= 1 to N do marc[x]:= 0;
    for x:= 1 to N do
      if marc[x]=0 then
         begin
           CautaPrinCuprindere(x);
```

```
          writeln
        end
  end; {ParcurgerePrinCuprindere}
```

- În figura 10.4.4.2.a se prezintă un exemplu de traversare prin cuprindere a unui graf reprezentat prin structuri de adiacențe.





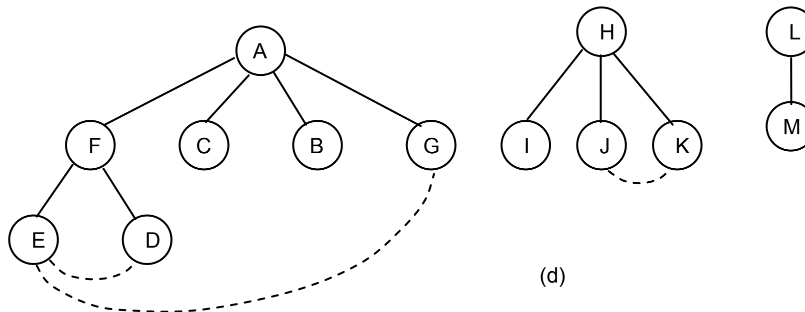


Fig.10.4.2.2.a. Traversarea prin cuprindere a unui graf

- Se consideră graful din figura 10.4.2.2.a (a) reprezentat prin structura de adiacențe din aceeaşi figură (b),
- Ordinea de parcurgere a arcelor va fi următoarea: `AF,AC,AB,AG,FA,FE,FD,CA,BA,GE,GA,DF,DE,EG,EF,ED,HI,HJ,HK,IH,JH,JK,KH,KJ,IM,MI`.

* Evoluția conținutului cozii pe parcursul traversării şi ordinea în care sunt traversate nodurile apar în fig. 10.4.2.2.a (c).

* În mod similar cu traversarea bazată pe căutarea în adâncime, se poate construi o pădure de arbori de acoperire ("**spanning trees**") specifică căutării prin cuprindere.

* În acest caz, un arc (x,y) se consideră **ramură** a arborelui de căutare, dacă în bucla **`for`** a secvenței [10.4.2.a], respectiv **`while`** a secvenței [10.4.2.2.a] nodul `y`, respectiv t^.urm este vizitat întâia dată venind dinspre nodul `x`.

* În cazul căutării prin cuprindere în grafuri neorientate, fiecare arc care **nu** este o ramură a arborelui de cuprindere, este un **arc de trecere** care conectează două noduri dintre care niciunul **nu** este strămoşul celuilalt.

* Arborii de căutare aferenți parcurgerii grafului (a) din figura fig. 10.4.2.2.a apar în aceeaşi figură (d).

    * După cum s-a precizat, aceşti arbori cuprind acele arce care conduc întâia dată la un anumit nod.

* Utilizând arbori de căutare prin cuprindere, verificarea existenței **ciclurilor** în cadrul unui graf, poate fi realizată în O*(n)* unități de timp, indiferent de numărul de arce.

    * După cum s-a precizat în §10.1, un graf cu `n` noduri şi `n` sau mai multe arce, trebuie să aibă cel puțin un ciclu.

* Cu toate acestea şi un graf cu `n` noduri şi `n-1` sau mai puțin arce poate avea cicluri, dacă conține două sau mai multe componente conexe.

* O modalitate sigură de a determina ciclurile unui graf este aceea de a-i construi pădurea arborilor de căutare prin cuprindere.

    * Fiecare arc de trecere, reprezentat punctat în figură, închide un ciclu simplu care cuprinde arcele arborelui care conectează cele două noduri, prin cel mai apropiat strămoş comun al lor.

10.4.2.3. Analiza căutării ”prin cuprindere”

* Din punctul de vedere al timpului de execuție, complexitatea algoritmului de traversare (căutare) prin cuprindere este aceeaşi ca şi la căutarea în adâncime.

* Fiecare nod al grafului este plasat în coadă o singură dată, astfel corpul buclei **`while`** (secvența [10.4.2.2.a]) se execută o singură dată pentru fiecare nod.

* Fiecare arc `(x,y)` este examinat de două ori, odată pentru `x` şi odată pentru `y`.

* Astfel, dacă graful are `n` noduri şi `a` arce, timpul de execuție al algoritmului de căutare prin cuprindere este O*(max (n,a))* dacă utilizăm reprezentarea prin structuri de adiacență.

    * Deoarece în general `a≥n`, de regulă se va considera timpul de execuție al căutării prin cuprindere O*(a)*, ca şi în cazul căutării în adâncime.

10.4.3. Comparație între tehnicile fundamentale de traversare a grafurilor

- Traversarea unui graf, indiferent de metoda utilizată, are principial un caracter unitar, particularizarea rezultând din structura de date utilizată în implementare. Astfel, în ambele tehnici de traversare, nodurile pot fi divizate în trei clase:
  - (1) Clasa "**arbore**"– care cuprinde nodurile care au fost extrase din structura de date utilizată în traversare, adică nodurile deja vizitate colorate în negru;
  - (2) Clasa "**vecinătate**" – care cuprinde nodurile adiacente nodurilor traversate sau în curs de traversare.
    - Aceste noduri au fost luate în considerare dar nu au fost încă vizitate şi se găsesc introduse în structura de date utilizată în traversare.
    - Sunt nodurile colorate în gri;
  - (3) Clasa "**neîntâlnite**" - care cuprinde nodurile la care nu s-a ajuns până la momentul considerat, adică cele colorate în alb.
- Un **arbore de căutare** ia naştere conectând fiecare nod din clasa "**arbore**" cu nodul care a cauzat introducerea sa în structura de date utilizată în parcurgere.
- Pentru a parcurge în mod sistematic o componentă conexă a unui graf (deci pentru a implementa o procedură "parcurge") se introduce un nod oarecare al componentei în clasa "vecinătate" şi toate celelalte noduri în clasa "neîntâlnite".
- În continuare, până la vizitarea tuturor nodurilor se aplică următorul procedeu:
  - Se mută un nod – fie acesta `x` – din clasa "vecinătate" în clasa "arbore" ş
  - Se trec în clasa "vecinătate" toate nodurile din clasa "neântâlnite" care sunt adiacente lui `x`.
- Metodele de parcurgere a grafurilor se diferențiază după maniera în care sunt alese nodurile care se trec din clasa "vecinătate" în clasa "arbore".
  - (1) La parcurgerea "**în adâncime**" se alege din vecinătate nodul cel mai recent întâlnit (ultimul întâlnit) ceea ce corespunde cu utilizarea unei stive pentru păstrarea nodurilor din "vecinătate".
  - (2) La parcurgerea "**prin cuprindere**" se alege nodul cel mai devreme întâlnit (primul întâlnit) ceea ce presupune păstrarea într-o coadă a nodurilor din clasa "vecinătate".
- Se pot utiliza în acest scop şi alte structuri de date, spre exemplu cozi bazate pe prioritate [Cr 87] în cazul grafurilor ponderate.
- Contrastul dintre cele doză metode de parcurgere este şi mai evident în cazul grafurilor de mari dimensiuni.

- Căutarea "în adâncime" se avântă în profunzime de-a lungul arcelor grafului, memorând într-o stivă punctele de ramificaţie.

- Căutarea "prin cuprindere" mătură prin extindere radială graful, memorând într-o coadă frontiera locurilor deja vizitate.

- La căutarea "în adâncime" graful este explorat căutând noi noduri, cât mai departe de punctul de plecare şi luând în considerare noduri mai apropiate numai în situaţia în care nu se poate înainta mai departe

- Căutarea prin cuprindere acoperă complet zona din jurul punctului de plecare mergând mai departe numai când tot ceea ce este în imediatat sa apropiere a fost parcurs.

- Este însă evident faptul că în ambele situaţii, ordinea efectivă de parcurgere a nodurilor depinde:

  - (1) Pe de o parte de structura de date utilizată pentru implementarea grafului

  - (2) Pe altă parte de ordinea în care sunt introduse iniţial nodurile în această structură.

- În afara diferenţelor rezultate din manierele de operare ale celor două metode, se remarcă diferenţe fundamentale în ceea ce priveşte **implementarea**.

  - Căutarea "în adâncime" poate fi foarte simplu exprimată în manieră recursivă ea bazându-se pe o structură de date stivă

  - Căutarea "prin cuprindere" admite o implementare foarte simplă nerecursivă fiind bazată pe o structură de date coadă.

- Acesta este încă un exemplu care evidenţiază cu pregnanţă legătura strânsă care există ăntre o anume structură de date şi algoritmul care o prelucrează.

10.5. Aplicaţii ale traversării grafurilor

- În cadrul paragrafului de faţă se prezintă câteva dintre aplicaţiile tehnicilor de traversare a grafurilor.

- Se au în vedere în acest context:

  - Unele aspecte legate de conexitate

  - Punctele de articulaţie ale unui graf

  - Componentele biconexe ale unui graf.

10.5.2. Grafuri şi conexiuni

* În cadrul grafurilor, noțiunea de **conexiune** joacă un rol central şi ea este strâns legată de noțiunea de **arc**, respectiv de noțiunea de **drum**.

* Astfel în paragraful §10.1. se definesc pornind de la aceste elemente noțiunile de **graf conex**, respectiv de **componentă conexă** a unui graf.

* Se reaminteşte că un **graf conex** este acela în care pentru fiecare nod al său există un drum spre oricare alt nod al grafului.

* Un graf care **nu** este conex este format din **componente conexe**.

* Deoarece în anumite situații prelucrarea grafurilor este simplificată dacă grafurile sunt partajate în componentele lor conexe, în continuare se vor prezenta unele tehnici de determinare a acestor componente.

* De asemenea sunt prezentate noțiunile de graf biconex şi punct de articulație precum şi tehnicile determinării acestora.

10.5.2.1. Determinarea componenetelor conexe ale unui graf

* Oricare dintre **metodele de traversare** a grafurilor prezentate în paragraful anterior poate fi utilizată pentru determinarea componentelor conexe ale unui graf.

* Acest lucru este posibil deoarece toate metodele de traversare se bazează pe aceeaşi **strategie generală** a vizitării tuturor nodurilor dintr-o componentă conexă înainte de a trece la o altă componentă.

* O manieră simplă de a vizualiza componentele conexe este aceea de a modifica procedura recursivă de traversare prin căutare în adâncime spre exemplu aşa cum se sugerează în procedura `ComponenteConexe` secvența [10.5.2.1.a].

```
------------------------------------------------------------

{Determinarea componentelor conexe ale unui graf reprezentat
prin SA implementate cu ajutorul listelor înlănţuite simple}

procedure Componente Conexe;
  var id,x: integer;
      marc: array[1.maxN] of integer;

  procedure Componenta(x: integer);
    var t: RefTipNod;
    begin
      id:= id+1; marc[x]:= id;
      write(t^.nume);
      t:= StrAdj[x];
      while t<>nil do
        begin
          if marc[t^.nume]=0 then
             Componenta(t^.nume);
          t:= t^.urm
        end
    end; {Componenta}                          [10.5.2.1.a]
```

```
begin
  id:= 0;
  for x:= 1 to N do marc[x]:= 0;
  for x:= 1 to N do
    if marc[x]=0 then
      begin
        Componenta(x);
        writeln
      end
end; {ComponenteConexe}
```

- Alte variante ale procedurii de căutare în adâncime cum ar fi cea aplicată grafurilor reprezentate prin matrice de adiacenţe sau cea nerecursivă, precum şi căutarea prin cuprindere, modificate în aceeaşi manieră, vor evidenţia aceleaşi componente conexe, dar nodurile vor fi vizualizate în altă ordine.

- Funcţie de natura prelucrărilor ulterioare ale grafului pot fi utilizate şi alte metode.

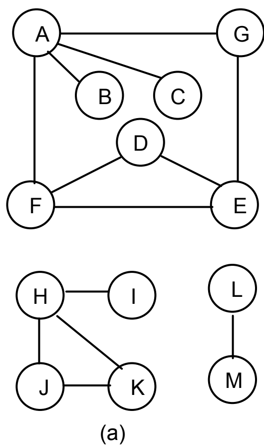
- Astfel spre **exemplu**, se poate introduce tabloul `invmarc` ("inversul" tabloului `marc`) care se completează ori de câte ori se completează tabloul `marc`, respectiv când `marc[x]:= id` se asignează şi `invmarc[id]:= x`.

  - În tabloul `invmarc` intrarea `id` conţine indexul celui de-al `id`-lea nod vizitat. În consecinţă, nodurile aparţinând aceleeaşi componente conexe sunt contigue adică ocupă poziţii alăturate.

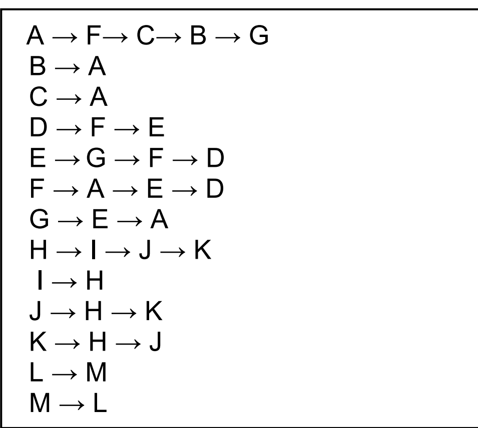
  - În acest tablou indexul care precizează o nouă componentă conexă, este dat de valoarea lui `id` din momentul în care procedura `Componenta` este apelată din procedura `ComponenteConexe`.

  - Aceste valori pot fi memorate separat sau pot fi marcate în tabloul `invmarc`, spre exemplu primind valori negative (opuse).

- În fig.10.5.2.1.a (c) se prezintă valorile pe care le conţin aceste tablouri, în urma execuţiei procedurii `ComponenteConexe` asupra grafului (a), reprezentat prin structura de adiacenţe din aceeaşi figură (b).



(a)



(b)

x	1	2	3	4	5	6	7	8	9	10	11	12	13
nume[x]	A	B	C	D	E	F	G	H	I	J	K	L	M
marc[x]	1	7	6	5	3	2	4	8	9	10	11	12	13
invmarc[x]	-1	6	5	7	4	3	2	-8	9	10	11	-12	13

(c)

Fig. 10.5.2.1.a. Evidențierea componentelor conexe ale unui graf

- În activitatea practică, este deosebit de avantajoasă utilizarea unor astfel de tehnici de partajare a grafurilor în componente conexe în vederea prelucrării ulterioare a acestor componente în cadrul unor algoritmi complecşi.

    - Astfel, algoritmii de complexitate mai ridicată sunt eliberaţi de detaliile prelucrării unor grafuri care nu sunt conexe şi în consecinţă devin mai simpli.

10.5.2.2. Puncte de articulaţie şi componente biconexe

- În anumite situaţii este util a se prevedea mai mult decât un drum între nodurile unui graf cu scopul de a rezolva posibile căderi ale unor puncte de contact (noduri).

    - Astfel, spre exemplu în reţeaua feroviară există mai multe posibilităţi de a ajunge într-un anumit loc.

    - De asemenea într-un circuit integrat liniile principale de comunicaţie sunt adesea dublate astfel încât circuitul rămâne încă în funcţiune dacă vreo componentă cade.

- Un **punct de articulaţie** al unui graf conex este un nod, care dacă este suprimat, graful se rupe în două sau mai multe bucăţi.

- Un graf care nu conţine puncte de articulaţie se numeşte **graf biconex**.

    - Într-un **graf biconex**, fiecare pereche de noduri este conectată prin cel puţin două drumuri distincte.

- Un graf care **nu** este biconex se divide în **componente biconexe**, acestea fiind mulțimi de noduri mutual accesibile via două drumuri distincte.

- În fig. 10.5.2.2.a apare un graf conex care însă nu este biconex.

- **Punctele de articulație** ale acestui graf sunt:

    - `A` care leagă pe `B` de restul grafului
    - `H` care leagă pe `I` de restul grafului,
    - `L` care leagă pe `M` de restul grafului
    - `G` prin a cărui suprimare graful se divide în trei părți.

- În concluzie în cadrul grafului din figură există şase **componente biconexe**:

    - (1) Grupul de noduri `{A,C,G,D,E,F}`
    - (2) `{G,J,H,K}`
    - (3) Nodul individual `B`
    - (4) Nodul individual `I`
    - (5) Nodul individual `L`
    - (6) Nodul individual `M`.

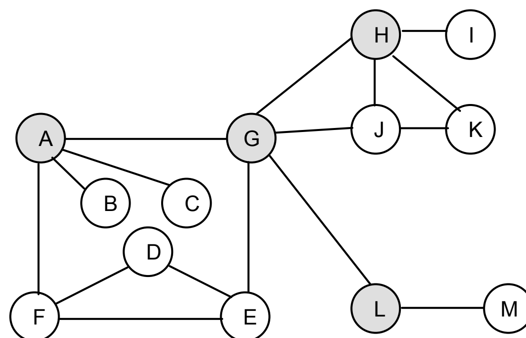


Fig.10.5.2.2.a. Graf care nu este biconex

- În acest context una din problemele care se ridică este aceea a **determinării** punctelor de articulație ale unui graf.

    - Se face precizarea că aceasta este una din mulțimea de probleme deosebit de importante referitoare la conexitatea grafurilor.

* Astfel, ca şi un exemplu de aplicație al conexității grafurilor poate fi prezentată o rețea care este de fapt un graf în care nodurile comunică unele cu altele.

    * În legatură cu această rețea se ridică următoarea întrebare fundamentală: “Care este capacitatea de supraviețuire a rețelei atunci când unele dintre nodurile sale cad?”

    * Cu alte cuvinte în ce condiții, în urma căderii unor noduri rețeaua rămâne încă funcțională?

* Un graf are **conexitatea k** sau altfel spus are **numărul de conexitate egal cu k** dacă prin suprimarea a oricare `k-1` noduri ale sale graful rămâne conex [FK69].

    * Spre exemplu, un graf are conexitatea doi, dacă şi numai dacă **nu are puncte de articulație**, cu alte cuvinte, dacă şi numai dacă este **biconex**.

* Cu cât numărul de conexitate al unui graf este mai mare, cu atât capacitatea de supraviețuire a grafului la căderea unora din nodurile sale este mai mare.

10.5.2.3. Determinarea punctelor de articulație ale unui graf

* În vederea determinării punctelor de articulație ale unui graf conex, poate fi utilizată, printr-o extensie simplă, traversarea grafurilor prin tehnica căutării în adâncime.

    * Absența punctelor de articulație precizează un graf biconex.

* Se consideră spre **exemplu**, graful din figura 10.5.2.2.a şi un arbore de căutare în adâncime asociat, ca şi cel din fig.10.5.2.3.a.

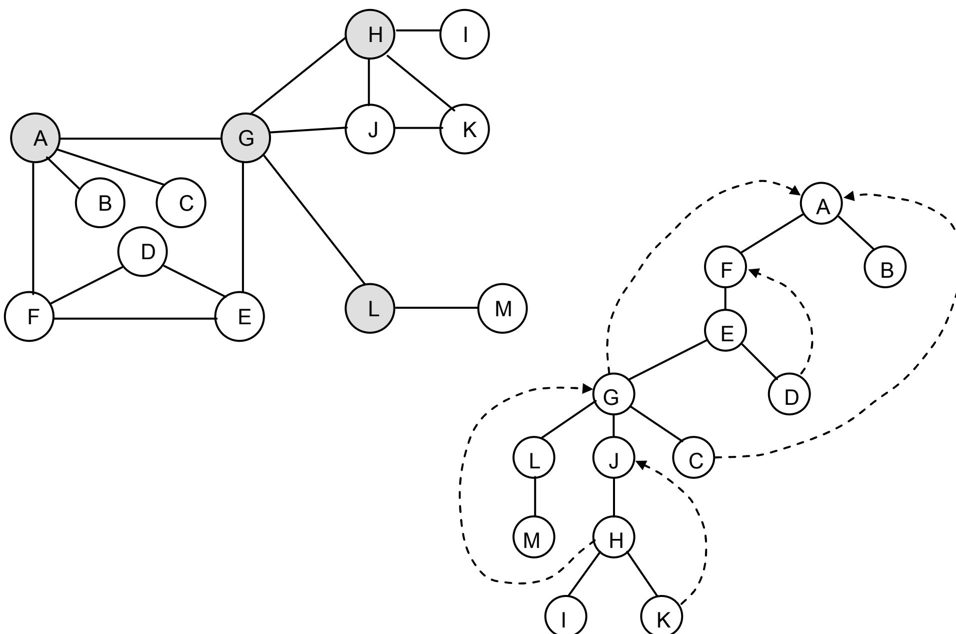


Fig.10.5.2.3.a. Arbore de căutare în adâncime pentru determinarea punctelor de articulație ale unui graf conex

- Se observă că suprimarea nodului `E` **nu** conduce la dezmembrarea grafului deoarece ambii fii ai acestuia, `G` respectiv `D` sunt conectați prin arce de retur (linii punctate) cu noduri situate deasupra în arbore.

- Pe de altă parte suprimarea lui `G` conduce la scindarea grafului deoarece nu există astfel de alternative pentru nodurile `L` sau `J`.

- Un nod oarecare `x` al unui graf **nu** este un punct de articulație dacă fiecare fiu `y` al său are printre descendenți vreun nod conectat (printr-o linie punctată) cu un nod situat în arbore deasupra lui `x`, cu alte cuvinte, dacă există o conexiune alternativă de la `x` la `y`.

  - Această verificare **nu** este valabilă pentru rădăcina arborelui de căutare în adâncime deoarece nu există noduri situate "deasupra" acesteia.

  - **Rădăcina** este un punct de articulație dacă are doi sau mai mulți fii deoarece singurul drum care conectează fii rădăcinii trece prin rădăcina însăşi.

- Determinarea punctelor de articulație poate fi implementată pornind de la căutarea în adâncime, prin transformarea procedurii de căutare într-o funcție care returnează pentru nodul furnizat ca parametru, cel mai înalt punct din cadrul arborelui de căutare întâlnit în timpul căutării, adică nodul cu cea mai mică valoare memorată în tabloul `marc`.

- Algoritmul implementat în forma funcției `Articulație` apare în secvența [10.5.2.3.a]

```
{Determinarea punctelor de articulație ale unui graf
reperezentat prin SA}

function Articulatie(x: integer): integer;
  var t: RefTipNod;
      m,min: integer;

begin
    id:= id+1; marc[x]:= id; min:= id;
    t:= StrAdj[x];
    while t <> z do
      begin
        if marc[t^.nume]=0 then                    [10.5.2.3.a]
          begin
            m:= Articulatie(t^.nume);
            if m<min then min:= m;
            if m >=marc[x] then writeln(x) {punct de
          end                                 articulație}
        else
          if marc[t^.nume]<min then min:= marc[t^.nume];
        t:= t^.urm
      end
    Articulatie:= min
  end; {Articulatie}
```

- Referitor la această funcție se fac următoarele precizări:

1. La căutarea în adâncime într-un graf, valoarea lui `marc[x]` pentru orice nod `x` al grafului precizează numărul de ordine al nodului `x` în cadrul traversării.
    - Aceeaşi ordine rezultă şi din traversarea în preordine a nodurilor arborelui de căutare în adâncime.

2. Pentru fiecare nod `x`, valoarea `min` returnată de funcția `Articulație` este cel mai mic număr de ordine întâlnit în cadrul traversării.

    - Acest număr poate corespunde chiar lui `x` sau oricărui nod `z` la care se poate ajunge pornind de la `x`, coborând zero sau mai multe arce ale arborelui până la un descendent `w` (`w` poate fi chiar `x`), iar apoi urmând un arc de retur `(w,z)`.

    - Valoarea `min` se calculează pentru toate nodurile `x`, traversând arborele în postordine.

        - Astfel, când se procesează nodul `x` valoarea `min` a fost deja calculată pentru toți fiii `y` ai lui x.

3. Pentru un nod oarecare `x`, valoarea `min` asociată este cea mai mică valoare dintre:

    - Numărul de ordine al lui `x;`

    - Numărul de ordine al oricărui nod `z` pentru care există în arborele de căutare în adâncime un arc de retur `(x,z);`

    - Valoarea lui `min` pentru toți fii `y` ai lui `x`.

4. Punctele de articulație se găsesc astfel;

    - Rădăcina este punct de articulație dacă şi numai dacă are doi sau mai mulți fii.

    - Un nod `x`, altul decât rădăcina este un punct de articulație dacă şi numai dacă există vreun fiu `y` al lui `x` astfel încât valoarea `min` pentru `y` este mai mare sau egală cu numărul de ordine al lui `x.`

        - În acest caz, `x` deconectează pe `y` şi descendenții acestuia de restul grafului.

5. Dacă valoarea `min` pentru toți fii `y` ai lui `x` este mai mică decât numărul de ordine al lui `x`, atunci există cu siguranță un drum pe care se poate ajunge de la oricare din fiii `y` ai lui `x` înapoi la un strămoş propriu `z` a lui `x` (nodul care are numărul de ordine egal cu `min` pentru `y`) şi în consecință suprimarea nodului `x` nu conduce la deconectarea nici unui fiu `y` al lui `x` sau a descendenților săi de restul grafului [AH85].

- Funcția `Articulație` determină de fapt în manieră recursivă cel mai înalt punct al arborelui care poate fi atins via un arc de retur pentru orice descendent al unui nod `x` furnizat ca parametru şi utilizează această informație pentru a stabili dacă `x` este un punct de articulație.

- După cum s-a precizat, aceasta presupune o simplă comparaţie între valoarea `m`, adică minimul determinat pentru nodul curent şi `marc[x]`, adică numărul de ordine al lui `x`.

- În plus, mai este necesar a se verifica dacă `x` nu este cumva rădăcina arborelui de căutare în adâncime cu alte cuvinte, dacă `x` nu este cumva primul nod parcurs în apelul funcţiei `Articulaţie` pentru componenta conexă a grafului care conţine nodul `x`.

- Această verificare se face în afara funcţiei recursive, motiv pentru care ea nu este prezentă în secvenţa [10.5.2.3.a].

- Această secvenţă care de fapt afişează punctele de articulaţie, poate fi uşor extinsă pentru a realiza şi alte prelucrări asupra punctelor de articulaţie sau a componentelor biconexe.

- Deoarece procedeul derivă din traversarea grafurilor prin tehnica căutării în adâncime, efortul său de execuţie este proporţional cu $O(n+a)$ în reprezentarea grafurilor prin structuri de adiacenţe respectiv cu $O(n^2)$ în reprezentarea bazată pe matrice de adiacenţe, `n` reprezentând numărul de noduri, iar `a` numărul de arce al grafului.