Laborator 08: Aplicatii DFS

Obiective laborator

Întelegerea noţiunilor teoretice:

- tare conexitate, componente tare conexe (pentru grafuri orientate)
- punct de articulaţie (pentru grafuri neorientate)
- punţi (pentru grafuri neorientate)
- componente biconexe (în general, pentru grafuri neorientate)

Înţelegerea algoritmilor ce rezolvă aceste probleme şi implementarea acestor algoritmi.

Importanţă – aplicaţii practice

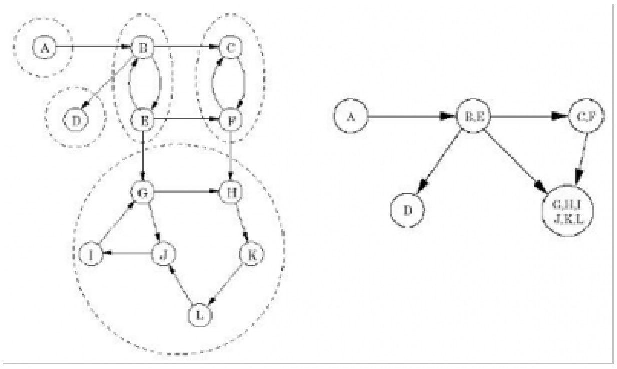
- Componentele biconexe au aplicaţii importante în reţelistică, deoarece o componentă biconexă asigură redundanţa.
- Descompunerea în componente tare conexe: data mining, compilatoare, calcul ştiinţific, 2SAT

Notiuni teoretice

- **Tare conexitate.** Un graf orientat este tare conex, dacă oricare ar fi două vârfuri u şi v, ele sunt tare conectate (strongly connected) - există drum atât de la u la v, cât şi de la v la u.

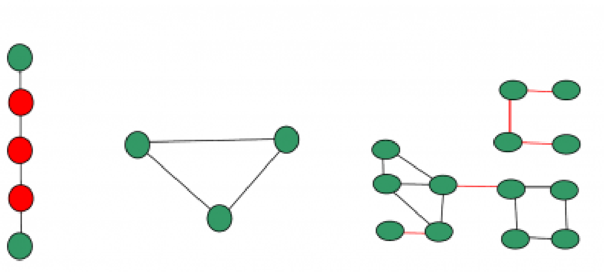
- O componentă tare conexă este un subgraf maximal tare conex al unui graf orientat, adică o submulţime de vârfuri U din V, astfel încât pentru orice u şi v din U ele sunt tare conectate. Dacă fiecare componentă tare conexă este redusă într-un singur nod, se va obţine **un graf orientat aciclic**.

De exemplu:



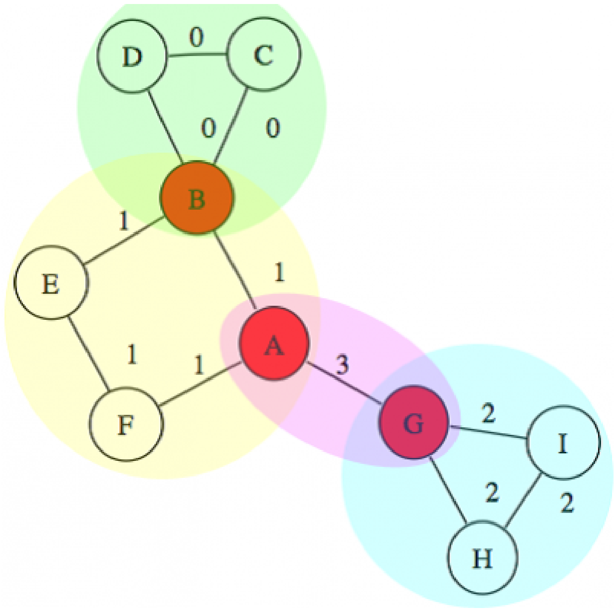
- **Un punct de articulaţie (cut vertex)** este un nod al unui graf a cărui eliminare duce la creşterea numărului de componente conexe ale acelui graf.

- **O punte (bridge)** este o muchie a unui graf (se mai numeşte şi **muchie critică**) a cărei eliminare duce la creşterea numărului de componente conexe ale acelui graf.



- **Biconexitate**. Un graf biconex este un graf conex cu proprietatea că eliminând oricare nod al acestuia, graful rămâne conex.

- O componentă biconexă a unui graf este o mulţime **maximală** de noduri care respectă proprietatea de biconexitate.



Componente tare conexe

Vom porni de la definiţie pentru a afla componenta tare conexă din care face parte un nod v. Vom parcurge graful (DFS sau BFS) pentru a găsi o mulţime de noduri S ce sunt accesibile din v. Vom parcurge apoi graful transpus (obţinut prin inversarea arcelor din graful iniţial), determinând o nouă mulţime de noduri T ce sunt accesibile din v în graful transpus. Intersecţia dintre S şi T va reprezenta componenta tare conexa. Graful iniţial şi cel transpus au aceleaşi componente conexe.

Algoritmul lui Kosaraju

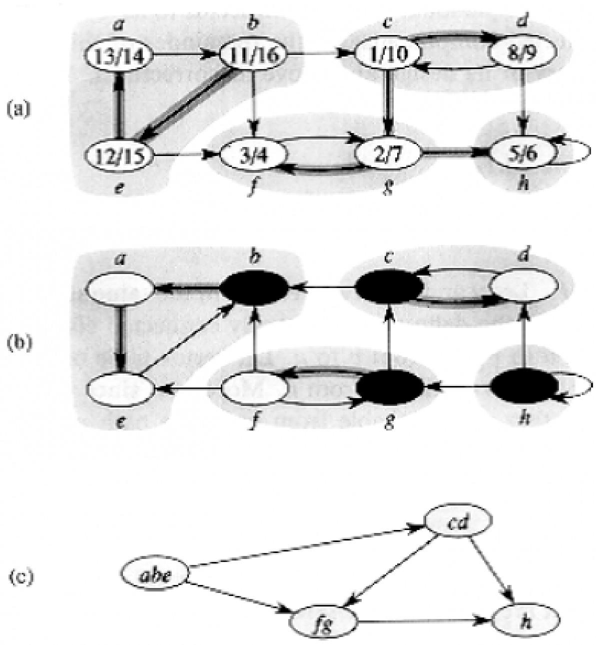
Algoritmul foloseşte două DFS (una pe graful iniţial şi una pe graful transpus) şi o stivă pentru a reţine ordinea terminării parcurgerii nodurilor grafului original (evitând astfel o sortare a nodurilor după acest timp la terminarea parcurgerii).

```
ctc(G = (V, E))
        S <- stiva vida
        culoare[1..n] = alb
        cat timp exista un nod v din V care nu e pe stiva
                dfs(G, v)
        culoare[1..n] = alb
        cat timp S != stiva vida
                v = pop(S)
                dfsT(GT, v) /* toate nodurile ce pot fi vizitate din v fac  parte din ctc; dupa vizitare, aceastea sunt scoase din S si din G */

dfs(G, v)
        culoare[v] = gri
        pentru fiecare (v, u) din E
        daca culoare[u] == alb
                dfs(u)
        push(S, v) // nodul este terminat de expandat, este pus pe stiva
        culoare[v] = negru

dfsT(G, v) – similar cu dfs(G, v): fara stiva, dar cu retinerea solutiei
```

Complexitate: O(|V| + |E|)



Algoritmul lui Tarjan

Algoritmul foloseşte o singură parcurgere DFS şi o stivă. Ideea de bază a algoritmului este că o parcurgere în adâncime porneşte dintr-un nod de start. Componentele tare conexe formează subarborii arborelui de căutare, rădăcinile cărora sunt de asemenea rădăcini pentru componentele tare conexe.

Nodurile sunt puse pe o stivă, în ordinea vizitării. Când parcurgerea termină de vizitat un subarbore, nodurile sunt scoase din stivă şi se determină pentru fiecare nod dacă este rădăcina unei component tare conexe. Dacă un nod este rădăcina unei componente, atunci el şi toate nodurile scoase din stivă înaintea lui formează acea componenta tare conexă.

Pentru a determina daca un nod este radacina unei componente conexe, calculam pentru fiecare nod:

```
idx[v]     -> nivelul / ordinea de vizitare
lowlink[v] -> min { idx[u] | u este accesibil din v, folosind doar noduri din subarborele DFS al lui v (inclusiv v) }
```

v este rădăcina unei componente tare conexe ⇔ lowlink[v] = idx[v].

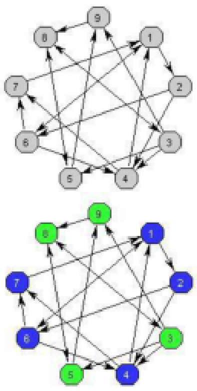
```
ctc_tarjan(G = (V, E))
        index = 0
        S = stiva vida
        pentru fiecare v din V
                daca (idx[v] nu e definit) // nu a fost vizitat
                        tarjan(G, v)

tarjan(G, v)
        idx[v] = index
        lowlink[v] = index
        index = index + 1
        push(S, v)

        pentru (v, u) din E
                daca (idx[u] nu e definit)
                        tarjan(G, u)
                        lowlink[v] = min(lowlink[v], lowlink[u])
                altfel
                        daca (u e in S)
                                lowlink[v] = min(lowlink[v], idx[u])

        daca (lowlink[v] == idx[v])
                // este v radacina unei CTC?
                print "O noua CTC: "
                repeat
                        u = pop(S)
                        print u
                until (u == v)
```

Complexitate: O(|V| + |E|)



	id	pre	low
1	1	0	10
2	1	1	10
3	2	5	10
4	1	2	10
5	2	6	10
6	1	4	10
7	1	3	10
8	2	7	10
9	2	8	10

Puncte de articulatie

Pentru determinarea punctelor de articulaţie într-un graf neorientat se foloseşte o parcurgere în adâncime modificată, reţinându-se informaţii suplimentare pentru fiecare nod. Acest algoritm a fost identificat tot de către Tarjan si este foarte similar cu algoritmul pentru determinarea CTC în grafuri orientate prezentat anterior. Fie T un arbore de adâncime descoperit de parcurgerea grafului. Atunci, un nod v este punct de articulaţie dacă:

- v este rădăcina lui T şi v are doi sau mai mulţi copii

sau

- v nu este rădăcina lui T şi are un copil u în T, astfel încât nici un nod din subarborele dominat de u nu este conectat cu un strămoş al lui v printr-o muchie înapoi (copii lui nu pot ajunge pe altă cale pe un nivel superior în arborele de adâncime).

Găsirea punctelor care se încadrează în primul caz este uşor de realizat.

Notăm:

```
idx[u] = timpul de descoperire a nodului u
low[u] = min( {idx[u]} U { idx[v] : (u, v) este o muchie înapoi } U
        { low[vi] : vi copil al lui u în arborele de adâncime} )
```

v este punct de articulaţie ⇔ low[u] ≥ idx[v], pentru cel puţin un copil u al lui v în T.

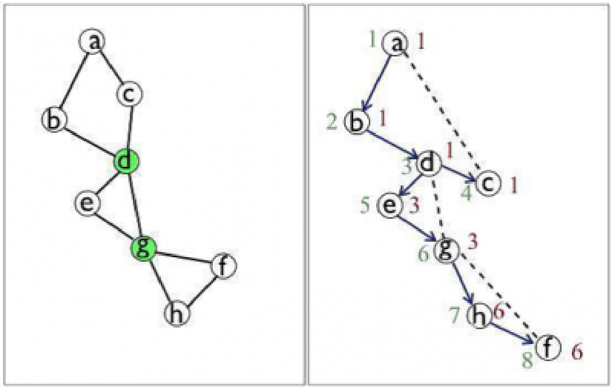
```
puncte_articulatie(G = (V, E))
        timp = 0
```

```
        pentru fiecare v din V
        daca (idx[v] nu e definit)
                dfsCV(G, v)

dfsCV(G, v)
        idx[v] = timp
        low[v] = timp
        timp = timp + 1
        copii = { } // multime vida
        pentru fiecare (v, u) din E
                daca (idx[u] nu e definit)
                        // inseamna ca nodul u este nedescoperit, deci alb
                        copii = copii U {u}
                        dfsCV(G, u)
                        low[v] = min(low[v], low[u])
                altfel
                        // inseamna ca nodul u este descoperit, deci gri, iar muchia v->u este muchie inapoi
                        low[v] = min(low[v], idx[u])

        daca v radacina arborelui
                daca |copii| >= 2
                        v este punct de articulatie
        altfel
                daca (∃u ∈ copii) astfel incat (low[u] >= idx[v])
                        v este punct de articulatie
```

Complexitate: O(|V| + |E|)



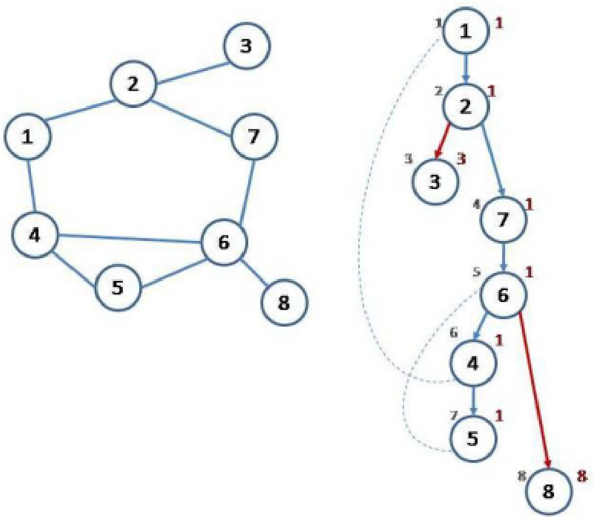
Punti

Pentru a determina muchiile critice se foloseşte tot o parcurgere în adâncime modificată, pornind de la următoarea observaţie: **muchiile critice sunt muchiile care nu apar în niciun ciclu.** Prin urmare, o muchie de întoarcere nu poate fi critică, deoarece o astfel de muchie închide întotdeauna un ciclu. Trebuie să verificăm pentru muchiile de avansare (în număr de |V| - 1) dacă fac parte dintr-un ciclu. Să considerăm că dorim să verificăm muchia de avansare (v, u).

Ne vom folosi de low[v] (definit la punctul anterior): dacă din nodul u putem să ajungem pe un nivel mai mic sau egal cu nivelul lui v, atunci muchia nu este critică, în caz contrar ea este critică.

```
dfsB(G, v, parinte)
        idx[v] = timp
        low[v] = timp
        timp = timp + 1
        pentru fiecare (v, u) din E
                daca u nu este parintele lui v
                        daca idx[u] nu este definit
                                dfsB(G, u, v)
                                low[v] = min(low[v], low[u])
                                daca (low[u] > idx[v])
                                        (v, u) este muchie critica
                        altfel
                                low[v] = min(low[v], idx[u])
```

Complexitate: O(|V| + |E|)



Componente biconexe

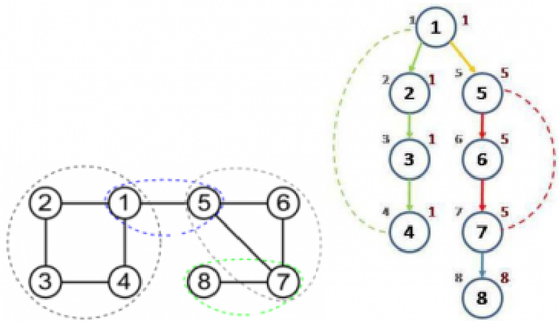
O componenta biconexă (sau biconectată) este o componentă a grafului care nu conţine puncte de articulatie. Astfel după eliminarea oricărui vârf din componenta curentă, restul vârfurilor vor rămâne conectate întrucât între oricare două vărfuri din aceeași componentă biconexă există cel puțin două căi disjuncte.

Astfel, pentru a determina componentele biconexe ale unui graf, vom adapta algoritmul de aflare a punctelor critice, reţinând şi o stivă cu toate muchiile de avansare şi de întoarcere parcurse până la un moment dat. La întâlnirea unui nod critic v se formează o nouă componentă biconexă pe care o vom determina extrăgând din stivă muchiile corespunzătoare. Nodul **v** este critic dacă am găsit un copil **u** din care nu se poate ajunge pe un nivel mai mic în arborele de adâncime pe un alt drum care foloseşte muchii de întoarcere **(low[u] >= idx[v])**. Atunci când găsim un astfel de nod **u**, toate muchiile aflate în stivă până la muchia (v, u) inclusiv formează o nouă componentă biconexă.

Nodul rădăcină trebuie tratat separat. Conform regulilor acestuia pentru a fi marcat nod critic (numărul de copii >= 2), este suficient să considerăm că fiecare copil face parte dintr-o componentă biconexă separată.

Atenție! Împărțirea în componente biconexe a unui graf neorientat reprezintă o partiție disjunctă a muchiilor grafului (împreună cu vârfurile adiacente muchiilor corespunzătoare fiecărei componente în parte). Acest lucru implică că unele vârfuri pot face parte din mai multe componente biconexe diferite. Care sunt acestea?

Complexitate: O(|V| + |E|)



Concluzii

Algoritmul de parcurgere în adâncime poate fi modificat pentru calculul componentelor tare conexe, a punctelor de articulaţie, a punţilor şi a componentelor biconexe. Complexitatea acestor algoritmi va fi cea a parcurgerii: O(|V| + |E|).

Referinte

[1] Introducere in Algoritmi, Thomas H. Cormen, Charles E. Leiserson, Ronald L. – Capitolul 23 Algoritmi elementari pe grafuri http://net.pku.edu.cn/~course/cs101/resource/Intro2Algorithm/book6/chap23.htm [http://net.pku.edu.cn/~course/cs101/resource/Intro2Algorithm/book6/chap23.htm]

[2] Wikipedia – Algoritmul lui Tarjan http://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm [http://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm]

[3] Wikipedia – Algoritmul lui Kosaraju http://en.wikipedia.org/wiki/Kosaraju%27s_algorithm [http://en.wikipedia.org/wiki/Kosaraju%27s_algorithm]

[4] Wikipedia - Componente biconexe http://en.wikipedia.org/wiki/Biconnected_component [http://en.wikipedia.org/wiki/Biconnected_component]

[5] Infoarena - Arhiva educationala - Componente tari conexe http://infoarena.ro/problema/ctc [http://infoarena.ro/problema/ctc]

[6] Infoarena - Arhiva educationala - Componente biconexe http://infoarena.ro/problema/biconex [http://infoarena.ro/problema/biconex]

[7] Infoarena - Arhiva educationala - 2SAT http://infoarena.ro/problema/2sat [http://infoarena.ro/problema/2sat]

Exercitii

In acest laborator vom folosi scheletul de laborator din arhiva skel-lab08.zip.

Inainte de a rezolva exercitiile, asigurati-va ca ati citit si inteles toate precizarile din sectiunea Precizari laboratoare 07-12 [https://ocw.cs.pub.ro/courses/pa/skel_graph].

Prin citirea acestor precizari va asigurati ca:

- cunoasteti **conventiile** folosite
- evitati **buguri**
- evitati **depunctari** la lab/teme/test

CTC

Se da un graf **orientat** cu **n** noduri si **m** arce. Sa se gaseasca **componentele tare-conexe**.

Restrictii si precizari:

- $n <= 10^5$
- $m <= 2 * 10^5$
- timp de executie
  - C++: 1s
  - Java: 2s

Rezultatul se va returna sub forma unui vector, unde fiecare element este un vector (o CTC).

CUT VERTEX

Se da un graf **neorientat conex** cu **n** noduri si **m** muchii. Se cere sa se gaseaca toate **punctele critice**.

Restrictii si precizari:

- $n <= 10^5$
- $m <= 2 * 10^5$
- timp de executie
  - C++: 1s
  - Java: 2s

Rezultatul se va returna sub forma unui vector cu **X** elemente, unde X este numarul de puncte critice din graf.

CRITICAL EDGE

Se da un graf **neorientat conex** cu **n** noduri si **m** muchii. Se cere sa se gaseaca toate **muchiile critice**.

Restrictii si precizari:

- $n <= 10^5$
- $m <= 2 * 10^5$
- timp de executie
  - C++: 1s
  - Java: 2s

Rezultatul se va returna sub forma unui vector cu **X** elemente, unde X este numarul de muchii critice.

BONUS

Se da un graf **neorientat conex** cu **n** noduri si **m** muchii. Se cere sa se gaseaca toate **componentele biconexe**.

Pentru testare **trebuie** sa folositi problema biconex [https://infoarena.ro/problema/biconex] de pe infoarena.

Extra

Rezolvati problema retele [https://infoarena.ro/problema/retele] pe infoarena.
Rezolvati **ACASA** problema clepsidra [https://infoarena.ro/problema/clepsidra] pe infoarena.
Rezolvati **ACASA** problema course-schedule [https://leetcode.com/problems/course-schedule/description/] pe leetcode. (aplicatie tipuri de muchii)