Laborator 08: Parcurgerea grafurilor. Aplicații (2/2)

Responsabili:

- Darius-Florentin Neațu (2017-2021) [mailto:neatudarius@gmail.com]
- Radu Nichita (2021) [mailto:radunichita99@gmail.com]
- Ștefan Popa (2018-2020) [mailto:stefanpopa2209@gmail.com]

Autori:

- Darius-Florentin Neațu (2021) [mailto:neatudarius@gmail.com]
- Radu Nichita (2021) [mailto:radunichita99@gmail.com]

Obiective laborator

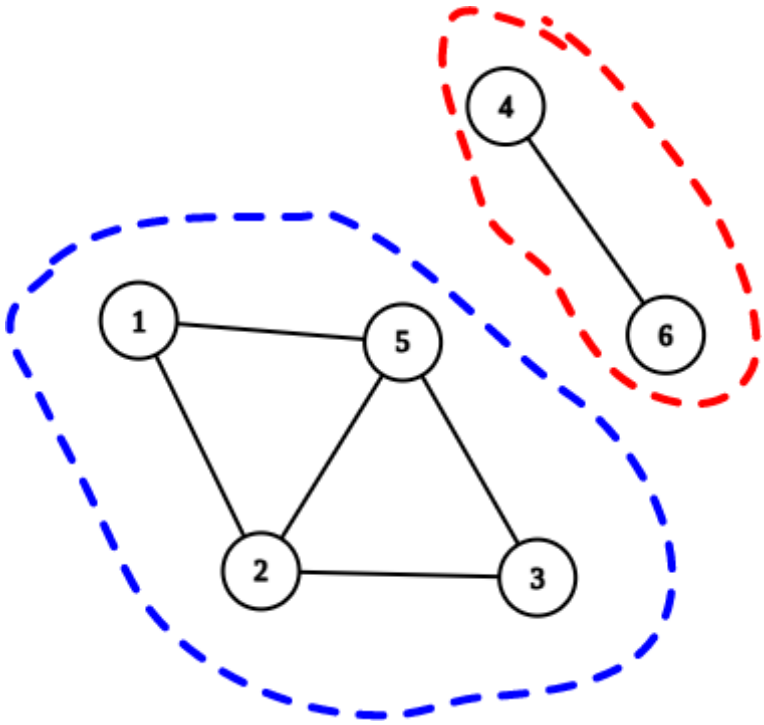
- Înțelegerea conceptelor de graf, reprezentare și parcugere
- Studierea unor aplicații pentru parcurgeri

Componente Conexe

> O **componentă conexă** (**CC**) / **Connected Component** (**CC**) într-un graf **neorientat** este o submulțime maximală de noduri, cu proprietatea că oricare ar fi două noduri x și y din aceasta, există drum de la x la y.

$n = 6\ m = 6$

$muchii: (1, 2); (1, 5); (2, 5); (2, 3); (3, 5); (4, 6);$



Sunt **2 CC**-uri în graful dat:

- {1, 2, 3, 5}
- {4, 6}

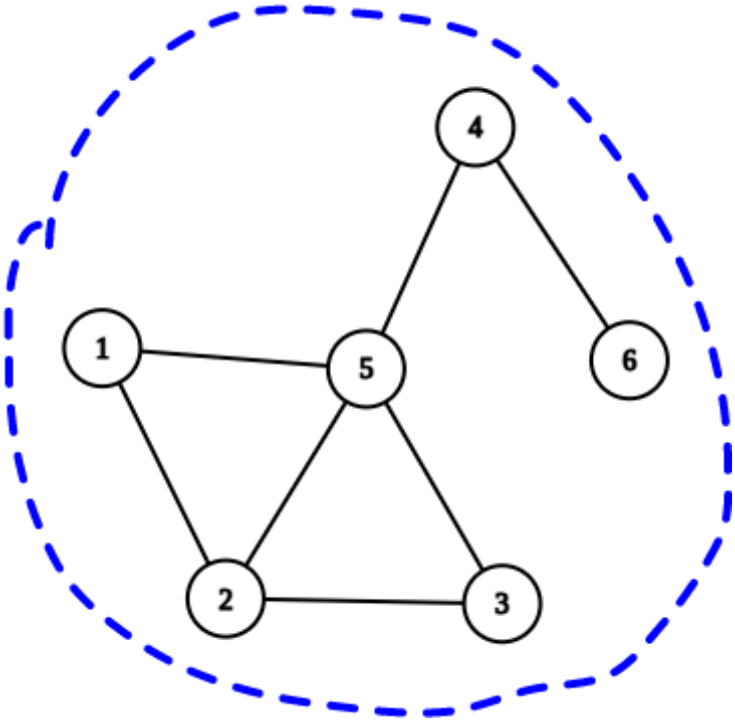
Explicație:

- Cele 2 sunt mulțimi maximale pentru care se respectă proprietatea de conexitate.
- 4 și 6 nu sunt accesibile din nodurile 1, 2, 3 și 5, prin urmare, acestea trebuie să facă parte din componente diferite.

Un graf **neorientat** este **conex** dacă conține **o singură** componentă conexă.

$n = 6\ m = 7$

$muchii: (1,2); (1,5); (2,5); (2,3); (3,5); (4,6); (5,4)$



Graful dat este conex - există **1 CC**: {1, 2, 3, 4, 5, 6}.

Explicație: Se poate ajunge de la oricare nod la oricare altul.

O componentă conexă reprezintă o partiție a nodurilor în submulțimi! ⇔ Fiecare nod face parte dintr-o singură componentă conexă!

Algoritmi

DFS

CC cu DFS:

- În algoritmul clasic de parcurgere DFS, de fiecare dată când se găsește un nod fără părinte și se apelează DFS_RECURSIVE, se descoperă o nouă componentă conexă.

- Toate nodurile vizitate în acel subarbore fac parte din aceeași componentă conexă.

Complexitate

$T = O(n + m)$

BFS

CC cu BFS:

- Se parcurge lista de noduri.
  - Pentru fiecare nod care nu are părinte, se pornește o nouă parcurgere BFS din nodul curent.
  - Toate nodurile vizitate într-o parcurgere BFS fac parte din aceeași componentă conexă.
- Observație: Se păstrează lista de părinți de la o parcurgere la alta.

Complexitate

$T = O(n + m)$

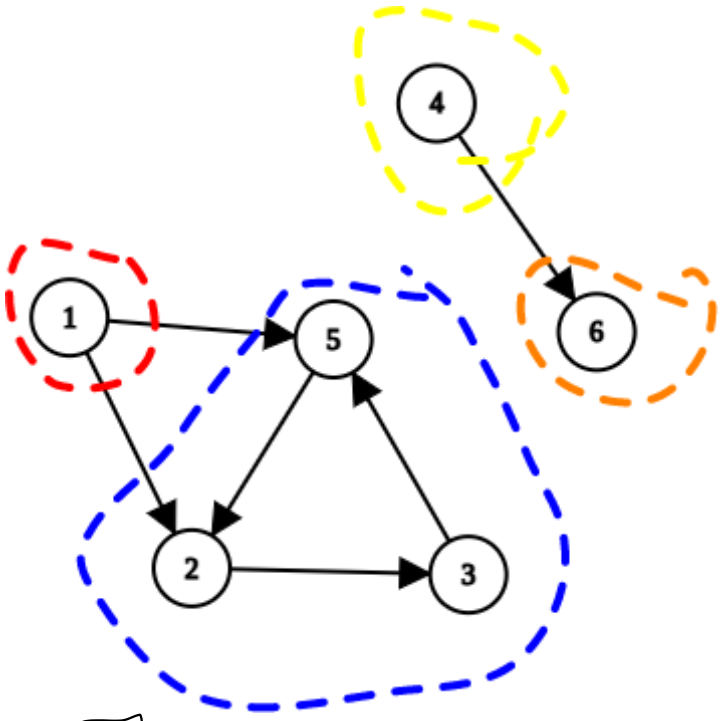
Deși ambele abordări au aceeași complexitate, recomandăm abordarea cu DFS pentru simplitate.

Componente Tare Conexe

> O **componentă tare conexă** (**CTC**) / **Strongly Connected Component** (**SCC**) într-un graf **orientat** este o submulțime maximală de noduri, cu proprietatea că oricare ar fi două noduri x și y din aceasta, există drum de la x la y.

$n = 6\ m = 6$

$arce: (1,2); (1,5); (5,2); (2,3); (3,5); (4,6)$

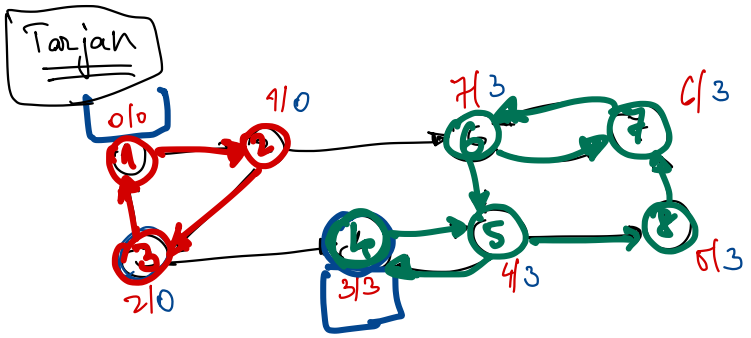


$\forall x, y,$ ∃ drum de la x la y și invers

Sunt **4** **SCC**-uri în graful dat:

- {1}

Tarjan



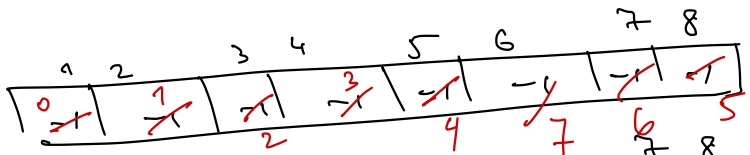
1: 2
2: 3 6
3: 1 4
4: 5
5: 4 8
6: 5 7
7: 6
8: 7

n=8 noduri
m=12 arce

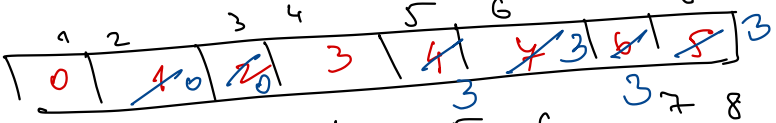
idx ← found

idx | lowlink

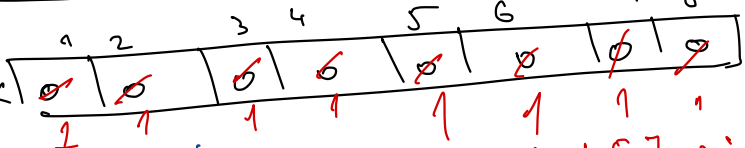
idx



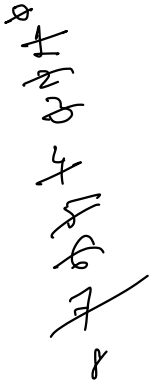
lowlink

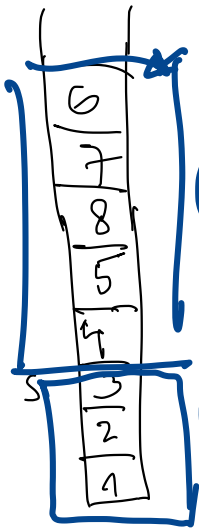


in-stack



indices





{4,5,8,7,6}

{1,2,3}

f(1) x=1 → y=2 lowlink[1] = min(0,0) = 0

f(2) x=2 → y=3 lowlink[2] = min(1,0) = 0

y=6 x

f(3) x=3 → y=1 → lowlink[3] = min(2,0) = 0

y=4 → lowlink[3] = min(0,3) = 0

idx[x] = lowlink[x] ⇓ x nod pt CTC

f(4) x=4 → y=5 lowlink[4] = min(3,3) = 3

f(5) x=5 → y=4 → lowlink[5] = min(4,3) = 3

y=8 lowlink[5] = min(3,3) = 3

f(8) x=8 → y=7 lowlink[8] = min(5,3) = 3

f(7) x=7 → y=6 lowlink[7] = min(3,6) = 3

$f(6)$: $x=6 \rightarrow y=5$ $\quad lowlink[6] = \min(3,7) = 3$

$y=7$ $\quad lowlink[6] = \min(3,6) = 3$

- {2, 3, 5}
- {4}
- {6}

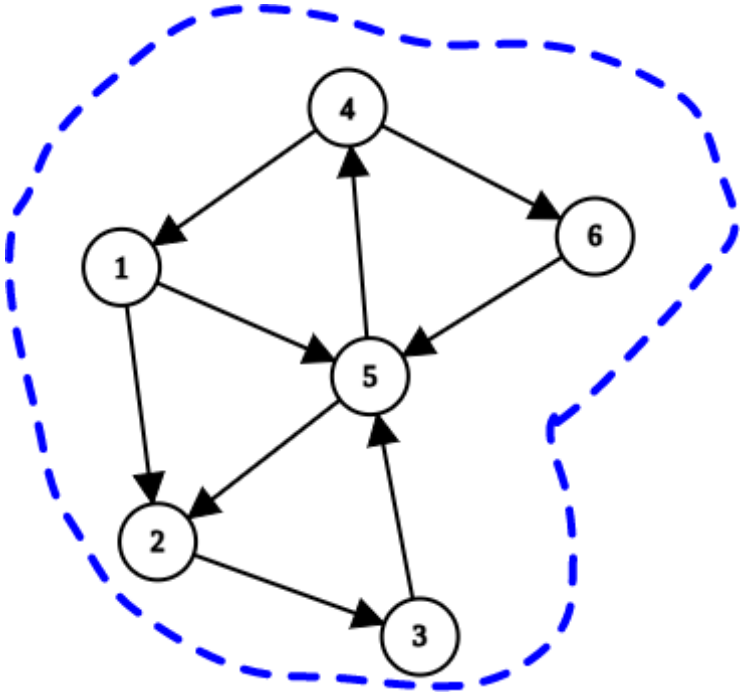
Explicație:

- În nodul 1 nu se poate ajunge, prin urmare acesta formează o componentă separată. Analog pentru 4.
- Similar, din nodul 6 nu se poate ajunge în alt nod, deci și acesta formează singur o componentă.
- Nodurile 2, 3 și 5 formează un ciclu, prin urmare se poate ajunge de la oricare la oricare.

Un graf **orientat** este **tare conex** dacă conține o **singură componentă** tare conexă.

$n = 6\ m = 6$

$arce: (1,2); (1,5); (5,2); (2,3); (3,5); (4,6); (4,1); (5,4); (6,5)$



Graful este tare conex - există **1 SCC**: {1, 2, 3, 4, 5, 6};

Explicație: Se poate vedea că pentru fiecare nod x se poate ajunge în oricare alt nod y.

-

O componentă tare conexă reprezintă o partiție a nodurilor în submulțimi! ⇔ Fiecare nod face parte dintr-o singură componentă tare conexă!

Algoritmi

TARJAN SCC

Algoritmul lui Tarjan pentru SCC [https://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm] foloseşte o singură parcurgere DFS în urma căreia rezultă o pădure de arbori DFS. Componentele tare conexe vor

fi subarbori în această pădure. Rădăcinile acestor subarbori se vor numi **rădăcinile componentelor tare conexe** (**SCC roots**).

Nodurile sunt puse pe o stivă, în ordinea vizitării. Când parcurgerea termină de vizitat un subarbore, se determină dacă rădăcina arborelui care s-a terminat de vizitat este și rădăcina unui SCC. Dacă un nod este rădăcina unei componente, atunci el şi toate de deasupra sa din stivă formează acea componentă tare conexă.

Pentru a determina dacă un nod este rădăcina unei componente tare conexe, se definesc:

```
// the timestamp when node was found (when started to visit its subtree)
found[node] = start[node];

// the minimum accessible timestamp that node can see/access
low_link[node] =  min { found[x] | x is node OR x in ancestors(node) OR x in descendants(node) };
```

Tarjan SCC: **node** is root for a SCC if **low_link[node]** == **found[node]**.

found[node] reprezintă timpul de **start** din DFS, definit în laboratorul anterior. În implementare reținem o variabilă **timestamp** care se incrementează de fiecare dată când se vizitează un nod. Noua valoare a lui **timestamp** este **found[node]** (momentul la care **node** a fost găsit).

low_link[node] reprezintă cel mai mic timp de descoperire al unui nod **x** la care se poate ajunge pornind din **node** și mergând pe arcele/muchii nevizitate (se poate coborî sau urca). Practic, nodul cu cel mai mic timp de descoperire care se poate atinge prin traversarea a 0 sau mai multe arce.

- observații prelimilare:
  - nodurile vizitate înaintea lui **node** au valoare **found** mai mică (deci și orice strămoș a lui **node** - mulțimea **ancestors(node)**)
  - nodurile descendente ale lui **node** au valoare **found** mai mare (mulțimea **descendants(node)**)
  - întrucât și **node** face parte din subarbore, inițializăm **low_link[node]** = **found[node]**. Valoarea finală va fi mai mică sau egală decât aceasta (conform definiției, vom căuta un minim).
- după ce **toate** nodurile accesibile din **node** au fost vizitate, se cunoaște ***valoarea finală** a lui **low_link[node]** și putem avea 2 cazuri:
- **low_link[node]** == **found[node]**
  - dacă valoarea finală a rămăs cea inițială, înseamnă că **NU** s-a urcat în arbore (altfel am fi întâlnit valori mai mici decât cea inițială)
  - prin urmare **node** este rădăcina unui SCC (primul nod întâlnit din acest SCC)
  - nodurile din vârful stivei de deasupra lui **node** formează SCC-ul găsit
- **low_link[node]** < **found[node]**
  - dacă valoarea finală pentru **low_link[node]** este mai mică decât cea inițială, înseamnă că s-a urcat în arbore
  - în acest caz există cel puțin o muchie **(y, x)** unde **x** este strămoș și **y** este descendent pentru **node**, prin care **y** (si implicit și **node**) își actualizează minimul cu valoarea din **x**
  - drumul **x** - ... - **node** -... **y** - ... - **x** este atunci un ciclu care face parte dintr-un SCC; **node** este un nod oarecare dintr-un astfel de ciclu (“la mijloc”, întrucât mai sus de el există cel puțin un nod mai aproape de “începutul ciclului”, adică nodul **x**)
  - prin urmare, suntem siguri că **node** nu este rădăcina unui SCC

Algoritm

| TARJAN_SCC

```
// Tarjan_SCC
// * visit all nodes with DFS
//      * compute found[node] and low_link[node]
```

```
//      * extract SCCs
//
// nodes     = list of all nodes from G
// adj[node] = the adjacency list of node
//             example: adj[node] = {..., neigh, ...} => edge (node, neigh)
TARJAN_SCC(G = (nodes, adj)) {
    // STEP 1: initialize results
    // parent[node] = parent of node in the DFS traversal
    //
    // the timestamp when node was found (when started to visit its subtree)
    // Note: The global timestamp is incremented everytime a node is found.
    //
    // the minimum accessible timestamp that node can see/access
    // low_link[node] =  min { found[x] | x is node OR x in ancestors(node) OR x in descendants(node) };
    //
    foreach (node in nodes) {
        parent[node] = null; // parent not yet found
        found[node] = +oo; // node not yet found
        low_link[node] = +oo; // value not yet computed
    }
    nodes_stack = {}; // visiting order stack

    // STEP 2: visit all nodes
    timestamp = 0; // global timestamp
    foreach (node in nodes) {
        if (parent[node] == null) { // node not visited
            parent[node] = node; // convention: the parent of the root is actually the root

            // STEP 3: start a new DFS traversal this subtree
            DFS(node, adj, parent, timestamp, found, low_link, nodes_stack);
        }
    }
}

DFS(node, adj, parent, ref timestamp, found, low_link, nodes_stack) {
    // STEP 1: a new node is visited - increment the timestamp
    found[node] = ++timestamp; // the timestamp when node was found
    low_link[node] = found[node]; // node only knows its timestamp
    nodes_stack.push(node); // add node to the visiting stack

    // STEP 2: visit each neighbour
    foreach (neigh in adj[node]) {
        // STEP 3: check if neigh is already visited
        if (parent[neigh] != null) {
            // STEP 3.1: update low_link[node] with information gained through neigh
            // note: neigh is in the same SCC with node only if it's in the visiting stack;
            // otherwise, neigh is from other SCC, so it should be ignored
            if (neigh in nodes_stack) {
                low_link[node] = min(low_link[node], found[neigh]);
            }

            continue;
        }

        // STEP 4: save parent
        parent[neigh] = node;

        // STEP 5: recursively visit the child subtree
        DFS(neigh, adj, parent, timestamp, found, low_link, nodes_stack);

        // STEP 6: update low_link[node] with information gained through neigh
        low_link[node] = min(low_link[node], low_link[neigh]);
    }

    // STEP 7: node is root in a SCC if low_link[node] == found[node]
    // (there is no edge from a descendant to an ancestor)
    if (low_link[node] == found[node]) {
        // STEP 7.1: pop all elements above node from stack => extract the SCC where node is root
        new_scc = {};
        do {
            x = nodes_stack.pop();
            new_scc.push(x);
        } while (x != node); // stop when node was popped from the stack

        // STEP 7.2: save / print the new SCC
```

```
        print(new_scc);
    }
}
```

Observații:

- La pasul **3.1** se încearcă actualizarea lui **low_link[node]** cu informația din **neigh** doar dacă **neigh** este în stivă.
    - Nodul **neigh** are deja părinte, deci poate fi în unul din următoare 2 cazuri:
        - **neigh** este în curs de vizitare (deci este în stivă) ⇒ **neigh** este strămoș a lui **node**
            - Reactualizăm **low_link[node]** cu valoarea din **neigh**.
        - **neigh** este deja vizitat (deci a fost scos din stivă) ⇒ **neigh** face parte din alt subarbore, terminat anterior.
            - Prin urmare, anterior s-a stabilit că **neigh** face parte dintr-un alt SCC și trebuie ignorat (întrucât sigur are valoare **found** mai mică decât a lui **node** și ar reactualiza **low_link[node]** în mod eronat.
    - Se face această actualizare doar dacă **neigh** este strămoș al lui

Complexitate

- **complexitate temporală** : $T = O(n + m)$
- **complexitate spațială** : $S = O(n)$
    - recursivitate + câteva structuri de date de lungime $O(n)$

Kosaraju

Există și alt algoritm pentru determinarea componentelor tare conexe. Algoritmul lui Kosaraju se bazează pe compactarea ciclurilor. Deoarece are aceeași complexitate ca și Tarjan, nu îl vom studia la laborator la PA. Am ales algoritmul lui Tarjan întrucât îl putem modifica ușor pentru a produce și alte rezultate.

Puteți consulta următoarele materiale dacă doriți să aflați mai multe:

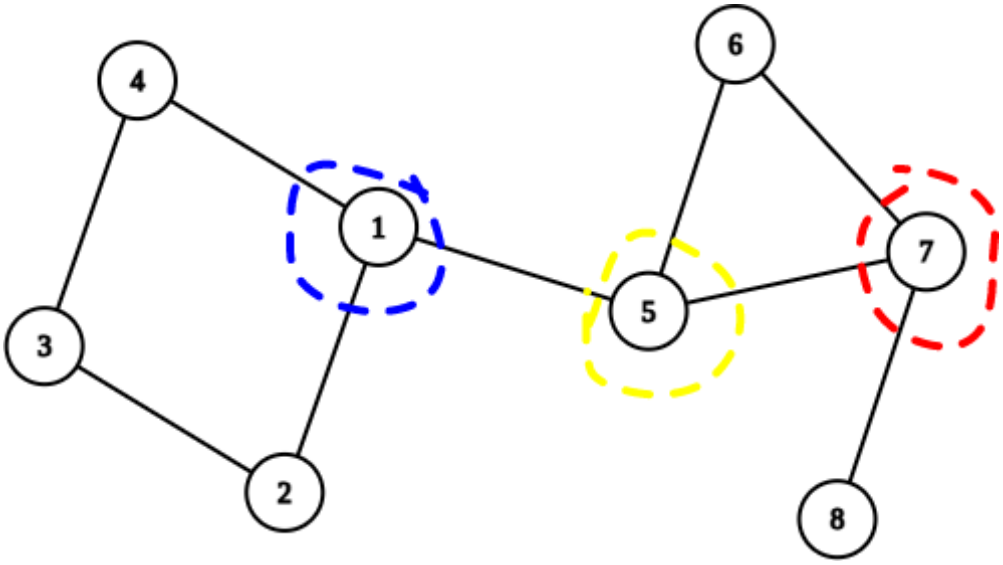
- https://www.youtube.com/watch?v=RpgcYiky7uw [https://www.youtube.com/watch?v=RpgcYiky7uw]
- https://iq.opengenus.org/kosarajus-algorithm-for-strongly-connected-components/ [https://iq.opengenus.org/kosarajus-algorithm-for-strongly-connected-components/]

Puncte de articulație

> **Punct de articulație** / **nod critic** / **Cut Vertex** (**CV**) este un nod într-un graf **neorientat** a cărui eliminare duce la creșterea numărului de componente conexe (CC) - se elimină nodul împreună cu muchiile incidente.

$n = 8\ m = 6$

$muchii: (1,2); (2,3); (3,4); (4,1); (1,5); (5,6); (6,7); (7,5); (7,8)$



Sunt **3 CV**-uri în graful dat: 1, 5 și 7.

Explicație:

- Dacă ștergem nodul 1, graful se sparge în 2 CC-uri: {2, 3, 4}, {5, 6, 7, 8}.
- Dacă ștergem nodul 5, graful se sparge în 2 CC-uri: {1, 2, 3, 4}, {6, 7, 8}.
- Dacă ștergem nodul 7, graful se sparge în 2 CC-uri: {1, 2, 3, 4, 5, 6}, {8}.
- Dacă ștergem oricare alt nod, ,graful rămâne conex.

TARJAN CV

Putem modifica ușor algoritmul TARJAN SCC astfel încât să obținem Algoritmul lui Tarjan pentru CV [https://en.wikipedia.org/wiki/Biconnected_component].

În mod analog, pentru a determina dacă un nod este CV, se definesc și folosesc **found** și **low_link**.

> **TARJAN CV**: **node** is **CV** if
> **i)** node is NOT root and **low_link[neigh]** >= **found[node]** for at least one **neigh** in **adj[node]**
> OR
> **ii)** node is root and children(node) > 1

Dacă **node** este rădăcină într-un subarbore, acesta are valoarea **found** mai mică decât a oricărui nod. Prin urmare, condiția **low_link[neigh]** >= **found[node]** ar fi adevărată mereu și nu ne-ar putea furniza o informație utilă. De aceea, cazul **i)** nu este aplicabil pentru rădăcină. Putem trata foarte simplu cazul pentru rădăcină folosind **ii)**: dacă **node** este rădăcină a unui subarborele și are cel puțin 2 copii, atunci, prin eliminarea lui **node**, arborele acestuia se sparge într-un număr de subarbori egal cu numărul de copii.

found[node] are aceeași semnificație ca la SCC.

low_link[node] are aceeași semnificație ca la SCC.

- observații/diferențe:
  - Nu este nevoie de folosirea stivei de vizitare.
    - La SCC aveam nevoie de stiva de noduri pentru a nu folosi o muchie **node** → **neigh** (arc) care unea 2 SCC-uri.
    - Într-un graf neorientat nu putem avea o muchie care să unească 2 subarbori, deoarece în momentul în care un capăt este vizitat, adaugă și celălalt capăt în același subarbore.

- **neigh** este copil al lui **node** în parcurgerea DFS ⇒ **found[neigh]** > **found[node]**
- după ce un copil **neigh** este vizitat, se cunoaște ***valoarea finală** a lui **low_link[neigh]** și putem avea 2 cazuri:
  - **low_link[neigh]** < **found[node]**
    - inițial **low_link[neigh]** = **found[neigh]**, deci **low_link[neigh]** > **found[node]**
    - în acest caz există cel puțin o muchie **(y, x)** unde **x** este strămoș și **y** este descendent pentru **node** prin care **y** (și implicit și **node**) își actualizează minimul cu valoarea din **x**
    - drumul **x** – ... – **node** – **neigh** – ... **y** – **x** este atunci un ciclu
      - dacă **node** este eliminat din graf, toate nodurile din subarborele lui **neigh** vor rămâne conectate de restul grafului prin muchia **(y, x)**
      - deci nu putem trage vreo concluzie doar analizând vecinul curent **neigh**, trecem la următorul
  - **low_link[neigh]** >= **found[node]**
    - în acest caz nu există ciclul **x** – ... – **node** – **neigh** – ... **y** – **x** de la cazul anterior
    - eliminarea lui **node** ar duce la separarea subarborelui lui **neigh** de restul grafului
    - prin urmare, numărul de componente conexe crește cu cel puțin 1, deci **node** este sigur un **CV** (concluzie corectă chiar dacă ne-am uitat la un singur vecin **neigh**)

Punem la dispoziție un diff de pseudocod: TARJAN_SCC vs TARJAN_CV [https://pastebin.com/raw/8th3Pnjg]. Se observă că este același algoritm, singurele diferențe relevante sunt:

- STEP **3.1**: condiția după care se reactulizează **low_link[node]** în funcție de **neigh** atunci când cel din urmă este deja vizitat
- STEP **7**: condiția prin care se determină dacă **node** este o rădăcină de SCC / CV.

Complexitate

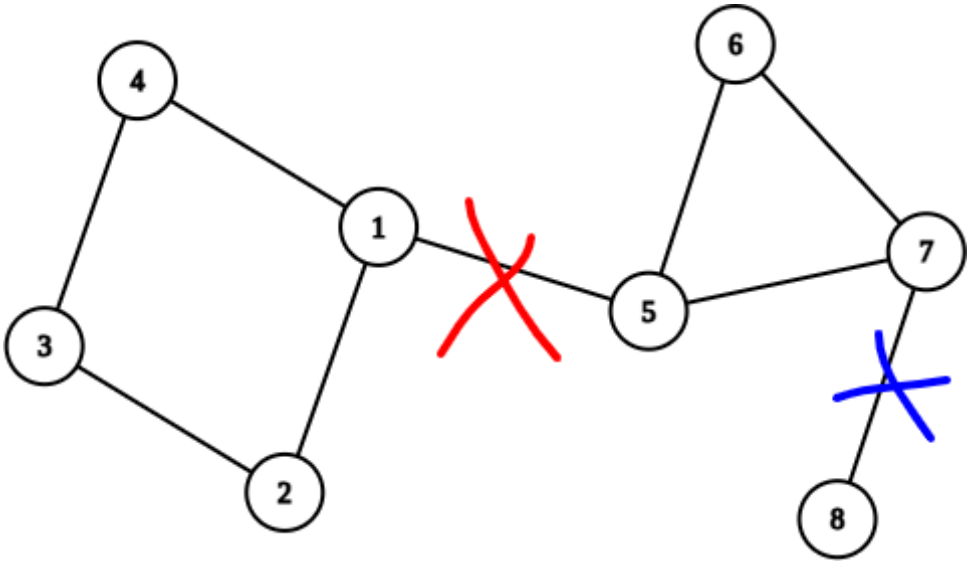
- **complexitate temporală** : $T = O(n + m)$
- **complexitate spațială** : $S = O(n)$
  - recursivitate + câteva tablouri auxiliare de lungime n

Punți / muchii critice

> **Punte** / **muchie critică** / **Critical Edge** (**CE**) este o muchie într-un graf **neorientat** a cărei eliminare duce la creșterea numărului de componente conexe (CC) – se elimină muchia, fără a se sterge capetele (nodurile) acesteia.

$n = 8\ m = 6$

$muchii : (1, 2); (2, 3); (3, 4); (4, 1); (1, 5); (5, 6); (6, 7); (7, 5); (7, 8)$



Sunt **2 CE**-uri în graful dat: (1, 5) și (7,8)

Explicație:

- Dacă ștergem muchia (1, 5), graful se sparte în 2 CC-uri: {1, 2, 3, 4}, {5, 6, 7, 8}.
- Dacă ștergem muchia (7, 8), graful se sparte în 2 CC-uri: {1, 2, 3, 4, 5, 6, 7}, {8}.
- Dacă ștergem oricare altă muchie, graful rămâne conex.

TARJAN CE

Se modifică algoritmul de CV. Se folosesc aceleași definiții și semnificații pentru **found** și **low_link**.

> **TARJAN CE**: **(node, neigh)** is a **CE** if **low_link[neigh]** > **found[node]** where **neigh** in **adj[node]**.

Există 2 tipuri de muchii în parcugerea DFS într-un graf neorientat:

- **(node, neigh)**: muchiile din arbore (numită și muchie de arbore)
- **(y, x)**: muchie de la un nod **y** la un strămoș **x** (numită și muchie înapoi)

Tipul al 2-lea de muchie închide un ciclu, deci clar nu reprezintă un CE. Prin urmare trebuie să căutăm toate CE-urile printre muchiile **(node, neigh)** din arbore.

Când se termină de vizitat subarborele lui **neigh** și cunoaștem valoarea finală a lui **low_link[neigh]** putem avea:

- **low_link[neigh]** ⇐ **found[node]**:
  - analog explicațiilor de la CV, din subarborele lui **neigh** se poate urca până la un nod **x** (x este **node** SAU un strămoș al lui **node**) ⇒ muchia **(node, neigh)** face parte dintr-un ciclu
  - prin urmare, dacă se taie aceasta, toate nodurilor de pe ciclu rămân conectate, deci nu este CE
- **low_link[neigh]** > **found[node]**:
  - înseamnă că nu există acel ciclu de la pasul anterior (nu s-a putut urca în arbore mai sus de **node**)
  - prin urmare, **(node, neigh)** este **CE**

Complexitate

- **complexitate temporală** : $T = O(n + m)$
- **complexitate spațială** : $S = O(n)$

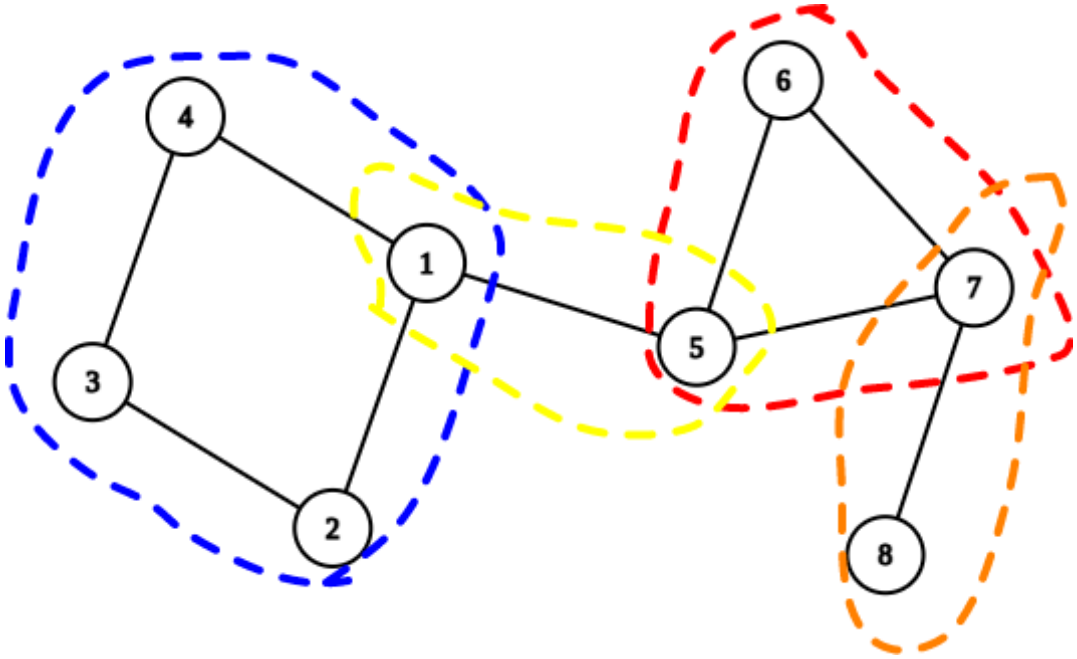
- recursivitate + câteva structuri de date de lungime $O(n)$

Componente Biconexe

O **componentă biconexă** / **BiConnected Component** (**BCC**) într-un graf **neorientat** este o submulțime maximală de noduri cu proprietarea că nu conține puncte de articulație - oricare nod s-ar elimina, nodurile rămase sunt încă conectate.

$n = 8\ m = 9$

$muchii$: $(1, 2); (2, 3); (3, 4); (4, 1); (1, 5); (5, 6); (6, 7); (7, 5); (7, 8)$



Sunt **4 BCC**-uri în graful dat: * {1, 2, 3, 4} * {1, 5} * {5, 6, 7} * {7, 8}

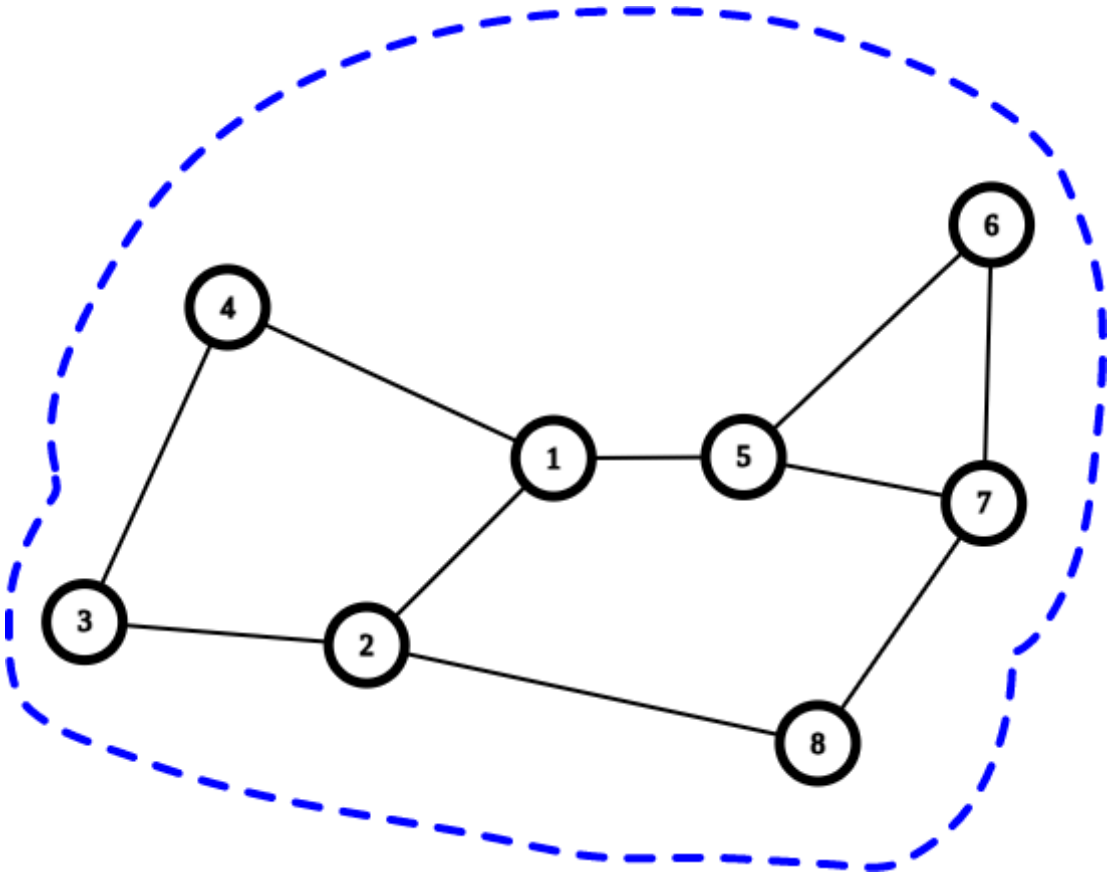
Explicație:

- Dacă ștergem muchia (1, 5), graful se sparge în 2 CC-uri: {1, 2, 3, 4}, {5, 6, 7, 8}.
- Dacă ștergem muchia (7, 8), graful se sparge în 2 CC-uri: {1, 2, 3, 4, 5, 6, 7}, {8}.
- Dacă ștergem oricare altă muchie, graful rămâne conex.

Un graf **neorientat** este **biconex** dacă nu conține puncte de articulație - conține o singură componentă biconexă.

$n = 8\ m = 10$

$muchii$: $(1, 2); (2, 3); (3, 4); (4, 1); (1, 5); (5, 6); (6, 7); (7, 5); (7, 8); (8, 2)$



Sunt **1 BCC**-uri în graful dat: {1, 2, 3, 4, 5, 6, 7, 8}

Explicație:

- Nu există noduri/puncte critice în graf (se poate șterge orice nod și graful rămâne conex).

Împărțirea în componente biconexe a unui graf neorientat reprezintă **o partiție disjunctă a muchiilor grafului** (împreună cu vârfurile adiacente muchiilor). Acest lucru implică faptul că unele vârfuri pot face parte din mai multe componente biconexe diferite (vezi BCC - exemplu 01) - mai exact, punctele de articulație vor face parte din mai multe componente.

TARJAN BCC

Se modifică algoritmul de CV. Se folosesc aceleași definiții și semnificații pentru **found** și **low_link**.

- Se folosește o stivă **edges_stack** în care se adaugă toate muchiile **(node, neigh)** atunci când se înaintează în recursivitate.
- Atunci când se termină de vizitat un copil **neigh**, dacă se îndeplinește condiția de **CV** (**low_link[neigh]** >= **found[node]**), înseamnă că prin eliminarea lui **node** tot subarborele **node - neigh - ...** rămâne deconectat. Prin urmare, toate muchiile din stivă de deasupra muchiei **(node, neigh)** (inclusiv) formează o componentă biconexă (mulțimea de noduri formată din capetele acestor muchii).
- Se termină de vizitat copilul curent și se trece la următorul. De fiecare dată când se găsește un copil **neigh** cu **low_link[neigh]** >= **found[node]** se formează o nouă **BCC**.

Complexitate

- **complexitate temporală** : $T = O(n + m)$
- **complexitate spațială** : $S = O(n + m)$

- recursivitate + câteva structuri de date de lungime $O(n)$ / $O(m)$
    - ATENȚIE! În plus, față de CE/CV , se stochează o stivă de muchii.

Importanţă - aplicaţii practice

- SCC: Data Mining, Compilatoare, problema 2-SAT.
- BCC: cele mai importante aplicații se găsesc în rețelele de calculatoare, deoarece un BCC asigură redundanţă (există cel puțin 2 căi de a conecta o entitate la celelalte).

TLDR

- Se poate folosi/modifica algoritmul lui Tarjan pentru a determina **SCC**, **CV** / **CE** / **BCC**.
- Deoarece algoritmul se folosește de o parcurgere DFS, complexitatea este liniară în toate cazurile.

Exercitii

Scheletul de laborator se găsește pe pagina pa-lab::skel/lab08 [https://github.com/acs-pa/pa-lab/tree/main/skel/lab08].

Înainte de a rezolva exercițiile, asigurați-vă că ați citit și înțeles toate precizările din secțiunea Precizari laboratoare 07-12 [https://ocw.cs.pub.ro/courses/pa/skel_graph].

Prin citirea acestor precizări vă asigurați că:

- știți **convențiile** folosite
- evitați **buguri**
- evitați **depunctări** la lab/teme/test

Se dă un graf **orientat** cu **n** noduri și **m** arce. Să se găsească **componentele tare-conexe** folosind algoritmul lui **Tarjan**. Secțiunea de teorie conține exemple grafice explicate.

Restricții și precizări:

- $n <= 10^5$
- $m <= 2 * 10^5$
- timp de execuție
    - C++: 1s
    - Java: 4s

Se dă un graf **neorientat conex** cu **n** noduri și **m** muchii. Se cere să se găsească toate **punctele critice** folosind algoritmul lui **Tarjan**. Secțiunea de teorie conține exemple grafice explicate.

Restricții și precizări:

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

- Java: 4s

Se dă un graf **neorientat conex** cu **n** noduri și **m** muchii. Se cere să se găsească toate **muchiile critice** folosind algoritmul lui **Tarjan**. Secțiunea de teorie conține exemple grafice explicate.

Restricții și precizări:

- $n <= 10^5$
- $m <= 2 * 10^5$
- timp de execuție
  - C++: 1s
  - Java: 4s

Se dă un graf **neorientat conex** cu **n** noduri și **m** muchii. Se cere să se găsească toate **componentele biconexe** folosind algoritmul lui **Tarjan**. Secțiunea de teorie conține exemple grafice explicate.

Restricții și precizări:

- $n <= 10^5$
- $m <= 2 * 10^5$
- timp de execuție
  - C++: 1s
  - Java: 4s

Extra

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

Referințe

[0] Chapter **Elementary Graph Algorithms**, “Introduction to Algorithms”, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein

[1] https://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm [https://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm]

[2] https://en.wikipedia.org/wiki/Biconnected_component [https://en.wikipedia.org/wiki/Biconnected_component]

[3] “Depth-first search and linear graph algorithms”, R.Tarjan

[4] https://en.wikipedia.org/wiki/Kosaraju%27s_algorithm [https://en.wikipedia.org/wiki/Kosaraju%27s_algorithm]

pa/laboratoare/laborator-08.txt · Last modified: 2021/05/09 22:38 by gabriel.bercaru