

Proiectarea Algoritmilor 2009-2010

Laborator 7

Aplicații DFS

Cuprins

1	Obiective laborator	1
2	Importanță – aplicații practice	1
3	Noțiuni teoretice	2
4	Componente tare conexe	3
5	Puncte de articulație	5
6	Punți	6
7	Componente biconexe	7
8	Concluzii	7
9	Referințe	7
10	Problema 1	9
11	Problema 2	9
12	Problema 3	9

Obiective laborator

Înțelegerea noțiunilor teoretice: tare conexitate, componente tare conexe, punct de articulație, punți componente biconexe.

Înțelegerea algoritmilor ce rezolvă aceste probleme și implementarea acestor algoritmi.

Importanță – aplicații practice

Componentele biconexe au aplicații importante în rețelistică, deoarece o componentă biconexă asigură redundanța.

Tare conexitate: problema 2-SAT.

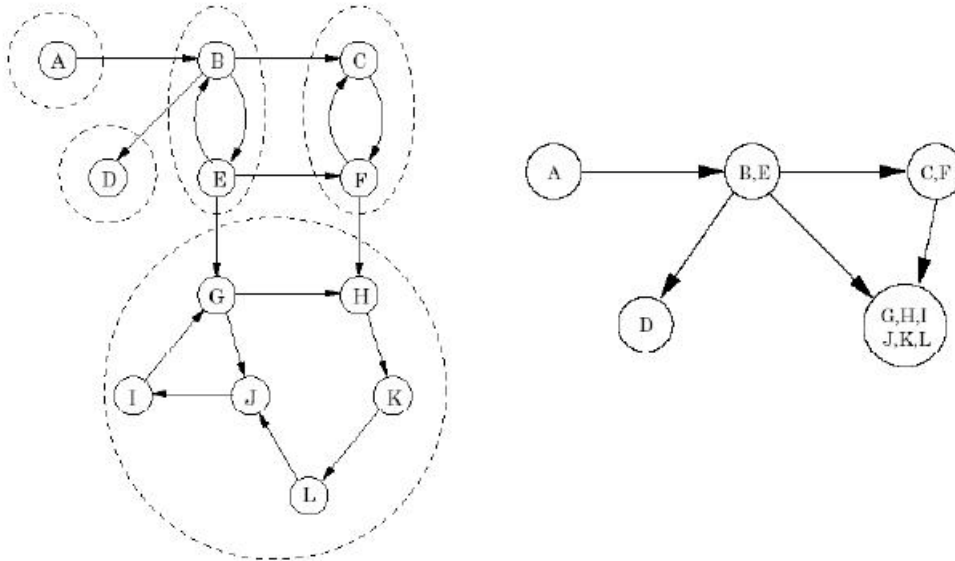
Descompunerea în componente tare conexe: data mining, compilatoare, calcul științific.

Noțiuni teoretice

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

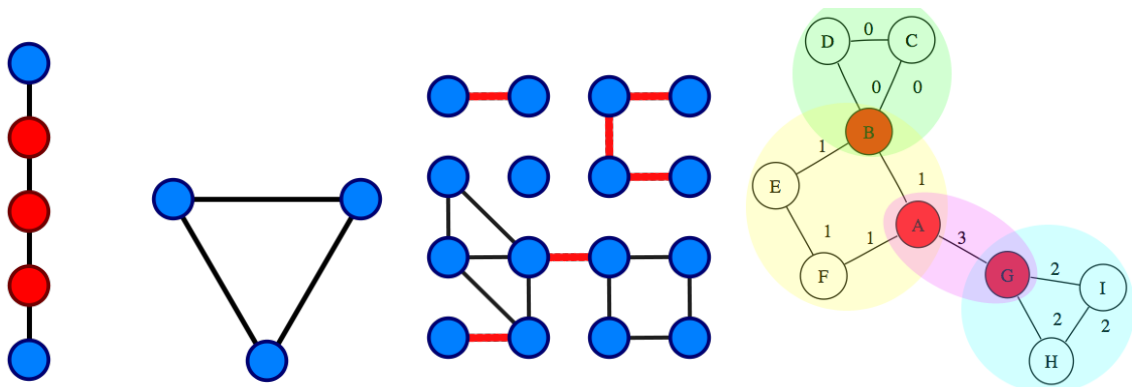
O **componentă tare conexă** este un subgraf maximal tare conex al unui graf orientat, adică o submulțime de vârfuri \cup din v , astfel încât pentru orice u și v din \cup ele sunt tare conectate.

Dacă fiecare componentă tare conexă este redusă într-un singur nod, se va obține un graf orientat aciclic. De exemplu:



Un **punct de articulație** (cut vertex) este un nod al unui graf a cărui eliminare duce la creșterea numărului de componente conexe ale unui același graf.

O **punte** (bridge) este o muchie a unui graf (se mai numește și muchie critică) a cărei eliminare duce la creșterea 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 mulțime maximală de muchii care respectă proprietatea de biconexitate.

Componente tare conexe

Vom porni de la definiție pentru a afla componenta tare conexă din care face parte un nod v . Vom parcurge graful (DFS sau BFS) pentru a găsi o mulțime de noduri S ce sunt accesibile din v . Vom parcurge apoi graful transpus (obținut prin inversarea muchiilor din graful inițial), determinând o nouă mulțime de noduri T ce sunt accesibile din v în graful transpus. Intersecția dintre S și T va fi reprezenta componenta tare conexa. Graful inițial și cel transpus au aceleași componente conexe.

Algoritmul lui Kosaraju

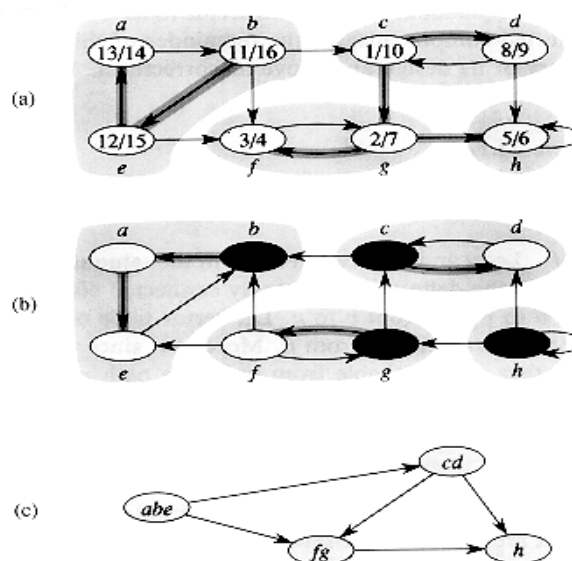
Algoritmul folosește două DFS (una pe graful inițial și una pe graful transpus) și o stivă pentru a reține ordinea terminării parcurgerii nodurilor grafului original (evitând astfel o sortare a nodurilor după acest timp la terminarea parcurgerii).

```
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, acestea 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 folosește o singură parcurgere DFS și o stivă. Ideea de bază a algoritmului este că o parcurgere în adâncime pornește dintr-un nod de start. Componentele tare conexe formează subarborii arborelui obținut în urma parcurgerii în adâncime, rădăcinile cărora sunt de asemenea rădăcini pentru componentele tare conexe (numim rădăcina unei componente tare conexe nodul care domină subarborii ale cărui noduri formează o componentă tare conexă).

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

Cea mai importantă parte este să determinăm dacă un nod este sau nu rădăcina unei componente tare conexe. Pentru a face asta, fiecărui nod i se atribuie un index în urma parcurgerii în adâncime ($idx[v]$), ce numără nodurile în ordinea în care sunt descoperite.

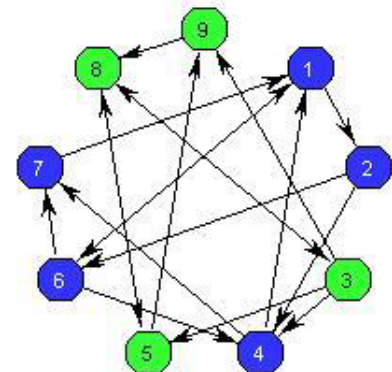
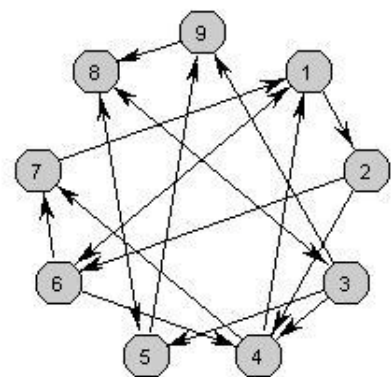
În plus, pentru fiecare nod se ține și o valoare $lowlink[v] = \min \{ idx[u] \mid u \text{ este accesibil din } v \}$. Prin urmare, v este rădăcina unei componente tare conexe dacă și numai dacă $lowlink[v] = idx[v]$. $lowlink[v]$ se calculează în timpul parcurgerii în adâncime.

```
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 articulație

Pentru determinarea punctelor de articulație într-un graf neorientat se folosește o parcurgere în adâncime modificată, reținându-se informații suplimentare pentru fiecare nod.

Fie T un arbore de adâncime descoperit de parcurgerea grafului. Atunci, un nod v este punct de articulație dacă:

- v este rădăcina lui T și v are doi sau mai mulți copii

sau

- v nu este rădăcina lui T și are un copil u în T , astfel încât nici un nod din subarborele dominat de u nu este conectat cu un strămoș 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 ușor de realizat.

Notăm:

```
d[v] = timpul de descoperire a nodului u
low[v] = min { { d[v] } U { d[x] | (v, x) este o muchie înapoi de la un
descendent v al lui u } }
```

Pentru a putea găsi ușor punctele de articulație care se încadrează în cel de-al doilea caz se va calcula și $low[v]$ pentru fiecare vârf, în timpul parcurgerii în adâncime.

v va fi punct de articulație dacă și numai dacă $low[u] \geq d[v]$, pentru un copil u al lui v în T .

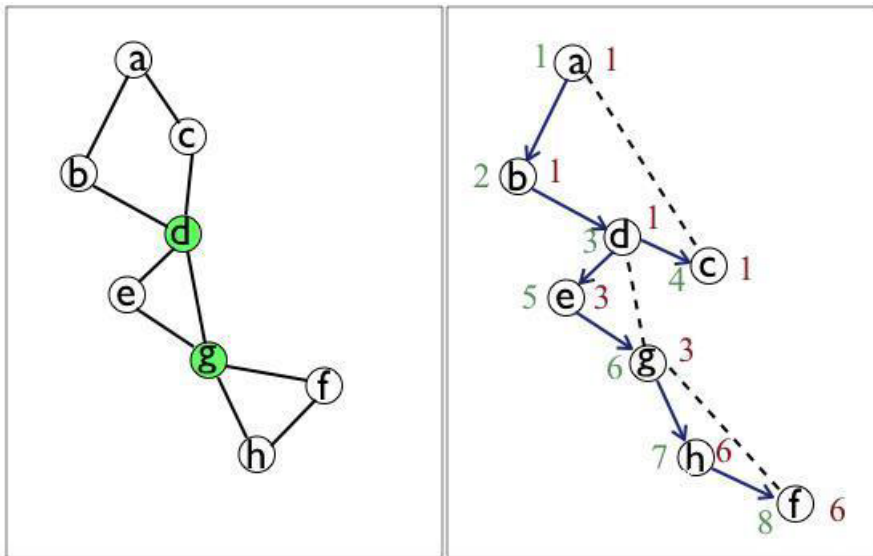
Funcția low se poate calcula folosind relația de recurență:

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

```
puncte_articulatie(G = (V, E))
  timp = 0
  pentru fiecare v din V
    daca (d[v] nu e definit)
      dfsCV(G, v)
```

```
dfsCV(G, v)
  d[v] = timp
  timp = timp + 1
  low[v] = d[v]
  copii = { } // multime vida
  pentru fiecare (v, u) din E
    daca (d[u] nu e definit)
      copii = copii U {u}
      dfsCV(G, u)
      low[v] = min(low[v], low[u])
    altfel daca u != parinte(v)
      low[v] = min(low[v], d[u])
  daca v radacina arborelui
    daca |copii| >= 2
      v este punct de articulatie
  altfel daca (∃u ∈ copii) astfel incat (low[u] >= d[v])
```

v este punct de articulatie
 Complexitate: $O(|V| + |E|)$



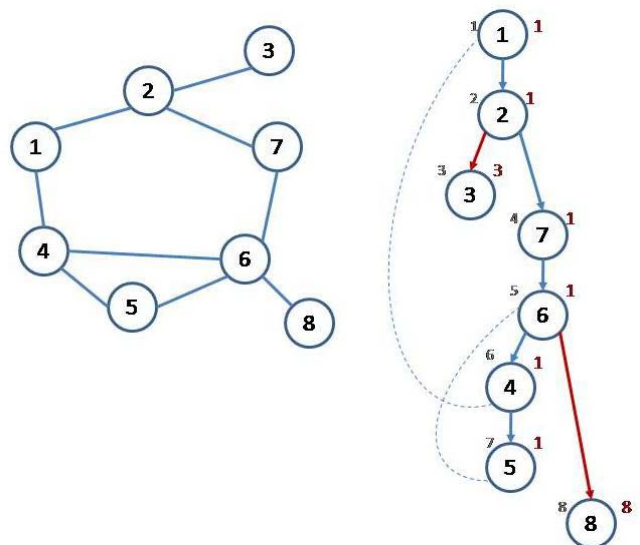
Punți

Pentru a determina muchiile critice se folosește tot o parcurgere în adâncime modificată, pornind de la următoarea observație: 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)
d[v] = timp
timp = timp + 1
low[v] = d[v]
pentru fiecare (v, u) din E
    daca (d[u] nu e definit)
        dfsB(G, v)
        low[v] = min(low[v], low[u])
        if (low[u] > d[v])
            (v, u) este muchie critica
        altfel daca u != parinte(v)
            low[v] = min(low[v], d[u])
```

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



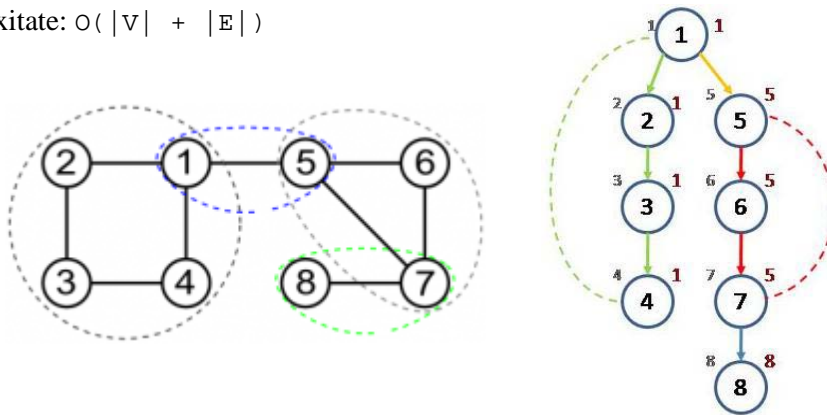
Componente biconexe

Împărțirea în componente biconexe nu introduce ca și până acum (conexitate, tare conexitate) o partiție a nodurilor grafului, ci o partiție a muchiilor acestuia.

Se remarcă ușor că o componentă biconexă este o componentă care nu conține puncte critice.

Astfel, pentru a determina componentele biconexe ale unui graf, vom adapta algoritmul de aflare a punctelor critice, reținând și o stivă cu toate muchiile de avansare și 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 folosește muchii de întoarcere ($low[u] \geq d[v]$). Atunci când găsim un astfel de nod u , toate muchiile aflate în stivă până la muchia (u, v) inclusiv formează o nouă componentă biconexă.

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



Concluzii

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

Responsabil laborator: [Andrada Georgescu \(georgescu.andrada@gmail.com\)](mailto:georgescu.andrada@gmail.com)

Referințe

[1] Introducere în 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>

[2] Wikipedia – Algoritmul lui Tarjan

http://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm

[3] Wikipedia – Algoritmul lui Kosaraju

http://en.wikipedia.org/wiki/Kosaraju%27s_algorithm

[4] Gazeta Informatică, Vol 10, nr. 3 – Introducere in teoria grafurilor

http://www.ginfo.ro/10_4/051.shtml

Proiectarea Algoritmilor 2009-2010

Laborator 7

Aplicații DFS - aplicații

Pentru toate problemele, fișierele de intrare au formatul: pe prima linie, două numere naturale N și M ce reprezintă numărul de noduri din graf și numărul muchiilor. Pe următoarele M linii se vor afla câte două numere naturale u și v , separate prin spațiu, reprezentând muchia (u, v) .

Problema 1

Se dă un graf orientat. Determinați componentele tare conexe ale acestuia folosind algoritmul lui Kosaraju.

[Bonus] Implementați algoritmul lui Tarjan în loc de Kosaraju.

Puteți încerca să vă testați implementarea (C/C++) aici: <http://infoarena.ro/problema/etc>

Problema 2

Se dă un graf neorientat. Determinați punctele de articulație.

[Bonus] Determinați și componentele biconexe ale grafului.

Puteți încerca să vă testați implementarea (C/C++) aici: <http://infoarena.ro/problema/biconex>

Problema 3

Se dă un graf neorientat. Determinați muchiile sale critice.