Grafuri

Traversari. Parcurgerea in latime.
Parcurgerea in adancime. Aplicatii

Definitie

- Parcurgerea unui graf: procesul de vizitare (actualizare, verificare) a fiecarui nod din graf.
- Parcurgerile sunt clasificate dupa ordinea in care sunt vizitate varfurile din graf:
  - Parcurgere in adancime (depth-first search (DFS))
  - Parcurgere in latime (breadth-first search (BFS))

Metodologie

- Algoritmii de parcurgere a grafurilor incep cu un varf de start si incearca sa viziteze restul varfurilor incepand de la varful de start.
- Cazuri exceptionale:
  - Incepand de la un varf dat e posibil sa nu se poata ajunge la celelalte varfuri – graf ne-conex.
  - Daca exista cicluri in graf – ne asiguram ca algoritmul evita intrarea intr-o bucla infinita de parcurgere.
  - Pentru a evita cazurile de mai sus se foloseste o eticheta – VIZITAT pentru a marca varfurile parcurse

Metodologie

```
void graphTraverse(Graph  G) {
    int v;
    for (v=0; v<G.nodeCount(); v++)
        G.setValue(v, null); // Initialize
    for (v=0; v<G.nodeCount(); v++)
        if (G.getValue(v) != VISITED)
                doTraversal(G, v);
}
```

Functia doTraversal se poate implementa folosind:

- **parcurgere in adancime**
- **parcurgere in latime**

Parcurgerea in latime

- *Breadth first search (BFS)*
- Functioneaza atat pe grafuri orientate cat si pe grafuri neorientate
- Parcurge toate varfurile conectate la nodul curent inainte de a trece mai departe
- Foloseste o coada pentru a mentine ordinea de parcurgere
- Verifica daca un nod a fost vizitat inainte sa il puna in coada

Parcurgere in latime - pseudocod

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2       u.color = WHITE
 3       u.d = ∞
 4       u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11       u = DEQUEUE(Q)
12       for each v ∈ G.Adj[u]
13            if v.color == WHITE
14                 v.color = GRAY
15                 v.d = u.d + 1
16                 v.π = u
17                 ENQUEUE(Q, v)
18       u.color = BLACK
```

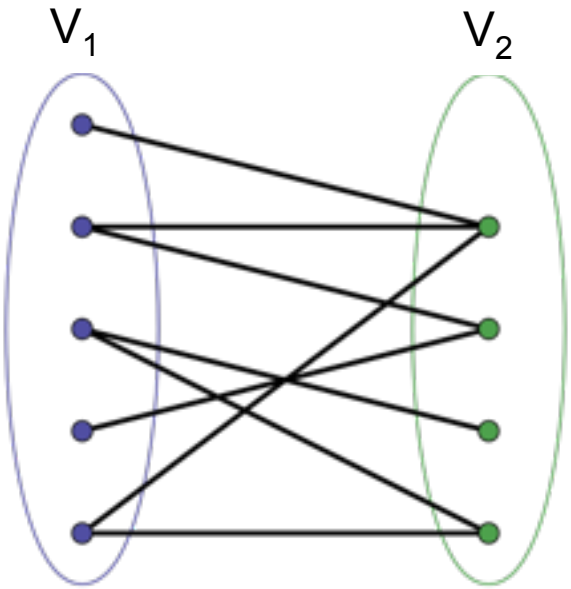
- Culori pentru a tine evidenta starii de parcurgere a nodurilor
- Calculeaza distanta de la nodul sursa la toate nodurile conectate cu el
- Produce un arbore de parcurgere in latime pentru fiecare nod sursa, *s*
- Expandeaza frontiera dintre nodurile descoperite si cele nedescoperite in mod uniform pe latimea ei (toate nodurile de la dist *k* sunt descoperite inainte de cele de la dist *k+1*

Parcurgere in latime

- Atributul *distanta* al fiecarui nod este folosit pentru a cauta cel mai scurt drum intre doua noduri din graf.
- La inceputul algoritmului distanta pentru fiecare nod este infinit – ceea ce semnifica faptul ca nodul respectiv nu a fost vizitat.
- Atributul *parinte* al fiecarui nod poate fi folosit pentru a accesa nodurile din drumul cel mai scurt
- *Eficienta?*

Parcurgerea in latime - aplicatii

- Gasirea drumurilor de lungime minima
- Graf bipartit
  - Gasirea unui ciclu de lungime impara



G=(V,E) – bipartit iff
$V = V_1 \cup V_2$, $V_1 \cap V_2 = \emptyset$,
$\forall (u, v) \in E \Rightarrow u \in V_1, v \in V_2$ ***or*** $u \in V_2, v \in V_1$

Un graf bipartit nu poate contine un ciclu de lungime impara!!

Graf bipartit

```
1  Assume graph G is connected. Otherwise, we can run the algorithm for each connected
2  Let q be an empty queue
3
4  Pick any vertex s ∈ V and color it Red
5  q.enqueue(s)
6
7  while !q.empty()
8    u = q.dequeue()
9    foreach v in u.adjList:
10     if v.color is nil:
11       v.color = (u.color == Red) ? Black : Red
12       q.enqueue(v)
13     elif v.color == u.color:
14       return "Not Bipartite"
15 return "Bipartite"
```

Se poate aplica si DFS!

Cum se modifica algoritmul pentru a detecta ciclul de lungime impara?

Aplicatii:

- teoria codurilor - decodificare
- retele Petri - analiza si simularea sistemelor concurente

Parcurgere in adancime

- *Depth first search (DFS)*
- se aplica pe grafuri orientate si neorientate
- Muchiile sunt explorate pornind din varful *v* cel mai recent descoperit, care mai are inca muchii neexplorate ce pleaca din el.
- Cand toate muchiile care pleaca din *v* au fost explorate, parcurgerea revine pe propriile urme pentru a explora muchiile care pleaca din varful din care a fost descoperit *v*
- Parcurgerea foloseste o structura de tip ***stiva*** pentru a stoca muchiile care pleaca din *v* si nu au fost explorate

Pseudocod – parcurgere in adancime

- **Intrare**: Un graf G si un varf *v* din G
- **Iesire**: Arborele de parcurgere in adancime, care contine varfurile la care se poate ajunge pornind de la *v*
- Varianta iterativa:

```
1  procedure DFS-iterative(G,v):
2      let S be a stack
3      S.push(v)
4      while S is not empty
5            v = S.pop()
6            if v is not labeled as discovered:
7                label v as discovered
8                for all edges from v to w in
G.adjacentEdges(v) do
9                    S.push(w)
```

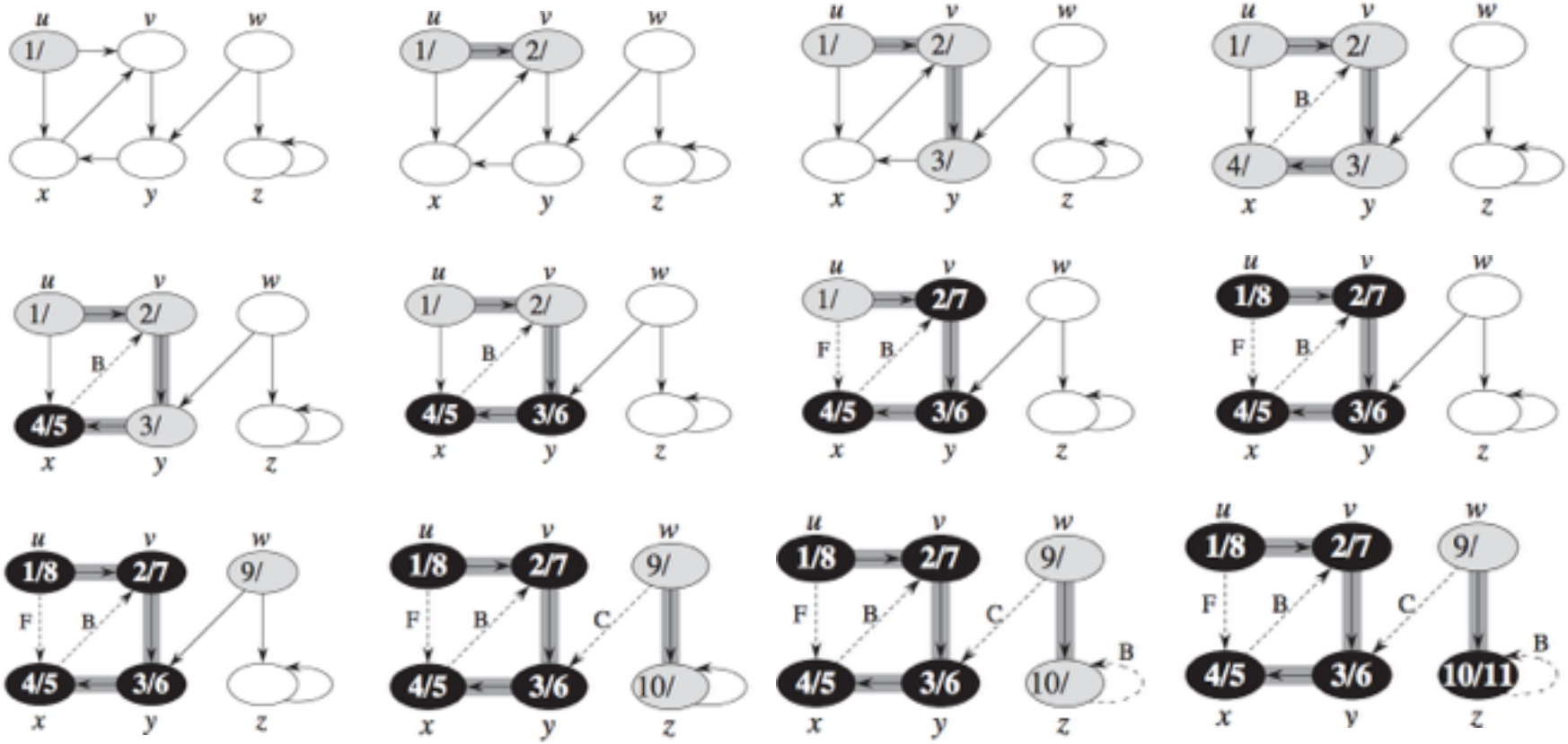
Pseudocod – parcurgere in adancime

- **Intrare**: Un graf G si un varf *u* din G, de pornire
- **Iesire**: Arborele de parcurgere in adancime, care contine varfurile la care se poate ajunge pornind de la u
  - Inregistreaza momentele de timp – descoperire, finalizare
- Varianta recursiva:

```
DFS-VISIT(G, u)
 1  time = time + 1                  // white vertex u has just been discovered
 2  u.d = time
 3  u.color = GRAY
 4  for each v ∈ G.Adj[u]            // explore edge (u, v)
 5      if v.color == WHITE
 6          v.π = u
 7          DFS-VISIT(G, v)
 8  u.color = BLACK                  // blacken u; it is finished
 9  time = time + 1
10  u.f = time
```

Eficienta?

Parcurgerea in adancime: Exemplu



Parcurgerea in adancime: Alte exemple

DFS - iterativ:
https://drive.google.com/open?id=0B0abXFghjTINT2JoVGxuZlpUOUk

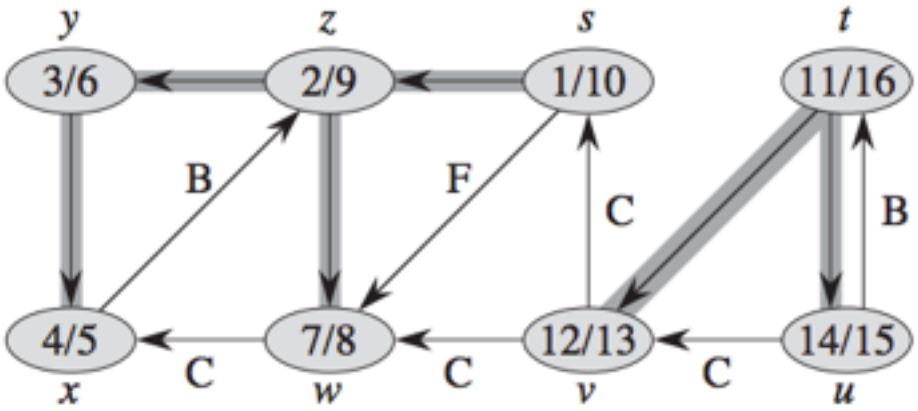
DFS - recursiv:
https://drive.google.com/open?id=0B0abXFghjTINTC14cHRBU2p6djg

Proprietati ale parcurgerii in adancime

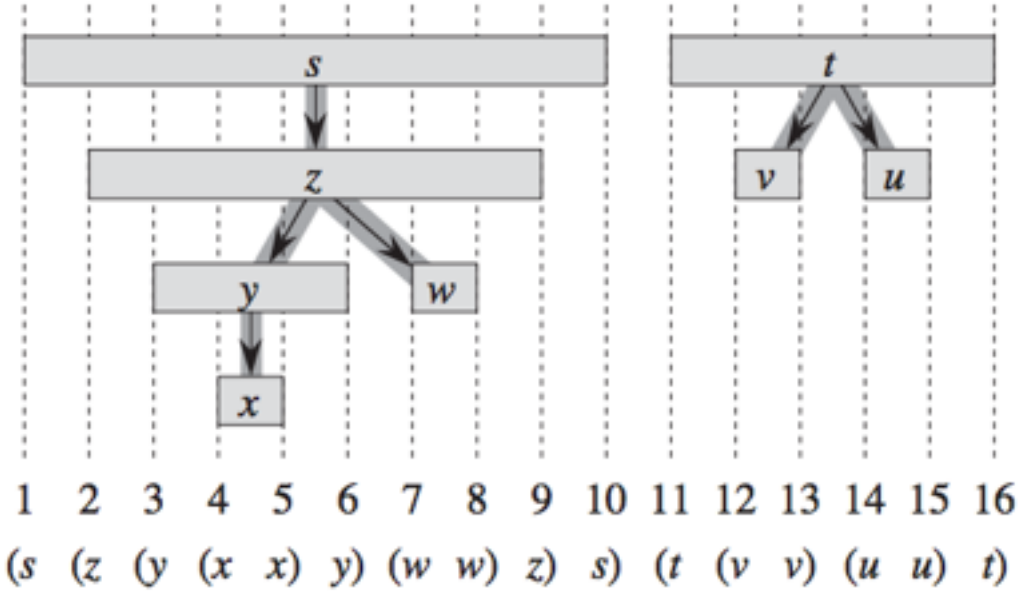
- Cautarea in adancime ofera multe informatii despre structura unui graf.
- ***Structura de paranteza*** a timpilor de descoperire si terminare:
  - Daca reprezentam descoperirea unui varf u printr-o paranteza deschisa **(u**, si terminarea sa printr-o paranteza inchisa ***u)***, atunci istoria descoperirilor si a terminarilor formeaza o expresie bine formata (i.e. parantezele sunt corect imperecheate)

Teorema parantezelor

- In orice traversare DFS a unui graf G=(V,E), pentru oricare 2 varfuri – *u* si *v*, exact una din urmatoarele 3 conditii are loc:
  - Intervalele ***[u.d, u.f]*** si ***[v.d, v.f]*** sunt complet ***disjuncte*** (nici u, nici v nu este descendent al celuilalt nod in padurea de arbori de parcurgere in adancime)
  - ***[u.d, u.f]*** e inclus in ***[v.d, v.f]*** (*u* este descendent al lui *v* intr-un arbore DFS)
  - ***[v.d, v.f]*** este inclus in ***[u.d, u.f]*** (*v* este descendent al lui *u*)



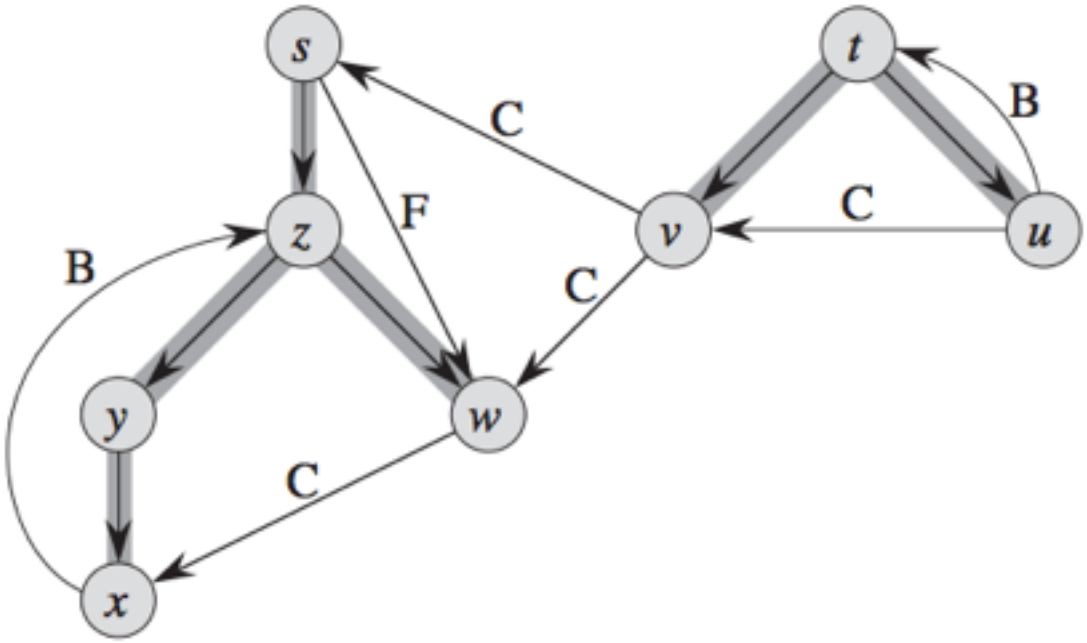
Teorema parantezelor – exemplu



Proprietati ale parcurgerii in adancime

- Clasificarea muchiilor:
- Tipuri de muchii:
  - Muchii de arbore
  - Muchii inapoi
  - Muchii inainte
  - Muchii transversal
- Algoritmul DFS poate fi modificat pentru a clasifica muchiile pe masura ce le intalneste.

Clasificarea muchiilor - exemplu



Aplicatii ale parcurgerii in adancime

- **Determinarea ciclurilor intr-un graf (cum?)**
- **Determinarea componentelor conexe/ puternic conexe ale unui graf**
- **Sortarea topologica**
- Gasirea puntilor (*bridges*) dintr-un graf nedirectionat
  - Punte = muchie a carei stergere creste numarul de componente conexe ale grafului
- Testarea planaritatii
  - Graful poate fi desenat in plan fara a exista intersectii de muchii
- Etc.

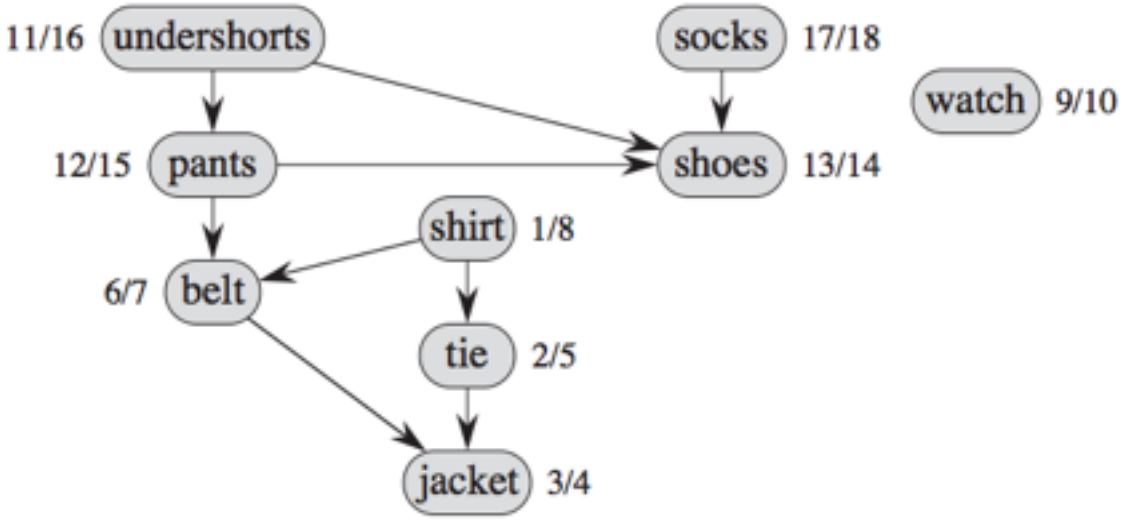
Sortare Topologica

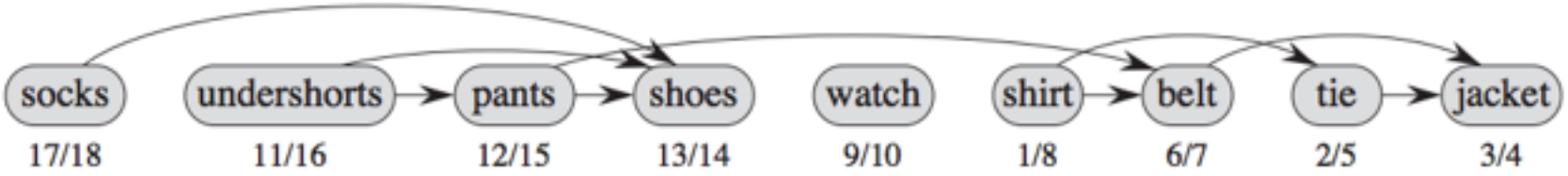
- ***Sortare topologica*** a unui DAG – G=(V,E): ordonare liniara a tuturor varfurilor, astfel incat daca G contine muchia *(u,v)* => *u* apare inainte lui *v* in ordonare
  - Daca graful contine cicluri, nu are sortare topo!
  - Precedenta intre evenimente

TOPOLOGICAL-SORT(G)

1 call DFS(G) to compute finishing times $v.f$ for each vertex v
2 as each vertex is finished, insert it onto the front of a linked list
3 **return** the linked list of vertices

Sortare Topologica – exemplu





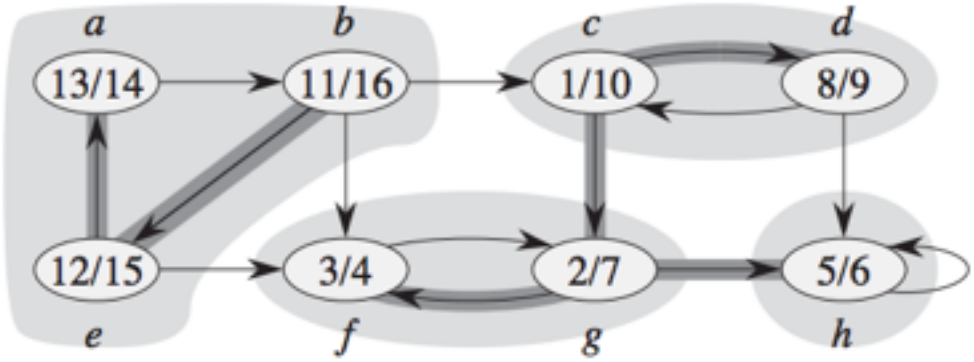
Eficienta?

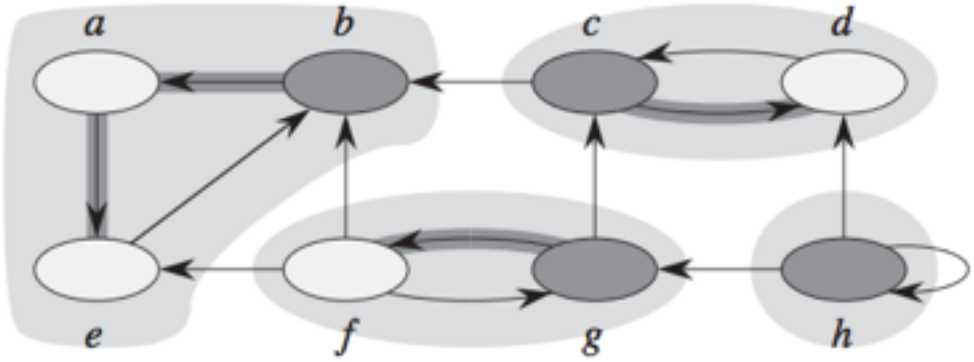
Determinarea componentelor puternic conexe - Grafuri orientate

- Componenta puternic conexa: multime maximala de varfuri astfel incat oricare 2 varfuri din multime sunt conectate ($u \rightsquigarrow v$ and $v \rightsquigarrow u$)

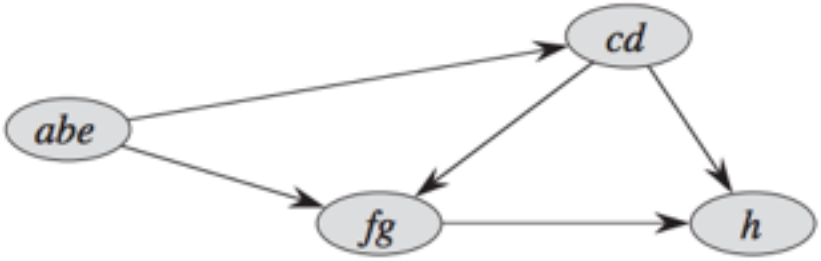
STRONGLY-CONNECTED-COMPONENTS(G)

1 call DFS(G) to compute finishing times $u.f$ for each vertex u
2 compute G^T
3 call DFS(G^T), but in the main loop of DFS, consider the vertices in order of decreasing $u.f$ (as computed in line 1)
4 output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component





Componente puternic conexe - exemplu



Bibliografie

- CLR, “Introduction to Algorithms”, chapter 22 - Elementary Graph Algorithms